Võistlusprogrammeerimine

ehk kuidas kirjutada kiiret koodi

Targo Tennisberg, Katrin Gabrel

0 Sissejuhatus ... 14
0.1 Mis on võistlusprogrammeerimine? ... 14
0.2 Mida siit õpikust leida võib? ... 15
0.3 Kontrollülesannete lahendamine ... 17
0.4 Kasutatud ülesanded ja pildid ... 17
0.5 Autoritest ... 17
0.6 Tänuavaldused ... 18
0.7 Veaparandused ja soovitused ... 18
1 Programmide sisemaailm ... 19
1.1 Programmi elutsükkel ... 20
1.1.1 Lähtekood ... 20
1.1.2 Lähtekoodi transleerimine ... 20
1.1.3 Teegid ... 21
1.1.4 Linkimine ... 22
1.1.5 Laadimine ja täitmine ... 22
1.2 Andmete hoidmine ja töötlemine arvutis ... 22
1.2.1 Protsessori tööpõhimõte ... 22
1.2.2 Protsessori jõudlus ... 23
1.2.3 Käsukonveier ... 24
1.2.4 Hargnemise ennustamine ... 25
1.2.5 Andmete liikumine ... 27
1.2.6 Mälu ... 27
1.3 Andmetüübid ... 29
1.3.1 Täisarvud ... 29
1.3.2 Ujukomaarvud ... 30
1.3.3 Püsikomaarvud ... 35
1.3.4 Märgid ... 35
1.3.5 Tõeväärtused ... 35
1.3.6 Viidad ... 36
1.3.7 Struktuursed tüübid ehk liittüübid ... 36
1.3.8 Kirjed ... 37
1.3.9 Stringid ... 37
1.3.10 Massiivid ... 37
1.4 Operatsioonid stringidega ... 38
1.4.1 Märk kohal i ... 38
1.4.2 Stringide võrdlemine ... 38
1.4.3 Stringide ühendamine ehk liitmine ... 39
1.5 Sisend-väljund ... 41
1.5.1 Standardvood ... 41

1.5.2 Failidest lugemine ja kirjutamine ... 42
1.5.3 Sisendi töötlus ... 43
1.5.4 Väljund ... 44
1.5.5 Jooksvalt töötamine ... 45
1.6 Programmeerimiskeelte võrdlus ... 46
1.6.1 C++ ... 46
1.6.2 Java ... 47
1.6.3 Python ... 48
1.7 Testimine ... 50
1.7.1 Piirjuhud ... 50
1.7.2 Õigsuse kontroll ... 51
1.7.3 Algoritmi jõudluse kontroll ... 52
1.8 Silumine ... 53
1.8.1 Arenduskeskkonnad ... 53
1.8.2 Aja mõõtmine ... 54
1.9 Kontrollülesanded ... 55
1.9.1 Järelmaks ... 56
1.9.2 Kell ... 56
1.9.3 Tigu ... 57
1.9.4 3D printer ... 57
1.9.5 Mõttemeister ... 58
1.9.6 Male ... 58
1.9.7 Riigihanked ... 59
1.9.8 Bender ... 60
1.9.9 Interpretaator ... 60
1.9.10 Pangatellerid ... 61
1.10 Viited lisamaterjalidele ... 62
2 Läbivaatus- ja otsingualgoritmid ... 63
2.1 Alamprogrammid ... 63
2.1.1 Pinumälu ... 63
2.1.2 Funktsiooni kohalikud andmed ... 64
2.1.3 Pinu ületäitumine ... 64
2.1.4 Pinu kasutamine protsessoris ... 65
2.1.5 Kuhimälu ... 66
2.1.6 Funktsiooni parameetrid ... 67
2.1.7 Objektide edastamine parameetritena ... 68
2.2 Rekursioon ... 70
2.2.1 Hargnemiseta rekursioon ... 70
2.2.2 Rekursioonivalem ... 71
2.2.3 Hargnemistega rekursioon ... 72
2.2.4 Sabarekursioon ja väljakutsete optimeerimine ... 74
2.3 Variantide läbivaatamine ... 76
2.3.1 Tagurdusmeetod ... 76
2.3.2 Iteratsioonimeetod ... 76
2.3.3 Tagurduse ja iteratsiooni võrdlus ... 77
2.3.4 Variantide läbivaatamine tagurdusmeetodiga ... 77
2.3.5 Variantide läbivaatamine iteratsioonimeetodiga ... 78
2.3.6 Variantide läbivaatus programmeerimisvõistlustel ... 78

2.4 Läbivaatuse optimeerimine ... 79
2.4.1 Lihtne tagurdusmeetod ... 79
2.4.2 Andmete optimeerimine ... 80
2.4.3 Ettearvutamine ... 81
2.4.4 Otsingupuu ahendamine ... 82
2.4.5 Sisemiste tsüklite optimeerimine ... 83
2.4.6 Andmestruktuuride optimeerimine ... 84
2.4.7 Rekursiooni eemaldamine ... 86
2.5 Kahendotsing ... 87
2.5.1 Vastuse kahendotsing ... 87
2.5.2 Kahendotsing andmetest ... 89
2.5.3 Topeltkahendotsing ... 90
2.5.4 Otsing kahemõõtmelisest tabelist sadulameetodil ... 91
2.6 Jaga ja valitse ... 95
2.7 Sissejuhatus kombinatoorikasse ... 96
2.7.1 Permutatsioonid ... 96
2.7.2 Permutatsioonide konstrueerimine ... 97
2.7.3 Kombinatsioonid ... 99
2.8 Kontrollülesanded ... 101
2.8.1 Autoturg ... 101
2.8.2 Kaupmees ja maksukoguja ... 102
2.8.3 Lippude vasturünnak ... 102
2.8.4 Akulaadija ... 103
2.8.5 Pildi pakkimine ... 103
2.8.6 Stern-Brocot' puu ... 104
2.8.7 Viinamarjad ... 105
2.8.8 Erinevad summad ... 106
2.8.9 Kodumasinate näidikud ... 107
2.8.10 Graafi värvimine ... 108
2.9 Viited lisamaterjalidele ... 108
3 Algoritmi keerukus ja põhilised andmestruktuurid ... 109
3.1 Algoritmi keerukus ... 109
3.1.1 Keskmine ja halvim keerukus ... 110
3.1.2 Asümptootiline hinnang ... 110
3.1.3 Kasvuseoste omadused ... 112
3.2 Algoritmi keerukuse hindamine ... 112
3.2.1 Hargnemiste keerukuse hindamine ... 112
3.2.2 Tsüklite keerukuse hindamine ... 113
3.2.3 Mitmekordsete tsüklite keerukus ... 113
3.2.4 Rekursiooni keerukuse hindamine ... 114
3.3 Tavalised keerukusklassid ... 114
3.3.1 Keerukusklass ja ajalimiit ... 116
3.4 Andmestruktuurid ... 116
3.4.1 Andmestruktuuride klassifitseerimine ... 117
3.5 Massiiv ... 117
3.5.1 Massiivid Pythonis ... 117
3.5.2 Massiivi võimalused ja piirangud ... 118
3.5.3 Mitmemõõtmelised massiivid ... 118

3.5.4 Dünaamilised massiivid ... 119
3.6 Ahel ... 119
3.7 Pinu ... 121
3.8 Järjekord ... 122
3.8.1 Kaheotsaline järjekord ... 123
3.8.2 Eelistusjärjekord ... 123
3.9 Pinu ja järjekorra kasutamine ... 123
3.10 Kahendpuu ... 127
3.10.1 Kahendpuu esitus ... 128
3.10.2 Kahendotsingu puu ... 128
3.11 Kujutis ... 128
3.11.1 Kujutis programmeerimiskeeltes ... 129
3.12 Praktiline ülesanne ... 130
3.13 Kuhi ... 131
3.14 Sortimine ... 132
3.14.1 Mullimeetod ... 133
3.14.2 Valikmeetod ... 134
3.14.3 Pistemeetod ... 135
3.14.4 Põimemeetod ... 135
3.14.5 Kiirmeetod ... 137
3.14.6 Võrdlustel põhineva sortimise keerukuse alampiir ... 138
3.15 Sortimise erimeetodid ... 139
3.15.1 Loendamismeetod ... 139
3.15.2 Positsioonimeetod ... 140
3.15.3 Kimbumeetod ... 140
3.16 Mitme kriteeriumi järgi sortimine ... 140
3.17 Programmeerimiskeelte standardteegid ... 140
3.17.1 C++ ... 141
3.17.2 Java ... 141
3.17.3 Python ... 141
3.18 Kontrollülesanded ... 143
3.18.1 Vabrikud ... 143
3.18.2 Jalgpalliturniir ... 144
3.18.3 Erdöse arvud ... 145
3.18.4 Programmeerimisvõistlus ... 146
3.18.5 Pannkoogid ... 146
3.18.6 Kaartide segamine ... 147
3.18.7 Kass klaviatuuril ... 147
3.18.8 Pinugrammid ... 148
3.18.9 Parv ... 149
3.18.10 Ahelmurrud ... 150
3.19 Viited lisamaterjalidele ... 150
4 Arvuteooria ... 151
4.1 Jaguvus ja jääk ... 151
4.1.1 Jaguvus ... 151
4.1.2 Jääk ... 152
4.1.3 Jäägi leidmine programmeerimiskeeltes ... 152
4.1.4 Ülesanne: loosiümbrikud ... 153

4.1.5 Arvutamine moodulitega ... 154
4.1.6 Ülesanne: anagrammid 2 ... 156
4.2 SÜT ja VÜK ... 158
4.2.1 Eukleidese algoritm suurima ühisteguri leidmiseks ... 159
4.2.2 Vähima ühiskordse leidmine ... 159
4.2.3 Ülesanne: hammasrattad ... 160
4.2.4 Diofantilised võrrandid ... 161
4.3 Algarvud ... 162
4.3.1 Eratosthenese sõel ... 162
4.3.2 Algarvulisuse kontroll ... 164
4.3.3 Ülesanne: algarvu-Scrabble ... 166
4.3.4 Arvu algtegurid ... 169
4.4 Positsioonilised arvusüsteemid ... 170
4.4.1 Süsteemide vahel teisendamine ... 170
4.4.2 Ülesanne – positsioonilised arvusüsteemid ... 172
4.4.3 Arvutamine kahendsüsteemis ... 177
4.5 Suurte arvudega arvutamine ... 177
4.5.1 Suurte arvude esitus ... 177
4.5.2 Suurte arvude liitmine ja lahutamine ... 178
4.5.3 Suurte arvude korrutamine ja astendamine ... 180
4.5.4 Efektiivne astendamine ... 181
4.5.5 Suurte arvude jagamine ... 182
4.5.6 Ülesanne – anagrammid 3 ... 182
4.5.7 Suured arvud erinevates programmeerimiskeeltes ... 183
4.5.8 Logaritm ja eksponent suurte arvudega arvutamisel ... 184
4.6 Kontrollülesanded ... 186
4.6.1 Suur arv ... 186
4.6.2 Faktoriaali lõpp ... 186
4.6.3 Goldbachi hüpotees ... 187
4.6.4 Klaaskuulid ... 187
4.6.5 VÜK-võimsus ... 188
4.6.6 Suurim algtegur ... 188
4.6.7 Vahetusraha ... 189
4.6.8 Taandumatud murrud ... 189
4.6.9 Mertensi funktsioon ... 190
4.6.10 Mõrvamüsteerium ... 190
4.7 Viited lisamaterjalidele ... 191
5 Dünaamiline planeerimine algajatele ... 192
5.1 Sissejuhatav ülesanne – Fibonacci jada ... 192
5.1.1 Tavaline rekursioon ehk jõumeetod ... 193
5.1.2 Mäluga rekursioon ... 194
5.1.3 Alt üles DP ... 195
5.1.4 Kokkuhoidlik DP ... 196
5.1.5 DP retsept ... 197
5.2 Lineaarne väärtuste tabel - optimaalne maksmine ... 197
5.2.1 Ahne algoritm ... 197
5.2.2 Kõigi variantide läbivaatamine (rekursioon) ... 198
5.2.3 DP lahendus ... 199

5.3 Pikima kasvava osajada leidmine ... 201
5.3.1 Kõigi võimaluste läbivaatus ... 201
5.3.2 DP lahendus ... 202
5.3.3 Tagurdusmeetodiga lahendus ... 203
5.4 Kahemõõtmeline väärtuste tabel – pikim ühine osajada ... 204
5.4.1 DP lahendus ... 204
5.5 Ristsummade loendamine ... 206
5.5.1 Ristsummade arvutamine ... 206
5.5.2 DP lähenemine ... 206
5.5.3 DP Exceliga ... 207
5.5.4 Tabeli koostamine programselt ... 208
5.5.5 Ülesande lahendus (tabeli kasutamine) ... 209
5.6 Mitmemõõtmeline DP tabel – Miljonär ja vaeslapsed ... 210
5.6.1 Kõikide võimaluste läbivaatus ... 211
5.6.2 Korduvad harud ... 211
5.6.3 DP tabeli koostamine ... 212
5.6.4 Lahendus ... 213
5.7 Tõeväärtuste tabeliga DP - õiglane jagamine ... 215
5.7.1 Kõik kombinatsioonid ... 215
5.7.2 DP lahendus ... 215
5.8 Bitimaskide põhine väärtuste tabel - moekunstnik ja koiliblikas ... 217
5.8.1 Kõigi läbivaatuste puu ... 218
5.8.2 Tee DP poole ... 219
5.8.3 Bitimaskide kasutamine tabeli indeksitena ... 219
5.8.4 DP lahendus ... 221
5.9 Kuidas ja millal DP-d kasutada ... 222
5.9.1 Ülalt alla vs alt üles DP ... 223
5.9.2 Kidunud DP ... 223
5.10 DP praktilised kasutusalad ... 224
5.11 Kontrollülesanded ... 225
5.11.1 Aktsiaturg ... 225
5.11.2 Protsessori planeerimine ... 225
5.11.3 Maksmise võimalused ... 226
5.11.4 Statistika manipuleerimine ... 226
5.11.5 Lennukikütus ... 227
5.11.6 Kahjuritõrje ... 228
5.11.7 Torni ehitamine ... 229
5.11.8 Putin sukeldumas ... 230
5.11.9 Arvude liitmine ... 230
5.11.10 Templisambad ... 231
5.12 Viited lisamaterjalidele ... 231
6 Sissejuhatus graafiteooriasse ... 232
6.1 Graafiteooria terminid ... 233
6.1.1 Suunatud ja suunamata servad ... 233
6.1.2 Teed graafis ... 233
6.1.3 Sidususkomponendid ... 234
6.2 Graafide esitusviisid ... 234
6.2.1 Naabrusmaatriks ... 235

6.2.2 Tippude loend ... 235
6.2.3 Servade loend ... 236
6.2.4 Regulaarsete servadega graafid ... 237
6.3 Graafi läbimine ... 237
6.3.1 Sügavuti läbimine ... 237
6.3.2 Ülesanne: Ratsu teekond ... 239
6.3.3 Laiuti läbimine ... 242
6.3.4 Ülesanne: Ratsu teekond 2 ... 243
6.4 Sidususkomponentide leidmine ... 245
6.4.1 Ülesanne: Ratsud ... 245
6.5 Üleujutamine ... 247
6.5.1 Ülesanne: Veekogud ... 247
6.6 Kahealuseline graaf ... 248
6.7 Topoloogiline sorteerimine ... 249
6.7.1 Ülesanne: Loomad ... 250
6.8 Kaalutud servadega graafid ... 252
6.9 Dijkstra lühima tee leidmise algoritm ... 252
6.9.1 Ülesanne: sõiduaeg ... 255
6.10 Puud ... 257
6.10.1 Puud kui graafid ... 257
6.10.2 Puu definitsioon ... 257
6.10.3 Graafitöötlusalgoritmid puul ... 257
6.10.4 Kahendpuu sügavuti läbimine ... 258
6.10.5 Puu laiuti läbimine ... 259
6.10.6 Ülesanne: Kahendpuud ... 259
6.11 Kontrollülesanded ... 262
6.11.1 Robotivõistlus ... 262
6.11.2 Civilization ... 263
6.11.3 Doominod ... 263
6.11.4 Projektiplaan ... 264
6.11.5 Sõnateisendused ... 264
6.11.6 Kahevärviprobleem ... 265
6.11.7 Joonejälgija robot ... 266
6.11.8 Numbriruudud ... 267
6.11.9 Optimaalne ruuting ... 268
6.11.10 Liftid ... 269
6.12 Viited lisamaterjalidele ... 270
7 Efektiivne programmeerimistehnika ... 271
7.1 Koodistiil ... 272
7.1.1 Pikaajaline ja lühiajaline lähenemine ... 272
7.1.2 Alamprogrammid ... 272
7.1.3 Näide: Tankid ... 274
7.1.4 Kommentaarid ... 278
7.1.5 Nimed ... 280
7.1.6 Funktsioonide nimetamine ... 281
7.2 Testimine ... 281
7.2.1 Testide koostamine ... 282
7.2.2 Ümbermõõt ... 282

7.2.3 Testide genereerimine ... 285
7.2.4 Käsuinterpretaatorid ja skriptimine ... 287
7.2.5 Valideerimine ... 289
7.3 Tunne oma keelt ... 290
7.3.1 Teegid ... 290
7.3.2 Andmestruktuurid ... 290
7.3.3 Makrod ... 291
7.4 Võistluste strateegia ... 292
7.5 Ahned algoritmid ... 292
7.5.1 Mündid ... 293
7.5.2 Kingid ... 294
7.5.3 Majakavahid ... 297
7.5.4 Veel ahnetest algoritmidest ... 299
7.5.5 Suveniirid ... 299
7.6 Kontrollülesanded ... 303
7.6.1 Onu Robert ... 303
7.6.2 Bussijuhid ... 304
7.6.3 Hernehirmutised ... 304
7.6.4 Kolimine ... 305
7.6.5 Krokodillid ... 305
7.6.6 Lohe Gorõnitš ... 306
7.6.7 Fraktsioonid ... 307
7.6.8 Bitimask ... 307
7.6.9 LED-lambid ... 308
7.6.10 Antimonotoonne jada ... 309
7.7 Viited lisamaterjalidele ... 309
8 Dünaamiline planeerimine edasijõudnutele ... 310
8.1 Seljakoti pakkimine ... 310
8.1.1 Ülesanne: Kohver ... 310
8.1.2 Dünaamiline planeerimine ... 311
8.1.3 Rekursioonivalem ... 311
8.1.4 Tabeli täitmine ... 311
8.1.5 Kood ... 312
8.1.6 Tee taastamine ... 312
8.1.7 Keerukus ... 313
8.1.8 Seljakoti erinevad variatsioonid ... 313
8.2 Rändkaupmehe ülesanne ... 313
8.2.1 Ülesanne: kaupmees ... 314
8.2.2 Rekursioonivalem ... 314
8.2.3 Valemi kasutus ... 314
8.2.4 DP tabel ... 315
8.2.5 Kood ... 316
8.2.6 Tee taastamine ... 317
8.2.7 Keerukus ... 317
8.3 Levenshteini kaugus ... 317
8.3.1 Ülesanne: Levenshteini kaugus ... 318
8.3.2 Rekursioonivalem ... 319
8.3.3 DP tabel ... 319

8.3.4 Teisenduse taastamine tabelist ... 320
8.3.5 Ajaline ja mahuline keerukus ... 320
8.3.6 Levenshteini kauguse omadused ... 321
8.3.7 Levenshteini kauguse variatsioone ... 321
8.4 Geenijärjestuste võrdlemine ... 321
8.4.1 Ülesanne: Sarnasus ... 322
8.4.2 Ülesanne: Mis on ühist? ... 323
8.4.3 Rekursioonivalem ... 323
8.4.4 Tabel ... 324
8.4.5 Kood ... 324
8.4.6 Tee taastamine ... 324
8.5 Dünaamiline Aja painutamine (Dynamic Time Warping) ... 325
8.5.1 Ülesanne: helid ... 325
8.5.2 Rekursioonivalem ... 326
8.5.3 Tabel ... 326
8.5.4 Kood ... 326
8.6 Lõikudega ülesanded ... 327
8.6.1 Ülesanne: palgi lõikamine ... 327
8.6.2 Tee DP-ni ... 327
8.6.3 Rekursioonivalem ... 327
8.6.4 Tabeli täitmine ... 327
8.6.5 Kood ... 329
8.6.6 Tee taastamine ... 329
8.6.7 Keerukus ... 330
8.7 Maatriksite korrutamine ... 330
8.7.1 Ülesanne: maatriksid ... 330
8.7.2 Võimaluste arv ... 330
8.7.3 Tee DPni ... 331
8.7.4 Tabeli täitmine ... 331
8.7.5 Kood ... 332
8.7.6 Keerukus ... 332
8.8 Ülesanne: mäed ... 333
8.8.1 Rekursioonivalem ... 334
8.8.2 Tabeli täitmine ... 335
8.8.3 Kood ... 335
8.8.4 Keerukus ... 336
8.9 Kontrollülesanded ... 337
8.9.1 Torn ... 337
8.9.2 Bussireisid ... 338
8.9.3 Sillad ... 339
8.9.4 Mäng ... 340
8.9.5 Uuring ... 341
8.9.6 Alkeemik ... 342
8.9.7 Puhkus ... 343
8.9.8 Tango ... 344
8.9.9 Valimised ... 345
8.9.10 Plaan ... 346
8.10 Viited lisamaterjalidele ... 346

9 Eriotstarbelised puud 347
9.1 Lõikude puu 347
9.1.1 Puu loomine jadast 348
9.1.2 Lõigul väärtuse leidmine 349
9.1.3 Väärtuse muutmine 351
9.1.4 Väärtuste muutmine tervel lõigul 351
9.2 Laisk väärtustamine lõikude puudes 352
9.2.1 Paberiribade kleepimine 355
9.2.2 Kood 359
9.2.3 Ülesanne: Sein 362
9.3 Fenwicki puu 364
9.3.1 Summa leidmine 365
9.3.2 Elemendi lisamine ja muutmine 366
9.3.3 Üksiku elemendi leidmine 366
9.3.4 Indeksi leidmine kumulatiivse summa järgi 367
9.3.5 Teatud lõigu summa leidmine 367
9.4 2-mõõtmeline Fenwicki puu 367
9.4.1 Lisamine 368
9.4.2 Loendamine 368
9.5 Ajalooga andmestruktuurid 368
9.5.1 Ajalooga lõikude puu 369
9.5.2 Ülesanne: Tekstiredaktor 370
9.5.3 Kood 372
9.6 Hõre tabel 373
9.6.1 Tabeli koostamine 373
9.6.2 Tabelist lugemine 374
9.7 Ühisosata hulgad 374
9.7.1 Kasutamine 376
9.8 Kontrollülesanded 377
9.8.1 Laoseisud 377
9.8.2 Korrutamismäng 378
9.8.3 Külaskäigud 379
9.8.4 Berliini müür 379
9.8.5 Lambipirnid 380
9.8.6 Taani valimised 381
9.8.7 Rivistus 381
9.8.8 Madalad majad 382
9.8.9 383
9.8.10 Paberi lõikamine 384
9.9 Viited lisamaterjalidele 384
10 Graafiteooria 385
10.1 Floyd-Warshalli algoritm 385
10.1.1 Ülesanne: tuleb hiljem 385
10.1.2 Rekursioonivalem 386
10.1.3 Tabeli täitmine 386
10.1.4 Floyd-Warshalli algoritmi kasutusi 388
10.2 Toese mõiste 389
10.2.1 Servade liigitamine 389

10.2.2 DFS koos servade liigitamisega ... 390
10.3 Sillad ja Artikulatsioonipunktid ... 391
10.3.1 Ülesanne: Muinastulede öö ... 392
10.3.2 Sildade leidmine ... 396
10.4 Euleri Graaf ... 397
10.4.1 Euleri tsükli leidmine ... 397
10.4.2 Loomanimed ... 398
10.5 Minimaalne Toes ... 400
10.5.1 Minimaalse toese leidmine ... 401
10.5.2 Ülesanne: Elektrikatkestus ... 402
10.6 Maksimaalne voog ja minimaalne lõige ... 404
10.6.1 Võrk ... 404
10.6.2 Voog ... 405
10.6.3 Maksimaalne voog ... 405
10.6.4 Maksimaalse voo leidmine ... 406
10.6.5 Mitme lähte ja suudmega ülesanded ... 408
10.6.6 Kaaludega tipud võrgus ... 408
10.6.7 Lõige ... 409
10.6.8 Vangid ... 410
10.7 klikid ... 413
10.7.1 Bron-Kerboschi algoritm ... 413
10.7.2 Sõbrad ... 414
10.8 Kontrollülesanded ... 416
10.8.1 Laadimispunktid ... 416
10.8.2 Internetiühendus ... 417
10.8.3 Kiviheitemasin ... 418
10.8.4 Uus tee ... 419
10.8.5 Mõjuvõim ... 420
10.8.6 Civilization ... 421
10.8.7 Põgenemine ... 422
10.8.8 Teepuhastus ... 423
10.8.9 Doomino ... 423
10.8.10 Valimised ... 424
10.9 Viited lisamaterjalidele ... 424
11 Tekstialgoritmid ... 425
11.1 Teksti parsimine ... 425
11.1.1 Ülesanne: arvutustehe ... 426
11.2 Alamstringi leidmine ... 428
11.2.1 Mõisted ... 428
11.2.2 Lihtne alamstringi otsimise algoritm ... 429
11.3 Knuth-Morris-Pratti algoritm ... 430
11.3.1 Kokkulangevuste tabel ... 431
11.3.2 Hulknurgad ... 432
11.4 Sufiksipuud ... 434
11.4.1 Prefiksipuu ... 434
11.4.2 Sufiksipuu ... 435
11.4.3 Sufiksipuu kasutamine ... 437
11.4.4 Sufiksipuu loomine ... 440

11.5 Sufiksiloend ... 440
11.5.1 Loomine ... 441
11.5.2 Sufiksiloendi kasutamine ... 443
11.6 Räsi ... 444
11.6.1 Alamstringi leidmine räsi abil ... 445
11.6.2 Räsi moodularvutusega ... 446
11.6.3 Jagamise vältimine ... 447
11.6.4 Rabin-Karpi algoritm ... 447
11.6.5 Ülesanne ... 448
11.7 Kontrollülesanded ... 450
11.7.1 Telefoninumbrid ... 450
11.7.2 Enigma ... 451
11.7.3 Universaalgeen ... 451
11.7.4 Arelofansi keel ... 452
11.7.5 Koodigeneraator ... 453
11.7.6 Automaathelistaja ... 454
11.7.7 Risti-rästi ... 455
11.7.8 Ahvitembud ... 455
11.7.9 Palindroomid ... 456
11.7.10 Otsing ... 457
11.8 Viited lisamaterjalidele ... 458
12 Arvutusgeomeetria ... 459
12.1 Geomeetrilised objektid ... 459
12.1.1 Punkt ... 459
12.1.2 Lõik ... 459
12.1.3 Vektor ... 460
12.1.4 Sirge ... 460
12.1.5 Kiir ... 461
12.1.6 Murdjoon ... 461
12.1.7 Hulknurk ... 462
12.1.8 Ringjoon ... 462
12.1.9 Kaar, kõõl, sektor ja segment ... 463
12.2 operatsioone punktide ja sirgetega ... 463
12.2.1 Punkti pööramine tasandil ... 463
12.2.2 Kahe punkti vaheline kaugus ... 463
12.2.3 Paralleelsuse ja samasihilisuse kontroll ... 464
12.2.4 Sirgete lõikepunkt ... 464
12.2.5 Lõikude lõikumine ... 465
12.3 Hulknurk ... 467
12.3.1 Hulknurga ümbermõõt ja pindala ... 467
12.3.2 Picki teoreem ... 469
12.3.3 Punkti sisaldumine hulknurgas ... 469
12.4 Koordinaatide pakkimine ... 470
12.4.1 Ülesanne: Tikuruudud ... 472
12.5 Kumer kate ... 474
12.6 Ülesanne: ring ja ruut ... 478
12.7 Lähim punktipaar ... 482
12.8 Magic missile (ülesanne) ... 483

12.9 Kontrollülesanded 488
12.9.1 Liitumine 488
12.9.2 Augumäng 489
12.9.3 Pallimäng 489
12.9.4 Nelinurgad 490
12.9.5 Üleujutus 491
12.9.6 Park 491
12.9.7 Müür 492
12.9.8 Spioonid 493
12.9.9 Märklaud 494
12.9.10 Viirus 495

0 SISSEJUHATUS

0.1 MIS ON VÕISTLUSPROGRAMMEERIMINE?

Programmeerimise eesmärk on panna arvuti tegema midagi kasulikku. Enamasti on see mingi tegevus, mille muidu oleks ära teinud inimesed. Nemad saavad nüüd aga teha midagi meeldivamat – las masin töötab, tema on rauast. Distsipliinina on programmeerimine üsna noor, kuid ometi läbi teinud pead pööritama paneva arengu, eelkõige kasutatavate vahendite (arvutid ja programmeerimist abistav tarkvara) osas. Kaasaegne riistvara võimaldab „rauast masinal“ sooritada teatud ülesandeid miljardeid kordi kiiremini kui ükski inimene suudaks. Samas mõeldakse pidevalt välja uusi ja keerulisemaid ülesandeid, mis lükkavad tagant ka programmeerimise sisemist võimekust ehk algoritmide ja meetodite arengut.

Uute algoritmide ja meetodite väljamõtlemine loob aga viljaka pinnase programmeerijatele üksteisega mõõduvõtmiseks. Kes kirjutab kiiremini efektiivsema programmi etteantud ülesande lahendamiseks? Selles vallas toimub palju võistlusi, kus pannakse proovile osalejate teadmised algoritmide alal, võimekus oma mõtteid kiiresti programmideks realiseerida ning oskus kirjutada veavaba koodi.

Eestis on selles vallas peamine võistlus Eesti Informaatikaolümpiaad, mis on suunatud eelkõige kooliõpilastele, kuid mille raames toimub ka kõigile avatud üritusi. Edukamatel kooliõpilastel on võimalus osaleda regionaalsel Balti Informaatikaolümpiaadil (kus osalevad Läänemere äärsed riigid) ning Rahvusvahelisel Informaatikaolümpiaadil (IOI).

Ülikoolitasemel on peamine võistlus ACM International Collegiate Programming Contest (ACM-ICPC). Neil, kes on koolid juba lõpetanud, on võimalik osaleda erafirmade korraldatud võistlustel, nagu Google Code Jam või Facebook Hacker Cup. Peale selle on olemas hulk internetipõhiseid võistluskeskkondi, kus toimuvad regulaarsed üritused. 2017 alguse seisuga on neist tuntumad CodeForces, TopCoder, HackerRank ja CodeChef.

Programmeerimisvõistlustel on üldjuhul ajapiirang (näiteks 2-5 tundi), mille jooksul on antud lahendamiseks hulk ülesandeid (enamasti 3-6). Ülesannetele antakse sisendandmed, mille põhjal tuleb leida väljundandmed, mille õigsust kontrollitakse.

Lahendus võib olla iseseisev programm (sel juhul loeb ta sisendit ja väljundit failist või standardsisendist ja kirjutab väljundi teise faili või standardväljundisse) või siis lihtsalt funktsioon, mis lingitakse testprogrammi külge ja mida testprogramm välja kutsub, andes sisendi ette funktsiooni parameetritena.

Suurim väljakutse on sageli mitte lihtsalt vastuse leidmiseks sobiva lahenduse leidmine, vaid sobiva **kiire** lahenduse leidmine. Lahendustel on antud ajalimiit, mille jooksul tuleb vastus väljastada, tänapäeval tavaliselt sekund või paar.

Siin on üks lihtne näide võistlusstiilis ülesandest:

> Malelaual peab ratsu liikuma N käiguga ühelt etteantud väljalt teisele. Mitu erinevat teekonna võimalust tal selleks on?

Kui võistleja on lahenduse valmis programmeerinud, esitab ta selle testimiseks. Olenevalt võistluse formaadist võib testimine toimuda kas kohe või lahendamisaja lõpus. Kohese testimise puhul

öeldakse võistlejale, kas programm läbis testid või mitte, nii et tal on võimalus oma lahendust parandada.

Testimisel on üldjuhul neli võimalikku tulemust:

- OK – kõik töötas õigesti.
- Vale vastus – programm ei väljastanud oodatud tulemust.
- Ajalimiit ületatud – programm ei lõpetanud ajapiiri jooksul oma tööd.
- Käivitusaja viga – programmi töös juhtus viga (näiteks mittelubatud mälupiirkonnast lugemine või nulliga jagamine), mis ei lasknud tal tööd lõpetada.

Mõnedel võistlustel on ülesande eest punktide saamiseks vaja korrektselt läbida kõik testid, teistel saab vastavalt läbitud testide arvule osalise arvu punkte. Suurima punktide arvuga võistleja saab auhinna või siis lihtsalt au ja kuulsust.

Selle teksti kirjutamise ajal on maailma parim võistlusprogrammeerija Gennadi Korotkevitš Valgevenest, kes praegu esindab St.Peterburgi ITMO ülikooli. Korotkevitš hakkas olulisi võistlusi võitma juba 11-aastaselt, sai IOI-lt kokku kuus kuldmedalit ning on hetkel olulisematel võistlussaitidel maailma kõrgeima reitinguga.

Eesti seni edukaim võistlusprogrammeerija on olnud Martin Pettai, kes sai aastatel 1999-2002 IOI-lt kolm kuld- ja ühe hõbemedali.

0.2 MIDA SIIT ÕPIKUST LEIDA VÕIB?

Käesoleva õpiku peamine eesmärk on ehitada sild programmeerimisoskuse ja teoreetilise arvutiteaduse vahele. Eesti keeles on mitmesuguseid programmeerimist õpetavaid raamatuid, kus keskendutakse programmeerimistehnikale. Samuti on ilmunud arvutiteaduse sissejuhatavaid õpikuid, kus räägitakse mitmesugustest teoreetilistest küsimustest.

Ka isiklikult olen ma elus kohanud paljusid inimesi, kes jäävad kas ühte või teise serva: ühel pool on puhtad praktikud, kes pole põhjalikumat teooriat omandanud või on selle unustanud kui „mittevajaliku“. Teisel pool on teoreetikud, kes tunnevad paljusid algoritme, kuid kellele kuluks ära rohkem praktilist kogemust nende realiseerimisel.

Siin raamatus tuuakse teoreetiline ja praktiline pool kokku ja räägitakse mõlemast. Käsitletakse nii võtteid efektiivsemaks programmeerimiseks kui ka algoritme, mis võimaldavad esmapilgul võimatuid ülesandeid mõistliku ajaga lahendada.

Tutvustatakse mitmeid algoritme ja andmestruktuure, millest mõned kuuluvad arvutiteaduse parimate saavutuste sekka, kuid teoreemide ja tõestuste asemel on siin hulk praktiliselt läbiprogrammeeritavaid näiteid. Raamatuga kaasas olevas lisas on võimalik tutvuda paljude näidisprogrammidega ja lahendada mitmesuguse raskusega katseülesandeid enesekontrolliks.

Õpiku ülesehitus võimaldab seda kasutada struktureeritud kursustel, kus õpetatakse programmeerimist või arvutiteadust üldiselt või siis tegeletakse spetsiifiliselt programmeerimisvõistlusteks ettevalmistamisega. Teiselt poolt võiks see olla kasutatav praktilise käsiraamatuna, kust on vajadusel võimalik ühe või teise algoritmi kohta infot leida.

Tegemist ei ole programmeerimise algkursusega, eelduseks on, et lugejad on algtasemel programmeerimist juba õppinud ning oskavad oma lemmikkeeles põhiasjadega toime tulla.

Raamatust võivad kasu saada mitmesugused huvilised:



Esiteks **kooliõpilased**, kellel on olemas teadmised programmeerimise põhialustest ja kes soovivad teistega mõõtu võtta. Raamat sisaldab suurel hulgal teadmisi, mida on vaja informaatikaolümpiaadil. 2016. aasta lõpu seisuga on raamatu esimese osa läbitöötamine piisav, et olla edukas Eesti olümpiaadidel ja teine osa võimaldaks koju tuua medali rahvusvahelistelt võistlustelt.

Teiseks **üliõpilased ja kraadiõppurid**, kes soovivad täiendada oma teadmisi arvutiteaduses, neid samal ajal praktiliste oskustega seostades. Õpik pakub mitmeid kasulikke lähenemisviise, mis võimaldavad uurimistöödes vajalikke programme efektiivsemalt kirjutada.

Kolmandaks **professionaalsed programmeerijad**, kes soovivad omandada algoritmiliselt keerulisemate ülesannete lahendamise oskust. Võistlustel ja olümpiaadidel programmeerimine on nagu rallisõit, kus läheb vaja spetsiifilisi oskusi. Igapäevane tavaprogrammeerimine mõnes ettevõttes on selle kõrval nagu liinibussi või takso juhtimine – sarnaste elementidega, aga siiski mitte päris sama. On aga olukordi, kus rallioskused kuluvad ka igapäevaelus marjaks ära. Olen ise oma karjääri jooksul nii mõnelgi korral suutnud „võimatut“ teha ja mõne asja kümneid või sadu kordi kiiremini tööle panna, samal ajal kui teised olid juba valmis massiivseks täiendava riistvara ostuks. Ja kes ei tahaks vahel kangelane olla :)

Neljandaks **õpetajad ja õppejõud**, kes soovivad oma õpilasi olümpiaadideks ja võistlusteks ette valmistada. Raamat on üles ehitatud struktuurselt, iga järgnev peatükk kasutab ja täiendab eelmistes õpitut. Tekstis on palju ülesandeid, mida võib anda lahendamiseks nii grupitööna kui individuaalselt.

Viiendaks **ambitsioonikad spetsialistid**, kes soovivad minna tööle mõnesse maailma „kõrgliiga“ tehnoloogiaettevõttesse. Ettevõtetes nagu Google, Facebook või Palantir on tavaline, et tööintervjuudel antakse kandidaatidele ülesandeid näiteks graafiteooria või dünaamilise planeerimise vallast. Seega on intervjuu läbimiseks vaja ka võistlusprogrammeerimise alaseid teadmisi. Eestis pole nii põhjalik valmistumine tööintervjuudeks veel väga levinud, aga maailma tipptasemel juhtub sageli, et kandidaadid õpivad intervjuudeks nädalaid või isegi kuid.



Ja lõpuks kõik ülejäänud, kellele meeldib programmeerimine ning teadmiste omandamine.

Raamat on jagatud kaheks osaks ja neist kumbki kuueks peatükiks, millest igaüks käsitleb konkreetset teemat.

Iga peatükk sisaldab järgmisi osi:

1. Tekst, mis avab teema olemust ja teoreetilist tausta.
2. Näidisülesanded, kus põhjalikult selgitatakse, kuidas kirjutada programme vastavate algoritmide realiseerimiseks.
3. Veel mõned ülesanded, kus on antud lahenduse idee ja viited näidislahendustele.
4. Ülesanded enesekontrolliks (või kontrolltöödeks akadeemilises keskkonnas). Kõigi ülesannete lahendusi saab esitada *online* võistluskeskkonnas, kus neid automaatselt testitakse.

Kõik näited on saadaval kolmes programmeerimiskeeles:

- C++ kui võistlustel ja olümpiaadidel enim kasutatav keel
- Java kui Eesti tarkvaratööstuses enim kasutatav keel
- Python kui hariduses ja programmeerimise õppimisel populaarne keel.

Tekstis toodud näited kasutavad C++ varianti, teistes keeltes kirjutatud vasted on saadaval lisamaterjalina.

0.3 KONTROLLÜLESANNETE LAHENDAMINE

Iga peatüki lõpus on toodud rida kontrollülesandeid. Neid on võimalik lahendada omal käel või kui raamatut kasutatakse õppetöös, saab kursuse juhendaja korraldada nende põhjal kontrolltöö.

Ülesandeid saab lahendada online võistluskeskkonnas CodeForces, kus:

- Igale ülesandele on üles seatud automaatsed testid.
- Kasutajad saavad veebibrauseri kaudu esitada oma koodi ning saavad kohest tagasisidet selle kohta, kas programm läbis testid või ei.

Ülesannete lahendamiseks:

- Loo CodeForces'i konto (või logi sisse Google'i kontoga)
- Ühine Eesti Võistlusprogrammeerimise grupiga aadressil http://codeforces.com/group/jODaK73gf0/
- Kontrolltööde jaoks on loodud võistlused, vali neist asjakohane

Lahendused peavad lugema andmed standardsisendist ja kirjutama vastuse standardväljundisse. Kui väljundisse kirjutada veel midagi muud, on tulemuseks tõenäoliselt *Wrong Answer*. Kõik testid vastavad ülesande kirjelduses toodud spetsifikatsioonile ja nende korrektsust eraldi kontrollima ei pea.

0.4 KASUTATUD ÜLESANDED JA PILDID

Näidisülesanded kuuluvad enamikus arvutiteaduse klassikasse loodud spetsiaalselt käesoleva raamatu jaoks, osad on ka taaskasutatud mõnelt Eesti olümpiaadilt. Teistest allikatest pärit ülesanded on vastavalt viidatud.

Kontrolltööde ülesanded on loodud kas selle raamatu jaoks või kohandatud teiste riikide võistlustelt. Kuna enamasti on teiste võistluste ülesanded toodud koos lahendustega, pole neid raamatust otse viidatud, viite saab küsimise peale autoritelt.

Kasutatud pildid on võetud vabadest pildipankadest nagu Pixabay, Pexels, Wikimedia Commons jt, kus nad on antud maailmale kasutamiseks CC0 litsentsi alusel.

0.5 AUTORITEST

Targo õppis programmeerima aastal 1990, mil tal oli ainult tekstirežiimi võimaldav kasutatud IBM XT arvuti, millel polnud mänge, mida mängida. Lahenduseks oli ise mängud kirjutada, mis eeldas omakorda programmeerimise äraõppimist. Pärast seda on ta programmeerinud igasuguseid asju klaasilõikepinkidest ja lennuki elektrisüsteemidest dokumendihalduse ja mobiilirakendusteni. Praegu

juhib Targo ka Eesti Informaatikaolümpiaadi. Kuna talle meeldib rohkem endast rääkida, tähistab tekstis „mina“ enamasti Targot.

Katrin huvitus mängudest ise vähem, kuid kirjutas neid oma nooremale vennale. Hiljem on ta õppinud arvutuslingvistikat ja töötanud süsteemianalüütiku ning programmeerijana, samuti tegelenud võistlusprogrammeerimise õpetamisega Eesti Informaatikaolümpiaadi raames.

0.6 TÄNUAVALDUSED

Tahame tänada paljusid inimesi, kes raamatu valmimisele kaasa aitasid:

- Mihkel Kree ja Jaak Vilo organisatsioonilise toetuse eest,
- Härmel Nestra, Tähvend Uustalu, Liisa Jõgiste, Oliver-Matis Lill, Tiina Kull ja Heno Ivanov sisuliste soovituste ja veaparanduste eest,
- Toomas Tennisberg ja Oliver Tennisberg ülesannete läbilahendamise ja testide koostamise ning kontrollimise eest.

0.7 VEAPARANDUSED JA SOOVITUSED

Tegemist on elava dokumendiga, millest loodame aja jooksul uusi versioone luua. Valdkond areneb kiiresti, samuti võib esile tulla vajadusi materjali täpsustamiseks, mitmesuguseid vigu jne.

Kõik parandused ja soovitused võib saata aadressil targot@gmail.com.

ESIMENE OSA - ALGAJATELE

Esimene osa on jõukohane neile, kel on olemas programmeerimise alusteadmised. Täiendava materjalina võib kasutada oma programmeerimiskeele dokumentatsiooni.

1 PROGRAMMIDE SISEMAAILM

Võrreldes teiste inseneridistsipliinidega on programmeerijate arv maailmas viimasel paarikümnel aastal ülikiiresti kasvanud. Samuti leiutatakse igal aastal palju uusi tehnoloogiaid, meetodeid, raamistikke ja muid vahendeid. Tagajärjena pole kujunenud traditsiooni, kus teadmisi ühelt põlvkonnalt teisele edasi antaks ja nii pole programmeerijatel sellist sügavat ja stabiilset kultuurkihti nagu teistel inseneridel.

Näiteks teede ja sildade ehitamise oskus areneb aeglasemalt, mistõttu on koolis võimalik anda hulk baasteadmisi praktikas kasutatavatest materjalidest ja meetoditest. Käsiraamatutes on tabelid erinevate materjalide ning struktuuride kohta. Veergudena on toodud betoon, puit, teras ja teised materjalid, nende erikaal, tugevus, soojuspaisuvus ja muud näitajad. Iga sillaehitaja teeb eelnevalt arvutused, kui tugevat konstruktsiooni tal vaja on ja valib selle põhjal õiged vahendid.

Programmeerimise maailmal on klassikalise insenerindusega palju ühist, aga ka palju erinevat. Esimene suur erinevus on see, et programmeerijad ei vaja füüsilisi materjale, vaid teevad „mittemillestki midagi“. Seetõttu on võimalik tööd teha suuresti katse ja eksituse meetodil – katkise asja minemaviskamine ja uuesti proovimine ei võta muid ressursse peale aja ja elektri. Teine suur erinevus seisneb selles, et kord valmis tehtud tulemust on võimalik kuitahes palju kordi kopeerida.

Need kaks erisust koos annavad efekti, kus on võimalik saavutada ontlik tulemus lihtsalt kusagilt leitud olemasolevaid tükke kokku sobitades ja nii kaua proovides, kuni asi enam-vähem töötab. Kui algaja programmeerija kirjutab oma esimese „Hello, World“ programmi, siis paneb ta ka mingeid etteantud tükke kokku – esimesed korrad võib-olla natuke valesti, aga lõpuks ilmub kiri ekraanile ning autor on rahul. Üldjuhul ei saa ta aga kohe aru, miks ja kuidas programm sellise tulemuse andis – see nõuab juba põhjalikumat kasutatud vahendite uurimist.

Algaja puhul on selline „tulemus on tähtsam kui mõistmine“ lähenemine aktsepteeritav, kuid paraku on maailmas miljoneid päris keerulisi ja olulisi programme, mis on koostatud samal katse-eksituse meetodil. Sageli sobitatakse vajalikku kohta esimene enam-vähem sobiv komponent, mille osas aga ei mõisteta selle sisulisi karakteristikuid, näiteks jõudluse, ühilduvuse või kõrvalmõjude osas. Ka ilusaid tabeleid, kus need andmed kirjas oleksid, on harva saada. Sillaehituse analoogiat jätkates kujutlegem, et ehitajatel on vaja teatud kujuga detaili. Nad



leiavadki sobiva, see näeb õige välja ja kui ehitajad ise üle silla kõnnivad, siis midagi halba ei juhtu ka. Reaalselt on detail aga mitte metallist, vaid pehmest plastikust ja esimene silda ületav veoauto kukub jõkke.

Võistlusprogrammeerimises, kus rõhk on kirjutatava programmi jõudlusel, on eriti tähtis teada, mis on tegelikult „karu kõhus“, millised on erinevate kasutatavate komponentide omadused ja mis nende komponentide sees toimub. Siin peatükis käsitletaksegi mitmeid programmeerimise ehituskive ja nende omadusi.

1.1 PROGRAMMI ELUTSÜKKEL

Vaatame alustuseks, mida masin meie kirjutatud programmidega teeb. Klassikaline esimene ülesanne on kõigis programmeerimisõpikutes sama.

Kirjutada programm, mis väljastab ekraanile teksti „Tere, maailm!“.

1.1.1 Lähtekood

Kõigepealt tuleb valida endale meelepärane programmeerimiskeel ning seejärel kirjutada selles keeles **lähtekood**. Lähtekood on tavaline tekst, mis vastab mingi programmeerimiskeele reeglitele. Koodi võib kirjutada igas tekstiredaktoris, kuid enamik programmeerijaid kasutab abiprogramme, mida nimetatakse integreeritud arenduskeskkondadeks (*integrated development environment - IDE*). Viimastest tuleb rohkem juttu punktis 1.10.

Ülesannet lahendav lähtekood keeles C++ on järgmine:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Tere, maailm!";
    return 0;
}
```

Programmeerimisvõistlustel tulebki harilikult esitada vaid lähtekood ja selle edasine töötlemine ja käivitamine toimub testserveris.

1.1.2 Lähtekoodi transleerimine



Lähtekood tuleb **transleerida** ehk tõlkida arvutile arusaadavasse keelde – **masinakeelde.** Selleks on kaks erinevat lähenemist: kompileerimine ja interpreteerimine.

Kompileerimine on kogu lähtekoodi tõlkimine mõnda **vahekeelde** või kohe masinakeelde. Programmi, mis oskab kompileerida, nimetatakse **kompilaatoriks.** Kompileeritud kood salvestatakse üldjuhul uude faili, kust seda saab iseseisvalt käivitada ja lähtekoodi pole sinna juurde enam vaja. Maailma esimese kompilaatori kirjutas 1952. aastal Grace Hopper (pildil), kellest sai hiljem USA mereväe kontradmiral.

C++ puhul eelneb kompileerimisele veel eeltöötluse ehk preprotsessori samm, kus programmi koodile lisatakse juurde päisfailide sisu.

Interpretaator transleerib samuti lähtekoodi, kuid teeb seda nii-öelda jooksvalt: tõlgib ühe lause, täidab selles oleva(d) käsu(d), siis tõlgib järgmise lause, täidab selle jne. Selle lähenemise puhul on käivitamisel alati vaja ka lähtekoodi ennast.

Programmeerimiskeeli saabki laias laastus liigitada interpreteeritavateks keelteks ja kompileeritavateks keelteks. C++ ja Java on disainitud kompileeritavateks, Python aga interpreteeritavaks keeleks. Tegelikult võib iga keele jaoks kirjutada kompilaatori või interpretaatori. Nii on olemas ka C interpretaatoreid ja Pythoni kompilaatoreid.

Kuna masinakeelt on inimesel praktiliselt võimatu lugeda ja kirjutada, siis on kasutusel **assemblerkeel** (*assembly language*), milles numbrilised käsud on asendatud tähekombinatsioonidega ja kasutatakse muid süntaktilisi abivahendeid, mis teevad keele inimesele kergemini loetavamaks ja kirjutatavaks. Olenevalt kompilaatorist ja platvormist võib sama koodi kompileerimine anda erineva tulemuse. Alati jääb küll samaks põhimõte, et masinakeelne kood koosneb **masinakäskudest**, mida protsessor saab vahetult täita.

Ülaltoodud „Tere, maailm!“ näite kompileerimine võib anda näiteks sellise tulemuse (esimeses tulbas masinakäsk, edasi assemblerkeelne käsk). Assembleri käsud on omakorda kahes tulbas: esimeses käsk, teises tema argumendid. Kolmetähelised lühendid argumentide seas tähistavad **registreid** (protsessorisiseseid mälupesi), nurksulgudes on antud mäluaadressid. Täpne tulemus oleneb protsessori tüübist ja kompilaatorist, mistõttu sina võid oma arvutis saada teistsugused masinakäsud.

```
; paneme registrisse ecx trükitava stringi aadressi
8B 0D 24 30 20 01     mov         ecx,dword ptr ds:[10D3044h]
; kutsume välja stringi väljastamise operaatori
E8 95 04 00 00   call        std::operator<<<std::char_traits<char> > (010D1740h)
; eax register sisaldab funktsiooni tagastatavat väärtust, meil on vaja tagastada 0.
; eax-i xor iseendaga on efektiivne viis tema nulliks seadmiseks
33 C0                 xor         eax,eax
```

1.1.3 Teegid

Transleeritud programm ei tööta üldjuhul üksinda, vaid kasutab täiendavat välist funktsionaalsust. See funktsionaalsus on realiseeritud abistavates programmides, mida nimetatakse **teekideks**. Tavaliselt on hulk teeke kaasas meie kasutatava programmeerimiskeskkonnaga, millest tähtsaim on käitusteek (*runtime library).* Selles on tavaliselt realiseeritud programmeerimiskeele sisseehitatud funktsioonid, veahaldus, operatsioonisüsteemiga suhtlemine jne.

Mõned käitusteegid on väga väikesed, näiteks C keele crt0 teek annab programmile edasi operatsioonisüsteemilt saadud parameetrid ja tagastab töö lõppedes veakoodi, kuid ei tee eriti muud huvitavat. Teiselt poolt Java Virtuaalmasin hoolitseb programmi mäluhalduse, täiendavate teekide laadimise, koodi õigsuse kontrolli, Java programmi baitkoodist masinkoodi transleerimise ja palju muu eest.

Peale käitusteegi saavad programmid tavaliselt kasutada veel paljusid teisi teeke, milles on mitmesugust abistavat funktsionaalsust alates lihtsatest matemaatilistest funktsioonidest kuni graafiliste animatsioonideni.

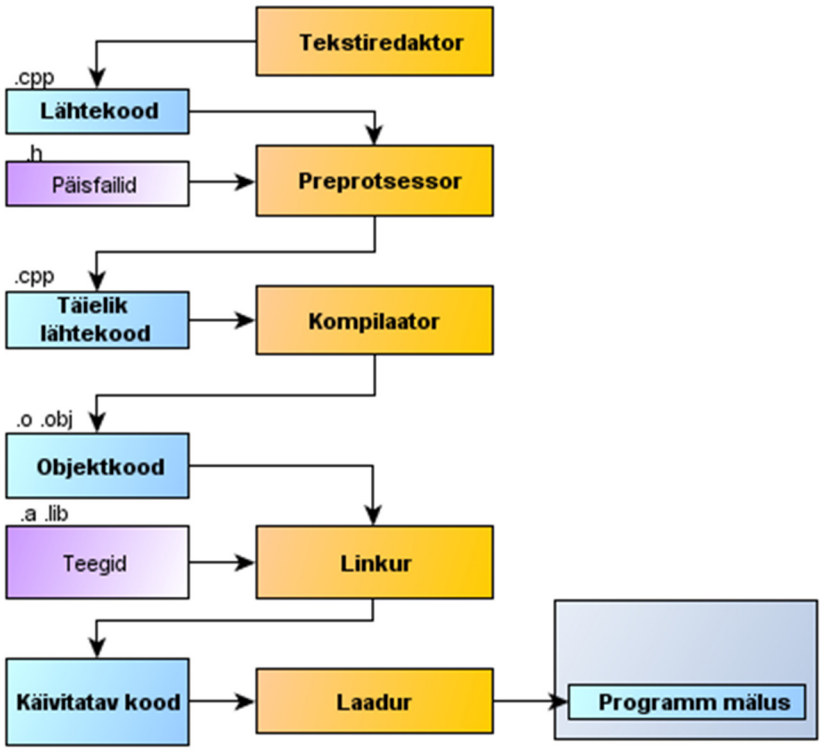
1.1.4 Linkimine

Kompilaatori töö tulemust nimetatakse ka **objektkoodiks**, mida hoitakse objektfailides. Sageli vastab igale lähtekoodi failile eraldi objektfail. Objektfailid sisaldavad masinakäske, mis vastavad esialgsele lähtekoodile, aga pole veel iseseisvalt käivitatavad. Et saada lõplikku programmi, tuleb objektfailid **linkida** omavahel ja ka vajalike väliste teekidega. Linkimine võib olla staatiline (kõik vajalikud failid pannakse kokku üheks käivitatavaks programmifailiks) või dünaamiline (teeke saab laadida programmi töötamise ajal, vastavalt sellele, millal neid vaja läheb). Programmi, mis objektfailid omavahel lingib, nimetatakse **linkuriks.**

1.1.5 Laadimine ja täitmine

Kompileeritud programm tuleb käivitada. Programmi käivitamisel laaditakse masinakeelne programm kõigepealt arvuti mällu. Seda nimetatakse **laadimisjärguks**. Sellele järgneb **käitusjärk**, mille jooksul protsessor mällu laetud masinakäske täidab.

Järgmisel joonisel on skemaatiliselt kujutatud C++ keelse programmi jõudmine lähtekoodi loomisest tekstiredaktoris programmi käituseni:



1.2 ANDMETE HOIDMINE JA TÖÖTLEMINE ARVUTIS

Fundamentaalselt on arvuti vahend andmete töötlemiseks. (Arvuti)programm on kogum instruktsioone arvutile andmete töötlemiseks.

1.2.1 Protsessori tööpõhimõte

Protsessor (*Central Processing Unit*) on see osa arvutist, mis – nagu džinn muinasjutus – täidab kõik su käsud (aga mitte tingimata selle, mida tegelikult soovisid!). Protsessori tähtsamad komponendid

on **juhtseade** käskude juhtimiseks ja **aritmeetika-loogikaseade** (**ALU**) tehete teostamiseks. Protsessori tööks vajalikke andmeid ja tulemusi hoitakse **registrites**.

Igal protsessoril on **käsustik** – konkreetne hulk käske, mida see protsessor täita oskab. Käske on peamiselt kolme tüüpi:

1. Andmeedastuskäsud (näiteks andmete liigutamiseks registrist mällu ja vastupidi).
2. Andmetöötluskäsud, mis sooritavad aritmeetika-loogikatehteid.
3. Siirdekäsud, mis muudavad käskude täitmise järjekorda.

Allpool on näide käsust, mis liidab arvule mälupesas aadressiga 1 väärtuse ning salvestab tulemuse mälupessa aadressiga 2 (MIPS32 protsessori ADDi käsk):

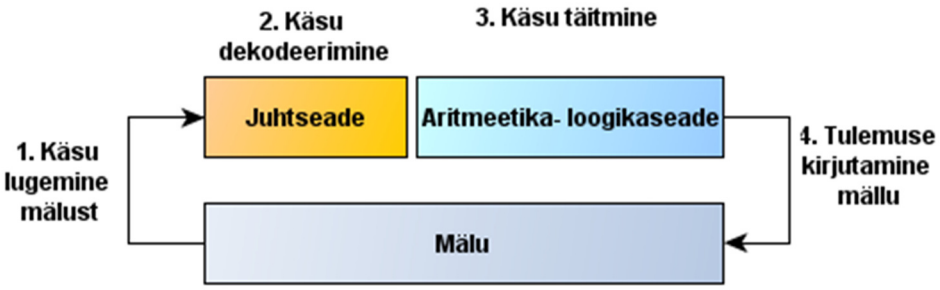
```
Käsukood  Aadress1 Aadress2 Väärtus
00100000001000100000000101010111
```

Protsessorit iseloomustab veel protsessori **sõna** (*word*) pikkus. Sõna on protsessori poolt ühe üksusena töödeldav andmehulk. Registri pikkus peab vastama vähemalt sõna pikkusele.

Masinakeeles programm koos kasutatavate andmetega on salvestatud mällu ning protsessor asub sealt järjest infot lugema ja käske täitma.

Lihtne protsessori töötsükkel on järgmine:

1. Käsuloenduris on järgmise käsu aadress, sellelt aadressilt loetakse uus käsk käsuregistrisse. Käsuloendur ja käsuregister on spetsiaalsed registrid: esimene neist on järgmiste käskude mäluaadresside ja teine täidetavate käskude hoidmiseks.
2. Käsu dekodeerimine.
3. Tehte sooritamine.
4. Tulemuse kirjutamine registrist mällu.



1.2.2 Protsessori jõudlus

Peamine protsessori jõudluse mõõdik on IPS (*Instructions Per Second* – instruktsiooni sekundis), mugavuse mõttes ka MIPS (miljon IPSi). Mikroprotsessorite jõudlus on viimastel aastakümnetel kasvanud mõnelt MIPSilt mõnesaja tuhandeni (ehk sadade miljardite üksikoperatsioonideni sekundis!). Selle saavutamiseks on kasutatud mitmesuguseid võtteid, millest peamised on:

- **Taktsageduse** suurenemine – aastal 1980 2MHz, aastal 2002 3GHz, umbes sinna on taktsagedus ka pidama jäänud.
- **Käsukonveierid** (vt lähemalt allpool) – käskude täitmise jagamine etappideks, mis võimaldab samaaegselt täita erinevate käskude erinevaid etappe.
- **Mitmetuumalised** protsessorid – MIPSe loetakse tavaliselt kõigi tuumade peale kokku.

- **Superskalaarne arhitektuur** ehk mitme käsu samaaegne täitmine samas tuumas. Selleks on sama tuuma sees mitu (sageli erinevatele operatsioonitüüpidele spetsialiseeritud) ALUt, mis käske paralleelselt töötlevad. Siin on oluline, et nad ei rikuks ära üksteise andmeid – mõned programmid on kergemini paralleliseeritavad kui teised.

Tänapäevased paljutuumalised protsessorid suudavad täita sadu miljardeid instruktsioone sekundis, aga seda vaid väga spetsiifilistes oludes, kus neil on kogu aeg töötlemiseks sobivad andmed ees ja tööd on võimalik efektiivselt paralleliseerida. Enamasti ei võimalda asjaolud protsessoritel nii intensiivselt töötada. Tavaprogrammidel on enamasti pudelikaelaks andmete lugemine ja kirjutamine. Võistlusprogrammidel on andmeid tavaliselt üsna vähe ja enamik programmi tööajast kulubki konkreetselt arvutamisele.

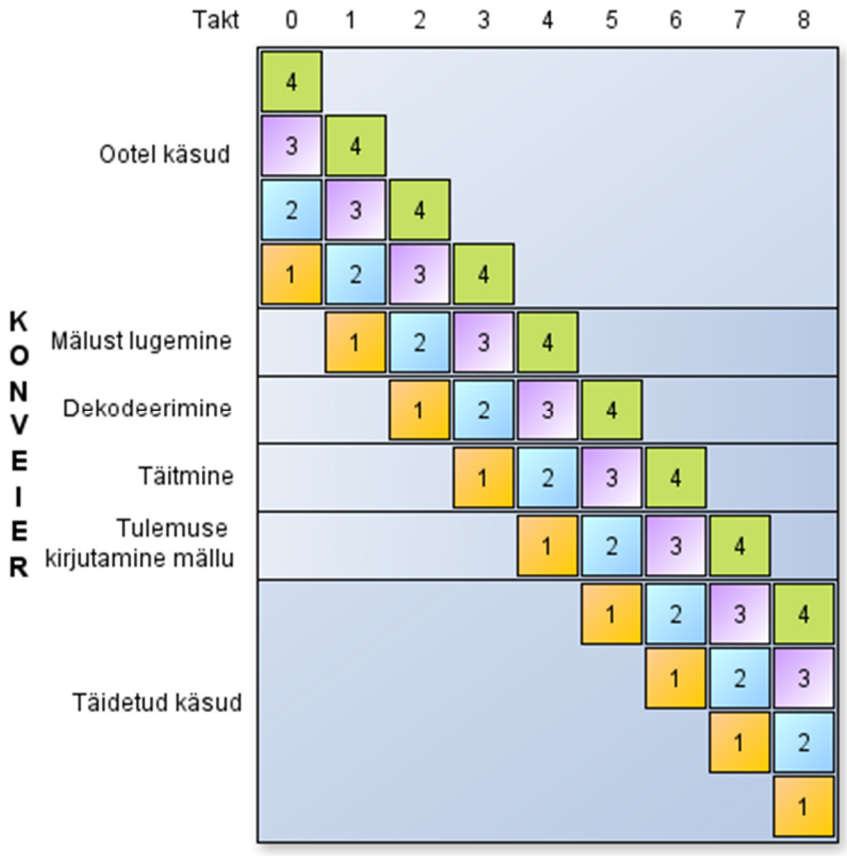
Rusikareeglina soovitatakse võistlusprogrammides arvestada umbes sada miljonit „tavapärast“ operatsiooni sekundi kohta, kus tavapärane on näiteks massiivis kahe väärtuse vahetamine, paarist tehtest koosnev aritmeetiline avaldis või muu suhteliselt lihtsa sisemise keerukusega operatsioon.

See reegel võimaldab hinnata, kas plaanitav algoritm on üldse realistlik. Kui sekundi jooksul tuleb sooritada näiteks üle miljardi operatsiooni, ei töötaks kavandatud lähenemine tõenäoliselt niikuinii ja tuleb leida parem meetod.

1.2.3 Käsukonveier

Protsessor töötab **taktide** kaupa. Ühe masinkeelse käsu täitmiseks kulub mitu takti. Vaatleme eelpool kirjeldatud neljasammulist töötsüklit, kus iga sammu täitmine võtab täpselt ühe takti. Sellisel juhul kulub ühe käsu jaoks neli takti. Kui töödelda käske järjest, nii et iga järgmise käsu töötlust alustatakse alles siis, kui eelmine käsk on lõpetatud, võtab nelja järjestikuse käsu töötlus aega 4*4 = 16 takti. Samas on igal taktil töös erinevad komponendid (näiteks käsu dekodeerimisel dekooder, täitmisel aritmeetika-loogikaüksus). Selleks, et kasutada protsessori aega optimaalsemalt, kasutatakse käsukonveierit – kui üks käsk liigub näiteks dekooderist ALUsse täitmiseks, liigub järgmise käsu info juba dekooderisse.

Protsessi illustreerib järgmine skeem:



Esimesel taktil loetakse esimene käsk mälust, seitsmendal juba kirjutatakse neljanda käsu tulemus mällu – seega kulub 16 takti asemel konveiermeetodit kasutades vaid 7 takti.

Kaasaegsed protsessorid kasutavad lisaks samme, et järjestada käsud optimaalsemalt ümber ja leida, milliseid käske on võimalik käivitada paralleelselt.

1.2.4 Hargnemise ennustamine

Vaatleme näidet, mis illustreerib protsessorite jõudluse sõltuvust sisendandmetest. Olgu meil kood, mis genereerib hulga juhuarve ja liidab suuremad neist kokku:

```
const unsigned arraySize = 32768;
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256; // loome massiivi juhuslikest arvudest

long long sum = 0;
// kordame arvutust 100000 korda, et saada täpsemat tulemust
for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
          sum += data[c];
    }
}
```

Minu arvutis võttis selle koodi jooksutamine aega umbes 12 sekundit. Nüüd teeme aga väikese muudatuse:

```
const unsigned arraySize = 32768;
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;

// sorteerime andmed enne arvutamist
std::sort(data, data + arraySize);

long long sum = 0;
// kordame arvutust 100000 korda, et saada täpsemat tulemust
for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128) // loome massiivi juhuslikest arvudest
          sum += data[c];
    }
}
```

Muudetud programm võttis aega vähem kui 2 sekundit! Milles on asi?

Kõige rohkem käivitatav rida siin programmis on kontroll `if (data[c] >= 128)`. Minu arvutis kompileeritakse see kood järgmisteks käskudeks (esimeses tulbas on käsu aadress mälus):

```
; laadime data[c] väärtuste registrisse eax
00A51320  mov         eax,dword ptr [ecx-8]
; võrdleme eax registri sisu 16-süsteemi arvuga 80h (kümnendsüsteemis 128).
00A51323  cmp         eax,80h
; kui võrdluse tulemus oli negatiivne, hüppame aadressile 0A5132Fh. "jl" tähistab
; korraldust "jump if less" ehk "hüppa, kui on väiksem".
; Pane tähele, kuidas kompilaator keeras esialgse võrdluse teistpidi,
; sest nii on hüppamine loogilisem.
00A51328  jl          main+8Fh (0A5132Fh)
; teeme liitmistehte
00A5132A  cdq
00A5132B  add         edi,eax
00A5132D  adc         esi,edx
; jl tulemus maandub siia
00A5132F  mov         eax,dword ptr [ecx-4]
```

Protsessori jaoks tähendab see, et siin on **hargnemine:** kui andmed on ühesugused, tuleb järgmiseks täita üks käsk; kui teistsugused, siis teine. Naiivne lähenemine oleks ära oodata, mis on kontrolli väärtus ja siis täita käsk kas ühest harust või teisest. See tähendaks aga, et meil pole kasu oma võimsast käsukonveierist ja parallelismist.

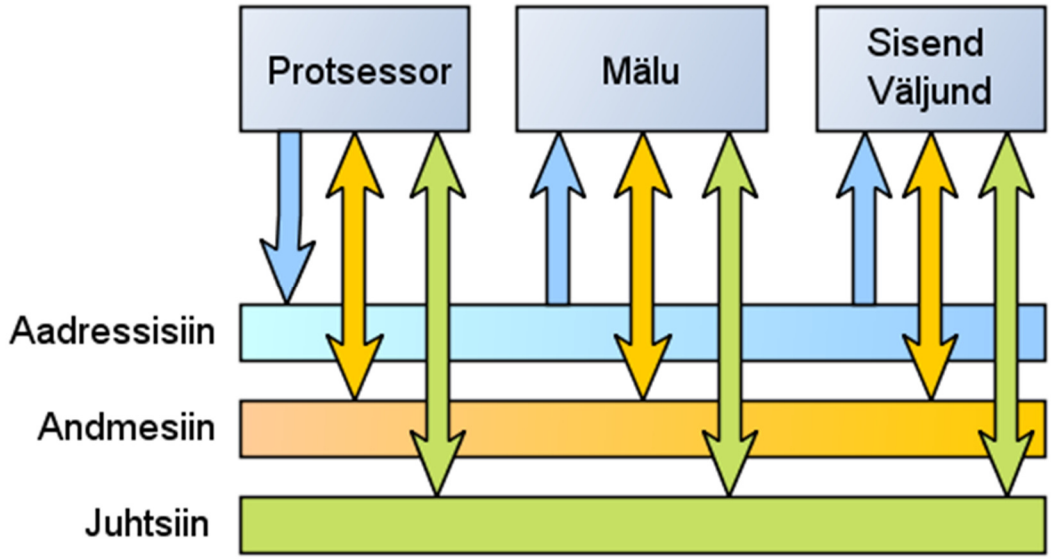
Sellepärast püüabki protsessor **hargnemist ennustada**, s.t mõistatada, kumba haru pidi edasi minnakse. Käsukonveierisse laaditakse vastava haru käsud ette ära. Kui ennustus oli õige, läheb protsess täie kiirusega edasi. Kui aga oli ekslik, siis visatakse konveierist järgmised käsud minema ja laaditakse teise haru käsud.

Kuidas ennustamine toimub? Mõistagi ei saa protsessor meie koodi sisust aru, aga ta saab statistiliselt jälgida, mitu korda mindi selle käsu juures ühele poole ja mitu korda teisele poole. Koodi esimese variandi puhul minnakse hargnemiskohas võrdse tõenäosusega erinevates suundades, mis tähendab, et pooltel kordadel tuleb käsukonveieri sisu ära visata. Teises variandis minnakse aga palju

kordi ühele poole ning siis palju kordi teisele poole. Kui samas hargnemiskohas on korduvalt mindud ühes suunas, saab protsessor sellest aru ning laadib edaspidi just selle haru käsud konveierisse. Efekt on suur – programmide kiiruse erinevus oli enam kui kuuekordne.

1.2.5 Andmete liikumine

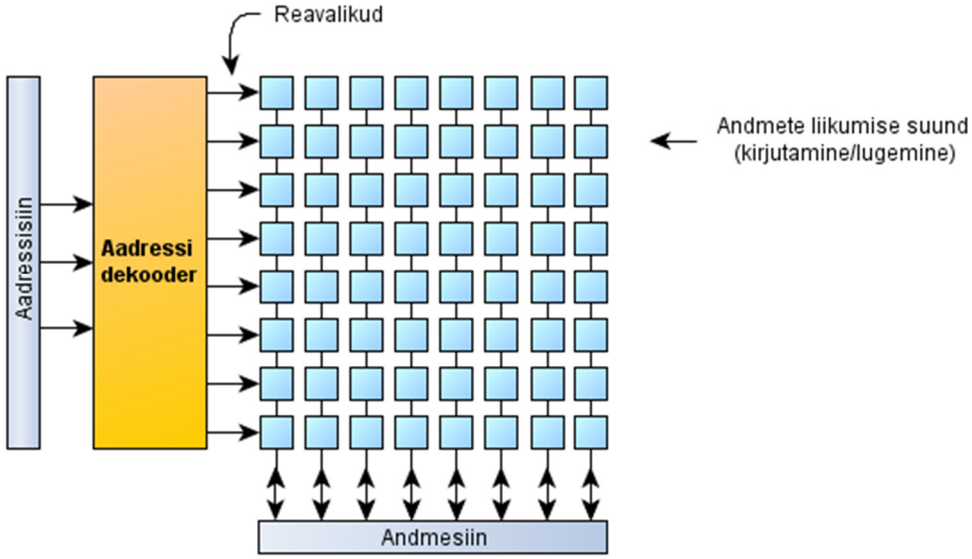
Andmete transportimiseks mälu, protsessori ja sisend-väljundseadmete vahel kasutatakse **siine** (*bus*): **andmesiinil** (*data* bus) liiguvad andmed, **aadressisiinil** (*address bus*) on info, kuhu andmed liiguvad, ja **juhtsiinil** (*control bus)* info, mis suunas ja mille vahel andmed parajasti liiguvad. Kui protsessor tahab mingi väärtuse mällu kirjutada, pannakse vastava mälupesa aadress aadressisiinile ning väärtus ise andmesiinile.



1.2.6 Mälu

Andmeid hoitakse mälus. Mälu jaguneb sisemäluks ehk lihtsalt mäluks ja välismäluks (kõvaketas, mälupulgad ja muud välised vahendid). Sisemälu jaguneb omakorda põhimäluks, vahemäluks ja registriteks.

Põhimälu on enamasti realiseeritud suvapöördusmäluna (*Random access memory - RAM*). See koosneb mälupesikutest. Igas pesik hoiab täpselt 1 biti infot. Igal teatud pikkusega pesikute jadal on oma aadress. Mälu võib ette kujutada ruudustikuna, kus igal real on aadress, esimeses veerus on esimene bitt infot, teises teine jne.



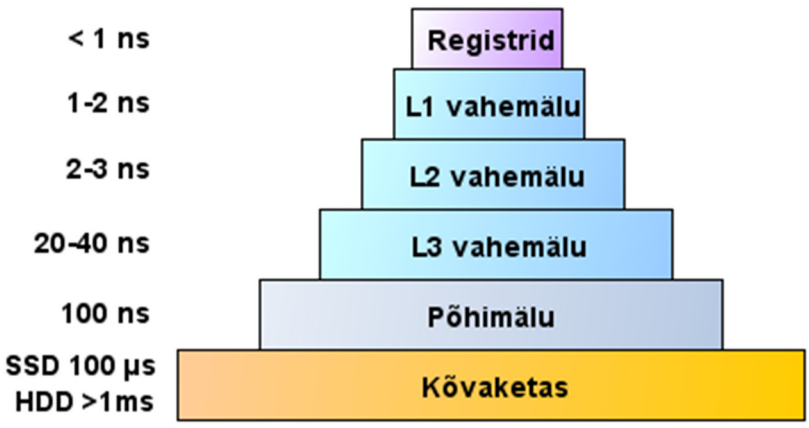
Mälu loeb aadressisiinilt aadressi, aadressi dekooder saadab signaali soovitud aadressliinile ning sellel liinil olevad bitid kirjutatakse andmesiinile või loetakse andmesiinilt vastavale aadressiliinile.

Tavaliselt on korraga kasutusel mitu paralleelset mälukiipi, mis võimaldab korraga salvestada või lugeda pikemat sõna. Osa sõnast salvestatakse ühele kiibile, osa järgmisele jne, aga alati ühele aadressile. Iga kiip kasutab kirjutamiseks ja lugemiseks ainult teatud lõiku andmesiinist. Teise võimalusena on üks sõna mitmel füüsilisel aadressliinil ja aadressi dekooder tõlgib aadressi vastavalt. Siis kirjutatakse ühe rea bitid ühele andmesiini osale, teise rea omad selle kõrvale jne.

Aadressid on alati järjestatud ja seetõttu on andmete leidmine operatiivmälust kiire, kuid siiski mitte nii kiire kui protsessorite töökiirus.

Et andmetega töötamist kiirendada, on protsessoritel oma **vahemälu** (*cache*), kus hoitakse koopiaid põhimälu nendest kohtadest, mis on sagedases kasutuses. Vahemälusid on omakorda mitu taset, esimene tase (L1) on ühe konkreetse tuuma jaoks, järgmised tasemed (L2, L3) on tuumade vahel jagatud. L1 tüüpiline suurus on tavaliselt kümnetes kilobaitides, L2 umbes megabait, L3 8 megabaiti.

Mäludega suhtlemise aega mõõdetakse nanosekundites, mis on näiteks kõvakettaga võrreldes tuhandeid kordi kiirem:



Efektiivse programmi kirjutamise seisukohalt on vahemälu suurus oluline – kui programmi kõik andmed mahuvad ära võimalikult kiiresse vahemällu, võib see tööd oluliselt kiirendada.

1.3 ANDMETÜÜBID

Arvuti mälus hoitakse kõiki andmeid kahendarvudena, programmides on aga vaja mitmesuguseid erinevaid arve, tekste ja paljut muud. Erinevat tüüpi andmete kahendarvudena esitamiseks on välja mõeldud **andmetüübid**.

1.3.1 Täisarvud

Täisarvud on esitatud bitijadana kahendsüsteemis. Täisarvutüübid võivad olla erineva pikkusega ja see defineerib arvu maksimaalse väärtuse. Lisaks võivad täisarvutüübid olla märgiga tüübid (lubavad nii positiivseid kui negatiivseid väärtusi) või märgita tüübid (negatiivsed pole lubatud).

Programmeerimiskeeltes on täisarvu tüüp defineeritud erinevalt. „Masinalähedastes“ keeltes nagu C/C++ on täisarvutüüpe mitu, need vastavad arvudele, millega protsessor opereerib, ja seega sõltub nende pikkus protsessori arhitektuurist. Kunagi oli näiteks tavaline, et C `int` tüüp oli 16-bitine, tänapäeval on ta pea kõikjal 32-bitine. Seetõttu võib juhtuda, et ühes masinas töötab programm korrektselt, teises aga mitte, kuna tekib ületäitumine (*overflow*).

Teine variant on abstraktsem, piiramatu pikkusega täisarv, mida kasutatakse näiteks Pythonis. Kuna protsessoris on arvu pikkus piiratud, peab Python looma pikkade arvude jaoks oma keerulisema struktuuri, kuidas pikka arvu meeles pidada ja sellega tehteid teha. Kui arv on kuni 64-bitine, saab tehte teha vahetult protsessoris, vastasel korral tuleb kasutada eraldi algoritmi, mis on oluliselt aeglasem. Isegi väiksemate arvude puhul peab Python kulutama aega kontrollimiseks, kas arv on liiga suur või mitte ning kas arvutamiseks peab kasutama madalama või kõrgema taseme lähenemist. C/C++ jätab sellised otsused programmeerija hooleks.

Analoogne lahendus on ka Java `BigInteger`, mis pole niivõrd täisarvutüüp kui objekt, mille sees on realiseeritud pikkade arvudega arvutamise tehete algoritmid.

Enamlevinud täisarvutüübid ja nende pikkused on toodud järgnevas tabelis:

Bittide arv	Kümnend-kohti (ligi-kaudu)	Vahemik	C++	Java
8	3	-128 – 127	`char`	`byte`
16	4	-32 768 – 32 767	`short/int*`	`short`
32	10	-2 147 483 648 – 2 147 483 648	`long*`	`int`
64	19	-9 223 372 036 854 775 808 – 9 223 372 036 854 775 807	`long long*`	`long`

**C++* `int` *on vähemalt 16 bitti,* `long` *vähemalt 32 bitti ja* `long long` *vähemalt 64 bitti. GCC realisatsioonides on tavaliselt* `int` *ja* `long` *mõlemad 32 bitti,* `long long` *64 bitti.*

C++ keeles on võimalik kasutada ka märgita tüüpe, nt `ushort`, `ulong`, `uint`. Javas on märgita 16-bitine `char`.

Vahel on (näiteks bitikaupa operatsioonide sooritamisel) vaja teada, kuidas arvud arvutis täpselt esitatud on. Täisarvudel tekib see küsimus eelkõige negatiivsete arvude korral. Märgiga täisarvude kahendsüsteemis esitamiseks on mitmeid erinevaid võimalusi:

- **Pöördkoodis** on negatiivse arvu kõik bitid vastupidised ehk inverteeritud võrreldes vastava arvu absoluutväärtusega.
- **Täiendkoodis** madalamad bitid kuni esimese 1-ni kopeeritakse, edasi inverteeritakse. Täiendkood on võrdne pöördkoodiga, millele on liidetud 1. See on enim levinud viis negatiivsete täisarvude esitamiseks, sest täiendkoodis arvudega toimub liitmine ja korrutamine täpselt sama moodi kui positiivsete täisarvudega, nt 1010 + 0111 = 0001 ja 1110 * 0011 = 1010. Täiendkood võimaldab esitada arve vahemikus $-2^{n-1}...2^{n-1}-1$.
- **Märgibitiga esitus** on kõige lähedasem harjumuspärasele kümnendsüsteemis negatiivsete arvude esitamise viisile. Üks bitt, tavaliselt kõige suurema positsiooniga, määrab arvu märgi, ülejäänud bitid annavad arvu absoluutväärtuse. Seda esitust tänapäeval täisarvude jaoks eriti ei kasutata, kuid kasutatakse ujukomaarvude tüvekohtade esitamiseks.
- **Nihkega esituse** korral tähistab 0 väikseimat võimalikku väärtust, 1 sellest ühe võrra suuremat väärtust jne. Null jääb niimoodi skaala keskele. Nihkega esitust kasutatakse näiteks ujukomaarvude eksponendi hoidmisel.

Arv kümnend-süsteemis	Pöördkood	Täiendkood	Märgibitiga esitus	Nihutamisega esitus
127	01111111	01111111	01111111	11111111
6	00000110	00000110	00000110	10000110
0	00000000	00000000	00000000	10000000
-0	11111111	*puudub*	10000000	*puudub*
-6	11111001	11111010	10000110	01111010
-127	10000000	10000001	11111111	00000001
-128	*puudub*	10000000	*puudub*	00000000

Ületäitumine

Sagedane probleem täisarvudega opereerimisel on **ületäitumine**. Ületäitumine tekib siis, kui täisarvudega sooritatava tehte tulemus ei mahu enam oodatud täisarvutüübi piiridesse. Märgiga täisarvude korral muutub sellisel juhul ootamatult vastuse märk. Näiteks 8-bitiste arvude korral liites 01111111 (127) + 01111111 (127) = 11111110 (-2). Ületäitumise vältimiseks:

1. Vali sobiva pikkusega täisarv, arvestades, et ka vajalikud vahetulemused piiridesse ära mahuks.
2. Kui probleemiks on piiridest väljuvad vahetulemused, mõtle hoolega läbi teostavate tehete järjekord.

1.3.2 Ujukomaarvud

Reaalarvude esitamiseks arvutis kasutatakse peamiselt ujukomaarve. Ujukomaarv koosneb kolmest osast: märgist, tüvenumbritest ehk **mantiss**ist ja eksponendist mingil määratud alusel. Harilikult on selleks aluseks kasutusel 2, 10 või 16.
Esituse põhimõte on sarnane kümnendsüsteemist tuntud esitusele:

$12.34 = 1234 * 10^{-2} = 1.234 * 10^{1}$

Viimast varianti, kus on täpselt üks tüvekoht enne koma, nimetatakse arvu normaliseeritud kujuks.

Tavaliselt kasutatakse ujukomaarvude esitamiseks eksponenti alusel 2. Nagu kümnendsüsteemis $1.625 = 1 + 6*10^{-1} + 2*10^{-2} + 5*10^{-3}$, nii ka kahendsüsteemis arv $1.101 = 1 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$. Kahendsüsteemis ujukomaarve hoitakse arvutis tavaliselt normaliseeritud kujul. Kuna

kahendsüsteemis on esimeseks tüvekohaks alati 1, siis sageli seda ei märgita ja võidetakse nii üks lisabitt mantissi märkimiseks. Seda märkimata bitti nimetatakse ka peidetud bitiks. Mõned näited:

Kümnend-esitus	Kahend-esitus	Normali-seeritud	Nihutatud eksponent	Märk, eksponent, mantiss
1.625	1.101	$1.101*2^{0}$	127	`0 01111111 10100000000000000000000`
-7.835	-111.111	$-1.11111*2^{2}$	129	`1 10000001 11111000000000000000000`
0.15625	0.00101	$1.01*2^{-3}$	124	`0 01111100 01000000000000000000000`

Nagu täisarvude puhul, tuleb ka ujukomaarvu esitamisel saada hakkama teatud hulga bittidega. Seetõttu tuleb leida tasakaal arvu täpsuse ja võimalike väärtuste piiride vahel.
Standardina on kasutusel järgmise täpsusega ujukomaarvud (alusel 2):

Nimetus	Bittide arv	Sellest tüvekoha jaoks	Tüvekohti kümnendsüsteemis (ligikaudu)	C/C++	Java	Python
single precision	32	24	7	float	float	-
double precision	64	53	16	double	double	float
double extended	79	64	19	long double	-	-

Lisaks on defineeritud ka spetsiaalsed väärtused: +∞, -∞ ja NaN (*not a number*). Nende esitus:

Väärtus	Märk, eksponent, mantiss
+0	`0 00000000 00000000000000000000000`
-0	`1 00000000 00000000000000000000000`
+∞	`0 11111111 00000000000000000000000`
-∞	`1 11111111 00000000000000000000000`
NaN	`x 11111111 xxxxxxxxxxxxxxxxxxxxxxx`

Probleemid ujukomaarvudega

Kahe ujukomaarvu liitmisel viiakse need arvud kõigepealt ühisele eksponendile, milleks on suurema arvu eksponent. Selleks nihutatakse mantissi vastava hulga bitte paremale. Seejärel teisendatakse mantiss täiendkoodi ning teostatakse arvutustehe. Vastus teisendatakse täiendkoodist tagasi märgiga esitusse ning seejärel normaliseeritakse.

Sellest tulenevalt tekib kaks probleemi: väga erineva eksponendiga arvude liitmisel läheb suur osa väiksema arvu täpsusest kaduma (paremale nihutamise tagajärjel). Väga suurele arvule väga väikese arvu liitmisel väike arv lihtsalt nullitakse ära. Näites kui ülesandes tuleb liita palju väikeseid arve, mille oodatav summa on suur, siis järjest summale juurde liites tegelikult viimaseid arve ei arvestatagi. Sellisel juhul on parem liita kõigepealt väikseimad arvud ja siis saadud vastusele suuremad.

Teiseks probleemiks on väga lähedaste arvude lahutamine. Kuna arvud on ligikaudsed, siis väiksemad bitid on ebaolulisemad ja ebatäpsemad. Lahutades aga kaks lähestikku asuvat arvu, jäävad alles ainult need kahtlase väärtusega bitid, mis nihutatakse normaliseerimise käigus kõrgematele kohtadele.

Reaalarvude hulk on lõputult tihe, kuid teatud arvu bittidega saab edasi anda vaid osa neist arvudest. Väärtused, mida ei saa täpselt esitada, ümardatakse. Kümnendmurruna ei saa täpselt edasi anda arvu 1/3. Võime kirjutada 0,33333333 ja kuitahes palju kolmesid, kuid tulemus on ikka ebatäpne – alati peame ümardama. Analoogselt ei saa kahendsüsteemis täpselt edasi anda arvu 1/10 – tekib lõpmatu perioodiline murd, milles on paratamatult ümardamisviga.

Teeme kontrolliks läbi järgmise näite Pythonis:

```
>>> x = 7*0.01
>>> y = x*100
>>> print(x, y, y-7)
0.07 7.000000000000001 8.881784197001252e-16
```

Nendest ebatäpsustest tulenevalt on kahe ujukomaarvu võrdsust keeruline hinnata, näiteks eelmist näidet jätkates:

```
>>> y == 7
False
```

Seetõttu on praktiline kasutada reaalarvude võrdlemisel väikest arvu, mida tavaliselt nimetatakse epsiloniks, ning võrrelda, kas kahe arvu vahe absoluutväärtus on väiksem kui epsilon:

```
>>> e = 1.0e-10
>>> abs(y - 7) < e
True
```

Pythonis on selleks ka funktsioon `math.isclose`:

```
>>> math.isclose(y,7)
True
```

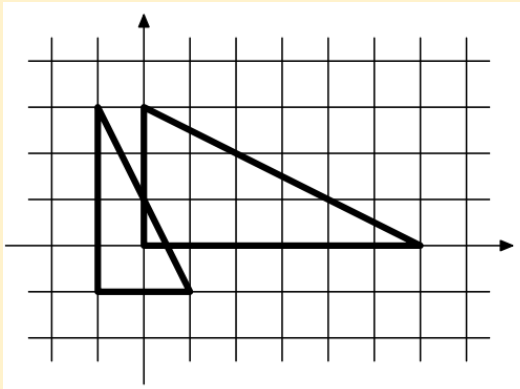
Ujukomaarvu konverteerimisel täisarvuks kasutatakse enamasti 0 suunas ümardamist, mis tähendab murdosa ärajätmist. Seetõttu võib täisarvuks konverteerimine anda ootamatuid tulemusi. Soovitatav on kasutada alati eelnevalt mõnd sobivat ümardamismeetodit ning vajadusel võtta taas abiks epsilon.

Kirjeldus	Näited	C++	Java	Python
Ümardab lähima väärtuseni, vahepealsed paarisarvuks	22.5 -> 22 1.5 -> 2	-	rint	round
Ümardab lähima väärtuseni, vahepealne nullist eemale	22.5 -> 23 -2.5 -> -3	round	round	-
Ümardab väiksemaks	22.5 -> 22 1.9-> 1	floor	floor	math.floor
Ümardab suuremaks	22.5 -> 23 1.1 -> 2	ceiling	ceil	math.ceil
Ümardab nulli suunas	22.5 -> 22 -2.5 -> -2	trunc		math.trunc

Tänapäeval muutub järjest sagedasemaks ka alusel 10 ujukomaarvude kasutamine. Pythonis on selleks klass `decimal`, Javas `BigDecimal`. Peamiseks eeliseks on see, et arvude täpsus on inimesele harjumuspärasem, st 0,1 ja 0,01 on täpselt esitatavad. Eriti oluline on see rahaliste väärtustega opereerimisel, kus deebet ja kreedit peavad alati klappima ning ümardamisest tulenevad erinevused võivad segadust tekitada.

Vaatame ujukomaarvudega seoses ka üht konkreetset ülesannet 2016. aasta informaatikaolümpiaadi eelvoorust:

Antud on kahe kolmnurga tippude koordinaadid. Leia, kas kolmnurgad on sarnased ja kui on, siis kui palju on esimene kolmnurk teisest suurem (sarnasustegur). Sisendi esimesel real on kuus täisarvu lõigust -10^9 kuni 10^9: esimese kolmnurga tippude x- ja y-koordinaadid. Teisel real on samuti kuus arvu: teise kolmnurga tippude koordinaadid. Tipud võivad olla antud nii päripäeva kui vastupäeva järjekorras. Antud punktid moodustavad alati kolmnurga (pole ühtelangevaid punkte ega sirgnurki). Kui kolmnurgad on sarnased, siis väljastada täpselt üks reaalarv, mis näitab, mitu korda on esimene kolmnurk suurem kui teine (kui esimene kolmnurk on väiksem, on ka vastus väiksem kui 1). Kui kolmnurgad ei ole sarnased, väljastada `-1`.



NÄIDE 1:

```
0 0 6 0 0 3
-1 3 1 -1 -1 -1
```

Vastus: `1.5` (vt joonist)

NÄIDE 2:

```
0 0 3 0 1 1
0 0 2 0 1 1
```

Vastus: `-1`

Esimene pähetulev lahendus on selline:

```
#include <math.h>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
	int x1[3];
	int y1[3];
	int x2[3];
	int y2[3];
	cin >> x1[0] >> y1[0] >> x1[1] >> y1[1] >> x1[2] >> y1[2];
	cin >> x2[0] >> y2[0] >> x2[1] >> y2[1] >> x2[2] >> y2[2];

	double k1[3];
	double k2[3];

	for (int i = 0; i < 3; i++) {
	  // hypot funktsioon leiab Pythagorase teoreemi abil täisnurkse kolmnurga
	  //hüpotenuusi
		k1[i] = hypot(x1[i] - x1[(i + 1) % 3], y1[i] - y1[(i + 1) % 3]);
		k2[i] = hypot(x2[i] - x2[(i + 1) % 3], y2[i] - y2[(i + 1) % 3]);
	}

	// et vältida kõigi kombinatsioonide võrdlemist, sordime küljed
	sort(k1, k1 + 3);
	sort(k2, k2 + 3);

	double r[3];
	// leiame külgede suhted
	for (int i = 0; i < 3; i++){
		r[i] = k1[i] / k2[i];
	}
```

```
	double epsilon = 0.0000001;
	// võrdleme suhteid omavahel, kasutades epsiloni
	if (abs(r[0] - r[1]) < epsilon && abs(r[0] - r[2]) < epsilon) {
		cout << r[0];
	}
	else {
		cout << -1;
	}
}
```

See pole halb lahendus, kuid siiski on siin probleemid. Mis oleks näiteks mõistlik epsilon? Kui võtta liiga suur epsilon ja teine kolmnurk on esimesest nt miljon korda suurem, võime kergesti saada valepositiivse tulemuse. Kui epsilon aga väga väikeseks muuta, võime saada valenegatiivse tulemuse väga suurte kolmnurkade puhul.

Hoopis kavalam on aga üldse vältida ujukomaarvudega arvutamist. Praegune lahendus võrdleb põhimõtteliselt kolmnurkade külgede suhteid ja kontrollib, kas mingite arvude puhul kehtib a/b=c/d. See on aga ekvivalentne võrdusega a*d=b*c, nii et meil pole üldse jagamist vaja. Ja mis veelgi parem, see on ka ekvivalentne võrdusega $a^2*d^2=b^2*c^2$, nii et meil pole iga külje jaoks ka ruutjuurt vaja (isegi kui ruutjuur on hypot funktsiooni sisse peidetud)!

Tulemusena saame järgmise koodi:

```
#include <math.h>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
	int x1[3];
	int y1[3];
	int x2[3];
	int y2[3];
	cin >> x1[0] >> y1[0] >> x1[1] >> y1[1] >> x1[2] >> y1[2];
	cin >> x2[0] >> y2[0] >> x2[1] >> y2[1] >> x2[2] >> y2[2];

	double k1[3];
	double k2[3];

	// küljepikkuste ruudud
	double kr1[3];
	double kr2[3];

	for (int i = 0; i < 3; i++) {
		int dx = x1[i] - x1[(i + 1) % 3];
		int dy = y1[i] - y1[(i + 1) % 3];
		kr1[i] = dx*dx + dy*dy;
		dx = x2[i] - x2[(i + 1) % 3];
		dy = y2[i] - y2[(i + 1) % 3];
		kr2[i] = dx*dx + dy*dy;
	}

	sort(kr1, kr1 + 3);
	sort(kr2, kr2 + 3);
```

```
	// kasutame asjaolu, et a/b = c/d <=> a*d = b*c <=> a^2*d^2 = b^2*c^2
	if (kr1[0] * kr2[1] == kr1[1] * kr2[0] && kr1[0] * kr2[2] == kr1[2] * kr2[0]) {
		cout << sqrt((double)kr1[0] / kr2[0]);
	}
	else {
		cout << -1;
	}
}
```

Sellised ujukomaarvudest vabanemise võtted tasub meeles pidada, vahel päästavad nad salakavalatest vigadest.

1.3.3 Püsikomaarvud

Püsikomaarvus on ette antud, millised bitid moodustavad arvu täisosa ja millised murdosa. Nii on täis- ja murdosa pikkused fikseeritud, mistõttu esitatavate arvude ulatus on väiksem kui ujukomaarvudel. Püsikomaarvude peamiseks eeliseks on see, et nendega saab arvutada samamoodi kui täisarvudega. Tänapäeval kasutatakse arvutites püsikomaarve harva.

1.3.4 Märgid

Märgid (*character*) on kirjamärgid (tähed, numbrid, kirjavahemärgid, tühik) ja juhtmärgid. Märke hoitakse mälus täisarvudena ehk koodidena ja see, mis märk mis koodile vastab, on määratud kasutatava **märgistik**uga. Enim kasutatavad märgistikud on nt ASCII ja Unicode.

C/C++ ja Javas on märgitüübiks `char`. C/C++ keeles on märgi pikkuseks harilikult 8 bitti, Javas 16 bitti. Python eraldi tüübina märki ei toeta, märk on string pikkusega 1.

Kuna märke hoitakse arvutis nagu tavalisi täisarve, saab nendega sooritada tehteid nagu täisarvudega. Nt 'S' + 'a' – 'A' = 's'. Tulemus sõltub muidugi kasutatavast märgistikust.

1.3.5 Tõeväärtused

Tõeväärtusmuutuja ehk loogikamuutuja väärtuseks saavad olla loogikaväärtused tõene (*true*) ja väär (*false*). Kahe väärtuse jaoks piisaks muidugi juba ühest bitist, kuid isegi keeltes, kus on loogikamuutuja eraldi tüübina olemas, on selle pikkuseks harilikult terve protsessori sõna ehk vähemalt üks bait.

	C++	Java	Python
Loogikamuutuja tüüp	`bool`	`boolean`	`bool`
Tõese/väära väärtuse konstant	`true/false`	`true/false`	`True/False`

C++ keeles on küll defineeritud tüüp `bool`, kuid toimub automaatne teisendamine täisarvutüübi `int`'i ja `bool`'i vahel. Täisarvust tõeväärtuseks teisendamisel 0 = `false` ja iga muud väärtust käsitletakse kui tõest. Vastupidisel teisendusel `false` = 0 ja `true` = 1. Nii saab kasutada täisarve tõeväärtustena ja vastupidi.

Sarnane lähenemine on ka Pythonis, kus `bool` on täisarvutüübi `int` alamklass. Sellel on kaks väärtust: `True` ja `False`, mis on eriversioonid 1-st ja 0-st ja käituvad aritmeetilistes tehetes vastavalt:

```
>>> True + True
2
```

Tavalist arvväärtust 0, null-väärtust, tühja stringi ja tühje loendeid käsitletakse loogikatehetes `False`'ina, kõik ülejäänud väärtused vastavad väärtusele `True`.

Javas on lihttüüp `boolean`, millel saavad olla väärtused `true` ja `false`. Javas automaatset teisendamist ei toimu ja Java `boolean`i ei saa vahetult teisendada `int`'iks ega vastupidi (vähemalt mitte üldjuhul). Vajadusel saab teisendada järgmiste võtetega:

```
boolean b;
int a = 4;
b = (a != 0); /*kui a = 0, siis b = false, vastasel juhul b = true*/
int a = b ? 1 : 0; /*kui b = true, siis a = 1, vastasel juhul a = 0*/
```

Loogikatehted

Enamikus programmeerimiskeeltes, sh siin raamatus käsitletavates keeltes, on olemas operaatorid loogikatehete JA, VÕI ning EI jaoks. Arvutused teostatakse vasakult paremale. See on oluline, kuna teatud juhtudel ei arvutata kõikide operandide väärtusi välja:

- JA korral, kui vasakpoolne avaldis on väär, siis kogu tulemus on väär ja parempoolset avaldist ei arvutata.
- VÕI korral, kui vasakpoolne avaldis on tõene, siis kogu tulemus on tõene ja parempoolset avaldist ei arvutata.

Näiteks lauses

```
if ((i < 10) && ( ++i < n)) { /*...*/ }
```

suurendatakse i väärtust ainult siis, kui avaldis i < 10 on tõene.

Algajad programmeerijad õpivad seda reeglit tundma enamasti läbi vigade. Kui põhimõte on aga selge, siis saab sama omadust ära kasutada ka teadlikult, paigutades odavama kontrolli esimeseks.

1.3.6 Viidad

Viit on andmetüüp mingi objekti mäluaadressi hoidmiseks. Viidad on vajalikud kahel peamisel põhjusel:

- Kui me anname funktsioonile ette parameetreid, siis näiteks arvuliste parameetrite korral tehakse funktsiooni jaoks neist eraldi koopia. Kui parameetrina on vaja edastada aga tuhandest väärtusest koosnev objekt, pole mõtet sellest koopiat teha, parameetrina võib kasutada lihtsalt selle objekti aadressi ehk viita.
- Kui meil on vaja sama muutujaga opereerida mitmes funktsioonis, nii et ühes funktsioonis tehtud muudatus on näha ka teises, peab mõlemal olema viit sellele muutujale.

C/C++ keeles peab programmeerija viitu ise käsitlema. Javas ja Pythonis luuakse viidad automaatselt ning see, mida me programmi tekstis näeme kui objekti, on tegelikult viit objektile.

1.3.7 Struktuursed tüübid ehk liittüübid

Väärtuste ühekaupa muutujatesse salvestamine on suuremate andmehulkade puhul üsna tülikas, aga nende järgnev töötlemine oleks üldse praktiliselt võimatu. Kui suuremat kogust andmeid on vaja korraga meeles pidada, kasutatakse nn struktuurseid andmetüüpe. See tähendab, et ei ole üksikut väärtust, vaid on palju väärtuseid, mis on ühendatud ühtsesse struktuuri ja lisaks võib andmete

paiknemine struktuuris anda edasi täiendavat infot (näiteks nimed tähestiku järjekorras). Järgmistes punktides on kirjeldatud tavalisemaid liittüüpe:

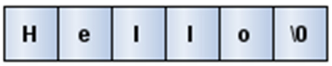
1.3.8 Kirjed

Kirje (*record*) võimaldab loogiliselt koos hoida erinevaid andmeid, mis käivad sama asja kohta, näiteks inimese nime, isikukoodi ja silmavärvi.

1.3.9 Stringid

String ehk sõne on märkide jada ja kuna seda kasutatakse palju, on see sageli defineeritud eraldi andmetüübina. Stringide hoidmiseks mälus on kasutusel peamiselt kolm erinevat võimalust:

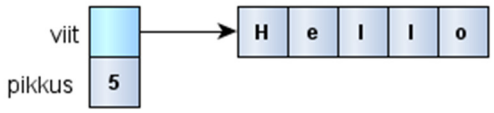
- **Nulliga lõpetatud** string ehk C string lõpeb märgiga, mille väärtus on 0 ehk NUL. String pikkusega n võtab mälus n + 1 kohta.



- **Pikkusega algava** stringi algusest on eraldatud teatud hulk baite stringi pikkuse hoidmiseks. Selliseid stringe kutsutakse Pascali ehk P-stringideks.



- Stringid on esitatud **kirjete** või **objektidena**, kus pikkus ja märgijada salvestatakse eraldi muutujatesse. Enamasti on selliste kirjete sisemine struktuur küll peidetud ja otse neid muutujaid kasutada ei saa.



1.3.10 Massiivid

Massiiv (*array*) on liittüüp, kus hoitakse ainult ühte tüüpi väärtuseid. Igal väärtusel ehk elemendil on oma kindel koht massiivis, elemendid on järjestatud ja neile pääseb ligi indeksi järgi. Indeks on elemendi asukoha kaugus massiivi algusest.

C/C++ keeles on massiivist elementide leidmine hästi lihtne. Massiivi elemendid säilitatakse mälus järjestikustel aadressidel ning kuna iga elemendi andmetüüp ja seega pikkus on sama, siis elemendi leidmise operatsioon on tegelikult massiivi viidale elemendi järjenumbri juurde liitmine, et leida elemendi aadress mälus. Seetõttu on C/C++ massiivid hästi kiired.

C/C++ massiive illustreerib järgmine näide, mida ma olen kasutanud tööintervjuudel, et kontrollida kui hästi kandidaat keelt tunneb:

Mida väljastab selline kood?

```
cout << 4["hello world"];
```

Enamik kandidaate arvab, et selline asi pole võimalik, kuid tegelikult on loogika järgmine:

```
4["hello world"] == 4+"hello world" == "hello world"+4 == "hello world"[4]
```

Kompilaator teeb selle teisenduse meile nähtamatult ja kood väljastab lihtsalt vastava märgi stringist „hello world“ ehk o. Sellised trikid pole reaalselt muidugi soovitatavad, kuid illustreerivad C/C++ suuremat vabadust ja lihtsusega kaasnevat jõudlust. Teisest küljest on ka C++ keeles vigade tegemine kergem ja nende avastamine sageli keerulisem.

Python klassikalist massiivi ei kasuta, seal on kasutusel **järjend** (*list*). Järjend on sarnane massiivile, kuid kui massiivis on enamasti elemendid järjestikuste mäluplokkidena, siis järjendis on viidad elementidele.

1.4 OPERATSIOONID STRINGIDEGA

1.4.1 Märk kohal i

Programmeerimisvõistlustel kasutatavates stringiülesannetes on pea alati vaja string märk-märgilt läbi käia ja iga märki eraldi vaadelda. Võimalused stringis s kohal i asuva märgi saamiseks on toodud järgmises tabelis:

	C++	Java	Python
Süntaks	`s[i]` `s.at(i)`	`s.charAt(i)`	`s[i]`

Näide s="kala"	`s[2] #'l'`
	`s.at(2) #'l'`
		`s.charAt(2) #'l'`	`s[2] #"l"`
			`a[-4] # "k"`
			`(negatiivne argument loendab alates lõpust)`

Erisused	`s[i]` ei hoiata, kui loetakse väljaspoolt stringi piire		Tagastab stringi pikkusega 1

Kui aga on soov märki kohal i muuta, siis enamikus keeltes seda analoogselt teha ei saa. Näide Pythonis:

```
>>> s = "kala"
>>> s[2] = "n"
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    s[2] = "n"
TypeError: 'str' object does not support item assignment
```

C++ keeles saab nii muuta märke „C stiilis“ stringidel, mis on lihtsalt märkide massiivid. C++ standard library `string` klassi kasutamisel pole üksikute märkide muutmine võimalik.

1.4.2 Stringide võrdlemine

Sageli on vaja üht stringi võrrelda teisega. Leksikograafilisel (tähestiku järgi) võrdlemisel loetakse sõnad võrdseks siis, kui nad on märk-märgilt samad. Kahest stringist väiksem on see, mis asub tähestikus eespool. C++ ja Pythonis saab stringide leksikograafiliseks võrdlemiseks kasutada tavalisi operaatoreid (==, !=, >, <, <=, >=), Javas saab küll stringide vahele kirjutada võrdlusoperaatorid, kuid siis ei võrrelda mitte leksikograafiliselt stringe, vaid stringiobjektide aadresse. Kui on vaja teada, kas kaks stringi on leksikograafiliselt võrdsed või mitte, saab Javas kasutada meetodit `equals`. C++ ja Javas on stringide leksikograafiliseks võrdlemiseks meetod `compare`, mis tagastab täisarvu 0, kui stringid on võrdsed, arvu 1, kui esimene on suurem kui teine, ja arvu -1, kui teine on suurem kui esimene.

	C++	Java	Python
Süntaks	`s <= t` `s.compare(t)`	`s.compare(t)` `s.equals(t)`	`s <= t`
Näide **s="kala"** **t="kana"**	`s <= t #true` `s.compare(t) #-1`	`s.compare(t) #-1` `s.equals(t) # false`	`s <= t #true`

1.4.3 Stringide ühendamine ehk liitmine

Stringide liitmine on samuti tihti kasutust leidev operatsioon. Stringide liitmise süntaks on lihtne: nii Javas, Pythonis kui ka C++ kasutatakse liitmisoperaatorit + stringide ühendamiseks. Pythoni näide:

```
>>> "kala" + "kana"
'kalakana'
```

See pealtnäha lihtne operatsioon võib aga tõsiseid probleeme tekitada. Vaatame järgmist ülesannet:

On antud string s, mis koosneb ainult väikestest ladina tähtedest. Väljastada sama pikk string, kus võrreldes esialgsega on a-tähed asendatud järgmise reegli alusel: Kõige esimene a jääb muutumatuks, järgmine on asendatud b-ga, kolmas c-ga jne. Kui a-sid on rohkem kui 26, siis hakkab ring otsast peale: 27. a jääb alles, 28. asendatakse jälle b-ga jne kuni stringi lõpuni.
NÄIDE:
`s = vanapagan`
Vastus: `vanbpcgdn`

Vaatame lähemalt järgmist Javas kirjutatud meetodit, mis antud ülesande lahendab:

```
int len = s.length();
String s2 = "";
int alpha = 0;
for (i = 0; i < len; i++) {
    char ch = s.charAt(i);
    if (ch == 'a') {
        ch += alpha;
        alpha = (alpha + 1) % 26;
    }
    s2 += ch;
}
System.out.println(s2);
```

Selleks, et töödelda 50000 märgist koosnev sisend, kus on 5000 a-tähte, kulub minu arvutis üle 2 sekundi, mis on ilmselgelt liiga palju.

Kuhu kõik see aeg kulub? Suurim probleem on stringide liitmine: iga kord, kui stringile b liidetakse juurde täht ’a’, tekitatakse Javas tegelikult tulemuse jaoks täiesti uus stringiobjekt. Samamoodi käitub stringide liitmisel ka Python.

Kui pöördume tagasi teema juurde, kuidas stringe mälus esitatakse, saab ka selgeks, miks see nii on: stringil on mälus mingi fikseeritud koht ja me ei saa sinna lihtsalt niisama midagi juurde lisada. Seetõttu käib stringide liitmine enamikus keeltes järgmiste sammudena:

- Võtame mälust uue puhvri, mille pikkus on esialgsete stringide pikkuste summa
- Kopeerime esimese stringi sinna puhvrisse
- Kopeerime teise stringi sinna puhvrisse

Seega, kui meil on vaja liita 5000 väikest stringi, siis kopeerime esimese stringi sisu 4999 korda, mis võtabki kokku palju aega.

Mis võiks olla lahendus? Üks võimalus on vältida oma algoritmis stringide kleepimist. Näiteks antud juhul võime teisendatud märgid kohe välja kirjutada, kasutades sellist koodi:

```
int len = s.length();
int alpha = 0;
for (i = 0; i < len; i++) {
    char ch = s.charAt(i);
    if (ch == 'a') {
        ch += alpha;
        alpha = (alpha + 1) % 26;
    }
System.out.print(ch);
```

Selle jõudlus on palju parem, aega kulub 0,1 sekundit, aga ka see pole väga efektiivne, sest peame iga märgi jaoks väljundfunktsiooni välja kutsuma.

Üldjuhul on paljude stringide liitmise korral lahenduseks kohe piisavalt suure puhvri võtmine, selleks on erinevates keeltes spetsiaalsed abistavad klassid, Javas näiteks `StringBuilder. StringBuilder` võtab sisemiselt kohe suurema puhvri, mille sees liitmist toimetab. Kui puhver saab täis, võetakse uus puhver, kuid taas varuga.

Ülesannet lahendav kood on nüüd selline:

```
int len = s.length();
StringBuilder sb = new StringBuilder();
int alpha = 0;
for (i = 0; i < len; i++) {
    char ch = s.charAt(i);
    if (ch == 'a') {
        ch += alpha;
        alpha = (alpha + 1) % 26;
    }
    sb.append(ch);
}
System.out.println(sb);
```

Tööajaks 0,002 sekundit ehk esialgsest 1000 korda vähem aega!

C++ stringid on fikseeritud ja puhverdatud variandi hübriidid. Nendes on mälu võetud teatud varuga ning kui kleepimise tulemus mahub selle varu sisse ära, pole uut puhvrit vaja võtta. Kui ära ei mahu, võetakse siiski uus puhver, taas teatud varuga.

Vastav C++ kood on siin:

```
string s;
string s2 = "";
char alpha = 'a';
for (int i = 0; i < s.length(); i++)
{
	if (s[i] == 'a')
	{
		s2 += alpha;
		alpha++;
		if (alpha > 'z')
		{
			alpha = 'a';
		}
		continue;
	}
	s2 += s[i];
}
```

Käivitamise aeg on 0,03 sekundit ehk samuti kusagil vahepeal. Spetsialiseeritud kiirem C++ klass stringide liitmiseks on `std::stringstream`.

1.5 SISEND-VÄLJUND

Programmeerimisvõistlustel antakse programmile ette ülesandes kirjeldatud reeglitele vastav sisend ning programm peab oma töö tulemusena väljastama etteantud kirjeldusele vastava väljundi. Üldiselt võib võistleja eeldada, et sisend vastab tõesti täpselt kirjeldatud formaadile ning täiendavaid kontrolle sisendi korrektsuse osas ei ole vaja võistlusprogrammis realiseerida. Samas eeldatakse, et ka väljund vastab täpselt kirjeldusele ja selle eest hoolitsemine on programmeerija mure.

Sisend- ja väljundseadmetele ligipääsu ja töökorraldust juhib operatsioonisüsteem. Selleks, et programmeerija tööd lihtsustada, on programmeerimiskeeltes olemas vahendid sisendi ja väljundiga töötamiseks, mis teevad ära vajaliku töö operatsioonisüsteemiga suhtlemiseks.

Sisestamine ja väljastamine toimub üldjuhul kas standardsisendist/väljundist või siis etteantud nimega failist.

1.5.1 Standardvood

Standardvood on eelseadistatud sisend- ja väljund-suhtluskanalid programmi ja keskkonna vahel, kus programm käivitati. Standardvoogudeks on standardsisend (stdin), standardväljund (stdout) ja voog veateadete jaoks (stderr).

C++
C++ keeles on päisfailis `iostream` defineeritud vood standardsisendi- ja väljundi jaoks: `cin` on sisendvoog `ja` `cout` väljundvoog. Voogudel on antud eritähendus operaatoritele `>> ja <<`. Nooled näitavad andmete liikumise suunda: >> loeb voost ja << kirjutab voogu. Kui on vaja, et info kohe ekraanile jõuaks, siis tuleb lisada lõppu `endl`, mis lisab reavahetuse ja väljastab teksti.

```
string s;
cin >> s;
cout << s << endl;
```

Java

Javas kasutatakse standardsisendi ja -väljundi jaoks klassi `System` nimeruumis `java`. Sisendvooks on `System.in` ja väljundvooks `System.out`. Väljundiga on lihtne: sellel on meetodid `println` ja `print`, mis väljastavad stringi. Kui on vaja tagada, et info kohe ekraanile jõuaks, kasutatakse selleks `flush` meetodit:

```
System.out.println("kala");
System.out.flush();
```

Standardsisendist lugemiseks kasutatakse tavaliselt `java` klassi `Scanner` või suuremate sisendite korral klassi `BufferedReader`. Mõlemad klassid on defineeritud nimeruumis `java.io`.

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

BufferedReader br = new BufferedReader(System.in);
String s = br.ReadLine();
```

Python

Pythonis saab kasutada standardsisendile ja -väljundile ligipääsuks objekte `sys.stdin` ja `sys.stdout`. Sisendist lugemiseks on meetod `readline` ja kirjutamiseks `write`, info koheseks kirjutamiseks puhvrist ekraanile on meetod `flush`:

```
s = sys.stdin.readline()
sys.stdout.write(s + "\n")
sys.stdout.flush()
```

Sisendvoo katkestamine

Mõnedes ülesannetes öeldud, et kasutaja võib sisestada klaviatuurilt suvalise hulga teksti ja seejärel oleks vaja kuidagi teada anda, et kasutaja on lõpetanud. Selleks on spetsiaalne märk EOT (*end of transmission*), eesti keeles siis „side lõpp“. See, kuidas kasutaja seda väljendada saab, sõltub konkreetsest keskkonnast, enamasti sobib ctrl + D või ctrl + Z.

Standardvoogudest lugemise ja kirjutamise näited on Eesti Informaatikaolümpiaadide lehel: http://eio.ut.ee/KKK/StdIO

1.5.2 Failidest lugemine ja kirjutamine

Eesti programmeerimisvõistlustel tuleb enamasti lugeda sisendandmed etteantud nimega failist ja vastus kirjutada teise, samuti etteantud nimega faili. Selleks on hea teada vastavaid meetodeid:

	C/C++	C++	Python
päisfail	`csdtio`	`fstream`	-
Failimuutuja	`FILE`	`ifstream` `ofstream`	
Faili avamine lugemiseks fn - failinimi	`fopen(fn, "rd")`	`open(fn)`	`open(fn, "r")`
Faili avamine kirjutamiseks	`fopen(fn, "wt")`		`open(fn, "w")`
Failist lugemine (nt kaks täisarvu x1 ja x2 failist f)	`fscanf(f, "%d %d", &x1, &x2)`	`f >> x1 >> x2`	

Rea lugemine s – rida n – rea pikkus f - failimuutuja	`fgets(s, n, f)`	`getline(f, s)`	`f.readline()`
Faili kirjutamine (nt kaks täisarvu x1 ja x2 faili f)	`fprintf(f, "%d %d", x1, x2 )`	`f <<x1<<" "<<x2;`	`f.write()`
Faili sulgemine	`fclose(f)`	`f.close()`	`f.close()`

Javas toimub failidest lugemine sarnaselt standardsisendist lugemisele: appi võetakse `Scanner` või `BufferedReader`. Failivoo avab `FileReader`. Näide nende kasutusest failide lugemisel on kohe järgmise punkti all.

Faili kirjutamisel kasutatakse klassi `PrintWriter` ja failivoo jaoks `FileWriter`. `PrintWriter` klassil on meetodid `println` ja `print`:

```
PrintWriter pw = new PrintWriter(new FileWriter("arvud.txt"));
pw.println("kala");
```

Failidest lugemise ja kirjutamise põhjalikud näited on Eesti Informaatikaolümpiaadide lehel: http://eio.ut.ee/KKK/Failid

1.5.3 Sisendi töötlus

Sisendi lugemise ja käsitlemise erinevate võimaluste illustreerimiseks vaatame järgmist ülesannet:

Failis `arvud.txt` on miljon täisarvu. Leida ja väljastada nende summa.

Nagu eelnevalt juttu oli, on Javas sisendi lugemiseks kaks levinumat viisi: `Scanneri` ja `BufferedReaderi` kasutamine.

Esimene lahendus:

```
BufferedReader br = new BufferedReader(new FileReader("arvud.txt"));
long a = 0;
StringTokenizer st = new StringTokenizer(br.readLine());
for (int i = 0; i < 1000000; i++) {
     a += Integer.parseInt(st.nextToken());
}
System.out.println(a);
```

Teine lahendus:

```
Scanner scanner = new Scanner(new FileReader("arvud.txt"));
long a = 0;
for (int i = 0; i < 1000000; i++) {
     a += scanner.nextInt();
}
System.out.println(a);
```

Esimesel lahendusel läks summa leidmiseks 0,3 ja teisel 1,2 sekundit.

`Scanner` parsib sisendit ja sellel on mitmed programmeerija jaoks mugavad meetodid, nagu näiteks `nextInt()`. `BufferedReader` on lihtsalt üks suur puhver ja sinna loetud infot tuleb programmeerijal

ise edasi parsida. Nagu näha, on suurte sisendandmete korral nn toores andmete lugemine ja nende hilisem töötlemine oluliselt kiirem.

C++ puhul on meil valik, kas kasutada C või C++ stiilis sisendi lugemist – neid on illustreeritud järgmistes näidetes:

C++ stiilis:

```
file.open("arvud.txt");
sum = 0;
for (int i = 0; i < n; i++)
{
    int a;
    file >> a;
    sum += a;
}
file.close();
cout << sum << endl;
```

C stiilis:

```
sum = 0;
FILE* fl = fopen("arvud.txt", "r");
for (int i = 0; i < n; i++)
{
      int a;
      fscanf(fl, "%d", &a);
      sum += a;
}
printf("%d\n", sum);
```

C++ stiil on natuke mugavam kirjutada, kuid töötab aeglasemalt. Antud test võttis minu arvutis vastavalt 0,7 ja 0,1 sekundit.

1.5.4 Väljund

Väljundi osas tuleb tavaliselt lahendada kaks küsimust: korrektsus ja efektiivsus.

Korrektsuse seisukohalt on ülesannetel vahel eritingimused, mis nõuavad väljundi kirjutamist konkreetsete reeglite alusel formaadituna – näiteks peab reaalarvuline vastus olema antud täpselt kolme kohaga pärast koma. Selliste reeglite realiseerimiseks on andmete väljastamise funktsioonidel parameetrid, mille kaudu saab vastavaid reegleid spetsifitseerida. Järgnevas tabelis on selleks mõned näited:

	Reaalarv täpselt 3 kohaga pärast koma	**Kuupäev ja kellaaeg *aeg* kujul Päev.Kuu.Aasta Tund:Minut:Sekund**
C	`fprintf(vf, "%0.3lfn", arv);`	`fprintf(vf, "%d.%m.%y %H:%M:%S", aeg);`
C++	`vf << fixed << setprecision(3) << arv`	`vf << put_time(&aeg, "%d.%m.%Y %H:%M:%S");`
Java	`vf.println(String.format("%.3f", arv))`	`DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss"); vf.println(dateFormat.format(aeg));`
Python (vana)	`vf.write("%.3fn" %arv)`	
Python (uus)	`vf.write('{:.3f}'.format(arv))`	`vf.write('{:%d.%m.%Y %H:%M:%S}'.format(aeg))`

Efektiivsuse küsimus tuleb mängu juhul, kui väljastada tuleb palju infot ja see tuleb kirjutada faili. Kui kirjutame väljundit näiteks märgikaupa, ei kirjuta väljundteek seda üldjuhul kohe faili, vaid hoiab vastavas puhvris. Kui puhver saab täis või kui failipide suletakse, kirjutatakse kogunenud info kõik korraga faili.

Efektiivsema testimise ja silumise huvides on vahel kasulik puhvri tühjendamist eraldi välja kutsuda, kuid tuleb arvestada, et see aeglustab programmi tööd.

Puhvri tühjendamiseks on eri keeltes järgmised meetodid:

	C/C++	C++	Java	Python
Standard-väljundisse	`fflush(stdout)`	`cout << endl`	`System.out.flush()`	`sys.stdout.flush()`
Faili: f – faili-muutuja	`fflush(f)`	`f << endl` `f.flush()`	`f.flush()`	`f.flush()`

1.5.5 Jooksvalt töötamine

Suure sisendi/väljundi puhul võtavad andmete sisestamine, töötlemine ja väljastamine suurusjärguliselt sama palju aega. Sellisel juhul tuleks uurida, kas meil on võimalik andmed lihtsalt „jooksvalt läbi vaadata“, ilma et peaks kogu sisendit mällu lugema.

Vaatame selle kontseptsiooni illustreerimiseks järgmist ülesannet:

Failis on miljon kuuetähelist stringi. Leida ja väljastada neist leksikograafiliselt esimene.

Naiivne meetod selleks on teha nii:

```
string* ar = new string[n];
for (int i = 0; i < n; i++)      {
      file >> ar[i];
}
sort(ar, ar + n);
cout << ar[0] << endl;
```

Tegelikult pole meil aga muidugi vaja kõiki stringe mällu lugeda, vaid võime senist parimat vahetulemust lihtsat jooksvalt meelest pidada:

```
string vas = "zzzzzz";
for (int i = 0; i < n; i++)
{
      string vv;
      file >> vv;
      if (vas.compare(vv) > 0)
      {
            vas = vv;
      }
}
cout << vas << endl;
```

1.6 PROGRAMMEERIMISKEELTE VÕRDLUS

Programmeerimisega esmakordselt kokku puutujad küsivad sageli, milline programmeerimiskeel on parim. Vastus oleneb loomulikult konkreetsetest asjaoludest, erinevate programmeerimiskeelte eripäradest on kirjutatud sadu raamatuid. Kuna selle raamatu näited on C++, Java ja Pythoni baasil, siis toon välja, mis on nende peamised jooned. Esmased erinevused on mõistagi süntaktilised, kuid neil keeltel on ka väga erinev disainifilosoofia ja eesmärgid.

Vastus küsimusele „millist keelt ma peaksin õppima?“ on aga „erinevaid!“. Ühe programmeerimiskeele oskaja on nagu töömees, kellel on vaid haamer, samas kui erinevad keeled täidavad su tööriistakasti ning avavad võimalusi ülesannete lahendamiseks mitmel viisil. Olen ise professionaalselt kasutanud vähemalt kümmet erinevat programmeerimiskeelt ning kokku puutunud veel palju enamatega. Mõistagi kujunevad mõned, milles vilumus on suurem, kuid hea on teada, et saad ka teistsugustes olukordades hakkama.

Konkreetselt C++, Java ja Pythoni osas on mugav mõelda, et neile vastavad konkreetsed märksõnad, mis on keele disainimisel olnud esmase prioriteediga: C++ puhul kiirus, Javal ohutus ning Pythonil lihtsus. Samuti on igal vaadeldaval keelel erinev saamislugu: C++ tuli akadeemiast, Java ärimaailmast ning Python leiutati hobi korras.

1.6.1 C++

Taani arvutiteadlane Bjarne Stroustrup uuris 1980. aastate alguses oma doktoritöö jaoks objekt-orienteeritud programmeerimist, kasutades peamiselt Simula 67 keelt. Simula oli praktiliseks kasutamiseks liiga aeglane, mistõttu Stroustrup hakkas lisama objekt-orienteerituse printsiipe C keelele. Aastal 1983 valmiski C++ keele esimene variant. Paljud teised tänapäeval laialt levinud keeled nagu Java ja C# on sellest tugevalt mõjutatud.

Objekt-orienteeritud paradigma aitab kirjutada programme, mis on kergemini hallatavad ja täiendatavad. Võistlusprogrammeerimise kontekstis, kus programmid valmivad mõne tunniga ja hiljem neid tavaliselt ei täiendata, on need omadused vähemtähtsad. Samas on C++ põhiprintsiipideks olnud:

- Keelekonstruktsioonide lihtne ja otsene ülekantavus riistvaras toimuvale.
- Ilma lisakuluta abstraktsioonimehhanismid, mida Bjarne Stroustrup on ka kokku võtnud kui „What you don’t use, you don’t pay for“ (mida sa ei kasuta, selle eest sa ei maksa) printsiipi.

C++ jõudlusele optimiseeritust illustreerib järgmine näide. Olgu meil andmestruktuur, mis hoiab endas tasandil asuva punkti koordinaate:

```
class Point { int x; int y; };
```

ja objekt sellest klassist:

```
Point xy {2,5};
```

C++ kirjutab objekti väljad lihtsalt järjest mällu:



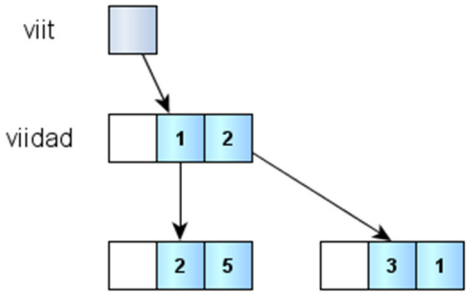
Kui meil on massiiv punktidest

segment a[ ] = { {2,5}, {3,1} };

siis kirjutatakse ka need lihtsalt järjest mällu.



Võrdlusena hoitakse paljudes „puhastes objekt-orienteeritud keeltes“ kõiki kasutajaobjekte viitadena, mis tähendab, et saame järgmise struktuuri:



Need põhimõtted võimaldavad C++ abil kirjutada kiiremat ja efektiivsemat koodi kui enamikus teistes kaasaegsetes keeltes. Objektide kompaktselt hoidmine sobib hästi näiteks protsessorites kasutatavate andmete vahemällu laadimise strateegiatega, kus vahemällu lisatakse mitte lihtsalt üksikute mälupesade sisu, vaid terve lähestikku asuv piirkond.

Vanemal C keelel on samad head jõudluse omadused, kuid C++ lisab olulise abina *standard library*, kus on realiseeritud palju kasulikke andmestruktuure (nendest lähemalt kolmandas peatükis). Nendel põhjustel on C++ programmeerimisvõistlustel tavaliselt populaarseim valik.

1.6.2 Java

Java loodi 1990. aastate alguses Sun Microsystemsis, eesmärgiga kasutada seda mitmesugustes programmeeritavates seadmetes nagu kaabel-TV boxid. Java peamine autor oli Kanada arvutiteadlane James Gosling ja selle avalik väljalase toimus aastal 1995. Java esmane disainiprintsiip oli WORA (*Write Once, Run Anywhere*), mis pidi võimaldama sama Java koodi ohutult käivitada suvalisel platvormil. Ohutus tähendab antud kontekstis, et Java programm ei tohi segada teiste programmide tööd ega töökeskkonda mingil viisil ära rikkuda.

Sama koodi mitmel platvormil kasutatavuse ehk porditavuse eesmärk saavutatakse järgmiselt:

1. Java kompilaator transleerib lähtekoodi baitkoodiks.
2. Erinevatel platvormidel on realiseeritud erinevad variandid Java virtuaalmasinast (JVM), mis võivad baitkoodi kas interpreteerida või siis platvormispetsiifiliseks masinkoodiks ümber kompileerida (märksõna JIT – *just-in-time compilation*).

Java teeb ära mitmed asjad, mille C++ jätab programmeerija hooleks, näiteks mäluhalduse. Võistlusprogrammeerimise seisukohalt üks tähtsamaid mugavusi on automaatne massiivi piiride kontroll. C/C++ keeled ignoreerivad massiivi piiridest väljapoole lugemist ja kirjutamist (operatsioonisüsteem võib küll vastu näppe anda), aga Java annab selle peale ise vea. Selline kontrollimine tuleneb otseselt ohutu porditavuse eesmärgist, kuid vähendab programmi kiirust.

Ajalooliselt on Javal olnud veel mitmeid põhjusi, mis selle oluliselt aeglasemaks muutsid:

- Varased JVMid võimaldasid ainult baitkoodi interpreteerimist.
- Aeglane käivitusaeg, mis tuleneb vajadusest laadida hulk erinevaid teeke.
- Mäluhaldus, kus objektide alt vabaks jäänud mälu automaatselt taas kättesaadavaks märgitakse (*garbage collection)*. Varasemates JVM versioonides võttis see märkimisväärselt aega.



Neile probleemidele on leiutatud mitmeid lahendusi ning tänapäeva Java on palju kiirem kui vana aja Java. Sellegipoolest saan ma vahel vastava särgi selga panna ning Java-programmeerijate linnaosades neid narrimas käia.

Kaasajal sõltub kiirusevahe konkreetsest programmist. Rusikareeglina on Eesti olümpiaadidel arvestatud, et Java programmid võiks käia 1,5-2 korda aeglasemalt kui sama ülesannet lahendavad C++ programmid. Rahvusvahelistel võistlustel on ajalimiidid erinevates keeltes kirjutatud programmidele üldjuhul samad.

1.6.3 Python

1989. aasta jõulude ajal oli Hollandi programmeerija Guido van Rossumi kontor suletud ja ta ei saanud tööd teha. Seetõttu istus van Rossum nädal aega kodus ja leiutas Pythoni, esialgu lihtsa skriptimiskeelena. Esmakordselt jõudis Python avalikkuse ette 1991. aastal.

Pythoni disain rõhutab koodi loetavuse tähtsust ja üldist arusaadavuse lihtsust. See on ka üks põhjusi, miks Python on tänapäeval populaarne keel programmeerimise õpetamiseks.

Käesoleva peatüki esimese näiteülesande saab Pythonis kirjutada lihtsalt nii:

```
print("Tere, maailm")
```

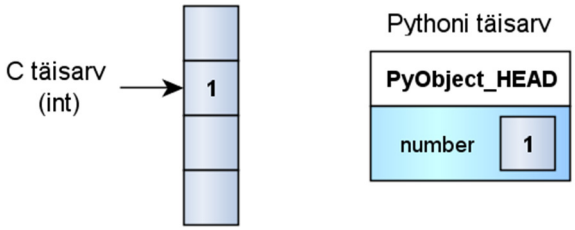
Pole vaja mingeid loogelisi sulge, spetsiaalseid funktsiooninimesid ega viiteid välistele teekidele.

Võrreldes C++ ja Javaga on Python tavaliselt oluliselt aeglasem. Eesti olümpiaadidel on Pythonis kirjutatud lahenduste jaoks üldjuhul eraldi ajalimiit, mis on enamasti 10 korda kõrgem kompileeritud keelte (C/C++, Java, C# jt) omast.

Pythoni aeglusel on mitmed põhjused:

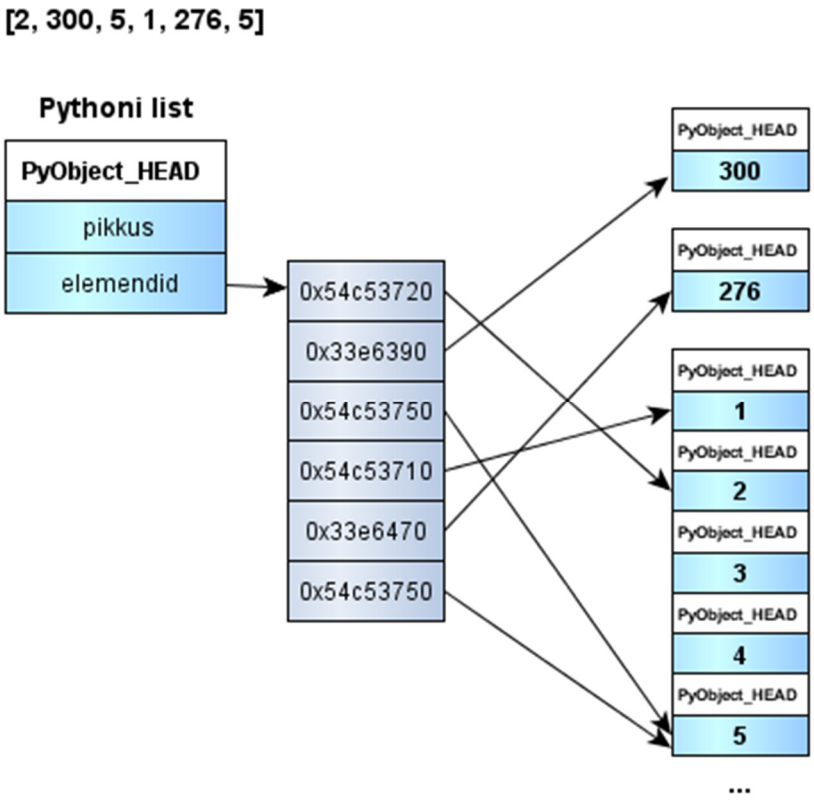
Esiteks on Python on **interpreteeritav** keel, mis tähendab, et koodi ei kompileerita, kompilaatorid aga optimeerivad koodi väga suure määral.

Teiseks on Pythonil **dünaamiline tüübisüsteem**, mis tähendab, et Pythoni interpretaator ei tea ette, mis on muutuja tüüp. Iga Pythoni muutuja on objekt. Kui asutakse sooritama mingit tehet objektidega, siis kõigepealt vaatab interpretaator järele, mis on nende objektide tüüp ning alles seejärel saab sooritada vastava tehte. Samuti on vaja vastuse jaoks luua uus objekt. See tähendab palju rohkem samme, kui staatilise tüübisüsteemiga keeltes vaja teha on.



Kolmandaks muudab Pythoni andmemudel **ligipääsu mälule** ebaefektiivseks. Kui näiteks C++ massiiv on viit järjestikusele mälupuhvrile, siis Pythoni listis on igale listi liikmele vastava objekti aadressid, objektid ise võivad olla siin-seal laiali. Sellise ülesehituse korral on arusaadav, et massiivi/listi kõikide elementide läbikäimine võtab rohkem aega, kuna tehakse palju rohkem tööd.

Järgmises näites on lihtne täisarvude massiiv, mille C++ kirjutaks järjestikuste baitidena mällu, aga Pythoni puhul luuakse viidad aadressidele, kus nendele täisarvudele vastavad objektid mälus asuvad. Väiksematele arvudele vastavad objektid luuakse sageli korraga, nii et nad ise paiknevad mälus järjest.



Pane tähele mäluaadresse: väikestel arvudel on need järjest, suuremad paiknevad teises kohas ja on loodud esinemise järjekorras ning omavahel lähestikku. Samuti tasub tähele panna, et mõlemad ’5’-d on tegelikult sama objekt.

Lisaks aeglusele on Pythoni suureks probleemiks Pythoni erinevate versioonide (Python 2 ja Python 3) omavaheline mitteühilduvus. Mõlema versiooni jaoks kirjutatakse aktiivselt uusi programme ja teeke, kuid sageli tuleb teha valik, kas kasutada üht või teist versiooni või siis kulutada ekstra aega mõlema toetamiseks.

1.7 TESTIMINE

Tarkvaratööstuses on tavaline, et ühed inimesed kirjutavad programme ja teised testivad neid. Selle osas, kas taoline jaotus on praktiline, on palju hääli kähedaks vaieldud ja mitmesuguseid organisatsioonilisi eksperimente läbi viidud, kus testimise eest pannakse vastutama erinevad inimesed. Viimasel ajal levib järjest enam lähenemine, kus tarkvaraarendajad testivad ise oma koodi, kasutades selleks üldjuhul automaatseid teste.

Programmeerimisvõistlustel toimub testimine samuti automaatselt: ülesande koostaja on ette valmistanud hulga testsisendeid, mille eesmärgiks on kontrollida, kas programm saab kõigi võimalike andmetega hakkama.

Et võistlusel edukalt hakkama saada, on kasulik mõelda nagu testide koostaja ja ette näha, mis on tavalised asjad, mida kontrollitakse. Kui see oskus on kord tekkinud, on sellest ka palju kasu „tavaelu“ programmeerimises, kus päris kasutajad on sageli haruldaselt leidlikud sinu programmi jaoks ootamatute olukordade loomisel.

Järgnevalt mõned olulised testide kategooriad:

1.7.1 Piirjuhud

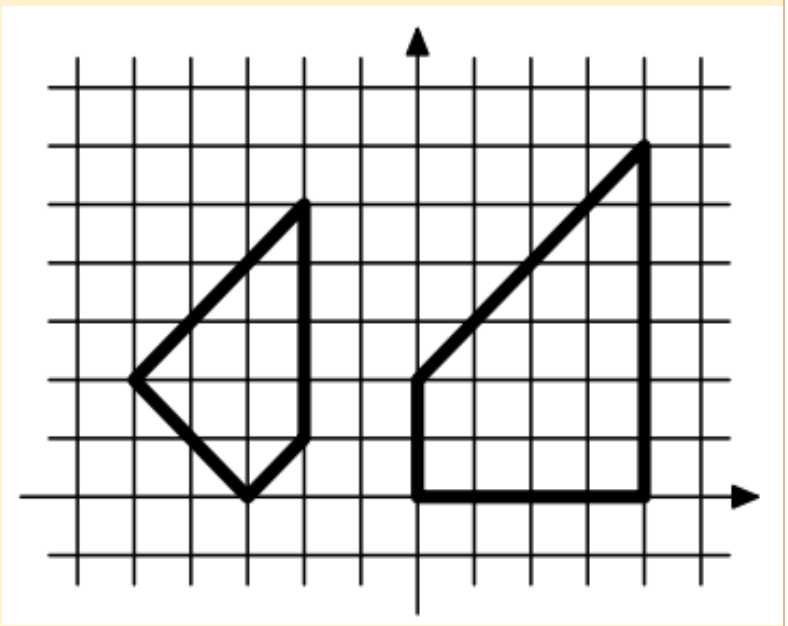
Enamikul ülesannetel esinevad **piirjuhud**, mida testidega üldjuhul kontrollitakse. Kui ülesandeks on leida tee linnast A linna B, tuleb kindlasti vaadelda juhtu, kus A ja B on sama linn. Kui vaja on leida arvu tegurid, tuleb käsitleda ka arvu 1. Kui juttu on ruudustikul toimuvast lauamängust, peab vaatama ka 1x1 „ruudustikku“. Selline mõtteviis on väga kasulik harjumus ka tavapäraselt tarkvaratööstuses töötades.

Võistlusülesannetel on sisendi osas üldjuhul ette antud mingid piirid – kindlasti tuleb oma programmi testida kõigi nende piirväärtustega. Kui mingitele parameetritele pole piire antud, tuleb ise igasuguseid ekstreemsusi proovida.

1.7.2 Õigsuse kontroll

Vaatame näitena ülesannet Eesti 2016. aasta informaatikaolümpiaadi eelvoorult. Tegu on päris raske ülesandega, aga see illustreerib hästi erinevaid kavalusi, mida saab testide loomisel ära kasutada.

Juku õpib koolis hulknurkade sarnasust ja saab teada, et hulknurgad on sarnased, kui nende vastavate nurkade suurused on võrdsed ja vastavate külgede pikkused võrdelised. Sarnased hulknurgad võivad olla omavahel pööratud, peegeldatud ja nihutatud. Sarnaste hulknurkade vastavate külgede pikkuste jagatist nimetatakse nende sarnasusteguriks.
Kodutööna saab ta hulga hulknurki, mille sarnasustegureid on vaja määrata. Jukul on fanaatiline matemaatikaõpetaja, kes andis tööna väga paljude nurkadega hulknurki. Aita Juku hädast välja.
Sisend. Tekstifaili esimesel real on hulknurga tippude arv `N` (`3 ≤ N ≤ 200 000`).
Faili teisel real on `2·N` täisarvu lõigust -10^9 kuni 10^9: esimese hulknurga tippude x- ja y-koordinaadid. Kolmandal real on samuti `2·N` arvu: teise hulknurga tippude koordinaadid. Tipud võivad olla antud nii päripäeva kui vastupäeva järjekorras. Antud punktid moodustavad alati hulknurga, milles pole ühtelangevaid punkte, sirgnurki, ega endaga lõikumisi.
Väljund. Kui hulknurgad on sarnased, siis kirjutada väljundi esimesele reale täpselt üks reaalarv, mis näitab, mitu korda on esimene hulknurk suurem kui teine (kui esimene hulknurk on väiksem, on ka vastus väiksem kui 1). Teisele reale kirjutada täisarv, mis näitab, mitmes teise hulknurga tipp vastab esimese hulknurga esimesele tipule (mõlema hulknurga tipud on nummerdatud alates ühest nende failis esitamise järjekorras).
Kui hulknurgad ei ole sarnased, kirjutada väljundi ainsale reale −1.
Hindamine. Pooltes testides on teada, et iga hulknurga küljed on kõik erineva pikkusega.



NÄIDE (vastavad hulknurgad on ka joonisel):

```
4
0 0 4 0 4 6 0 2
-2 5 -2 1 -3 0 -5 2
```

Vastus:

```
1.414213
3
```

Vastuse teine osa on 3, sest teise hulknurga kolmas punkt (−3; 0) vastab esimese hulknurga esimesele punktile (0; 0).

Antud juhul tuleb juba ülesande tekstist välja rida asjaolusid, mida kontrollida:

Hulknurgast võib saada teise sarnase hulknurga nihutamise, pööramise, suurendamise/vähendamise või peegeldamise kaudu. Seega peab proovima teste, kus me kõiki neid teisendusi kasutame. Võib arvata, et ülesande koostaja on loonud testid, kus on sees need teisendused nii ühekaupa kui ka kombineeritult.

Järgmiseks tuleb proovida kõigi nende teisenduste ekstreemseid väärtusi. Kas lahendus saab hakkama, kui hulknurka nihutada lubatud väärtuste ühest piirist teise? Kas vastus on õige, kui hulknurka suurendada näiteks miljard korda?

Pärast hulknurga enda teisendusi tuleb tähele panna, et tipud võivad olla antud nii päripäeva kui vastupäeva järjestuses. Olenevalt kasutatud algoritmist võib see anda punktide sobitamiseks neli erinevat võimalust: peegeldatud ja peegeldamata juhtum ning päripäeva ja vastupäeva punktid. Ka need juhud tuleb konkreetselt läbi testida.

Veel tuleb keskenduda ülesande geomeetrilisele osale: hulknurgad on sarnased parajasti siis, kui nende vastavad nurgad on võrdsed ja küljepikkused võrdelised. Kui üks neist asjaoludest tähelepanuta jääb, on ka kontroll vigane. Seega tuleb testida ka olukorda, kus hulknurkade nurgad küll klapivad, aga küljepikkused mitte ning vastupidi.

Kokkuvõttes tuleb ülesandepüstitus hoolikalt läbi lugeda, selle teksti analüüsida ja oma programmi testida kõigi kirjeldatud ja ka ridade vahelt selguvate asjaolude osas – taas üks harjumus, mis ka tavaprogrammeerimises väga abiks on.

1.7.3 Algoritmi jõudluse kontroll

Võistlusülesannete tõenäoliselt kõige huvitavam osa on efektiivse algoritmi leidmine. Ülesande lahendamiseks on sageli võimalik kasutada erinevaid algoritme, mille eest saab erineval määral punkte, lihtsama ja aeglasema eest vähem, kiirema eest rohkem. Enamik võistlejaid saab tavaliselt mingi osa punktidest ning parimad saavad täispunktid. Kogu ülesandekomplekt proovitakse koostada nõnda, et summaarsed maksimumpunktid saaksid mõned üksikud võistlejad.

Vaadeldaval hulknurkade ülesandel saab kasutada kolme erinevat lahendust: esimene on aeglasem, teine on kiire, aga ei lahenda ära kõiki juhtumeid, ning kolmas on kiire ja täielik.

Kõigi puhul võime kõigepealt leida mõlema kujundi küljepikkused ja nurgasuurused. See osa on tavaline geomeetria ja trigonomeetria, mis nõuab natuke tehnilist tööd, aga ei midagi erakordset.

Edasi jõuame huvitavama osani: esimese kavalusena saab mõlemad hulknurgad muuta ühesuuruseks, mis lihtsustab edasist tööd. Selleks mõõdame lihtsalt kummagi hulknurga ümbermõõdu. Ümbermõõtude suhe annab otsitava sarnasusteguri ning nüüd saame ühe hulknurga kõik küljepikkused korrutada selle teguriga.

Nüüd on meil olemas kaks järjendit küljepikkustest ja nurgasuurustest ja tuleb leida, kas neid on võimalik omavahel üksühesesse vastavusse viia. Kõige lihtsam meetod klapitamiseks on järgmine:

1. Proovime teise hulknurga iga tipu puhul, kas see võiks esimese hulknurga esimese tipuga sobida (nii, et nurkade suurused on võrdsed ja küljepikkused õige suhtega).
2. Sobitamiseks liigume vaadeldavatest tippudest edasi ja proovime järjest kõik tipud läbi, et leida, kas ka need sobivad.

Siin tekib probleem, et teisel sammul võime saada palju sobivusi järjest ja siis äkki mõne ebasobivuse. Testid on ka spetsiaalselt nõnda koostatud, et selliseid olukordi tekitada, kasutades näiteks kõrvaloleval joonisel toodud põhimõtet.



Halvimal juhul võime seega käia läbi kõik tipud ja iga tipu jaoks peame omakorda kontrollima kõiki tippe, kokku N^2 kontrolli.

Võistlusülesannetel on ajapiiriks tavaliselt 1 sekund testi kohta, mis võimaldab tänapäeva arvutitel sooritada suurusjärgus 100 miljonit lihtsat kontrollimist. Antud juhul võimaldab see lahendada teste, kus N on kuni 10 000. Tavaliselt saab võistlusel selliste lahenduste eest mingi osa punkte, aga mitte maksimumi.

Siit edasi võime tähele panna tekstis toodud tingimust, mis lubab pooled punktid testide eest, kus kõik küljed on erinevate pikkustega. Selliste juhtude jaoks võib välja mõelda järgmise algoritmi:

1. Leida mõlema hulknurga pikim külg.
2. Minna sellest pikimast küljest edasi ja kontrollida, kas ka kõik teised sobivad.

Mõlemat sammu saab läbi viia N kontrolliga. Kuna N ≤200 000, on aega piisavalt, seega on nüüd pooled punktid käes.

Antud ülesande üldjuhu lahendamiseks parim algoritm on aga hoopis tekstitöötluse valdkonnast. Kui mõelda küljepikkuse ja nurgasuuruse kombinatsioonist kui tähemärgist, siis taandub ülesanne kontrollile, kas kaks stringi on saadud üksteise esimese ja tagumise jupi äravahetamise teel. Näiteks stringid CABAABBA ja BBACABAA on selles mõttes ekvivalentsed.

Nüüd on esialgne geomeetriaülesanne muundunud tekstiülesandeks, aga see on vaid hea, sest selles valdkonnas on olemas mitmed standardalgoritmid. Tekstiga tegelemisel on tekstist otsingusõna või ka pikema fraasi leidmine klassikaline probleem. Üks hea algoritm selleks on Knuth-Morris-Pratti ehk KMP algoritm (https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm). KMP algoritmist tuleb pikemalt juttu 11. peatükis, aga praegu piisab teadmisest, et see võimaldab vajalikku tekstisobitamist mitte rohkem kui N kontrollimisega, mis on meie vajaduste jaoks piisav.

Näidisprogramm, mis hulknurkade ülesande KMP algoritmi abil ära lahendab, on raamatu lisas.

1.8 SILUMINE

Mida teha, kui programm ei tee seda, mida vaja, vaid hoopis midagi muud? Koos keerulisuse kasvuga kasvab ka selle „muu tegemise“ tõenäosus. Üks võimalus on panna programm väljastama ohtralt infot selle kohta, mida ta parajasti teeb ja mis on erinevate muutujate väärtused, aga see on küllaltki ebaefektiivne.

Tõhusam on kasutada spetsiaalset **silumisprogrammi** ehk **silurit**, mis võimaldab koodi samm-sammult läbi käia ja vaadata, mis seal sees täpselt toimub. Sellist tegevust nimetatakse **silumiseks** (*debugging* ehk meeleolukalt „putukate ärastamine“). Põhimõtteliselt võib siluris avada iga programmi, kuid kompileeritud koodi puhul saame seda üldjuhul näha ja läbi käia ainult assembleri tasemel. Kui meil on olemas ka lähtekood, on kompileeritud koodi võimalik sellega vastavusse viia, kasutades **silumissümboleid** (*debug symbols*). Silumissümbolid luuakse kompileerimise ajal ja nad võivad olla kas kompileeritud failis või eraldi failis. Kui programmi jaoks on olemas nii lähtekood kui sümbolid, saab seda siluda lähtekoodi tasemel.

1.8.1 Arenduskeskkonnad

Minimaalselt on programmeerimiseks vaja tekstiredaktorit ja vastava keele kompilaatorit või interpretaatorit.

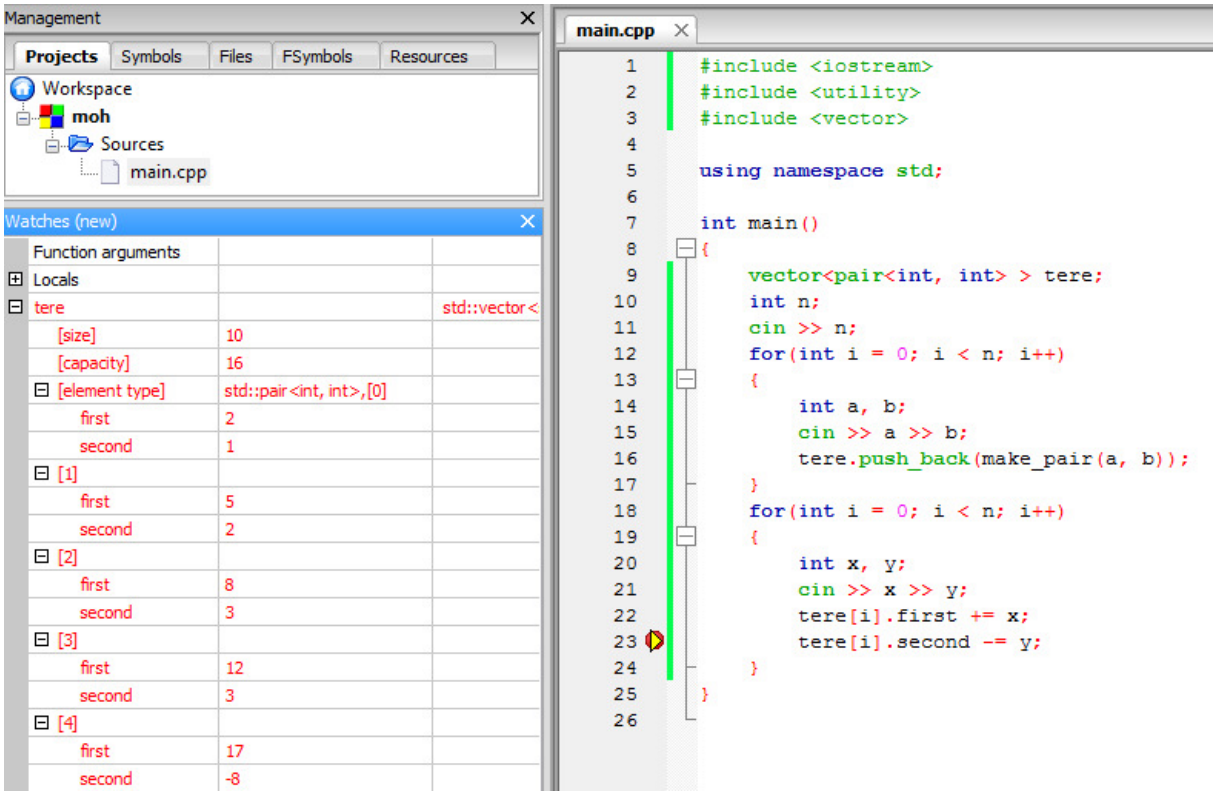
Enamik programmeerijaid kasutab tänapäeval aga pigem mõnd **integreeritud arenduskeskkonda** (*integrated development environment, IDE*), kus on koos lähtekoodi redigeerimise, kompileerimise ja

silumise võimalused. Tavaliselt on kompilaator ja silur siiski eraldi programmid, kuid arenduskeskkond peidab nad oma kasutajaliidese taha.

Erinevaid arenduskeskkondi ja silureid on palju ning nendest pikalt rääkimine ei mahuks siia raamatusse. Võistluse seisukohalt on oluline teada, kuidas toimub sinu eelistatud arenduskeskkonnas või siluris **katkepunktide** (*breakpoint*) seadmine. Kui programmi töö jõuab katkepunktini, siis silur peatab programmi töö ja sul on võimalik vaadata erinevate muutujate väärtusi.

Eriti kasulik võimalus on **dünaamiliste** katkepunktide seadmine, mille puhu programmi töö peatub vaid juhul, kui täidetud on mõni konkreetne tingimus. Kui uurime probleemi, mis juhtub ainult tsükli viiesaja kuuekümnendal sammul, siis on üsna tüütu seda enne 559 korda läbi käia. Selle asemel saab seada dünaamilise katkepunkti, mis aktiveerub parajasti siis, kui tsüklimuutuja väärtus on 560.

Siin on pilt arenduskeskkonnast Code::Blocks. Real 23 on seatud katkepunkt ning vasakul on näha muutuja „tere“ sisu:



Teine oluline teadmine on klahvikombinatsioonid programmi rida-realt läbikäimiseks, sealhulgas meetodid funktsioonidesse sisenemiseks ja väljumiseks, konkreetse reani joosta laskmiseks jne. Võistlusel on aeg sageli napp ning siluri kiire ja efektiivne kasutamine aitab võistlusel seda kokku hoida.

1.8.2 Aja mõõtmine

Käesoleva raamatu läbivaks teemaks on kiirete programmide kirjutamine. Et aga päriselt aru saada, mis kui kaua aega võtab, on vaja seda kiirust mõõta. Õigemini on vaja täpselt mõõta tegevuste sooritamiseks kulunud aega, milleks on erinevates programmeerimiskeeltes taas omad vahendid.

Tavaliselt pole eesmärk ka mitte lihtsalt ajamõõtmine, vaid külaltki suure täpsusega ajamõõtmine, mille jaoks on spetsiaalsed funktsioonid.

Näide C++ ajamõõtmisest:

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
	int c = 0;
	double start = clock();
	for (int i = 0; i < 100000; i++) {
		c *= 20;  // simuleerime keerulisi arvutusi
		c -= 1;
	}
	double end = clock();
	cout << "Kulus" << (end - start) / CLOCKS_PER_SEC << "sekundit" << endl;
}
```

Näide Javas:

```
long start = System.nanoTime();
/*tee midagi*/
long stopp = System.nanoTime();

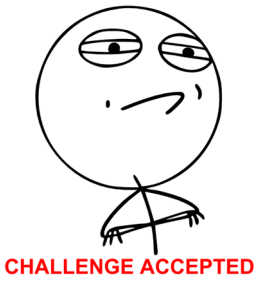
System.out.println(stopp - start);
```

Näide Pythonis (alates 3.3):

```
import time

start = time.perf_counter()
# tee midagi
stopp = time.perf_counter()
print( stopp – start)
```

1.9 KONTROLLÜLESANDED



Esimese peatüki kontrollülesanded on mõeldud üldise programmeerimistehnika harjutamiseks ega vaja sügavamaid teadmisi algoritmidest, vastates raskuselt ligikaudu Eesti olümpiaadide lihtsaimatele ülesannetele. Kui need ülesanded liiga kerged tunduvad, pole vaja muretseda, järgmistes peatükkides läheb asi raskemaks.

1.9.1 Järelmaks

Andres ostis uue arvuti, miks maksis H (`100 ≤ H ≤ 10000`) eurot. Ostmisel pakuti talle 0% intressiga järelmaksu ja muidugi võttis ta hea pakkumise vastu. Järelmaks kestab N kuud (`1 ≤ N ≤ 120`). Iga kuu lõpus peab Andres tasuma $\frac{1}{N}$ arvuti ostuhinnast. Samas on teada, et arvutite hind langeb üsna kiiresti, kaotades igas kuus P protsenti (P on täisarv 1-20) oma hetkeväärtusest. Seega võib arvuti väärtus mingil perioodil olla madalam kui maksta jäänud jääk.
Leia, mitmendal kuul ületab arvuti väärtus järelmaksujäägi.
Sisendi ainsal real on 3 täisarvu: H, N ja P. Väljundisse kirjutada, mitmenda kuu lõpuks ületab arvuti väärtus järelmaksujäägi.

NÄIDE 1:
`1000 10 5`
Vastus: 1 (arvuti jääkväärtus on alati suurem kui järelmaksujääk, seega tekib nõutav olukord juba esimese kuu lõpuks)



NÄIDE 2:
`1000 20 5`
Vastus: 2 (Esimese kuu lõpus on väärtused võrdsed, seega peame ootama teise kuuni)

NÄIDE 2:
`2000 20 10`
Vastus: 17

Märkus: nagu paljusid võistlusülesandeid, saab ka seda ülesannet lahendada mitmel viisil. Võib kogu protsessi simuleerida või siis tuletada üldvalemi vastuse koheseks leidmiseks.

1.9.2 Kell

Seinal on tunniosuti ja minutiosutiga kell, kus minutiosuti näitab parajasti täpset minutit. Leida tunniosuti ja minutiosuti vaheline nurk.
Sisendi ainsal real on antud kellaaeg. Väljundisse kirjutada osutitevahelise nurga suurus kraadides ning väljastada see täpsusega täpselt 3 kohta pärast koma. Nurk tuleb väljastada vahemikus 0-180 (s.t 3:00 ja 21:00 on 90 kraadi, mitte 270).

NÄIDE 1:
`9:00`
Vastus: 90



NÄIDE 2:
`8:10`
Vastus: 175

NÄIDE 3:
`23:59`
Vastus: 5.5

1.9.3 Tigu

H meetri sügavuse kaevu põhjas on tigu, kes proovib sealt välja roomata. Iga päev suudab tigu roomata U meetrit ülespoole ja igal öösel libiseb ta D meetrit allapoole. Kuna kaevus pole toitu, väsib tigu igal järgmisel päeval järjest rohkem ära. Teisel päeval suudab ta ronida T protsenti vähem kui esimesel päeval, kolmandal päeval 2T protsenti vähem kui esimesel päeval jne.
Sisendis on antud neli täisarvu H ($1 \le H \le 1000$), U ($1 \le U \le 10$), D ($1 \le D \le 10$) ja T ($1 \le T \le 100$). Väljundisse kirjutada kaks täisarvu. Kui tigu pääseb kaevust välja, kirjutada väljundisse 1 ja selle päeva järjenumber, millal tigu välja saab. Kui tigu vajub kaevu põhja tagasi, kirjutada väljundisse 0 ja põhjavajumise päeva järjenumber.

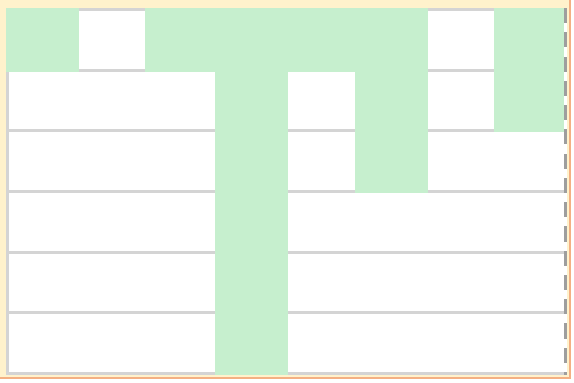


NÄIDE 1:
```
6 3 1 10
```
Vastus:
```
1 3
```
NÄIDE 2:
```
50 5 3 14
```
Vastus:
```
0 7
```
NÄIDE 3:
```
50 6 4 1
```
Vastus:
```
0 68
```
Märkus: ole tähelepanelik erinevate piirjuhtudega!

1.9.4 3D printer

3D printimise põhimõtteks on materjalikihtide lisamine etteantud kohtadesse. Meil on lihtne 3D printer, mis oskab printida plastikust kujundeid nii, et iga uus kiht on eelmise peal, aga ei tohi ulatuda eelmisest üle. Nii saame luua näiteks joonisel kujutatud kujundi. Printer kirjutab kihte ükshaaval, alt üles, lülitades plastiku lisamist kas sisse või välja. Antud on kujundi andmed, leida, mitu korda tuleb plastiku lisamist sisse lülitada.
Sisendi esimesel real on täisarv N ($1 \le N \le 10000$), mis tähistab prinditava figuuri laiust.
Järgmisel N real on igaühel üks täisarv (samuti lõigust 1-10000), mis tähistab figuuri lõplikku kõrgust vastavas lõigus.
Väljundisse kirjutada täpselt üks täisarv: plastikujoa sisselülitamise kordade arv.

NÄIDE (vastab joonisele):
```
8
5
6
5
0
5
3
6
4
```
Vastus: 14



1.9.5 Mõttemeister

Mõttemeister on kahe mängija mäng, kus kasutatakse kuut eri värvi mängunuppe. Esimene mängijatest (Kodeerija) mõtleb välja neljast nupust koosneva koodi ja teine (Arvaja) proovib seda ära arvata. Mäng koosneb voorudest, kus Arvaja paigutab lauale neli nuppu vastavat sellele, milline tema arvates kood võiks olla, ning Kodeerija annab talle selle põhjal hinde, pannes lauale kuni neli punast või valget nuppu. Punaste hindenuppude arv tähistab õiget värvi ja õigel kohal asuvate nuppude arvu, valgete hindenuppude arv tähistab õiget värvi, kuid valel kohal asuvate nuppude arvu.
Olgu värvideks punane (P), kollane (K), roheline (R), valge (V), sinine (S) ja oranž (O).
Sisend koosneb kahest reast: esimesel real Kodeerija loodud kood ja teisel real Arvaja esitatud pakkumine.
Väljundisse kirjutada kaks arvu: punaste ja valgete hindenuppude arv.



NÄIDE 1:

```
P K K R
K P K V
```

Vastus: `1 2`

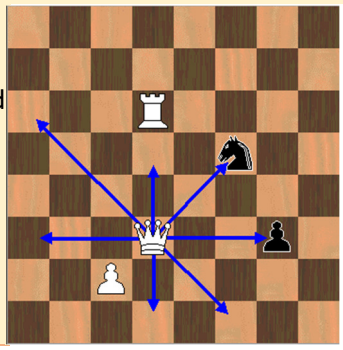
NÄIDE 2:

```
O R V K
P P P P
```

Vastus: `0 0`

1.9.6 Male

Males saab lipp käia horisontaalsetes, vertikaalsetes ja diagonaalsetes suundades kuitahes pikalt kuni malelaua servani või järgmise nupuni. Olgu malelaual valge lipp ja hulk musti malendeid. Leida, mitu mustadest nuppudest on valge lipu tules.
Sisendi esimese real on valge lipu asukoht, teisel real tühikutega eraldatud mustade malendite asukohad. Väljundisse kirjutada üks täisarv – mustade nuppude arv, mida valge lipp saab lüüa.



NÄIDE:

```
a3
a1 a6 a8 f4 c4 b4
```

Vastus: 3 (a1, a6 ja b4)

1.9.7 Riigihanked

Kui riigiasutustel on vaja midagi osta, korraldavad nad selleks hanke, kuhu pannakse kirja nõuded, mida pakkujad peavad täitma. Kuna hanke korraldajad peavad hiljem läbima auditi, kus kontrollitakse, et nad kedagi ebaausalt ei eelistanud, ei tohi inimesed pakkumisi subjektiivselt hinnata, vaid on vaja automaatset süsteemi, mis kontrollib pakkumiste vastavust nõuetele.
Hanke võidab pakkumine, mis vastab suurimale arvule nõuetele. Kui kaks pakkumist vastavad samale arvule nõuetele, tuleb valida odavam nendest.
Sisendi esimesel real on kaks täisarvu: nõuete arv `N` (`0 ≤ N ≤ 100`) ning pakkumiste arv `P` (`1 ≤ P ≤ 100`). Järgmisel `N` real on toodud nõuded, üks string igal eraldi real. Edasi tuleb pakkumiste info. Pakkumise info algab pakkumise nimega, järgmisel real on kaks täisarvu: pakkumise hind ja pakutava kauba või teenuse omaduste arv `A` (`0 ≤ A ≤ 100`).
Programm peab võrdlema pakutava kauba omadusi nõuetega ja leidma parima pakkumise.
Väljundisse kirjutada võitva pakkumise nimi.

NÄIDE:

```
6 4
mootor
pidurid
navigatsiooniseade
katuseluuk
nahkistmed
talverehvid
Ford
10000 3
mootor
nahkistmed
suunatuled
Lada
1000 5
talverehvid
pukseerimisköis
pidurid
mootor
suunatuled
Honda
20000 5
mootor
pidurid
talverehvid
suunatuled
navigatsiooniseade
Antoni romula spetsiaal
5000 4
talverehvid
navigatsiooniseade
nahkistmed
katuseluuk
```



Vastus: `Antoni romula spetsiaal` (odavaim pakkumine, mis vastab neljale tingimusele).

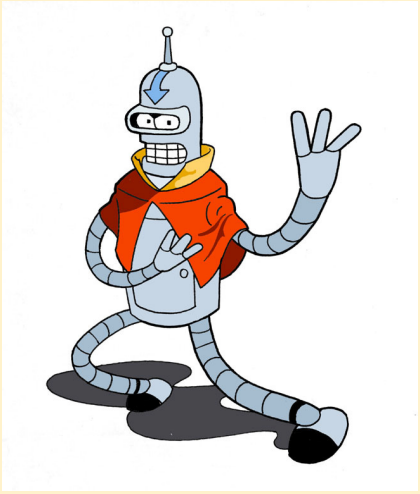
1.9.8 Bender

Aastal 2996 valmistatakse Mehhikos, Tijuanas, *Fábrica Robótica De La Madre's* (Ema Robotivabrikus) robot Bender. Benderi ülesandeks on painutada sirget terastraati vastavalt etteantud intruktsioonile. Benderil on `N` (`1 ≤ N ≤ 1000000`) sentimeetri pikkune traat, mida ta peab iga sentimeetri tagant painutama. Bender alustab painutamist traadi lõpust, `N-1`. sentimeetril ja lõpetab 1. sentimeetril. Kõik painutused on 90-kraadised ja painutatud lõik saab olla esialgse traadi suunaga võrreldes suunatud kas üles, alla, vasakule või paremale.

Leida, millises suunas on pärast kõiki painutusi traadi lõpp.

Sisendi esimesel real on arv `N`. Järgmisel `N-1` real on igaühel üks tähemärk: kas `U` (üles), `D` (alla), `R` (paremale) või `L` (vasakule).

Väljastada täpset üks märk - üks kuuest suunast, milles võib pärast kõiki painutusi olla traadi lõpp: lisaks juba mainitud `U`, `D`, `R` ja `L` suundadele ka `F` (otse, samasuunaline traadi algusega) ja `B` (tagasi, vastassuunaline traadi algusega).



NÄIDE:

```
4
R
U
R
```

Vastus:

```
B
```

1.9.9 Interpretaator

Üks ebaviisakalt nimetatud programmeerimiskeel (https://en.wikipedia.org/wiki/Brainfuck) on disainitud võimalikult minimalistlikuks. Käesolevas ülesandes uurime selle veelgi vähendatud varianti. Programm kasutab viita, mis võib näidata erinevatele mälupesadele ja nendes olevaid väärtusi muuta. Töö alguses näitab viit esimesele mälupesale. Programmi tekstis saab kasutada nelja operaatorit:

> suurendab viida aadressi ühe võrra.
< vähendab viida aadressi ühe võrra.
+ suurendab viidatavat väärtust ühe võrra.
- vähendab viidatavat väärtust ühe võrra.

Masinal, mis nendest käskudest koosnevat programmi täidab, on 100 pesast koosnev mälu ning üks viit sellele mälule. Mälu on ringikujuline: kui me läheme ühelt poolt üle serva, satume tagasi teise serva. Mälupesades hoitav maksimaalne väärtus on 255 ja need väärtused on samuti „ringikujulised", s.t 255 + 1 = 0 ja 0 – 1 = 255.

Sisendi ainsal real on antud Brainfuckis kirjutatud programm. Väljundisse kirjutada 100 tühikutega eraldatud täisarvu: mälupesade väärtused pärast programmi lõpuni jooksmist.

NÄIDE:

```
>---<++<><>++++++++++++++++++++++++++>>>+++<+++<<>++++++++++++++++++++++++++>>>+++
<+++<><>++++++++++++++++++++++++++>>>+++
```

Vastus:

```
31 253 32 3 29 3 0 0 0
```

1.9.10 Pangatellerid

Pangas on N tellerit, kes kliente teenindavad. Kui klient tuleb panka, läheb ta esimese telleri juurde. Kui esimene teller on hõivatud, läheb ta teise juurde jne. Kui kõik tellerid on hõivatud, siis inimesed ootavad järjekorras ja lähevad selle telleri juurde, kes parajasti vabaneb.
Tellerid püüavad muuhulgas klientidele mitmesuguseid teenuseid müüa, aga neil on selleks erinev võimekus. Müüdud teenuste väärtus on telleri müügioskuse ja kliendi pangakonto seisu korrutis jagatud tuhandega.
Meil on pangapäeva kohta teada, millal kliendid saabusid ning lahkusid, samuti iga kliendi pangakonto seis ja iga telleri müügivõimekus. Leida, kui palju pangateenuseid sel päeval müüdi.
Sisendi esimesel real on kaks täisarvu: tellerite arv N ($1 \leq N \leq 100$) ja klientide arv M ($1 \leq M \leq 10000$). Järgmistel N real on igaühel üks täisarv, mis tähistab vastava telleri müügivõimekust (täisarv 1-100). Järgmistel M real on klientide pangakontode seisud (täisarvud 1 - 10000). Lõpuks tuleb 2M rida, igaühel üks täisarv, mis tähistavad klientide saabumise ja lahkumise järjekorda. Positiivsed arvud tähistavad vastava numbriga kliendi saabumist ja negatiivsed arvud tema lahkumist.



NÄIDE:

```
2 4
5
2
100
500
1000
2000
3
1
2
4
-1
-3
-2
-4
```

Vastus: 16,2

Näite selgitus:
- Klient 3 läheb esimese telleri juurde – tulu 5*1000/1000=5.
- Klient 1 läheb teise telleri juurde – tulu 2*100/1000=0,2.
- Klient 2 saabub ja ootab.
- Klient 4 saabub ja ootab
- Klient 1 lahkub, klient 2 läheb teise telleri juurde – tulu 2*500/1000=1.
- Klient 3 lahkub, klient 4 läheb esimese telleri juurde – tulu 5*2000/1000=10.

1.10 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk1 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Tere.cpp, Tere.java, Tere.py	„Tere, maailm“
Kolmnurk1.cpp, Kolmnurk1.java, Kolmnurk1.py	Kolmnurga ülesanne, esimene lahendus
Kolmnurk2.cpp, Kolmnurk2.java, Kolmnurk2.py	Kolmnurga ülesanne, esimene lahendus
Asendaa.cpp, Asendaa.java, Asendaa.py	Stringide liitmise ülesanne
Summa.cpp, Summa.java, Summa.java, Summa.py	Arvude liitmine failist.
VahimSona.cpp, VahimSona1.java, VahimSona2.java, VahimSona.py	Vähima sõna leidmine failist.
Hulknurk1.cpp, Hulknurk1.java, Hulknurk1.py	Hulknurga ülesanne – otsene lahendus
Hulknurk2.cpp, Hulknurk2.java, Hulknurk2.py	Hulknurga ülesanne – erineva pikkusega küljed
Hulknurk3.cpp, Hulknurk3.java, Hulknurk3.py	Hulknurga ülesanne – Knuth-Morris-Pratti algoritmi kasutav lahendus

2 LÄBIVAATUS- JA OTSINGUALGORITMID

Arvutiprogrammid modelleerivad enamasti mingit aspekti tegelikust elust. Elu on aga suur ja keeruline, mistõttu ka programmid muutuvad kergesti suurteks ja keerulisteks. Nende aastakümnete vältel, mil tarkvaratehnika on distsipliinina eksisteerinud, on väga suur osa sellealasest uurimistööst olnud pühendatud kahe probleemi lahendamisele:

- Kuidas jagada liiga suured ülesanded väiksemateks osadeks, nii et neid oleks kergem kirjutada, mõista ja hallata?
- Kuidas vältida sama ülesande mitmekordset lahendamist, seda nii koodi kirjutamise kui ka käivitamise mõttes?

Käesolevas peatükis vaatleme tehnikaid keeruliste ülesannete lihtsustamiseks.

2.1 ALAMPROGRAMMID

Programmid koosnevad instruktsioonidest arvutile: tee seda, siis toda. Kuna ülesanded on keerulised, muutuvad ka programmid pikaks ja raskesti arusaadavaks, mistõttu need jagatakse sageli alamprogrammideks.



Alamprogrammis on hulk instruktsioone pakendatud konkreetse alguse ja lõpuga kogumiks ning seda kogumit saab edaspidi kasutada nagu üksikoperatsiooni. Üldjuhul täidab iga alamprogramm mingit piiritletud alamülesannet. Kui see alamülesanne on aga omakorda liiga suur, saab osa tööst eraldada uueks alamülesandeks. Kokkuvõttes kutsub põhiprogramm välja alamprogramme, need omakorda teisi alamprogramme jne. Selline lähenemine võimaldab igast üksikust alamülesandest paremini aru saada ja selle lahendust vajadusel muuta ning hallata. Vahel on kasulik alamprogrammid üles ehitada nii, et ta lahendab ära mingi osa ülesandest ning kutsub siis uuesti välja iseennast, et lahendada järgmine osa. Sellist iseenda väljakutsumist nimetatakse **rekursiooniks.**

Programmi jagamine alamosadeks on kasulik mitmel põhjusel:

- Keerulise ülesande jagamine väiksemateks osadeks.
- Mitmekordselt sama koodi kirjutamise vältimine (taaskasutus).
- Programmi ülevaatlikkuse parandamine, mis aitab vigu leida.

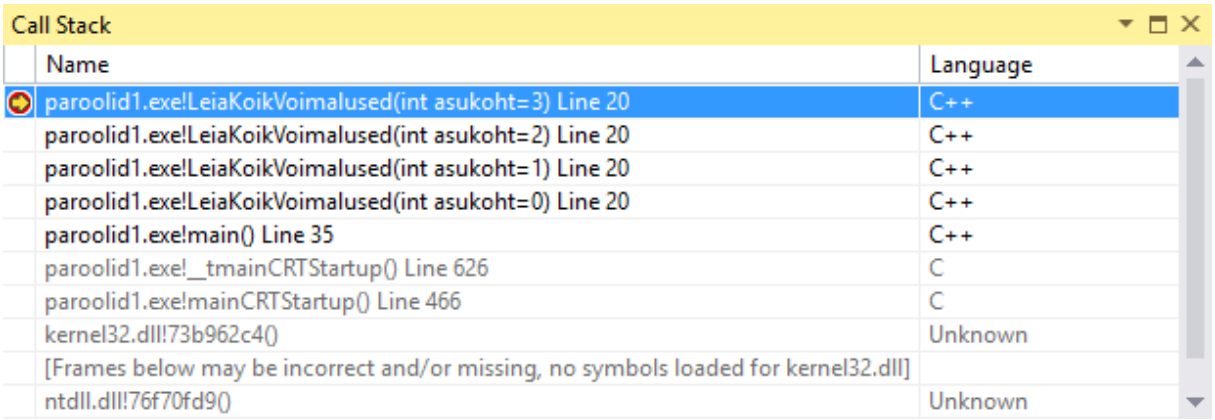
Sõltuvalt programmeerimiskeelest võidakse alamprogramme nimetada funktsioonideks, protseduurideks või meetoditeks. C ja sellest põlvnevad keeled kasutavad tavaliselt terminit „funktsioon“, mis viitab sarnasusele matemaatilise funktsiooniga. Nagu matemaatilisel funktsioonilgi, võivad programmi funktsioonil olla parameetrid ning üldjuhul on tal üks konkreetne tagastatav väärtus. Siin raamatus kasutatakse edaspidi samuti funktsiooni terminit.

2.1.1 Pinumälu

Programmi käivitamisel pannakse käima „main“ funktsioon, mis omakorda võib välja kutsuda teisi funktsioone, need kolmandaid jne kasvõi sadu kordi. Kui funktsioon töö lõpetab, annab protsessor järje jälle eelmisele funktsioonile ja niimoodi tullakse mööda ahelat tagasi.

Kui programmi töö mingil hetkel siluris peatada, siis ongi programmi hetkeseis kujutatav funktsioonide jadana, mis on üksteist ridamisi välja kutsunud. See põhimõte pole keelespetsiifiline, vaid peegeldab seda, kuidas operatsioonisüsteemid ja protsessorid tavapäraselt töötavad.

Järgmisel pildil on siluris peatatud funktsiooni `LeiaKoikVoimalused` rekursiivne väljakutse. Kõigepealt kutsub `main` funktsioon välja `LeiaKoikVoimalused` parameetriga 0, seejärel kutsub funktsioon ennast välja parameetriga 1, siis 2 ja peatamise hetkel on väljakutse parameetriga 3.



Enne „päris“ programmi käivitamist on näha käitusteegi (antud juhul C Runtime ehk CRT) funktsioonid ning enne neid operatsioonisüsteemi funktsioonid.

2.1.2 Funktsiooni kohalikud andmed

Peamine funktsiooni iseloomustav omadus on tema **skoop**. See tähendab, et muutujad, mis on defineeritud mingis funktsioonis, on nähtavad ainult selle funktsiooni sees, mitte teistes alamprogrammides.

Funktsiooni sees defineeritud kohalike muutujate ja funktsiooni parameetrite väärtusi hoitakse konkreetses mälupiirkonnas, mida nimetatakse **pinukaadriks** (*stack frame*). Kui funktsioon kutsub välja teise funktsiooni, reserveeritakse eelmise kaadri kõrvalt uus pinukaader, milles hoitakse teise funktsiooni parameetrite ja muutujate jaoks vajalikku mälu. Kaadri suurus oleneb kohalike muutujate arvust ja tüübist. Kui funktsioon lõpetab töö, siis vastav mäluplokk vabastatakse. Kõigi pinukaadrite mälu kokku nimetatakse **pinuks** (*stack*).

Pinu töötab alati LIFO (*last in, first out* – viimasena sisse, esimesena välja) põhimõttel – viimati reserveeritud plokk on alati esimene, mis vabastatakse. See teeb arvepidamise lihtsaks, kõik väärtused kirjutatakse järjest mällu ning protsessor hoiab meeles **pinuviita** (*stack pointer*) aadressile, kus käesoleva funktsiooni andmed on. Pinukaadrid paneb üldjuhul paika kompilaator, mis arvestab, kui palju ruumi erinevate muutujate jaoks vaja on.

2.1.3 Pinu ületäitumine

Pinu suurus on fikseeritud ning kui see täis saab, tekib **ületäitumine** (*stack overflow*). Tavaliselt juhtub ületäitumine siis, kui funktsioon või funktsioonid jäävad iseennast või teineteist välja kutsuma. Vahel võib ületäitumine tekkida ka muudel juhtudel – tavaliselt siis, kui funktsioonis on defineeritud palju kohalikke muutujaid.

Ületäitumise illustreerimiseks kasutame järgmist iseennast väljakutsuvat funktsiooni:

```
int test(int n) {
    if (n == 0) return 0;
    cout << n << endl;
    return 1 + test(n - 1);
}
```

Minu arvutis annab programm vea umbes 250000 väljakutse järel. Kui parameetrite arvu suurendada, kahaneb ka võimalike väljakutsete sügavus. Järgmine programm teeb ainult 40000 väljakutset.

```
int test(int n, int a, int b, int c) {
    if (n == 0) return 0;
    cout << n << endl;
    return 1 + test(a - 1, b - 1, c - 1, n - 1);
}
```

Kui aga kasutada kohalike muutujate jaoks arvestataval hulgal mälu, võib limiit kiiresti kahaneda. Järgmine programm saab vea juba 250 väljakutse järel.

```
int test(int n, int a, int b, int c) {
    int d[1000]; // mälu sellele massiivile võetakse kõik pinust
    int sum = 0;
    for (int i = 0; i < 1000; i++)
    {
        // teeme andmetega midagi, muidu võib kompilaator muutuja eemaldada
        sum += d[i];
    }
    if (n == 0) return 0;
    cout << n << " " << sum << endl;
    return 1 + test(a - 1, b - 1, c - 1, n - 1);
}
```

Pythonis on võimalik ületäitumise ennetamiseks määrata rekursiooni maksimaalne sügavus. Vaikimisi määratud sügavust saab teada kasutades funktsiooni `sys.getrecursionlimit()` ning vajadusel muuta funktsiooniga `sys.setrecursionlimit(N)`, kus `N` on soovitav maksimaalne rekursiooni sügavus. Kui seatud piir programmi töö ajal ületatakse, tekib vastav käitusaegne viga (*runtime error*).

2.1.4 Pinu kasutamine protsessoris

Pinukaadrid on toetatud protsessori tasemel, protsessoritel on spetsiaalsed käsud funktsioonide väljakutsumiseks ja nendest väljumiseks. Inteli protsessorites on nendeks käsud `call` ja `ret`. `call` käsk salvestab käsuloenduri sisu pinusse ja „hüppab“ väljakutsutava funktsiooni esimese käsu juurde. `ret` loeb salvestatud käsuaadressi pinust tagasi ja funktsiooni töö saab jätkuda sealt, kus see enne väljakutset pooleli jäi.

Väljakutsuv funktsioon peab kutsutavale funktsioonile edasi andma ka parameetrid: need kirjutatakse enne väljakutset pinusse.

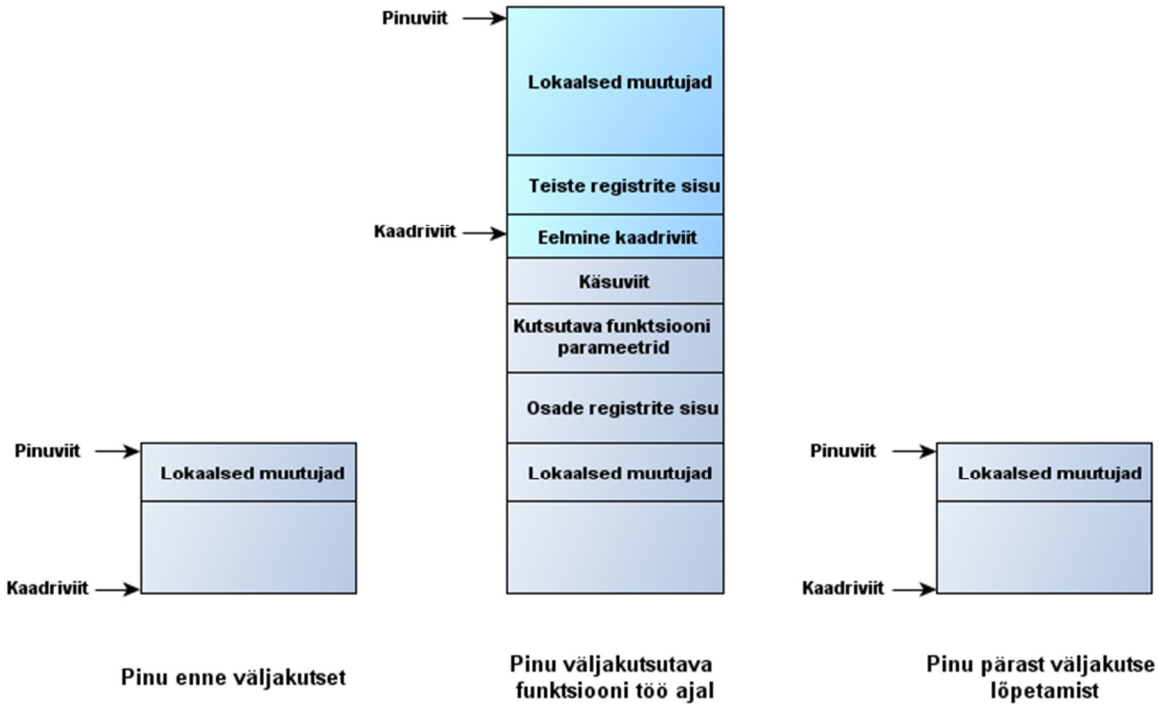
Ka väljakutustaval funktsioonil endal võivad olla muutujad. Needki hoitakse pinus. Siin tekib aga probleem, et kui kutsutud funktsioon asub välja kutsuma omakorda järgmist funktsiooni jne, siis läheb keeruliseks arvet pidada, kus üks pinukaader hakkab ja teine lõpeb. Selleks on kasutusel spetsiaalne register **kaadriviida** (*frame pointer*) jaoks. Kaadriviit on aadress pinus, millest viimane kaader algab. Kui uus funktsioon välja kutsutakse, muutub mõistagi ka kaadriviit, seepärast salvestab

väljakutsutud funktsioon väljakutsuja kaadriviida väärtuse pinusse. Uus kaadriviit seatakse vana viida salvestamiseks kasutatud aadressile.

Kuna erinevad funktsioonid kasutavad samu registreid, salvestatakse pinusse ka muude registrite sisu. Registrite salvestamine on ära jagatud kutsuja funktsiooni ja kutsutava funktsiooni vahel ning neid hoitakse salvestava funktsiooni kaadris.

Tagastatav väärtus hoitakse tavaliselt registris (x86 protsessoritel enamasti eax), kust väljakutsuv funktsioon selle enne registrite taastamist kätte saab ning vajalikku kohta liigutab.

Pinukaadrite struktuur on kokkuvõttes selline:



Kui funktsioon töö lõpetab, siis vabastatakse kohalike muutujate all olev mälu ja taastatakse osade registrite sisu, seejärel taastatakse kaadriviit ja käsuviit ning vabastatakse funktsiooni parameetrid. Seejärel salvestatakse vajadusel funktsiooni tagastusväärtus registrist mujale ning taastatakse ülejäänud registrite sisu. Vabastamine, nagu enne mainitud, tähendab vaid pinuviida nihutamist.

Täpsem näide sellest, millist koodi funktsiooni väljakutsete jaoks genereeritakse, on punktis 2.2.4 Sabarekursioon.

2.1.5 Kuhimälu

Pinumälus hoitakse neid muutujaid, mille vajadusest kompilaator kohe aru saab ja neile mälu ära reserveerib. Pinust üle jäävat mäluosa nimetatakse **kuhimäluks** (*heap memory*) ning sealt saavad programmid vajalike objektide jaoks veel täiendavalt mälu juurde võtta.

Pinu ja kuhja erinevusi illustreerib järgmine tabel:

	Pinumälu	Kuhimälu
Suurus	Pannakse paika programmi (täpsemalt iga uue lõime) töö alguses ja hiljem ei muutu. Tavaliselt mõni megabait, mida saab muuta vastava kompilaatorivõtmega.	Piiratud operatsioonisüsteemi seadetega. Kui programm vajab rohkem mälu, küsitakse seda operatsioonisüsteemilt juurde.
Objektidele mälu andmine ja tagastamine	Mälu võetakse funktsiooni kohalikele muutujatele korraga ning vabastatakse automaatselt.	Mälu võetakse igale objektile eraldi ja see tuleb pärast kasutamist tagastada.
Kiirus	Mälu andmine ja tagastamine käib pinuviida väärtuse suurendamise ja vähendamise kaudu, mis teeb operatsiooni kiireks. Samuti on pinu sageli lihtsam protsessori vahemällu laadida.	Erinevate objektide mälu võib olla siin-seal laiali, mis võib muuta kuhjas olevate objektidega töötamise aeglasemaks.

2.1.6 Funktsiooni parameetrid

Sellest, kuidas väljakutsuv funktsioon väljakutsutavale parameetrid pinus edasi annab, oli punktis 2.1.4 põhjalikult juttu. Millised bitid ja baidid aga täpselt väljakutsutavale funktsioonile parameetrina edasi antakse, sõltub nii konkreetsest programmeerimiskeelest kui ka sellest, mis tüüpi muutuja parameetriks on.

Vaatame järgmist koodi:

```
void vaheta(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a = 5;
    int b = 8;

    vaheta(a, b);
}
```

Ilmselt soovis programmeerija kirjutada funktsiooni, mis vahetab etteantud muutujate väärtused. Vahetus tõepoolest tehakse, aga seda vaid funktsiooni `vaheta` skoobis. `main` funktsiooni tagasi pöördudes on muutujate a ja b väärtused samad, mis enne väljakutset, sest parameetriteks kopeeriti muutujate a ja b väärtused. Sellist kutset, milles antakse edasi parameetrite väärtused, nimetatakse **väärtuskutseks** (*call by value*). Väärtuskutses ei saa kutsutav funktsioon muuta kutsuva funktsiooni muutujate väärtusi.

C++ keeles on võimalik edasi anda ka parameetrite aadressid. Järgmine funktsioon tõepoolest teeb, mis soovitud:

```
void vaheta(int& a, int& b)
{
    int temp = a;
```

```
    a = b;
    b = temp;
}
```

Sellist kutset, milles kutsuv funktsioon annab kutsutavale funktsioonile parameetritena üleantavate muutujate aadressid, nimetatakse **aadresskutseks** (*call by reference*). Aadresskutse puhul saab kutsutav funktsioon muuta kutsuva funktsiooni muutujate väärtusi.

2.1.7 Objektide edastamine parameetritena

Kui kasutada funktsiooni parameetrina objekte, käituvad keeled mõnevõrra erinevalt. Järgnevalt mõned näited, milles kasutatakse klassi `Isik`, millel on väljad `Nimi` ja `Vanus` ning konstruktor, mis seab ka nime. C++ on klass defineeritud järgmiselt:

```
class Isik
{
    public:
        char Nimi[6];
        int Vanus;
        Isik(char* s);
};

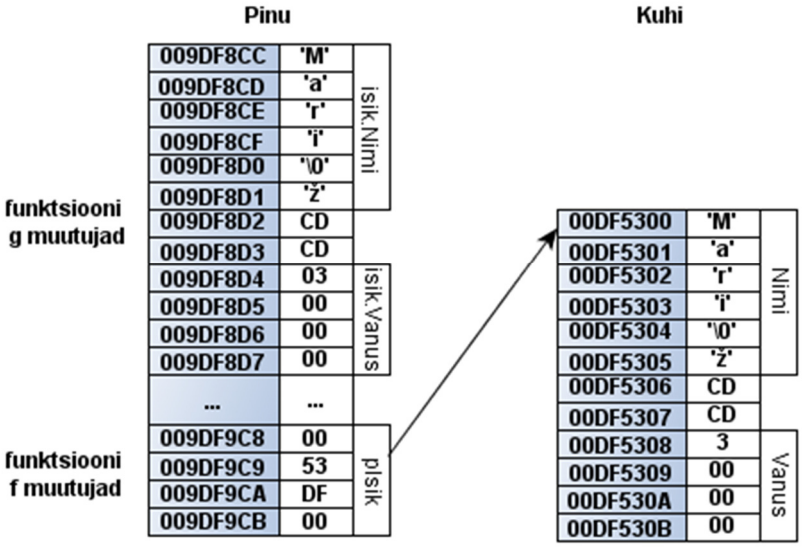
Isik::Isik(char* s)
{
    strcpy(Nimi, s);
}
```

Objekt parameetrina

Kui C++ keeles anda alamfunktsioonile parameetrina objekt, siis funktsiooni väljakutsel kogu objekt kopeeritakse pinusse. Nii juhtub järgmises koodis:

```
void f()
{
    Isik* pIsik;
    pIsik = new Isik("Mari");
    pIsik->Vanus = 3;
    g(*pIsik);
    cout << pIsik->Vanus;
}

void g(Isik isik) {
    isik.Vanus = 9;
}
```



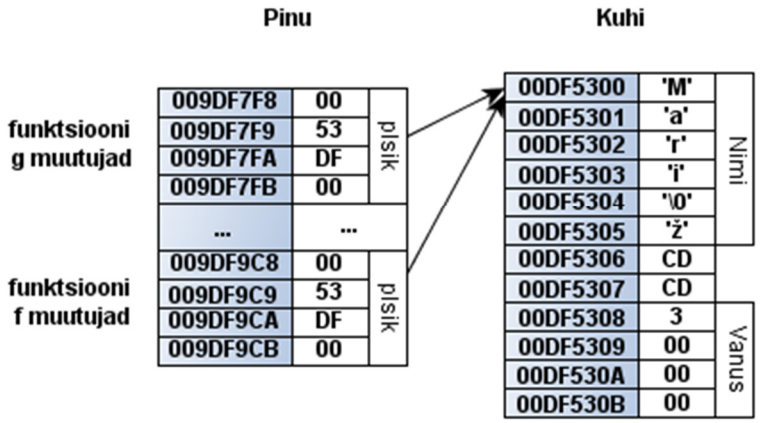
Funktsioonis f on kohaliku muutujana viit kuhimälus loodud Isik-tüüpi objektile. Funktsiooni g väljakutsel luuakse tervest objektist koopia pinus ning g muudab pärast seda ainult oma lokaalset koopiat. Funktsiooni f juurde tagasi pöördudes g koopia „kaob“. Seetõttu väljastab antud programmijupp vanuse „3“ ning funktsioon g teeb tühja tööd.

Aadressiviit parameetrina

Enamasti antakse terve objekti asemel alamfunktsioonile parameetrina objekti asemel edasi hoopis viit objektile ehk muutuja, mille väärtuseks on objekti aadress. Sellele vastab järgmine kood:

```
void f()
{
    Isik* pIsik;
    pIsik = new Isik("Mari");
    pIsik -> Vanus = 3;
    g(pIsik);
    cout << pIsik->Vanus
}

void g(Isik* pIsik) {
    pIsik->Vanus = 9;
}
```



Nüüd viitavad mõlema funktsiooni muutuja i samale objektile ning funktsioon g muudab vanuse ning programm väljastab seekord 9.

C++ puhul on oluline meeles pidada, et parameetritega ümberkäimine on üsna vaba ja seega tuleb hoolega tähele panna, mida edasi antakse ja mida muudetakse. Mõned andmetüübid, nagu näiteks massiivid, on ise viidad ja käituvad ka parameetritena vastavalt.

Python

Pythonis on kõik muutujad tegelikult viidad objektidele. Objekte üldjuhul ei kopeerita. Pythonit kasutades on oluline meeles pidada, millist tüüpi objektid on muudetavad ja millised mitte. Kui väljakutsutav funktsioon muudab objekti, mis ei ole muudetav, näiteks täisarvu või stringi, siis tekitatakse uus objekt ning väljakutsutava funktsiooni muutuja hakkab viitama sellele uuele objektile, kuid väljakutsuva funktsiooni muutuja viitab endiselt vanale objektile. Kui aga objekt on muudetav, näiteks list või objekt, siis muudetakse sama objekti ja muutus on nähtav ka väljakutsuvale funktsioonile. Pythoni mäluhaldusest tuleb rohkem juttu järgmises peatükis.

Java

Javas lihttüüpide väärtused kopeeritakse, keerulisemad andmetüübid antakse edasi viitadena.

Järgnevalt üks näide Javas:

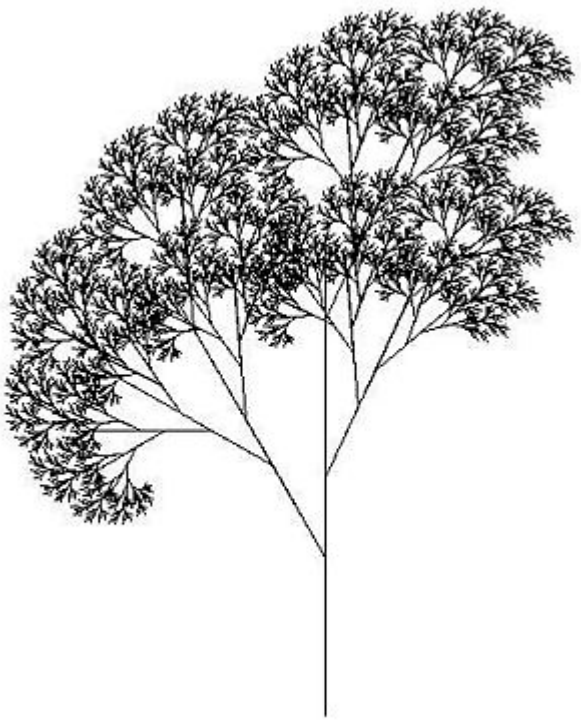
```
public void f() {
    Isik i = new Isik("Mari");
    g(i);
    System.out.println(i);
}
public void g(Isik i) {
    i.nimi = "Maria";
}
```

Antud näites muudetakse nimi kenasti ära ning väljastatakse nimi „Maria“.

2.2 REKURSIOON

Rekursiooni mõiste illustreerimiseks võib kasutada kõrvalolevat joonist: joonistatud puu koosneb alamosadest, mis on terviku sarnased. Puu tervikuna on keeruline, aga iga üksik haru on lihtne: üks kriips ja mõned hargnemised. Puu joonistamiseks on üks võimalus kirjutada kood, mis joonistab puud tervikuna, aga see oleks üksjagu pikk ja keeruline. Teine võimalus on aga märgata neid **korduvaid alamosi** ning kirjutada palju lihtsam alamprogramm, mis joonistab üksikut haru. Kogu puu joonistamiseks peab see alamprogramm tegema kahte asja:

1. joonistama etteantud pikkusega ja etteantud nurga all ühe kriipsu,
2. **kutsuma mitu korda välja iseennast**, aga vähendatud pikkusega ning natuke muudetud nurkadega.



Programmeerimiskeeles Logo rekursiivselt joonistatud puu

Tehnilise definitsioonina on rekursioon alamprogrammi sees sama alamprogrammi väljakutse, enamasti väiksemal andmemahul. Muidugi tekib siin kohe probleem: kui alamprogramm kutsub ennast uuesti ja uuesti välja, võib see ju lõputult kesta? Lõputu töö vältimiseks peab eksisteerima **baasjuhtum** ehk **baas**, kust edasi pole vaja rekursiivseid väljakutseid teha. Antud puujoonistamise puhul võib baasjuhtumiks olla oksa pikkus – kui see on piisavalt väike, pole veel väiksemaid osi vaja joonistada (eeltoodud algoritmist jääb alles ainult punkt 1).

Rekursiooni saab liigitada mitut moodi. Üks võimalus on eristada ühekordset ehk hargnemisteta rekursiooni ja mitmekordset ehk hargnemistega rekursiooni. Hargnemisteta rekursiooni korral kutsub funktsioon end enda sees välja vaid ühe korra, hargnemistega funktsiooni korral aga mitu korda.

Hargnemistega rekursiooni ongi lihtne ette kujutada puuna ja seda nimetataksegi ka puurekursiooniks. Puu juur on töö alguspunkt, lehed on baasjuhtumid ning hargnemiskohad rekursiivsed väljakutsed. Oksi ehk servasid mööda liigub väljakutsete info ja funktsiooni tagastused.

2.2.1 Hargnemiseta rekursioon

Lihtsaimal juhul on funktsioonis ainult üks rekursiivne kutse. Olgu ülesandeks leida arvu n faktoriaal. Sel juhul on üks võimalik rekursiivne lahendus järgmine:

```
int fact(int n)
{
    if (n <= 1) return 1;
    return n * fact(n - 1);
}
```

Hargnemisteta rekursiooni nimetatakse ka lineaarseks rekursiooniks ja seda saab kujutada väljakutsete lineaarse jadana:



Lineaarne rekursioon. Ringid tähistavad funktsiooni väljakutseid, sirged nooled väljakutsete suunda (`main` *kutsub välja* `fact(4)` *jne), kaarjad nooled tagastatavat väärtust.*

2.2.2 Rekursioonivalem

Eelmises ülesandes on baasiks $fact(n) = 1, kui\ n \leq 1$ ning rekursiooni sammuks $fact(n) = n * fact(n-1)$. Kokku moodustavad need juhud **rekursioonivalemi:**

$$fact(n) = \begin{cases} 1, kui & n \leq 1 \\ n * fact(n-1), kui & n > 1 \end{cases}$$

Nii erinevaid samme kui ka baase võib olla mitu.

Enne rekursiivse funktsiooni programmeerima asumist on hea oma idee rekursioonivalemina üles kirjutada. Nii on lihtsam hinnata oma potentsiaalse lahenduse korrektsust. Samuti annab see kohe selgema pildi, millistel juhtudel programm töö lõpetab, kus peaks asuma rekursiivsed väljakutsed koodis ja milliste parameetritega neid tuleb teha. Sageli aitab valemi kirjapanek kaasa ka iteratiivse lahenduse leidmisele, aga ka keerukuse hindamisele (sellest järgmises peatükis) ning võimalikele alternatiivsete lahenduste, näiteks dünaamilist planeerimist kasutavate algoritmide loomisele (sellest lähemalt 5. peatükis).

2.2.3 Hargnemistega rekursioon

Vaatame nüüd tõsisemat rekursiooniülesannet:

Roswelli linnakeses New Mexico osariigis on tegeletud UFOde uurimisega juba aastast 1947 saadik. Muuhulgas on kohalikud teadlased kogunud ka leitud tulnukate DNA proove. Nendes proovides leitud DNA ahel on sarnane inimeste omale, koosnedes samuti neljast nukleotiidist: adeniinist (A), guaniinist (G), tsütosiinist (C) ja tümiinist (T). Teadlased panid aga tähele, et uuritavates ahelates ei esine kaks sama nukleotiidi kõrvuti, samuti ei olnud üheski proovis A ja T kõrvuti ning C-le ei järgnenud kordagi G. Leia kõik võimalikud etteantud pikkusega DNA ahelad, mis võivad kuuluda tulnukatele.

NÄIDE:

3

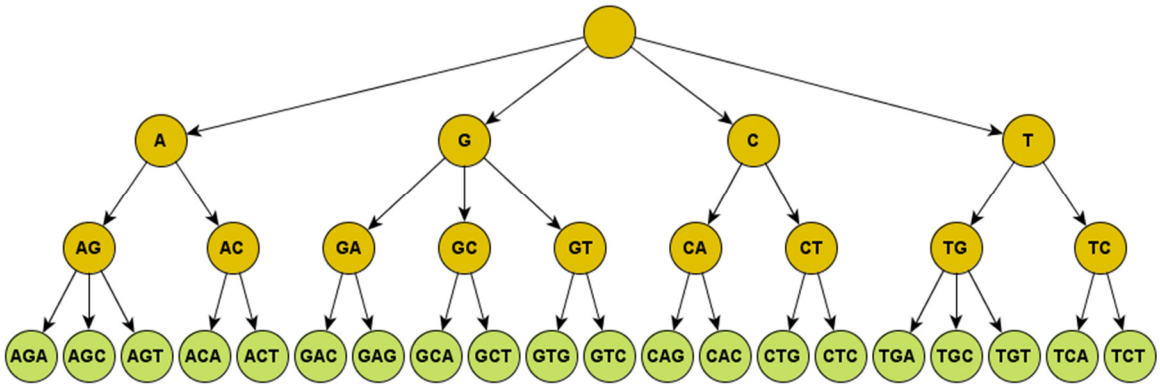
<u>Vastus:</u>

ACA
ACT
AGA
AGC
AGT
CAC
CAG
CTC
CTG
GAC
GAG
GCA
GCT
GTC
GTG
TCA
TCT
TGA
TGC
TGT



Vaatame kõigepealt, kuidas ahelad moodustuvad. Kõigepealt saame esimesele kohale valida ükskõik millise neljast nukleotiidist. Seejärel tuleb valida teine, siis kolmas jne nukleotiid, kuni ahel on soovitud pikkusega. Igale kohale saame valida kuni neli väärtust.

Selles ülesandes on mitu piirangut, mis tähendab, et kõik ahelad, mis tekivad igal korral suvalist nukleotiidi lisades, ei sobi lahenditeks. Sellisel juhul on kaks võimalikku lähenemist: genereerida kõik võimalused ning hinnata iga võimaluse sobivust või arvestada piirangutega juba hargnemisel. Rekursiivse lahenduse korral on viimane variant muidugi eelistatud, kuna vähendab oluliselt rekursiivseid kutseid ja sellega koos programmi tööaega:



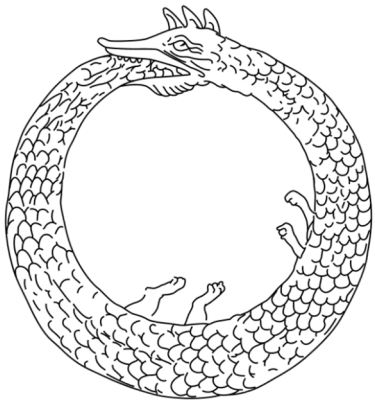
Piirangutega arvestav skeem ülesande lahendite kujunemisest

```
char* vastus;
int pikkus;

void LeiaAhel(int asukoht)
{
    if (asukoht == pikkus) { //oleme konstrueerinud nõutud pikkusega ahela
        for (int i = 0; i < pikkus; i++) { //väljastame selle
            cout << vastus[i];
        }
        cout << endl;
        return; //ja väljume funktsioonist
    }
    if (asukoht == 0) { //esimesel kohal piiranguid pole, valikus on kõik tähed
        vastus[asukoht] = 'A';
        LeiaAhel(asukoht + 1);
        vastus[asukoht] = 'T';
        LeiaAhel(asukoht + 1);
        vastus[asukoht] = 'C';
        LeiaAhel(asukoht + 1);
        vastus[asukoht] = 'G';
        LeiaAhel(asukoht + 1);
        return;
    }
    //ei ole veel nõutud pikkusega ahel, proovime olemasolevale lisada kõik võimalikud
    //nukleotiidid
    char eelmine = vastus[asukoht - 1]; //viimane täht
    switch (eelmine) {
        case 'A':  //A ja T korral saab järgmine täht olla C või G
        case 'T':
            vastus[asukoht] = 'C';
            LeiaAhel(asukoht + 1);
            vastus[asukoht] = 'G';
            LeiaAhel(asukoht + 1);
            break;
        case 'G': //G korral saab järgmine olla C või ka A või T
            vastus[asukoht] = 'C';
            LeiaAhel(asukoht + 1);
            //siin break käsku ei ole, täidetakse ka järgmised laused
        case 'C': //C (ja G) korral saab järgmine olla A või T
            vastus[asukoht] = 'A';
            LeiaAhel(asukoht + 1);
            vastus[asukoht] = 'T';
            LeiaAhel(asukoht + 1);
            break;
    }
}
```

```
int main()
{
    cin >> pikkus;
    vastus = new char[pikkus];
    LeiaAhel(0);
}
```

2.2.4 Sabarekursioon ja väljakutsete optimeerimine



Vahel on rekursiivne väljakutse viimane operatsioon, mis funktsioonis sooritatakse ja selle tagastatavat vastust väljakutsuvas funktsioonis enam ei töödelda. Sellist juhtu nimetatakse **sabarekursiooniks** (*tail recursion*).

Kuna sabarekursioon ei muuda enam funktsiooni olekut, on ka funktsioonist sisenemise ja väljumise töö (pinuviida nihutamine, registrite taastamine jne) ebavajalik. Seetõttu oskavad paljud kompilaatorid sabarekursiivse väljakutse eemaldada. Väljakutse asemel muudetakse funktsiooni parameetrite väärtused lihtsalt ära ning hüpatakse funktsiooni algusse tagasi.

Vaatame näitena rekursiivset funktsiooni, mis teisendab stringi sellele vastavaks arvuks:

```
int strtoint(const char *str, int n)
{
    if (str == 0 || *str == 0)
        return n;
    // str on viit stringi algusele, str+1 viitab siis järgmisele märgile
    // str - '0' annab tulemuseks vastava märgi väärtuse: '0'-'0'=0, '1'-'0'=1 jne
    return strtoint(str + 1, n * 10 + *str - '0');
}
```

Ilma optimeerimiseta genereerib minu kompilaator järgmise koodi:

```
00C81680   push    ebp                     ; eelmise kaadriviida salvestamine
00C81681   mov     ebp,esp
00C81683   mov     eax,dword ptr [str]     ; stringi lõpu kontroll
00C81686   movsx   ecx,byte ptr [eax]
00C81689   test    ecx,ecx                 ; test seab protsessoris vastava lipu,
                                           ; kui tema parameeter on 0
00C8168B   jne     strtoint+12h (0C81692h) ; jne="jump if not equal". Kui eelmise
                                           ; rea tulemus polnud 0, siis jätkame
                                           ; realt 0C81692h
00C8168D   mov     eax,dword ptr [n]       ; kui oli 0, siis tagastame n väärtuse ja
00C81690   jmp     strtoint+30h (0C816B0h) ; hüppame aadressile, kus algab
                                           ; funktsioonist väljumine
00C81692   imul    edx,dword ptr [n],0Ah   ; Aritmeetilised tehted, kümnega
                                           ; korrutamine (0Ah=10d)
00C81696   mov     eax,dword ptr [str]
00C81699   movsx   ecx,byte ptr [eax]
00C8169C   lea     edx,[edx+ecx-30h]       ; Lahutame 30h=48d, mis on märgi '0'
                                           ; ASCII kood
00C816A0   push    edx                     ; Lükkame funktsiooni parameetrid
                                           ; pinusse, kõigepealt n hetkeväärtus
00C816A1   mov     eax,dword ptr [str]     ; Paneme vaadeldava stringi alguse eax
                                           ; registrisse
00C816A4   add     eax,1                   ; Nihutame viida stringi järgmisele
                                           ; märgile
```

```
00C816A7   push    eax                      ; Lükkame viida väärtuse pinusse
00C816A8   call    strtoint (0C81680h)      ; rekursiivne väljakutse
00C816AD   add     esp,8
00C816B0   pop     ebp                      ; kaadriviida taastamine
00C816B1   ret                              ; funktsioonist väljumine
```

Nagu näha, teeb protsessor funktsiooniväljakutsete haldamisega üksjagu tööd. Kui kompileerida kood optimeerimisega, on tulemus hoopis selline:

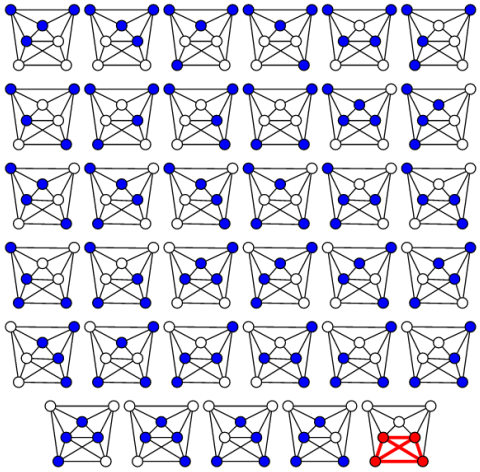
```
012812A0   mov     al,byte ptr [ecx]        ; stringi lõpu kontroll
012812A2   test    al,al
012812A4   je      strtoint+1Bh (012812BBh) ; je="jump if equal". Kui eelmise rea
                                            ; tulemus oli 0, hüppame funktsiooni
                                            ; lõppu.Tagastatava väärtuse käsitlemine
                                            ; on nüüd lihtsam ja hüppamist vähem.
012812A6   movsx   eax,al                   ; Nüüd on arvutamine ka teistsugune.
                                            ; Kümnega korrutamise ja 48 lahutamise
                                            ; asemel
012812A9   lea     edx,[edx+edx*4]          ; korrutatakse neljaga, liidetakse
                                            ; esialgne väärtus,
012812AC   lea     edx,[edx-18h]            ; lahutatakse 24 (18h) ja korrutatakse
                                            ; kahega. Põhjuseks on see, et kahe
                                            ; astmetega korrutamine on protsessoris
                                            ; efektiivsem, see on vaid bittide
                                            ; nihutamine.
012812AF   lea     ecx,[ecx+1]              ; Stringi viida nihutamine
012812B2   lea     edx,[eax+edx*2]
012812B5   mov     al,byte ptr [ecx]        ; stringi lõpu kontroll uuesti
012812B7   test    al,al
012812B9   jne     strtoint+6h (012812A6h)  ; Kui string pole veel lõppenud, hüppame
                                            ; tagasi aritmeetika osa algusse
012812BB   mov     eax,edx                  ; Funktsiooni tulemus tagastatakse
                                            ; tavaliselt eax registris
012812BD   ret                              ; Funktsioonist väljumine
```

Funktsiooni keskmine tööaeg kahanes minu arvutis kaheksakohalise sisendi puhul 30 nanosekundilt kümnele. Enamasti on selline vahe mõistagi tühiasi ja selle pärast ei pea muretsema. Kui meil on aga funktsiooni väljakutsed mitmemiljonilistes tsüklites, võib see täiendav ajakulu muutuda oluliseks. Nagu näha, teeb kompilaator koodi kiirendamiseks mitmesuguseid trikke, aga sellele ei saa kindel olla – kui vaadeldav funktsioon on teistsuguse struktuuriga, võib rekursioon kergesti alles jääda. Seega tasub mõelda, kuidas ise rekursioonist lahti saada ja seda näiteks tsükliga asendada, sarnaselt kompilaatori kasutatud meetodile.

Vahel saab muuta rekursiooni sabarekursiooniks, nii et kompilaatoril tekib võimalus väljakutse optimiseerimiseks. Üks võimalus on kasutada lisaparameetrit tulemuse hoidmiseks ja selle edasiandmiseks sügavuti. Näiteks varasemast tuttava faktoriaali ülesande saab lahendada järgmise sabarekursiivse funktsiooniga:

```
int fact(int n, int vastus)
{
    if (n <= 1) return vastus;
    return fact(n - 1, n*vastus);
}
```

2.3 VARIANTIDE LÄBIVAATAMINE



Paljude ülesannete puhul tekib suur hulk erinevaid andmeid või võimalusi, mille seast on vaja leida sobivaim vastus. Vahel saab seda võimaluste hulka mitmesuguste võtete abil vähendada, aga sageli tuleb sooritada **variantide läbivaatus** (*complete search*), s.t uurida ükshaaval kõiki või vähemalt väga paljusid erinevaid võimalusi. Kõrvalolev joonis illustreerib alamtäisgraafi (sellise alamgraafi, kus kõik tipud on omavahel ühendatud) ehk kliki leidmist kõigi variantide läbivaatuse abil.

Enne arvutite leiutamist oli võimalik lahendada ainult suhteliselt väikesi läbivaatusülesandeid. Kuna aga arvutile on jõukohane vaadata läbi miljoneid variante sekundis, on nüüd võimalik lahendada täiesti uusi ülesannete klasse, alates malest kuni börsi- või kliimasimulatsioonideni.

Siin raamatus käsitletakse kaht peamist lähenemist variantide läbivaatamisele: **tagurdusmeetodit** ja **iteratsioonimeetodit**. Mõlemad meetodid kirjeldavad korduvalt täidetavaid protsesse.

2.3.1 Tagurdusmeetod

Vahel koosneb „sobiv variant“ erinevatest tasemetest või elementidest: kõigepealt on vaja paika panna esimene element, siis teine jne. Kui mõnel sammul ei õnnestu järgmise elemendi jaoks sobivat võimalust leida, tuleme tagasi eelmisele tasemele.

Sellist eelmisele tasemele tagasitulekut nimetatakse **tagurdusmeetodiks** (*backtracking*).

Tagurdusmeetodit võib ette kujutada nagu labürindi läbimist. Sinu eesmärgiks on läbi uurida kõik teed labürindis (labürindis pole tsükleid ja seega ei saa sa samasse kohta tagasi jõuda). Kui satud teede hargnemiskohta, siis valid ühe haru ja lähed mööda seda edasi. Kui jõuad tupikusse, tuled tagasi ja valid viimasest hargnemiskohast uue tee. Kui kõik harud on läbi käidud, tuled tagasi eelviimase hargnemise juurde ja nõnda edasi, kuni jõuad algusesse tagasi.

Tavaline tagurdusmeetodiga lahendatav ülesanne on näiteks Sudoku. Võib proovida panna number 1 esimesse ruutu, siis number 2 teise ruutu jne. Kui tekib vastuolu, tuleme eelmise ruudu juurde tagasi ja proovime sinna panna järgmist numbrit.

2.3.2 Iteratsioonimeetod

Üldmõistena tähendab **iteratsioon** (*iteration*) sama tegevuse korduvat läbiviimist. Kõige tavalisemad näited iteratsioonist on programmeerimiskeeltes kasutatavad **korduslaused** nagu `while` ja `for`-tsükkel.

Variantide läbivaatuse kontekstis tähendab iteratsioonimeetod, et

1. leiame kõigepealt ühe lahendi,
2. muudame selle lahendi mingit parameetrit, et saada teist lahendit
3. kordame tsüklis sammu 2, kuni kõik lahendid on leitud.

2.3.3 Tagurduse ja iteratsiooni võrdlus

Kirjeldatud meetodite illustreerimiseks vaatame järgmist ülesannet:

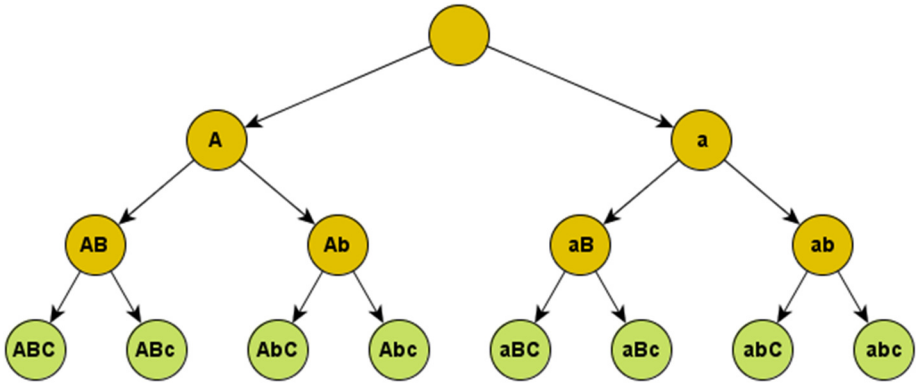
James Bond on tunginud vaenlase peakorterisse, kus superkurjategija parajasti maailma hävitamise masinasse parooli sisestab. Bond näeb sõrmede liigutamise järgi, millised märgid paroolis on, kuid ei näe, millisel hetkel pahalane shift-klahvi all hoiab. Seetõttu ei tea ta, millised on suur- ja millised väiketähed. Kirjuta programm, mis väljastab superkurjategija kõik võimalikud paroolid. Programm saab sisendina parooli pikkuse P (1≤P≤20) ning ladina tähestiku väiketähtedest koosneva parooli.
NÄIDE:
3
abc
Vastus:
abc
abC
aBc
aBC
Abc
AbC
ABc
ABC

2.3.4 Variantide läbivaatamine tagurdusmeetodiga

Tagurdusmeetodi kasutamiseks on siin ülesandes olemas loomulikud elemendid: iga märk paroolis on üks element.

Kui sisendiks on ainult ühetäheline parool, näiteks ’a’, siis vastuseks on kaks võimalust: ’a’ ja ’A’. Kui sisendiks on kaks tähte ’ab’, on võimalusi esimese tähe jaoks 2 ja teise jaoks ka 2: kokku 2x2=4 erinevat parooli (’ab’, ’Ab’, ’aB’ ja ’AB’). Kui lisame kolmanda tähe, siis toimub jälle kaheks hargnemine: kõikidele olemasolevatele paroolidele võime lisada kas ’C’ või ’c’.

Saame järgmise skeemi:



Ülesande lahendamiseks tagurdusmeetodiga saame kirjutada järgmise funktsiooni:

```
void LeiaKoikVoimalused(int asukoht)
{
    if (asukoht == pikkus) { //baasjuht, oleme kõik parooli tähed läbi vaadanud
        for (int i = 0; i < pikkus; i++) {
            cout << vastus[i]; //väljastame kõik parooli tähed
        }
        cout << endl;
        return;
    }
    //parool pole veel valmis
    vastus[asukoht] -= ('a' - 'A'); //paneme käesoleva tähe suureks
    LeiaKoikVoimalused(asukoht + 1); //ja liigume edasi järgmisele tähele
    vastus[asukoht] += ('a' - 'A'); //paneme käesoleva tähe tagasi väikeseks
    LeiaKoikVoimalused(asukoht + 1); //ja liigume järgmisele tähele
}
```

2.3.5 Variantide läbivaatamine iteratsioonimeetodiga

Selles ülesandes on igal sammul ainult 2 valikut. Sellisel juhul on üheks kavalaks võimaluseks kasutada erinevate variantide tähistamiseks bitivektoreid: igale väiksele tähele seame vastavusse `0` ja igale suurele 1. Siis nt vektor `101` tähistab parooli `AbC` ja vektor `011` `aBC`. Bitivektoreid ei ole vaja eraldi andmestruktuurina realiseerida, kuna tavalised täisarvud on esitatud arvutis kahendsüsteemis ja seega on need loomulikud bitivektorid. Enamikus keeltes saab teha täisarvudel ka bitikaupa loogikatehteid. Selline võte teeb lihtsaks antud ülesande iteratiivse e tsükliga lahendamise:

```
int suurArv = 1 << pikkus; //nihutame 1 pikkuse võrra bitte vasakule, suurArv=2^pikkus
for (int i = 0; i < suurArv; i++) { //vaatame läbi kõik arvud 0 – 2^pikkus-1
    for (int j = 0; j < pikkus; j++) { //vaatame läbi kõik kohad
        char taht = vastus[j]; //valime sisestatud paroolist j+1. tähe
        if ((1 << j) & i) { //kui valitud kohal on arvus i bitt seatud
            taht -= ('a' - 'A'); //muudame tähe suureks
        }
        cout << taht; //väljastame tähe
    }
    cout << endl;
}
```

Järgmises tabelis on toodud paroolile pikkusega 3 vastavad arv kümnendsüsteemis, kahendsüsteemis ning sellele arvule vastav parool sisendi abc korral:

Arv kümnendsüsteemis	Arv kahendsüsteemis (3 viimast bitti)	Parool
0	000	abc
1	001	abC
2	010	aBc
3	011	aBC
4	100	Abc
5	101	AbC
6	110	ABc
7	111	ABC

2.3.6 Variantide läbivaatus programmeerimisvõistlustel

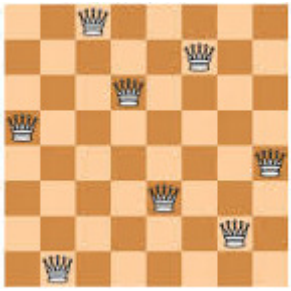
Variantide läbivaatus on enamasti küllaltki aeglane ja kohmakas võte ning mitmete ülesannete lahendamiseks on olemas kiiremad algoritmid. Siiski on variantide läbivaatusel programmeerimisvõistlustel oluline roll.

Variantide läbivaatust kasutatakse tavaliselt järgmistel juhtudel:

- Kui sisend on piisavalt väike. Kuna variantide läbivaatus on küllaltki lihtsalt teostatav, siis pole vaja aega keerulisemale lahendusele kulutada.
- Kui efektiivsemat algoritmi ei oska kirjutada, võib variantide läbivaatusel põhinev lahendus läbida vähemalt lihtsamad testid, mille eest (olenevalt võistluse tingimustest) võib punkte saada.
- Kiirema, aga keerulisema algoritmi õigsuse kontrollimiseks, eriti võistlustel, kus esitamiste arv on piiratud või vale vastuse eest punkte maha võetakse.

2.4 LÄBIVAATUSE OPTIMEERIMINE

Variantide läbivaatamist läheb vaja ka paljudes tavaprogrammeerimise ülesannetes, mistõttu on kasulik tunda viise selle kiirendamiseks. Mõned sellised viisid leiavad käsitlemist käesolevas alapeatükis. Selle jaoks võtame ühe tagurdusmeetodiga lahendatava ülesande ning eesmärgiks on muuta lahendus nii kiireks kui võimalik.



Klassikaline tagurdusmeetodi illustreerimise ülesanne on 8 lipu asetamine: paigutada malelauale kaheksa lippu nii, et ükski neist ei tulistaks ühtki teist. Kõrvaloleval joonisel on üks võimalik lahendus.

8x8 malelauaga saab hakkama ka küllaltki naiivne algoritm, suurema väljakutse huvides vaatleme natuke keerulisemat varianti. Algselt oli see ülesanne Eesti olümpiaadikoondise valikvõistlusel aastal 2016.

Mitu võimalust on N (4≤N≤15) lipu paigutamiseks NxN malelauale, nii et ükski neist ei tulistaks teisi? Lipud tulistavad teineteist, kui nad asuvad samal real, veerul või diagonaalil.
NÄIDE:
N=4
Vastus: 2.
Ajapiirang 1 sekund või siis nii hea, kui suudad.

2.4.1 Lihtne tagurdusmeetod

Üks üsna otsene lahendus ülesandele on järgmine:

```
#define MAXN 16
bool board[MAXN][MAXN];
bool ischecked(int x1, int y1, int x2, int y2) {
    // kaks lippu tulistavad üksteist, kui nad on samal real, veerul või diagonaalil
    return x1 == x2 || y1 == y2 || x2 - x1 == y2 - y1 || x2 - x1 == y1 - y2;
}
```

```
int tryrow(int y, int n) { // y = mitmendale reale proovime lippu panna
    int res = 0;
    for (int x = 0; x < n; x++) { // proovime kõiki veerge
        for (int y1 = 0; y1 < y; y1++) {     // kas eelnevatel ridadel oli mõni lipp
            for (int x1 = 0; x1 < n; x1++) { // mis praegust tulistab
                if (board[x1][y1] && ischecked(x, y, x1, y1)) goto cont; // ei sobi
            }
        }

        if (y == n - 1) // oleme viimasel real, liidame vastusele 1
            res++;
        else {
            board[x][y] = true;  // märgime ruudu kasutatuks
            res += tryrow(y + 1, n); // liidame kõik järgmiste ridade võimalused
            // tagurdusmeetodi võtmekoht - proovides järgmist varianti
            // tuleb taastada eelnev seis!
            board[x][y] = false;
        }
    cont: ;
    }
    return res;
}

int main()
{
    int n;
    cin >> n;
    // alustame esimeselt realt, kust kutsutakse
    // rekursiivselt välja järgmiste ridade proovimisi
    int res = tryrow(0, n);
    cout << res;
}
```

See programm töötab, kuid aeglaselt. Juba N=14 puhul läheb aega 30 sekundit, N=15 võtab mitu minutit. Kas on võimalik algoritmi mitmesajakordselt kiirendada?

2.4.2 Andmete optimeerimine

Esimene tüüpiline optimeerimine, mida saab teha, on algoritmist **mittevajalike andmete väljaviskamine**. Antud juhul on mittevajalik info mängulauda kajastav massiiv. Terve laud meid tegelikult ei huvita, vaja on vaid eelnevate lippude asukohti. Kui need on teada, saab ka loobuda kulukast kahekordsest tsüklist, mis laua ruute läbi käib ja vaatab, kas seal on juba mõni lipp, mis vaatlusalust ruutu tulistab.

Tulemuseks on järgmine kood (`ischecked` ja `main` funktsioonid on samad kui eelmises näites, muutus ainult `tryrow` funktsioon):

```
int queens[MAXN];
int tryrow(int y, int n) { // y = mitmendale reale proovime lippu panna
    int res = 0;
    for (int x = 0; x < n; x++)  { // proovime kõiki veerge
        for (int y1 = 0; y1 < y; y1++) { // kontrollime kõiki eelnevaid lippe
            if (ischecked(x, y, queens[y1], y1)) goto cont; // ei sobi
        }

        if (y == n - 1)
            res++; // oleme viimasel real, liidame vastusele 1
        else {
            queens[y] = x; // jätame uue lipu asukoha meelde
            res += tryrow(y + 1, n); // liidame järgmiste ridade võimalused
        }
    cont: ;
    }
    return res;
}
```

14 lipu puhu kulub seekord 5 sekundit, kuid 15 puhul 35 sekundit.

2.4.3 Ettearvutamine

Järgmise sammuna saab proovida aritmeetiliste operatsioonide „ettearvutamist“. Senistes lahendustes kutsutakse miljoneid kordi välja funktsiooni `ischecked`, mis sooritab palju kordi samu tehteid samade parameetritega. Selle asemel saab nende tehete tulemuse ette meelde jätta, märkides ära, millistel veergudel ja diagonaalidel praegused lipud paiknevad. Üldine idee on selles, et läbivaatusalgoritmil oleks sobivad andmed kohe käepärast olemas.

Vastav lahendus on järgmine:

```
// peame meeles, millistel veergudel ja diagonaalidel lipud on
char col[MAXN]; // (i, j) -> j
char updiag[2 * MAXN]; // (i, j) -> i + j
char downdiag[2 * MAXN]; // (i, j) -> i - j + N

int tryrow(int y, int n) {// y = mitmendale reale proovime lippu panna
    if (y == n) return 1;

    int res = 0;
    for (int x = 0; x < n; x++) { // proovime kõiki veerge
        // kontrollime, kas veerg + diagonaalid on vabad
        if (!col[x] && !updiag[y + x] && !downdiag[y - x + MAXN]) {
            col[x] = 1; // märgime veeru kasutatuks
            updiag[y + x] = 1;
            downdiag[y - x + MAXN] = 1;

            res += tryrow(y + 1, n); // liidame kõik järgmiste ridade võimalused

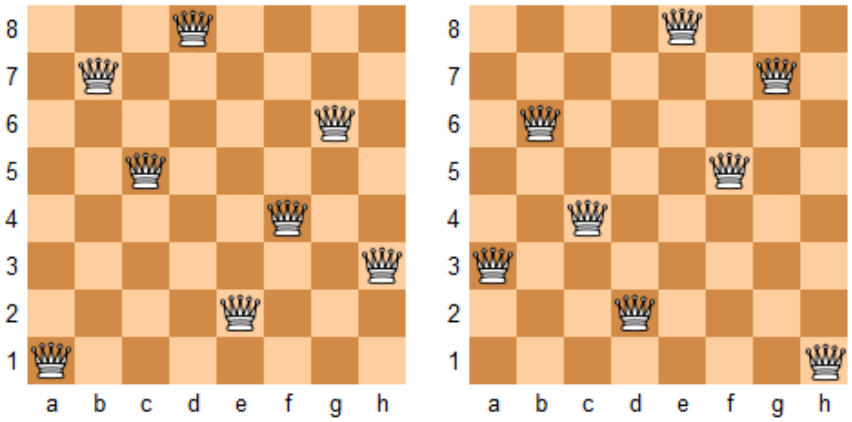
            col[x] = 0;  // taastame eelneva seisu
            updiag[y + x] = 0;
            downdiag[y - x + MAXN] = 0;
        }
    }

    return res;
}
```

Tulemuseks N=14 puhul 2 sekundit, N=15 puhul 14 sekundit.

2.4.4 Otsingupuu ahendamine

Variantide tasemekaupa läbivaatamisel tekib meil nn **otsingupuu**. Kõigi selliste lahenduste puhul tuleks uurida, kas seda puud on võimalik „pügada“, jättes välja harusid, mis ei anna täiendavat infot. Antud juhul on üks ilmne võimalus kasutada ära malelaua sümmeetriat. Kui me paneme lipu esimesel real näiteks esimesele ruudule (a1) ja see annab lahendi, siis peab lahendi andma ka lipu asetamine esimese rea viimasele ruudule (h1) – teised lipud asetsevad lihtsalt peegelpildis. Seega võib esimese rea esimesele poolele vastavate lahenduste arvu lihtsalt kahega korrutada.



Peegelpildis lahendused

Vastava täiendusega lahendus on selline:

```
// peame meeles, millistel ridadel, veergudel ja diagonaalidel lipud on
char row[MAXN];
char col[MAXN]; // (i, j) -> j
char updiag[2 * MAXN]; // (i, j) -> i + j
char downdiag[2 * MAXN]; // (i, j) -> i - j + N
int tryrow(int y, int lim, int n) { // lim = mitut esimese ruutu proovida
    int res = 0;
    if (y == n) {
        res++;
        if (row[0] < n / 2) res++; // esimese rea esimese poole loeme topelt
        return res;
    }

    for (int x = 0; x < lim; x++) { // proovime kõiki veerge
        // kontrollime, kas veerg+diagonaalid on vabad
        if (!col[x] && !updiag[y + x] && !downdiag[y - x + MAXN]) {
            row[y] = x;

            col[x] = 1; // märgime veeru kasutatuks
            updiag[y + x] = 1;
            downdiag[y - x + MAXN] = 1;

            res += tryrow(y + 1, n, n); // liidame kõik järgmiste ridade võimalused

            col[x] = 0;
            updiag[y + x] = 0;
            downdiag[y - x + MAXN] = 0;
        }
    }

    return res;
}
```

```
int main()
{
    int n;
    cin >> n;
    // alustame esimeselt realt, kust kutsutakse
    // rekursiivselt välja järgmiste ridade proovimisi
    int res = tryrow(0, (n + 1) / 2, n); // esimese rea limiit on poole rea pikkuseni
    cout << res;
}
```

N=15 puhul peaks esimesel real proovima nüüd 15 asemel 8 ruutu ning tõepoolest, tööaeg kahanebki peaaegu poole võrra, 7 sekundile.

2.4.5 Sisemiste tsüklite optimeerimine

Otsingupuude puhul on meil rekursiivses funktsioonis tavaliselt mingi tsükkel, milles järgmise taseme variante läbi vaadatakse. Rekursiooni esimesel tasemel käiakse seda tsüklit läbi üks kord. Kui tsükkel koosneb kümnest elemendist, kutsutakse teist taset välja kümme korda. Iga väljakutse kohta läbitakse tsükkel teisel tasemel uuesti ning saadakse suurem arv kolmanda taseme väljakutseid. Nii saame kaugematel tasemetel juba sadu või tuhandeid tsükliläbimisi.

Antud lippude ülesande puhul muutuvad tsüklid järgmistel tasemetel aga lühemaks, viimasel tasemel on lipu paigutamiseks maksimaalselt üks võimalus. Seega saabub maksimaalne töö hulk mõned tasemed enne lõppu.

Need maksimaalse töö hulgaga tasemed on koht, kus tasub optimeerimise mõttes proovida kivist vesi välja pigistada. Antud juhul on parima senise lahenduse kõige intensiivsema töö read need:

```
    for (int x = 0; x < lim; x++) { // proovime kõiki veerge
        // kontrollime, kas veerg+diagonaalid on vabad
        if (!col[x] && !updiag[y + x] && !downdiag[y - x + MAXN]) {
```

Malelaua alumistel ridadel on aga teada, et sobivaid ruute on pigem üsna vähe, mistõttu enamik neist kontrollidest tagastab negatiivse tulemuse. Sellegipoolest läbitakse N=15 puhul tsüklit ikka 15 korda, kuigi vabu veerge on ehk ainult paar tükki. Et tsüklit lühendada, on parem vabu veerge (s.t neid, kuhu pole eelmistel ridadel veel ühtki lippu pandud) spetsiaalselt meeles pidada.

Meelespidamiseks on vaja järjestatud andmestruktuuri, kuhu saab kergesti elemente lisada ja neid sealt eemaldada. Selleks sobib hästi topeltseotud ahel (ahelatest ja teistest andmestruktuuridest on põhjalikumalt juttu järgmises peatükis).

Ahela realiseerimiseks kasutame antud juhul kaht massiivi: üks neist peab iga elemendi kohta meeles, mis on antud veerust järgmine vaba veerg ning teine peab meeles, mis on eelmine vaba veerg.

```
int pr[MAXN + 2]; // eelmine vaba veerg
int nx[MAXN + 2]; // järgmine vaba veerg

int tryrow(int y, int lim, int n) { // lim = mitut esimese rea ruutu proovida
    int res = 0;
    if (y == n) {
        res++;
        if (row[0] <= n / 2) res++; // esimese rea esimese poole loeme topelt
        return res;
    }

    for (int x = nx[0]; x <= lim; x = nx[x]) { // proovime ainut vabu veerge
        // kontrollime, kas diagonaalid on vabad
        if (!updiag[y + x] && !downdiag[y - x + MAXN]) {
            row[y] = x;

            col[x] = 1; // märgime veeru kasutatuks
            updiag[y + x] = 1;
            downdiag[y - x + MAXN] = 1;

            nx[pr[x]] = nx[x]; // eemaldame veeru ahelast
            pr[nx[x]] = pr[x];

            res += tryrow(y + 1, n, n);

            nx[pr[x]] = x; // taastame veeru ahelasse
            pr[nx[x]] = x;

            col[x] = 0;
            updiag[y + x] = 0;
            downdiag[y - x + MAXN] = 0;
        }
    }

    return res;
}
```

Nüüd on tööaeg kahanenud 3,5 sekundile, aga tööriistakastis on veel kavalusi peidus.

2.4.6 Andmestruktuuride optimeerimine

Senises algoritmis on peamine kasutatav andmestruktuur tõeväärtusi hoidev massiiv. Tegelikult on iga selle massiivi element omaette täisarv, aga praegune lahendus kasutab ainult selle viimast bitti. Kui on vaja selle massiiviga mingeid operatsioone sooritada, tuleb muuta iga elementi eraldi. Samas eksisteerib ka „naturaalne“ bitimassiiv ehk tavaline täisarv. Praegune ülesanne pakub suurepärase illustratsiooni tõeväärtuste massiivi asendamisest täisarvuga.

Kui asendada kõik veergude ja diagonaalide meelespidamiseks kasutatavad massiivid täisarvudega ja kasutada nendega opereerimiseks bitioperatsioone, on tulemuseks järgmine kood:

```
int tryrow(int y, int lim, int n, int col, int updiag, int downdiag) {
    int res = 0;
    if (y == n) {
        res++;
        if (row) res++; // esimese rea esimese poole loeme topelt
        return res;
    }

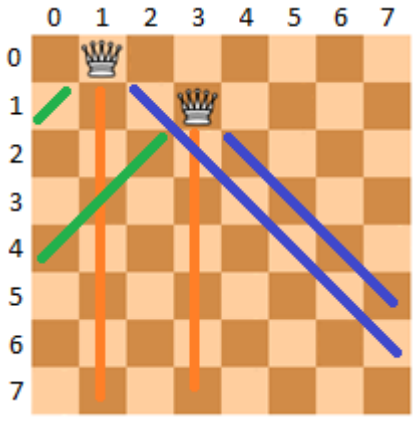
    int positions = (1 << lim) - 1; // arv, kus lim bitti on ühed
    positions &= ~col;      // nullime hõivatud veerud
    positions &= ~updiag;   // nullime hõivatud diagonaalid
    positions &= ~downdiag; // nullime hõivatud diagonaalid

    while (positions > 0) // vabu ruute on niikaua kui mõni bitt on mittenull
    {
        int lsb = lsb = -(positions) & positions; // alumine vaba bitt
        if(y == 0)
            row = (lsb < (1 << n/2)); //Kas oleme esimese rea esimeses pooles?
        int newupdiag = (updiag | lsb) >> 1;     // järgmise rea üks diagonaal
        int newdowndiag = (downdiag | lsb) << 1; // järgmise rea teine diagonaal
        int newcol = col | lsb;  // järgmise rea hõivatud veerud
        positions &= ~lsb;       // nullime praeguse ruudu
        res += tryrow(y + 1, n, n, newcol, newupdiag, newdowndiag);
    }

    return res;
}

int main()
{
    int n;
    cin >> n;
    // alustame esimeselt realt, kust kutsutakse
    // rekursiivselt välja järgmiste ridade proovimisi
    // esimese rea limiit on poole rea pikkuseni, algul on veerud ja diagonaalid vabad
    int res = tryrow(0, (n + 1) / 2, n, 0, 0, 0);
    cout << res;
}
```

Tööaeg oli minu arvutis 1,1 sekundit!



Joonisel on kahe asetatud lipuga malelaud. Oranžid jooned märgivad blokeeritud veerge, rohelised mööda üht diagonaali ja sinised mööda teist diagonaali tule all olevaid ruute. Järgnevas tabelis on toodud muuujate `col`, `updiag` ja `downdiag` väärtused enne lipu asetamist vastavale reale. col väärtused püsivd paigal, updiag väärtused nihkuvad igal järgmisel real ühe koha võrra vasakule, downdiag väärtused paremale.

Rida	col	updiag	downdiag
0	00000000	00000000	00000000
1	01000000	10000000	00100000
2	01010000	00100000	00011000

Kui bitioperatsioonidega mitte sina peal olla, siis vajavad mõned kavalused siin koodis selgitamist:

`1 << lim` loob arvu 2^{lim} ehk sellise, kus ühe biti väärtus on 1 ja sellele järgneb `lim` nulli. Kui sellest arvust lahutada 1, saame arvu, kus on `lim` bitti väärtusega 1.

`~col` tagastab arvu, kus muutuja `col` bitid on ümber pööratud. `positions &= ~col` nullib seega kõik bitid, mis muutujas `col` olid võrdsed ühega.

`-(positions) & positions` tagastab muutuja `positions` kõige alumise biti, mille väärtus on 1. See võimaldab üliefektiivset tsüklit üle vabade ruutude.

2.4.7 Rekursiooni eemaldamine

Viimase nipina tuletame meelde, et rekursioon toob kaasa teatava lisakulu. Seega proovime teha parameetrite käsitlemist paremini, kui kompilaator sellega hakkama saab. Selle asemel, et diagonaalide ja veergude seisu parameetritena järgmisele tasemele edasi anda, loome massiivid, kus on igale tasemele vastav seis.

Tulemuseks on selline funktsioon:

```
int nqueens(int n) {
    int col[MAXN];  // parameetrite massiivid
    int updiag[MAXN];
    int downdiag[MAXN];
    int positions[MAXN];

    int half = (n + 1) / 2;
    int halfwayBit = n % 2 == 1 ? 1 << half - 1: -1;

    int res = 0;
    int fullMask = (1 << n) - 1;
    int halfMask = (1 << half) - 1;

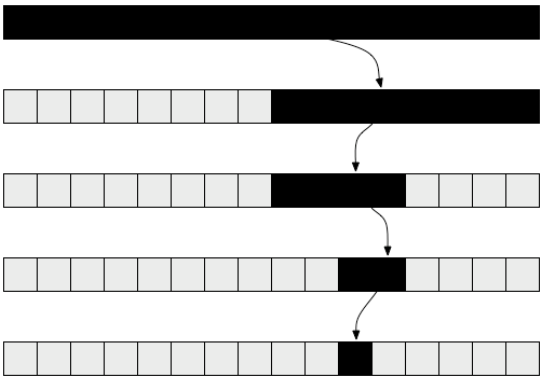
    int y = 0;
    positions[0] = halfMask;
    col[0] = updiag[0] = downdiag[0] = 0;
    int points = 2;
    int bits = positions[0];
    for (;;)
    {
        if (bits == 0) { // praegusel real võimalused otsas
            if (y == 0) break; // esimene rida läbi
            bits = positions[--y]; // tagasi eelmisele reale
        }
        else
        {
            int lsb = lsb = -(bits) & bits; // alumine vaba bitt
            if (y == 0 && lsb == halfwayBit) {
                points = 1;
            }

            bits &= ~lsb;     // nullime praeguse ruudu
            if (y < n - 1) {  // saab minna järgmisele reale
```

```
                positions[y] = bits;  // salvestame eelmise positsiooni
                int prev = y++;
                updiag[y] = (updiag[prev] | lsb) >> 1;     // järgmise rea diagonaal
                downdiag[y] = (downdiag[prev] | lsb) << 1; // järgmise rea diagonaal
                col[y] = col[prev] | lsb;  // järgmise rea hõivatud veerud
                bits = fullMask & ~(col[y] | updiag[y] | downdiag[y]); // vabad bitid
                positions[y] = bits;
            }
            else
                res += points; // viimane rida, suurendame skoori
        }
    }
    return res;
}
```

See versioon lahendab N=15 korral ülesande 0,9 sekundiga, nii et olemegi saavutanud alguses sõnastatud mitmesajakordse kiirendamise eesmärgi. Samas muudab viimane trikitamine koodi üksjagu pikaks ja „karvaseks“, mis on tüüpiline kaasnev efekt, kui koodi kiirust üle teatud piiri optimeerida. Põhimõtteliselt saaks siit veel natuke edasi minna, kuid siis muutuks kood juba väga raskesti arusaadavaks – enamiku tavaliste situatsioonide jaoks on sobivaim ilmselt eelviimane lahendus.

2.5 KAHENDOTSING



Laste seas on populaarsed arvu äraarvamise mängud: üks mõtleb mingi arvu teatud vahemikus ja teine katsub selle ära arvata. Mõtleja annab arvajale tagasisidet, kas tema mõeldud arv on suurem või väiksem kui arvaja pakutu. Kes seda mängu rohkem mängisid, taipasid enamasti, et kiiremaks äraarvamiseks on kasulik pakkuda arvu võimalikult lubatud vahemiku keskele. Näiteks kui arv võib olla lõigus 1...100, siis esimene hea pakkumine on 50. Edasi sõltub muidugi mõtleja vastusest: kui tema arv on suurem, tuleb järgmine pakkumine teha lõigu 51…100 keskele (75), kui aga väiksem, siis lõigu 1…49 keskele (25) jne, kuni õige arv leitud. Sellisel moel tuleb suvalise arvu vahemikus 1…100 äraarvamiseks teha maksimaalselt 7 pakkumist ($\log_2 100$).

2.5.1 Vastuse kahendotsing

Sarnast vastuse otsimise strateegiat läheb sageli vaja ka programmeerimisülesannete lahendamiseks. Sellist lähenemist nimetatakse **vastuse kahendotsinguks** (*binary search the answer*).

Kostja kolib Itaaliasse ja pakib rutakalt asju kastidesse. Kuna tal on sellega kiire ja ta on ka üksjagu laisk, siis võtab ta asju sellises järjekorras, nagu kätte juhtub, ja asetab neid järjest kastidesse – kõigepealt ainult esimesse kasti, seejärel ainult teise jne. Kirjuta programm, mis aitab Kostjal valida väikseima võimaliku kasti suuruse, nii et kõik ta `M` asja mahuksid tema pakkumismeetodi juures ära maksimaalselt `N` kasti.
Sisendi esimesel real on antud kastide arv `N` (`1≤N≤100000`), teisel real asjade arv `M` (`1≤M≤100000`) ja viimasel real asjade suurused pakkimise järjekorras ($1 \le M_i \le 1000$).
NÄIDE:

```
3
6
2 16 3 7 10 12
```

Vastus: 20 (esimesse kasti 1. ja 2. ese, teise 3., 4. ja 5. ese ning kolmandasse viimane ese).

Ülesandes on vaja leida ainult üks arv – väikseim võimalik kasti suurus. Leiame kõigepealt piirid, mille vahele vastus ehk kasti suurus peab jääma. Väikseim võimalik väärtus on 1 ja suurim väärtus kõikide asjade suuruste summa (siis mahuvad kõik asjad ühte kasti). Kui otsitava kasti suurus on x, siis juhul kui valiksime x-ist väiksema kasti suuruse, siis asjad enam ettenähtud arvu kastidesse ära ei mahuks ning õige vastus peab olema valitud väärtusest suurem. Kui aga valime suurema kasti suuruse kui x, siis mahuvad asjad kindlasti ära ning otsitav suurus ei saa olla enam suurem valitud väärtusest. Seega saab iga suvalise väärtuse y jaoks proovida, kas pakkimine õnnestub, ja vastavalt sellele otsustada, kas $x > y$ (ei mahtunud) või $x \le y$ (mahtusid). Sellisel juhul saab kasutada kahendotsingut.
Parema loetavuse huvides kasutame eraldi funktsiooni `mahub`, mis katsetab, kas asjad mahuvad valitud suurusega kasti:

```
bool mahub(int suurus)
{
    int mitmesKast = 1;
    int praeguneKast = 0;
    int i = 0;
    while (i < m) { //käime kõik asjad läbi
        if (mitmesKast > n) {
           return false; //ei mahtunud n kasti.
        }
        if (praeguneKast + s[i] > suurus) { //ese ei mahu pakitavasse kasti
            mitmesKast++; //võtame järgmise..
            praeguneKast = 0; //tühja kasti
        }
        else {
            praeguneKast += s[i]; //kui ese mahub, paneme selle kasti
            i++; //ja võtame järgmise eseme
        }
    }
    return true; //kõik asjad on maksimaalselt n kastis
}
```

Siin on kahendotsingu osa:

```
int vahim = 0; //alumine piir (kasti suurus, kuhu asjad ei mahu ära)
int suurim = summa; //ülemine piir (kasti suurus, kuhu esemed mahuvad ära)
while (vahim + 1 < suurim) {//kuni alam- ja ülempiiri vahel on veel proovimata suurusi
    int proov = (vahim + suurim) / 2; //proovime keskmist suurust
    if (mahub(proov)) //kui mahub,
        suurim = proov; //oleme leidnud väiksema suurima kasti, muudame ülempiiri
    else
        vahim = proov; //ei mahtunud, tõstame alampiiri
}
```

Tsükli lõpuks on muutujas `suurim` ülesande vastus.

Pane tähele, et täisarvude puhul, nagu antud ülesandes, on lihtne otsustada, millal õige vastus käes on. Kui ülesandes tuleb otsida reaalarvulist vastust, siis tuleb lahendajal ise otsustada, millal võiks olla käes piisavalt täpne vastus. Peale vahe võrdlemise on üheks võimaluseks veel ette ära hinnata või välja arvutada, mitu jagamist oleks mõistlik kokku teha.

2.5.2 Kahendotsing andmetest

Vastuse kahendotsingu juures on oluline, et võimalikud vastusevariandid on järjestatud ning vastusel on olemas mingi ülem- ja alampiir. Näiteülesandes otsisime vastust kindlast naturaalarvude lõigust, kuid kahendotsingu tuntumaks kasutusalaks on mingi konkreetse väärtusega elemendi otsimine sorteeritud andmejadast. Täpsemalt saab kahendotsingu abil leida, mitmendal kohal jadas otsitav element asub või kas see üldse antud jadas esineb.

Jadast otsimise algoritm on sarnane vastuse kahendotsingule, kuid väärtuse ülem- ja alampiiri asemel saab jaotamisel oluliseks hoopis elementide arv jadas. Algoritm ise on lihtne: leiame otsitava jada piirides kõige keskel asuva elemendi ja võrdleme selle väärtust otsitavaga. On kolm võimalust:

1. Väärtused on võrdsed, element jadast on leitud.
2. Otsitav väärtus on väiksem, järelikult peab see asuma võrreldavast väärtusest eespool.
3. Otsitav väärtus on suurem, järelikult peab see asuma võrreldavast elemendist tagapool.

Aga mis siis, kui otsitavat elementi jadas ei leidugi? See on neljas võimalus, millega andmetest kahendotsingul tuleb kindlasti arvestada. Enamasti tagastatakse leidumise korral vastav indeks ning mitteleidumise korral -1 (aga võib ka tagastada tõeväärtuse, kas element leidub jadas või mitte). Saame järgmise rekursiivse definitsiooni:

$$otsi(v, s_{1,} \dots, s_n) = \begin{cases} -1, & kui\ S = \emptyset \\ n/2\,, & kui\ v = s_{n/2} \\ otsi(v, s_{1,} \dots, s_{n/2-1,}), & kui\ v < s_{n/2} \\ otsi(s_{n/2+1,} \dots, s_n), & kui\ v > s_{n/2} \end{cases}$$

Tavaliselt on lihtsam anda funktsioonile sisendiks terve algne jada (või kasutada jada jaoks üldse globaalset muutujat) ning jada jagamise asemel muuta elementide indekseid, mille vahel jadas otsime. Siin on rekursiivne funktsioon, mis just nii teeb:

```
int* jada;
int otsi_rekursiivselt(int vaartus, int algus, int lopp)
{
    if (algus > lopp) return -1; //ei leitud sellist väärtust
    int keskkoht = (algus + lopp) / 2; //leiame jada keskmise elemendi indeksi
    //kui otsitav väärtus on väiksem kui keskmise elemendi väärtus
    if (vaartus < jada[keskkoht])
        //otsime jadast eestpoolt
        return otsi_rekursiivselt(vaartus, algus, keskkoht - 1);
    //kui otsitav väärtus on suurem kui keskmine elemendi väärtus
    if (vaartus > jada[keskkoht])
        //otsime jadas tagantpoolt
        return otsi_rekursiivselt(vaartus, keskkoht + 1, lopp);
    //vaartus == jada[keskmine] ja tagastame sobiva elemendi indeksi
    return keskkoht;
}
```

Ja siin iteratiivne lahendus:

```
int iotsing(int* jada, int vaartus, int algus, int lopp)
{
      while (algus <= lopp) {
            int keskkoht = (algus + lopp) / 2;
            if (vaartus == jada[keskkoht]) return keskkoht;
            else if (vaartus < jada[keskkoht])
                  lopp = keskkoht - 1;
            else if (vaartus > jada[keskkoht])
                  algus = keskkoht + 1;
      }
      return -1;
}
```

Javas on olemas meetod `java.util.Arrays.binarySearch()`, mis saab argumendiks sorteeritud jada ning otsitava väärtuse ning tagastab otsitava väärtuse indeksi või -1, kui väärtust jadas ei ole.

C++ standardteegis on funktsioon `binary_search`, mis tagastab tõeväärtuse, kas element on jadas või mitte. Kui on vaja elemendi indeksit, saab kasutada funktsiooni `lower_bound`.

Pythonis on C++ `lower_bound`'iga analoogne funktsioon `bisect.bisect_left`, mis saab argumentideks sorteeritud listi otsitava väärtuse. Lisaks saab kasutada alampiiri, mis on vaikimisi 0 ja ülempiiri, mis on vaikimisi listi pikkus. See funktsioon tagastab esimese koha, kuhu antud element sisestada tuleks, et jada sorteeritud oleks.

2.5.3 Topeltkahendotsing

Vaatame veel kord arvu äraarvamise mängu. Mida teha siis, kui vahemik on jäänud kokku leppimata? Sellisel juhul on üheks võimaluseks kõigepealt proovida arvata ära vahemik. Kas arv on suurem kui 10? Suurem kui 100? Suurem kui 1000? jne. Õnneks inimestel pole kombeks mõelda lõputult suuri arve, nii et sel moel vahemikku arvates saab selle küllalki kiiresti teada.

Veelgi parem on kasvatada vahemikku iga kord mitte 10, vaid 2 korda. Inimesele ehk ebamugav, kuid arvutile, vastupidi, sobivamgi. Sellist vahemiku määramise viisi tuntakse kui **topeldusmeetodit** ning kogu otsingut kokku nimetatakse **topeltkahendotsinguks.**

```
long long topeltKahendOtsi() {
    long long algus = 0, lopp = 0; //alustame nullist
    int samm = 1;
    char ch;
    while (lopp >= 0) { //ei ole ületäitumist
        cout << lopp << endl;
        cin >> ch;
        if (ch == '=') return lopp; //leidsime otsitava arvu
        else if (ch == '<') break; //leidsime piiri
        else if (ch == '>') { //otsitav arv on suurem
            algus = lopp; //nihutame alampiiri
            lopp = algus + samm; //tõstame ülempiiri sammu võrra
            samm *= 2; //suurendame sammu 2 korda
        }
        else
            cout << "sisesta '>', '<' või '='" << endl;
    }
    if (algus <= lopp)
        return kahendOtsi(algus, lopp); //tavaline kahendotsing leitud vahemikust
    else
        return -1;
}
```

Topeltkahendotsingut on hea kasutada, kui otsitavad väärtused on otsimise alguskoha lähedal või kui funktsioon ja tema väärtuste arvutamise hind kasvab kiiresti. Samuti võimaldab see meetod kasutada kahendotsingut siis, kui ülem- või alamtõke ei ole teada.

2.5.4 Otsing kahemõõtmelisest tabelist sadulameetodil

Mõnikord on vaja otsida arvu kahemõõtmelisest tabelist, mille read ja veerud on sorteeritud nagu järgmises ülesandes:

Igaüks on vast tajunud, et tuulise ilmaga tundub välisõhu temperatuur mõnevõrra teistsugune kui tuulevaikuses. Selleks, et täpsemini hinnata välitingimuste mõju inimesele, on kasutusele võetud tuule-külma indeks, mis näitab, kui tugev külmakraad tegelikult kehale mõjub arvestades õhutemperatuuri ja tuule kiirust.
Õpetaja Tõnu andis oma õpilastele ülesandeks jälgida nädal aega järjest õhutemperatuuri ja tuule kiirust ning leida tuule-külma indeksi tabelist naha poolt tegelikult tajutav temperatuur. Kuna ta teab, et mitmed õpilased on laisad ja pakuvad numbreid huupi tabelisse vaatamata, aita tal kirjutada programm, mis kontrollib, kas õpilase sisestatud andmed on kooskõlalised, st kirjas olev number peab tabelis olemas olema.
Esimesel real on antud tabeli mõõtmed t ja k. Teisel real õhutemperatuurid $10.0 \geq t_i \geq -50.0$ mittekasvavas järjekorras. Igal järgneval k real on tajutavad temperatuurid $10.0 \geq a_i \geq -100.0$, kusjuures esimene rida vastab tuulekiirusele 1 m/s, teine 2m/s jne. Sisendi viimasel real on 7 arvu: õpilase poolt pakutud tajutavad temperatuurid. Väljastada samuti 7 arvu: iga õpilase pakutud temperatuuri kohta vähim tegelik temperatuur, millel selline tajutav temperatuur olla sai. Kui sellist temperatuuri tabelis pole, väljastada selle asemel POLE.
NÄIDE:

```
4 3
10.0 7.0 4.0 1.0
9.9 6.6 3.5 -0.3
9.2 5.7 2.2 -1.3
8.0 4.3 0.6 -3.1
9.2 7.7 1.3 -1.3 6.6 2.2 -0.6
```

Vastus:

```
10.0 POLE POLE 1.0 7.0 4.0 POLE
```

Kõige lihtsam on seda ülesannet lahendada sadulameetodil. Otsingut alustatakse tabelis väärtusest, mis on oma reas kõige suurem ja samal ajal oma veerus kõige väiksem. Kuna tabel on sümmeetriline, võib alustada ka väärtusest, mis on veerus suurim ja reas vähim. Selliseid väärtusi nimetatakse matemaatikas maatriksi sadulpunktideks, sellest ilmselt tuleb ka antud meetodi nimetus.

See on huvitav punkt, sest kui selles kohas asuvat väärtust a_{ij} võrrelda otsitava väärtusega v, saab edasisest otsingualast välistada terve veeru või rea. Kui valida rea suurim väärtus, mis on samal ajal oma veerus vähim, siis juhul kui $v < a_{ij}$ ja $a_{ij} \leq a_{i,j+1} \leq \cdots \leq a_{im}$, kus m on ridade arv, võib i. veeru otsinguruumist välja jätta. Analoogselt saab näidata, et kui $v > a_{ij}$, siis saame kõrvale jätta terve j. rea.

```
pair<int, int> sadulotsing(float** tabel, int m, int n, float vaartus)
{
    int rida = 0;
    int veerg = n-1;

    while (rida < m && veerg > -1) {
        if (abs(vaartus - tabel[rida][veerg]) < 0.01) return make_pair(rida, veerg);
        else if (vaartus > tabel[rida][veerg]) {
            veerg--;
        }
        else {
            rida++;
        }
    }
    return make_pair(-1, -1);
}
```

Tuule kiirus m/sek	Õhutemperatuur °C																		
	10	7	4	1	-2	-5	-8	-11	-14	-17	-20	-23	-26	-29	-32	-35	-38	-41	-44
2	9,2	5,7	2,2	-1,3	-4,8	-8,3	-12	-15	-19	-22	-26	-29	-33	-36	-40	-43	-47	-50	-54
3	8,5	4,9	1,3	-2,3	-5,9	-9,5	-13	-17	-20	-24	-28	-31	-35	-38	-42	-46	-49	-53	-56
4	8	4,3	0,6	-3,1	-6,8	-10	-14	-18	-22	-25	-29	-33	-36	-40	-44	-47	-51	-55	-58
5	7,6	3,8	0,1	-3,7	-7,4	-11	-15	-19	-23	-26	-30	-34	-38	-41	-45	-49	-53	-56	-60
6	7,2	3,4	-0,4	-4,2	-8	-12	-16	-19	-23	-27	-31	-35	-39	-42	-46	-50	-54	-58	-61
7	6,9	3,1	-0,8	-4,6	-8,5	-12	-16	-20	-24	-28	-32	-36	-39	-43	-47	-51	-55	-59	-63
8	6,7	2,8	-1,1	-5	-8,9	-13	-17	-21	-25	-29	-32	-36	-40	-44	-48	-52	-56	-60	-64
9	6,4	2,5	-1,5	-5,4	-9,3	-13	-17	-21	-25	-29	-33	-37	-41	-45	-49	-53	-57	-61	-65
10	6,2	2,2	-1,8	-5,7	-9,7	-14	-18	-22	-26	-30	-34	-38	-42	-46	-50	-53	-57	-61	-65
11	6	2	-2	-6	-10	-14	-18	-22	-26	-30	-34	-38	-42	-46	-50	-54	-58	-62	-66
12	5,8	1,8	-2,3	-6,3	-10	-14	-18	-23	-27	-31	-35	-39	-43	-47	-51	-55	-59	-63	-67
13	5,6	1,6	-2,5	-6,6	-11	-15	-19	-23	-27	-31	-35	-39	-43	-47	-51	-55	-59	-64	-68
14	5,5	1,4	-2,7	-6,8	-11	-15	-19	-23	-27	-31	-35	-40	-44	-48	-52	-56	-60	-64	-68
15	5,3	1,2	-2,9	-7	-11	-15	-19	-24	-28	-32	-36	-40	-44	-48	-52	-56	-61	-65	-69
16	5,2	1	-3,1	-7,2	-11	-16	-20	-24	-28	-32	-36	-40	-45	-49	-53	-57	-61	-65	-69
17	5	0,9	-3,3	-7,5	-12	-16	-20	-24	-28	-32	-37	-41	-45	-49	-53	-57	-62	-66	-70
18	4,9	0,7	-3,5	-7,6	-12	-16	-20	-24	-29	-33	-37	-41	-45	-50	-54	-58	-62	-66	-70
19	4,8	0,6	-3,6	-7,8	-12	-16	-20	-25	-29	-33	-37	-42	-46	-50	-54	-58	-63	-67	-71
20	4,7	0,4	-3,8	-8	-12	-17	-21	-25	-29	-33	-38	-42	-46	-50	-55	-59	-63	-67	-71
21	4,5	0,3	-3,9	-8,2	-12	-17	-21	-25	-29	-34	-38	-42	-46	-51	-55	-59	-63	-68	-72
22	4,4	0,2	-4,1	-8,3	-13	-17	-21	-25	-30	-34	-38	-42	-47	-51	-55	-60	-64	-68	-72
	Madal risk alajahtuda				Risk alajahtuda, kui ilma vastava kaitseta liiga kaua õues viibida				Risk tõuseb: katmata nahk kahjustub 10-30 minutiga			Kõrge risk: katmata nahk kahjustub 5-10 minutiga		Katmata nahk kahjustub 2- 5 minutiga		Kõrge risk: katmata nahk kahjustub alla 2 minutiga. Õues viibimine on tervistkahjustav			

Tuule-külma tabel terviseameti kodulehelt http://terviseamet.ee/keskkonnatervis/vaelisohk/tuule-kuelma-indeks.html

Sadulameetod kahendotsinguga

Sadulameetodit saab kombineerida kahendotsinguga. Selle asemel, et suunamuutuskohta lineaarselt otsida, võib kasutada pöördekoha leidmiseks kahendotsingut.

```
pair<int, int> sadulotsing2(float** tabel, int m, int n, float vaartus)
{
    int rida = 0;
    int veerg = n - 1;

    while (rida < m && veerg > -1) {
        if (abs(vaartus - tabel[rida][veerg]) < 0.01) return make_pair(rida, veerg);
        else if (vaartus > tabel[rida][veerg]) {
            veerg = otsiReast(tabel, rida, 0, veerg, vaartus);
        }
        else {
            rida = otsiVeerust(tabel, veerg, rida, n, vaartus);
        }
    }
    return make_pair(-1, -1);
}
```

Reast kahendotsimise funktsioon:

```
int otsiReast(float** tabel, int rida, int algus, int lopp, float vaartus)
{
    if (algus > lopp) return -1;
    while (algus < lopp) {
        int keskkoht = (algus + lopp) / 2;
        if (vaartus > tabel[rida][keskkoht])
            lopp = keskkoht;
        else if (vaartus <= tabel[rida][keskkoht])
            algus = keskkoht + 1;
    }
    return algus - 1;
}
```

Veerust kahendotsimise funktsioon:

```
int otsiVeerust(float** tabel, int veerg, int algus, int lopp, float vaartus)
{
    if (algus > lopp) return -1;
    while (algus < lopp) {
        int keskkoht = (algus + lopp) / 2;
        if (vaartus >= tabel[keskkoht][veerg])
            lopp = keskkoht;
        else if (vaartus < tabel[keskkoht][veerg])
            algus = keskkoht + 1;
    }
    return algus;
}
```

Sadulameetod topeltkahendotsinguga

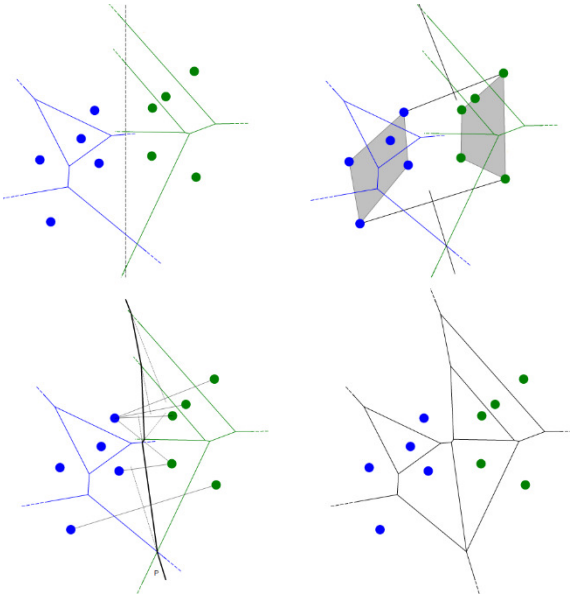
Vaatame veel korra lähemalt, kuidas arvud ridu ja veergu pidi sorteeritud tabelis paiknevad. Järgmisel joonisel on 20x20 tabel, milles on arvud vahemikus 0-1000. Sinised ruudud näitavad teed arvu 409 otsimisel lineaarselt liikudes:

43	49	91	127	145	147	166	170	171	188	237	325	339	376	402	423	435	437	440	471
53	96	100	131	153	170	189	214	254	272	320	337	373	396	406	431	442	446	456	517
80	97	103	143	159	187	241	257	282	327	331	339	375	400	414	440	456	470	508	519
81	99	112	146	207	236	249	267	300	329	346	358	394	401	440	467	467	480	511	534
131	148	192	218	239	245	257	280	305	336	397	418	452	458	510	533	554	556	561	597
155	179	199	238	244	259	265	302	317	351	434	447	464	467	523	550	588	589	597	607
177	185	203	254	254	298	343	357	381	411	465	474	498	506	531	588	593	595	605	607
192	195	234	259	270	320	345	369	388	429	470	485	503	513	539	590	612	629	634	648
215	223	238	272	286	335	351	385	389	468	472	488	507	530	556	592	615	629	682	697
229	230	276	277	313	342	376	426	426	485	493	496	556	564	599	632	640	642	690	701
253	301	343	347	353	409	436	446	459	504	510	545	577	583	600	635	646	650	714	731
301	343	353	366	403	430	442	459	513	519	526	550	578	598	606	650	655	687	729	769
334	350	363	395	416	478	506	513	538	555	586	588	604	637	641	653	659	736	757	789
372	422	450	471	480	510	524	539	543	565	590	627	669	690	706	710	751	774	785	813
376	431	467	477	504	540	548	586	598	606	615	644	683	710	727	728	757	781	810	825
404	447	467	511	514	546	573	622	639	658	687	706	718	724	733	749	763	805	825	833
404	463	508	514	527	571	620	637	644	693	704	707	718	728	748	800	801	812	830	841
457	480	508	518	530	583	634	641	691	697	713	743	754	765	774	827	859	868	897	914
475	484	513	527	547	597	657	675	711	762	773	819	834	839	895	903	905	928	938	946
493	506	521	551	718	720	733	743	757	844	854	858	867	877	898	912	915	933	944	960
493	506	521	551	718	720	733	743	757	844	854	858	867	877	898	912	915	933	944	960

Esimest rida mööda liikudes minnakse suhteliselt kaugele, kuid sealt edasi on suunamuutused päris tihedad. See on suvaliste andmete korral omane mitte ainult konkreetse näite korral, vaid enamikul juhtudel. Joonisel on väiksemad arvud punasemad ja suuremad rohelisemad. Nii on visuaalselt hästi näha, et tekivad diagonaalsed triibud sarnaste väärtustega aladest. Juba esimese reast või veerust otsimisel liigume otsitava arvu väärtusele üsna lähedale ehk selle arvuga sama värvi alasse. Edasine otsimine toimub sama tooni alas ja kuna alad on diagonaalsed, on keeramiskohad ka enamasti lähedal. Seetõttu on efektiivsem kasutada suunamuutmise kohtade leidmiseks kasutada topeltkahendotsingut. Selle algoritmi kirjutamine jääb lugejale kodutööks.

Konkreetsete andmete iseloomust sõltuvalt on mõnikord mõttekas kasutada mööda ridu liikudes üht meetodit, mööda veerge aga teist.Terviseameti tuule-külmaindeksi tabelis kahanevad väärtused mööda ridu kiiremini kui mööda veerge ning tekkivad sarnaste suurustega alad on järsemad – see tähendab, et rida mööda liigutakse enamasti väga lähedale, mööda veerge aga tuleb rohkem ka pikemaid liikumisi. Sellisel juhul on mõttekas kasutada mööda ridu liikudes näiteks topeltkahendotsingut (või minna lineaarselt, sest võistlustel tuleb kindlasti arvestada ka koodi implementeerimise ajaga) ning mööda veerge kasutada kahendotsingut.

2.6 JAGA JA VALITSE



„Jaga ja valitse“ (*divide and conquer* ehk *D&C*) algoritmi korral jagatakse ülesanne kaheks või enamaks väiksemaks osaks, kuni osad on sellise suurusega, mida on lihtne lahendada. Need väiksemad alamülesanded lahendatakse ja nende vastustest pannakse kokku kogu ülesande vastus. Seda tüüpi lähenemise puhul on alamülesanded enam-vähem võrdse suurusega. Kõrvalolev joonis illustreerib Voronoi diagrammi leidmist „jaga ja valitse“ tüüpi algoritmiga. Voronoi diagramm on pilt, mis on jagatud piirkondadeks selle alusel, millised alad on kõige lähemal etteantud punktidele.

Mõnikord liigitatakse ka kahendotsing „jaga ja valitse“ tüübi alla, kuid erinevus on selles, et kahendotsingu korral töödeldakse edasi ainult üht jagatud osa. Seetõttu on kahendotsingut nimetatud ka „jaga ja vali“ (*divide and choose*) algoritmiks.

Kõige tuntumaks seda tüüpi algoritmiks on põimemeetodil sortimine (*mergesort*). Kuigi seda tüüpi ülesandeid kuigi sageli ette ei tule, on see kasulik paradigma, mis võiks igale võistlejale tuttav olla.

On antud kuni miljard väikest, sarnase suurusega reaalarvu. Leida nende summa.

Esimesena tuleb muidugi pähe arvud lihtsalt järjest kokku liita, kuid eelmisest peatükist on ehk meeles täpsuse probleemid ujukomaarvude liitmisel. See tähendab, et kui liidame viimaseid arve, on summa juba kasvanud nii suureks, et nende viimaste arvude tüvekohad pole enam olulised ning tegelikult need arvud jäävad summale lisamata. Selle probleemi vältimiseks saab kasutada paarikaupa liitmist (*pairwise summation*), mis on justnimelt „Jaga ja valitse“-tüüpi algoritm:

```
long double* liidetavad;

long double liida(int start, int end)
{
      if (end == start) //ainult üks arv
              return liidetavad[start]; //tagastame selle
       int keskkoht = (start + end) / 2; //muidu jagame ülesande pooleks
       //ja tagastame poolte summa summa
      return (liida(start, keskkoht) + liida(keskkoht + 1, end));
}
```

Tegemist on väga lihtsa rekursiivse funktsiooniga. Liidetavad jagatakse iga kord kaheks ja leitakse kummagi poole summa eraldi. Baasjuhuks on see, kui järel on vaid üks arv, sel juhul tagastatakse see arv. Muudel juhtudel leitakse antud alamjada kummagi poole summa eraldi ning liidetakse need kokku. Rekursioonivalemina avaldub see kujul:

$$f(s_{1,}\dots,s_n)=\begin{cases} s_{1,}, & n=1 \\ f(s_{1,}\dots,s_{n/2}) + f(s_{n/2+1,}\dots,s_n), & n>1 \end{cases}$$

Sellisel moel liitmine on täpsem, kuid samas ka palju aeglasem.

Üheks võimaluseks seda algoritmi kiirendada ilma täpsuses oluliselt kaotamata, on rekursioonibaasi muutmine. Kuna väheste sarnaste arvude liitmisel veel tõsist probleemi täpsusel ei teki, võetakse baasjuhuks mingi teatud pikkusega jada, mis siis tavapäraselt liidetakse. Järgmises funktsioonis on võetud baasjuhu liidetavate arvuks 1000:

$$f(s_{1,}\dots,s_n)=\begin{cases}\sum_{i=1}^{n} s_i, & n\le 1000\\ f(s_{1,}\dots,s_{n/2}) + f(s_{n/2+1,}\dots,s_n), & n>1000\end{cases}$$

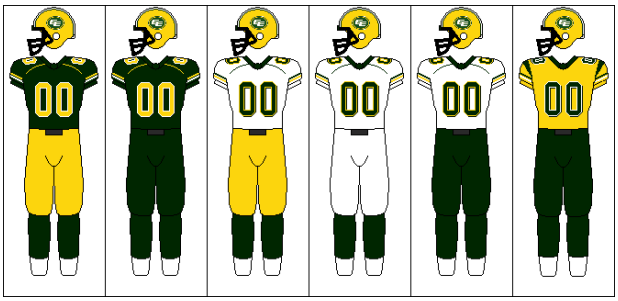
```
long double liidaKiirem(int start, int end)
{
	if (end - start < 1000) { //liita pole rohkem kui 1000 arvu, liidame tavaliselt
		long double vas = 0;
		for (int i = start; i <= end; i++){
			vas += liidetavad[i];
		}
		return vas;
	}
	//liita on rohkem arve, jagame ülesande pooleks
	 int keskkoht = (start + end) / 2;
	return (liidaKiirem(start, keskkoht) + liidaKiirem(keskkoht + 1, end));
}
```

Tulemused erinevate meetoditega liitmisel:

Meetod	Tulemus
liida	5000428.6446836758
liidaKiirem	5000428.6446836758
Tavaline järjest liitmine	5000428.6446819436

2.7 SISSEJUHATUS KOMBINATOORIKASSE

2.7.1 Permutatsioonid



Permutatsioonid on teatava põhihulga kõikidest elementidest moodustatud erinevad järjestatud hulgad. Näiteks hulga $\{a,b,c\}$ permutatsioonid on $\{a,b,c\}$, $\{a,c,b\}$, $\{b,a,c\}$, $\{b\,,c,a\}$, $\{c,a,b\}$ ja $\{c,b\,,a\}$. Elemendid hulgas on kõik alati samad, oluline on nende järjestus. Vaadeldud kolmeelemendilisel hulgal oli kuus permutatsiooni: esimesele kohale saime valida ükskõik millise kolmest elemendist, teisele kohale ükskõik kumma ülejäänud kahest elemendist ja kolmandale kohale jäi kolmas element. Kui n-elemendilise hulga kõik elemendid on erinevad, saab esimesele kohale valida elemendi kõigi n elemendi seast, teisele kohale $n-1$ elemendi seast jne, kuni viimasele kohale jääb ainus valimata element. Nii on permutatsioone kokku

$$n\cdot(n-1)\cdot\ldots\cdot 1 = n!$$

Kui aga hulgas on korduvaid elemente, siis on erinevaid permutatsioone vähem, nt hulgal $\{a,a,b\}$ on vaid 3 permutatsiooni: $\{a,a,b\}$, $\{a,b,a\}$ ja $\{b,a,a\}$. Permutatsioonide arv avaldub valemiga

$$\frac{n!}{m_1!\,m_2!\,\ldots m_k!}$$

kus k on erinevate elementide arv ja m_i on i. elemendi korduste arv.

2.7.2 Permutatsioonide konstrueerimine

Ülesandeid, kus tuleb konstrueerida erinevaid permutatsioone, tuleb programmeerimisvõistlustel üsna sageli ette. Järgmiseks üks lihtne näiteülesanne:

Elle soovib enda ja oma klassikaaslaste eesnimedest lõbusaid anagramme koostada. Aita teda ja kirjuta programm, mis leiab etteantud nimest kõik erinevad anagrammid.
NÄIDE:
4
elle
Vastus:
eell
elel
elle
leel
lele
llee

Üks võimalus on luua permutatsioonid rekursiivselt.

```
void LeiaKoikVoimalused(int asukoht)
{
    if (asukoht == pikkus) {
        for (int i = 0; i < pikkus; i++) {
            cout << vastus[i];
        }
        cout << endl;
        return;
    }
    for (int i = 0; i < pikkus; i++) {
        if (kasVoetud[i] ||
            (i > 0 && !kasVoetud[i - 1] && esialgne[i - 1] == esialgne[i])) continue;
        kasVoetud[i] = 1;
        vastus[asukoht] = esialgne[i];
        LeiaKoikVoimalused(asukoht + 1);
        kasVoetud[i] = 0;
    }
}
```

C++ keeles on olemas standardteegis funktsioon järgmise permutatsiooni leidmiseks. See on defineeritud päisfailis `<algorithm>` ja kannab nime `next_permutation`. Järgmine näitekood kasutab seda funktsiooni:

```
sort(vastus, vastus + pikkus);
bool veelPermutatsioone = 1;
while (veelPermutatsioone) {
    for (int i = 0; i < pikkus; i++) {
        cout << vastus[i];
    }
    cout << endl;
    veelPermutatsioone = next_permutation(vastus, vastus + pikkus);
}
```

Pythonis on `itertools` moodulis funktsioon `permutations`, mis koostab järjest kõik permutatsioonid. Peamine erinevus C++ funktsiooniga on aga selles, et Python arvestab listi kõik elemendid erinevateks. Üheks võimaluseks samasugustest elementidest lahti saada on moodustada hulk ehk `set`:

```
import itertools
import sys
str = sys.stdin.readline().rstrip()
for p in set(itertools.permutations(list(str))):
    print("".join(p))
```

Javas sellist toredat funktsiooni valmiskujul pole, kuid vaatame, kuidas sarnast funktsiooni ise kirjutada ja seda ilma rekursioonita.

Esimeseks eeltingimuseks, et me saame üldse järgmist permutatsiooni leida, on see, et iga suvalise permutatsiooni korral on üheselt määratud, milline on järgmine permutatsioon. Kõige lihtsamalt tagab selle, kui üksikud elemendid, millest permutatsioone moodustame, on järjestatavad ja omavahel võrreldavad. Tähed ja arvud on järjestatavad, kuid tegelikult annab pea igasuguse hulga elementidele vajadusel mingi järjestuse määrata.

Algoritm ise on järgmine:

1. Vaatame antud sõna paremalt vasakule ja leiame esimese märgi x, mis on väiksem kui eelmine.
2. Liigume tagasi paremale ning leiame kõige väiksema elemendi y, mis on suurem kui x.
3. Vahetame need elemendid.
4. Keerame elemendid alates x-le järgnevast kuni lõpuni vastupidisesse järjekorda (st kui nad punkt 1 järgi pidid enne kahanema, siis pärast keeramist on need kasvavas järjekorras).
5. Kui kõik elemendid on tagurpidises järjekorras, siis see on nii-öelda viimane permutatsioon, tagastame järgmiseks esimese ehk siis lihtsalt ümberpööratud jada.

```
string jargmine_perm(string s, int pikkus)
{
    char* vastus = new char[pikkus];
    for (int i = 0; i < pikkus; i++) {
        vastus[i] = s[i];
    }

    for (int i = pikkus - 2; i >= 0; i--) {
        //võrdleme, millise indeksini on tagumine osa kahanevas järjekorras
        if (vastus[i] < vastus[i + 1]) {
            //läheme tagasi täheni, mis on <= kui vaadeldav, asendame 1 enne seda
            int j = pikkus;
            while (!(vastus[i] < vastus[--j]));
            //vahetame i ja j
            char tmp = vastus[j];
            vastus[j] = vastus[i];
            vastus[i] = tmp;
            //keerame jada saba pärast i-d ümber
            reverse(vastus + i + 1, vastus + pikkus);
            string str(vastus, pikkus);
            return str;
            }
        }
        //Kui kõik on algusest kahanevas järjekorras, keerame kogu jada ümber
        reverse(vastus, vastus + pikkus);
        string str(vastus, pikkus);
        return str;
}
```

2.7.3 Kombinatsioonid

Hulga A **alamhulgaks** nimetatakse hulka B, mille kõik elemendid kuuluvad antud hulka A.

Hulga teatud arvust elementidest koosnevaid erinevate elementidega alamhulki nimetatakse **kombinatsioonideks**. Näiteks hulga $\{a, b, c\}$ 2-elemendilised kombinatsioonid on $\{a, b\}$, $\{a, c\}$ ja $\{b, c\}$. Elementide järjestus hulgas ei ole oluline. Igal hulgal on täpselt üks 0-elemendiline kombinatsioon: tühi hulk { } ning üks n-elemendiline kombinatsioon – hulk ise. n-elemendilise hulga kõigi m-elemendiliste kombinatsioonide arv avaldub valemiga:

$$C_n^m = \binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

n-elemendilise hulga kõigi võimalike ($0 \ldots n$-elemendiliste) kombinatsioonide arv on võrdne 2^n -ga.

Kõikide alamhulkade genereerimine on algoritmi poolest lihtne: iga elemendi puhul on kaks võimalust – element kuulub alamhulka või mitte:

```
void kombinatsioonid(int i)
{
    if (i == n) {
        for (int j = 0; j < n; j++) {
            if (lisan[j]) {
                cout << arvud[j] << " ";
            }
        }
        cout << endl;
        return;
    }
    lisan[i] = 0;
    kombinatsioonid(i + 1);
    lisan[i] = 1;
    kombinatsioonid(i + 1);
}
```

See on oma olemuselt väga sarnane paroolide ülesandele selle peatüki alguses. Samamoodi, nagu paroolиülesande sai lahendada, kasutades kahendarvu bitivektorina, nii saab seda teha ka alamhulkade konstrueerimisel: 0 tähistab tühja alamhulka, 2^n-1 hulka ennast ning muidu iga biti korral kahendarvus vaatame, kas see on 0 (vastava positsiooniga element ei kuulu hulka) või 1 (kuulub hulka):

```
int n2 = 1 << n;
for (int i = 0; i < n2; i++) {
    int loendur = 0;
    int i2 = i;
    while (i2 > 0) {
        if (i2 % 2 == 1) {
            cout << arvud[loendur] << " ";
        }
        loendur++;
        i2 /= 2;
    }
    cout << endl;
}
```

Järgmise ülesande lahendamisel saab kirjeldatud võtteid kasutada:

Kuna palju inimesi jätab oma väljaostetud lennukipiletid kasutamata, on lennufirmadel tavaks müüa välja üle 100% lennukis olevatest kohtadest. Kui siiski rohkem reisijaid kohale tuleb, makstakse neile kompensatsiooni. Lennufirmal „Hello Kitty“ (www.evakitty.com) on sageli vaja otsustada, millised reisijad lennule pääsevad ja kes peab järgmist lendu ootama. Lisaks kohtade arvule tuleb lennul arvestada ka reisijate kogukaaluga, mis ei tohi ületada etteantud normi. Iga reisija on maksnud pileti eest erinevat hinda ning kuna lennufirma peab mahajääjatele piletiraha kompenseerima, eelistatakse kallima pileti ostnud kliente. Koosta programm, mis aitab lennufirmal otsustada, millised reisijad antud lennule paigutada, nii et reisijate kogumass ei ületa lubatud piiri ja makstavad kompensatsioonid oleksid minimaalsed.
Esimesel real on antud lennuki kohtade arv k ja lennukile lubatavate reisijate maksimaalne mass r. Teisel real on lennule soovivate reisijate arv n. Järgneb n rida, kus igal real on ühe reisija makstud piletihind p ja tema mass m. Väljasta esimesele reale reisijatele tagasimakstav summa ja järgmisele reale lennule pääsevate reisijate järjekorranumbrid (sisendile vastavas järjekorras).
NÄIDE:

```
4 400
7
200 300
150 100
150 60
50 20
100 150
170 120
150 40
```

Vastus:

```
350
2 3 6 7
```

Selles ülesandes on vaja leida parim võimalik kombinatsioon. Üheks võimaluseks on kõik kombinatsioonid läbi proovida ja valida neist parim. Sarnaselt DNA ülesandele on ka siin hulk piiranguid, millega arvestada. Piirangud annavad võimaluse tagurdusmeetodi kasutamiseks, kus saab sobimatud harud kohe ära lõigata.

Peamine tingimustes toodud piirang on kaalupiirang. Kui raskemaid inimesi töödelda kombinatsioonis eespool, saab kombinatsiooni kiiremini ülekaalu tõttu välistada. Näiteks kui meil on reisija, kelle mass üksi lubatud piiri ületab, siis teda esimesena käsitledes saame kõik seda reisijat sisaldavad kombinatsioonid välja jätta. Kui aga lisame enne kergemad ja siis lõpuks selle ülekaaluka tegelase, teeme tühja tööd. Seetõttu on hea mõte sisendandmed sorteerida, antud näites siis kaalu järgi.

Siin on rekursiivne funktsioon parima paigutuse leidmiseks. Algne väljakutse on `LeiaVastus(0, 0, 0, 0)`

```
void LeiaVastus(int asukoht, int kaal, int reisijad, int hind)
{
    if (asukoht == n) { //kõik on läbi vaadatud
        if (hind > koguHind) {//ja on leitud parema hinnaga kombinatsioon
            koguHind = hind; //salvestame parema koguhinna
            for (int i = 0; i < n; i++) {
                vastus[i] = vaheVastus[i]; //ja lennule pääsevad reisijad
            }
        }
        return;
    }
    //kõik ei ole veel vaadatud
    //kui on veel vabu kohti ja vaba //kaalu
    if (reisijad < k && kaal + kaalud[asukoht] <= v) {
        vaheVastus[asukoht] = 1; //lisame reisija
        LeiaVastus(asukoht + 1, //ja vaatame edasi järgmist reisijat
                 kaal + kaalud[asukoht], //uuendatud kogukaalu,
                 reisijad + 1, //reisijate arvu
                 hind + hinnad[asukoht]); //ja hinnaga.
    }
    vaheVastus[asukoht] = 0; //jätame selle reisija lisamata
    LeiaVastus(asukoht + 1, kaal, reisijad, hind); //ja proovime järgmist reisijat.
    //Kuna eelmist ei lisanud, siis kaal, reisijate arv ja hind ei muutu
}
```

2.8 KONTROLLÜLESANDED

2.8.1 Autoturg

Autoturul on müügil N erineva hinnaga autot ($1 \le N \le 50000$) ja iga päev tuleb turule M ostjat ($1 \le M \le 25000$). Igal ostjal on mõttes, kui kallist autot ta osta tahab. Leida iga ostja jaoks talle sobivaima hinnaga auto. Sobivuse kriteeriumiks on müüdava auto hinna ja ostja soovitud hinna vahe absoluutväärtus. Kui müügil on kaks autot, mille hind erineb soovitud hinnast samal määral, siis valib ostja odavama auto. Kui üks ostja on auto välja valinud, jääb see siiski turule alles ja järgmised ostjad saavad samuti seda autot valida.

Sisendi esimesel real on kaks positiivset täisarvu: N ja M. Järgmisel N real on autode hinnad, järjestatuna odavaimast kalleimani. Lõpuks on M real ostjate soovitud hinnad.

Väljundisse kirjutada ostjate järjekorras neile enim sobivate autode hinnad.

NÄIDE:

```
4 4
1 4 5 7
10 4 6 8
```

Vastus: `7 4 5 7`



2.8.2 Kaupmees ja maksukoguja

Vanal hallil ajal, enne e-riiki, käibedeklaratsioone ja e-maksuametit, käisid kaupmeeste juures maksukogujad, kes vaatasid, kui palju kaupmees on tulu saanud ning määrasid selle põhjal maksu. Kaupmees Humbertil on laev, mida ta saab igal kuul saata kas ostu- või müügireisile, esimesega kaasneb kulu, teisega tulu. Samas külastab Humbertit vahel maksukoguja, kes tahab alati näha käesoleva aasta viimase viie kuu (jaanuar-mai, veebruar-juuni jne) kokkuvõtet, et selle põhjal maksu määrata. Enne maid maksukoguja ei käi, sest viis kuud pole veel täis. Humbert tahab maksukogujale alati näidata, et ta on kahjumis, kuid tegelikult raha teenida. Kuidas ta peaks oma kaubareisid selleks ajastama ja kui palju kasumit on tal võimalik saada?

Sisendis on kaks täisarvu: müügireisi tulu ning ostureisi kulu. Väljundisse kirjutada parim finantstulemus, mis Humbertil on võimalik aastaga saavutada, kui kõik viiekuised lõigud peavad eraldi võttes olema kahjumlikud.



(Pildil Marinus van Reymerswaele maal „Maksukogujad“, 16.saj.)

NÄIDE 1:

```
59  237
```

Vastus: 116 (osta mais ja oktoobris, kõigil teistel kuudel müüa)

NÄIDE 2:

```
375  743
```

Vastus: 28

NÄIDE 3:

```
2500000  8000000
```

Vastus: `-12000000`

2.8.3 Lippude vasturünnak

Malelauale on asetatud kaheksa lippu, igaüks neist erineval real. Seega ei saa lipud üksteist horisontaalselt rünnata, kuid saavad seda teha vertikaalselt või diagonaalselt. Eesmärgiks on saavutada olukord, kus ükski lipp teisi ei ründa. Selleks võime laual olevaid lippusid liigutada, aga ainult mööda horisontaale. Leida minimaalne arv lippe, mida on vaja eesmärgi saavutamiseks paigast nihutada.

Sisendi ainsal real on kaheksa täisarvu: lippude asukohad oma ridadel, järjestatud ridade kaupa. Väljundisse kirjutada täpselt üks arv: mitu lippu on vaja paigast nihutada.

NÄIDE 1:

```
1 2 3 4 5 6 7 8
```

Vastus: 7

NÄIDE 2:

```
1 5 2 1 3 7 4 8
```

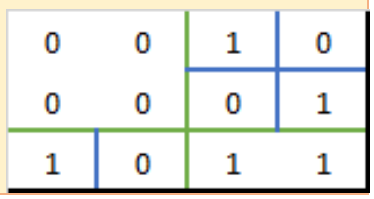
Vastus: 1 (piisab neljanda lipu liigutamisest kuuendale ruudule).

2.8.4 Akulaadija

Ahto telefonil on laetav aku, millel on järgmine omadus: iga kord, kui aku saab täiesti tühjaks, väheneb selle mahutavus ühe ühiku võrra. Ahto teeb iga päev mingi hulga telefonikõnesid ja laadib õhtul aku ära, kuni võimaliku maksimumini. Iga telefonikõne kulutab aku laengut ühe ühiku võrra. Antud on telefonikõnede arv päevade kaupa. Leida, mis võis olla aku esialgne minimaalne mahutavus.
Sisendi esimesel real on päevade arv `N` (`1≤N≤100000`). Teisel real on `N` täisarvu lõigust `1-107`, mis tähistavad kõnede arvu.
Väljundisse kirjutada üks täisarv: aku minimaalne mahutavus protsessi alguses.
NÄIDE 1:
```
5
1 5 1 4 2
```
Vastus: `5` (teisel päeval saab tühjaks ja edasi on mahutavus 4).
NÄIDE 2:
```
3
6 7 7
```
Vastus: 8

2.8.5 Pildi pakkimine

JPEG formaadis piltide pakkimine töötab lihtsustatult järgmisel põhimõttel:
1. Pilt jagatakse ruutudeks
2. Kui mingi ruut on üleni ühte värvi, kirjutataksegi faili, et seal on seda värvi ruut
3. Kui ruut koosneb erinevatest värvidest, jagatakse ta uuesti väiksemateks ruutudeks ja korratakse sammu 2.
Mingil hetkel võime otsustada, et käesolev ruut on „enamvähem“ ühevärviline ja seda mitte edasi jagada. Sellest on tingitud ka silmaga eristatavad ruudud halvema kvaliteediga (s.t kõrgema pakkimisastmega) fotodel.
Praeguses ülesandes vaatame veel lihtsustatud varianti, kus on ainult kaks värvi, valge on `1` ja must on `0`. Antud on pakkimata pilt, ülesandeks on see pakkida. Reeglid on järgmised:
1. Kui vaadeldav ristkülik on üleni sama värvi, kirjutada see värv väljundisse.
2. Kui ristkülik on sisaldab erinevaid värve, kirjutada väljundisse `D`, jagada ristkülik neljaks ja korrata protseduuri saadud osade jaoks. Osade järjestus on vasak ülemine, parem ülemine, vasak alumine, parem alumine. Kui ristkülikul on paaritu arv ridu, tuleb ülemised osad teha ühe võrra kõrgemad kui alumised. Kui ristkülikul on paaritu arv veerge, tuleb vasakpoolsed osad teha ühe võrra laiemad kui parempoolsed.
3. Kui ristkülik on tühi (0 rea või veeruga), siis ignoreeri.
Sisendi esimesel real on kaks arvu `K` ja `L` (`1≤K,L≤200`), mis tähistavad vastavalt pildi kõrgust ja laiust. Teisel real on string, mis koosneb `KxL` märgist ja kus on ridade kaupa pildi pikslid.
NÄIDE:
```
3 4
001000011011
```
Vastus: `D0D1001D101` (Esimene `D` tähendab, et kogu ristkülik tuleb ära jagada, saadavad alamristkülikud genereerivad vastavalt järjendid `0`, `D1001`, `D10` ja `1`).

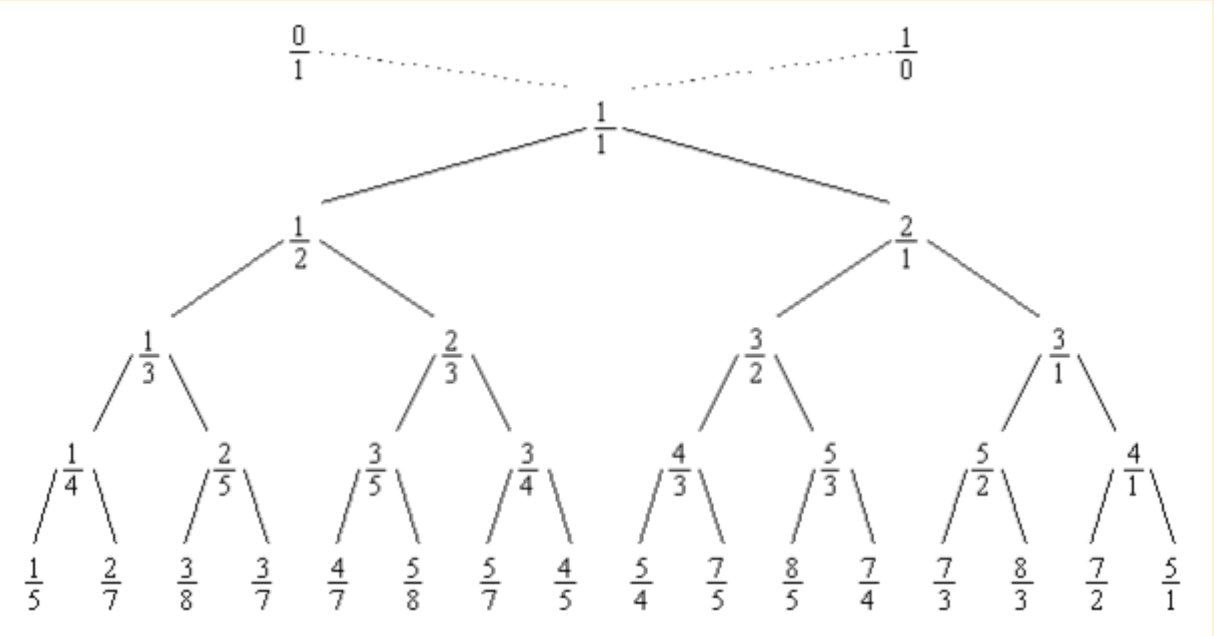


2.8.6 Stern-Brocot' puu

19. sajandi keskpaiku avastasid Saksa matemaatik Moritz Stern ja prantsuse kellassepp Achille Brocot teineteisest sõltumatult elegantse ratsionaalarvude genereerimise viisi. Meetodit nimetatakse tänapäeval Stern-Brocot' puuks. Puu konstrueeritakse järgmiselt:

- Alustame murdudest $\frac{0}{1}$ ja $\frac{1}{0}$
- Iga murrupaari $\frac{m}{n}$ ja $\frac{m'}{n'}$ vahele kirjutame murru $\frac{m+m'}{n+n'}$.

Tulemuseks on joonisel näidatud puu:



Näiteks $\frac{4}{3}$ on saadud murdudest $\frac{1}{1}$ ja $\frac{3}{2}$. Äärmiste väärtuste jaoks tuleb appi võtta esialgsed $\frac{0}{1}$ ja $\frac{1}{0}$, näiteks $\frac{4}{1}$ saadakse $\frac{3}{1}$ ja $\frac{1}{0}$ abil. Matemaatiliselt saab kergesti näidata, et kõik saadud murrud on taandumatud ja erinevad ning lõputu puu sisaldab parajasti kõiki positiivseid ratsionaalarve.

Viimase omaduse abil saame iga ratsionaalarvu esitada tema asukohana Stern-Brocot' puus, pannes kirja, kas mingis harus tuleb minna vasakule (L) või paremale (R).

Teatavasti saab iga ratsionaalarvu esitada taandatud lihtmurruna. Sisendis on antud kaks positiivset täisarvu, mis tähistavad otsitava arvu lihtmurruna esituse lugejat ja nimetajat. Väljundisse kirjutada string, mis vastab arvu asukohale Stern-Brocot' puus.

NÄIDE 1:

5 7

Vastus: LRRL

NÄIDE 2:

878 323

Vastus: RRLRRLRLLLLRLRRR

2.8.7 Viinamarjad

Maailma põhjapoolseim riik (antud juhul defineeritud kui riik, mis ulatub kõige vähem lõunasse), kus saab vabas õhus viinamarju kasvatada, on Läti. Heno soovib pensionipõlveks osta Lätis põllumaad ja hakata seal viinamarju kasvatama. Pakkuda on maa künka küljel, kus pind tõuseb läänest itta ning lõunast põhja. Heno on teinud hulga arvutusi tuule suuna, päikesepaiste nurga, mulla kvaliteedi ning veel paarisaja muu pisiasja alusel ning leidnud, millised viinamarjasordid sobivad millisel kõrgusel kasvatamiseks. Iga sordi kohta on teada, mis on selle jaoks sobiv minimaalne ja maksimaalne kõrgus. Lisaks on ta geomeetriline perfektsionist ning soovib kindlasti ruudukujulist maatükki. Ülesandeks on leida maksimaalse suurusega maatükk, mille Heno võiks osta.

Sisendi esimesel real on kolm täisarvu: müügil oleva maa laius `N` ja pikkus `M` ($1 \le M, N \le 500$) ning viinamarjasortide arv `Q` ($1 \le Q \le 10000$). Järgmistel `N` real on igaühel `M` positiivset täisarvu, mis tähistavad maapinna kõrgust selles asukohas. Kõrgus on igas reas alati vasakult paremale mittekahanev ning igas veerus alt üles mittekahanev. Lõpuks on `Q` real igaühel 2 täisarvu, mis tähistavad vastava viinamarjasordi minimaalset ja maksimaalset võimalikku kasvukõrgust.

Väljundisse kirjutada `Q` täisarvu: maksimaalse pindalaga ruudukujulise maatüki küljepikkus, mis on võimalik müügil olevast maast välja lõigata ja mille kõik ruudud vastavad tingimustele.

NÄIDE 1:

```
4 5 3
23 51 66 83 93
16 33 33 45 50
16 21 33 35 35
13 21 25 33 34
22 90
33 35
20 100
```

Vastus:

```
3
2
4
```

23	51	66	83	93
16	33	33	45	50
16	21	33	35	35
13	21	25	33	34

23	51	66	83	93
16	33	33	45	50
16	21	33	35	35
13	21	25	33	34

23	51	66	83	93
16	33	33	45	50
16	21	33	35	35
13	21	25	33	34

NÄIDE 2:

```
4 4 4
11 19 30 41
7 10 15 17
5 8 10 12
1 7 9 11
6 20
7 9
10 10
13 14
```

Vastus:

```
3
1
1
0
```



Sabile viinamarjaistandus Lätis

2.8.8 Erinevad summad

Antud on hulk arve ja summa, mis neid liites tuleb kokku saada. Leida kõik võimalused summa saavutamiseks. Kui arvuhulk on näiteks [4, 3, 2, 2, 1, 1] ja summa on 4, siis on kolm võimalust selle saavutamiseks: 3+1, 2+2 ning 2+1+1.
Sisendi esimesel real on kaks positiivset täisarvu: `S` (`1≤S≤1000`), mis tähistab nõutavat summat, ning `N` (`1≤N≤12`), mis tähistab arvude arvu. Teisel real on `N` positiivset täisarvu lõigust `1-100`, mis tähistavad liidetavaid arve.
Väljundisse kirjutada kõik võimalikud lahendused. Lahendused peavad olema sorditud kõigepealt esimese liidetava suuruse järjekorras, siis teise liidetava suuruse järjekorras jne.

NÄIDE 1:

```
4 6
4 3 2 2 1 1
```

Vastus:

```
4
3 1
2 2
2 1 1
```

NÄIDE 2:

```
5 3
2 1 1
```

Vastus: (tühi string)

NÄIDE 3:

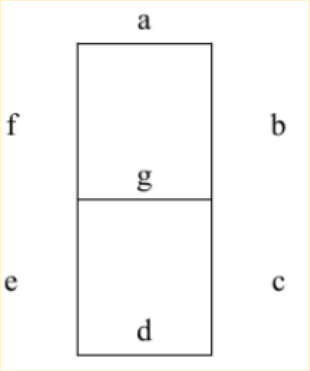
```
400 12
50 50 50 50 50 50 25 25 25 25 25 25
```

Vastus:

```
50 50 50 50 50 50 25 25 25 25
50 50 50 50 50 25 25 25 25 25 25
```

2.8.9 Kodumasinate näidikud

Paljudel kodumasinatel on LED-tuledest koosnev lihtne displei, mis võimaldab näidata erinevaid numbreid. Iga üksiku numbri jaoks on seitse piklikku LEDi, mis asetsevad nagu kõrvaloleval joonisel näidatud. Erinevate numbrite näitamiseks põlevad tuledest a-g järgmised kombinatsioonid (1 on sees, `0` on väljas):




```
0: 1111110
1: 0110000
2: 1101101
3: 1111011
4: 0110011
5: 1011011
6: 1011111
7: 1110000
8: 1111111
9: 1111011
```

Displeid juhib mikrokontroller, millel on funktsioon numbrite allapoole loendamiseks. Sul tuleb kirjutada programm, mis kontrollib põlevate tulede alusel, kas loenduri funktsionaalsus töötab õigesti. Kahjuks on aga testis kasutatavad LEDid ise väga ebakvaliteetsed ja võivad tihti rikki minna ning põlemast lakata, seda nii enne testi kui ka testimise ajal. Kui mõni LED on rikki läinud, siis ta enam korda ei lähe. Seega tuleb pidada testi läbivateks ka juhtusid, kus mõned LEDid on katki läinud. Sisendi esimesel real on näitude arv `N` (`1≤N≤10`). Järgmisel `N` real on igaühel 7-kohaline kahendarv, mis näitab, kas vastavad tuled põlevad või ei. Kui sisend vastab ükskõik millisele järjestikusele alamjadale jadast `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]` (arvestades ka võimalusega, et osad tuled ei tööta), siis kirjutada väljundisse `JAH`. Vastasel korral kirjutada väljundisse `EI`.

NÄIDE 1:

```
2
NNNNNNN
NNNNNNN
```

Vastus: `JAH`

NÄIDE 2:

```
2
YYYYYYY
YYYYYYY
```

Vastus: `EI`

NÄIDE 3:

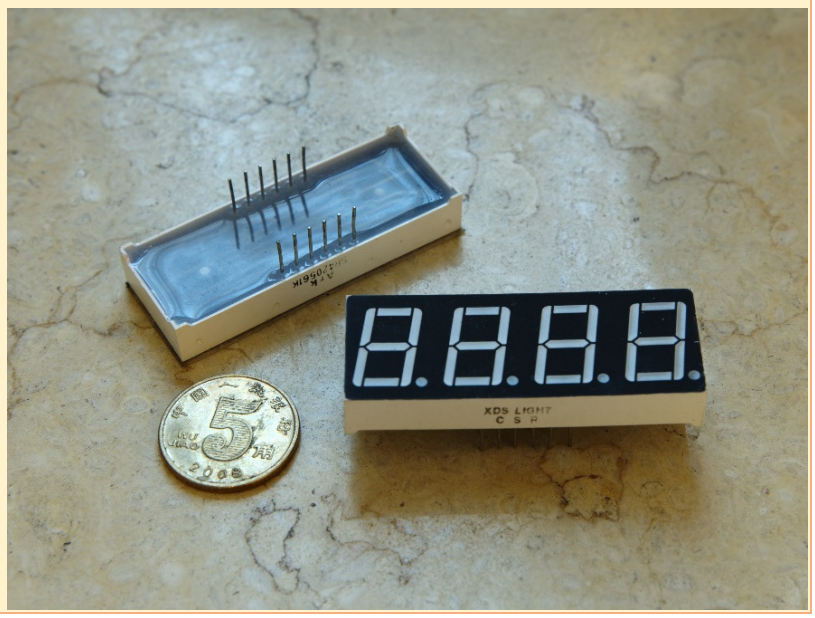
```
3
YNYYYYY
YNYYNYY
NYYNNYY
```

Vastus: `JAH`

NÄIDE 4:

```
3
YNYYYYN
YNYYNYN
NYYNYYN
```

Vastus: `EI`



2.8.10 Graafi värvimine

Graafide värvimine (https://en.wikipedia.org/wiki/Graph_coloring) on graafiteooria valdkond, milles on sõnastatud ja lahendatud palju erinevaid ülesandeid, neist tuntuim on ilmselt nelja-värvi-ülesanne (https://en.wikipedia.org/wiki/Four_color_theorem). Praegusel juhul on tegu lihtsa värvimisega, kus graafi kõik tipud tuleb värvida kas mustaks või valgeks. Tingimuseks on, et mustad tipud ei tohi jääda kõrvuti ning eesmärgiks leida selline värvimine, kus mustade tippude arv on maksimaalne. Joonisel on näide sellisest värvimisest.

Sisendi esimesel real on kaks täisarvu, vastavalt graafi tippude arv `N` (`1≤N≤100`) ja servade arv `M`. Järgmistel `M` real on graafi servade info: igal real kaks täisarvu, mis tähistavad tippude numbreid, mida see serv ühendab.

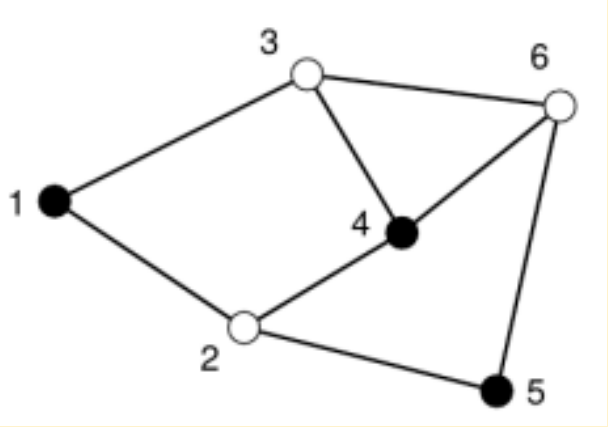
Väljundisse kirjutada kaks rida: esimesele mustaks värvitavate tippude arv, teisele see, millised tipud värvitakse, sordituna väiksemast suuremani. Kui võimalikke lahendusi on mitu, siis väljastada neist minimaalne, s.t selline, kus esimese tipu number on väikseim võimalik, seejärel teise tipu number on väikseim võimalik jne.

NÄIDE (vastab joonisele):

```
6 8
1 2
1 3
2 4
2 5
3 4
3 6
4 6
5 6
```

Vastus:

```
3
1 4 5
```



2.9 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk2 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Fakt.cpp, Fakt.java, Fakt.py	Rekursiivne faktoriaal
DNA.cpp, DNA.java, DNA.py	Tulnukate DNA ahel
Paroolid1.cpp, Paroolid1.java, Paroolid1.py	Paroolid rekursiivselt
Paroolid2.cpp, Paroolid2.java, Paroolid2.py	Paroolid bitivektoriga
Lipud[1-7].cpp, Lipud[1-7].java, Lipud[1-7].py	Lipuülesande erinevad variandid
Kastid.cpp, Kastid.java, Kastid.py	Kolimise ülesanne (vastuse kahendotsing)
Otsing.cpp, Otsing.java, Otsing.py	Kahendotsing rekursiivselt ja iteratiivselt
Ilm.cpp, Ilm.java, Ilm.py	Tuulekülm sadulotsinguga
Liitmine.cpp, Liitmine.java, Liitmine.py	Ujukomaarvude liitmine
Perm.cpp, Perm.java, Perm.py	Permutatsioonid rekursiivselt
JPerm.cpp, JPerm.java, JPerm.py	Järgmise permutatsiooni leidmine
Komb1.cpp, Komb1.java, Komb1.py	Kombinatsioonide leidmine rekursiivselt
Komb2.cpp, Komb2.java, Komb2.py	Kombinatsioonide leidmine bitivektoriga
Lend.cpp, Lend.java, Lend.py	Lennu komplekteerimise ülesanne

3 ALGORITMI KEERUKUS JA PÕHILISED ANDMESTRUKTUURID



Arvutiprogrammid lahendavad mitmesuguseid ülesandeid arvutuste, andmetöötluse ja automaatsete otsuste tegemise alal. Selleks on programmil vaja sooritada hulk tegevusi. Nende tegevuste järjestatud jada nimetatakse **algoritmiks**.

Sõna „algoritm“ ise pärineb Pärsia matemaatiku al-Khwārizmī (780-850) nimest. 825. aasta paiku kirjutas ta araabiakeelse töö, milles kirjeldas arvutamist kümnendsüsteemis. 12. sajandil tõlgiti see ladina keelde pealkirjaga *„Algoritmi de numero Indorum“* ehk „Algoritmi indialaste numbritest“. „Algoritmi“ oli siin lihtsalt al-Khwārizmī nime kohandus ning indialaste numbrid olid need, mida Euroopas hakati tundma araabia numbrite nime all – nulli sisaldavad kümnendsüsteemi numbrid. Edaspidi tähistaski algoritmi termin peamiselt kümnendsüsteemi ja seonduvaid arvutusreegleid.

Tolleaegne arvutusreeglistik (ehk algoritm!) oli esitatud värsivormis ja moodustas mitmesajarealise poeemi. Luulevorm pole aga täpsete ja keeruliste eeskirjade jaoks kõige efektiivsem, seetõttu on hilisemad teadlased uurinud meetodeid algoritmide kiiremaks ja kompaktsemaks esitamiseks.

19. sajandil leiutasid George Boole ja Giuseppe Peano aritmeetika kirjapanemiseks konkreetsete reeglite ja sümbolitega süsteemi, sellest ajast alates võib ka rääkida algoritmidest tänapäevases tähenduses. Arvutiteaduse seisukohast kõige olulisema aluse pani algoritmiteooriale 1930. aastatel Alan Turing, kes kirjeldas **Turingi masina** idee**.** Turingi masin on teoreetiline masin, mille eesmärkideks on:

1. võimalus täita kõiki võimalikke algoritme,
2. võimalikult väike masinasse ehitatud operatsioonide arv.

Tänapäeval ongi arvutussüsteemide hindamise oluliseks kriteeriumiks see, kas süsteemi abil on võimalik luua Turingi masinat – see näitab, et süsteem on üldotstarbeliselt kasutatav igasuguste algoritmide täitmiseks.

3.1 ALGORITMI KEERUKUS

Programmeerimisvõistlustel tuleb välja mõelda ja programmeerida algoritm, mis antud ülesande lahendab. Samas ainult korrektsest algoritmist ei piisa – programm peab vastuse leidma etteantud aja jooksul ning see ei tohi kasutada ka ülemääraselt palju mälu. Nii aja- kui ka mälulimiit on harilikult iga ülesande juures välja toodud. Seetõttu on kasulik enne koodi kirjutama asumist hinnata, kas väljamõeldud lahendus töötab piisavalt kiiresti ega kasuta liiga palju mälu.

Programmi töökiirus sõltub mitmetest teguritest. Mõned neist on vähemalt osaliselt programmeerija kontrolli all, teised mitte. Näiteks ei saa programmeerija üldjuhul muuta arvuti protsessori töökiirust, kuid ta saab valida ülesande lahendamiseks sobivama algoritmi. Seetõttu on programmeerija jaoks oluline, milline on tema valitud algoritmi **ajaline keerukus** (*time complexity*).

Algoritmi ajaline keerukus sõltub enamasti olulisel määral algandmete mahust – näiteks suure faili pakkimine võtab harilikult kauem aega kui väikese faili pakkimine ja miljonielemendilise jada sorteerimine on aeganõudvam kui kümnest elemendist koosneva jada sorteerimine. Algoritmi ajalist keerukust väljendatakse funktsiooniga f, mis seab igale selle algoritmi järgi lahendatavale

konkreetsele ülesandele andmemahuga n vastavusse selle lahendamiseks sooritatavate operatsioonide arvu $f(n)$.

Teades operatsioonide arvu $f(n)$ ja protsessori töökiirust, saab ligikaudu arvutada, kui palju aega konkreetse ülesande lahendamiseks kulub. Erinevate operatsioonide täitmine võib aga võtta erineva hulga aega, seetõttu kasutatakse **elementaaroperatsiooni** mõistet. Elementaaroperatsiooniks nimetatakse sellist operatsiooni, mille arvuti suudab täita konstantse ajaga, sõltumata argumendi väärtusest. Näiteks kuni 64-bitiste arvude liitmine, lahutamine ja korrutamine on kaasaegses arvutis elementaaroperatsioonid, neist pikemate arvudega opereerimine aga tõenäoliselt mitte. Küll aga võib algoritmide omavahelisel võrdlemisel operatsiooni mõistet veelgi lihtsustada või võrreldagi ainult mingit konkreetset tüüpi operatsioonide arvu. Sel juhul saame üldiselt hinnata, kumb algoritm on kiirem, kuid mitte arvutada ligikaudset tööaega.

3.1.1 Keskmine ja halvim keerukus

Mõnel juhul ei sõltu algoritmi ajaline keerukus ainult andmemahust, vaid ka sellest, milline on etteantud andmestik. Näiteks kui otsida jadast mingi konkreetse väärtusega elementi, siis üheks võimalikuks lahenduseks on käia kogu jada elementhaaval läbi. Parimal juhul on otsitava väärtusega element kohe jada esimesel kohal ja pole mingit põhjust teisi elemente enam läbi vaadata, seega piisas vaid ühest võrdluse operatsioonist. Kui aga otsitava väärtusega element on jadas viimane, tuleb kogu jada läbi vaadata ja n-elemendilises jadas tehakse sel juhul n võrdlemist. Viimane kirjeldatud juhtum on selgelt halvim võimalik, sest kunagi ei ole vaja teha ka rohkem kui n võrdlust. Kirjeldatud olukordi nimetatakse vastavalt algoritmi ajaliseks keerukuseks parimal juhul ja halvimal juhul. Praktikas on sageli kasulik teada ka algoritmi keerukust keskmisel juhul ehk operatsioonide arvu, mida tuleb keskmiselt sooritada konkreetse ülesande lahendamiseks andmemahu n korral. Ülalkirjeldatud jadast otsimise korral tuleb keskmiselt teha $n/2$ võrdlusoperatsiooni.

3.1.2 Asümptootiline hinnang

Kuna väikesed andmemahud enamasti niikuinii probleeme ei tekita, on algoritmi töökiiruse hindamisel kõige olulisemaks see, kuidas töökiirus muutub andmemahu piiramatul kasvul. Sellist hinnangut nimetatakse **asümptootiliseks hinnanguks** ning see keskendubki just keerukusfunktsiooni kasvukiiruse uurimisele. Algoritmi keerukusfunktsiooni kasvukiirus näitab, kui kiiresti kasvab selle algoritmi alusel koostatud programmi tööaeg töödeldavate andmete mahu kasvades. Programmeerimisvõistlustel annab see just aimu, kas väljamõeldud algoritm suudab töödelda vajaliku andmemahu etteantud aja jooksul.

Funktsioonide kasvukiiruse võrdlemisel kasutatakse tavaliselt järgmisi seoseid:

- O (tavaline suurtäht O) tähistab, et funktsioon ei kasva kiiremini võrdlusfunktsoonist.
- Ω (kreeka täht oomega) tähistab, et funktsioon ei kasva aeglasemini võrdlusfunkstioonist.
- Θ (kreeka täht teeta) tähistab seda, et funktsioonid on sama kasvuga.

Formaalselt väljendudes $f(n) \in O(g(n))$, kui funktsiooni $f(n)$ kasvukiirus ei ületa funktsiooni $g(n)$ kasvukiirust. See tähendab, et leiduvad positiivsed konstandid c ja n_0, mille korral kehtib

$$\forall n \geq n_0 \colon f(n) \leq cg(n).$$

Sümbol ∀ tähendab „iga“. Kogu avaldis tähendab siis, et funktsioon f alates kohast n_0 ei kasva kiiremini kui võrdlusfunktsioon g. Piiri kasutamine on vajalik seetõttu, et väikeste väärtuste puhul võib algoritmi tööajas esineda moonutusi. Tegevused, mida tuleb teha vaid üks kord sõltumata andmemahust (näiteks muutujatele algväärtuse andmine), võivad võtta väikeste

andmemahtude korral lõviosa algoritmi tööajast. Suurte andmemahtude korral on ühekordsetele tegevustele kuluv aeg täpselt sama, kuid see on tühine, võrreldes kogu tööajaga.

Definitsioon ei piira funktsioonide $f(n)$ ja $g(n)$ enda väärtusi, vaid nende väärtuste suhet $f(n)$ / $g(n)$. Võib öelda ka, et kasv on ülalt tõkestatud. Seetõttu aga võib anda O-hinnanguid ükskõik kui suure liiaga ehk kui $f(n) \in O(n^2)$, siis kehtib ka $f(n) \in O(n^3)$.

$f(n) \in \Omega(g(n))$, kui leiduvad positiivsed konstandid c ja n_0, mille korral kehtib

$$\forall n \geq n_0 : f(n) \geq cg(n).$$

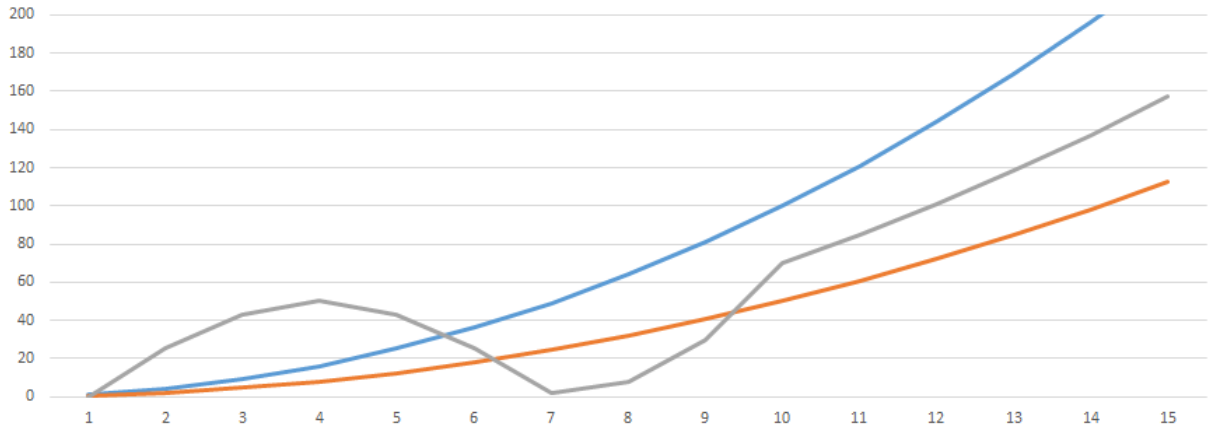
Seekord on funktsiooni f kasv alt tõkestatud funktsiooniga g. Väited $f(n) \in O(g(n))$ ja $g(n) \in \Omega(f(n))$ on samaväärsed.

$f(n) \in \Theta(g(n))$, kui leiduvad positiivsed konstandid c_1, c_2 ja n_0, mille korral kehtib

$$\forall n \geq n_0 : c_1 g(n) < f(n) \leq c_2 g(n).$$

Sellisel juhul on funktsioon f tõkestatud funktsiooniga g nii ülalt kui alt, st kehtib nii $f(n) \in O(g(n))$ ja $f(n) \in \Omega(g(n))$.

Järgneval graafikul on kujutatud sinisega funktsioon $g(n) = n^2$, oranžiga funktsioon $h(n) = 0{,}5 * n^2$ ja halliga uuritav funktsioon $f(n)$, mille keerukust proovime hinnata:



Kuigi funktsioon $f(n)$ kasvab väikeste n väärtuste korral kiiremini kui funktsioon $g(n)$ siis on jooniselt näha, et väärtuse $n = 6$ korral jääb $f(n)$ funktsioonist $g(n)$ maha (ja ei möödu sellest kunagi) ehk iga n väärtuse korral $n \geq 6$ kehtib $f(n) < g(n)$. Seega $f(n) \in O(g(n))$ ehk $f(n) \in O(n^2)$. Analoogselt alates väärtusest $n = 10$ on funktsiooni $f(n)$ väärtus alati suurem kui funktsiooni $h(n) = 0{,}5 * g(n)$ väärtus, seega $f(n) \in \Omega(g(n))$ ehk $f(n) \in \Omega(n^2)$. Seega, kui võtta $n_0 = 10$, on täidetud ka tingimus Θ-hinnanguks:

$$\forall n \geq 10 : 0{,}5n^2 < f(n) \leq n^2 \rightarrow f(n) \in \Theta(n^2).$$

Praktikas kasutatakse sageli O-notatsiooni ehk antakse vaadeldavale algoritmile ülempiiri hinnang, millest rohkem aega ei tohiks selle töö võtta. Isegi juhtudel, kui tegelikult saaks kasutada ka Θ-hinnangut ehk tugevamat väidet, kasutatakse mitteformaalsetes algoritmialastes aruteludes pigem O-d. Seetõttu tuleb erinevate materjalidega töötades olla hoolikas.

3.1.3 Kasvuseoste omadused

Kasvuseostel kehtivad järgmised kasulikud omadused:

1. $\forall c > 0: cf(n) \in \Theta(f(n))$. See tähendab, et funktsiooni korrutamine konstandiga ei muuda selle kasvukiirust. Erijuhul $c = 1$ saame $f(n) \in \Theta(f(n))$, samuti $f(n) \in O\big(f(n)\big)$ ja $f(n) \in \Omega(f(n))$. Kui $f(n) = 1$, siis $c \in \Theta(1)$, mis tähendab, et kõigi konstantsete funktsioonide kasvukiirused võrdsed. See ei tähenda, et kaks programmi töötaksid sama kaua, aga nende tööaeg kasvab samal viisil.
2. Kui $f(n) \in O\big(g(n)\big)$ ja $g(n) \in O\big(h(n)\big)$, siis $f(n) \in O\big(h(n)\big)$. See tähendab, et seos O on transitiivne. Samuti on transitiivsed seosed Ω ja Θ.
3. Kui $f(n) \in \Theta\big(h(n)\big)$ ja $g(n) \in O\big(h(n)\big)$, siis $(f(n) + g(n)) \in \Theta\big(h(n)\big)$. See tähendab, et kahe funktsiooni summa kasvu määrab kiirema kasvuga liidetav. Näiteks kui $f(n) \in \Theta(n^3)$ ja $g(n) \in \Theta(n)$, siis $(f(n) + g(n)) \in \Theta(n^3)$.
4. Kui $f_1(n) \in \Theta\big(g_1(n)\big)$ ja $f_2(n) \in O\big(g_2(n)\big)$, siis $f_1(n)f_2(n) \in O\big(g_1(n)g_2(n)\big)$.
   Kui $f_1(n) \in \Theta\big(g_1(n)\big)$ ja $f_2(n) \in \Omega\big(g_2(n)\big)$, siis $f_1(n)f_2(n) \in \Omega\big(g_1(n)g_2(n)\big)$.
   See tähendab, et kahe funktsiooni korrutise kasv on tegurite kasvude korrutis.
5. Kui $p(n)$ on d-astme polünoom, siis $p(n) \in \Theta(n^d)$.
6. $\forall 0 \leq r \leq s: n^r \in O(n^s)$. See tähendab, et madalama astmega astmefunktsioon ei kasva kiiremini kõrgema astmega astmefunktsioonist. Kehtib ka tugevam väide: madalama astmega funktsioon kasvab alati rangelt aeglasemalt kui kõrgema astmega astmefunktsioon.
7. $\forall k \geq 0, b > 1: n^k \in O(b^n)$. See tähendab, et mitte ükski astmefunktsioon ei kasva kiiremini ühestki eksponentfunktsioonist. Tegelikult kasvab astmefunktsioon alati rangelt aeglasemalt.
8. $\forall k > 0, b > 1: \log_b n \in O(n^k)$. See tähendab, et ükski logaritmfunktsioon ei kasva kiiremini ühestki astmefunktsioonist. Tegelikult kasvab logaritmfunktsioon alati rangelt aeglasemalt.
9. $\forall a > 1, b > 1: \log_a n \in \Theta(\log_b n)$. See tähendab, et kõik logaritmfunktsioonid kasvavad sama kiirusega.
10. $\forall r \geq 0: \sum_{i=1}^{n} i^r \in \Theta(n^{r+1})$. See tähendab, et r. järku astmerea summa kasvab nagu $(r + 1)$. astme polünoom.

3.2 ALGORITMI KEERUKUSE HINDAMINE

Kui algoritm koosneb ainult järjestikustest käskudest, siis kogu algoritmi täitmisaeg on nende järjestikuste käskude täitmisaegade summa. Formaalselt avaldub k järjestikuse käsuga algoritmi täitmisaeg kujul

$$T(n) = f_1(n) + f_2(n) + \cdots + f_k(n),$$

kus $f_i(n)$ on i. käsu täitmisaeg.

3.2.1 Hargnemiste keerukuse hindamine

Kui algoritmis esineb hargnemisi, siis keskmise täitmisaja hindamiseks tuleb iga hargnemise korral arvesse võtta tõenäosust, millega mingit haru täidetakse. Näiteks lihtsa tingimuslause korral kontrollitakse kõigepealt tingimust ja selle tulemusena täidetakse vastavalt kas esimest või teist haru. Sel juhul avaldub täitmisele kuluv keskmine koguaeg järgmiselt:

$$T(n) = f_o(n) + pf_1(n) + (1 - p)f_2(n),$$

kus $f_o(n)$ on tingimuse kontrollimise kulu, $f_1(n)$ ja $f_2(n)$ vastavalt esimese ja teise haru kulu ning p

tõenäosus, et täidetakse esimene haru.
Halvimaks juhuks on see, et täidetakse alati haru, mille täitmine võtab kauem aega. Sel juhul avaldub koguaeg

$$T(n) = f_o(n) + \max\big(f_1(n), f_2(n)\big).$$

3.2.2 Tsüklite keerukuse hindamine

Tsüklitega ehk iteratiivse algoritmi hindamisel tuleb arvestada, et tsükli sisus olevaid käske täidetakse korduvalt. Kui tsüklit täidetakse k korda ja tsükli sisu ühekordse täitmise ajaline keerukus on $f(n)$, siis tsükli täitmise ajaline keerukus on $kf(n)$.

Kui korduste arv k on konstant, siis $kf(n) \in \Theta(f(n))$, ehk algoritmi ajalist keerukust mõjutavad oluliselt ainult tsüklid, mille korduste arv on sõltuvuses andmemahust n. Näiteks kui vaatame tsüklit, mis käib läbi jada pikkusega n ja iga elemendi korral teeb ühe võrdlustehte, mis on konstantse keerukusega, see tähendab $f(n) \in \Theta(1)$, saame tsükli täitmise kuluks $nf(n) \in \Theta(n)$.

3.2.3 Mitmekordsete tsüklite keerukus

Mitmekordsete tsüklite korral saab neid vaadata seestpoolt väljapoole. Kui sisemine tsükkel käib andmed läbi n korda ja on seega keerukusega $t(n) \in \Theta(n)$ ning välimist tsüklit täidetakse samuti n korda, siis välimise tsükli täitmine on keerukusega $nt(n) \in \Theta(n \cdot n) = \Theta(n^2)$.

Päriselt ei koosne kõik tsüklid muidugi täpselt n sammust. Näiteks kui me tahame leida jadast pikkusega n kõikvõimalikud elementide paarid, siis välimine tsükkel käib läbi kõik elemendid, kuid sisemise puhul piisab, kui alustada tsüklit vaadeldavale elemendile järgnevast elemendist. Sellisel juhul käib sisemine tsükkel esimesel korral läbi $n-1$ elementi, teisel $n-2$ jne. Kogukeerukus avaldub siis kujul

$$\begin{aligned} T(n) &= (n-1)f(n) + (n-2)f(n) + \cdots + (n-n)f(n) \\ &= \big((n-1) + (n-2) + \cdots + (n-n)\big)f(n). \end{aligned}$$

Sulgudes on meil tegelikult jada $0, 1 \ldots n-1$. Esitame selle kujul $0 + \sum_{i=1}^{n} i - n$. Kuna $0 \in \Theta(1)$, $\sum_{i=1}^{n} i \in \Theta(n^2)$ (10. omadus) ja $n \in \Theta(n)$. Kui $f(n) \in \Theta(1)$, saame

$$(\Theta(1) + \Theta(n^2) - \Theta(n))\Theta(1) = \Theta(n^2).$$

Kui täpsed valemid pole meeles või ei anna ühtki neist konkreetsel konstruktsioonil kasutada, siis tuleb appi O-hinnang. Nagu eelnevalt mainitud, võib O-hinnanguid anda liiaga. Antud näites

$$\begin{aligned} T(n) &= (n-1)f(n) + (n-2)f(n) + \cdots + (n-n)f(n) \leq \\ &\leq nf(n) + nf(n) + \cdots + nf(n) = n^2 f(n)\,. \end{aligned}$$

Kuna $n^2 f(n) \in \Theta(n^2)$, siis $T(n) \in O(n^2)$.

Rusikareeglina saabki kasutada O-hinnangut, et konstantse sisuga ühekordne tsükkel on keerukusega $O(n)$, kahekordne $O(n^2)$, kolmekordne $O(n^3)$ jne. See on mõistlik küll vaid juhul, kui igas tsüklis tsüklimuutujat muudetakse lineaarselt, see tähendab, et tsükli korduste arv on $k * n + c$, kus k ja c on konstandid. Nt tsüklid päistega

```
for (int i=0; i<2*n; i++)
for (int i=100; i<n+10; i++)
for (int i=0; i<n; i+=2)
```

on kõik keerukusega $O(n)$, kuigi esimene teeb $2n$, teine $n-90$ ja viimane $n/2$ kordust.

Samas tsükkel päisega `for (int i = 1; i <= n; i*= 2)` on hoopiski $O(logn)$ ($\log_2 n$ kordust) ning tsükkel päisega `for (int i=0; i<(n-2)*n; i++)` on ruutkeerukusega $O(n^2)$.

3.2.4 Rekursiooni keerukuse hindamine

Rekursiivse algoritmi täitmisele kuluvat aega hinnatakse rekurrentse võrrandiga. Näiteks „jaga ja valitse“ tüüpi ülesannetel, kus ülesanne jagatakse kaheks võrdseks (andmemahuga $n/2$) alamülesandeks ja mõlemad lahendatakse samal meetodil, siis juhul, kui alamülesannete lahenduste töötlemiseks kulub veel n sammu, avaldub kogu sammude arv järgmisel kujul:

$$T(n) = \begin{cases} 0, kui\ n = 1 \\ 2T\left(\frac{n}{2}\right) + n, kui\ n > 1 \end{cases}$$

Pealtnäha pole sellisest võrrandist suurt abi, kuna selle lahendamine pole just kõige lihtsam, kuid paljudel juhtudel saab sellisel kujul avaldatud lahendusaega hinnata järgmise seose abil:
Kui $T(n) = aT(n/b) + f(n),\ kus\ a \geq 1\ ja\ b > 1$, siis

$$T(n) = \begin{cases} \Theta\big(f(n)\big), f(n) \in \Omega\big(n^{\log_b a+e}\big)\ ja\ af\left(\frac{n}{b}\right) \leq cf(n), \text{mingi konstandi } c \leq 1 \text{ ja suure n korral} \\ \Theta\big(n^{\log_b a} \cdot \log n\big), \quad f(n) \in \Theta(n^{\log_b a}) \\ \Theta\big(n^{\log_b a}\big), \quad f(n) \in O\big(n^{\log_b a-e}\big) \text{ ja } e > 0 \end{cases}$$

Ülalpool toodud näite korral $a = b = 2$, $\log_b a = 1$ ja $f(n) \in \Theta(n) = \Theta(n^{log_b a})$. Nüüd saame teise rea põhjal, et $T(n) \in \Theta(n \log n)$.

Sagedamini esinevate rekursiivsete algoritmide võrrandid on toodud järgnevas tabelis:

T(n)	Keerukus	Näide algoritmist
$\boldsymbol{T(n/a) + \Theta(1)}$	$\Theta(\log n)$	Kahendotsing jadast (a=2)
$\boldsymbol{T(n-1) + \Theta(1)}$	$\Theta(n)$	Faktoriaali leidmine rekursiivselt
$\boldsymbol{bT(n/b) + \Theta(1)}$	$\Theta(n)$	“Jaga ja valitse” ilma valitsemiseta
$\boldsymbol{cT(n/c) + \Theta(n)}$	$\Theta(n \log n)$	“Jaga ja valitse”, nt põimemeetod
$\boldsymbol{T(n-1) + \Theta(n)}$	$\Theta(n^2)$	Kiirmeetodil sorteerimise halvim juht
$\boldsymbol{dT(n-1) + \Theta(1)}$	$\Theta(d^n)$	Hanoi tornid (d=2)

3.3 TAVALISED KEERUKUSKLASSID

Vaatame siin algoritmide tavalisemaid keerukusklasse ja nende esinemise juhtusid.

$O(1)$ - **konstantse** keerukusega on sellised ülesanded, mille lahendamise aeg ei sõltu oluliselt sisendi suurusest.

$O(log\ n)$ – **logaritmilise** keerukusega on harilikult algoritmid, mis jagavad sisendi igal sammul enam-vähem võrdseteks osadeks. Näiteks kahendotsingu algoritm on logaritmilise keerukusega.

$O(n)$ – **lineaarse** keerukusega ülesanded on enamasti sellised, kus sisend käiakse läbi konstantne arv kordi. Lineaarse keerukusega on näiteks suvalise jada kõikide elementide summa leidmine või mingi kindla väärtusega elemendi leidmine sorteerimata jadast.

$O(n\ log\ n)$ - Sellise keerukusega on tavaliselt parimad sortimisalgoritmid ja seega ei saa madalamasse keerukusklassi kuuluda ükski algoritm, mis kasutab võrdlustel põhinevat sortimist.

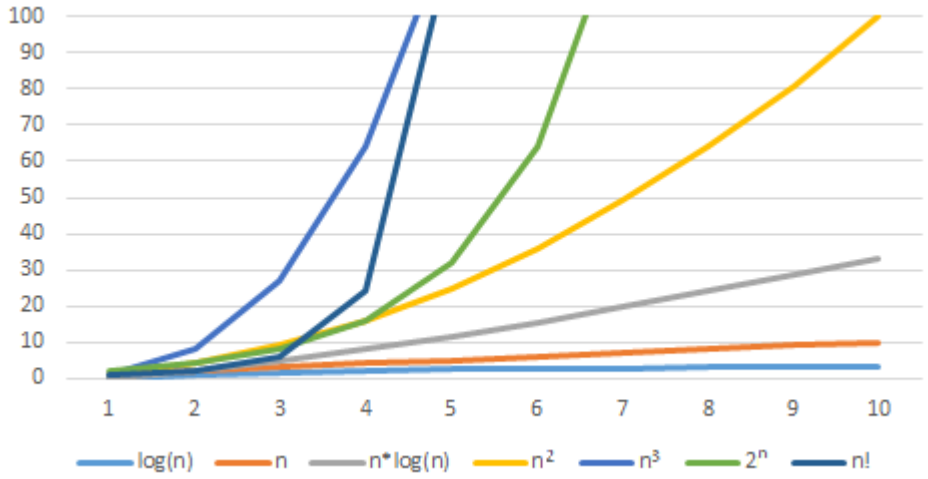
$O(n^2)$ – **ruutkeerukusega** algoritmid on tavaliselt kahekordse tsükliga algoritmid – niimoodi on näiteks võimalik läbi vaadata sisendi elementide kõikvõimalikud paarid.

$O(n^3)$ – **kuupkeerukusega** algoritmid on harilikult kolmekordse tsükliga algoritmid.

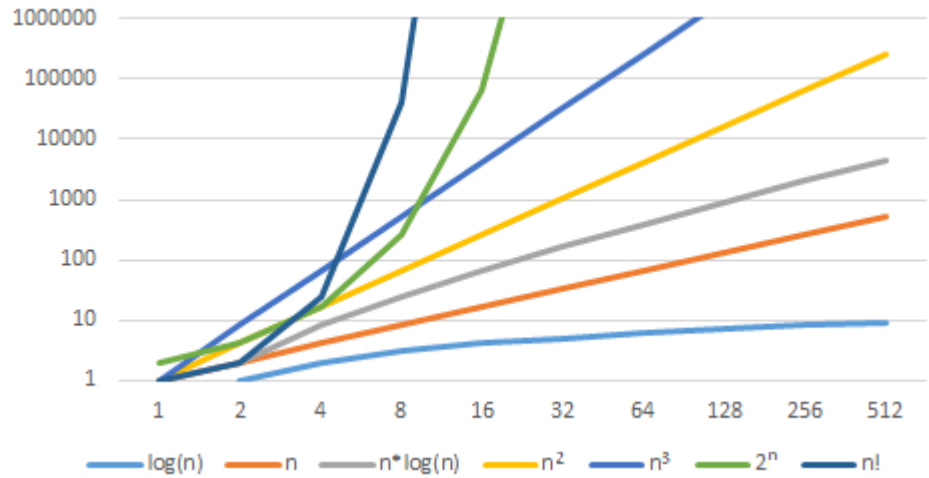
$O(2^n)$ – **eksponentsiaalse** keerukusega algoritmid käivad tavaliselt läbi sisendi kõikvõimalikud alamhulgad.

$O(n!)$ – **faktoriaalse** keerukusega algoritmid enamasti vaatavad läbi kõik sisendi permutatsioonid.

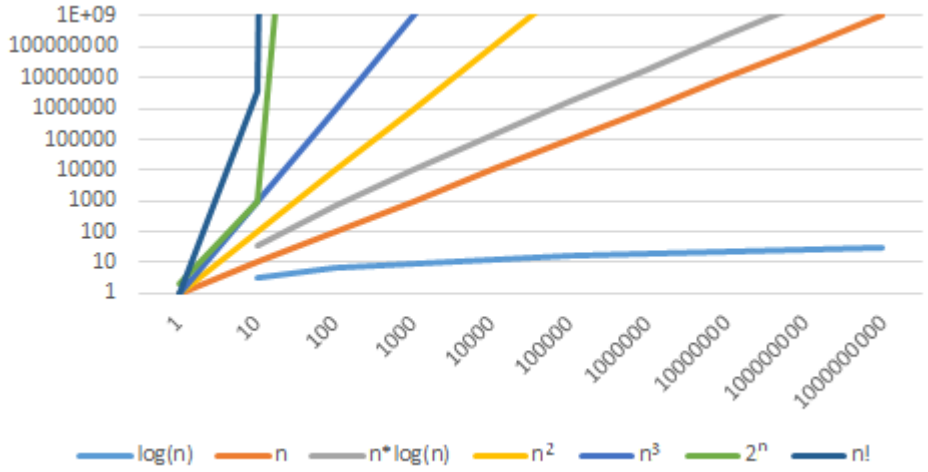
Nii näevad tavaliste keerukusklasside funktsioonid välja graafikul:



Järgmisena on esitatud samad funktsioonid logaritmilisel skaalal:



Veel üks graafik praktikas enam kasutatavate väärtustega:



3.3.1 Keerukusklass ja ajalimiit

Võistlustel on tavaliselt ette antud ajalimiit ning sisendandmete lubatud maht. Selle põhjal saab ennustada, millise keerukusklassiga algoritm lahenduseks sobib. Järgmises tabelis on toodud sisendi suurus ja algoritmi keerukusklass, mis sellise sisendi suudab töödelda ligikaudu sekundiga, kui protsessor suudab sooritada 100 miljonit operatsiooni sekundis:

n	Keerukusklass	Näide
10 – 11	$O(n!), O(n^6)$	Kõigi permutatsioonide leidmine
15 – 18	$O(2^n * n^2)$	Rändkaupmehe ülesanne dünaamilise planeerimisega
18 – 22	$O(2^n * n)$	Dünaamiline planeerimine bitimaskiga
100	$O(n^4)$	
400	$O(n^3)$	Floyd-Warshalli algoritm
2000	$O(n^2 \log n)$	
10000	$O(n^2)$	Aeglased sorteerimisalgoritmid
1000000	$O(n \log n)$	Kiiremad sorteerimisalgoritmid, mitmed “jaga ja valitse” tüüpi ülesanded.
100 000 000*	$O(n), O(\log n), O(1)$	

* enamasti jäävad võistlustel andmemahud miljoni piiresse, kuna suuremate sisendite töötlus võtab liiga palju aega.

3.4 ANDMESTRUKTUURID

Iga mittetriviaalne programm kasutab oma töös küllaltki suurt hulka andmeid. Ilmselgelt pole võimalik iga väärtuse jaoks luua eraldi muutujat, vaid andmed tuleb grupeerida ja korrastada. Juba enne arvutite leiutamist mõtlesid inimesed andmete korrastamiseks välja näiteks loendid, tabelid ja sõnastikud. Sarnased abistavad struktuurid on ehitatud ka enamikus programmeerimiskeeltes. Andmete korrastamine täidab kaht peamist eesmärki:

- Programmi on võimalik lihtsama vaevaga lugeda, mõista ja parandada.
- Õigesti valitud struktuuri korral on andmeid on võimalik kiiresti kätte saada ja muuta.

Andmestruktuur on vahend suure hulga samatüübiliste andmete hoidmiseks ja neile efektiivse juurdepääsu ja töötlemise korraldamiseks.

Andmestruktuurid otseselt ülesandeid ei lahenda, kuid erinevatel struktuuridel töötavad erinevad algoritmid ning seega aitab õige andmestruktuuri valik kaasa efektiivse lahenduse leidmisele ja kasutusele.

3.4.1 Andmestruktuuride klassifitseerimine

Andmestruktuure võib jagada **konkreetseteks** ja **abstraktseteks**. Konkreetsed andmestruktuurid kirjeldavad andmete tegelikku paigutust arvutis, sellised on näiteks massiiv ja ahel. Abstraktsed andmestruktuurid kirjeldavad pigem võimalusi, mida selle andmestruktuuriga teha saab, samas võib andmeid nende sees esitada mitmel viisil. Abstraktsed andmestruktuurid on näiteks pinu ja järjekord.

Sageli võib abstraktse andmestruktuuri luua nii, et ta kasutab sisemiselt mõnd lihtsamat konkreetset andmestruktuuri.

Andmestruktuurid võivad olla kas **staatilised** (s.t nende suurus pannakse alguses paika ja neisse ei saa uusi elemente lisada või neid eemaldada) või **dünaamilised** (saab elemente lisada ja eemaldada).

3.5 MASSIIV

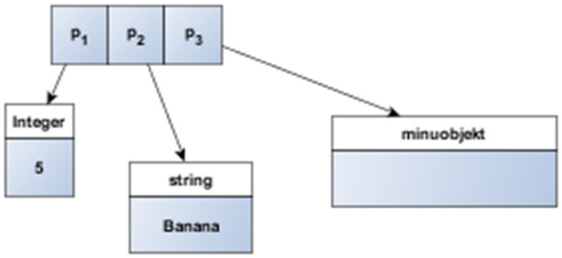


Massiiv (*array*) on andmestruktuur, mis koosneb indekseeritud ühetüübilistest elementidest. See tähendab, et igal elemendil massiivis on kindel järjenumber ehk indeks. Elementide arvu massiivis nimetatakse massiivi pikkuseks. Lühidalt oli massiividest juttu juba esimeses peatükis, kuna enamasti on programmeerimiskeeltes massiivi jaoks olemas lausa omaette andmetüüp.

Esimeses peatükis oli ka juttu, kuidas massiive mälus hoitakse. Harilikult antakse massiivi deklareerimisel ette selle pikkus ja elementide andmetüüp. Selle info põhjal reserveeritakse massiivi jaoks vajaliku pikkusega mäluplokk. Teades esiteks seda, millisel mäluaadressil massiiv algab, ja teiseks elemendi andmetüübi pikkust, on lihtne arvutada, kust algab 2. element, kust 3. jne. Tänu sellele on indeksile vastava elemendi leidmine massiivist kiire ja lihtne operatsioon. Asjaolu, et terve massiiv hoitakse mälus koos, teeb massiivi järjestikuse elementhaaval läbikäimise samuti väga kiireks, kuna juba esimese elemendi lugemisel lisatakse vahetult järgnevad elemendid suure tõenäosusega protsessori vahemälusse.

3.5.1 Massiivid Pythonis

Pythonis kasutatakse massiivi asemel listi mõistet ning selle ülesehitus on veidi teistsugune. Nagu juba esimeses peatükis selgus, on Pythoni list sisuliselt viitade massiiv. Seetõttu võimaldab Python erinevalt tavapärasest massiivist hoida listis erinevat tüüpi elemente – see, kui palju mingi element mälus ruumi võtab, ei ole sellise ülesehituse seisukohast oluline:



3.5.2 Massiivi võimalused ja piirangud

Massiivi elementide väärtust saab lihtsasti muuta tavalise omistamistehtega, kuid massiivi elemente lisada ja neid sealt eemaldada üldjuhul ei saa. Kuna aga elementide väärtusi saab muuta, siis saab simuleerida ka elementide lisamist ja eemaldamist. Selleks võib programmeerija deklareerida piisavalt suure pikkusega massiivi ja pidada ise järge, kui mitu elementi tal tegelikult parasjagu massiivis on. Kui parasjagu on kasutusel m elementi, siis uue elemendi lisamine on selle väärtuse salvestamine kohale a[m] (juhul, kui indekseerimine algab 0-st, nagu siin raamatus käsitletud keeltes on). Sellise lisamise töömaht ei sõltu massiivi pikkusest ega elementide arvust selles ja on seega konstantse keerukusega. Kui aga sooviks on lisada element massiivi keskele kohale k, tuleb elemendid kohtadel $k \ldots m-1$ tõsta ühe koha võrra edasi. Selle keerukus on juba $O(n)$. Juhul, kui ei ole vaja säilitada elementide omavahelist järjekorda, võib muidugi salvestada a[k] asuva elemendi väärtuse massiivis viimaseks ja salvestada kohale a[k] uus, soovitud väärtus. Sama on eemaldamisega: kui soovitakse eemaldada element kusagilt kasutuses oleva osa keskelt kohalt k, siis tuleb kõik elemendid kohtadel $k+1..m-1$ tõsta üks koht ettepoole. Kiiresti saab eemaldada ainult viimast elementi. Kui elementide järjekorra säilitamine ei ole oluline, saab tegelikult salvestada eemaldatava väärtuse kohale a[k] viimase väärtuse a[m-1] ning tagada sellega alati konstantse keerukuse ka eemaldamisel. Sageli siiski on järjekorra säilitamine oluline.

3.5.3 Mitmemõõtmelised massiivid

Massiivid võivad olla ka mitmemõõtmelised. Mitmemõõtmelisi massiive on kõige lihtsam ette kujutada tabelina ja tavaliselt räägitaksegi kahemõõtmeliste massiivide puhul ridadest ja veergudest.

Joonisel on kujutatud kahemõõtmeline massiiv A elementide a_{11}-a_{34} hoidmiseks.

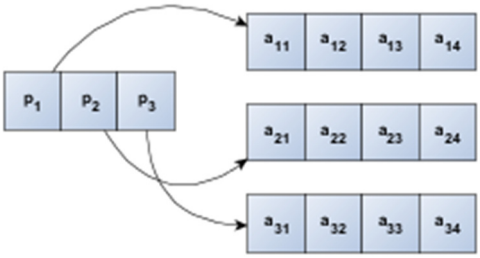
a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}

C/C++ keeles ei erine mitmemõõtmeline massiiv mälus tegelikult ühemõõtmelisest: elemendid on kas ridade või veergude kaupa järjest:

	a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	

Ka nii on elemendi leidmine lihtne, kuna andes ette rea ja veeru indeksi, saab kiiresti arvutada elemendi tegeliku asukoha ehk kauguse massiivi algusest. Kui indekseerimist alustada nullist, siis kaugus avaldub valemiga $k = (r-1) \cdot vn + (v-1)$, kus r on rea number, v on veeru number ja vn veergude arv. Kui hakata ridu ja veerge ka 0-st loendama, nagu programmeerimises tavaks, on valemi kuju veelgi lihtsam: $k = r \cdot vn + v$. Sarnaselt saab arvutada ka suuremamõõtmeliste massiivide elementide kaugusi massiivi algusest, näiteks kolmemõõtmelise massiivi puhul $k = i_3 + N_3 \cdot (i_2 + N_2 \cdot n_1)$, kus i tähistab vastava mõõtme indeksit ja N elementide arvu vastava mõõtme suunal.

Java mitmemõõtmelised massiivid on üles ehitatud veidi teistsugusel põhimõttel: need on pigem massiivid massiividest, mis siis omakorda võivad koosneda massiividest jne.



Selline lähenemine võimaldab luua „hambulisi“ (*jagged*) massiive, kus read ei ole võrdse pikkusega. Sarnane põhimõte on ka Pythonis: listi elemendiks võib olla teine list jne. Loomulikult on võimalik luua ka C++ massiive viitadest (pointeritest), mis viitavad massiividele.

3.5.4 Dünaamilised massiivid

Massiivi pikkus on tavaliselt fikseeritud ja programmi käitusajal seda muuta ei saa. See tekitab mitmeid ebameeldivusi, mistõttu on kasutusele võetud dünaamilised massiivid, mille pikkust saab muuta. Esialgu eraldatakse dünaamilisele massiivile mingi pikkusega osa mälust – olgu see n. Kuni meil ongi vaid kuni n elementi, töötab kõik nagu tavalises massiivis. Kui on vaja lisada $n + 1$. element, siis

1. eraldatakse pikem mäluosa, harilikult pikkusega $2n$,
2. kopeeritakse elemendid esialgsest massiivist uude massiivi,
3. esialgsele massiivile kuulunud mäluosa vabastatakse.

Muidugi võtab elementide kopeerimine uude massiivi aega, kuid suurendades iga kord vajadusel massiivi pikkust 2 korda, on see küllaltki harv olukord ning keskmine elemendi lisamise keerukus massiivi lõppu on enamikel juhtudel konstantne. Teisalt jälle võtab dünaamiline massiiv enamasti mälus rohkem ruumi, kui seal olevate elementide hoidmiseks tegelikult vaja on.

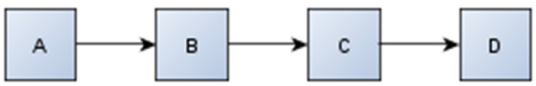
Pythoni `list` ongi dünaamiline massiiv. C++ standardteegis realiseerib dünaamilise massiivi klass `vector` ja Javas `ArrayList`. Tavaliselt on sellistel klassidel meetodid elemendi massiivi lõppu lisamiseks ja lõpust eemaldamiseks, mis on enamasti konstantse keerukusega. Samuti on realiseeritud elemendi lisamine ja eemaldamine suvaliselt positsioonilt, kuid kuna sel juhul tõstetakse tagapool asuvad elemendid ümber, on need operatsioonid lineaarse keerukusega.

Massiive kasutatakse väga palju nii iseseisva andmestruktuurina kui ka teiste andmestruktuuride, sealhulgas pinude, järjekordade ja kuhjade realiseerimiseks.

3.6 AHEL

Massiiviga seotud lisamise ja eemaldamise probleemidest on vaba **ahel**. Ahel on andmestruktuur, kus iga element alates esimesest teab, kus asub talle järgnev element:





Elementide arv ahelas ei ole fikseeritud, neid saab vabalt lisada ja eemaldada konstantse ajaga:



Samas, kuna elemendi asukoht ei ole kindlalt määratud, siis elemendi lugemine ahelast on keerukusega $O(n)$ – iga kord tuleb alustada esimesest elemendist ja käia ahelat läbi kuni soovitud elemendini. See kehtib ka lisamis- ja eemaldamiskoha otsimise jaoks.

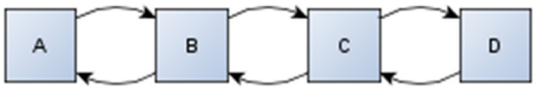
Ahelal on massiivi ees järgmised eelised:

- dünaamilisus – ahel kasvab ja kahaneb vastavalt vajadusele loomulikult;
- lisamis- ja eemaldamisoperatsioonide lihtsus ja kiirus;
- puudub vajadus määrata algsuurust;

Ahelal on ka negatiivsed küljed:

- Suurem mäluvajadus ühe elemendi hoidmiseks, kuna lisaks elemendi väärtusele tuleb meeles pidada ka viit järgmisele elemendile;
- Elemendid ei ole kergesti leitavad, lugemiseks tuleb alustada alati algusest;
- Kuna elemendid võivad paikneda mälus juhuslikult, siis mälust lugemine võib võtta kauem aega, sest ahelas lähestikku paiknevad elemendid ei pruugi olla mälus lähestikku ega jõua protsessori vahemällu.
- Ahelas tagurpidi liikumine on väga kohmakas –iga kord tuleb alustada algusest.

Viimase probleemi lahenduseks kasutatakse topeltseotud ahelaid, kus lisaks järgnevale elemendile teab iga element ka talle eelnevat elementi. See aga nõuab iga elemendi kohta juba kahe viida meelespidamist.



Kuigi ahela näol on tegemist väga tuntud andmestruktuuriga, mida tutvustavad enam-vähem kõik programmeerimisõpikud, tuleb võistlusülesannete lahendamisel tegelikult harva ette ülesandeid, kus on praktiline kasutada ahelat. Eelised dünaamilise massiivi ees tulevad praktikas küllaltki harva välja. Protsessori vahemälu kasvamisega on tegeliku töökiiruse seisukohalt järjest olulisemaks saanud sarnaste andmete füüsiline lähedus mälus.

C++ keeles realiseerib ahelat klass `foward_list` (päisfailis `<forward_list>`) ja topeltseotud ahelat klass `list` (päisfailis `<list>`) . Javas on topeltseotud ahela jaoks klass `LinkedList` (`java.util*`). Pythoni standardisse ahel ei kuulugi.

Järgmises tabelis on toodud dünaamilise massiivi ja ahela tavaliste operatsioonide jaoks kuluv sammude arv:

Operatsioon elemendiga	Dünaamiline massiiv	Topeltseotud ahel
Lugemine suvalisest kohast	1	n/4 keskmiselt, otstest 1
Lõppu lisamine	1, halvimal juhul n	1
Lisamine kohale i	n/2 keskmiselt	n/4 keskmiselt, algusesse lisamine 1
Eemaldamine kohalt i	n/2 keskmiselt	n/4 keskmiselt, otstest 1
Eemaldamine käesolevast kohast	n/2 keskmiselt	1
Lisamine käesolevasse kohta	n/2 keskmiselt	1

3.7 PINU



Pinu (*stack*) on abstraktne andmestruktuur, kus elementidele pääseb juurde kindlas järjekorras – elemendi lisamisel paigutatakse see kõige viimaseks ja elemendi eemaldamisel võetakse ära kõige viimasena lisatud element. Sellise juurdepääsusüsteemi kohta kasutatakse sageli inglisekeelset lühendit LIFO (*last in, first out* – viimasena sisse, esimesena välja).

Pinul on kaks olulist operatsiooni: elemendi lisamine ja elemendi lugemine ja eemaldamine. Suvalisest kohast elementi lugeda/eemaldada ei saa, selleks tuleb eemaldada kõik „pealpool“ olevad elemendid. Samuti ei saa elementi lisada suvalisse kohta pinus – jällegi, selleks tuleks eemaldada „pealpool“ olevad elemendid ja need pärast lisamist tagasi panna.

Pinuga tutvusime põgusalt juba eelmises peatükis alamprogrammide teema juures. Seal selgus, et alamprogrammide väljakutseid hoitakse pinumälus. Pinu põhimõttel töötab ka veebilehitsejates nupp „Tagasi“ – vaadatud leheküljed pannakse järjest pinusse ja „Tagasi“ nupu vajutusel võetakse sealt viimati lisatud lehekülg. Samuti töötab ka tekstiredakorites tagasivõtt (*undo*).

Pinu kasutatakse ka mitmest tehtest koosnevate arvutustehete vastuse leidmiseks. Lihtsam on selleks arvutustehe esitada postfiks-kujul ehk pööratud Poola kujul. See tähendab, et tehtemärk kirjutatakse operandide järele, mitte vahele, nagu tavakasutuses (infiks-kujul). Nii 3 + 4 on postfiks-kujul 3 4 + ja (3 + 4)*5 on 3 4 + 5 *. Postfiks-kuju eeliseks on see, et sulge ei ole vaja kasutada. Postfiks-kujul arvutamine pinu abil toimub järgmiselt:

1. Loe avaldisest järgmine operand või operaator.
2. Kui järgmist operandi/operaatorit pole, loe vastus pinust ja lõpeta töö.
3. Kui on operand (arv), pane see pinusse.
4. Kui on operaator (tehtemärk) loe pinust kaks operandi, soorita tehe ja pane vastus pinusse.
5. Korda esimesest punktits alates.

Pinu realiseeritakse enamasti (dünaamilise) massiivina. Massiivi korral tuleb valida piisavalt suur massiiv ning hoida meeles elementide arvu pinus. Tähistades massiivi A-ga ja pinus olevate elementide arvu n-ga, siis lisamisel suurendatakse n-i ühe võrra ja lisatakse uus väärtus kohale a_n. Eemaldamisel tagastatakse kohal a_n olev väärtus ja vähendatakse n-i ühe võrra.

Pinu on võimalik realiseerida ka ahelana. Sellisel juhul lisatakse uus element alati ahela algusesse ning eemaldamise korral tagastatakse ja eemaldatakse esimene element ahelast.

Mõlemal juhul on eemaldamise ja lisamise keerukus $O(1)$.

Ajalooliselt on pinu kohta eesti keeles kasutatud ka sõna „magasin“. Folkloori kohaselt kasutati Tartu Ülikoolis üht terminit ja Tallinnas teist, aga tänapäeval on „pinu“ rohkem levinud. Kõige paremini saab magasini mõiste selgeks aga siis, kui sul on kell 9 algoritmide ja andmestruktuuride eksam, aga ärkad kell 9:20. Siis lööd käega vastu otsmikku: „Oh, magasin!“

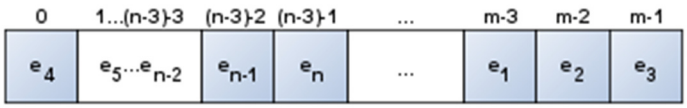
3.8 JÄRJEKORD



Ka järjekord on selline andmestruktuur, kus elementidele pääseb ligi kindlas järjekorras. Järjekorra põhimõte on nagu tavalises poesabas: uus element lisatakse alati lõppu ja järgmine element loetakse ja eemaldatakse alati algusest. Sellist põhimõtet tähistab lühend FIFO (*first in, first out* – esimesena sisse, esimesena välja).

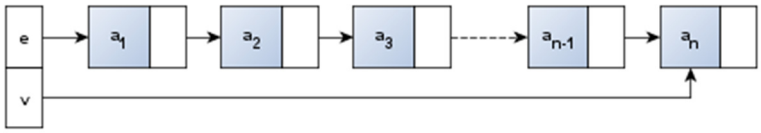
Sarnaselt pinuga on järjekorral operatsioonid elemendi lisamiseks ja eemaldamiseks (koos lugemisega).

Järjekord realiseeritakse tavaliselt samuti massiivina või ahelana, kuid mõlemad on veidi keerulisemad kui pinu korral. Massiivi kasutades on probleemiks see, et järjekorra korral ei püsi kumbki ots muutumatuna. Esimeseks võimaluseks on muidugi kas lisamisel või eemaldamisel nihutada kõiki järjekorras olevaid elemente massiivis, kuid sellisel juhul on lisamise või eemaldamise protseduuri keerukuseks juba $O(n)$. Teiseks võimaluseks on meeles pidada lisaks elementide arvule ka esimese elemendi indeks, mis siis elementi eemaldades nihkub ühe võrra edasi. Sellisel juhul aga on vaja päris korraliku pikkusvaruga massiivi ning ületäitumise oht on väga suur. Selle lahendusena käiakse massiivis ringiratast – massiivi viimasele elemendile järgneb järjekorras jälle esimene. Kuna on teada, et algus on alati lõpust eespool, siis selline lahendus toimib päris hästi.



Järjekord massiivina. m - massiivi pikkus, n - elementide arv järjekorras, $e_1 \ldots e_n$ – elemendid järjekorras.

Vajadus elementidele mõlemast otsast ligi pääseda, teeb ka lihtahela kasutamise järjekorra jaoks mõnevõrra keerulisemaks kui pinu realiseerimisel. Topeltseotud ahelat siiski vaja ei ole, piisab, kui peame eraldi meeles viita viimasele elemendile – sel moel saab lisada uut elementi kiiresti järjekorra lõppu. Tähelepanelik tuleb aga olla olukorras, kus järjekorras ongi vaid üks element – sellisel juhul on see korraga nii esimene kui ka viimane ning selle eemaldamisel tuleb nullida ahela viit nii esimesele kui ka viimasele elemendile.



Järjekord lihtahelana. e - viit esimesele elemendile, v – viit viimasele elemendile, $a_1 \ldots a_n$ – elemendid järjekorras.

Ka järjekorras on mõlemal juhul eemaldamise ja lisamise keerukus $O(1)$.

3.8.1 Kaheotsaline järjekord

Kaheotsaline järjekord (*double ended queue, deque*) on selline andmestruktuur, kus elemendi lisamine ja võtmine on lubatud mõlemast otsast. Praktikas on see andmestruktuur huvitav sellepärast, et seda saab omakorda kasutada nii järjekorra kui ka pinu realiseerimiseks.

3.8.2 Eelistusjärjekord

Eelistusjärjekord (*priority queue*) on selline järjekord, kus igal elemendil on prioriteet ning elemente võetakse järjekorrast vastavalt elemendi prioriteedile. Päris elus kasutatakse eelistusjärjekorda näiteks erakorralise meditsiini osakonnas (EMO) - kõigepealt hinnatakse patsiendi seisundi tõsidust ja kõrgema prioriteediga patsiente teenindatakse enne. Veelgi keerulisemaks teeb olukorra see, et mõnel juhul – nagu ka EMOs – võib elemendi prioriteet muutuda ning siis on vaja seda elementi järjekorras „nihutada“. Seetõttu peab eelistusjärjekorra realiseerimisel arvestama sellega, et elemendi võtmisel eelistusjärjekorrast tuleb otsida pidevalt, milline on suurima prioriteediga element, või tuleb hoida elemente pidevalt prioriteedi järgi sorteerituna, mis teeb jällegi lisamise ja prioriteedi muutmise keerulisemaks. Seetõttu on eelistusjärjekorra realiseerimiseks kõige parem andmestruktuur hoopiski kuhi, millest tuleb juttu alampeatükis 3.9.

3.9 PINU JA JÄRJEKORRA KASUTAMINE

Enamasti on keelte standardteekides pinu juba valmiskujul olemas. C++ on päisfailis `<stack>` klass `stack`, Javas on parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Deque`. Pythonis eraldi klassi pinu jaoks ei ole, kuid Pythoni `listil` on meetodid `append()` järjendi lõppu elemendi lisamiseks ja `pop()` elemendi eemaldamiseks lõpust.

Järgmine tabel illustreerib pinu kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Pinu p loomine (täisarvude jaoks)	stack<int> p	Deque<Integer> p = new ArrayDeque<Integer> ()	p = []
Elemendi 'a' lisamine	p.push(a)	p.push(a)	p.append(a)
Elemendi eemaldamine	p.pop()	a = p.pop()	a = p.pop()
Pealmise elemendi vaatamine	a = p.top()	a = p.peek()	a= p[-1]
Kas pinu on tühi?	p.empty()	p.isEmpty()	not p

Järjekorra leiab C++ keeles päisfailist `<queue>` (klass queue); Javas on taas üldjuhul parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Queue`. Pythonis saab kasutada struktuuri `collections.deque`.

Järgmine tabel illustreerib järjekorra kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine (täisarvude jaoks)	queue<int> s	Queue<Integer> s = new ArrayDeque<Integer> (); Queue<Integer> s = new LinkedList<Integer> ();	s = deque([])
Elemendi 'a' lisamine	s.push(a)	s.add(a)	s.append(a)
Elemendi eemaldamine	s.pop()	a = s.remove(); a = s.poll()	a = s.popleft()
Esimese elemendi vaatamine	a = s.front()	a = s.element(); a = s.peek()	a= s[0]
Kas järjekord on tühi?	s.empty()	s.isEmpty()	not s

Vaatame vahepeal ühe ülesande näitel, kuidas neid andmestruktuure kasutada:



Ühes pisikeses linnakeses elavad lemmingud.
Nende linnas käib ringiratast trammiliin, millel on n peatust.
Trammil on ainult üks uks ja tramm ise on kitsas.
Seetõttu sisenevad lemmingud trammi ükshaaval, nii et esimesena sisenenud lemming liigub trammi tahaosse, tema järel sisenenud jääb tema ette jne.
Väljumine trammist toimub sisenemisele vastupidises järjekorras: kõige viimasena sisenenud lemming väljub esimesena jne.
Igas peatuses on platvorm pikkusega p, millele mahub ootama täpselt p lemmingut. Platvormilt trammi sisenevad lemmingud täpselt ootejärjekorras: kes oli järjekorras enne, siseneb enne.
Liiklemine on korraldatud järgmiselt:
1. Tramm peatub peatuses platvormi lõpus ning lemmingud väljuvad järjest trammist.
2. Kui järgmisena väljuv lemming on jõudnud oma soovitud peatusesse, siis ta lahkub, kui mitte, siis läheb ta platvormil olevasse ootejärjekorda trammile sisenemist ootama.
3. Kui järgmine väljumist ootav lemming ei ole oma peatuses ning platvormil enam ruumi ei ole, siis rohkem lemminguid selles peatuses ei välju, isegi, kui see on nende sihtpeatus.
4. Keegi kellestki mööda ei trügi.
5. Lõpuks sõidab tramm platvormi etteotsa ning võtab järjekorrast peale nii palju lemminguid, kui trammi mahub.
Seejärel sõidab tramm järgmisesse peatusesse ning seal kordub sama protseduur.
Õhtul lahkuvad kõik lemmingid töölt samal ajal ja moodustavad peatustesse järjekorrad. Leia, kui palju aega kulub trammil kõigi lemmingute kodupeatustesse toimetamiseks aega, kui iga sisenemine ja väljumine võtab 1 sekundi, platvormi ühest otsast teise sõit võtab trammil 5 sekundit ja peatuste-vaheline sõit kestab täpselt 1 minuti. Tramm alustab oma teekonda esimesest peatusest.
Sisendi esimesel real on kolm arvu: peatuste arv $n(2 \le n \le 100)$, kohtade arv trammis $k(1 \le k \le 100)$, platvormi pikkus $p(1 \le p \le 100)$. Peatuste platvormid on kõik sama pikkusega.
Järgneval n real on iga peatuse kohta `1...n` seal ootavate lemmingute arv $l(0 \le l \le p)$ ja selle järel iga lemmingu l_i $(0 \le l_i \le l)$ kohta peatuse number n_j $(1 \le n_j \le n)$, kuhu ta sõita soovib. Järjekorras esimene lemmingu peatus on antud esimesena, teise peatus teisena jne.

NÄIDE:

```
5 2 3
3 4 5 2
2 1 3
0
3 3 4 1
1 3
```

Vastus:

```
1220
```

Kui nüüd pinu ja järjekorra mõisted värskelt meeles on, siis on kerge taibata, et trammi sisenemine ja väljumine käib pinu põhimõttel ning peatustes olevad trammisabad on järjekorrad. Muud polegi vaja teha, kui sisendandmed sisse lugeda ja seejärel kogu situatsioon „läbi mängida.“ Iga peatuse jaoks teeme eraldi järjekorra, mida hoiame ühes järjekordade massiivis.

```cpp
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main() {
    int peatuste_arv, kohtade_arv, jk_pikkus;

    //pinu trammis olevate reisijate hoidmiseks, viimasena sisenenu väljub esimesena
    stack<int> tramm;
    //Massiiv trammisabadest. Iga saba on järjekord, kes enne sabas, läheb enne peale
    queue<int> trammisabad[100];

    cin >> peatuste_arv >> kohtade_arv >> jk_pikkus;

    for (int i = 0; i < peatuste_arv; i++) { // iga peatuse kohta
        int ootajate_arv;
        cin >> ootajate_arv; // loeme ootajate arvu peatuses
        for (int j = 0; j < ootajate_arv; j++) { // ja iga ootaja kohta
            int sihtkoht;
            cin >> sihtkoht; // loeme tema sihtkoha
            trammisabad[i].push(sihtkoht - 1); // ning paneme selles peatuses sabasse
        }
    }

    int peatus = 0, kulunud_aeg = 0; // alustame esimesest peatusest
    while (true) {
        // mahalaadimine
        while (!tramm.empty() && // kuni trammis on reisijaid ja
            ((trammisabad[peatus].size() < jk_pikkus) || // peatuses on ruumi või
                tramm.top() == peatus)) { // on järgmine mahamineja jõudnud kohale
            if (tramm.top() != peatus) { // kui järgmine väljuja ei ole oma peatuses
                trammisabad[peatus].push(tramm.top()); // läheb ta selle peatuse sappa
            }
            tramm.pop(); // eemaldame väljuja trammist
            kulunud_aeg++; // lisame väljumiseks kulunud aja
        }

        bool valmis = tramm.empty(); // kas tramm on tühi?
        for (int i = 0; i < peatuste_arv; i++) {
            valmis &= trammisabad[i].empty(); // kas kõigi peatuste sabad on tühjad?
        }
        if (valmis)
            break; // kõik reisijad on kohal
        kulunud_aeg += 5; // liidame peatuses sõitmiseks kulunud aja
        // pealelaadimine
        while ((tramm.size() < kohtade_arv) && // kuni trammis on ruumi
                !trammisabad[peatus].empty()) { // ja peatuses on ootajaid
            tramm.push(trammisabad[peatus].front()); // paneme trammi esimese ootaja
            trammisabad[peatus].pop(); // ja eemaldame ta trammisabast
            kulunud_aeg++; // lisame sisenemiseks kulunud aja
        }

        peatus = (peatus + 1) % peatuste_arv; // järgmine peatus
        kulunud_aeg += 60; // liidame peatuste vahel sõitmiseks kulunud aja
    }

    cout << kulunud_aeg << endl;

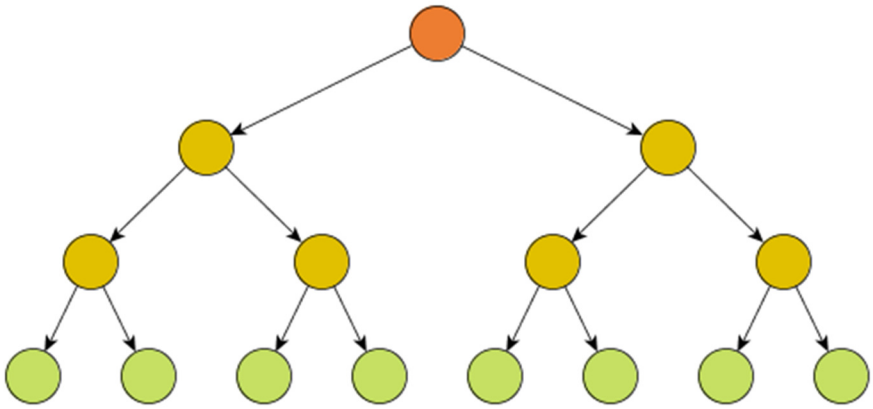
    return 0;
}
```

3.10 KAHENDPUU

Puu on dünaamiline andmestruktuur, mille elemente nimetatakse **tippudeks**. Tipud on korraldatud range hierarhiana, see tähendab, et igal tipul võivad olla **alluvad**, aga neil saab olla ainult üks vahetu **ülemus**. **Juurtipp** kõige kõrgema taseme ülemus, sellel ülemust enam pole. Tippu, millel ei ole alluvaid, nimetatakse **leheks,** alluvatega tippu nimetatakse **vahetipuks** ehk **sõlmeks**.

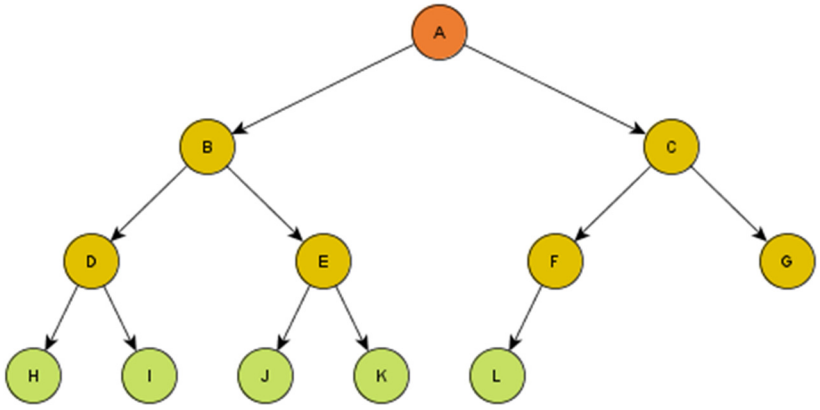
Juurtipp asub esimesel tasemel, kõik tema vahetud alluvad 2. tasemel, nende vahetud alluvad omakorda 3. tasemel jne. Tasemete arvu puus nimetatakse puu **kõrguseks.**

Puud, mille igal sõlmel on kuni n alluvat, nimetatakse n-puuks. Kõige levinum n-puu on **kahendpuu**. Puud nimetatakse **täielikuks**, kui tema igal vahetipul on sama palju alluvaid ning kõik lehed asuvad samal tasemel. n tipuga täieliku kahendpuu kõrgus on $\log_2 n$.



Täielik kahendpuu. Juurtipp on punane, lehed rohelised.

Kui täielikust kahendpuust on viimaselt tasemelt paremalt vasakule eemaldatud 0 või rohkem lehte, siis nimetatakse sellist kahendpuud **kompaktseks**.



Kompaktne kahendpuu.

Kahendpuus on sageli kasulik eristada, kas tipp on oma ülemuse vasak või parem alluv, seda ka juhul, kui ülemusel ongi vaid üks alluv.

Puudel tavaliselt realiseeritavatest operatsioonidest ja nende kasutamises tuleb põhjalikumalt juttu peatükis 6, Graafiteooria.

3.10.1 Kahendpuu esitus

Kahendpuud võib realiseerida nii, et iga tipp on objekt, millel on väljadena viidad tema vasakule ja paremale alluvale.

Sageli on efektiivsem ja lihtsam realiseerida kahendpuu massiivi peale. Puu juur on massiivi esimene element ehk a[0]. Selle vasakpoolne alluv on kohal a[1] ja parempoolne kohal a[2]. a[1] vasakpoolne alluv on kohal a[3] ja parempoolne alluv a[4] jne. Selline esitus sobib hästi kompaktsete kahendpuude hoidmiseks, kuna n-tipuline puu täidab täpselt massiivi pikkusega n.



Eelmisel joonisel kujutatud kompaktse kahendpuu esitus massiivina

Tipp	Asukoht massiivis
Puu juurelement	a[0]
Elemendi a[k] vasak alluv	a[2k+1]
parem alluv	a[2k+2]
ülemus	a[(k-1)/2]

3.10.2 Kahendotsingu puu

Olgu puu igas tipus võtmega kirje. Kahendotsingu puuks (*binary search tree*) nimetatakse sellist puud, mis on tühi või mille vasaku alampuu kõigi tippude võtmed on väiksemad kui juurtipu võti ning parema alampuu tippude võtmed on kõik suuremad kui juurtipu võti ning mõlemad alampuud on kahendotsingu puud.

Kahendotsingu puul kasutatavad operatsioonid on võtme otsimine, uue kirje lisamine ja kirje eemaldamine, samuti vähima ja suurima võtmega kirje leidmine.

Otsimine otsingupuust on lihtne: võrdleme kõigepealt otsitavat väärtust juurtipu võtmega – kui otsitav väärtus on suurem, peab see asuma parempoolses alampuus, kui aga väiksem, siis vasakpoolses. Sama reegel kehtib ka alampuudest otsimisel. Kuna igal sammul liigutakse puus ühe taseme võrra alla, siis sellise otsimise korral on halvima juhu keerukuseks puu kõrgus. Seetõttu on oluline hoida kahendotsingu puu kõrgus võimalikult väike ehk hoida puu tasakaalus.

Tasakaalustatud kahendotsingu puuks nimetatakse sellist kahendotsingu puud, mille iga tipu vasak ja parem alampuu on enam-vähem sama kõrgusega. Kui ühegi tipu alampuude kõrgused ei erine rohkem kui ühe võrra, siis sellist kahendotsingu puud nimetatakse **AVL-puuks.** Nimi tuleb selle puu leiutajate G. Adelson-Velski ja E. Landise nimetähtedest.

Ühtlase kõrguse hoidmine on oluline selleks, et sellises puus on tavapärased operatsioonid teostatavad keerukusega $O(log\ n)$. Loomulikult võib nii tipu lisamine kui ka eemaldamine viia sellise puu tasakaalust välja, seetõttu on oluline puu vajadusel pärast lisamist/eemaldamist tasakaalustada.

3.11 KUJUTIS

Kujutis (*map*) on andmestruktuur, mille elementideks on kaheosalised kirjed – võtme ja väärtuse paarid. Võtmed on kordumatud, st ühe ja sama võtmega elemente võib olla vaid üks. Peamisteks operatsioonideks on kirje lisamine ja eemaldamine ning võtme järgi kirje leidmine.

Kui võtmete võimalik väärtusvahemik on teada ning see ei ole kuigi ulatuslik, saab kasutada massiivi koos **vahetu indekseerimisega** – võtmele v vastava element kirjutatakse massiivi kohale $a[v]$. Puuduvate võtmete kohale kirjutatakse mingi spetsiaalne väärtus, et neid tegelikest võtmetest eristada. Sellisel moel on nii lisamine, eemaldamine kui ka võtme järgi elemendi leidmine keerukusega $O(1)$.

Kui aga võtmeid on rohkem, kasutatakse kujutise realiseerimiseks **paisktabelit** *(hash table)*. Paisktabel on samuti massiiv, kuid elementi võtmega v ei kirjutata otse kohale a[v], vaid kasutatakse **paiskfunktsiooni**, mis seab igale võtmele vastavusse mingi indeksi. See tähendab, et võtmele v vastav kirje kirjutatakse paisktabelisse kohale a[$f(v)$], kus f on paiskfunktsioon.

Paiskfunktsioon peab olema kiiresti arvutatav. Harilikult valitakse paiskfunktsiooniks $f(v) = v \bmod m$, kus m on paisktabeli ridade arv.

Paisktabeli realiseerimisel võetakse tavaliselt mugavalt palju ruumi, et andmed paisktabelisse vabalt ära mahuvad. Sellegipoolest võib paisktabelis esineda **põrkeid** ehk **kollisioone**. Põrge on selline olukord, kus paiskfunktsioon f annab kahe erineva võtme v_1 ja v_2 jaoks sama väärtuse, st $f(v_1) = f(v_2)$, kui $v_1 \neq v_1$.. Sisuliselt tähendab see, et kaks kirjet peaks asuma paisktabelis samal kohal.

Kokkupõrgete lahendamiseks kasutatakse peamiselt kahte erinevat võimalust. **Välisahelate** meetodi korral on igal real kaks lisavälja: üks selle jaoks, et märkida, kas rida on juba hõivatud ja teine on viit lihtahelale. Kui objekti võtmega v salvestatakse reale $f(v)$, kuid see rida on juba hõivatud, salvestatakse uus objekt realt $f(v)$ algavasse lihtahelasse.

Lahtise adresseerimise meetodi korral lahendatakse põrge järgmiselt: kui võtmele v vastav rida $f(v)$ on juba hõivatud, otsitakse eelmine vaba rida ning lisatav kirje salvestatakse sinna. Lugemisel tuleb siis samuti otsida võtit igalt eelnevalt realt, kuni võti leitakse või jõutakse tühja reani, mis tähendab, et antud võtit tabelis ei esine. Veelgi parem on kasutada topeltpaiskamisega lahtist adresseerimist, mille korral ei otsita vaba kohta järjest, vaid kasutatakse mingit teist funktsiooni $f_2(v)$, mis tagastab sammu pikkuse, mitu kohta tuleb esialgsest asukohast edasi minna, et leida võtmele vastav uus asukoht. Sellise meetodiga leitakse kirje otsimisel üldiselt kiiremini üles.

3.11.1 Kujutis programmeerimiskeeltes

C++ pakub standardteegis kahte klassi `map` ja `unordered_map`. Mõlemad on mõeldud võti-väärtus paaride hoidmiseks, kuid `map` on tavaliselt realiseeritud kahendotsingupuuna, mistõttu on sealt elemendi leidmine aeglasem operatsioon, see-eest võimaldab mugavamat elementide läbikäimist (elemendid on võtme järgi sorteeritud). `unordered_map` on enamasti realiseeritud paisktabelina ja võimaldab võtme järgi ligipääsu konstantse keerukusega.

Java valik pole halvem: seal on klassid `TreeMap` ja `HashMap` ning nendel liides `Map`.

Pythonis on olemas andmetüüp `dict` (*dictionary*) võti-väärtus paaride hoidmiseks. Realiseeritud on see topeltpaiskamisega lahtise adresseerimisega paisktabelina.

3.12 PRAKTILINE ÜLESANNE

Usina linna avatud ülikoolis on igaühel võimalik käia kuulamas erinevaid kursusi. Usinas linnas elavad muidugi usinad õppijad, kes enamasti osalevad aastas viiel kursusel. Seetõttu on ülikoolil plaanis pakkuda 5-kursuselisi valmiskomplekte. Esialgu otsustatakse katsetada kombinatsiooni kõige populaarsematest kursustest, selleks aga on vaja teada, millise viie kursuse kombinatsioon esineb kõige sagedamini. Erinevaid kursusi on palju ja need on nummerdatud arvudega 100…499.
Sisendi esimesel real on arv n(1≤n≤10000) mis näitab, kui palju on olnud eelneval aastal viie kursuse valijaid, ning järgneval n real igaühel ühe õppija valitud kursuste numbrid valimise järjekorras, tühikutega eraldatuna. Väljastada kõige sagedasem kursuse kombinatsioon. Kui sama sagedusega on mitu kombinatsiooni, väljastada kõik erinevad suurima sagedusega kombinatsioonid, iga kombinatsioon eraldi real.



NÄIDE:

```
5
101 102 103 104 105
217 301 101 105 400
101 301 400 217 105
101 301 103 104 105
105 104 103 102 101
```

Vastus

```
101 102 103 104 105
101 105 217 301 400
```

Siin on peamiseks küsimuseks, kuidas leida korduvad kombinatsioonid ja neid loendada. Erinevaid võimalikke kombinatsioone on $\binom{400}{5}$, mis on ligikaudu 83 miljardit. See ei aita meid kuidagi edasi. Üheks võimaluseks oleks välja mõelda funktsioon, mis igale kombinatsioonile seab vastavusse ühe arvu – nii saaks leida iga kombinatsiooni jaoks sisendis sellele vastava arvu ning loendada, kui mitu korda mingit arvu esineb. Antud juhul on loomulik meetod kasutada kursusenumbreid stringidena ja need kokku kleepida üheks pikaks stringiks. Kui enne kursused mingis järjekorras sorteerida, siis samale kombinatsioonile vastab sama string (ja vastupidi – ühele stringile vastab üks kindel kombinatsioon) ja seda stringi saame kasutada võtmena:

```
#include <map>
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    map<string, int> logi;

    // siin hoiame seni leitud suurimat esinemissagedust. Kuna ülesande tingimustes on
    //vähemalt 1 kombinatsioon olemas, võib selle panna üheks.
    int maxN = 1;
    string kursused[5]; //siin hoiame ühe inimese võetud kursusi
```

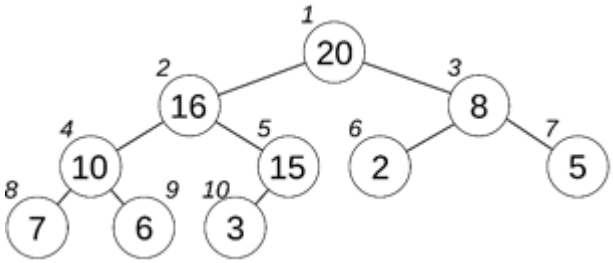
```
    for (int i = 0; i < n; i++) {
        cin>> kursused[0] >> kursused[1] >> kursused[2] >> kursused[3] >> kursused[4];
        sort(kursused, kursused + 5); // sorteerime kursused kasvavas järjekorras

        string voti;
        for (int j = 0; j < 5; j++) {
            voti += kursused[j]; // ühendame kursuste numbrid üheks stringiks
        }
        // kui sellist võtit pole (sellist kombinatsiooni ei esine)
        if (!logi.count(voti))
{           logi[voti] = 1; // lisame selle väärtusega 1.
        }
        else {  // kui võti on
            int m = logi[voti] + 1; // suurendame kombinatsiooni korduste arvu
            logi[voti] = m; // ja muudame vastavaks
            // kui on suurem korduste arv, kui senine maksimum, muudame maksimumi.
            maxN = max(maxN, m);
        }
    }

    map<string, int>::iterator it;
    for (it = logi.begin(); it != logi.end(); it++) { // käime mapi läbi
        if (it->second == maxN) // kui on suurim korduste arv
            for (int i = 0; i < 13; i += 3) {
                // eraldame võtmest kursuste numbrid ja väljastame need
                cout << it->first.substr(i, 3) << " ";
            }
        cout << endl;
    }

    return 0;
}
```

3.13 KUHI



Kuhi (*heap*) on selline puu, millel kehtib **kuhjaomadus**: ühegi tipu võti ei ole suurem kui tema ülemuse võti. Kui kehtib vastupidine omadus – ühegi tipu võti ei ole väiksem kui tema ülemuse võti – siis on tegemist **pöördkuhjaga**.

Kahendkuhi on kuhi, mis on ühtlasi ka kompaktne kahendpuu. Sageli mõeldaksegi kuhja all justnimelt kahendkuhja ning siingi peatükis vaatame edasi vaid kahendkuhja.

Kuna kuhi on kompaktne kahendpuu, siis sobib selle esituseks väga hästi massiiv.

Kuhjal on enamasti realiseeritud järgmised operatsioonid:

- Suurima (või vähima) elemendi (võtme)väärtuse leidmine, keerukus $\Theta(1)$;
- Lisamine, keerukus $O(log(n))$;
- Suurima (vähima) elemendi kustutamine, keerukus $\Theta(log(n))$
- Kuhjastamine ehk etteantud elementide kuhjaks paigutamine;

Operatsioone kuhjal võib realiseerida mitmel moel, lisamisel ja eemaldamisel tuleb aga hoolega silmas pidada, et kuhja omadus säiliks. Selleks on kuhjal enamasti meetodid elementide nihutamiseks üles- ja allapoole. Lisamine on harilikult realiseeritud nii, et uus element lisatakse kuhja lõppu ning

nihutatakse seda kuhjas ülespoole, kuni kuhja omadus on taastatud. Eemaldamisel asendatakse juurelement viimase elemendiga kuhjas, seejärel kukutatakse see element kuhjas õigesse kohta ja kuhjaomadus saab taastatud.

Kuhjasid kasutatakse:

- kuhjameetodil sorteerimiseks ($O(n \log n)$),
- valikualgorimide (*selection algorithms*) realiseerimiseks, kuna miinimum ja maksimumelemendi leidmine on kuhjas konstantse ajaga ning mediaani ja k. elemendi leidmine toimuvad vähem kui lineaarse ajaga,
- mitmete graafitöötlusalgoritmide realiseerimiseks,
- eelistusjärjekorra realiseerimiseks.

C++ keeles on päisfailis `<queue>` klass `priority_queue`. Selle esimene, eemaldatav element on alati suurim element. Kuna kuhja on mugav hoida massiivis, siis `priority_queue` kasutab vaikimisi `vector` klassi tegeliku konteinerina ning rakendab sellel operatsioone kuhjastruktuuri säilitamiseks.

Javas on klass `PriorityQueue`, mis on oma olemuselt pöördkuhi, nii et Javas võetakse eelistusjärjekorrast järgmisena kõige väiksem element.

Pythonis on hea kasutada `Lib/heapq.py`. Nagu Javagi, kasutab Python pöördkuhja ehk esimene element on kõige väiksem.

Järgmine tabel illustreerib kuhja kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine	priority_queue<int> e	PriorityQueue<Integer> e = new PriorityQueue<Integer> ();	e = []; heapify(e)
Elemendi 'a' lisamine	e.push(a)	e.add(a)	heappush(e, a)
Elemendi eemaldamine	e.pop()	a = e.remove(); a = e.poll()	a = heappop(e)
Esimese elemendi vaatamine	a = e.top()	a = e.element(); a = e.peek()	a= e[0]
Kas järjekord on tühi?	e.empty()	e.isEmpty()	not e

3.14 SORTIMINE

Sortimine on etteantud hulga elementide mingisse kindlasse järjestusse seadmine. Enamasti kasutatakse numbrilist või leksikograafilist järjestust. Elemendid antakse harilikult ette massiivina ning sortimise tulemuseks on massiiv, kus

- ükski eelnev element ei ole suurem järgnevast,
- väljund on sisendi permutatsioon.

Erinevaid sortimisalgoritme on päris palju. Sortimisalgoritmi nimetatakse **stabiilseks**, kui võrdsete elementide korral säilib nende omavaheline järjestus algses jadas. Näiteks, kui on antud mängukaardid järjestuses ♣7 ♥2 ♥7 ♣8, siis numbrite järgi sorteerides on korrektsed nii ♥2 ♣7 ♥7 ♣8 kui ka ♥2 ♥7 ♣7 ♣8, kuid ainult esimeses on säilinud seitsmete omavaheline algne järjestus: risti seitse on eespool ärtu seitsmest.

Öeldakse, et sortimisalgoritm töötab **kohapeal** *(in-place)*, kui algoritm ei vaja algandmete sorteerimiseks rohkem kui konstantset hulka lisamälu.

Sõnastame sorteerimisülesande ja vaatame selle erinevaid lahendusi:

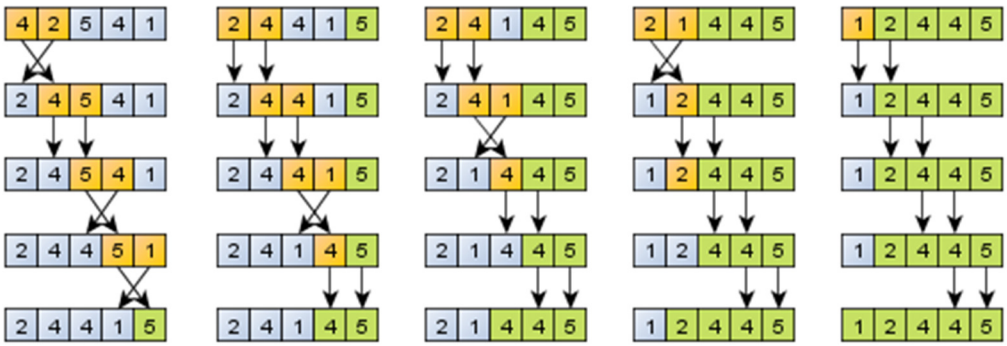
On antud n-elemendiline täisarvudest koosnev jada. Paiguta elemendid ümber nii, et $a_1 \leq a_2 \leq \cdots \leq a_n$.

3.14.1 Mullimeetod

Mullimeetod (*bubble sort*) on üks vanemaid ja tuntumaid sortimisalgoritme. Algoritm on lihtsasti mõistetav ning ka väga lihtne programmeerida, mistõttu tutvustatakse seda sageli programmeerimise algkursustel:

```
int mullimeetod(int* jada, int pikkus)
{
    bool vahetus = true;
    while (vahetus) { //kui on olnud vahetusi
        vahetus = false; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                vahetus = true; //ja peame meeles, et vahetus toimus
            }
        }
    }
}
```

Järgnev skeem kujutab selle algoritmi tööd:



Skeemilt on lihtne märgata, et esimesel läbikäimisel satub kõige suurem element jada lõppu, teisel läbikäimisel pannakse eelviimane element oma kohale jne, mis tähendab, et tegelikult ei ole vaja igal sammul kogu jada läbi käia. Veelgi enam – pole raske veenduda, et kõik elemendid, mis asuvad viimasest vahetusest tagapool, on juba sorteeritud. Siin on optimeeritud lahendus:

```
int optmullimeetod(int* jada, int pikkus)
{
    int viimane = pikkus;
    while (viimane > 0) { //kui on olnud vahetusi
        viimane = 0; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                viimane = i; //ja peame meeles viimase vahetuse koha
            }
        }
        pikkus = viimane;
    }
}
```

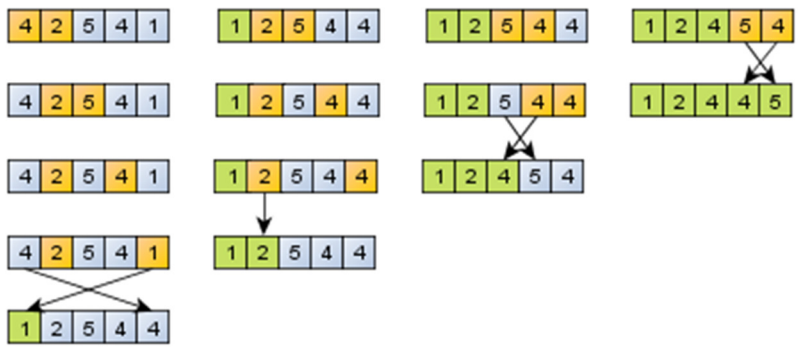
Mullimeetodi keerukus on $O(n^2)$ ning isegi teiste sama keerukusklassi sortimisalgoritmidega võrreldes (nt pistemeetod) on mullimeetod üldjuhul aeglasem, kuna keskmiselt tehakse rohkem elementide vahetusi.

3.14.2 Valikmeetod

Valikmeetod (*selection sort*) on samuti väga lihtne sortimisalgoritm: kõigepealt leitakse jadas kõige väiksem element ning vahetatakse see kõige esimese elemendiga. Seega pärast vahetust on kõige väiksem element oma kohal ning protseduuri korratakse ülejäänud jada elementidel, kuni kõik elemendid on õige koha leidnud:

```
void valikmeetod(int* jada, int pikkus)
{
    for (int j = 0; j < pikkus - 1; j++) { //üksik element on vähim, piisab
    // eelviimaseni vaatamisest
        int min = j; //paneme esimese õige asukohata elemendi minimaalseks
        for (int i = j + 1; i < pikkus; i++) { //käime jada lõpuni läbi
            if (jada[i] < jada[min]) { //leidime väiksema elemendi
                min = i; //jätame asukoha meelde
            }
        }
        if (min != j) { //kui vähim element ei ole juba õigel kohal
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[min];
            jada[min] = temp;
        }
    }
}
```

Siin on skeem, mis kujutab selle algoritmi tööd:



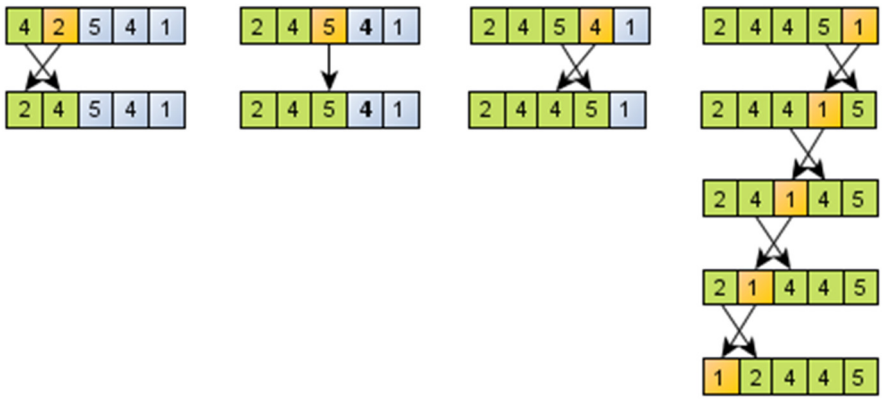
Valikmeetodi keerukus on samuti on $O(n^2)$. Samas on skeemilt kohe näha, et tegelikke vahetusi tehakse oluliselt vähem kui mullimeetodit kasutades – vahetuste arv ei ole kunagi suurem kui elementide arv.

3.14.3 Pistemeetod

Kolmas $O(n^2)$ keerukusega sortimisalgoritm on pistemeetod (*insertion sort*). Pistemeetod on sageli kasutuses väiksematel andmemahtudel ning on keskmiselt kiirem kui mulli- või valikmeetod ning sedagi on lihtne implementeerida:

```
void pistemeetod(int* jada, int pikkus)
{
    for (int i = 1; i < pikkus; i++) { //vaatame iga elementi alates teisest
    //kuni vaadeldava elemendini jada[i] on jada sorteeritud. Otsime õige koha jadas
        for (int j = i; j > 0 && jada[j - 1] > jada[j]; j--) {
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[j - 1];
            jada[j - 1] = temp;
        }
    }
}
```

Pistemeetod on kujutatud järgmisel skeemil:



Pistemeetodi tegelik keerukus oleneb algandmetest. Nagu mainitud, siis halvimal juhul – kui jada on kahanevas järjestuses – tuleb teha n^2 võrdlust ja vahetust, samas, kui andmed on juba õiges järjekorras, tehakse vaid n võrdlust. Väikeste jadade korral ($n < 20$) on pistemeetod kiirem kui keerulisemad sortimisalgoritmid ja seda kombineeritakse sageli kiirmeetodiga.

3.14.4 Põimemeetod

Põimemeetod (*merge sort*) on juba mõnevõrra keerulisem sortimisalgoritm. Selle mõtles 1945. aastal välja arvutiteaduse üks „isasid“ John von Neumann. Tegemist on „jaga ja valitse“ tüüpi algoritmiga, kus jada jagatakse pooleks ja sorteeritakse kumbki pool eraldi, hiljem sorteeritud pooled põimitakse. Loomulikult ei piirduta vaid ühekordse poolitamise ja põimimisega, vaid seda tehakse seni, kuni alamjadad on soovitud pikkusega.

Algoritmi põhiosa on lihtne:

```
void poimemeetod(int* jada, int v, int p)
{
    if (v < p) {
        int keskkoht = (v + p) / 2;
        poimemeetod(jada, v, keskkoht);
        poimemeetod(jada, keskkoht + 1, p);
        poimi(jada, v, keskkoht, p);
    }
}
```

Põimimise osa nõuab rohkem näputööd:

```
void poimi(int* jada, int v, int keskkoht, int p)
{
    int i, j, k;
    int vpikkus = keskkoht - v + 1; // esimese lõigu pikkus
    int ppikkus = p - keskkoht; // teise lõigu pikkus

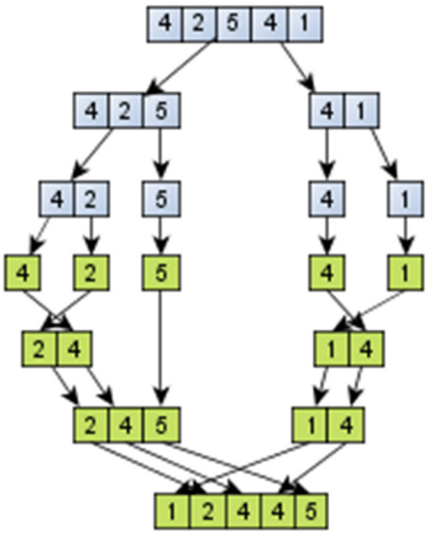
    int *V = new int[vpikkus]; // põimimiseks teeme vasaku ja
    int *P = new int[ppikkus]; // parema lõigu jaoks eraldi massiivid

    for (i = 0; i < vpikkus; i++)
        V[i] = jada[v + i];
    for (j = 0; j < ppikkus; j++)
        P[j] = jada[keskkoht + 1 + j];

    i = 0; // vasakpoolse jada esimese elemendi indeks
    j = 0; // parempoolse jada esimese elemendi indeks
    k = v; // põimitava koha algusindeks jadas
    while (i < vpikkus && j < ppikkus) { // kuni mõlemas jadas on vaatamata elemente
        if (V[i] <= P[j]) { // vasakul on mittesuurem paigutamata element
            jada[k++] = V[i++]; // paneme selle jadasse
        }
        else { // paremal on väiksem element
            jada[k++] = P[j++]; // paigutame selle jadasse
        }
    }

    while (i < vpikkus) {//kui vasakus jadas on veel elemente
        jada[k++] = V[i++]; //kopeerime need järjest jadasse
    }

    while (j < ppikkus) { // kui paremas jadas on veel elemente
        jada[k++] = P[j++]; // kopeerime need järjest jadasse
    }
}
```



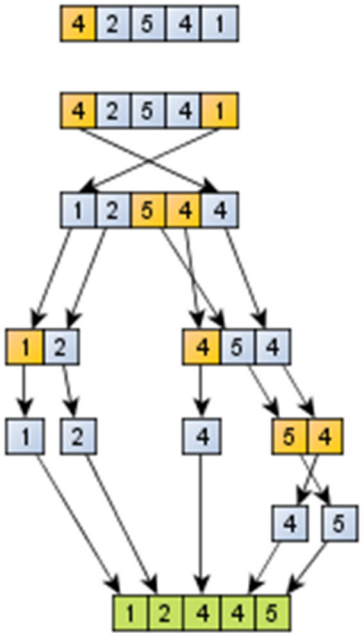
Põimemeetodi keerukusklass (ka halvimal juhul) on $O(n \log n)$ ja see sobib eelkõige suuremate jadade sorteerimiseks. Põimemeetodi miinuseks on, et see meetod nõuab lisamälu mahus $\Omega(n)$. Eksisteerib ka selline variant põimimisest, mis teeb seda kohapeal, ilma märkimisväärselt lisamälu kasutamata, kuid see algoritm on juba päris keeruline. Huvilistele: https://github.com/liuxinyu95/AlgoXY/blob/algoxy/sorting/merge-sort/src/mergesort.c
Põimemeetod sobib väga hästi ahelate sorteerimiseks. Sellisel juhul puudub ka vajadus lisamälu jaoks.

3.14.5 Kiirmeetod

Praegu tuntud algoritmidest on üldjuhul kõige kiirem meetod suurte jadade sortimiseks kiirmeetod (*quicksort*). Ka kiirmeetod on „jaga ja valitse“ tüüpi algoritm. Algoritmi idee on järgmine: sorteerimata jadast võetakse üks element, nn veelahe (*pivot*) ning võrreldakse teisi elemente sellega. Kõik elemendid, mis on veelahkmest väiksemad või võrdsed sellega, paigutatakse valitud elemendist vasakule ehk ettepoole, need aga, mis on suuremad, paigutatakse paremale ehk tahapoole. Selle tulemusena on veelahkmest vasakul ainult sellest mittesuuremad elemendid ja paremal ainult suuremad elemendid. Seejärel sooritatakse sama protseduur valitud elemendist vasakul ja paremal pool eraldi. Kui vasakul või paremal on vähem kui kaks elementi, on see osa juba sorteeritud ja rohkem ei olegi vaja midagi teha.

```
int jaota(int* jada, int v, int p)
{
    int e = jada[v];
    int i = v - 1;
    int j = p + 1;
    while (true) {
        while (jada[++i] < e) {}
        while (jada[--j] > e) {}
        if (i >= j) return j;
        int temp = jada[i];
        jada[i] = jada[j];
        jada[j] = temp;
    }
}
```

```
void kiirmeetod(int* jada, int v, int p)
{
    if (v < p) {
        int e = jaota(jada, v, p);
        kiirmeetod(jada, v, e);
        kiirmeetod(jada, e + 1, p);
    }
}
```



Kiirmeetodi keskmine keerukus on $O(nlog\ n)$. Halvimal juhul on kiirmeetodi keerukus küll $O(n^2)$, kuid praktikas on tegemist siiski kõige kiirema sorteerimisalgoritmiga. Sageli kombineeritakse kiirmeetodit pistemeetodiga – kui sorteeritava alamjada pikkus on väike, näiteks kuni 20 elementi, sorteeritakse see pistemeetodil, vastasel juhul vastavalt kiirmeetodi algoritmile.

Kiirmeetodi probleemseks kohaks võib osutuda selle meetodi ebastabiilsus.

3.14.6 Võrdlustel põhineva sortimise keerukuse alampiir

Eelnevalt vaadeldud sortimisalgoritmid põhinesid kõik elementide võrdlemisel. Selliste algoritmide keerukuse alampiir on $O(n \log n)$. Miks?

Iga võrdlustehe annab meile informatsiooni jada järjestuse kohta. Kokku on n-elemendilisel jadal $n!$ erinevat võimalikku järjestust. Saame konstrueerida otsustuste puu, kus lehtedeks on kõik võimalikud n permutatsioonid ja sõlmedeks võrdlustehted.

Võrdlustehtega sõlmel on

- vasakuks alluvaks on võrdlustehe, mille algoritm sooritab siis, kui võrdluse tulemus on negatiivne,
- paremaks alluvaks on võrdlustehe, mis sooritatakse siis, kui tulemus oli positiivne.

Iga permutatsioonini viib täpselt üks otsustuste tee. Kõige lühem selline tee ehk lehe vähim võimalik kaugus puu juurest vastab parimale juhule – kõige pikem tee – ehk puu kõrgus vastab halvimale võimalikule juhule. $n!$ lehega kahendpuu minimaalseks kõrguseks on $\log_2(n!) = \log_2 1 + \log_2 2$

+…+ $\log_2 n$. Kuna hetkel huvitab meid ainult alampiir, siis pole vaja leida täpset tulemust. Selle summa viimased $n/2$ liiget $\log_2 \frac{n}{2} \ldots \log_2 n$ ei ole ükski suurem kui $\log_2 \frac{n}{2}$. Seega

$$\log_2(n!) \geq \frac{n}{2}\log_2 \frac{n}{2} = \frac{n}{2}\log_2 n - \frac{n}{2}.$$

Järelikult on sellise puu minimaalne kõrgus $O(n \log n)$ ja seega ei saa võrdlemistel põhineva sorteerimisalgoritmi keerukus halvimal juhul olla parem kui $\Omega(n \log n)$.

3.15 SORTIMISE ERIMEETODID

Kui sisendandmed vastavad teatud täiendavatele piirangutele, siis mõningatel juhtudel on võimalik kasutada ka efektiivsemaid sortimisalgoritme, mille keerukusklass on $O(n)$. Tuntumad neist on loendamismeetod (*count sort*), positsioonimeetod (*radix sort*) ja kimbumeetod (*bucket sort*). Kõik need meetodid kasutavad lisamälu mahus $\Omega(n)$.

3.15.1 Loendamismeetod

Loendamismeetodit saab kasutada, kui andmed on teadaolevas ja küllaltki kitsas vahemikus. Loendamismeetodi idee seisneb selles, et luuakse eraldi loendusmassiiv, kus igale indeksile vastab üks elemendi võimalik väärtus. Kui suurim jadas esinev väärtus on max ja vähim min, siis on loodava jada pikkus $max - min + 1$. Seejärel käiakse esialgne jada elementhaaval läbi ning suurendatakse loendusmassiivis elementi kohal a_{i-min}. Sellisel moel loendatakse iga element.

0	1	2	3	4	indeks
1	1	0	2	1	
1	**2**	**3**	**4**	**5**	**väärtus, mida loendame**

Massiivile 4,5,2,4,1 vastav loendusmassiiv

Kui loendasime arve, pole edasi vaja teha muud, kui vastavalt loendusmassiivile kirjutada igale indeksile vastavat väärtust esialgsesse massiivi nii mitu korda, kui seda esines. Kui tegemist on aga objektidega, siis saab arvutada, mitmendal kohal peaks sorteeritud massiivis olema antud võtmeväärtusega element. Selleks arvutatakse loendusmassiivis iga elemendi m[i] jaoks alates teisest m[i] + m[i-1], s.t leitakse selle võtmega elemendi maksimaalne positsioon sorteeritud jadas. Loendusmassiiv on siis kujul [1, 2, 2, 4, 5].

Käime esialgse jada tagurpidi läbi ja iga elemendi jaoks leiame selle positsiooni sorteeritud jadas: j = m[a[i]- v] – 1. Kirjutamegi elemendi sellele positsioonile uude vastusjadasse ning vähendame loendusmassiivis selle elemendi võtmele vastavat väärtust. Näiteks, pannud viimase elemendi (1) vastusjadasse `[1,x,x,x,x]`, vähendame loendusmassiivis väärtust kohal 1-1 ja saame `[0,2,2,4,5]`. Järgmiseks paigutame massiivi eelviimase elemendi 4, ning saame vastusmassiivi `[1,x,x,4,x]` ning loendusmassiivi `[0,2,2,3,5]` – järgmine „4“, mis algses jadas ette tuleb, paigutatakse nüüd kohale 3 - 1 = 2. Sellisel moel elemente paigutades on tagatud ka stabiilsus.

See meetod vajab lisamälu vähemalt loendusmassiivi pikkuse k jagu. Objektide korral on lisaks sellele vaja veel ühe n-elemendilise jada jagu mälu sorteeritud elementide hoidmiseks, mis teeb kokku lisamälu vajaduseks $\Omega(n + k)$. Ka loendusmassiiv tuleb korra läbi käia, mistõttu omab selle algoritmi kasutamine mõtet siis, kui k on väike.

3.15.2 Positsioonimeetod

Positsioonimeetodiga sorteeritakse harilikult positiivseid, mitte väga pikki täisarve. Meetod seisneb selles, et arve ei vaadelda arvudena, vaid numbritest koosnevate liitvõtmetena. Seega kui näiteks kolmekohalisi arve sorteerides sorteerime need esmalt üheliste järgi, seejärel kümneliste järgi ja siis sajaliste järgi, saamegi tulemuseks sorteeritud jada.

Alamalgoritmina kasutatakse sorteerimiseks loendamismeetodit, seetõttu peavad võtmed olema kõik sama pikkusega. Kasutus ei piirdu siiski ainult arvudega, sel moel saab sorteerida ka sõnu.

3.15.3 Kimbumeetod

Kimbumeetod seisneb selles, et väärtuste järgi jaotatakse elemendid eraldi kimpudesse või ämbritesse, mida siis sorteeritakse edasi kas kimbumeetodil või mõnel muul viisil. Näiteks sorteerides jada, kus on arvud 0…99, eraldame ühte kimpu kõik arvud 0..9, teise 10…19 jne. Sorteerides need kimbud eraldi, saame tulemustest kokku panna sorteeritud jada.

3.16 MITME KRITEERIUMI JÄRGI SORTIMINE

Mõnikord võib olla vaja sorteerida mitme kriteeriumi järgi, näiteks kaarte masti ja väärtuse järgi. Üheks võimaluseks on kasutada mõnda stabiilset sortimismeetodit ning sortida iga kriteeriumi järgi eraldi, alustades kõige vähemtähtsamast. Kui kriteeriume on k, tuleb jada sorteerida k korda ehk sellise lähenemise keerukuseks on $O(kn \log n)$.

Tavalisemaks lahenduseks on koostada oma spetsiaalne võrdlemisfunktsioon, mis suudab objektid etteantud järjestuses reastada, võrreldes kõigepealt kõige olulisemat kriteeriumi ja kui selle järgi on objektid võrdsed, siis tähtsuselt järgmist jne. Halvimal juhul tuleb iga võrdlusoperatsiooni korrata k korda – iga kriteeriumi jaoks – ja seegi lähenemine annab halvimal juhul keerukuseks $O(kn \log n)$. Päriselt töötab see lahendus aga kiiremini kui eelmine, sest enamasti ei ole vaja siiski võrrelda kõiki kriteeriume ning teiseks päästab see lähenemine vajadusest kasutada stabiilset sortimisalgoritmi.

3.17 PROGRAMMEERIMISKEELTE STANDARDTEEGID

Enamasti pole siiski vaja jalgratast leiutada, kuna mõistlik üldine sorteerimine on keelte standardteekides olemas. Eriti oluline on see võistlustel, kus lahendamisaeg on piiratud ning oma sortimisalgoritmi kirjutamine ja testimine on liigne ajakulu. Seetõttu peaks iga võistleja end kurssi viima oma keele pakutavate võimalustega.

Keele spetsifikatsioon määrab enamasti ära, mis tingimused sortimismeetodile kehtima peavad. Praegu nõuavad nii C++, Java kui ka Python, et sortimisfunktsiooni halvima juhu keerukus on $O(n \log n)$. Java ja Pythoni sortimisfunktsioonid on stabiilsed, C++ seda nõuet ei esita, rõhutakse pigem nn kohapeal sortimisele. Millist algoritmi sortimiseks konkreetselt kasutatakse, sõltub juba konkreetselt kasutatavast teegist. Näiteks GNU C++ standardteek kasutab *introsort* algoritmi kombineerituna pistemeetodiga. *Introsort* on kiirmeetodi ja kuhjameetodi hübriid. Pythoniga tuli kasutusse *Timsort* algoritm, mis on tuletatud põime- ja pistemeetodist. Ka uuemad Java versioonid kasutavad *Timsort*i objektide sortimiseks, arvutüüpidel kasutatakse kahe lahkmega kiirmeetodi varianti.

3.17.1 C++

C++ on päisfailis <algorithm> defineeritud funktsioon sort, millele antakse ette mingi järjendi algus ja lõpp. Näiteks massiivi korral:

```
int n = 5
array size int t[] = {4,2,5,3,5}
sort(t, t+n);
```

vektori (`vector`) korral:

```
sort(v.begin(), v.end());
```

Vaikimisi sorteeritakse elemendid kasvavas järjekorras ja võrdlusoperaatori järgi. Paaride (`pair`) ja ennikute (`tuple`) korral sorteeritakse need vaikimisi esimese elemendi järgi, kui esimesed elemendid on võrdsed, siis teise järgi jne. Enda defineeritud struktuuridel ja objektidel tuleb ise defineerida ka võrdlusoperaator '<'. Samuti saab sort funktsioonile parameetrina ette anda võrdlusfunktsiooni, mis saab ette kaks argumenti ja tagastab tõeväärtuse: tõese, kui esimene argument on väiksem kui teine ja väära muul juhul:

```
bool cmp(string a, string b)
{
    if (a.size() != b.size())
        return a.size() < b.size();
    return a < b;
}
sort(v.begin(), v.end(), cmp);
```

On väga oluline meeles pidada, et sort ei ole stabiilne. Kui on vaja tagada stabiilsus, tuleb kasutada funktsiooni stable_sort, mis enamasti põhineb põimemeetodil.

3.17.2 Java

Javas on päises `Java.Util.Arrays` meetod `Arrays.sort`, millele antakse ette jada elementidest. Kui on vaja sorteerida lõiku jadast, saab ette anda ka alguse ja lõpu indeksid. Tasub tähele panna, et algus kuulub lõiku, lõpp aga mitte.

3.17.3 Python

Pythonis saab sortimiseks kasutada kaht peamist viisi:

- funktsioon `sorted` saab ette järjendi ja tagastab teise, sorteeritud järjendi,
- järjendil endal on meetod `sort`.

Vaikimisi sorteeritakse elemendid kasvavas järjekorras:

```
>>> sorted([4,5,2,4,1])
[1, 2, 4, 4, 5]
```

Mõlemale funktsioonile saab ette anda lipu reverse, mille tõese väärtuse korral jada sorteeritakse kahanevalt.

Lisaks on neil funktsioonidel ka parameeter `key`, mis määrab, mille järgi tegelikult sorteeritakse. `key` on üheargumendiline funktsioon, mille argumendiks on element ja tagastatavaks väärtuseks selle elemendi võti, mida kasutatakse sorteerimiseks.

Kindlasti tasub tutvuda ka mooduliga `operator`, kus on funktsioonid `itemgetter`, `attrgetter` ja `methodcaller`, mis teevad võtme etteandmise oluliselt mugavamaks ja võimaldavad ka mitme parameetri järgi sorteerimist:

```
>>> from operator import itemgetter
>>> opilased = [('Andres', 14 , 8),('Kati', 10, 3),('Mati', 9, 3)]

>>> sorted(opilased, key=itemgetter(2, 0, 1))
[('Kati', 10, 3), ('Mati', 9, 3), ('Andres', 14, 8)]
```

Muidugi võib võtme leidmise funktsiooni alati ise kirjutada. Võtme leidmise funktsioon ei pea olema rangelt objekti juures, vaid saab kasutada ka muid funktsioone:

```
>>> opilased = ['Andres', 'Kati', 'Mati']
>>> hinded = {'Mati': '3', 'Kati':'4', 'Andres': '5'}
>>> sorted(opilased, key=hinded.__getitem__, reverse=True)
['Andres', 'Kati', 'Mati']
```

Python3 versioonides kasutab sort objektide `__lt__()` funktsiooni. Kui oma objektil see meetod ise defineerida, saab määrata ka objektide sorteerimise järjestust. Python2 kasutades on soovitatav defineerida kõik võrdlusfunktsioonid.

Üks näiteülesanne:

Pärtlil on `n` mängukaarti. Ta soovib sorteerida kaardid nii, et kõige peal on potid, siis ärtud, seejärel ruutud ning kõige lõpuks ristid. Lisaks soovib ta, et iga mast on sorteeritud nii, et kõige peal on äss, seejärel kuningas, emand, soldat ja siis numbrid kahanevas järjekorras (10...2). Sisendi esimesel real on kaartide arv `n(1≤n≤1000000).` Järgmisel real on n kahemärgilist stringi, millest igaüks vastab mingile kaardile kaardipakist. Esimene tähemärk tähistab masti (C – risti, D – ruutu, H – ärtu ja S – poti), teine märk väärtust (2...9, T – 10, J – soldat, Q – emand, K – kuningas ja A – äss). Väljundiks on sorteeritud kaardipakk, nii et esimene on pealmine kaart, 2. pealmisest järgmine kaart jne.

NÄIDE:

```
5
C8 D8 SK DQ SK
```

Vastus:

```
SK SK DQ D8 C8
```

```cpp
#include <iostream>
#include <string>
#include <algorithm> //siin on sort meetod
using namespace std;

string mudel = "AKQJT98765432";

bool on_pealpool(string a, string b) // võrdlusfunktsioon kaardistringide võrdlemiseks
{
    if (a[0] != b[0])
        return (a[0] > b[0]); // mastid on tähestiku järjekorras tagurpidi
    else {
       // võrdleme teise tähemärgi positsioone mudelstringis: see, mis on eespool, on
       // väiksem (ehk pakis pealpool)
        return mudel.find(a[1]) < mudel.find(b[1]);
   }
}
```

```
int main()
{
    int n;
    cin >> n;
    string* pakk = new string[n];

    for (int i = 0; i < n; i++)
        cin >> pakk[i];

    // sorteerime paki, kasutades oma võrdlusfunktsiooni
    sort(pakk, pakk + n, on_pealpool);
    for (int i = 0; i < n; i++)
        cout << pakk[i] + ' ';
    return 0;
}
```

3.18 KONTROLLÜLESANDED

3.18.1 Vabrikud

Päikeselinnas on N vabrikut ($1 \le N \le 100$), millest igaüks valmistab oma graafiku alusel kaupa. Täpsemalt kulub igal vabrikul uue kaubapartii valmistamiseks P_i päeva, kus $1 \le i \le N$. Kui kaup on valmis, tullakse sellele veoautoga järele. Igal päeval, mil mõnes vabrikus on kaupa, tuleb veoautojuht ja toob kõigist vabrikutest valminud kaubad ära. Juht töötab aga ainult tööpäevadel, laupäeviti ja pühapäeviti tuleb valminud kaubad lihtsalt kohapeal ära müüa.

Antud on periood, mida uurida ning vabrikute andmed. Leida, mitu reisi peab veoautojuht selle aja jooksul tegema. Aja lugemine algab alati esmaspäevast.

Sisendi esimesel real on uuritava perioodi pikkus T ($1 \le T \le 10000$). Teisel real on vabrikute arv N. Järgmisel N real on arvud P_i ($1 \le Pi \le 1000$).

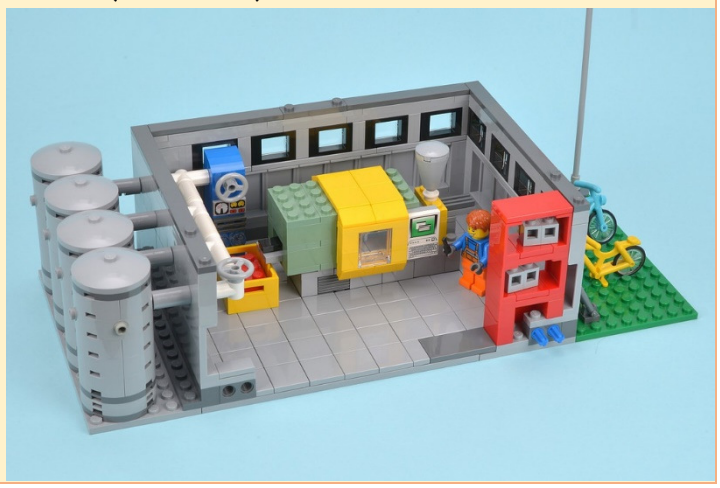
NÄIDE:

14
3
3
4
8

Vastus:

5

(esimene vabrik väljastab kaupa 3., 6., 9. ja 12. päeval, teine vabrik 4., 8. ja 12. päeval, kolmas ainult 8. päeval. Kuues päev jääb lugemata, sest see on laupäev).



3.18.2 Jalgpalliturniir

Jalgpalliturniiril saadakse 3 punkti võidu, 1 punkt viigi ning 0 punkti kaotuse eest. Meeskonnad järjestatakse järgmiste kriteeriumide alusel (kui kõigi eelmiste kriteeriumide tulemus on võrdne, võetakse järgmine)

1. Punktide arv (rohkem on parem)
2. Võitude arv (rohkem on parem)
3. Väravate vahe (suurem on parem)
4. Löödud väravate arv (rohkem on parem)
5. Mängitud mängude arv (vähem on parem)
6. Tähestikuline järjekord

Antud on jalgpalliturniiri mängude tulemused. Leida selle põhjal meeskondade järjestus.

Sisendi esimesel real on meeskondade arv `N(1≤N≤1000)`. Järgmisel `N` real on igaühel ühe meeskonna nimi. Nimed koosnevad kuni 32 märgist ja ainult ladina tähtedest. Meeskondadele järgneval real on mängude arv `M (1≤M≤1000)`. Järgmisel `M` real on igaühel ühe mängu tulemus järgmises formaadis: <Nimi1>-<Nimi2> <skoor1>:<skoor2>, näiteks `Eesti-Soome 1:2` Väljundisse kirjutada meeskondade järjestus ülaltoodud reeglite alusel.

NÄIDE (unistamine on ju lubatud, eks):

```
10
Eesti
Lati
Leedu
Poola
Taani
Norra
Rootsi
Soome
Prantsusmaa
Inglismaa
12
Eesti-Lati 2:1
Lati-Leedu 0:1
Poola-Eesti 1:1
Poola-Eesti 0:0
Poola-Eesti 2:2
Taani-Eesti 3:2
Norra-Lati 2:0
Rootsi-Eesti 3:1
Soome-Lati 2:0
Prantsusmaa-Eesti 0:1
Inglismaa-Eesti 0:1
Prantsusmaa-Lati 0:1
```



Vastus:

```
Eesti
Rootsi
Norra
Soome
Taani
Leedu
Lati
Poola
Inglismaa
Prantsusmaa
```

3.18.3 Erdöse arvud

Ungari matemaatik Paul Erdös (pildil) oli kuulus selle poolest, et ta kirjutas oma elu jooksul palju teadusartikleid (üle 1500), tehes samas koostööd paljude teiste teadlastega. Selle põhjal on defineeritud Erdöse arvu mõiste. Nende teadlaste Erdöse arv, kes Erdösega koos midagi kirjutasid, on 1. Nende Erdöse arv, kes kirjutasid midagi koos Erdöse kaasautoritega, on 2 jne.
On antud hulk teadustöid koos autorite ninedega. Leida iga autori Erdöse arv. Kui autor pole Erdösega seotud, väljastada tema kohta -1.
Sisendi esimesel real on teadustööde arv `N (1<=N<=1000)`. Järgmisel `N` real on igaühel vastava teadustöö info: kõigepealt semikoolonitega eraldatud autorite nimed, siis töö pealkiri.
Väljundisse kirjutada tähestiku järjekorras autorid ja nende Erdöse arvud. Vältimaks kodeeringust tulenevaid probleeme, on Paul Erdös on alati esitatud kui Erdos, Paul. Paul Erdöst ennast pole vaja vastusesse kirjutada.
NB! Vastuses oodatakse, et nimed on harilikus leksikograafilises järjestuses, see tähendab näiteks, et Macaulay, Francis Sowerby on enne kui MacRobert, Thomas Murray.

NÄIDE:

```
10
Brown, Tom C.; Erdos, Paul; Freedman, A. R.; Quasi-progressions and descending
waves.
Li, Yong; Lipmaa, Helger; Pei, Dingyi; On delegatability of four designated
verifier signatures.
Mielikainen, Taneli; Ukkonen, Esko; The complexity of maximum matroid-greedoid
intersection and weighted greedoid maximization.
Brown, Tom C.; Pei, Dingyi; Shiue, Peter; Irrational sums.
Brazma, Alvis; Jonassen, Inge; Vilo, Jaak; Ukkonen, Esko; Pattern discovery in
biosequences
Targo Tennisberg; Katrin Gabrel; Võistlusprogrammeerimine
Bollobas, Bela; Erdos, Paul; Graphs of extremal weights.
Lipmaa, Helger; Mielikainen, Taneli; On private scalar product computation for
privacy-preserving data mining.
Bollobas, Bela; Das, Gautam; Gunopulos, Dimitrios; Mannila, Heikki; Time-series
similarity problems and well-separated geometric sets.
Mannila, Heikki; Ukkonen, Esko; A simple linear-time algorithm for in situ
merging.
```

Vastus:

```
Bollobas, Bela 1
Brazma, Alvis 4
Brown, Tom C. 1
Das, Gautam 2
Freedman, A. R. 1
Gunopulos, Dimitrios 2
Jonassen, Inge 4
Katrin Gabrel -1
Li, Yong 3
Lipmaa, Helger 3
Mannila, Heikki 2
Mielikainen, Taneli 4
Pei, Dingyi 2
Shiue, Peter 2
Targo Tennisberg -1
Ukkonen, Esko 3
Vilo, Jaak 4
```



3.18.4 Programmeerimisvõistlus

Programmeerimisvõistlusel on 1-100 osalejat ning 1-9 ülesannet. Osalejad saavad esitada lahendusi, mida loetakse kas õigeteks või valedeks. Tulemustes on osalejad järjestatud esmalt õigesti lahendatud ülesannete arvu järgi ja kui see on võrdne, siis kulunud aja järgi. Kui aeg on samuti võrdne, on osalejad oma numbrite järjekorras. Kulunud aega mõõdetakse vastavalt sellele, mitmendal võistluse minutil iga õige lahendus esitati. Kui võistleja esitas enne õiget lahendust samale ülesandele ka valesid lahendusi, liidetakse ajale iga vale esituse eest 20 minutit. Kui võistleja esitab valesid lahendusi pärast esimest õiget lahendust, siis nende eest midagi juurde ei liideta.
Sisendis on esimesel real esitatud lahenduste arv `N` ja järgmistel `N` real lahenduste logi ajalises järjestuses. Igal logi real on võistleja number, ülesande number, võistluse algusest kulunud aeg ning tulemus: õige (`C`) või vale (`I`).
Väljundisse kirjutada võistlejate paremusjärjestus vastavalt eespool kirjeldatud reeglitele. Iga võistleja kohta kirjutada tema number, lahendatud ülesannete arv ning kokku kulunud aeg.
NÄIDE:

```
1 2 9 I
3 1 11 C
1 3 19 I
1 2 21 C
1 1 25 C
```

Vastus:

```
1 2 66
3 1 11
```

Esimese võistleja lahendatud kolmas ülesanne ei mõjuta skoori, sest ta ei esitanud sellele ühtki õiget lahendust.

3.18.5 Pannkoogid

Vanaema teeb pannkooke ja asetab need taldrikule kuhja. Pannkoogid on natuke erineva suurusega, tähistame seda positiivse täisarvuga 1-100. Pannkoogikuhja on võimalik pannilabida abil osaliselt või täielikult ümber pöörata.
Olgu meil näiteks kuhi 8 4 6 7 5 2. Kui panna pannilabidas alt kolmanda koogi alla ja selle peale jäävad koogid ümber pöörata, saame kuhja 7 6 4 8 5 2. Kui järgmiseks teha pööre alates esimesest koogist, on kuhi pärast seda 2 5 8 4 6 7.
Sisendi esimesel real on pannkookide arv `N (1<=N<=30)`. Teisel real on pannkookide suurusi tähistavad N arvu. Eesmärgiks on sorteerida pannkoogid nii, et suuremad oleks allpool ja väiksemad peal. Väljundisse kirjutada selleks vajalikud pööramised.



NÄIDE:

```
5
5 4 3 2 1
```

Vastus:

```
1
```

NÄIDE:

```
5
5 1 2 4 3
```

Vastus:

```
1 4 2
```

3.18.6 Kaartide segamine



Tavalises kaardipakis on 52 mängukaarti. 19. sajandi teisel poolel elanud ja vesternitest tuntud kaardimängija Doc Holliday (pildil Val Kilmeri kehastatuna) oskas kaarte segada nii, et nad muutsid oma järjekorda spetsiifilisel viisil. Samas on tema käte liikumise järgi võimalik taibata, millist viisi ta parajasti kasutab. Leida, mis on kaartide lõplik järjekord eeldusel, et Doc Holliday alustab segamist uue pakiga, kus kaardid on sorteeritud.
Sisendi esimesel real on kaks täisarvu: võimalike segamisviiside arv `N (1<=N<=100)` ja segamiste arv `M (1<=M<=1000)`. Järgmisel `N` real on igaühel mingis arvud 1-52, mis näitavad, millisesse järjestusse esialgne pakk seda segamisviisi kasutades segatakse. Lõpuks tuleb `M` rida, millest igaühel on üks täisarv, mis näitab, mitmendat segamisviisi Doc Holliday järjest kasutab.
Väljundisse kirjutada arvud 1-52 sellises järjestuses nagu kaardid lõpuks on.

NÄIDE:

```
2 2
2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 51
52 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 1
1
2
```

Vastus:

```
51 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 2
```

Esimene segamine vahetab omavahel esimesed kaks ja viimased kaks kaarti. Teine segamine vahetab esimese ja viimase kaardi.

3.18.7 Kass klaviatuuril

Juhan kirjutab arvutis teksti, aga ei vaata samal ajal ekraanile. Samal ajal istub laual kass, kes läheb aeg-ajalt `Home` ja `End` klahvide pihta, viies Juhani kursori vastavalt teksti algusse või lõppu. Leida, milline tekst niiviisi lõpuks välja tuleb.



Sisendis on täpselt üks tekstirida, mis koosneb tavalistest tähtedest ja tühikutest, see vastab Juhani kirjutatavale tekstile. Kohad, kus kass vajutas `Home`, on tähistatud märgiga `[` ja kohad, kus kass vajutas `End`, on tähistatud märgiga `]`. Väljundisse kirjutada tegelik tekst.

NÄIDE:

```
tere [maa]ilm
```

Vastus:

```
maatere ilm
```

3.18.8 Pinugrammid

Pinu andmestruktuuri saab kasutada anagrammide koostamiseks. Tähistame pinu operatsiooni *push* tähega `i` ning *pop* tähega `o`. Esialgsest sõnast saab anagrammi järgmiselt:
1. Lükkame kõik selle sõna tähed pinusse.
2. Pinust välja tõmmatud tähtedest koostame uue sõna.

Vastavalt operatsioonide järjekorrale võib saada erinevaid anagramme.
Näiteks esialgsest sõnast TROT võib saada sõna TORT kahel viisil:

```
i i i i o o o o
i o i i o o i o.
```

Sisendi kahel real on antud sõnade paar, mis moodustavad anagrammi. Leida kõik võimalused, kuidas kirjeldatud pinuoperatsioonide abil on võimalik esimene sõna teiseks muuta. Võimalused esitada leksikograafilises järjekorras.

NÄIDE:

```
madam
adamm
```

Vastus:

```
i i i i o o o i o o
i i i i o o o o i o
i i o i o i o i o o
i i o i o i o o i o
```

NÄIDE:

```
bahama
Bahama
```

Vastus:

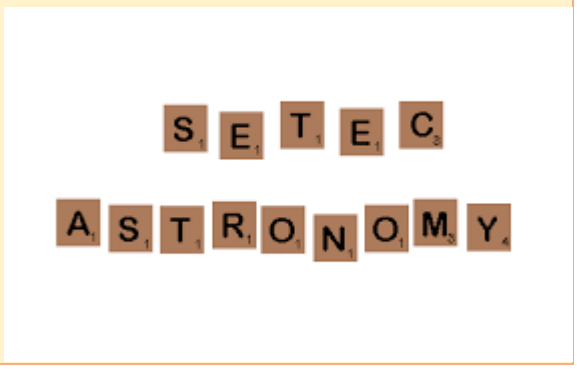
```
i o i i i o o i i o o o
i o i i i o o o i o i o
i o i o i o i i i o o o
i o i o i o i o i o i o
```

NÄIDE:

```
eric
rice
```

Vastus:

```
i i o i o i o o
```



3.18.9 Parv

Enamasti ületavad autod jõgesid sildade kaudu. Mõnes kohas pole siiski sildu ehitatud ja nende asemel kasutatakse parvesid. Kui ühtki autot ei ole, siis parv lihtsalt ootab. Kui autosid saabub, läheb parv vajadusel õigele kaldale, võtab auto(d) peale ning toimetab nad üle. Kui teisel pool on samal ajal autosid ootamas, võtab parv nad peale ja toob üle. Parv mahutab korraga N autot ja jõe ületamine võtab T minutit. Eeldame lihtsuse mõttes, et parve peale ja parvelt maha sõit aega ei võta. Alguses on parv jõe vasakul kaldal.

Sisendi esimesel real on kolm täisarvu: parvele korraga mahtuvate autode arv `N`, jõe ületamiseks kuluv aeg `T` ja autode koguarv `M`. Järgmistel `M` real on iga auto saabumise aeg ja kallas, kuhu ta saabub (`L` või `R`). Väljundisse kirjutada iga auto jaoks, millisel ajal ta vastaskaldale jõuab.

NÄIDE:

```
2 10 10
0 L
10 L
20 L
30 L
40 L
50 L
60 L
70 L
80 L
90 L
```

Vastus:

```
10
30
30
50
50
70
70
90
90
110
```



NÄIDE:

```
10 R
25 L
40 L
```

Vastus:

```
30
40
60
```

3.18.10 Ahelmurrud

Ahelmurruks nimetatakse avaldist kujul

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

Ahelmurdu saab lühendatult kirja panna kujul [a_0, a_1, a_2,…,a_n]. Näiteks järjend [2, 3, 1, 4] on võrdne murruga

$$2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{4}}} = \frac{43}{19}$$

Iga lihtmurdu on võimalik teisendada selliseks ahelmurruks. Antud on kaks positiivset täisarvu: lihtmurru lugeja ja nimetaja, leida sellele vastav ahelmurd ja väljastada see järjendiesituses.

NÄIDE:

```
43 19
```

Vastus:

```
2 3 1 4
```

NÄIDE 2:

```
1 2
```

Vastus:

```
0 2
```

3.19 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk3 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Lemming.cpp, Lemming.java, Lemming.py	Lemmingute ülesanne (pinu ja järjekord)
Kursused.cpp, Kursused.java, Kursused.py	Usina ülikooli ülesanne (kujutis)
Sortimine.cpp, Sortimine.java, Sortimine.py	Erinevad sorteerimismeetodid
Kaardid.cpp, Kaardid.java, Kaardid.py	Kaardipaki ülesanne (oma võrdlemine)

4 ARVUTEOORIA



Pierre de Fermat

Arvuteooria on matemaatika haru, mis tegeleb peamiselt täisarvudega, samuti ka objektidega, mis koosnevad täisarvudest. Sellised objektid on näiteks ratsionaalarvud ehk arvud, mida saab esitada kahe täisarvu jagatisena.

Kaasaegsele arvuteooriale panid aluse Pierre de Fermat (1601-1665) ja Leonhard Euler (1707-1783).

Füüsika on ajalooliselt põhinenud pigem reaalarvude ehk pideval matemaatikal, mistõttu viimane oli ka ülikoolide matemaatikakursustes esikohal.

Pärast arvutite leiutamist on arvuteooria ja selle üldistused nagu diskreetne matemaatika oma tähtsust aga suurendanud. Arvutid töötavad enamasti pigem diskreetsete objektidega ja ka kõik arvutis esinevad arvud on piisavalt madalale tasemele minnes täisarvud.

Donald Knuth ütles 1974. aastal: "Peaaegu kõik arvuteooria teoreemid on loomulikul viisil tarvilikud seoses vajadusega panna arvuteid sooritama võimalikult kiireid arvutusi".



Leonhard Euler

Kuna käesoleva raamatu läbivaks teemaks ongi kiirete arvutuste tegemine, on matemaatika, eriti arvuteooria tundmine suureks abiks. Eesti-suguses pisikeses riigis on ka hästi märgata, et need, kes saavad häid tulemusi informaatikaolümpiaadidel, on enamasti väga tugevad ka matemaatikas. Kindlasti pole see ainult juhus!

Matemaatika annab hulgaliselt kavalaid vahendeid, kuidas efektiivsemalt arvutada. Kuna arvutid suudavad teostada erinevaid tehteid ülikiiresti, siis võib tunduda, et neid teadmisi enam eriti vaja ei lähegi, kuid mitmete nippide abil on siiski võimalik programmi tööaega märgatavalt lühendada.

Vahel on ülesannete põhirõhk kusagil mujal, aga arvuteooria alased teadmised ja rakendused on mõne detaili juures abiks. Sageli on programmeerimisvõistlustel aga ka ülesandeid, mis ongi oma põhisisult matemaatikaülesanded. Siin peatükis käsitletakse programmeerimisülesannetes sagedamini ette tulevaid kontseptsioone ja meetodeid.

4.1 JAGUVUS JA JÄÄK

4.1.1 Jaguvus

Jaguvus on arvuteooria keskseks mõisteks. Öeldakse, et täisarv n **jagub** täisarvuga m, kui leidub selline täisarv k, et $n = km$. Jaguvust võib tähistada kahel moel: $n : m$ (arv n jagub arvuga m) või $m|n$ (arv m jagab arvu n; arv m on arvu n jagaja). Näiteks arv 12 jagub arvuga 4, sest $3 \cdot 4 = 12$. Arv 5 aga ei jaga arvu 12, sest ei leidu sellist täisarvu k, et $k \cdot 5 = 12$.

Jaguvusel on järgmised olulised põhiomadused:

1. $a|a$ – mistahes täisarv jagub iseendaga, näiteks $12|12$.
2. $1|a$ – arv 1 jagab mistahes täisarvu, näiteks $1|31$.
3. $a|0$ – null jagub mistahes arvuga, näiteks $5|0$.
4. Kui $c|a$ ja $c|b$, siis $c|(a+b)$ ja $c|(a-b)$.
5. Kui $c|(a+b)$ ja $c|a$, siis $c|b$.
6. Kui $c|a$ ja $c \nmid b$, siis $c \nmid (a+b)$.
7. Kui $c|a$, siis $c|ka$, kus $k \in Z$.
8. Kui $a|b$ ja $b|c$, siis $a|c$.
9. Kui $a|b$ ja $c|d$, siis $ac|bd$
10. Kui $a|b$ ja $b|a$ ja $a, b \in N$, siis $a = b$.
11. Kui $b|a$ ja $b = cd$, siis $c|a$ ja $d|a$.
12. Kui $a|c$ ja $b|c$ ja ei leidu sellist arvu $k > 1$, et $k|a$ ja $k|b$ (a ja b on ühistegurita), siis $ab|c$.
13. Kui $d|a$ ja $b|d$, siis $d/b|a/b$.
14. Kui on antud n järjestikust arvu, siis jagub alati täpselt üks neist arvuga n.

Enamik neist paistavad algul triviaalsed, kuid nende abil saab hiljem põhjendada juba oluliselt huvitavamaid tulemusi.

4.1.2 Jääk

Kui meil on täisarv n ja naturaalarv $m > 0$, saab alati leida täisarvud k ja r, nii et $n = km + r$, kus $|r| < m$.

Sellisel juhul nimetatakse k **jagatiseks** (n/m), m **mooduliks** ning r **jäägiks** mooduli m järgi. Juhul, kui $r = 0$, siis $n \vdots m$. Näiteks $12 = 2 \cdot 5 + 2$, seega arvu 12 jääk mooduli 5 järgi on 2. Arvu 12 jääk mooduli 8 järgi on 4, sest $12 = 1 \cdot 8 + 4$.

Jäägi absoluutväärtus on alati väiksem mooduli absoluutväärtusest. Näiteks arvu 12 saab esitada ka kujul $12 = 1 * 5 + 7$, kuid 7 ei ole jääk mooduli 5 järgi, sest $7 > 5$.

4.1.3 Jäägi leidmine programmeerimiskeeltes

Arvust jäägi leidmine mooduli järgi on programmeerimises väga tavapärane tehe. Siin raamatus käsitletavates keeltes kasutatakse tehtemärgina %-märki. Tehe `7 % 3` annab vastuseks ühe.

Positiivsete arvudega on asi alati ühtmoodi, kuid negatiivsete arvude korral on oluline teada, kuidas käitub konkreetne programmeerimiskeel. Peamiselt on kasutuses kolm varianti:

1. Jagatis ümardatakse alati nulli suunas. Sellisel juhul jääk on sama märgiga, mis jagatav: $r = a - n\ trunc(a/n)$. Seda varianti kasutavad nii Java kui ka C++ alates versioonist 11. Varasemates C++ versioonides jäeti selles osas vabad käed ning kasutus oleneb konkreetsest keele implementatsioonist.
2. Jagatis ümardatakse alati alla. Sellisel juhul jääk on sama märgiga, mis jagaja: $r = a - n \lfloor a/n \rfloor$. Seda varianti kasutab Python.
3. Jääk on alati positiivne. Sellisel juhul positiivse jagaja korral jagatis ümardatakse alla, negatiivse jagaja korral aga üles:
$$q = \begin{cases} \lfloor a/n \rfloor, kui\ n > 0, \\ \lceil a/n \rceil, kui\ n < 0, \end{cases}$$
ja jääk avaldub kujul $r = a - |n| \lfloor a/|n| \rfloor$.

Järgnevas tabelis on toodud näited tulemustest erinevatel juhtudel:

Tehe	Jagatava järgi	Jagaja järgi	Alati positiivne
5 % 3	2	2	2
-5 % 3	-2	1	1
5 % -3	2	-1	2
-5 % -3	-2	-2	1

Pane tähele, et negatiivsest tulemusest saab vajadusel alati positiivse, liites leitud jäägile juurde mooduli ehk jagaja absoluutväärtuse. Kui peaks olema vaja positiivsest negatiivset saada, siis piisab, kui lahutada mooduli absoluutväärtus.

Kui esmapilgul tundub tegemist olevat küllaltki ebaolulise detailiga, siis algajatel programmeerijatel võib tekkida sellega seoses ootamatuid vigu. Näiteks kirjutades funktsiooni `OnPaaritu` paarsuse kontrolliks kujul:

```
bool OnPaaritu(int a) {
    return a % 2 == 1;
}
```

võib selguda, et negatiivsete täisarvude korral tagastab see funktsioon alati väärtuse `false`, kuna näiteks `-1 % 2` tulemuseks on `-1`. Kindlam on alati kirjutada:

```
bool OnPaaritu(int a) {
    return a % 2 != 0;
}
```

4.1.4 Ülesanne: loosiümbrikud

Järgmiseks üks lihtne soojendusülesanne:



Suures kastis on hulk ümbrikke. Mõned neist on tühjad ja mõned täidetud. Lapsed võtavad ükshaaval kastist ümbrikke. Iga laps võtab korraga kastist täpselt 2 ümbrikku ning juhul, kui need on mõlemad täis või mõlemad tühjad, paneb ta kasti tagasi tühja ümbriku (kui mõlemad on täis, eemaldab ta ühest sisu ning paneb tühjendatud ümbriku tagasi), kui aga üks ümbrik on tühi ja teine täis, peab ta tagasi panema täidetud ümbriku. Lõpuks jääb kasti üks ümbrik. Kas see on tühi või täis? Sisendi esimesel ja ainsal real on kaks täisarvu `m` ja `n`, vastavalt tühjade ja täidetud ümbrike arv. Väljastada `TÜHI` kui viimane ümbrik on tühi, `TÄIS`, kui viimane ümbrik on täis või `EI TEA` , kui ei ole võimalik otsustada, kas ümbrik on tühi või täis.

NÄIDE :

`2 1`

Vastus:

`TÄIS`

(Kui esimene laps võtab mõlemad tühjad ümbrikud, siis ta paneb neist ühe tagasi ja teisele lapsele jääb kasti üks tühi ja üks täis ümbrik, millest ta peab tagasi panema täis ümbriku. Kui esimene laps võtab tühja ja täis ümbriku, siis ta peab tagasi panema täidetud ümbriku ja nii jääb teisele lapsele samuti tühi ja täis ümbrik, millest ta peab tagasi panema täidetud ümbriku. Nii jääb igal juhul lõpuks kasti täidetud ümbrik).

Kuigi esmapilgul võib see ülesanne tunduda üsna keerukas, siis lähemal vaatlusel pole raske märgata, et täis ümbrikud lahkuvad kastist ainult paarikaupa – kui keegi õnnelik saab korraga kaks täidetud ümbrikku. Samuti ei teki võtmise ajal täidetud ümbrikke juurde.

Viimasel võtmisel on kolm võimalust: mõlemad ümbrikud on tühjad, mõlemad on täis või on üks ümbrik täis ja üks tühi. Kui mõlemad ümbrikud on tühjad või täis, läheb kasti tagasi tühi ümbrik ja viimane ümbrik kastis on seega tühi. Kui üks ümbrik on tühi ja teine täis, siis sel juhul läheb kasti tagasi täis ümbrik, mis jääb sinna viimaseks. Kuna täis ümbrikud lahkuvad kastist ainult paarikaupa, siis saab lõpuks alles olla üks tühi ja üks täis ümbrik ainult siis, kui täis ümbrikke oli kastis kohe alguses paaritu arv. Kehtib ka vastupidine – kui täis ümbrikke oli alguses paaritu arv, siis peab enne viimast võtmist olema kastis täpselt üks täidetud ümbrik. Seega piisab kogu ülesande lahendamiseks ainult sellest, et vaatame, kas täidetud ümbrikke oli algselt kastis paaris või paaritu arv:

```
#include <iostream>

using namespace std;
int main()
{
    int tyhjad, tais;
    cin >> tyhjad >> tais;
    if (tais % 2 == 0) // Kui täis ümbrike on paarisarv,
        cout << "TÜHI"; // siis viimane ümbrik on tühi,
    else // muidu (kui on paaritu)
        cout << "TÄIS"; // on viimane ümbrik täis
    return 0;
}
```

Sellise ülesande lahendamiseks pole tegelikult muidugi isegi arvutit vaja, kuid võistlustel (ja ka päriselus!) esineb siiski aegajalt ülesandeid, kus keerulise vormi sisse on peidetud väga lihtne lahendus. Sarnased võtted võivad sageli osutuda oluliseks ka keerulisemate ülesannete jaoks optimaalsemate algoritmite leidmisel.

4.1.5 Arvutamine moodulitega

Sageli saab tehte jääki leida tehte operandide järgi, ilma tehte vastust välja arvutamata. Selleks on kõigepealt vaja teada kaht lihtsat, kuid olulist seost:

1. $a \bmod m = a,\ kui\ 0 \leq a < m$
2. $(a \bmod m) \bmod m\ =\ a \bmod m$

Teine seos tähendab, et mitmekordsel jäägi võtmisel sama mooduli järgi jääb tulemus samaks. Kuna $a \bmod m$ tulemus on hulgas $(0, 1 \ldots m-1)$, ning kehtib esimene seos, siis pole selle kehtivuses raske veenduda. Näiteks

$$(17 \bmod 3) \bmod 3\ =\ 2 \bmod 3\ =\ 2$$

Liitmine ja lahutamine

Liitmise jaoks saab kasutada seost:

$$(a + b) \bmod m\ =\ [(a \bmod m) +\ (b \bmod m)] \bmod m.$$

Näiteks:

$$(17 + 8) \bmod 3 = [(17 \bmod 3) + (8 \bmod 3)] \bmod 3 = (2 + 2) \bmod 3 = 4 \bmod 3 = 1.$$

Lahutamine on põhimõtteliselt sarnane liitmisele:

$$(17 - 8) \bmod 3 = [(17 \bmod 3) - (8 \bmod 3)] \bmod 3 = (2 - 2) \bmod 3 = 0 \bmod 3 = 0.$$

Aga mis teha, kui $a \ mod \ m < b \ mod \ m$, nagu tehtes $(9 - 7) \ mod \ 3$? Sellisel juhul

$$(9 - 7) \ mod \ 3 = [(9 \ mod \ 3) - (7 \ mod \ 3)] \ mod \ 3 = (0 - 1) \ mod \ 3 = -1 \ mod \ 3 = ?$$

Nagu juba teame, sõltub tehte $-1 \ mod \ 3$ vastus juba konkreetsest programmeerimiskeelest või lausa implementatsioonist: Python annab vastuseks sobivalt 2, Java aga -1. Sellisel juhul saab negatiivsest jäägist positiivse, liites vastusele juurde mooduli, antud näites siis $-1 + 3 \ = \ 2$.

Kindla peale kehtib seos:

$$(a - b) \ mod \ m = [(a \ mod \ m) - \ (b \ mod \ m) + \ m] \ mod \ m,$$

millest saame, et

$$(9 - 7) \ mod \ 3 = [(9 \ mod \ 3) - (7 \ mod \ 3) + 3] \ mod \ 3 = (0 - 1 + 3) \ mod \ 3 = 2 \ mod \ 3 = 2.$$

Korrutamine ja astendamine

Korrutamise puhul kehtib samuti seos:

$$(ab) \ mod \ m \ = \ [(a \ mod \ m)(b \ mod \ m)] \ mod \ m$$

Näiteks:

$$(5 * 8) mod \ 3 = [(5 \ mod \ 3) \cdot (8 \ mod \ 3)] mod \ 3 = (2 \cdot 2) mod \ 3 = 4 \ mod \ 3 = 1$$

Astendamise puhul kehtib seos:

$$a^b \ mod \ m \ = \ (a \ mod \ m)^b \ mod \ m$$

Näiteks:

$$5^4 \ mod \ 3 = (5 \ mod \ 3)^4 \ mod \ 3 = 2^4 \ mod \ 3 = 16 \ mod \ 3 = 1$$

Kuna just korrutades ja eriti astendades lähevad arvud kiiresti väga suureks, siis on see koht, kus moodularvutusest on sageli abi. Küllaltki levinud on ülesanded, kus tuleb leida mingi suure arvu, näiteks 574232^{100}, viimane koht. Viimase ehk üheliste koha leidmine on tegelikult jäägi leidmine mooduli 10 järgi (kahe viimase koha leidmine on ekvivalentne jäägi leidmisega mooduli 100 järgi jne).

Jagamine

Jagamine on mõnevõrra keerulisem, liitmise ja korrutamisega analoogsed seosed jagamise puhul ei kehti. Näiteks:

$$(30 \div 2) \ mod \ 8 = 15 \ mod \ 8 = 7,$$

aga

$$[(30 \ mod \ 8) \div \ (2 \ mod \ 8)] \ mod \ 8 \ = \ (6 \div 2) \ mod \ 8 \ = \ 3 \ mod \ 8 = 3.$$

Jagamise puhul tuleb appi võtta **modulaarne pöördväärtus**. Arvu a modulaarne pöördväärtus mooduli m suhtes on selline arv a^{-1}, mille korral kehtib seos $a \cdot a^{-1} \ mod \ m = 1$.

Jagamisel kehtib seos:

$$(a \div b) \ mod \ m = [(a \ mod \ m) \cdot (b^{-1} \ mod \ m)] \ mod \ m,$$

kus b^{-1} on arvu b modulaarne pöördväärtus mooduli m suhtes.

Näiteks:

$$(20 \div 5) \ mod \ 7 = [(20 \ mod \ 7) \cdot (3 \ mod \ 7)] \ mod \ 7 = (6 \cdot 3) \ mod \ 7 = 18 \ mod \ 7 = 4,$$

Kus arv 3 on arvu 5 modulaarne pöördväärtus mooduli 7 suhtes, sest $(3 \cdot 5) \ mod \ 7 = 1$.

Arvul eksisteerib modulaarne pöördväärtus parajasti siis, kui arv ja moodul on ühistegurita.

4.1.6 Ülesanne: anagrammid 2

Mõnikord on arvutusülesande vastus nii suur, et see ei mahu suurimassegi olemasolevasse täisarvutüüpi ära või ei peeta praktiliseks mitmekümnekohalist vastust küsida. Sageli küsitakse programmeerimisvõistlustel vastust mingi etteantud mooduli järgi.

Teises peatükis punkt „2.7.2 Permutatsioonid“ all oli ülesanne, kus tuli leida etteantud nimest kõik võimalikud anagrammid. Selles ülesandes jäid nimede pikkused küllaltki lühikesteks. Vaatame selle ülesande suuremat varianti, kus tähti, millest anagramme moodustatakse, on juba palju rohkem:



Ingliskeelses Scrabble'i komplektis on 100 täheklotsi: E tähega klotse on kõige rohkem - 12, A ja I tähega on 9, O-ga 8, N-i, R-i ja T-ga 6, D, L-i, S-i ja U-ga 4 klosti ja G-ga kolm klotsi. Kaks korda on B, C, F, H, M, P, V, W ja Y. J, K, Q, X ja Z esinevad ainult ühe korra. Lisaks on komplektis 2 tühja klotsi. Scrabble'i komplekt on olemas ka Eesti keele jaoks, selles on 102 tähte ning kõige rohkem on A tähega klotse – 10. Klotside jaotus on olemas ka tehiskeelte, näiteks Klingoni keele jaoks.
Väikesel Pärdil on ingliskeelne Scrabble'i komplekt. Lugeda ta veel ei oska, kuid talle meeldib neid klotse ühte ritta laduda, nii et moodustub üks n-täheline „sõna“. Pärdi õde Piret soovib aga välja arvutada, mitu erinevat n-tähelist „sõna“ on Pärdil võimalik väljavalitud klotsidest moodustada. Pärt ei vali kunagi tühje klotse.
Sisendi esimesel ja ainsal real on 26 arvu. Esimene arv on A-tähega klotside arv, teine B tähega klotside arv, jne, vastavalt inglise keele tähestikule. Viimane on seega Z-tähega klotside arv. Arvud on eraldatud tühikutega. Klotsid moodustavad alamhulga ingliskeelsest Scrabble'i komplektist.
Väljundisse kirjutada üks arv: Erinevate n-täheliste järjendite arv mooduli 10^9+7 järgi.

NÄIDE 1:

```
2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Vastus:

12

(AABC, AACB, ABAC, ABCA, ACAB, ACBA, BAAC, BACA, BCAA, CAAB, CABA, CBAA)

NÄIDE 2:

```
9 2 2 4 12 2 3 2 9 1 1 4 2 6 8 2 1 6 4 6 4 2 2 1 2 1
```

Vastus:

```
820218216
```

(Tegelik võimaluste arv on ligikaudu 9.74×10^{111})

Sel korral kõiki permutatsioone leida ei ole vaja, on vaja ainult nende arvu. Nagu juba 2. peatükis selgus, on selle jaoks olemas järgmine valem:

$$\frac{n!}{m_1!\, m_2! \ldots m_k!}$$

Nii ei ole tegelikult oluline teada, millised tähed konkreetselt kasutuses on, on vaja vaid täheklotside koguarvu ning iga erineva klotsi esinemissagedust.

Väikese n korral on tegu väga lihtsa arvutusülesandega, problemaatiliseks teeb antud ülesande see, et juba 22! on 21-kohaline ega ei mahu kuidagi suurimassegi täisarvutüüpi ära. Ülesandes aga võib murru lugejaks tulla lausa 98!, mis on juba 154-kohaline arv.

Samas ega täpset vastust leida polegi ju vaja, piisab jäägist mooduli 10^9+7 järgi. Eelnevalt tuli välja, et korrutamine jääkidega ei ole kuigi keeruline. Isegi 100! leidmine mooduli 1 000 000 007 järgi ei ole kuigi vaevanõudev, kasutades järgnevat funktsiooni:

```
int fakt(int n, int m)
{
    int i = 1;
    long long f = 1;
    while (i < n) {
        f *= ++i;
        f %= m; // iga korrutamise järel leiame jäägi
    }
    return f;
}
```

Keeruliseks teeb asja jagamistehe, sest nagu eespool selgus, siis pole lugeja ja nimetaja jääkide leidmisest eraldi kuigi palju kasu.

Murru $\frac{n!}{m_1!m_2!\ldots m_k!}$ saab teisendada täisarvude korrutiseks, kasutades omadust, et n järjestikuse täisarvu korrutis jagub alati $n!$-ga (huvi korral loe lähemalt https://math.stackexchange.com/questions/12065/the-product-of-n-consecutive-integers-is-divisible-by-n-factorial).

Näiteks kui on 10 klotsi: esimest tähte 4, teist 3, kolmandat 2 ja neljandat 1, siis erinevate järjendite arv on

$$\frac{10!}{4!\,3!\,2!\,1!} = \frac{1\cdot 2\cdot 3\cdot 4}{4!}\cdot\frac{5\cdot 6\cdot 7}{3!}\cdot\frac{8\cdot 9}{2!}\cdot\frac{10}{1!} = 1\cdot 35\cdot 36\cdot 10 = 12600.$$

Klotsid on hea esinemissageduse järgi sorteerida, nii saab paremini kontrollida, kui suureks tehete tulemused lähevad. Siinne näidislahendus annab õige vastuse ka siis, kui 12 E-tähe asemel on hoopis 12 Z-tähte, kuid sorteerimata jättes võib tekkida olukord, kus viimaseks lugejaks jääb $89\cdot 90\cdot\ldots\cdot 100$, mis ei mahu enam 64-bitisesse arvutüüpi ära.

Ülesande lahendus:

```
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    long long moodul = 1e9 + 7; // moodul, mille järgi vastuse leiame
    int klotsid[26] = {0}; // massiv erinevate tähtede arvu jaoks
    int tahti = 0; // erinevaid kasutusesolevaid tähti
    int i, j; // tsüklimuutujad
    int algus = 1; // abumuutuja n! tükeldamisel: uue korrutise algus
    int n = 0; // klotside arv kokku
    long long vastus = 1; // ülesande vastus
    for (i = 0; i < 26; i++) {
        int s;
        cin >> s;
        // kui seda tähte ei ole, siis jätkame järgmise lugemisega
        if (s == 0) continue;
        klotsid[tahti] = s;
        tahti++; // suurendame erinevate tähtede arvu
        n += s; // liidame kogu klotside arvule selle tähega klotside arvu
    }
    sort(klotsid, klotsid + tahti); // sorteerime suuruse järgi
    for (i = tahti-1; i >= 0; i--) { // alustame sagedasemast
        int lopp = algus + klotsid[i];
        long long f = 1; // järgmise klotsid[i] arvu korrutis
        long long f2 = 1; // m[i] faktoriaal
        for (j = algus; j < lopp; j++) {
            f *= j;
        }
        for (j = 2; j <= klotsid[i]; j++) {
            f2 *= j;
        }
        long long t = (f / f2) % moodul;
        vastus *= t;
        vastus = vastus % moodul;
        algus = lopp;
    }
    cout << vastus;
    cin >> i;
    return 0;
}
```

Tasub ka tähele panna, et esimene jagatis on tegelikult alati 1, täiesti mõistlik on see arvutamata jätta. Selleks piisab, kui põhitsüklit alustada `i = tahti-2` ning enne tsüklit muuta muutuja `algus` väärtust: `algus += klotsid[tahti-1];`.

4.2 SÜT JA VÜK

Arvuteoorias tähtsateks mõisteteks, millega ka koolimatemaatikas juba üsna varakult kokku puututakse, on suurim ühistegur (SÜT) ja vähim ühiskordne(VÜK).

Naturaalarvude a ja b **suurimaks ühisteguriks** nimetatakse suurimat naturaalarvu c, nii et $c|a$ ja $c|b$. Naturaalarvude a ja b **vähimaks ühiskordseks** nimetatakse vähimat naturaalarvu c, nii et $a|c$ ja $b|c$.

Koolimatemaatikas ja ka igapäevaelus on suurimat ühistegurit vaja enamasti murdude taandamiseks ning vähimat ühiskordset erinimeliste murdude liitmisel ühise nimetaja leidmiseks.

4.2.1 Eukleidese algoritm suurima ühisteguri leidmiseks

Kreeka matemaatik Eukleides Aleksandriast märkas juba 300 aastat enne meie aega, et arvude a ja b suurim ühistegur c jagab lisaks arvudele a ja b ka nende vahet $a - b$. See on tegelikult triviaalne, sest kui $c|a$, siis jaguvuse definitsiooni järgi $a = xc$ ja analoogselt, kui $c|b$, siis $b = yc$, seega

$$a - b = xc - yc = c(x - y) \implies c|(a - b).$$

Praktikas tähendab see, et kui meil on vaja leida kahe suure arvu a ja b $(a > b)$ suurimat ühistegurit, siis saab ülesande taandada lihtsamale kujule. Nimelt on a ja b suurim ühistegur sama, mis $a - b$ ja b ühistegur. Sama võtet võib korrata korduvalt. Näiteks arvude 63 ja 49 suurima ühisteguri saab leida nii:

$$\begin{aligned} SÜT(63,49) = SÜT(63-49,49) &= SÜT(14,49) = SÜT(14,49-14) = SÜT(14,35) \\ &= SÜT(14,35-14) = SÜT(14,21) = SÜT(14,21-14) = SÜT(14,7) \\ &= SÜT(14-7,7) = SÜT(7,7) = 7. \end{aligned}$$

Siin on täpselt sama vahetu lahendus C++ funktsioonina:

```
int syt(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        }
        else {
            b -= a;
        }
    }
    return a;
}
```

Ülaltoodud näites torkab silma, et näiteks arvu 14 lahutatakse mitu korda järjest, õigemini nii kaua, kuni saadud vahe on ≤14. Seega võiks ju lahutada kohe sobiva 14-kordse, antud näites siis $3 \cdot 14 = 42$. Veelgi enam: kui lahutada arvust a mingi arvu b kordset, kuni vahe on väiksem kui lahutatav arv, siis tegelikult leitakse arvu a jääki mooduli b järgi. Jäägi definitsioonist:

$$a = kb + r \implies a - kb = r\,,$$

Kus r on a jääk mooduli b järgi.

Suurima ühisteguri leidmine jäägi abil annab kompaktse rekursiivse lahenduse:

```
int syt(int a, int b) {
     return (b == 0) ? a : syt(b, a % b);
}
```

4.2.2 Vähima ühiskordse leidmine

Kahe arvu vähima ühiskordse leidmisel kasutatakse omadust, et kahe arvu suurima ühisteguri ja vähima ühiskordse korrutis on võrdne nende arvude korrutisega:

$$a \cdot b = SÜT(a,b) \cdot VÜK(a,b),$$

millest

$$VÜK(a,b) = \frac{a \cdot b}{SÜT(a,b)}.$$

Seda omadust kasutatakse programmides enamasti vähima ühiskordse leidmiseks:

```
int vyk(int a, int b) {
      int temp = syt(a, b);
      return temp ? (a / temp * b) : 0;
}
```

Siin tasub tähele panna, et `a*b/syt(a,b)` asemel tagastatakse `a/syt(a,b)*b` . Sellisel moel hoitakse arvud väiksemad ja välditakse nii võimalikku ületäitumist. Kuna definitsiooni järgi $SÜT(a,b)|a$, siis võib rahulikult jagamistehte `a/syt(a,b)` sooritada enne `b`-ga korrutamist.

Suurimat ühistegurit ja vähimat ühiskordset saab leida ka rohkem kui kahel arvul. Mitme arvu suurim ühistegur on suurim selline arv, mis jagab igaüht neist arvudest, ning vähim ühiskordne on kõige väiksem selline arv, mida kõik need arvud jagavad. $SÜT(a,b,c) = SÜT(a,SÜT(b,c))$ ja samuti $VÜK(a,b,c) = VÜK(a,VÜK(b,c))$.

4.2.3 Ülesanne: hammasrattad

Järgmine ülesanne „Hammasrattad“ oli 2017. aastal Eesti informaatikaolümpiaadi lõppvoorus:

Kellassepal on tööpink, mis suudab teha `M` kuni `N` hambaga hammasrattaid. Kirjutada programm, mis leiab, mitu erinevat kahest hammasrattast koosnevat ülekannet saab selle pingi abil teha. Kahte ülekannet loeme erinevaks, kui nende ülekandearvud (esimese ratta hammaste arv jagatud teise ratta hammaste arvuga) on erinevad.
Sisendi esimesel real on kaks tühikuga eraldatud täisarvu `M` ja `N` (`1≤M≤N≤1000`), mis tähistavad minimaalset ja maksimaalset hammaste arvu hammasratastel, mida antud pingil teha saab.



Väljundi ainsale reale väljastada võimalike ülekandesuhete arv.
NÄIDE:
`2 6`
Vastus:
`17`
(Võimalikud suhted on järgmised: 2 : 6; 2 : 5; 2 : 4 = 3 : 6; 3 : 5; 2 : 3 = 4 : 6; 3 : 4; 4 : 5; 5 : 6; 2 : 2 = 3 : 3 = 4 : 4 = 5 : 5 = 6 : 6; 6 : 5; 5 : 4; 4 : 3; 3 : 2 = 6 : 4; 5 : 3; 4 : 2 = 6 : 3; 5 : 2 ja 6 : 2.)

Lihtsa topelttsükliga saame konstrueerida kõik võimalikud hammasrataste paarid. Ülesande keerukamaks kohaks on see, kuidas leida sama suhtega paare. Suhteid saab vaadelda murdudena ja nii on sama suhtega paaride tuvastamine tegelikult murru taandamise ülesanne. Taandada on aga kõige lihtsam nii, et leiame lugeja ja nimetaja suurima ühisteguri ja jagame nii lugeja kui ka nimetaja sellega läbi. Kuna võib tulla mitmeid ekvivalentseid ülekandeid, siis on kõige mugavam hoida ülekandeid `set` andmestruktuuris – seal võib iga element esineda täpselt ühe korra ning katset lisada korduvat elementi ignoreeritakse.

Siin on programm, mis antud ülesande lahendab:

```
#include <iostream>
#include <set>

using namespace std;

int syt(int a, int b) {
    while (b > 0) {
        int c = b;
        b = a % b;
        a = c;
    }
    return a;
}

int main()
{
    int m, n;
    cin >> m >> n;

    std::set<pair<int, int>> s;
    for (int a = m; a <= n; a++) { // esimese hammasratta võimalikud hambad
        for (int b = m; b <= n; b++) { // teise hammasratta võimalikud hambad
            int d = syt(a, b); // leiame a ja b suurima ühisteguri
            // ja jagame a ja b sellega, et leida ülekannet
            pair<int, int> p = make_pair(a / d, b / d);
            s.insert(p);
        }
    }
    cout << s.size();
    return 0;
}
```

4.2.4 Diofantilised võrrandid

Diophantos Aleksandriast oli vanakreeka matemaatik, kes tegeles 3. sajandil algebraliste võrrandite lahendamisega. Tema järgi nimetataksegi täisarvuliste kordajatega algebralist määramata võrrandit, millele otsitakse täisarvulisi lahendeid, **diofantiliseks võrrandiks.** Võrrandit kujul $ax + by = c$ nimetatakse **lineaarseks diofantiliseks võrrandiks**. Koolis õpitakse, et võrrandi(süsteemi) lahendamiseks peab võrrandeid olema vähemalt sama palju kui tundmatuid. Hoolimata sellest, et antud juhul on ühe võrrandi kohta kaks tundmatut, on sellisel spetsiifilisel kujul võrranditele võimalik üldlahendid leida.

Selleks on kõigepealt vaja leida a ja b suurim ühistegur. Olgu see d. Võrrandil puuduvad lahendid, kui $d \nmid c$. Kui aga $d|c$, on võrrandil lõpmatu hulk lahendeid. Lahendamiseks saab kasutada veidi täiendatud Eukleidese algoritmi suurima ühisteguri leidmiseks:

```
int dsyt(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    int x1, y1, syt = dsyt(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return syt;
}
```

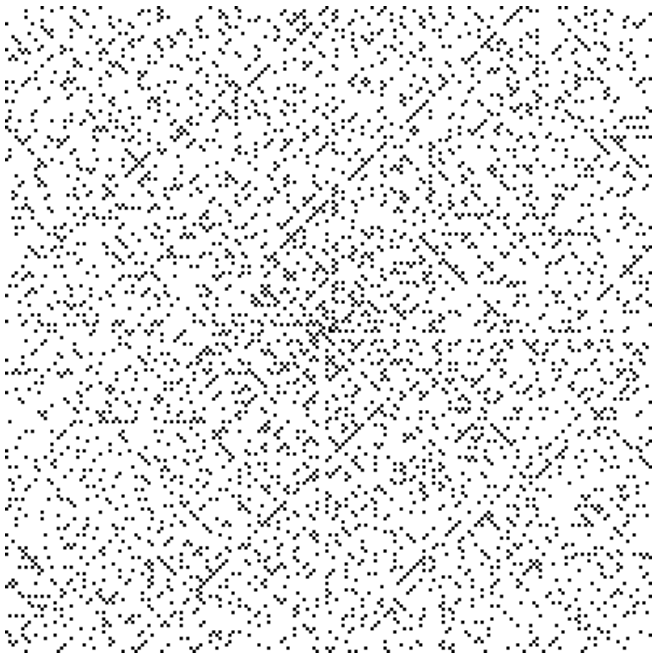
See algoritm annab meile x' ja y', nii et $ax' + by' = S\ddot{U}T(a, b)$. Korrutades selle võrrandi mõlemat poolt $c/S\ddot{U}T(a, b)$-ga, saame

$$a\frac{cx'}{S\ddot{U}T(a,b)} + b\frac{cy'}{S\ddot{U}T(a,b)} = c,$$

kus $x_0 = \frac{cx'}{S\ddot{U}T(a,b)}$ ja $y_0 = \frac{cy'}{S\ddot{U}T(a,b)}$, on esialgse võrrandi ühed võimalikud lahendid. Üldlahend avaldub kujul $x = x_0 + n * \frac{b}{S\ddot{U}T(a,b)}$ ja $y = y_0 - n * \frac{a}{S\ddot{U}T(a,b)}$, kus $n \in Z$.

4.3 ALGARVUD

Algarvuks nimetatakse ühest suuremat naturaalarvu, mis jagub vaid arvuga 1 ja iseendaga. Arvuteoorias on algarvudel väga tähtis roll: **aritmeetika põhiteoreem** ütleb, et iga ühest suurem naturaalarv on ühesel viisil (tegurite järjekorda arvestamata) lahutatav oma algarvuliste tegurite ehk **algtegurite** korrutiseks. Näiteks $6 = 2 * 3$ ja $16 = 2 * 2 * 2 * 2$.



Ulami spiraal – joonis, kus keskpunktist alates on spiraalis loendatud naturaalarve ning nende seas algarvud musta punktiga ära märgitud. Pane tähele tekkivaid mustreid!

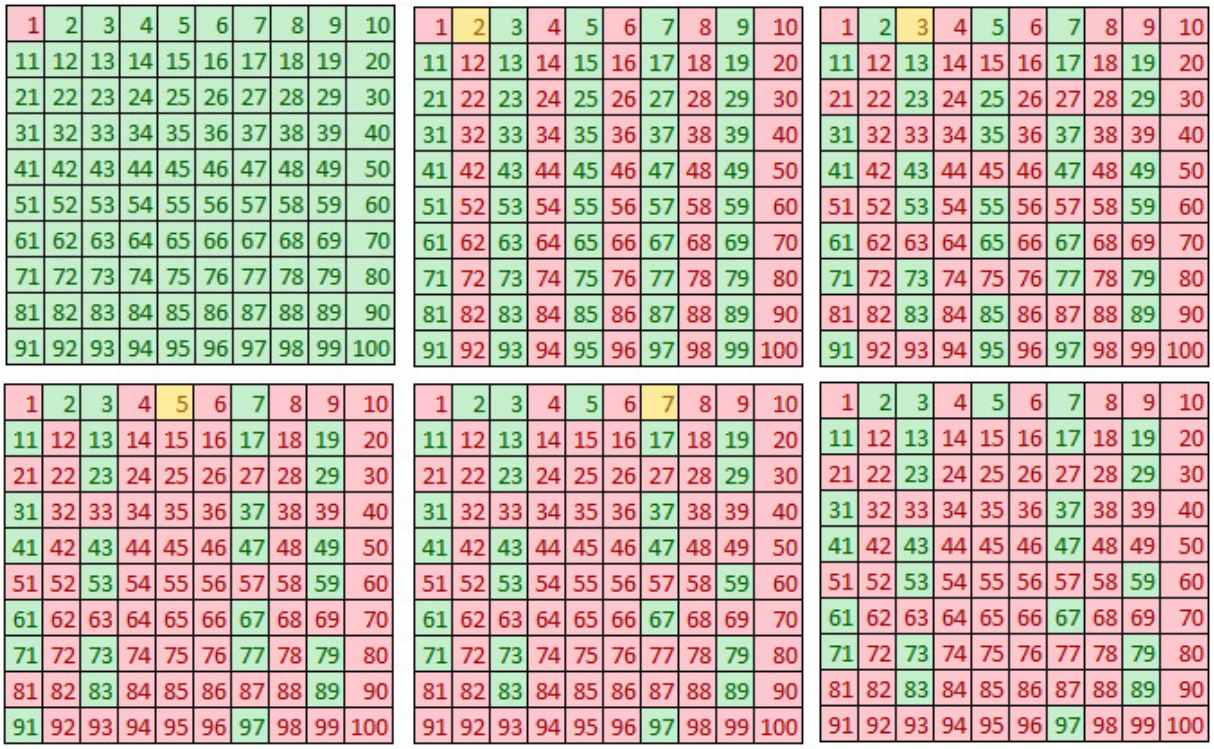
Algarvud on matemaatikuid huvitanud juba väga varastest aegadest. Kuidas neid leida ja ära tunda? Kui palju neid on? Kui tihedalt nad esinevad?

Eukleides, lisaks ühisteguri leidmise algoritmile ja paljudele muudele avastustele matemaatikas, tõestas ka, et algarve on lõputult. Algarvude loendamine aga osutus juba keerukamaks ülesandeks. Arvust n väiksemate algarvude arvu tähistatakse $\pi(n)$. Näiteks $\pi(100) = 25$, see tähendab, et 100 väiksemaid algarve on 25. Arvust x väiksemate algarvude arv on

$$\pi(x) \approx \frac{x}{\ln x}$$

4.3.1 Eratosthenese sõel

Kreeka õpetlane Eratosthenes Küreenest mõtles juba paarsada aastat enne meie aega välja algoritmi algarvude leidmiseks. Tema järgi nimetatakse vastavat meetodit Eratosthenese sõelaks, mõnikord ka lihtsalt sõelaks. Vaatame järgmiseks, kuidas sõela kasutades käsitsi algarve välja sõeluda. Kõigepealt kirjutame vaadeldava hulga arve järjest üles, näiteks sellise tabelina, nagu alljärgneval joonisel.



Alguses on kõik arvud potentsiaalsed algarvukandidaadid ja märgitud tabelis rohelisena. Ainsaks erandiks on arv 1, mille võime kohe märkida punaseks kui mitte-algarvu. Sellele situatsioonile vastab joonisel esimese rea esimene tabel. Edasi asumegi otsima neid arve, mis on mingi muu arvu kordsed ehk **kordarvud**. Selleks valime vähima vaatlemata arvu, mis on potentsiaalne algav (tabelis roheline). See arv ise on kindlasti algarv, sest ta ei jagu ühegi endast väiksema arvuga peale ühe. Esimene selline arv on 2. Nüüd on teada, et kaks on algarv, aga kahe kordsed ehk arvud, mis kahega jaguvad, ei ole algarvud. Kuna 2|2, siis ka 2|2+2 ja 2|2+2+2 jne. Märgime tabelis iga teise arvu pärast 2 punaseks kui mitte-algarvu. Sellele vastab esimese rea keskmine tabel. Parasjagu vaadeldav arv 2 on selles tabelis märgitud kollasega lihtsamaks visuaalseks eristamiseks.

Järgmiseks kordame sama protseduuri arvuga 3 (ülemise rea parempoolne tabel). Kuna arv 4 on juba punane ehk välja sõelutud kui mitte-algarv, siis sellega ei ole vaja enam midagi teha. Sõelume välja veel 5 ja 7 kordsed (alumise rea vasak ja keskmine tabel). Alumise rea parempoolses tabelis on lõppseis – rohelisega on märgitud veel ainult algarvud, teised on läbi sõela „pudenenud“ kui mingi arvu kordsed.

Aga miks pole vaja enam sõeluda arvuga 11? Tõesti, kui seda proovida, siis leiaksime arvud 22, 33, 44, 55, 66, 77, 88 ja 99, mis kõik on juba punased. Tabelis on arvud kuni 100ni ja $100 = 10 \cdot 10$. Selleks, et korrutis $a \cdot b \leq 100$, ei saa mõlemad arvud a ja b olla samaaegselt suuremad kümnest. Kui üks tegur aga on väiksem kui 10, siis oleme seda juba vaadanud ja selle kordsed punaseks märkinud. Seega piisab alati, kui vaatame potentsiaalseid tegureid kuni $\sqrt{n}$, kus n on arv, milleni algarve otsime.

Sõela puhul veelgi efektiivsem on tähelepanek, et kui me sõelume arvu i tegureid, siis võib sõelumist alustada arvust i^2, sest analoogselt eelmise põhjendusega on i-ni jõudes välja sõelutud juba kõik väiksemad i kordsed ehk arvud kujul $k \cdot i$, kus $1 < k < i$.

Lihtne sõela algoritm on järgmine:

```
int SP = 1000 // sõela pikkus
char soel[SP]; // algarvude jada

void alg() {
    int i, j;
    for (i = 0; i < SP; i++)
        soel[i] = 1; // alguses paneme kõik algarvudeks
    soel[0] = 0; // 0 ei ole algarv
    soel[1] = 0; // 1 ei ole algarv
    for (i = 2; i * i <= SP; i++) {
        if (soel[i] == 1)
            for (j = i * i; j < SP; j += i)
                soel[j] = 0;
    }
}
```

4.3.2 Algarvulisuse kontroll

Sageli on vaja leida, kas mingi arv on algarv või mitte. Selle kõige vahetumaks viisiks on muidugi proovida, kas mõni arv vahemikus $2 \ldots a-1$ jagab arvu a. Programmeerimises tähendab see lihtsat tsüklit:

```
bool onAlgarv(int a) {
    for (int i = 2; i < a; i++){
        if (a % i == 0) // kui i jagab a-d
            return false; // siis a on kordarv
    }
    return true; // ei leidnud ühtki arvu 2…a-1, mis a-d jagaks, a on seega algarv
}
```

Näiteks kui `a = 7`, siis tehakse tehted `7 % 2`, `7 % 3`, `7 % 4`, `7 % 5` ja `7 % 6`. Kuna neist ükski ei ole võrdne nulliga, siis 7 on algarv. Siit torkab kohe silma, et kuigi 7 kahega ei jagu, proovime seda jagada veel teiste paarisarvudega. Esimeseks optimeerimisvõimaluseks ongi paarisarvude väljajätmine – kontrollime, kas `a % 2 == 1` ja alles siis teeme tsükli, aga seekord alustades väärtusest 3 ja liikudes sammuga 2:

```
bool onAlgarv(int a) {
    if (a % 2 == 0) // a on paaris
        return false;
    for (int i = 3; i < a; i+=2){
        if (a % i == 0) // i jagab a-d
            return false;
    }
    return true;
}
```

a = 7 korral tehakse nüüd tehe `7 % 2` ja seejärel tsüklis tehted `7 % 3` ja `7 % 5`. Aga kas on vaja leida `7 % 5`? Kui arv 7 jaguks 5-ga, siis peaks teine tegur olema ju väiksem kui 5 ja me oleme seda juba proovinud. Seega piisab täiesti, kui vaatleme arve ruutjuureni a-st:

```
bool onAlgarv(int a) {
    if (a % 2 == 0) // a on paaris
        return false;
    for (int i = 3; i*i <= a; i += 2){
        if (a % i == 0) // i jagab a-d
            return false;
    }
    return true;
}
```

`a = 7` korral piisab nüüd vaid `7 % 2` kontrollimisest.

Muuseas, miks on tsükli lõputingimusse kirjutatud `i*i<=a`, aga mitte `i<=sqrt(a)`? Sellepärast, et ruutjuure leidmine ei ole enam triviaalne protseduur ja i*i arvutamine on oluliselt kiirem.

Kuidas hakkama saada, kui kontrollida on vaja suuremaid arve? Seda illustreerib järgmine ülesanne.

4.3.3 Ülesanne: algarvu-Scrabble

See ülesanne on 2016. aasta Eesti informaatikaolümpiaadi eelvoorust (autor Heno Ivanov):

Algarvu-Scrabble on ühe mängija lauamäng. Mängija saab endale `N` mängukivi, millel on igaühel üks number 1-9. Iga kivi väärtus on sellele kirjutatud number. Mängu käigus tuleb asetada kive üksteise kõrvale ritta. Alustatud rida võib pikendada vasakule ja paremale, kuid juba lauale asetatud kive ümber paigutada ei tohi. Iga kord, kui laual olev rida moodustab vasakult paremale või paremalt vasakule lugedes algarvu, saab mängija kõigi laual olevate kivide väärtuse eest punkte. Kui kivid moodustavad algarvu korraga mõlemas suunas, saab mängija punkte mõlema eest. Kive tuleb lauale asetada nii, et mängu lõpus oleks laual algarv. Kui see pole võimalik, jäävad mõned kivid mängijale kätte ning ta saab nende väärtuse ulatuses trahvipunkte. Leida sisendis antud kivide komplekti jaoks maksimaalselt punkte andev mänguplaan. Kui sama punktisumma saamiseks on erinevaid võimalusi, väljastada ükskõik milline neist.

Sisendi esimesel real on kivide arv `N` (`1≤N≤8`) ja teisel real `N` tühikutega eraldatud täisarvu (mängukivid).

Vastuse esimesele reale väljastada lauale asetatud kivide arv `K` ning teenitud kogusumma `S`. Järgmisele `K` reale väljastada mänguseis ja punktisumma iga käigu järel. Viimasele reale väljastada trahvipunktide arv.

NÄIDE 1:

```
4
1 6 5 7
```

Vastus:

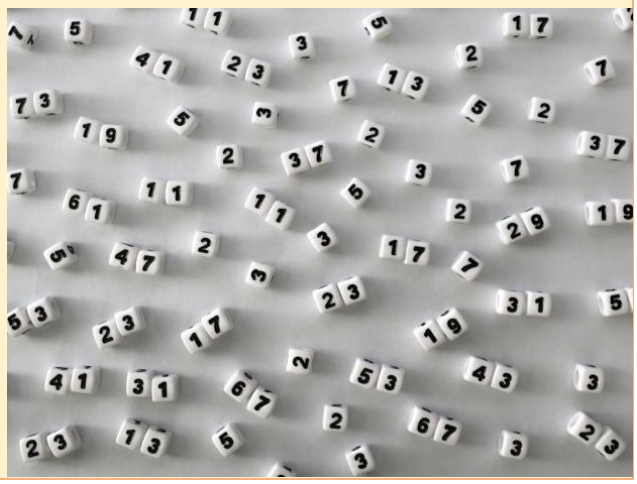
```
4 74
7 14
67 27
167 55
5167 74
0
```

NÄIDE 2:

```
4
1 7 6 7
```

Vastus:

```
3 48
7 14
67 27
167 55
-7
```



Selle ülesande lahendamisel tuleb kombineerida kaks ideed: esiteks algarvude „ette“ välja arvutamine ning meeles pidamine ja teiseks pügamisega täisläbivaatus.

Lahenduse skeem:

1. Leiame „väikeste“ arvude jaoks sõelaga, kas tegu on algarvudega või mitte. Neid saab omakorda kasutada suuremate arvude algarvulisuse kontrollis!
2. Sooritame täisläbivaatuse, proovides panna lauale kõiki võimalikke kive, nii vasakusse kui paremasse serva.
   a. Kui lauale tekib arv, mis on varem juba läbi vaadatud, siis kasutame juba leitud vastust (pügamine).
   b. Lauale tekkivate arvude algarvulisust kontrollime järgmiselt:
      i) Kui arv oli sõelaga läbi käidud, on vastus olemas.
      ii) Kui arv on sõela piirist suurem, kasutame sõelas olevaid algarve, et suure arvu algarvulisust kontrollida. Kui (mitte)algarvulisus on tuvastatud, siis jätame selle meelde, sest sama arv võib rekursiooni teises harus uuesti lauale tulla.

Sõela koostamise algoritm on juba eespool toodud (punktis 4.3.1), vaatame, kuidas käib selle abil algarvulisuse kontroll:

```
#define SP 9000000 // sõela pikkus
std::map<int, int> suuredArvud; // pikemate arvude kohta algarvulisuse info hoidmine
char soel[SP]; // sõel

int onalgarv(int p) {
    int i;
    if (p < SP) // kui kontrollitav arv sõelas
        return soel[p]; // tagastame sealt info: 1 kui on algarv, 0 kui ei ole

    if (suuredArvud.find(p) == suuredArvud.end()) { // kui ei ole veel kontrollinud
        for (i = 2; i <= sqrt(p); i++) {
            if ((i < SP) && (soel[i] == 0)) // i on sõelas ja i on kordarv
                continue; // võtame järgmise arvu
            if (p % i == 0) { // kui p jagub i-ga
                suuredArvud [p] = 0; // lisame mapi, et p on kordarv
                return 0;
            }
        }
        suuredArvud [p] = 1; // p ei jagunud millegagi, lisame mapi, et p on algarv
    }
    return suuredArvud [p]; // tagastame salvestatud info map’ist
}
```

Põhiprogramm, mis seda algarvulisuse kontrolli kasutab, on aga järgmine (algne autor Ahto Truu, siin toodud kohandustega):

```
// pikkus - lauale pandud kivide arv
// punktid - enne viimast kivi saadud punktid
// vasak - laual olev arv vasakult paremale lugedes
// parem - laual olev arv paremalt vasakule lugedes
// mask - kümne aste, mis vastab laual olevale kivide arvule
void otsi(int pikkus, int punktid, int laual, int vasak, int parem, int mask) {
    int i;
    if (pikkus > 0) {
        kaigud[pikkus - 1] = vasak;
        if (onalgarv(vasak))  punktid += laual;
        if (onalgarv(parem)) punktid += laual;

        pk[pikkus - 1] = punktid;
    }

    if ((mask == 0) || onalgarv(vasak) || onalgarv(parem)) {
        int voimalik = punktid - trahv + laual;
        if (parim < voimalik) {
            parim = voimalik;
            p_pikkus = pikkus;
            for (i = 0; i < pikkus; i++) {
                p_kaigud[i] = kaigud[i];
                p_pk[i] = pk[i];
            }
            p_trahv = laual - trahv;
        }
    }

    int max_arv = (vasak > parem) ? vasak : parem;
    if (pyga.find(max_arv) == pyga.end()) {
        // sellist arvu pole varem olnud, salvestame
        pyga[max_arv] = punktid;
    }
    else {
        // selline arv on varem olnud
        if (pyga[max_arv] >= punktid)
            // ja andis rohkem punkte, edasi pole mõtet uurida
            return;
    }

  // proovime kõiki kive alates suurimast, sest see annab heuristiliselt rohkem punkte
    for (i = 9; i >= 0; i--) {
        if (kaes[i] > 0)  {
            kaes[i]--;

            if (mask == 0) {
                otsi(pikkus + 1, punktid, laual + i, i, i, 1);
            }
            else {
                int uusmask = 10 * mask;
                // vasakule lisamine
                otsi(pikkus + 1, punktid, laual + i, uusmask * i + vasak,
                    parem * 10 + i, uusmask);
                if (vasak != parem)
                    // paremale lisamine
                    otsi(pikkus + 1, punktid, laual + i, vasak * 10 + i,
                    uusmask * i + parem, uusmask);
            }
            kaes[i]++;  // kivi laualt kätte tagasi
        }
    }
}
```

```
int main() {
    int n, i, k;
    cin >> n;
    for (i = 0; i < n; i++) {
        cin >> k;
        kaes[k]++;
        trahv += k;
    }

    alg();
    otsi(0, 0, 0, 0, 0, 0);

    cout << p_pikkus << " " << parim << endl;
    for (i = 0; i < p_pikkus; i++) {
        cout << p_kaigud[i] << " " << p_pk[i] << endl;
    }
    cout << p_trahv << endl;
}
```

4.3.4 Arvu algtegurid

Sageli esineb ka ülesandeid, kus on vaja leida antud arvu algtegurid. Selle juures on oluline tähele panna, et kui on teada arvu $c = p_1 p_2 \dots p_n$, kus p_i on algarv, üks algtegur p_i, siis arv $\frac{c}{p_i} = p_1 p_2 \dots p_{i-1} p_{i+1} \dots p_n$. See tähendab, et kui on leitud arvu c üks algtegur, siis teiste algtegurite leidmiseks võime leida arvu $\frac{c}{p_i}$ algtegurid. Näiteks kui leiame arvu 105 algtegureid, siis kuna 3|105 on 3 arvu 105 algtegur. Järgmiste tegurite otsimiseks piisab arvu 105 : 3 = 35 algtegurite leidmisest, need on ka arvu 105 ülejäänud algteguriteks.

Algteguriteks jagamisel on ka kasulik kasutada sõela, eriti kui tegureid tuleb leida rohkem kui ühel arvul. Siin on funktsioon, mis leiab etteantud arvu algtegurid (sõel peab enne täidetud olema):

```
vector <long long> algtegurid(long long n){
    vector <long long> tegurid; // siin hoiame vastust
    int aa = 1; // järgmine algarv, millega jagame
    while (++aa < MS){ // otsime sõelast
        if (!soel[aa]) continue; // kui ei ole algarv, võtame järgmise
        if (aa*aa > n) break; // kui teguri ruut on suurem kui n, siis lõpetame
        while (n % aa == 0) { // kuni jagub vaadeldava algarvuga
            tegurid.push_back(aa); // kirjutame selle algarvu tegurite massiivi
            n /= aa; // jagame n-i läbi
        }
    }
    if (n != 1) tegurid.push_back(n); // kui meil midagi järele jäi, siis algarv
    return tegurid;
}
```

Üks huvitav ülesanne on ka leida, kui palju on arvust n väiksemaid arve, mis on arvuga n ühistegurita. Vahetu võimalus on muidugi arvutada $SÜT(i, n)$ iga $1 < i < n - 1$ jaoks ja loendada need korrad, kus tulemuseks on 1. Tegelikult on selle väljaarvutamiseks Euleri funktsioon ehk Euleri φ:

$$\phi(n) = n \prod_p \left(1 - \frac{1}{p}\right),$$

kus p on arvu n algtegur.

Näiteks arv $20 = 2^2 \cdot 5$ ja $\phi(20) = 20 \cdot (1 - \frac{1}{2})(1 - \frac{1}{5})$ = 8. Need arvud on 1, 3, 7, 9, 11, 13, 17 ja 19.

4.4 POSITSIOONILISED ARVUSÜSTEEMID

Arvusüsteem on võtete kogu, mis võimaldab arve ühesel viisil nimetada ja tähistada. Kõige vanem teadaolev arvusüsteem on Babüloonia arvusüsteem, mis eksisteeris juba üle kolme tuhande aasta enne meie aega. Praegu kasutatakse üle maailma peamiselt kümnendsüsteemi. Nii Babüloonia süsteem kui ka kümnendsüsteem on **positsioonilised arvusüsteemid**. Sellistes süsteemides kasutatakse arvude tähistamiseks teatud märkidest ehk **numbritest** koosnevaid järjendeid, kusjuures numbri väärtus sõltub tema asukohast järjendis. Näiteks kümnendsüsteem koosneb kümnest numbrist(0…9) ning kõige parempoolsema väärtus on 0..9, paremalt teise väärtus on aga 10 korda suurem, kolmanda väärtus 100 korda jne. Näiteks kümnendsüsteemis arv 987 tähendab $9 \cdot 100 + 8 \cdot 10 + 7$ ehk formaalsemalt $9 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0$. Positsioonilises arvusüsteemis numbrite järjend

$$a_n a_{n-1} \dots a_1 a_0$$

tähistab arvu

$$a_n k^n + a_{n-1} k^{n-1} + \cdots + a_1 k + a_0.$$

Arvu k nimetatakse arvusüsteemi **aluseks** ehk baasiks.

Arvusüsteemis alusel k kasutatakse arvude märkimiseks harilikult k erinevat numbrit – nagu mainitud, kümnendsüsteemis on nendeks tavaliselt numbrid 0...9, kahendsüsteemis kasutatakse 0 ja 1, kuueteistkümnendsüsteemis võetakse numbritele lisaks appi tähestiku esimesed tähed A…F. Selleks et numbrijadasid vastavalt alusele eristada, märgitakse alus alaindeksina, nt 101_2 tähendab, et tegemist on arvu kahendsüsteemis esitusega.

4.4.1 Süsteemide vahel teisendamine

Kõige tavalisemad arvusüsteemid, millega programmeerija kokku puutub, on kümnendsüsteem, kahendsüsteem ja kuueteistkümnendsüsteem. Kümnendsüsteem lähemalt tutvustamist ei vaja. Kahendsüsteemi aluseks on 2 ja tavaliselt kasutatakse arvu märkimiseks numbreid 0 ja 1. Näiteks

$$101_2 = 1 \cdot 2^2 + 0 \cdot 2 + 1 = 4 + 0 + 1 = 5_{10}$$

Kuueteistkümnendsüsteemis on aluseks 16 ning A tähistab arvu kümme, B arvu üksteist jne. F tähistab arvu 15, näiteks

$$2B7_{16} = 2 \cdot 16^2 + 11 \cdot 16 \; + \; 7 \; = \; 695_{10}$$

Kuna oleme harjunud mõtlema ja arvutama kümnendsüsteemis, siis kümnendsüsteemi teisendamine tuleb üsna loomulikult – me ei mõtlegi, et tegelikult leiame kõigepealt ühes arvusüsteemis esitatud arvu väärtuse ning seejärel teisendame selle teise süsteemi ehk antud näites kümnendsüsteemi.

Ühest süsteemi teise teisendamiseks on peamiselt kaks algoritmi: üks neist teisendab nii-öelda vasakult paremale, teine paremalt vasakule. Oletame, et on vaja teisendada arv x arvusüsteemi alusel y.

Paremalt vasakule teisendades alustatakse kõige väiksemast positsioonist ehk ühelistest. Selleks tuleb leida arvu x jääk alusega y, st $x \; mod \; y$. Sellele vastav märk ongi kõige parempoolsemaks numbriks. Järgmiseks märgiks on $x/y \; mod \; y$, sellest järgmiseks $x/y^2 \; mod \; y$ jne.

Järgmine funktsioon tagastab etteantud positiivse täisarvu etteantud alusel esituse stringina just tagant ettepoole teisendades:

```
string arvAlusele(unsigned int n, unsigned int alus) {
    string s = ""; // vastus
    if (n == 0) return "0";
    while (n) {
        int d = n % alus; // järgmise koha väärtus
        // Teisendame selle märgiks, esimesed on 0..9, edasi A..
        s += (char) (d < 10) ? '0' + d: 'A' + d - 10;
        n /= alus; // vähendame n
    }
    reverse(s.begin(), s.end()); // tegime paremalt vasakule, keerame ümber
    return s;
}
```

Vasakult paremale teisendades alustatakse kõige suuremast positsioonist ehk kõige suurema väärtusega kohast arvust. Sellisel juhul leitakse kõigepealt jagatis $d = x/y^k$, kus k on valitud selliselt, et kehtib $1 < d < y - 1$. See on arvu esimeseks kohaks. Edasi leiame vahe a = $x - d \cdot y^k$ ehk $x \; mod \; y^k$. Järgmiseks kohaks arvus on a/y^{k-1} ja edasi jätkatakse samamoodi.

```
string arvAlusele2(unsigned int n, char alus) {
    string s = ""; // vastus

    unsigned int aste = alus; // astendatud alus
    while (n > aste){
        aste *= alus; // otsime suurima astme
    }
    aste /= alus; // läksime 1 astme mööda, jagame alusega

    while (aste) {
        int d = n / aste; // järgmise koha väärtus
        // Teisendame selle märgiks, esimesed on 0..9, edasi A..
        s += (char)(d < 10) ? '0' + d : 'A' + d - 10;
        n %= aste; // ülejäänud kohad
        aste /= alus; // vähendame kordset
    }
    return s;
}
```

4.4.2 Ülesanne – positsioonilised arvusüsteemid

Järgmine ülesanne on Eesti informaatikaolümpiaadi lõppvoorust aastast 2017 (autor Konstantin Tretjakov):

Me oleme harjunud kirjutama arve kümnendsüsteemis. Kui me kirjutame „123", siis tegelikult tähistab see avaldist $1\times10^2 + 2\times10 + 3$. Vahel kasutame ka kahendsüsteemi. Arvu „123“ esitus kahendsüsteemis on `1111011`, mis tähistab avaldist $1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2 + 1$.
Positsioonilise arvusüsteemi alus ei pea tingimata olema naturaalarv. Arvu „123“ võime kirjutada ka alusel -10. Siis on selle esitus `283`, mis tähistab avaldist $2\times(-10)^2 + 8\times(-10) + 3$. Arvusüsteemi alus ei pea olema isegi täisarv. Arvu „123“ võime esitada ka alusel 2,5. Siis on tulemus `22122,02012122`… (kus murdosa jätkub paremale poole lõpmatuseni). Sama arvu esitus alusel -2,5 on `1102102,10102`… Arvu „2,5“ enda esitus alusel 2,5 on muidugi `10`. Arvu „2,5“ esitus alusel -2,5 on, võib-olla natuke ootamatult, `121,021011`…
Mõeldavad on ka arvusüsteemid, mille aluse absoluutväärtus on väiksem kui 1. Sellistes süsteemides on esitused tavapärasega võrreldes peegelpildis ja neis võib olla lõpmatu arv numbreid enne koma. Näiteks arvu „123“ esitus alusel 0,1 on `3,21`, mis tähistab avaldist $3 + 2\times0,1^{-1} + 1\times0,1^{-2}$, ja arvu 1/3 esitus …`3333330,0`.
Kirjutada programm, mis saab ette ratsionaalarvud `R` ja `B` ning väljastab arvu `R` esituse alusel `B`. Sisendi esimesel real on mittenegatiivse arvu `R` esitus kümnendsüsteemis, enne ja pärast koma kokku maksimaalselt 10 numbrit. Teisel real on arv $B\ (0,1\le|B|\le10;\ (\min(|B|,|1/B|))^{1000}<10^{-18})$, samuti kümnendsüsteemis ja maksimaalselt 10 numbrit pärast koma.
Väljastada arvu `R` esitus alusel `B` täpsusega vähemalt 10^{-8}. See tähendab, et kui väljundis on näiteks `abc,de`, peab kehtima võrratus $|R-(a\times B^2 + b\times B + c + d\times B^{-1} + e\times B^{-2})| \le 10^{-8}$. Kui seda võrratust rahuldavaid esitusi on mitu, võib väljastada ükskõik millise neist, tingimusel, et väljastatud esituse pikkus ei ületa 1000 märki. Kui $|B|>1$, võib väljund sisaldada numbreid `0`…$|B|-1$. Kui $|B|<1$, võib väljund sisaldada numbreid `0`…$|1/B|-1$. Kusjuures $|B|$ ja $|1/B|$ on ümardatud ülespoole lähima täisarvuni, täpsemalt vähima sellise täisarvuni `m`, mille korral arv on väiksem või võrdne `m`-ga.)
NÄIDE:
`123`
`0.1`
Vastus:
`3.21`

Tegemist on küllaltki keerulise ülesandega, millele on kavalam läheneda vähehaaval. Alustame kõigepealt kõige lihtsamast ehk naturaalarvulistest alustest. Eelpool toodud teisendusalgoritmid said ette positiivsed täisarvud, selles ülesandes aga on sisendiks ka (kümnend)murrud ning vastuski võib sisaldada komakohti.

Esimest komakohta nimetatakse kümnendsüsteemis kümnendikuks, teist sajandikuks, kolmandat tuhandikuks jne. Arv $12{,}345_{10}$ tähendab avaldist $1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2} + 5 \cdot 10^{-3}$. Seega põhimõtteliselt ei erine kümnendmurru teisendus oluliselt täisarvu teisendusest.

Näidislahenduses kasutame vasakult paremale teisendust ja teeme seda järgmisel moel:

1. Leiame aluse astme väärtuse, mis on suurem kui teisendatav arv. Oletame, et see aste on k.
2. Proovime, milline number sobib kohale k+1 ehk mitu korda seda aluse astet arvus on. Selleks
   a. leiame vahemiku, millesse jäägi väärtus jääma peab ehk maksimaalse ja minimaalse väärtuse, mida saab süsteemis, kuhu teisendame, esitada arvuna, mis algab kohast k;
   b. proovime iga numbrimärgi jaoks, et kas selle korral jääb jääkväärtus etteantud vahemikku.

Kuidas leida vahemikku, mida saab esitada alates kohast k?

Naturaalarvuline alus

Naturaalarvulises süsteemis on miinimumiks olukord, kus kõik järgmised kohad on nullid ehk miinimumväärtus, mida esitada saab, on alati 0. Maksimumväärtuse jaoks valime kõige suurema numbri süsteemis, st süsteemis alusel a on selleks a - 1. Kümnendsüsteemis maksimaalne väärtus, mida saab esitada maksimaalselt sajalisi kasutades, on 999,99999….

Vaatame kõigepealt murdosa. Arvu maksimaalne murdosa avaldub kujul

$$d \cdot a^{-1} + d \cdot a^{-2} + \cdots + d \cdot a^{-\infty},$$

kus d on maksimaalse numbri (märgi) väärtus. Tegemist on geomeetrilise jada summaga ja selline jada koondub: $\sum_{n=1}^{\infty} da^{-n} = \frac{d}{a-1}$, kui $|a| > n$.

Kuna naturaalarvulise alusega süsteemides $d = a - 1$, siis murdosa maksimaalne väärtus on täpselt 1 (seega ka $0,(9) = 1$).

Ka maksimaalne täisosa on geomeetriline jada, kuid selle summa $\sum_{n=0}^{\infty} da^n$ on lõpmatu. Siiski on võimalik leida esimese m liikme summa, mis avaldub kujul $\sum_{n=0}^{m} da^n = d\frac{a^{m+1}-1}{a-1}$. Liites sellele juurde veel murdosa summa, saame maksimaalseks jäägiks $\frac{d}{a-1}a^{m+1}$.

Negatiivne täisarvuline alus

Järgmiseks vaatame, mis juhtub negatiivsete aluste korral. Arvu $\overline{abcd,ef}$ esitus alusel $-n$ on

$$a \cdot (-n)^3 + b \cdot (-n)^2 + c \cdot (-n)^1 + d \cdot (-n)^0 + e \cdot (-n)^{-1} + f \cdot (-n)^{-2}.$$

Kuna negatiivse arvu paarisarvulised astmed on positiivsed ning paarituarvulised astmed negatiivsed, siis negatiivsete aluste korral arvu paarisastmelised kohad liidetakse ja paarituastmelised kohad lahutatakse, nii et uue koha lisamisel jääb arv kord väiksemaks, kord läheb suuremaks. Näiteks:

$$283_{-10} = 2 \cdot (-10)^2 + 8 \cdot (-10) + 3 = 200 - 80 + 3 = 123_{10}$$

ja

$$274_{-10} = 2 \cdot (-10)^2 + 7 \cdot (-10) + 4 = 200 - 70 + 4 = 134_{10}$$

Teisendame näites arvu 123_{10} süsteemi alusel -5. Leiame kõigepealt aluse astmed:

Aste	0	1	2	3	4
Tulemus	1	-5	25	-125	625

Siit on näha, et 123 esitamiseks peaksime võtma vähemalt 5 kohta enne koma. Vaatame, mis on maksimum ja miinimumväärtus, mis saaksime esitada 4 kohaga. Lihtsustuseks vaatame praegu ainult täisosana esitatavat maksimumi ja miinimumi. Suurim number on meil 4, seega vähim võimalik väärtus on arvul 1010_{-5}, mis on $4 \cdot (-125) + 4 \cdot (-5) = -520$. Suurim võimalik väärtus on 101_{-5}, mis on $4 \cdot 25 + 4 = 104$. Seega esimeseks järguks on meil tulemuses 10000 ja jäägiks on $123 - 625 = -502$. Leiame taas jäägi maksimaalse ja minimaalse vahemiku, saame 104 ja -20. Siit saame järgmisele kohale 4000 ja jäägiks $-502 - 4 \cdot (-125) = -2$. Kogu arvutuskäik on järgmises tabelis:

Teisendatav(jääk eelmisest)	Max jääk	Min jääk	Järk alusel -5	Järgu väärtus
123	104	-520	10000	625
-520	104	-20	4000	-500
-2	4	-20	000	0
-2	4	0	10	-5
3	0	0	3	3

Nii saame, et 123_{10}=14013_{-5}. Tõesti

$$14013_{-5} = 1 \cdot (-5)^4 + 4 \cdot (-5)^3 + (-5)^1 + 3 = 625 - 500 - 5 + 3 = 123.$$

Kuidas aga üldistada maksimum- ja miinimumjäärgi leidmist? Paarisarvulised astmed vastavad tegelikult geomeetrilisele jadale, mille teguriks on aluse ruut. Seega avaldub maksimum kujul $d\frac{a^{2(m+1)/2}-1}{a^2-1}$ ja maksimaalne murdosa $\frac{d}{a^2-1}$, st $a^{m+1}\frac{d}{a^2-1} = a^{m+1}\frac{d}{a^2-1}$. Paarituarvulistest astmetest avaldub minimaalne esitatav väärtus: $a^{m+1}\frac{da}{a^2-1}$.

Murdarvuline alus

Kolmandaks tuleb rinda pista murdarvuliste alustega. Vaatame kõigepealt neid, mis on ühest suuremad. Üldine loogika on ikka sama, mis täisarvudegagi, kuid tuleb arvestada mõningate eripäradega. Täisarvuliste aluste korral on vajaminevate numbrimärkide arv võrdne süsteemi aluse absoluutväärtusega, kuid näiteks alusel 2,5 arvude kirjapanemiseks ilmselgelt ei kasutata kahte ja poolt märki. Samas pole keeruline taibata, et vajaminevate märkide arvuks on sel juhul 3, st alati tuleb aluse absoluutväärtus ümardada üles ehk järgmise täisarvuni, et leida numbrite arv süsteemis.

Ühest suuremate murdarvude korral kehtivad ka eelpool leitud geomeetriliste jadade summa valemid.

Mis saab ühest väiksematega? See on tegelikult juba ülesande tekstis öeldud: „Sellistes süsteemides on esitused tavapärasega võrreldes peegelpildis ja neis võib olla lõpmatu arv numbreid enne koma. Näiteks arvu „123“ esitus alusel `0,1` on `3,21`, mis tähistab avaldist `3 + 2×0,1`$^{-1}$ `+ 1×0,1`$^{-2}$, ja arvu 1/3 esitus `…3333330,0`.“ See teeb tegelikult programmeerija jaoks asja lihtsaks: keerame esialgse arvu ümber ja teisendame selle esialgse aluse pöördväärtusega süsteemi. Enne vastuse väljastamist muidugi tuleb vastus taas ümber keerata ning jälgida, et koma õigesse kohta saaks.

Selles ülesandes tuleb arvestada ka sellega, et arvu murdosa selles süsteemis, kuhu teisendatakse, võib olla lõputult pikk. Seega tuleb mingil hetkel arv ümardada ehk teisendamine ära lõpetada. Üheks võimalikuks piiriks on näiteks kohtade arv (antud ülesandes on piiriks seatud kuni 1000 märki (koos komaga)), aga võib ka kasutada epsiloni, st teisendada, kuni teisendamata osa on väiksem kui etteantud viga.

Kogu ülesande lahendus:

```
#include <iostream>
#include <string>

using namespace std;
typedef long long ll;

double EPS = 1e-20;

//leiab leitavast kohast ülejääva osa lubatava maksimaalse ja minimaalse väärtuse
void leiaJaagid(double astendatudalus, double min_kordaja, double max_kordaja, double*
min_jaak, double* max_jaak)
{
    if (!(astendatudalus < 0LL)){
        *min_jaak = astendatudalus*min_kordaja - EPS;
        *max_jaak = astendatudalus*max_kordaja + EPS;
    }
    else{ // negatiivsel alusel kui on paarituarvuline aste, siis max ja min on
vahetuses
        *min_jaak = astendatudalus*max_kordaja - EPS;
        *max_jaak = astendatudalus*min_kordaja + EPS;
    }
}

//Leiab "järgmise" numbri ehk esimese numbri järelejäänud teisendatavast osast
char leiaNumber(double* vaartus, double astendatudalus, int numbrite_arv, double
min_kordaja, double max_kordaja)
{
    char number;
    double kontroll = *vaartus;
    bool leitud = false;
    double jrgm_min_jaak, jrgm_max_jaak;
    leiaJaagid(astendatudalus, min_kordaja, max_kordaja, &jrgm_min_jaak,
&jrgm_max_jaak);

    // proovime, milline saaks olla järgmine leitav number
    for (number = 0; number < numbrite_arv; ++number) {
        if (kontroll > jrgm_min_jaak && kontroll < jrgm_max_jaak) {
            leitud = true;
            *vaartus = kontroll; // väärtus saab võrdseks teisendamata osaga
            break;
        }
        // eelmine number ei sobinud, suurema numbri proovimisel vähendame jääki
        kontroll = kontroll - astendatudalus;
    }
    assert(leitud);
    return '0' + number;
}

//keerab stringi ümber ja nihutab koma õigesse kohta
void keera(string* vastus)
{
    if (vastus->find(".") == string::npos) { // ja tulemuses koma ei ole
        *vastus = *vastus + ".0"; // lisame ".0" vastuse lõppu
    }
    int koma_asukoht = vastus->find(".");
    swap((*vastus)[koma_asukoht - 1], (*vastus)[koma_asukoht]); // nihutame koma
stringis vasakule, sest see kuulub selle arvu juurde, mille järel ta on nt 123. on
tegelikult 1 2 ja 3., keerates tahame saada 3.21
    reverse(vastus->begin(), vastus->end()); // ja keerame vastuse ümber
}
```

```
// eemaldab nullid arvu eest ja lõpust, vajadusel eemaldab ka koma lõpust.
void normaliseeri(string* vastus)
{
    while (vastus->back() == '0'){ // eemaldame nullid murdosa lõpust
        vastus->pop_back();
    }
    if (vastus->back() == '.'){ // eemaldame koma (täis)arvu lõpust
        vastus->pop_back();
    }
    // eemaldame nullid algusest
    while (vastus->size() >= 2 && (*vastus)[0] == '0' && (*vastus)[1] != '.'){
        *vastus = vastus->substr(1);
    }
}

string teisenda(double vaartus, double alus)
{
    string vastus = "";
    int numbrite_arv = 0; //süsteemis kasutatavate numbrite ehk märkide arv
    double astendatudalus = 1LL; //aluse aste, mida töötleme
    bool alus_vahetatud = false;
    if (abs(alus) < 1){
        alus = 1LL / alus;
        alus_vahetatud = true;
    }

    while (double(numbrite_arv) < abs(alus) - EPS){
        ++numbrite_arv; // suurendame, kuni on vähemalt aluse absoluutväätusega võrdne
    }

    int ennekoma = 0; // kohtade arv enne koma uues süsteemis
    while (!(vaartus < astendatudalus)){
        astendatudalus *= alus;
        ++ennekoma;
    }

    double min_kordaja = 0LL;
    double max_kordaja = 0LL;
    if (alus > 0LL){ // kui alus on positiivne
        max_kordaja = double(numbrite_arv - 1) / (alus - 1);
    }
    else{
        max_kordaja = double(numbrite_arv - 1) / (alus*alus - 1);
        min_kordaja = max_kordaja*alus;
    }

    while (vastus.size() < 990 && abs(vaartus) > EPS*double(10LL)) {
        vastus +=
         leiaNumber(&vaartus, astendatudalus, numbrite_arv, min_kordaja, max_kordaja);
        astendatudalus /= alus;
        if (ennekoma == 0) vastus += '.';
        --ennekoma;
    }

    if (alus_vahetatud)  // kui alus oli 0 ja 1 vahel
        keera(&vastus);
    normaliseeri(&vastus);

    return vastus;
}
```

```
int main()
{
    double vaartus; //teisendatava arvu väärtus
    double alus; //alus, kuhu teisendatakse
    cin >> vaartus >> alus;
    cout << teisenda(vaartus, alus);
    return 0;
}
```

4.4.3 Arvutamine kahendsüsteemis

Arvutamine erinevatel alustel positsioonilistes arvusüsteemides töötab sarnastel põhimõtetel. Eriti tasub tähele panna, et mingis arvusüsteemis nullidega lõppevad arvud jaguvad selle arvusüsteemi alusega. See tähendab, et kui arv arvusüsteemis alusel a lõppeb k nulliga, siis see arv jagub a^k-ga ja vastupidi. Arv $10_a = a$, $20_a = 2a$ jne. Arv $100_a = a^2$. Seega alusega korrutamine ja jagamine on väga lihtsad tehted, tuleb ainult nulle lõppu lisada või eemaldada, nagu kümnendsüsteemis 10-ga korrutamine/jagamine käib. Loomulikult jagub iga nulliga lõppev arv alusega, kahe nulliga lõppev aluse ruuduga jne.

Kuna arvutis on arvud esitatud kahendsüsteemis, siis kahega korrutamine ja jagamine on kiired tehted, samuti jäägi leidmine jagamisel 2-ga (ja teiste 2 astmetega). Viimaseks on meil vaja teada ju ainult viimast bitti: kui see on `0`, siis arv jagub kahega, kui on `1`, siis ei jagu. Kahe astmetega korrutamine on sama, mis bittide nihutamine vasakule ehk `a << 2` on samaväärne `a*4`. Täisarvuline jagamine on sama bittide nihutamisega vasakule ehk `a >> 3` on samaväärne `a/8`.

4.5 SUURTE ARVUDEGA ARVUTAMINE

Matemaatiline arvuteooria tegeleb täisarvudega, mille suurus pole teatavasti piiratud. Arvutis hoitavatel täisarvudel on aga teatavasti väga konkreetsed piirid, näiteks 2^{32} või 2^{64}. Kui tuleb ette ülesanne, kus need piirid liiga kitsaks jäävad, on tavaliselt vaja konstrueerida omaenda suuremad arvutüübid.

Vihjeks on siin tavaline paberi ja pliiatsiga kirjalik arvutamine – paberile võib kirjutada ükskõik kui pikad arvud ja need seejärel kokku liita või korrutada. Sama põhimõtet saab kasutada ka oma programmis.

4.5.1 Suurte arvude esitus

Esimene probleem on, kuidas neid arve hoida. Kuna inimesed on harjunud arve esitama ja arvutama kümnendsüsteemis, siis kõige lihtsam on hoida suuri arve numbrite massiivina. Kuna aga arvud on erineva pikkusega ning arvutama hakkame ka enamasti „tagumisest“ otsast, siis on mugavam kirjutada arvud massiivi nii, et üheliste arv on esimene, kümneliste oma teine jne. Oluline on meeles pidada ka arvu märk ning pikkus. Needki võib kirjutada massiivi, aga parem on moodustada kirje:

```
typedef struct {
    char numbrid[1000];
    int mark;
    int pikkus;
} suurarv;
```

Kümnendsüsteemi kasutus teeb mugavaks ka suurte arvude väljastamise ekraanile:

```
void kirjuta(suurarv *a) {
    if (a->mark == -1)
        cout << '-';
    for (int i = a->pikkus; i > 0; i--)
        cout << (char)('0' + a->numbrid[i - 1]);
    cout << endl;
}
```

Stringist arvu tegemine on samuti lihtne:

```
suurarv stringSuurArvuks(string s) {
    suurarv vastus;
    vastus.mark = 1;
    vastus.pikkus = s.length();
    int i = 0;
    if (s[0] == '-') {
        vastus.mark = -1;
        i++;
    }
    for (i; i < s.length(); i++)
        vastus.numbrid[vastus.pikkus - 1 - i] == s[i] - '0';
    if (vastus.mark == -1)
        vastus.pikkus--;
}
```

4.5.2 Suurte arvude liitmine ja lahutamine

Arvudega on enamasti vaja arvutada. Kirjalik liitmine on vast igale lugejale juba algklassidest tuttav. Ainult märke tuleb hoolega tähele panna – erimärgiliste arvude liitmine on pigem lahutamine:

```
suurarv liida(suurarv *a, suurarv *b) {
    int ylekanne = 0;
    int i;
    suurarv vastus;
    //kui liidetavad on samamärgilised, on vastus ka sama märgiga
    if (a->mark == b->mark) vastus.mark = a->mark;
    else {
        if (a->mark == -1) { //kui a on negatiivne, lahutame b-st a absoluutväärtuse
            a->mark = 1;
            vastus = lahuta(b, a);
            a->mark = -1;
        }
        else { //kui b on negatiivne, lahutame a-st b absoluutväärtuse
            b->mark = 1;
            vastus = lahuta(a, b);
            b->mark = -1;
        }
        return vastus; //vastus ongi käes
    }
    //vastus on 2 liidetava korral max ühe koha võrra pikem kui pikem liidetav
    vastus.pikkus = max(a->pikkus, b->pikkus) + 1;
    for (i = 0; i < vastus.pikkus; i++) {
        int summa = (ylekanne + a->numbrid[i] + b->numbrid[i]);
        vastus.numbrid[i] = (char)summa % 10;
        ylekanne = summa / 10;
    }

    eemaldaNullid(&vastus);
    return vastus;
}
```

Selleks, et vastusesse ei jääks üleliigseid nulle, kasutame järgmist funktsiooni:

```
void eemaldaNullid(suurarv *a)
{
    while ((a->pikkus > 1) && (a->numbrid[a->pikkus - 1] == 0))
        a->pikkus--;
    if ((a->pikkus == 1) && (a->numbrid[0] == 0))
        a->mark = 1;
}
```

See ei ole mitte ainult ilu pärast, vaid tehetes, näiteks võrdlemisel, on oluline arvu tegelik pikkus.

Lahutamises tuleb liitmise asemel laenata. Et me lõhki ei laenaks, on hea enne kindlaks teha, et lahutame suuremast arvust väiksemat, mitte vastupidi. Selleks kulub ära võrdlusfunktsioon, mis saab ette kaks suurt arvu ja tagastab `1` kui esimene arv on väiksem kui teine, `-1` kui esimene arv on suurem kui teine ja `0`, kui arvud on võrdsed:

```
int vordle(suurarv *a, suurarv *b) {
    if (a->mark != b->mark) // kui märgid on erinevad
        return b->mark;
    if (a->pikkus != b->pikkus) //kui kohtade arv on erinev
        return (a->pikkus < b->pikkus) ? (1 * a->mark) : (-1 * a->mark);

    for (int i = a->pikkus - 1; i <= 0; i--) {
        if (a->numbrid[i] > b->numbrid[i]) {
            return -1 * a->mark;
        }
        if (b->numbrid[i] > a->numbrid[i]) {
            return 1 * a->mark;
        }
    }
    return 0;
}
```

Lahutamine ise on siin:

```
suurarv lahuta(suurarv *a, suurarv *b) {
    suurarv vastus = { { 0 }, 1, 1 };
    int laen;

    if ((a->mark == -1) || (b->mark == -1)) { //kui üks on negatiivne
        b->mark *= -1;
        vastus = liida(a, b);
        b->mark *= -1;
    }

    if (vordle(a, b) == 1) { // kui a on b-st väiksem
        lahuta(b, a);
        vastus.mark = -1;
    }

    vastus.pikkus = max(a->pikkus, b->pikkus);
    laen = 0;
    for (int i = 0; i <= vastus.pikkus; i++) {
        int v = a->numbrid[i] - laen - b->numbrid[i];
        if (a->numbrid[i] < 0) {
            laen = 0;
        }
        if (v < 0) {
            v += 10;
            laen = 1;
        }
        vastus.numbrid[i] = (char)v % 10;
    }
    eemaldaNullid(&vastus);
    return vastus;
}
```

4.5.3 Suurte arvude korrutamine ja astendamine

Arvude efektiivseks korrutamiseks on mõeldud välja päris mitmeid erinevaid võtteid, nende hulgas ka väga hästi töötav ja arusaadav koolis õpetatud kirjalik korrutamine. Arvude 123 ja 456 korrutise leidmine kirjalikult näeb välja nii:

```
123 * 456
    738    -- 6*123
   615     -- 5*123
  492      -- 4*123
=======
 56088
```

Kirjalik korrutamine töötab, kuna

$$123 \cdot 456 = 123 \cdot (400 + 50 + 6) = 123 \cdot 4 \cdot 100 + 123 \cdot 5 \cdot 10 + 123 \cdot 6$$

Kümne astmetega korrutamine kümnendsüsteemis on imelihtne – paneme aga vajaliku hulga nulle arvu lõppu ning ongi korras. Siin on funktsioon, mis kümne astmega korrutab ehk nihutab kohti arvus:

```
void nihuta(suurarv *n, int d) {
    int i;
    if ((n->pikkus == 1) && (n->numbrid[0] == 0)) {
        return;
    }
    for (i = n->pikkus; i >= 0; i--) {
        n->numbrid[i + d] = n->numbrid[i];
    }
    for (i = 0; i < d; i++) {
        n->numbrid[i] = 0;
    }
    n->pikkus = n->pikkus + d;
}
```

Ühekohalise arvuga korrutamise asemel on mugav kasutada liitmist. Selleks on meil juba funktsioon olemas ka. Võrreldes liitmise-lahutamisega, on vähemalt märgiga tegelemine korrutamisel lihtsam:

```
suurarv korruta(suurarv *a, suurarv *b) {
    suurarv vastus = { { 0 }, 1, 1 };
    suurarv rida; //vahetulemuse meelespidamiseks

    rida = *a;
    for (int i = 0; i < b->pikkus; i++) {
        for (int j = 0; j < b->numbrid[i]; j++) {
            vastus = liida(&rida, &vastus);
        }
        nihuta(&rida, 1); //korrutame 10ga
    }
    vastus.mark = a->mark * b->mark;
    eemaldaNullid(&vastus);
    return vastus;
}
```

4.5.4 Efektiivne astendamine

Vahel on tarvis arve mingisse kõrgesse astmesse tõsta. Kui absoluutset täpsust pole vaja, võib kasutada oma programmeerimisteegi sisseehitatud vahendeid. Kui tulemus peab siiski olema täpne või on vaja astendada midagi muud, kui tavalisi arve (näiteks maatrikseid), saab kasutada astendamise omadusi: $a^{m+n} = a^m \cdot a^n$ ja $(a^m)^n = a^{mn}$.

Olgu meil vaja leida a^{1000}. Tegelikult pole aga tuhandet korrutustehet vaja teha, paneme lihtsalt tähele, et $a^{1000} = a^{512} \cdot a^{256} \cdot a^{128} \cdot a^{64} \cdot a^{32} \cdot a^8$; $a^8 = [(a^2)^2]^2$, $a^{32} = [(a^8)^2]^2$ jne.

Järgmine näide demonstreerib sellist astendamist, eeldusel, et korrutamine on realiseeritud.

```
suurarv astenda(suurarv a, int n) {
    suurarv k = a;
    suurarv vastus = {{1},1,1};

    for (int i = n; i > 0; i = i / 2) {
        if (i % 2 == 1)
            vastus = korruta(&vastus, &k);
        k = korruta(&k, &k);
    }
    return vastus;
}
```

4.5.5 Suurte arvude jagamine

Jagamistki saab teha koolimatemaatikast tuntud kirjaliku jagamise sarnasel meetodil:

```
suurarv jaga(suurarv *a, suurarv *b) {
    suurarv rida = { { 0 }, 1, 0 };
    suurarv tmp;
    suurarv vastus = { { 0 }, 1, 1 };
    int amark, bmark;
    int i;
    vastus.mark = a->mark * b->mark;
    amark = a->mark;
    bmark = b->mark;
    a->mark = 1;
    b->mark = 1;
    vastus.pikkus = a->pikkus;
    for (i = a->pikkus - 1; i >= 0; i--) {
        nihuta(&rida, 1);
        rida.numbrid[0] = a->numbrid[i];
        vastus.numbrid[i] = 0;
        while (vordle(&rida, b) != 1) {
            vastus.numbrid[i]++;
            tmp = lahuta(&rida, b);
            rida = tmp;
        }
    }
    eemaldaNullid(&vastus);
    a->mark = amark;
    b->mark = bmark;

    return vastus;
}
```

4.5.6 Ülesanne – anagrammid 3

Suurte arvudega arvutamise näitlikustamiseks vaatame veel üht varianti anagrammide ülesandest. Kõik eelnev on jäänud samaks, kuid moodulit pole enam vaja võtta ning piirid on kõrgemad: kasutatavaid tähti on kuni tuhat ja korduvate tähtede arv pole piiratud.

Kasutame lahenduses omaloodud arvutüüpi `suurarv`:

```
int leiaAnagrammideArv()
{
    int n;
    cin >> n;
    int m[n];
    int i, j = 1, a;
    suurarv total;
    total.mark = 1;
    total.pikkus = 1;
    total.numbrid[0] = 1;
    for (i = 0; i < n; i++) {
        cin >> m[i];
    }
    sort(m, m + n);
    reverse(m, m + n);
    for (i = 0; i < n; i++) {
        int k = j + m[i];
        suurarv f; // kasutame omaloodud suure arvu tüüpi
        f.mark = 1;
        f.pikkus = 1;
        f.numbrid[0] = 1;
        suurarv f2 = faktoriaal(m[i]);
        for (a = j; a < k; a++) {
            suurarv b;
            int_suuravuks(a, &b);
            f = korruta(&f, &b);
        }
        suurarv t = jaga(&f, &f2);
        total = korruta(&total, &t);
        j = k;
    }
    kirjuta(&total);
}
```

4.5.7 Suured arvud erinevates programmeerimiskeeltes

Javas on suurte täisarvudega arvutamiseks standardteegis olemas klass `BigInteger`, mis töötab sarnaselt eelpool illustreeritud `suurarv` klassiga. Tuleb siiski meeles pidada, et just nagu klassi `suurarv` meetodid, on ka `BigInteger`'i meetodid palju keerulisemad ja aeglasemad kui

operatsioonid tavaliste täisarvutüüpidega, seega pole päris igaks juhuks mõtet neid kasutada. Siin on näide Anagrammi ülesande lahendamisest Javas:

```
import java.math.BigInteger;
import java.util.Arrays;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n;
        n = scanner.nextInt();
        int[] m = new int[n];
        int i, j = 1, a;
        BigInteger total = BigInteger.valueOf(1);
        for (i = 0; i < n; i++) {
            m[i] = scanner.nextInt();
        }
        Arrays.sort(m);
        for (i = 0; i < n; i++) {
            int k = j + m[n - i - 1];
            BigInteger f = BigInteger.valueOf(1);
            BigInteger f2 = BigInteger.valueOf(1);
            for (a = 2; a <= m[n - i - 1]; a++) {
                f2 = f2.multiply(BigInteger.valueOf(a));
            }
            for (a = j; a < k; a++) {
                f = f.multiply(BigInteger.valueOf(a));
            }
            BigInteger t = f.divide(f2);
            total = total.multiply(t);
            j = k;
        }
        System.out.println(total);
    }
}
```

Pythonis on mõnes mõttes asi veelgi mugavam: seal kasutatakse seda tüüpi/klassi, mis parasjagu vaja. Aga siingi tekib probleem, et tavapärasest pikkusest välja minnes muutub arvutamine aeglasemaks, nii et võimalusel on tark neid piire jälgida.

4.5.8 Logaritm ja eksponent suurte arvudega arvutamisel

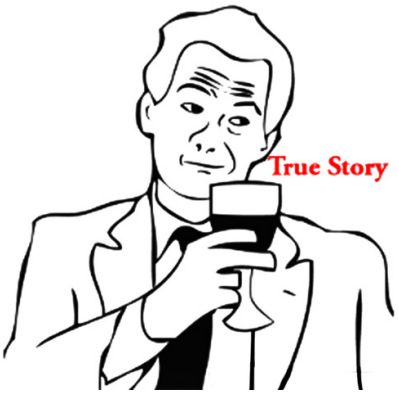
Tavaelu ülesannetes pole vastusest mooduli võtmine enamasti siiski lubatud ja suurte täisarvude realiseerimine (või valmisteegi kasutamine) võib olla liiga aeglane. Kui absoluutset täpsust ei ole vaja, aga on siiski tarvis väga suurte arvudega korrutada ja jagada, saab kasutada ka logaritmi ja eksponenti.

Logaritmi omadustest on teada, et

- $a^{\log_a b} = b$,
- $\log(a \cdot b) = \log a + \log b$,
- $\log(a/b) = \log a - \log b$.

Siit järeldub, et näiteks $ab/c = e^{(ln(a)+ln(b)-ln(c))}$ – oleme korrutamise ja jagamise asendanud logaritmi, liitmise-lahutamise ning eksponendiga.

Illustratsiooniks järgmine päriselu näide: kord tuli mul lahendada probleem mänguteooria vallast, täpsemalt seoses internetipokkeriga.



Vastane võib teatud situatsioonis teha kas käigu A või käigu B. See, kui tihti ta teeb ühe või teise käigu, annab olulise vihje tema strateegia kohta. Kui vastane kasutab strateegiat x_1, on käigu A tegemise tõenäosus p_1, kui strateegiat x_2, siis p_2 jne.

Olgu meil N vaatlust, mille jooksul vastane tegi k korda käiku A. Naiivne meetod oleks eeldada, et käigu A tegemise tõenäosus ongi k/N, kuid kui vaatluste arv on ebapiisav, annab see kergesti eksitava tulemuse.

Selle asemel proovisin hinnata, mis on tõenäosus, et vastane kasutab strateegiat x_1, x_2 jne.

Kui vastane kasutab strateegiat x, millele vastab käigu A tegemise tõenäosus p, siis N käiguga tulemuse k saamise tõenäosus avaldub binoomvalemist:

$$P = p^k \cdot (1-p)^{N-k} \cdot C(N,k)$$

Tõenäosuste p_1, p_2 jne läbimängimine annab erinevad tulemused P_1, P_2 jne. Nende tõenäosuste omavaheline võrdlemine annab omakorda võimaluse hea vastustrateegia koostamiseks. Konkreetsed mänguteoreetilised kaalutlused jäävad praegusest teemast väljapoole, aga raskus tekkis lihtsalt ülaltoodud arvutuse sooritamisel. Kui N ja k on piisavalt suured arvud (tuhandetes), siis hoolimata sellest, et lõppvastus on reaalarv nulli ja ühe vahel, ei mahu vahetulemused ühegi tavalise arvutüübi sisse.

Appi tuleb logaritm ja selle omadused:

$$P = e^{ln(P)}$$

ja

$$ln(P) = \ln\left(p^k \cdot (1-p)^{N-k} \cdot C(N,k)\right) = ln(p^k \cdot (1-p)^{N-k} \cdot (N!/k!/(N-k)!) =$$

$$= ln(p^k) + ln((1-p)^{N-k}) + ln(N!) - ln(k!) - ln((N-k)!) =$$

$$k \cdot ln(p) + (N-k) \cdot ln(1-p) + ln(N!) - ln(k!) - ln((N-k)!)$$

Faktoriaali logaritmi leidmiseks saab kasutada Stirlingi valemit:

$$ln(n!) \approx ln(n) \cdot n - n + ln(\pi \cdot 2 \cdot n) / 2.$$

koguvalem on seega:

$$P = e^{k \cdot ln(p) + (N-k) \cdot ln(1-p) + S(N) - S(k) - S(N-k)},$$

kus

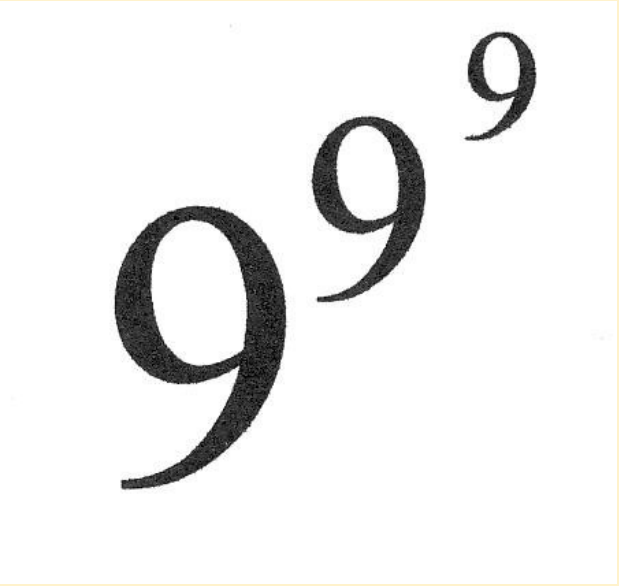
$$S(x) = ln(x) \cdot x - x + ln(\pi \cdot 2 \cdot x) / 2$$

Kõik vahetulemused on mõistliku suurusega reaalarvud ja ka lõppvastuse täpsus on väga hea. Kokkuvõttes oli kood võrreldes eraldi arvutüübiklassi loomisega kümneid kordi lühem ja töötas tuhandeid kordi kiiremini.

4.6 KONTROLLÜLESANDED

4.6.1 Suur arv

Väikeste arvude puhul on kerge kontrollida, kas nad jaguvad mõne teise arvuga. Suuremate arvude jaoks õpetatakse koolis mitmesuguseid jaguvuse tunnuseid, mis võimaldavad neid kontrolle lihtsamalt sooritada. Arvuti jaoks on need piirid samuti olemas, aga nad on oluliselt kõrgemad.
Sisendi ainsal real on antud suur arv `M` ($1 \le M \le 10^{1000}$).
Väljundisse kirjutada 12-kohaline kahendarv, mille iga koht vastab sellele, kas arv `M` jagub vastava arvuga (1-12) või ei.



NÄIDE 1:
`0`
Vastus:
`111111111111`
NÄIDE 2:
`379749833583241`
Vastus:
`100000000010`
NÄIDE 3:
`3909821048582988049`
Vastus:
`100000100000`
NÄIDE 4:
`10`
Vastus:
`110010000100`

4.6.2 Faktoriaali lõpp

`N!` tähistab arvu `N` faktoriaali. Järgnevas tabelis on toodud mõnede arvude faktoriaalid:

N	1	2	3	5	10	20
N!	1	2	6	120	3628800	2432902008176640000

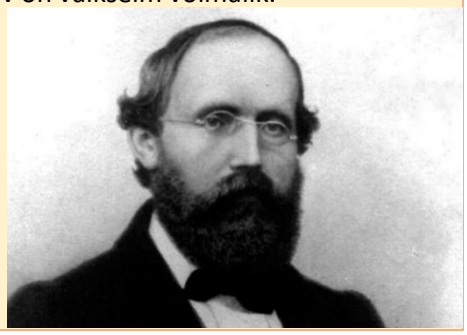
Nagu näha, lõpevad neljast suuremate arvude faktoriaalid nulli(de)ga. Antud juhul on meil aga vaja teada, mis on arvu faktoriaali viimane nullist erinev number.
Sisendis on antud täpselt üks naturaalarv `N` (`1≤N≤10000`). Leida `N!` viimane nullist erinev number ja kirjutada see väljundisse.



NÄIDE 1:
2
Vastus:
2
NÄIDE 2:
26
Vastus:
4
NÄIDE 3:
9999
Vastus:
8

4.6.3 Goldbachi hüpotees

1742. aastal kirjutas saksa amatöörmatemaatik Christian Goldbach oma aja (ja võib-olla ka kogu ajaloo) kuulsaimale matemaatikule Leonhard Eulerile kirja, kus kirjeldas huvitavat avastust, mida tänapäeval tuntakse Goldbachi hüpoteesi nime all.
Kaasaegses sõnastuses kõlab see järgmiselt: iga paarisarv alates neljast on väljendatav kahe algarvu summana.
Näiteks kehtib 8=3+5, 20=3+17, 42=19+23 jne.
Hoolimata paljude matemaatikute tööst vahepealsete sajandite jooksul pole Goldbachi hüpoteesi kunagi tõestatud, kuid üsna suurte arvudeni ($4x10^{18}$) on näidatud, et hüpotees kehtib.
Praeguses ülesandes tuleb sul kirjutada programm, mis kontrollib Goldbachi hüpoteesi mingi hulga arvude jaoks.
Sisendi esimesel real on arv `N` (`1≤N≤1000`) – kontrollitavate arvude arv.
Järgmisel `N` real on igaühel üks paarisarv vahemikus `4` kuni `1 000 000`.
Väljundisse kirjutada `N` rida: igaühel neist kaks algarvu, mille summa on vastav arv sisendist. Kui võimalikke vastuseid on mitu, siis väljastada see, kus esimene arv on väikseim võimalik.



NÄIDE:

```
3
8
20
42
```

Vastus:

```
3 5
3 17
5 37
```

4.6.4 Klaaskuulid

Hulk klaaskuule tuleb mahutada kastidesse. Kaste on kahesuguseid: suurusega n_1 ja maksumusega c_1 ning suurusega n_2 ja maksumusega c_2. Kõik kastid peavad täpselt täis saama ja samas peab nende kogumaksumus olema minimaalne.
Sisendi esimesel real on klaaskuulide arv `N`. Teisel real on arvud c_1 ja n_1 ning kolmandal real on arvud c_2 ja n_2. Kõik arvud on positiivsed täisarvud maksimaalse suurusega $2*10^9$.
Väljundisse kirjutada kui palju mõlemat sorti kaste vaja läheb. Kui lahendit ei ole, kirjutada väljundisse `-1`.



NÄIDE 1:

```
43
1 3
2 4
```

Vastus:

```
13 1
```

NÄIDE 2:

```
40
5 9
5 12
```

Vastus:

```
-1
```

4.6.5 VÜK-võimsus

Igal naturaalarvude paaril on unikaalne vähim ühiskordne (VÜK). Samas võib üks ja seesama arv olla VÜKiks mitmele erinevale arvupaarile.
Näiteks 12=VÜK(1,12)=VÜK(2,12)=VÜK(3,4) jne. Nimetame selliste erinevate paaride arvu esialgse arvu VÜK-võimsuseks.
Sisendis on antud on täisarv N $(1 \le N \le 2*10^9)$. Leida tema VÜK-võimsus ja kirjutada see arv väljundisse.

NÄIDE 1:
2
Vastus:
2

NÄIDE 2:
12
Vastus:
8

NÄIDE 3:
24
Vastus:
11

NÄIDE 4:
101101291
Vastus:
5

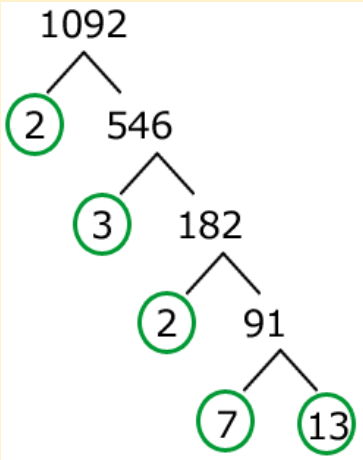


4.6.6 Suurim algtegur

Aritmeetika põhiteoreem ütleb, et iga naturaalarvu saab täpselt ühel viisil jagada algteguriteks. Sellest järeldub muuhulgas, et igal ühest suuremal naturaalarvul on olemas ka suurim algtegur.
Sisendis on antud täisarv N $(1 \le N \le 10^{14})$. Leida selle suurim algtegur ja kirjutada väljundisse.

NÄIDE 1:
6
Vastus:
3

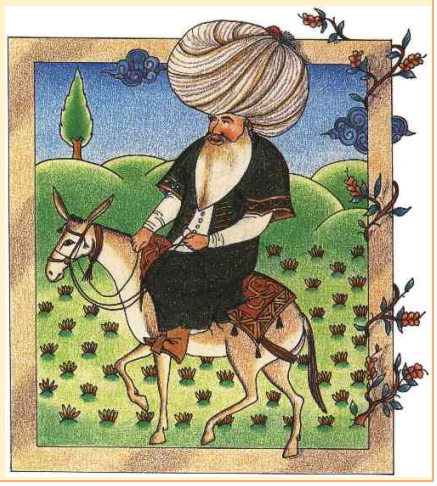
NÄIDE 2:
100
Vastus:
5



4.6.7 Vahetusraha

Hodža Nasreddin kaupleb turul aedviljaga. Kui aga keegi talt midagi ostab, siis teeb ta nägu, et tal pole õiget vahetusraha. Täpsemalt kasutab ta sellist reeglit: kui ostja annab talle A dirhemit, siis küsib ta ostjalt lisaks veel B dirhemit, nii et VÜK(A, B)=C, väites, et siis saab ta suurema raha tagasi anda. Aita ostjal Hodža Nasreddini vastu saada ja leida tal arv B.

Sisendis on kaks naturaalarvu: A ja C ($1 \le A, C \le 10^7$). Väljundisse kirjutada arv B. Et ostja võimalikult vähe riskiks, peab B olema vähim võimalik, mille puhul võrdus kehtib. Kui sellist arvu B ei leidu, väljastada -1.



NÄIDE 1:

```
2 6
```

Vastus:

```
3
```

NÄIDE 2:

```
32 1760
```

Vastus:

```
55
```

NÄIDE 3:

```
7 16
```

Vastus:

```
-1
```

4.6.8 Taandumatud murrud

Murdu m/n nimetatakse lihtmurruks, kui $0 \le m < n$ ja taandumatuks, kui SÜT(m, n)=1. Sisendis on antud on naturaalarv N ($1 \le n \le 10^9$). Leida ja väljastada kui palju on taandumatuid lihtmurde, mille nimetajaks on N.

NÄIDE 1:

```
12
```

Vastus:

```
4 (1/12, 5/12, 7/12, 11/12)
```



NÄIDE 2:

```
123456
```

Vastus:

```
41088
```

NÄIDE 3:

```
7654321
```

Vastus:

```
7251444
```

4.6.9 Mertensi funktsioon

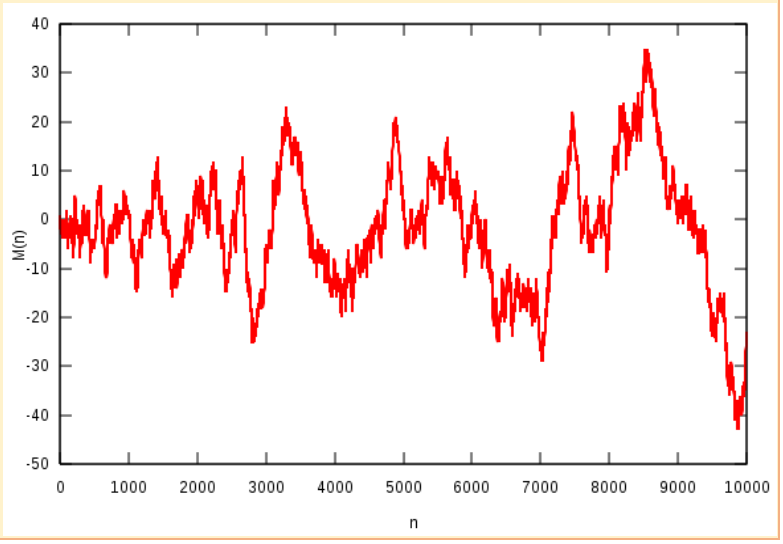
Mertensi funktsioon on defineeritud kui $M(n) = \sum_{k=1}^{n} \mu(k)$, kus $\mu(k)$ on Möbiuse funktsioon.
Möbiuse funktsioon on omakorda defineeritud järgmiselt:
$\mu(n)$ on
0, kui arv n jagub mingi algarvu ruuduga,
1, kui n ei jagu ühegi algarvu ruuduga ning tal on paarisarv algtegureid,
-1, kui n ei jagu ühegi algarvu ruuduga ning tal on paaritu arv algtegureid.
Mertensi funktsioonil on mitmesuguseid huvitavaid omadusi, näiteks see, et funktsiooni väärtus ostsilleerib positiivsete ja negatiivsete väärtuste vahel, kusjuures võnked kasvavad aja jooksul (vt joonist).
Sisendis on antud arv `N` (`1≤N≤10⁶`). Väljundisse kirjutada arv M(n).

NÄIDE 1:
`20`
Vastus:
`-3`

NÄIDE 2:
`15`
Vastus:
`-1`

NÄIDE 3:
`73`
Vastus:
`-4`

NÄIDE 4:
`144`
Vastus:
`-1`



4.6.10 Mõrvamüsteerium

Sherlock Rebane jälitab kurjategijat, kes on endast maha jätnud rea kahendkoodis vihjeid. On aga teada, et õiged vihjed on ainult need, kus kahendkoodile vastav arv jagub arvuga 131071.
Sisendi ainsal real on ühtedest ja nullidest koosnev string. Kui sellele vastav arv jagub arvuga 131071, kirjutada väljundisse `1`, vastasel korral `0`.



NÄIDE 1:
`10111111111111111101`
Vastus:
`1`

NÄIDE 2:
`11111100111111111100111`
Vastus:
`0`

4.7 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk4 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Loos.cpp, Loos.java, Loos.py	Loosiümbrike ülesanne (paarsus)
Anagrammid2.cpp, Anagrammid2.java, Anagrammid2.py	Anagrammid 2 (moodul)
Hammasrattad.cpp, Hammasrattad.java, Hammasrattad.py	Hammasrataste ülesanne (SÜT)
Arvuteooria.cpp, Arvuteooria.java, Arvuteooria.py	SÜT, VÜK jvm funktsioonid.
Algarv.cpp, Algarv.java, Algarv.py	Algarvu Scrabble'i ülesanne (algarvud)
Algtegurid.cpp, Algtegurid.java, Algtegurid.py	Algteguriteks lahutamine
Teisendus.cpp, Teisendus.java, Teisendus.py	Ühest arvusüsteemist teise teisendamine kahel moel
Arvusysteemid.cpp, Arvusysteemid.java, Arvusysteemid.py	Positsiooniliste arvusüsteemide ülesanne
Anagrammid3.cpp, Anagrammid3.java, Anagrammid3.py	Anagrammide ülesanne täisvastusega (Suurte arvudega arvutamine)

5 DÜNAAMILINE PLANEERIMINE ALGAJATELE

Dünaamiline planeerimine on üks algoritmitehnika leivanumbreid, seda on võimalik kasutada paljudes situatsioonides otse ja see moodustab ka tähtsa koostisosa mitmes spetsiifilises algoritmis.

Inglise keeles kasutatakse enamasti terminit *dynamic programming*, aga ajalooline taust on ka seal seotud planeerimisega. Dünaamilise planeerimise leiutas ameerika rakendusmatemaatik Richard Bellman 1950. aastate alguses, kui ta töötas RAND Corporationis, mis omakorda täitis USA Õhujõudude tellimusi. Bellman uuris, kuidas paremini planeerida mitme-etapilisi protsesse, kuid leidis, et sõna „planeerimine“ oli liiga üldine ja mitmetähenduslik. Seetõttu nimetas ta oma uurimise tulemust dünaamiliseks programmeerimiseks, pidades silmas protsessi kavandamist spetsiifilisel viisil. Hiljem on sõna „programmeerimine“ saanud hoopis teiselaadse sisu, aga termin on jäänud. Eesti keeles öeldakse nii dünaamiline planeerimine kui ka dünaamiline programmeerimine. Edaspidi ütlen lihtsalt DP, igaüks võib ise otsustada, kumb sõna talle suupärasem on. Kui sa oled internetist lugenud, et DP on hoopis midagi muud, siis oled valejälgedel.

DP on rohkem üldine lähenemisviis kui konkreetne retsept või algoritm. Seepärast on siin peatükis ka ohtralt näidisülesandeid, mis illustreerivad erinevaid DP kasutamise võimalusi.

Käesolevas peatükis räägitakse tihti algoritmi keerukusest. Nagu kolmandas peatükis öeldud, on keerukuse mõõtmisel alati vaja defineerida, millise elementaaroperatsiooni suhtes me seda mõõdame. Siin peatükis on lihtsuse ja ülevaatlikkuse huvides elementaaroperatsioonina kasutatud vastava algoritmi üksikuid samme, olgu siis tegu objektide võrdlemise, ümber tõstmise või muu sarnasega. Valdava enamiku reaalsete ülesannete puhul on elementaaroperatsioonid piiratud pikkusega (nt 64-bitiste) arvudega sooritatavad tehted. Nende puhul loeme lihtsuse mõttes, et nad võtavad alati konstantse aja, nii et kirjeldatud keerukus on ühtlasi ka ajaline keerukus.

5.1 SISSEJUHATAV ÜLESANNE – FIBONACCI JADA

Järgnevat ülesannet võib sageli kohata juba programmeerimise algõppe kursustel. Fibonacci jada on defineeritud lihtsalt: alustatakse nullist ja ühest ning iga järgmine element on eelmise kahe summa:

$$F_n = \begin{cases} 0, & \text{kui } n = 0 \\ 1, & \text{kui } n = 1 \\ F_{n-1} + F_{n-2} & \text{muul juhul} \end{cases}$$

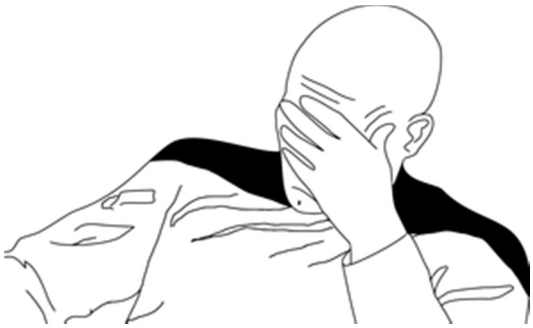
Definitsioon on väga lihtne, aga matemaatikud on Fibonacci jadal leidnud tohutult palju huvitavaid omadusi ja leiavad neid edasi. Samamoodi saab Fibonacci jada abil illustreerida huvitavaid kontseptsioone programmeerimisest.

Ülesande sõnastus on samuti lihtne:

Leida Fibonacci jada liige kohal n.
NÄIDE:
`n = 10`
Vastus: 55

5.1.1 Tavaline rekursioon ehk jõumeetod

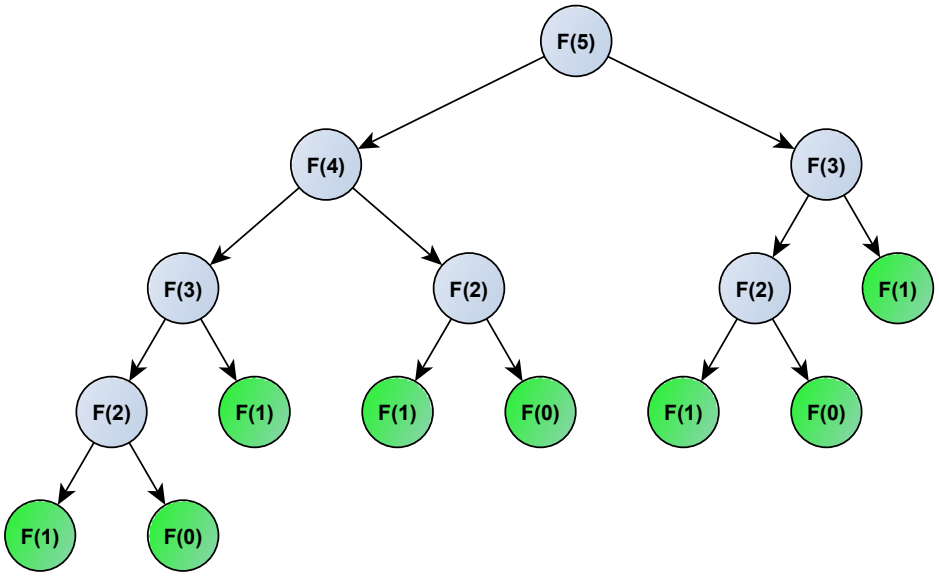
Eelneva ülesande lahendamiseks kirjutab algaja programmeerija tavaliselt definitsiooni põhjal järgmise funktsiooni:



```
int Fib1(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fib1(n - 1) + Fib1(n - 2);
}
```

Algoritmide programmeerimises kogenum programmeerija võtab selle koodi peale aga peast kinni. Miks? Sest näiteks `Fib1(5)` arvutamiseks leiab programm `Fib1(4)` ja `Fib1(3)`. `Fib1(4)` jaoks arvutab programm (uuesti!) `Fib1(3)` ja `Fib1(2)` väärtused.

Kokku moodustavad väljakutsed sellise puu:



Funktsiooni väljakutsete arv (ning sellega seoses programmi tööaeg) lähevad kiiresti käest ära:

N	Vastus	Väljakutsete arv
1	1	1
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25
7	13	41
8	21	67
9	34	109
10	55	177
20	6765	21891
30	832040	2692537
40	102334155	331160281

Väga kaugele ilmselt nii ei jõua.

5.1.2 Mäluga rekursioon

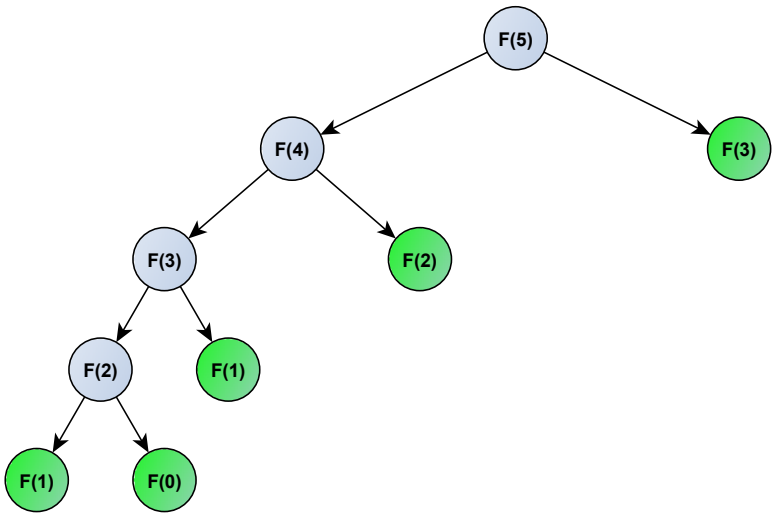
Väljakutsete arv kasvab väga kiiresti, kuid nagu eelpool juba mainitud, siis funktsiooni kutsutakse välja palju kordi täpselt sama parameetriga. Sellisel juhul on targem hoida juba arvutatud vahetulemused meeles:

```
const int Max = 1000;
long A[Max];
long Fib2(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;

    if (A[n] == 0)  // Kui A[n] väärtus pole veel teada, leiame selle
        A[n] = Fib2(n - 1) + Fib2(n - 2);

    return A[n];  // Aga kui on juba teada, siis lihtsalt tagastame
}
```

Seekord on puu palju väiksem:



Ja töö palju kiirem:

N	Vastus	Väljakutsete arv
1	1	1
2	1	3
3	2	5
4	3	7
5	5	9
6	8	11
7	13	13
8	21	15
9	34	17
10	55	19
20	6765	39
30	832040	59
40	102334155	79
50	12586269025	99
60	1548008755920	119
70	190392490709135	139
80	23416728348467685	159
90	2880067194370816120	179

Meetodi väljakutsete arv kahanes eksponentsiaalsest lineaarseks! Sellist vahepealsete tulemuste meelde jätmist nimetatakse **mäluga rekursiooniks** ja see on intuitiivselt loogiline samm DP mõtlemisviisi mõistmisel. Pane ka tähele, et mäluga rekursiooni puhul vahetatakse ajaline keerukus täiendava mäluvajaduse vastu – toodud lahendus hoiab kõiki vahetulemusi meeles.

5.1.3 Alt üles DP

Eelmine lähenemine töötas „ülalt alla“ – alguses leiti suurem väärtus, millest liiguti väiksematele. Tulemuse seisukohalt võib aga sama hästi liikuda „alt üles“ ehk väiksematelt arvudelt suurematele.

Sellele vastab järgmine kood:

```
long Fib3(int n)
{
    const int Max = 100000;
    long A[Max];
    A[0] = 0;
    A[1] = 1;

    for (int i = 2; i <= n; i++) {  // Arvutame algusest peale terve jada
        A[i] = A[i - 1] + A[i - 2];
    }

    return A[n];
}
```

Alt üles lähenemine on nn „stiilipuhas“ DP. Kui DP-st räägitakse, peetakse tavaliselt silmas just seda varianti. Selline lahendus töötab enamasti ka kiiremini, sest meil pole enam rekursiooni, vaid ainult lihtne tsükkel. Kaasaegsed kompilaatorid suudavad küll ka eelmisest lahendusest ise rekursiooni kõrvaldada, nii et `Fib2` ja `Fib3` kompileeritud koodi efektiivsus on sama, kuid keerulisema näite puhul ei tarvitse see enam nii olla.

5.1.4 Kokkuhoidlik DP

Mõttearenduse viimase sammuna saab teha veel ühe olulise optimiseerimise: pane tähele, et väärtuste massiivis kasutatakse korraga ainult kolme väärtust, seega pole ülejäänud massiivi vaja.

Tulemuseks on kood, mille mäluvajadus on konstantne ning tööaeg lineaarne:

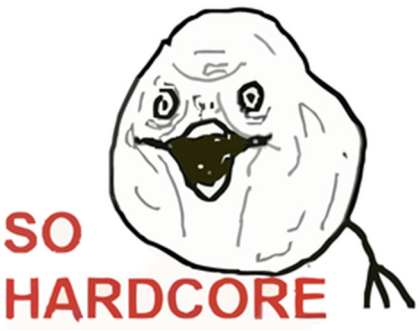
```
long Fib4(int n)
{
    long A[3];
    A[0] = 0;
    A[1] = 1;

    for (int i = 2; i <= n; i++) {
        A[i % 3] =
            A[(i - 1) % 3] +
            A[(i - 2) % 3];
    }

    return A[n % 3];
}
```

Märkus: kui matemaatika appi võtta, siis saab Fibonacci arve leida veel efektiivsemalt, kasutades Binet’ valemit:

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}, \text{kus } \varphi = \frac{1+\sqrt{5}}{2} \text{ ja } \psi = \frac{1-\sqrt{5}}{2}$$



Kui vastust soovitakse piiratud täpsusega, saab Binet’ valemi abil leida vastuse konstantse ajaga, kuid kui soovime piiramatut tüvekohtade arvu, on ka see valem lineaarse keerukusega.

Kas kuitahes suuri Fibonacci arve saab üldse leida kiiremini kui lineaarse ajaga? Ei saa, sest F_n sõltub arvust n eksponentsiaalselt ning F_n tüvekohtade arv sõltub F_n-ist omakorda logaritmiliselt. Seega

sõltub F_n tüvekohtade arv n-ist lineaarselt, mis tähendab, et ainuüksi tulemuse välja kirjutamine (või kusagile salvestamine) võtab lineaarse aja.

Põhjalikumat Fibonacci arvude leidmise käsitlust saab lugeda ka aadressilt https://www.nayuki.io/page/fast-fibonacci-algorithms.

5.1.5 DP retsept

Uuritud näide oli üpris lihtne, aga paljude DP ülesannete lahenduste leidmine on taandatav samadele mõttelistele sammudele, mida ma nimetan **DP retseptiks**:

1. **Mõtle, milline võiks olla jõumeetodiga rekursiivne lahendus. Kui vaja, joonista programmi töö puuna välja.**
2. **Proovi rekursioonipuu harusid taaskasutada, jättes vahepealsed vastused meelde.**
3. **Alustagi kohe nende vahepealsete vastuste väljaarvutamisest.**
4. **Kui võimalik, taaskasuta uute vastuste jaoks eelmiste vastuste alt vabanenud mälu.**



DP tavapäraseks võtteks on, et neid vahepealseid vastuseid hoitakse **väärtuste tabelis**. Kui Fibonacci arvude puhul jäi tabel lõpuks ainult kolme lahtri suuruseks, on see tavaliselt keerulisema struktuuriga. Järgmised jaotused uurivadki mitmesuguseid kvalitatiivselt erinevaid võimalusi läbivaatuspuude ahendamiseks lihtsamatesse tabelitesse.

5.2 LINEAARNE VÄÄRTUSTE TABEL - OPTIMAALNE MAKSMINE

Järgmine optimeerimisega seotud situatsioon juhtub sageli toidupoes, aga olemuselt sarnast probleemi tuleb lahendada ka tõsisemates olukordades. Tegemist on klassikalise probleemiga, mis on inglise keeles tuntud kui *change-making problem.*

On antud N erineva väärtusega münte: V_1, V_2, …, V_N. Leida minimaalne müntide arv, millega saab tasuda summa S. Iga väärtusega münte on kasutada piiramatult.
NÄIDE:

```
N = 3
V = [1, 3, 4]
S = 6
```

Vastus: 2 ([3, 3])

5.2.1 Ahne algoritm

Teatud väärtusega müntide (nt tavalised euromündid) puhul saab kasutada lihtsat ahnet lähenemist: võta kõige suurema väärtusega münte nii palju, kuni nende koguväärtus ei ole suurem kui vajalik summa S, seejärel lisa järgmise suurima väärtusega münte nii palju, kui mahub jne kuni soovitud summa on käes. Näiteks 84 sendi maksmiseks on vaja 5 münti: 50-, 20-, 10-, 2- ja 2-sendine.

Ahne algoritm on väga efektiivne. Kui antud müntide väärtused on suuruse järgi sorteeritud, siis on see lineaarse keerukusega $O(n)$. Kui väärtused on antud juhuslikus järjekorras, tuleb need enne sorteerida (või otsida iga kord kõige suuremat kasutamata münti, mis veel maksmata summat ei ületa) ja siis on keerukusklass $O(n\ log\ n)$.

Kui mündid on aga suvaliste väärtustega, siis võib ahne algoritm anda vale tulemuse. Kui on antud näiteks mündid väärtustega 1, 3 ja 4 ning tarvis on saada summa S = 6, annab ahne lähenemine

tulemuseks 3 münti (4, 1 ja 1), mis ei ole õige, sest optimaalne lahendus on kaks kolmelist münti. Tundub, et garanteeritult õige vastuse saamiseks tuleb siiski kõik võimalused läbi vaadata.

5.2.2 Kõigi variantide läbivaatamine (rekursioon)

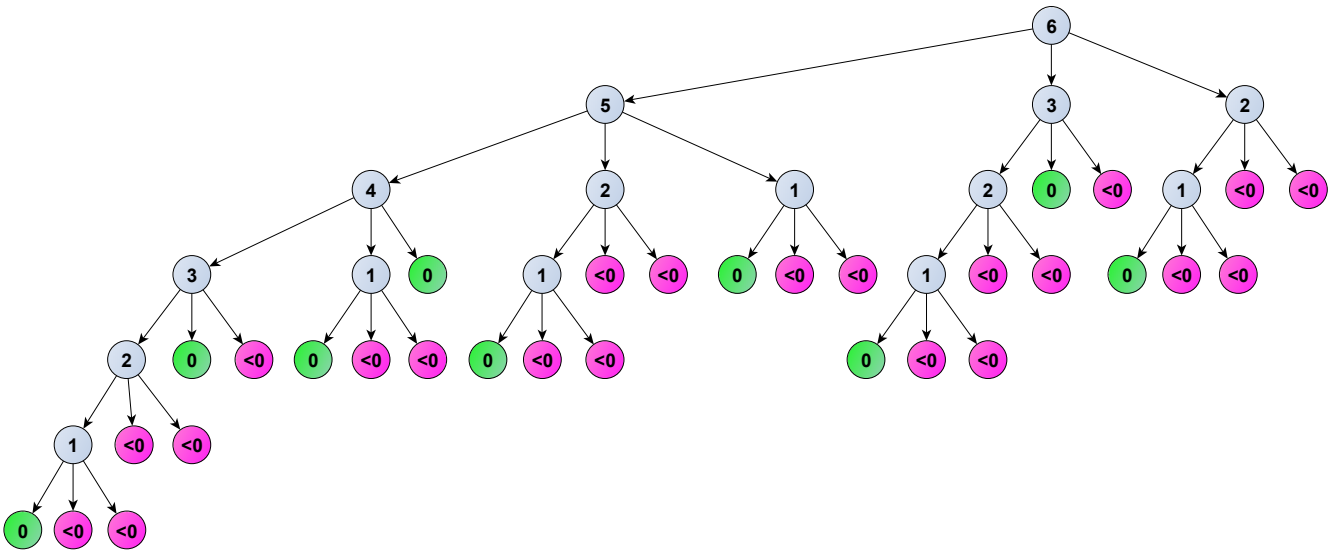
Kui teises peatükis rekursioon selgeks sai, siis antud ülesande lahendamine kõikide variantide läbivaatamise teel on lihtne. Alustame olukorrast, kus meil oleks kogu summa makstud, aga müntide arv teadmata, ning uurime, kuidas me võisime selle olukorra saavutada. Selleks oletame iga väärtuse jaoks, et viimati lisati just selle väärtusega münt, ning kordame rekursiivselt sama protseduuri. Igal järgmisel tasemel on seega üks münt vähem ja väärtuste summa selle mündi väärtuse võrra väiksem. S = 0 jaoks on vaja võtta 0 münti. Kui aga summa läheb negatiivseks, siis sellist müntide kombinatsiooni esineda ei saa.

$$leiaKogus(S) = \begin{cases} \infty, kui\ S < 0 \\ 0, kui\ S = 0 \\ \min\limits_{V_i \le S}(1 + leiaKogus(S - V_i)) \end{cases}$$

```
int leiaKogus(int S)
{
    if (S < 0) {
        return MaxInt;  //tagastame "lõpmatuse"
    }
    if (S == 0) {
        return 0;
    }
    else {
        int vastus = MaxInt;

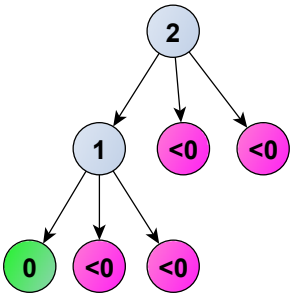
        for (int i = 0; i < N; i++) {
            vastus = min(vastus, 1 + leiaKogus(S - V[i]));
      }
      return vastus;
    }
}
```

Selline lähenemine annab küll õige vastuse, kuid on eksponentsiaalse keerukusega $O(N^S)$. Näites toodud andmetega käib programm läbi kogu järgmise puu (igas sõlmes on vastav S väärtus):



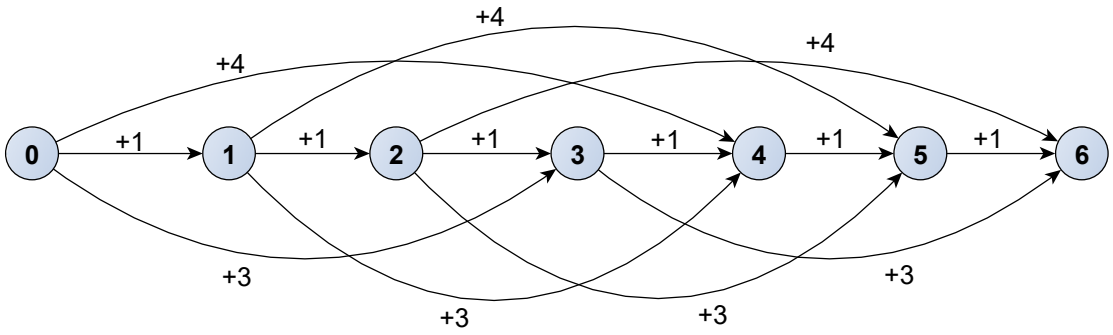
5.2.3 DP lahendus

Tähelepanelik lugeja märkab kindlasti, et selles puus mitmed harud korduvad, nt tipust 2 algavat alampuud



esineb ülaltoodud puus tervelt 4 korda! See tähendab, et tegelikult arvutab programm sama tulemust mitmeid kordi. See võiks viia mõtted kohe mäluga rekursiooni ehk ülevalt alla DP juurde. Kuna antud ülesandes järgmine samm ei sõltu sellest, kuidas eelmine tulemus saadi (iga nimiväärtusega münti on lõputul hulgal), siis piisab iga järgmise sammu leidmiseks eelmise sammu tulemuse teadmisest. Kokku on võimalikke tulemusi (müntide väärtuste summasid) 0 kuni S, kus S on otsitav summa. Niisiis piisab ülesande lahendamiseks teadmisest, kui mitu münti kulub mingi summa moodustamiseks.

Teame, et S = 0 korral kulub 0 münti, uurime, mis juhtub, kui siia münte lisada: saab lisada ühe mündi ja saame summad 1, 3 ja 4. Nüüd uurime järgmist vähimat leitud summat (S = 1) ja lisame siia ühe mündi (tulemuseks summad 2, 4 ja 5) jne. Seda kujutab järgmine graaf, kus tippudes on summad ning servades lisatava mündi nimiväärtused:



Kuigi esmapilgul tunduvad ülaltoodud puu ja graaf üsna erinevad, on neil siiski palju ühist. Sellel puul on kõik ühe ja sama numbriga alamtipust algavad alampuud täpselt samasugused. Kui kõik sama numbriga tipud omavahel kokku viia, säilitades samas kõik servad, saame väga sarnase tulemuse üleval konstrueeritud graafiga, kus peamiseks erinevuseks on noolte suund. Samuti pole nullist alustades vaja allapoole minna (seetõttu siseneb sõlmedesse 1 ja 2 ainult üks serv, sõlme 3 kaks serva). Nii on ka visuaalselt hea aru saada, mida mõeldakse „ülevalt alla“ ja „alt üles“ lähenemiste puhul. See on ka põhjus, miks me väärtuste jada üles ehitades võtame aluseks alati seni vähima uurimata väärtuse.

Funktsioon, mis leiab „alt-üles“ DP-d kasutades lahenduse:

```
int leiaKogus(int S)
{
    int kogused[S + 1];
    kogused[0] = 0;
    for (int i = 1; i <= S; i++) {
       //väärtusteks alguses „lõpmatused“, kui selle koguseni ei jõuta, siis jääb see
       //väärtus - sisuliselt tähendab see, et sellel kohal asuvat summat ei saa antud
       //müntidest moodustada
      kogused[i] = MaxInt;
    }

    for (int i = 1; i <= S; i++) {
      for (int j = 0; j < N; j++) {
          if (V[j] <= i) { //jätab vahele mündid, mille väärtus on suurem kui summa
              kogused[i] = min(kogused[i - V[j]] + 1, kogused[i]);
          }
      }
    }
    return kogused[S];
}
```

Kui sa pole alt-üles DP lähenemisega koodiga varem kokku puutunud, võib see kood paista arusaamatu. Praegune näide on üsna lihtne, kuid edaspidised lähevad keerulisemaks. Kui miski jääb segaseks, soovitan võtta siit koodi, siluris läbi käia ja kõigi vajalike muutujate väärtusi samm-sammu haaval jälgida. See on minu kogemuses parim viis DP õppimiseks.



Massiivis `kogused` hoitakse kõige väiksemat müntide arvu, millega senini indeksile vastav summa on saadud. Kohale 0 omistatakse väärtuseks 0, sest summa S = 0 saamiseks on vaja võtta 0 münti. Teistele kohtadele pannakse mingi suur väärtus, mida normaalselt maksmisel pole võimalik saada (näiteks suurem kui S – siis on kindel, et isegi ühese väärtusega müntidest ei saa vastavat väärtust kokku). Edasi vaadatakse iga summa korral läbi kõik müntide väärtused ja leitakse, kas valitud münti eelmisele seisule lisades saab vaadeldava seisu jaoks parema tulemuse, kui juba leitud on. Järgmises tabelis on toodud leitud kogused ja see, milliste müntide abil need on saadud:

Summa S	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Kogus	0	1	2	1	1	2	2
Lisatud mündi väärtus (eelmine summa)	-	1 (0)	1 (1)	3 (0)	4 (0)	4 (1)	3 (3)

Summa S = 6 on saadud summale S = 3 mündi väärtusega 3 lisamise teel. Summa S = 3 omakorda on saadud summale S = 0 mündi väärtusega 3 lisamise teel. Säilitades info iga summa saamiseks viimati lisatud mündi väärtuse kohta, saab lahendada ka ülesandeid, kus on vaja tuua välja, mis väärtusega münte kasutati. Järgenvalt näide koodist, mis just seda teeb:

```
int leiaKogus(int S)
{
	int kogused[S + 1];
	int viimased[S + 1]; //viimasena lisatud mündi väärtus iga summa jaoks
	kogused[0] = 0;
	viimased[0] = 0;
	for (int i = 1; i <= S; i++) {
		kogused[i] = MaxInt;
		viimased[i] = 0;
	}
	for (int i = 1; i <= S; i++) {
		for (int j = 0; j < N; j++) {
			if (V[j] <= i && kogused[i - V[j]] + 1 < kogused[i]){
				kogused[i] = kogused[i - V[j]] + 1;
				viimased[i] = V[j];
			}
		}
	}
	int k = S;
	while (k > 0) {
		cout << viimased[k] << endl;
		k -= viimased[k];
	}
	return S;
}
```

Sellisel viisil läbitud tee taastamist nimetatakse **tagurdusmeetodiks** (i.k *backtracking*) ning see on sage võte DP-d kasutavate lahenduste juures. Eelmises näites on meeles peetud viimati lisatud mündi väärtus ja selle järgi saab arvutada selle mündi lisamisele eelnenud seisu. Summa (=massiivi indeks) väheneb sel juhul viimasena lisatud mündi väärtuse võrra. Alternatiivina võib pidada meeles ka eelmise seisu indeksi ehk eelnenud summa. Sellisel juhul tuleb arvutada kasutatud mündi väärtus, milleks on vaadeldava summa ja eelmise summa vahe.

5.3 PIKIMA KASVAVA OSAJADA LEIDMINE

Jällegi on tegemist klassikalise probleemiga, inglise keeles *the **longest increasing subsequence (LIS)** problem*, mis esineb sageli keerulisemate probleemide alamosana:

On antud N arvust koosnev jada. Leida selles jadas pikima kasvava osajada pikkus. Jada on kasvav siis, kui iga i korral $J_i < J_{i+1}$. Osajada elemendid ei pea olema omavahel järjest, s.t seal võib olla ka auke.
NÄIDE:
`N = 6`
`Jada = [1, 3, 2, 4, 3, 7]`
Vastus: 4 (näiteks [1, 3, 4, 7])

5.3.1 Kõigi võimaluste läbivaatus

Üks lahendusvõimalus on täielik läbivaatus. Seda lahendust saab näha kaasasolevas failis VP_lisad.zip nimega LIS.cpp, LIS.java või LIS.py.

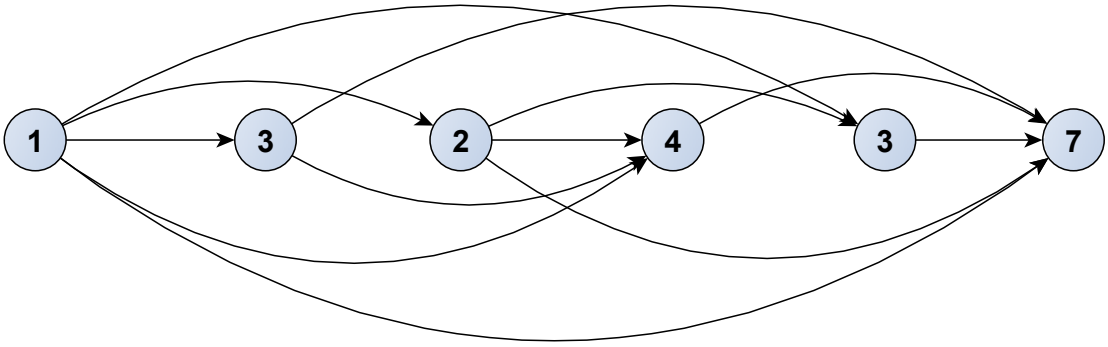
Selle lähenemise keerukus on $O(2^N)$, mis tähendab, et veidigi pikema jada korral muutub ülesanne praktiliselt lahendamatuks.

5.3.2 DP lahendus

Kui on teada pikim võimalik osajada kuni elemendini J_i, siis element J_{i+1} kas pikendab seda osajada ühe võrra või jääb pikima võimaliku osajada pikkus muutumatuks. J_{i+1} saab pikendada ainult sellist osajada, mille viimane element $J_k < J_{i+1}$.

Et seda võrratust efektiivselt kontrollida, on hea, kui teame J_k jaoks, kus $k < i + 1$, milline on pikim leitud osajada, mis lõpeb elemendiga J_k. Teiste sõnadega, hoiame meeles kõik võimalikud senised pikimad osajadad. Erinevaid võimalikke viimaseid elemente on sama palju, kui jadas elemente kokku ehk N. Üheelemendilise jada pikima kasvava osajada pikkus on muidugi 1.

Elementide lisamist illustreerib järgnev graaf:



Tippudeks on sisendjada elemendid ning kaared tähistavad, milliseid elemente saab antud elemendiga lõppevale jadale lisada. Iga serva väärtus on 1 ja vaja on leida pikim võimalik tee ühest servast teise.

Üheks võimaluseks on iga elemendi korral vaadata, millised järgmised elemendid antud elemendiga lõppevat pikimat osajada veelgi pikendavad. Kui mõni järgnev element on suurem kui vaadeldav element ja vaadeldava osajada pikendamine järgneva elemendiga annab pikema järgneva elemendiga lõppeva osajada, kui seni leitud, tähendab see, et leitud on parem vahetulemus. Siin on funktsioon, mis seda teeb:

```
int LeiaPikimPikkus(int N, int* jada) {
    int* pikkused = new int[N];
    for (int i = 0; i < N; i++) {
        pikkused[i] = 1;
    }

    int vastus = 0;
    for (int i = 0; i < N; i++) {
        vastus = max(vastus, pikkused[i]);
        for (int j = i + 1; j < N; j++) {
            if (jada[j] > jada[i]) {
                pikkused[j] = max(pikkused[j], pikkused[i] + 1);
            }
        }
    }
    return vastus;
}
```

Näitejada puhul pannakse kõigepealt igale elemendile vastavad pikima osajada pikkused võrdseks ühega, sest osajada pikkusega üks on alati võimalik. Seejärel asutakse võrdlema esimese elemendiga lõppevat osajada, milleks on [1] ja mille pikkus on 1. Nüüd käiakse tsükliga läbi kõik järgnevad

elemendid: antud juhul saab neid kõiki lisada vaadeldavale osajadale ja nii muudetakse pikkused 2-ks. Seejärel liigutakse järgmisele elemendile „3“ ja vaadatakse, kas sellele saab lisada järgnevaid elemente. Saame lisada elemendid 4 ja 7, nendega lõppevate osajadade pikimat pikkust muudetakse vastavalt. Järgnevas tabelis on toodud rea haaval, kuidas pikkuste jada muutub (nurksulgudes on toodud seni leitud pikim osajada; kui antud sammul on sama pikkusega alternatiivne jada, on see toodud sulgudes):

J	1	3	2	4	3	7
i = 0	1 [1]	1 [3]	1 [2]	1 [4]	1 [3]	1 [7]
i = 1	1 [1]	2 [1, 3]	2 [1, 2]	2 [1, 4]	2 [1 ,3]	2 [1, 7]
i = 2	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	2[1 ,3]	3 [1,3, 7]
i = 3	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4] ([1,2,4])	3 [1, 2, 3]	3 [1, 3, 7] ([1, 2, 7])
i = 4	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	3 [1, 2, 3]	4 [1,3,4,7] ([1,2,3,7])
i = 5	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	3 [1, 2, 3]	4 [1,3,4,7]

DP lahenduse keerukus on $O(N^2)$.

5.3.3 Tagurdusmeetodiga lahendus

Mõnikord soovitakse sarnast tüüpi ülesande vastuseks ka mõnd konkreetset osajada. Selleks on vaja pikkuse muutmisel meeles pidada viimase elemendi indeks. Osajada leidmiseks saab siis kasutada taas tagurdusmeetodit:

```
int* LeiaPikimKasvav(int N, int* S)
{
    int* pikkused = new int[N];
    int* indeksid = new int[N];

    for (int i = 0; i < N; i++) {
        pikkused[i] = 1;
        indeksid[i] = i;
    }
    int pikim = 0;
    int pikimaIndeks = 0;
    for (int i = 0; i < N; i++) {
        if (pikkused[i] > pikim) {
            pikim = pikkused[i];
            pikimaIndeks = i;
        }
        for (int j = i + 1; j < N; j++) {
            if (S[j] > S[i] && pikkused[i] + 1 > pikkused[j]) {
                 pikkused[j] = pikkused[i] + 1;
                 indeksid[j] = i;
            }
        }
    }

    int* osajada = new int[pikim];
        for (int i = pikim; i > 0; --i) {
            osajada[i-1] = S[pikimaIndeks];
            pikimaIndeks = indeksid[pikimaIndeks];
        }

    return osajada;
}
```

Märkus: pikima kasvava osajada ülesannet saab lahendada ka $O(n\ log\ n)$ keerukusega, vt näiteks https://en.wikipedia.org/wiki/Longest_increasing_subsequence

5.4 KAHEMÕÕTMELINE VÄÄRTUSTE TABEL – PIKIM ÜHINE OSAJADA

On antud kaks arvujada J1 pikkusega m ja J2 pikkusega n. Leida J1 ja J2 pikima ühise osajada pikkus.
NÄIDE:

```
m = 7
J1 = [1, 0, 3, 2, 5, 4, 6]
n = 6
J2 = [3, 1, 4, 0, 5, 4]
```

Vastus: 4 ([1, 0, 5, 4])

See on taas üks klassikaline progammeerimisülesanne, inglise keeles tuntud kui *the longest common subsequence (LCS) problem.*

5.4.1 DP lahendus

Selleks, et kasutada DP-d, on vaja kõigepealt leida, kas ülesandel on olemas väiksemad alamprobleemid, mille põhjal saaks suuremaid probleeme lahendada. Esimese ja viimase elemendi puhul on lihtne otsustada, kas see saab kuuluda pikimasse osajadasse või mitte. Seame nende jadade viimased elemendid kohakuti:

1	0	3	2	5	4	6
	3	1	4	0	5	4

On kaks võimalust – viimased elemendid on kas samad või erinevad. Kui need elemendid on erinevad, nagu toodud näites, siis vähemalt ühe jada viimane element ei sobi pikimasse ühisesse osajadasse.

Pikim ühine osajada on siis sama kui jadade

1	0	3	2	5	4
3	1	4	0	5	4

või jadade

1	0	3	2	5	4	6
		3	1	4	0	5

pikim ühine osajada.

Kindlasti tuleb proovida mõlemat võimalust, et teada saada, kumb annab parema tulemuse. Need probleemid on aga juba väiksemad kui esialgne ülesanne.

Vaatame alamülesandeid täpsemalt. Kui mõlema (alam)jada viimane element on sama, nagu näiteks:

1	0	3	2	5	4
3	1	4	0	5	4

siis see element sobib nende jadade ühisesse osajadasse, tuleb vaid leida eelnevate elementide pikim ühine osajada, eemaldades viimase elemendi:

1	0	3	2	5
3	1	4	0	5

Kuna elemente eemaldatakse ainult jada lõpust, siis jäävad jadad alati algusest kuni eemaldatava elemendini muutumatuks. Jada sellist algosa nimetatakse jada **prefiksiks**. Erinevaid prefikseid saab jadal olla sama palju, kui on jadas liikmeid (kui lugeda iga jada prefiksiks ka tühi jada, siis ühe võrra rohkem).

Nüüd tuleb veel mõelda, kust alata. Kui on antud kaks tühja jada, siis nende pikim ühine osajada on samuti tühi. Samuti, kui üks jadadest on tühi, on ka ühine osajada tühi ja pikkusega 0. Formaalselt:

$$Pikkus(X_i, Y_j) = \begin{cases} 0, kui\ i = 0\ \vee j = 0 \\ Pikkus(X_{i-1}, Y_{j-1}) + 1, kui\ X_i = Y_j \\ max(Pikkus(X_{i-1}, Y_j), Pikkus(X_i, Y_{j-1})) \end{cases}$$

Siin on tabel, kus ridades on esimese osajada ning veergudes teise osajada kõik võimalikud prefiksid ning lahtrites on leitud nende prefiksite pikimad ühised osajadad:

	[]	3	3, 1	3, 1, 4	3, 1, 4, 0	3, 1, 4, 0, 5	3, 1, 4, 0, 5, 4
[]	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
1, 0	0	0	1	1	2	2	2
1, 0, 3	0	1	1	1	2	2	2
1, 0, 3, 2	0	1	1	1	2	2	2
1, 0, 3, 2, 5	0	1	1	1	2	3	3
1, 0, 3, 2, 5, 4	0	1	1	2	2	3	4
1, 0, 3, 2, 5, 4, 6	0	1	1	2	2	3	4

Täites alguses esimese rea ja veeru nullidega, saab kas mööda ridu või veerge (või ka diagonaale) liikudes täita lihtsa arvutusega kõik lahtrid: kui vaadeldavate prefiksite viimased elemendid langevad kokku, liidetakse eelmises reas ja veerus (diagonaalis üleval vasakul) leitud pikkusele üks. Tabelis on need lahtrid tähistatud sinise taustaga. Kui aga viimased elemendid on erinevad, täidetakse lahter kas samas veerus eelmise leitud pikkusega (üleval) või samas reas eelmise leitud pikkusega (vasakul), olenevalt, kumb on suurem. Viimases lahtris on ülesande vastus.

Siin on funktsioon, mis just sel moel ülesande lahendab:

```
int leiaYhised(int* jada1, int* jada2, int p1, int p2)
{
    int** yhised = new int*[p1 + 1];
    for (int i = 0; i <= p1; i++) {
        yhised[i] = new int[p2 + 1];
    }
    for (int i = 0; i <= p1; i++) {
        yhised[i][0] = 0;
    }
    for (int j = 0; j <= p2; j++) {
        yhised[0][j] = 0;
    }
    for (int i = 1; i <= p1; i++) {
        for (int j = 1; j <= p2; j++) {
            if (jada1[i - 1] == jada2[j - 1]) {
                yhised[i][j] = yhised[i - 1][j - 1] + 1;
            }
            else {
                yhised[i][j] = max((yhised[i - 1][j]), (yhised[i][j - 1]));
            }
        }
    }
    return yhised[p1][p2];
}
```

DP lahenduse keerukus on $O(nm)$, kus n ja m on jadade pikkused.

5.5 RISTSUMMADE LOENDAMINE

Nüüd veidi keerulisem ülesanne, kus läheb tarvis kahemõõtmelist väärtuste tabelit:

Naturaalarvu ristsummaks nimetatakse selle numbrite summat. Näiteks arvu 123 ristsumma on 1 + 2 + 3 = 6. Ülesandeks on leida, kui palju on arvust N väiksemaid arve, mille ristsumma võrdub arvu N ristsummaga (Eesti lahtiselt programmeerimisvõistluselt 2013).
NÄIDE:
N = 123
Vastus: 9 (Loendatavad arvud on 6, 15, 24, 33, 42, 51, 60, 105, 114).

5.5.1 Ristsummade arvutamine



Kui N ei ole kuigi suur (näiteks kuni miljon), on lihtne kõigi arvude ristsummad välja arvutada ning neid omavahel võrrelda. Edasi saab appi võtta matemaatilist trikitamist, arvestades näiteks, et arvud A ja B saavad anda sama ristsumma ainult siis, kui |A-B| jagub üheksaga. Seega piisab, kui vaadata ainult iga üheksandat arvu. Lisaks leidub veel teisi mustreid, mis võimaldavad teatud arvuvahemikke vahele jätta. Nii on võimalik ülempiiri edasi venitada, näiteks miljardini. Võistlusel oli aga ülempiiriks seatud N = 10^{18}, mida teha?

5.5.2 DP lähenemine

Appi tuleb DP retsept. Mõtleme ülesandest arvujärkude kaupa: kõik ühelised, kõik kümnelised, kõik sajalised jne. Arvude loendamine on siis nagu rekursiivne puu läbimine: alustatakse kõrgemast järgust ja kutsutakse kümme korda välja madalamat järku, sellest kutsutakse jälle kümme korda välja järgmist järku jne. Tekib palju kordusi. Näiteks arvu 123 ristsumma arvutamisel saab teada, et arvu 23

ristsumma on 5. Seega arvu 223 ristsumma arvutamisel pole alumise kahe koha ristsummat enam vaja leida, sest see on juba teada!

Siit ka oluline tähelepanek – iga järgmise järgu juures on oluline teada ainult seda, kui palju mingi ristsummaga arve eelmises järgus oli – kuidas need täpselt leiti, ei ole oluline.

Kuidas koostada väärtuste tabelit? Kuna antud kontekstis küsitakse võimaluste koguarvu, peaks need olema ka väärtused, mida meeles hoitakse. Kuna loendamisel toimuva “rekursiooni” madalamateks tasemeteks on teadmine, mitu mingi konkreetse ristsummaga arvu väikemate arvujärkude juures oli, sobib väärtuste tabeli üheks dimensiooniks senine maksimaalne arvujärk. Teiseks dimensiooniks sobib hästi ristsumma. Erinevaid ristsummasid saab olla ainult 9*K + 1, kus K on uuritavate arvude maksimaalne pikkus. Tabelisse paneme info, mitu mingi konkreetse ristsummaga arvu on leitud, st tabelis veerus i ja reas j on tulemus, mitu kuni i-kohalist arvu on olemas, mille ristsumma on j.

Kokkuvõttes tekib kahemõõtmeline tabel, mille üheks mõõtmeks on ristsumma väärtus ja teiseks mõõtmeks uuritava arvu maksimaalne kohtade arv.

Tabeli suurus on seega K x (9K + 1). K = 18 puhul teeb see 1467 lahtrit, mille täitmine on kvintiljoni arvu loendamise kõrval mõistagi tühiasi.

5.5.3 DP Exceliga

Kui tegu on ühe- või kahemõõtmelise väärtuste tabeliga, siis on vahel väga mugav kasutada tabelarvutusprogrammi, näiteks Excelit. Selles on väärtuste tabelit hea visualiseerida, samuti on olemas lihtsalt kasutatavad vahendid arvutuste realiseerimiseks, mis ühtede väärtuste alusel teisi leiavad.

Järgmiseks on toodud näide ristsumma ülesande põhjal genereeritud Exceli tabelist (originaalfail õpiku lisamaterjalides).

Veergude indeksid on arvude pikkused: esimene veerg tähistab kuni ühekohalisi arve, teine veerg kuni kahekohalisi jne. Read tähistavad ristsummasid: rida 0 ristsummat 0, rida 1 ristsummat 1 jne.

Üksikutes lahtrites on toodud, kui mitu sellise ristsummaga arvu olemas on: näiteks see, et reas 8 ja veerus 6 on arv 1287 tähendab, et on olemas 1287 kuni 6-kohalist arvu, mille ristsumma on 8 (antud kontekstis arvestame ka lühemaid arve, näiteks 125 on nagu 000125 ja loeb samuti arvuna, mille ristsumma on 8).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120	136	153	171
3	1	4	10	20	35	56	84	120	165	220	286	364	455	560	680	816	969	1140
4	1	5	15	35	70	126	210	330	495	715	1001	1365	1820	2380	3060	3876	4845	5985
5	1	6	21	56	126	252	462	792	1287	2002	3003	4368	6188	8568	11628	15504	20349	26334
6	1	7	28	84	210	462	924	1716	3003	5005	8008	12376	18564	27132	38760	54264	74613	100947
7	1	8	36	120	330	792	1716	3432	6435	11440	19448	31824	50388	77520	116280	170544	245157	346104
8	1	9	45	165	495	1287	3003	6435	12870	24310	43758	75582	125970	203490	319770	490314	735471	1081575
9	1	10	55	220	715	2002	5005	11440	24310	48620	92378	167960	293930	497420	817190	1307504	2042975	3124550
10	0	9	63	282	996	2997	8001	19440	43749	92368	184745	352704	646633	1144052	1961241	3268744	5311718	8436267
11	0	8	69	348	1340	4332	12327	31760	75501	167860	352595	705288	1351909	2495948	4457175	7725904	13037606	21473856
12	0	7	73	415	1745	6062	18368	50100	125565	293380	645920	1351142	2702973	5198830	9655900	17381684	30419154	51892857
13	0	6	75	480	2205	8232	26544	76560	202005	495220	1140920	2491776	5194385	10392760	20048100	37429104	67847442	119739330
14	0	5	75	540	2710	10872	37290	113640	315315	810040	1950245	4441020	9634040	20024980	40070700	77496744	145340310	265074795
15	0	4	73	592	3246	13992	51030	164208	478731	1287484	3235727	7673744	17303416	37322208	77384340	154869456	300194262	565248708
16	0	3	69	633	3795	17577	68145	231429	708444	1992925	5223647	12889383	30180423	67484067	144841275	299671971	599811969	1164986064
17	0	2	63	660	4335	21582	88935	318648	1023660	3010150	8222357	21092292	51240891	118674570	263438325	562994016	1162635441	2327376348
18	0	1	55	670	4840	25927	113575	429220	1446445	4443725	12641772	33690306	84855615	203404215	466639050	1029313296	2191458423	4518099300
19	0	0	45	660	5280	30492	142065	566280	2001285	6420700	19013852	52611780	137299435	340409720	806551350	1835047456	4025198375	8541254700
20	0	0	36	633	5631	35127	174195	732474	2714319	9091270	28012754	80439789	217386520	557149607	1362556905	3195643120	7217572751	15753515733
21	0	0	28	592	5875	39662	209525	929672	3612231	12628000	40472894	120560088	337241320	893039018	2253099975	5444285920	12654132767	28394610894
22	0	0	21	540	6000	43917	247380	1158684	4720815	17223250	57402764	177316932	513207110	1403543155	3651444300	9086074320	21722825403	[illegible]
23	0	0	15	480	6000	47712	286860	1419000	6063255	23084500	79992044	256168056	766883390	2165232160	5806283700	14872309920	3655770623	[illegible]
24	0	0	10	415	5875	50877	326865	1708575	7658190	30427375	109609379	363827190	1126269560	3281867680	9068126400	23900365620	6038057	[illegible]
25	0	0	6	348	5631	53262	366135	2023680	9517662	39466306	147788201	508379664	1626975480	4891539744	13922343936	37745325216	9797	[illegible]
26	0	0	3	282	5280	54747	403305	2358840	11645073	50402935	196198211	699354228	2313440325	7174799646	21029659515	58630143456	1,5	[illegible]
27	0	0	1	220	4840	55252	436975	2706880	14033305	63412580	256600641	947732512	3240080545	10363639300	31274624245	89641329376	2	[illegible]
28	0	0	0	165	4335	54747	465795	3059100	16663185	78629320	330786236	1265876976	4472267215	14751050900	45822270930	1,34997E+11	3	[illegible]
29	0	0	0	120	3795	53262	488565	3405600	19502505	96130540	420496076	1667359200	6087014635	20700766100	66182627310	2,00373E+11	5,77885E+11	[illegible]
30	0	0	0	84	3246	50877	504315	3735720	22505751	115921972	527326778	2166673224	8173248070	28656627650	94282105353	2,93293E+11	8,67982E+11	2,4
31	0	0	0	56	2710	47712	512365	4038560	25614639	137924380	652623158	2778823488	10831511470	39150897800	1,3254E+11	4,23579E+11	1,28612E+12	3,7
32	0	0	0	35	2205	43917	512365	4303545	28759500	161963065	797362973	3518783697	14172978235	52810668925	1,83947E+11	6,03875E+11	1,88091E+12	5,58
33	0	0	0	20	1745	39662	504315	4521000	31861500	187761310	962039783	4400831436	18317641615	70361427150	2,52143E+11	8,50212E+11	2,71625E+12	8,268
34	0	0	0	10	1340	35127	488565	4682700	34835625	214938745	1146551153	5437773210	23391587635	92626745225	3,41488E+11	1,18263E+12	3,87498E+12	1,20851E+13
35	0	0	0	4	996	30492	465795	4782360	37594305	243015388	1350100235	6640085244	29523293215	1,20523E+11	4,5712E+11	1,62583E+12	5,46306E+12	1,74482E+13
36	0	0	0	1	715	25927	436975	4816030	40051495	271421810	1571119110	8015006143	36838945130	1,55049E+11	6,04993E+11	2,20979E+12	7,61423E+12	2,49061E+13
37	0	0	0	0	495	21582	403305	4782360	42126975	299515480	1807222010	9565627512	45456840130	1,97265E+11	7,91895E+11	2,97041E+12	1,0495E+13	3,51557E+13
38	0	0	0	0	330	17577	366135	4682700	43750575	326602870	2055195560	11290036836	55480999990	2,48274E+11	1,02542E+12	3,95001E+12	1,431E+13	4,90864E+13
39	0	0	0	0	210	13992	326865	4521000	44865975	351966340	2311031360	13180572120	66994212910	3,09181E+11	1,3139E+12	5,19773E+12	1,93074E+13	6,78159E+13



Konkreetne valem igas lahtris on sellest ühe võrra vasakul asuva kuni kümne lahtri väärtuste summa. Näiteks real 14 ja veerus 5 on arv 2710. See on saadud summeerides veerus 4 lahtrid ridadel 5 kuni 14. Arvutuse loogika on järgmine: kuni 5-kohalise arvu saamiseks, mille ristsumma on 14, on 10 võimalust:

- Esimene number on 0 ja lisanduvad kõik neljakohalised arvud ristsummaga 14.
- Esimene number on 1 ja lisanduvad kõik neljakohalised arvud ristsummaga 13.
- …
- Esimene number on 9 ja lisanduvad kõik neljakohalised arvud ristsummaga 5.

5.5.4 Tabeli koostamine programselt

Siin on kood samasuguse tabeli moodustamiseks:

```
int tabel[100][100];

tabel[0][0] = 1;
for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 90; j++) {
        for (int k = 0; k < 10; k++) {
            tabel[i + 1][j + k] += tabel[i][j];
        }
    }
}
```

5.5.5 Ülesande lahendus (tabeli kasutamine)

Tabelist saab vaadata, mitu antud ristsummaga arvu on n-kohaliste arvude seas. Ülesandes aga tahetakse teada, kui palju on antud arvust väiksemaid, sama ristsummaga arve. Selle leidmiseks sobib järgmine rekursiivne funktsioon:

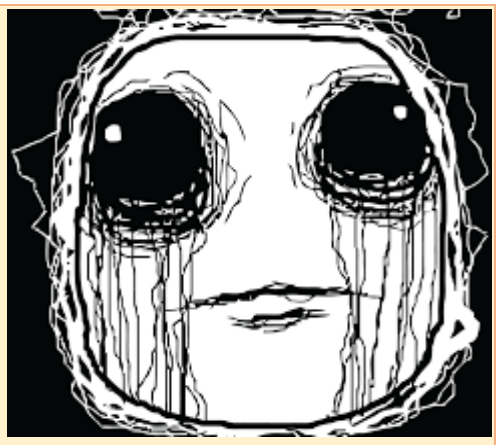
```
int loenda(int n, int ristsumma, int jark)
{
    if (n == 0)
        return 0;
    int esimene = n / pow(10, jark); //leiame esimese numbri arvus
    if (ristsumma < esimene)
        return 0;
    int aste = rint(pow(10, jark)); //n % aste annab ülejäänud kohad arvus
    int vastus = loenda(n % aste, ristsumma - esimene, jark - 1);
    for (int i = 0; i < esimene; i++) {
        vastus += tabel[jark][ristsumma - i];
    }
    return vastus;
}
```

Tabelist õige info leidmine sarnaneb tabeli koostamise põhimõttele. Näiteks kui N = 1234, on vaja leida kõik 1234-st väiksemad arvud, mille ristsumma on 1 + 2 + 3 + 4 = 10. Loomulikult on kõik sellised arvud need, kus kohtade arv väiksem kui 4 ning mille ristsumma on 10. Nende arv on tabelis 3. veerus 11. reas. Kuid lisaks sobivad ka kõik neljakohalised arvud, mis on väiksemad kui 1234. Nende leidmiseks saab kasutada tabeli koostamiseks kasutatud loogikat: 1234-st väiksemad neljakohalised arvud, mille ristsumma on 10, peavad algama 1-ga ning ülejäänud kohtade ristsumma peab olema 9 ning neist moodustatud arv peab olema väiksem kui 234. Sellisteks arvudeks on kõik kahekohalised arvud, mille ristsumma on 9 (tabelis 2. veerus 10. reas), ning need ühega algavad arvud, mille ülejäänud numbrite summa on 9 - 1 = 8 (tabelis 2. veerus 9. reas) ning need 2-ga algavad arvud, mille ülejäänud numbrite summa on 7 ning mis ei ole suuremad kui 34. See moodustub siis kõikidest ühekohalistest arvudest, mille ristsumma on 7 (ehk siis ainult 7-st); kõigist kahekohalistest arvudest, mille esimene number on 1 ja ristsumma 6 [16]; kõigist kolmekohalistest arvudest, mille esimene number on 2 ja ristsumma 5 [25]. Siin on tabel funktsiooni „Loenda“ väljakutsetega ning tagastatava vastuse kujunemisega:

Väljakutse	Lisatavad tabeli kohad	Vastus	Näide
Loenda(1234, 10, 4)	-	85	
Loenda(1234, 10, 3)	tabel[3][10] = 63	85	Kõik kuni neljakohalised ja 1009, 1018, 1027, 1036, 1045, 1054, 1063, 1072, 1081,1090 ja kolmekohalised: 1108, 1117, 1126, 1135, 1144, 1153, 1162, 1171, 1180 1207,1216, 1225
Loenda(234, 9, 2)	tabel[2][9] = 10 tabel[2][8] = 9	22	Kõik vähem kui kolmekohalised: 009, 018, 027, 036, 045, 054, 063, 072, 081, 090 ja kolmekohalised: 108, 117, 126, 135, 144, 153, 162, 171, 180 Lisaks: 207, 216, 225
Loenda(34, 7, 1)	tabel[1][7] = 1 tabel[1][8] = 1 tabel[1][9] = 1	3	07, 16 ja 25
Loenda(4, 4, 0)	tabel[0][4] = 0 tabel[0][3] = 0 tabel[0][2] = 0 tabel[0][1] = 0	0	-

5.6 MITMEMÕÕTMELINE DP TABEL – MILJONÄR JA VAESLAPSED

Järgmine ülesanne on Eesti lahtiselt programmeerimisvõistluselt aastast 2015. See on kõigi aegade kõige kurvema tekstiga programmeerimisülesanne, aga selle lahendus on üle kantav paljudele teistele statistilise modelleerimise ülesannetele.



Dickensi-aegsel Inglismaal elas miljonär Mortimer. Temaga samas linnas asusid kolm lastekodu, kus elasid vaeslapsed, kellele Mortimer tavatses jõulukinke teha. Kinkide jagamise protseduur oli järgmine:
1. Iga vaeslaps saab oma lastekodust korvi, millega ta kingi järele läheb.
2. Mortimer viskab kingitusi järjest laste sekka, mida nood oma korvidega püüavad.
3. Iga kingitus püütakse alati kinni.
4. Lapsed saavad kingitusi kätte juhuslikult, kuid tõenäosus, et konkreetne laps kingituse kätte saab, on võrdeline tema korvisuu pindalaga.
5. Sama lastekodu lastel on sama suurusega korvid.
6. Kui mõni laps saab kingituse kätte, läheb ta sellega kohe lastekodusse tagasi ja rohkem püüdmises ei osale.
7. Lapsi võib olla rohkem kui kingitusi :(
Igal kingitusel on väärtus. Leida iga lastekodu kohta, milline on selle kodu laste poolt saadud kingituste väärtuste keskmine eeldatav summa.

NÄIDE:
Esimesest lastekodust on kohal $L_1 = 1$ laps korvi pindalaga $K_1 = 1$.
Teisest lastekodust on kohal $L_2 = 1$ laps korvi pindalaga $K_2 = 2$.
Kolmandast lastekodust on kohal $L_3 = 1$ laps korvi pindalaga $K_3 = 3$.
Mortimeril on `N = 2` kingitust väärtustega `V = [10, 20]`.
Vastus:
L_1: 6,666667
L_2: 11,333333
L_3: 12

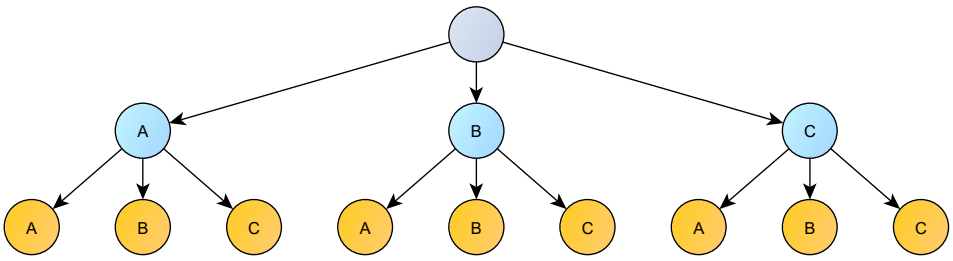
Näiteandmeid kasutades kujuneb vastus esimese lastekodu kohta järgnevalt:

- Esimese lastekodu ainus laps saab esimese kingituse kätte tõenäosusega $\frac{1}{6}$, see annab oodatavaks väärtuseks $10 * \frac{1}{6} = \frac{5}{3} \approx 1{,}666667$ ning teiste kingituste püüdmises ta ei osaleks.
- Tõenäosusega $\frac{1}{3}$ saab esimese kingi teise lastekodu laps ja läheb ära. Sel juhul on esimesel lapsel teise kingi saamise tõenäosus $\frac{1}{4}$. Oodatav väärtus on siis $20 * \frac{1}{4} * \frac{1}{3} = \frac{5}{3} \approx 1{,}666667$.
- Tõenäosusega $\frac{1}{2}$ saab esimese kingi kolmanda lastekodu laps ja lahkub. Sel juhul on esimesel lapsel teise kingi saamise tõenäosus $\frac{1}{3}$, oodatav väärtus $20 * \frac{1}{2} * \frac{1}{3} = \frac{10}{3} \approx 3{,}333333$.

Kokku tulebki vastuseks $\frac{5}{3} + \frac{5}{3} + \frac{10}{3} = \frac{20}{3} \approx 6{,}666667$. Samasugust arutlust saab kasutada ka teiste lastekodude jaoks.

5.6.1 Kõikide võimaluste läbivaatus

Tõenäosus, et kingi saab kätte mingi konkreetse lastekodu laps, sõltub kogu selle lastekodu püüdmisel osalevate korvide pindala suhtest kõigi veel püüdmisel osalevate korvide pindalasse. Iga kord, kui kink kätte saadakse, väheneb ühe lastekodu korvide kogupindala, kuna kingi saanud laps lahkub koos oma korviga. Seetõttu väheneb ka kõigi korvide pindala. Kinkide püüdmisele vastab järgmine puu:

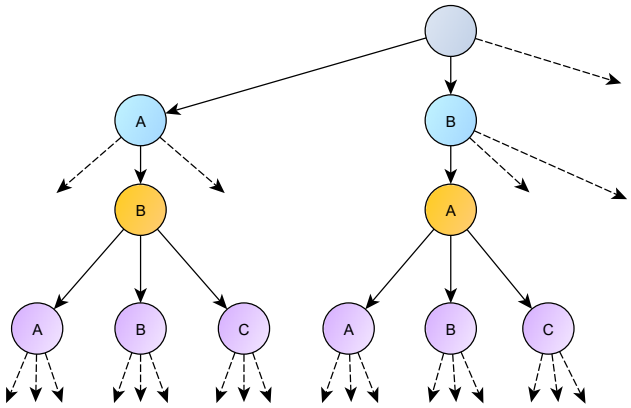


A, B ja C tähistavad erinevaid lastekodusid, esimene tase (sinised tipud) esimest kingitust, teine tase (oranžid tipud) teist kingitust. Igas tipus on sellele lastekodule vastav täht, kes vastava taseme kingituse sai. Kõikide variantide läbivaatamise keerukus sõltub kinkide arvust N ja on $O(3^N)$.

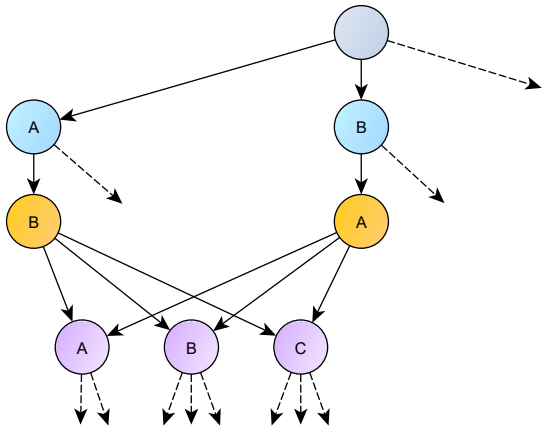
5.6.2 Korduvad harud

Kas läbivaatuste puus hakkavad mingid osad korduma? Lapsed saavad ükshaaval kinke ning lahkuvad seejärel. i-nda kingi jagamise ajaks on lahkunud i last ja lastekodude ning ka kogu korvide pindala on

vähenenud vastavalt sellele, milliste lastekodude lapsed olid juba lahkunud. Samas ei sõltu allesjäänud laste ja korvide arv sellest, millises järjekorras eelmised lapsed oma kinke said. Siin on alamosa sügavamast rekursioonipuust:



Ülesande seisukohast on pärast oranži taset toimuv kinkide jagamine täpselt sama:



5.6.3 DP tabeli koostamine

Kui kõik kingid on jagatud, siis kingi saamise tõenäosus on 0. Kui on alles viimane kink, siis tõenäosus, et selle saab mõni esimese lastekodu laps, on esimese lastekodu veel kohale jäänud laste arv LJ_1 korrutatud esimese lastekodu korvi pindalaga K_1 ning jagatud kõikide allesjäänud korvide pindalaga. Et seda leida, on vaja teada, mitu last igast lastekodust alles on. Hea on koostada kolmemõõtmeline massiiv, kus iga mõõde vastab ühe lastekodu kohal olevate laste arvule. Näiteks, kui on vaja jagada 5 kinki ja lastekodudest on kohal vastavalt 2, 5 ja 1 last, siis viimase kingi jagamise ajaks võib olla esimesest lastekodust alles 0 kuni 2 last, teisest 1 kuni 4 last ja kolmandast 0 või 1 last. Kokku on järel 2 + 5 + 1 - 4 = 4 last. Järgnevas tabelis on toodud viimase kingi jagamisest saadavad keskmised eeldatavad summad esimese lastekodu jaoks:

A	B / C	1	2	3	4	5
0	0	-	-	-	0	-
	1	-	-	0	-	-
1	0	-	-	$\frac{K_1}{K_1 + 3 * K_2} * V$	-	-
	1	-	$\frac{K_1}{K_1 + 2 * K_2 + K_3} * V$	-	-	-
2	0	-	$\frac{2 * K_1}{2 * K_1 + 2 * K_2} * V$	-	-	-
	1	$\frac{2 * K_1}{2 * K_1 + K_2 + K_3} * V$	-	-	-	-

A, B ja C tähistavad vastavalt esimesest, teisest ja kolmandast lastekodust alles jäänud laste arvu. K_1, K_2 ja K_3 tähistavad vastavalt lastekodude korvi pindalasid ning V viimase kingituse väärtust.

Kuidas leida, milline on oodatav kinkide väärtus üks käik enne seda, näiteks juhul, kui alles on kaks last esimesest, kaks teisest ja üks kolmandast lastekodust? Siis on 3 võimalust, millise lastekodu laps kingi saab:

1. Kingi saab esimese lastekodu laps ning pärast seda on situatsioon, kus lapsi on järel A = 1, B = 2 ja C = 1 (tabelis oranži taustaga lahter).
2. Kingi saab teise lastekodu laps ning pärast seda on lapsi järel A = 2, B = 1 ja C = 1 (tabelis rohelise taustaga lahter).
3. Kingi saab kolmanda lastekodu laps ning pärast seda on lapsi A = 2, B = 2 ja C = 0 (tabelis sinise taustaga lahter).

Tähistagu kingi saamise tõenäosusi antud sammul vastavalt T_A, T_B ja T_C ja parasjagu jagatava kingituse väärtust VP, siis

$$E(A_i, B_{j,} C_k) = \begin{cases} 0, kui\ i = 0, j = 0, k = 0 \\ T_A * VP\ +\ T_A * E(A_{i-1}, B_j, C_k)\ +\ T_B * E(A_i, B_{j-1}, C_k)\ +\ T_C * E(A_i, B_j, C_{k-1}) \end{cases}$$

Ülesandes tahetakse vastust kõigi kolme lastekodu jaoks, seega tuleb teha 3 tabelit: üks iga lastekodu jaoks.

5.6.4 Lahendus

Siin on funktsioon, mis antud ülesande lahendab:

```
int kingid(int La, int Lb, int Lc, int Ka, int Kb, int Kc, int N, int* kingid)
{
    double**** kv = new double***[La + 1]; //1. lastekodust kohalolevate laste arv
    for (int i = 0; i <= La; i++) {
        kv[i] = new double**[Lb + 1]; //2. lastekodust kohalolevate laste arv
        for (int j = 0; j <= Lb; j++) {
            kv[i][j] = new double*[Lc + 1]; //3. lastekodust kohalolevate laste arv
            for (int k = 0; k <= Lc; k++) {
                kv[i][j][k] = new double[3]; // lastekodu
            }
        }
    }

    int lapsiKokku = La + Lb + Lc;
    //Alustame sellest, kui kõik lapsed on lahkunud kuni selleni, kui kõik veel alles
    for (int al = lapsiKokku - N; al <= lapsiKokku; al++) {
        int m1 = min(La, al);//nii palju saab maksimaalselt olla alles 1. lastekodust
        for (int i = 0; i <= m1; i++) {
            //nii palju saab olla 2. lastekodust, kui esimesest on alles i last:
            int m2 = min(Lb, al - i);
            for (int j = 0; j <= m2; j++) {
                if (al - i - j <= Lc) {
                    int k = al - i - j; //nii palju on 3. lastekodust kohal
                    kv[i][j][k][0] = 0;
                    kv[i][j][k][1] = 0;
                    kv[i][j][k][2] = 0;
                    if (al == lapsiKokku - N) {
                        continue; //kedagi ei ole, jätkame
                    }
                    double p1 = i * Ka;
                    double p2 = j * Kb;
                    double p3 = k * Kc;
                    double kp = p1 + p2 + p3; //alles korvide kogupindala
                    double t1 = p1 / kp; //Tõenäosus, et kingi saab 1. lastekodu
                    double t2 = p2 / kp; //2. lastekodu
                    double t3 = p3 / kp; //3. lastekodu
                    int jk = kingid[lapsiKokku - al]; //jagatava kingi väärtus

                    if (i != 0) {
                        kv[i][j][k][0] += t1 * (jk + kv[i - 1][j][k][0]);
                        kv[i][j][k][1] += t1 * kv[i - 1][j][k][1];
                        kv[i][j][k][2] += t1 * kv[i - 1][j][k][2];
                    }
                    if (j != 0) {
                        kv[i][j][k][0] += t2 * kv[i][j - 1][k][0];
                        kv[i][j][k][1] += t2 * (jk + kv[i][j - 1][k][1]);
                        kv[i][j][k][2] += t2 * kv[i][j - 1][k][2];
                    }
                    if (k != 0) {
                        kv[i][j][k][0] += t3 * kv[i][j][k - 1][0];
                        kv[i][j][k][1] += t3 * kv[i][j][k - 1][1];
                        kv[i][j][k][2] += t3 * (jk + kv[i][j][k - 1][2]);
                    }
                }
            }
        }
    }

    cout << fixed << setprecision(10) << kv[La][Lb][Lc][0] << endl <<
        kv[La][Lb][Lc][1] << endl << kv[La][Lb][Lc][2] << endl;
    return 0;
}
```

Selles lahenduses on DP tabeli dimensioonideks kasutatud küll lastekodulaste arvu, kuid tegelikud arvutused toimuvad ainult laste arvu ja kinkide arvu vahe osas (täpsemalt võiks mõõtmeteks olla nt lahkunud laste arv, mis saab maksimaalselt võrduda kinkide arvuga). Seega on kogu selle lahenduse keerukus $O(N^3)$.

5.7 TÕEVÄÄRTUSTE TABELIGA DP - ÕIGLANE JAGAMINE

See ülesanne on teisend klassikalisest seljakoti pakkimise ülesandest. Seljakoti pakkimisest tuleb hiljem rohkem juttu kaheksandas peatükis – DP edasijõudnutele.

Kaks venda, Albert ja Benno, tahavad jagada omavahel hulka kingitusi. Iga kingituse peab andma kas Albertile või Bennole; kingitusi poolitada ei saa. Igal kingitusel on väärtus. A ja B tähistavad vastavalt Albertile ja Bennole antud kingituste kogusummat. Minimeerida vahe A - B absoluutväärtus. Kingituste arv N ei ületa 100. Kingituste väärtused on positiivsed täisarvud, mis ei ületa 200.
NÄIDE:
```
N = 4
V = [2, 3, 4, 7]
```
Vastus: 2 (näiteks jaotused [2, 7] ja [3, 4] või [7] ja [2, 3, 4])

5.7.1 Kõik kombinatsioonid

Esimese hooga tuleb mõte vaadata läbi kõik kombinatsioonid. Kuna kinkide jagamisel tekkiv vahe on sümmeetriline (st ei ole vahet, kas konkreetse komplekti kingitusi saab Albert ja ülejäänud jääb Bennole või vastupidi), siis piisab pooltest võimalustest. Näiteandmete jaoks on need toodud järgmises tabelis:

Alberti kinkide väärtused	2, 3, 4, 7	3, 4, 7	2, 4, 7	2, 3, 7	2, 3, 4	4, 7	3, 7	3, 4
Benno kinkide väärtused	0	2	3	4	7	2, 3	2, 4	2, 7
Vahe	16	12	10	8	2	6	4	2

Tundub hea plaan. Erinevaid kombinatsioone kinkide jagamiseks on 2^{N-1}. Kui N on väike, nagu antud näites, siis töötab kõik suurepäraselt, kuid kui N = 100, teeb see juba 2^{99} erinevat kombinatsiooni ja see on isegi masinale liiga palju tööd.

5.7.2 DP lahendus

Eelmises tabelis on näha, et üks kinkide jaotamisel tekkinud väärtuste vahe kordub. Kuna kinkide lubatud väärtus on kuni 200, siis nii väikese kinkide arvu juures nagu näites (N = 4) see ei pruugi ilmtingimata nii olla, kuid suurema arvu kinkide jagamisel ilmselt peavad hakkama vahed korduma, sest kõigi võimalike vahede arv on fikseeritud. Näiteks maksimumsisendi juures, kus kinke on 100 ja igaüks väärtusega 200, siis juhul kui Albert ahnitseb kõik kingitused endale, on Alberti kingituste koguväärtus kõigi kingituste väärtuste summa, mis on maksimaalselt 100 * 200 = 200 000. Benno kingituste koguväärtus on loomulikult 0, seega maksimaalne vahe saab olla 200 000. See tähendab aga, et nende kingituste ükskõik millisel jagamisel ei saa tekkida rohkem kui 200 001 erinevat vahet (0 sobib ka). Pane tähele: see kehtib seetõttu, et kingituste väärtused ja seega ka vahed saavad olla ainult täisarvud, reaalarvuliste väärtuste korral saab sellist lähenemist kasutada ainult teatud tingimustel.

Näites on kingituste kogusumma 2 + 3 + 4 + 7 = 16, järelikult võimalikud vahed on 0…16. Kuidas muutuvad vahed kingituste jagamise käigus?

Kui jagatakse ainult üks kingitus, saab vaheks olla ainult selle kingituse väärtus (tabelis '+' tähistab võimalikku vahet, '-' võimatut. Reas on kingituse nr(= väärtus)):

Vahe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Kui jagatakse järgmine kingitus, siis kõik võimalikud vahed saavad kas kasvada või kahaneda täpselt selle kingituse väärtuse võrra, nt kui esimese kingituse väärtusega 2 sai Albert ja teise, väärtusega 3, saab samuti Albert, on vahe 2 + 3 = 5. Kui teine kink läheb aga Bennole, on vahe $|2 - 3| = 1$.

Vahe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-

Kolmanda kingituse jagamisel saab vahe muutuda 4 võrra, sest see on kolmanda kingituse väärtuseks. Kui enne sai olla väärtuste vahe kas 1 või 5, siis nüüd on võimalikud vahed:

- 1 + 4 = 5,
- $|1 - 4| = 3$,
- 5 + 4 = 9 ja
- 5 – 4 = 1.

Samamoodi saab arvutada kõik võimalikud vahed ka viimase kingituse jaoks. Siin on kogu täidetud tabel näites toodud 4 kingi jaoks:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-
3	-	+	-	+	-	+	-	-	-	+	-	-	-	-	-	-	-
4	-	-	+	-	+	-	+	-	+	-	+	-	+	-	-	-	+

Siin on ka võimalus kokkuhoidliku DP kasutamiseks: iga kord kasutatakse järgmise rea arvutamiseks ainult eelmisel real olevat infot – see tähendab, et kui ülesande vastuseks soovitakse ainult minimaalset vahet, piisab tegelikult kahe reaga tabelist: vaja on meeles pidada eelmine rida ning käesolev rida.

Järgmine funktsioon illustreerib kokkuhoidlikku DP lahendust:

```
int LeiaVahe(int mituPakki, int* pakid)
{
    int suurimVahe = 0;
    for (int i = 0; i < mituPakki; i++) {
        suurimVahe += pakid[i]; //Leiame suurima võimaliku vahe
    }
    bool** tabel = new bool*[suurimVahe + 1];
    for (int i = 0; i <= suurimVahe; i++) {
        tabel[i] = new bool[2];//kokkuhoidliku DP jaoks on vaja vaid 2 rida
    }
    tabel[0][1] = 1; //kui midagi jagada pole, on vahe 0
    for (int i = 1; i <= suurimVahe; i++) {
        tabel[i][1] = 0; //ülejäänud vahed on võimatud
    }
    for (int i = 0; i < mituPakki; i++) {
        int praeguneRida = i % 2;
        int eelmineRida = (i + 1) % 2;
        for (int j = 0; j < suurimVahe; j++) {
            int eelmineVahe = j - pakid[i];
            if (eelmineVahe < 0){
                eelmineVahe = -eelmineVahe;
            }
            if (tabel[eelmineVahe][eelmineRida]) {
                tabel[j][praeguneRida] = 1;
                continue;
            }
            eelmineVahe = j + pakid[i];
            if (eelmineVahe > suurimVahe || !tabel[eelmineVahe][eelmineRida]) {
                tabel[j][praeguneRida] = 0;
                continue;
            }
            tabel[j][praeguneRida] = 1;
        }
    }
    int vastus;
    for (int i = 0; i < suurimVahe; i++) {
        int praeguneRida = (mituPakki - 1) % 2;
        if (tabel[i][praeguneRida]) {
            vastus = i; //leiame esimese võimaliku vahe, see ongi vastuseks
            break;
        }
    }
    return vastus;
}
```

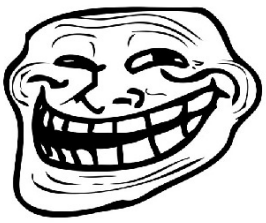
5.8 BITIMASKIDE PÕHINE VÄÄRTUSTE TABEL - MOEKUNSTNIK JA KOILIBLIKAS

Järgmine ülesanne on samuti Eesti lahtiselt programmeerimisvõistluselt aastast 2013. Ülesande originaaltekst kuulub selgelt fantastika valdkonda, kuid samal põhimõttel saab lahendada mitmesuguseid mänguteooria või riskianalüüsi situatsioone.

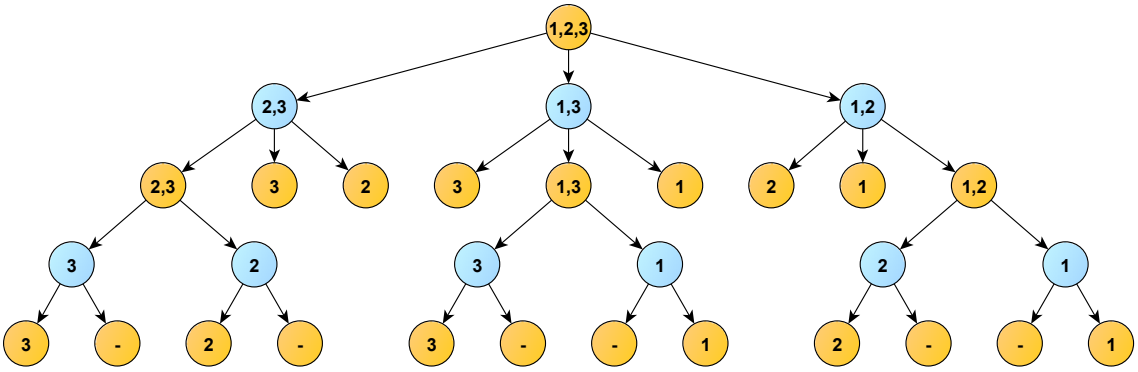
Moekunstnik Märdil on kapis N ilusat salli, millega ta käib laupäeviti moekunstnike koosolekul. Igal sallil on oma moeväärtus ja koosolekul saab Märt vastava hulga feimi. Kaks korda sama salliga kohale tulla oleks suur *faux pas* ja Märt ei tee seda kunagi. Kantud sallid paneb ta kappi tagasi, aga rohkem neid ei kanna. Märdi kapis elab ka koiliblikas Kärt, kes sööb igal pühapäeval ühele sallile augu sisse. Auguga salli enam kanda ei saa. Kärt moeväärtusest ei hooli, vaid sööb salle juhuslikult. Mingile sallile augu söömise tõenäosus on võrdeline salli pikkusega. Kärt võib sama salli süüa ka mitu korda. On selge, et Märt saaks koosolekul käia ülimalt N korda, kui Kärt sööks ainult juba kantud salle. Kui Kärt sööb mõnikord ka kandmata salle, jääb koosolekute arv sellevõrra väiksemaks. Kirjutada programm, mis leiab, kui palju Märt keskmiselt feimi saab, kui ta kasutab parimat võimalikku strateegiat, aga Kärt sööb salle juhuslikult. Tegevus algab nädala alguses, seega esimese koosoleku ajaks on kõik sallid veel terved. Sisendina on antud kõigi sallide pikkused ja moeväärtused.
Sisendi esimesel real on sallide moeväärtused ja teisel real nende pikkused.
NÄIDE:
Pikkused: `3 2 1`
Moeväärtused: `10 20 30`
Vastus:
48,333333

(Pildil Märt, Kärt ja ülesande autor Targo).





Märdi valikud ei muutu. Kui aga Kärt sõi mõne kandmata salli, jääb Märdile N - 2 valikut jne. Siin on puu kolme salli jaoks:



Numbrid tippudes näitavad, millised sallid on veel Märdile kasutatavad. Oranžides ringides on Märdi käikudele, sinistes Kärdi käikudele eelnev seis.

Märt saab igal sammul valida järelejäänud sallide hulgast – Kärt valib küll kõigi sallide hulgast, kuid tegelikult pole ülesande seisukohalt oluline, millise katkistest või kantud sallidest Kärt valib, neid võib vaadelda ühe juhuna. Sellisel juhul kõikide variantide läbivaatamisega lahenduse keerukus on $O((N!)^2)$. Seega juba N = 20 juures tuleb teha $(20!)^2 = 5{,}919*10^{36}$ läbivaatust. Arvestades 10^8 läbivaatust sekundis, kulub $2*10^{21}$ aastat!

5.8.2 Tee DP poole

Kui vaadata hoolega eespool joonistatud puud, pole raske märgata, et puus hakkavad harud korduma. Sisuliselt tähendab see seda, et kui mingil hetkel on Märdil valida täpselt samade sallide vahel, siis on tal tark täpselt samamoodi käituda ning ta saab keskmiselt samapalju feimi juurde eelnevalt kogutule: nii näiteks kolmanda päeva valik ei sõltu enam sellest, mida Märt ja Kärt esimesel kahel päeval ette võtsid, vaid ainult sellest, millised sallid veel selleks ajaks järel on. Piisab, kui samasuguseid alampuid arvutame vaid ühel korral, edaspidi võib kasutada meelde jäetud väärtust. Seega võiks mõte minna DP lahenduse peale. Erinevaid seise on sama palju kui parajasti alles olevate sallide võimalikke kombinatsioone ehk 2^N. Iga kombinatsiooni kohta on vaja meeles pidada parim tulemus. Kuidas aga järjestada kombinatsioone?

5.8.3 Bitimaskide kasutamine tabeli indeksitena

Siin on olukord, kus süsteemi võimalikud olekud on kirjeldatavad bitimaskidena: mingi hulk objekte on parajasti kas aktiivsed või mitteaktiivsed. N objekti puhul on selliseid võimalusi 2^N. Sellisel juhul on kaval luua 2^N elemendiga massiiv, kus iga element tähistab üht võimalikku kombinatsiooni. Kui massiivi indeksid on 0 kuni $2^N - 1$, annab see meile ka loomuliku viisi kombinatsioonide järjestamiseks: iga massiivi element tähistab parajasti nende objektide olemasolu, millele vastavad bitid on indeksi kahendesituses ühed.

Näiteks nelja objekti puhul tähendaks see järgmist tabelit:

Massiivi indeks	Kahendesitus	Allesolevad objektid
0	0000	Mitte ühtegi
1	0001	1.
2	0010	2.
3	0011	1. ja 2.
4	0100	3.
5	0101	1. ja 3.
6	0110	2. ja 3.
7	0111	1., 2. ja 3.
8	1000	4.
9	1001	1. ja 4.
10	1010	2. ja 4.
11	1011	1., 2. ja 4.
12	1100	3. ja 4.
13	1101	1., 3. ja 4.
14	1110	2., 3. ja 4.
15	1111	Kõik objektid

Madalamad bitid tähistavad esimesi objekte, sest siis me ei piira objektide koguarvu – kui neid on rohkem, saab võtta lihtsalt pikema arvu.

Tavapäraste operatsioonide sooritamiseks on kasulikud bitikaupa tehted:

Operatsioon	Operaator	Näide kahendesituses	Näide kümnendesituses
Bittide nihutamine vasakule. M << N on ekvivalentne tehtega $M*2^N$.	<<	0101 << 2 == 010100	5 << 2 == 20
Bitikaupa korrutamine	&	0101 & 0111 == 0001	5 & 7 == 1
Bitikaupa liitmine	|	0101 | 0011 == 0111	5 | 3 == 7
Bitikaupa eitus	~	~0101 == 1010	Vastus oleneb arvu pikkusest
Bitikaupa välistav või	^	0101 ^ 0011 == 0110	5 ^ 3 == 6

Nende abil saab teha järgmisi operatsioone väga efektiivselt:

- Kõigi kombinatsioonide arvu (2^N) leidmine.

```
1 << N
```

- Konkreetsele objektile vastava arvu leidmine. Tehe tagastab kahendarvu, kus altpoolt m. bitt on 1 ja teised bitid nullid.

```
1 << (m - 1)
```

- Kombinatsiooni arvu leidmine, kus on olemas kõik objektid peale ühe. Tegemist on eelmise tehte tulemuse bittide ümberpööramisega.

```
~(1 << (m - 1))
```

- Tsükkel, mis genereerib üksikutele objektidele vastavaid kahendarve (sisuliselt kahe astmeid):

```
for (int mbit = 1; mbit <= max; mbit *= 2)
```

- Kontroll, kas kombinatsioon i sisaldab objekti number m. Kui kombinatsioonile vastavat arvu bitikaupa korrutada ühele objektile vastava arvuga, on vastus positiivne parajasti siis, kui kombinatsiooni arvus on ka m. bitt seatud.

```
i & 1 << (m - 1) > 0
```

- Objekti eemaldamine kombinatsioonist (täpsemalt, kombinatsiooni i alusel sellise kombinatsiooni leidmine, kus puudub objekt m).

```
i & ~(1 << (m - 1))
```

- Objekti lisamine kombinatsioonile (täpsemalt, indeksi i alusel sellise indeksi leidmine, kuhu on lisatud objekt m):

```
i | 1 << (m - 1)
```

- Objekti sisse-väljalülitamine (lihtsustab koodi juhul, kui me juba ette teadsime, kas see objekt oli enne aktiivne või ei):

```
i ^ 1 << (m - 1)
```

Selline lähenemine on arvutuslikult üliefektiivne. Konkreetsetele objektikombinatsioonidele vastavate massiivi indeksite leidmine ja nendega opereerimine taanduvad täisarvutehetele, mille jaoks kaasaegsetes protsessorites on hästi optimiseeritud meetodid ja mille täitmist mõõdetakse üksikutes nanosekundites.

5.8.4 DP lahendus

Siin on põhjalike kommentaaridega funktsioon, mis antud ülesandele vastuse leiab:

```
double leiaFeim(int N, int* moevaartused, int* pikkused)
{
    // võimalike kombinatsioonide arv, millised sallid võivad söödud olla = 2^n
    int kombinatsioonid = 1 << N;
    int kogupikkus = 0;
    for (int i = 0; i < N; i++) {
        kogupikkus += pikkused[i];
    }
    // Siin massiivis hoiame parimaid võimalikke tulemusi, mida on võimalik saavutada,
    // kui meil on alles mingi konkreetne kombinatsioon salle.
    // Massiivi indeksid vastavad sallikombinatsioonide kahendarvulistele väärtustele.
    // Näiteks feimid[37] sisaldab maksimaalset võimalikku väärtust, mida Märdil on
    // võimalik saavutada juhul, kui alles on 1., 3. ja 6. sall (sest 10-süsteemis 37
    // on kahendsüsteemis 00100101, seatud on altpoolt lugedes 1., 3. ja 6. bitt)
    double* feimid = new double[kombinatsioonid];
    feimid[0] = 0;
```

```
    // konstrueerime kõik kombinatsioonid alates lihtsamatest
    for (int i = 1; i < kombinatsioonid; i++) {
        feimid[i] = 0;

        // Proovime uue kombinatsiooni üles ehitada väiksematest.
        // mvalik on Märdi tehtav valik, st mitmenda salli ta võtab.
        // Proovime kõiki Märdi valikuid
        for (int mvalik = 0, mbit = 1; mbit <= i; mbit *= 2, mvalik++) {
            // bitt pole seatud, st vaadeldav kombinatsioon ei sisalda seda salli
            if ((mbit & i) == 0){
                continue;
            }
            double alles = 1;  // tõenäosus, et ühtki head salli ei sööda ära
            double feim = moevaartused[mvalik];  // Märdi käigu väärtus
            // kvalik on Kärdi tehtav valik, st mitmenda salli sisse ta augu sööb
            for (int kvalik = 0, kbit = 1; kbit <= i; kbit *= 2, kvalik++) {
                if ((i & kbit) == 0 || mvalik == kvalik) continue;
                double prob = (double)pikkused[kvalik] / kogupikkus;
                // väärtusele lisandub selle allesjääva kombinatsiooni väärtus, kust
                //on eemaldatud Märdi ja Kärdi sall
                feim += prob * feimid[i ^ mbit ^ kbit];
                alles -= prob;
            }
            // Kui Kärt ei söönud ühtki head salli ära, lisandub selle kombinatsiooni
            // väärtus, kus on kõik sallid peale Märdi praeguse valiku
            feim += alles * (feimid[i ^ mbit]);
            if (feim > feimid[i]) {
                feimid[i] = feim;
            }
        }
    }
    return feimid[kombinatsioonid - 1];
}
```

5.9 KUIDAS JA MILLAL DP-D KASUTADA

Eelnevalt toodud näidetest selgus, et DP aitab algoritmi keerukust sageli dramaatiliselt parandada:

Ülesanne	Jõumeetodiga keerukus	DP keerukus
Fibonacci	$O(\varphi^N)$	O(N)
Optimaalne maksmine	$O(2^N)$	O(N)
Pikim kasvav osajada	$O(2^N)$	$O(N^2)$
Pikim ühine osajada	$O(2^{N+M})$	O(NM)
Ristsumma	O(N)	O(log N)
Õiglane jagamine	$O(2^N)$	O(N*log(W)), kus W on kingi maksimaalne väärtus
Miljonär ja vaeslapsed	$O(3^N)$	$O(N^3)$
Moekunstnik ja koiliblikas	$O(N!^2)$	$O(2^N)$

Kuidas aga ära tunda ülesandeid, kus on võimalik DP-d kasutada?

- Probleemi saab jagada väiksemateks alamprobleemideks.
- Suurema probleemi lahenduse saab leida ühe või enama alamprobleemi lahenduse põhjal.
- Alamprobleemid on järjestatavad.

- Suurema probleemi lahendus sõltub ühest või enamast alamprobleemi lahendusest, kuid mitte sellest, kuidas need lahendused täpselt saadi.

Tavalisi DP ülesannete struktuure:

Sisend	DP tabelis kasutatav(ad) väärtus(ed)	Üleminek	Näiteid
Jada $x_1, x_2, \ldots, x_n$	Jada element i	Vali jada esimesed i-1 elementi, töötle i, lisa i eelmisele seisule või mitte.	Pikim kasvav osajada
Kaks jada $x_1, x_2, \ldots, x_n$ ja $y_1, \ldots, y_m$	Jadade elementide paar i (esimesest jadast) ja j (teisest jadast)	Nagu eelmine, aga muuda üht indeksit korraga.	Pikim ühine osajada
Jada $x_1,\ldots x_i,\ldots x_j,\ldots,x_n$	Alamjada $x_i,\ldots,x_j$	Jaga lõik i,…,j lõikudeks i,…,k k+1,…,j	Maatriksite jada korrutamine (vt peatükk 8)
Suunatud tsükliteta graaf	Tipp graafis	Töötle vaadeldava tipu naabrid	Pikimate või lühimate teede leidmine graafis, teede loendamine
Piirav arvuline väärtus	Piiriväärtusest väiksem (või suurem) arv	Suurenda (või vähenda) vaadeldavat väärtust kuni jõuad piiriväärtuseni.	Seljakotiülesanne, mitmesugused maksmisülesanded
Objektide hulk	Objektide alamhulk (tavaliselt bitimaskina)	Lülita alamhulgas elemente sisse-välja	

5.9.1 Ülalt alla vs alt üles DP

	Ülalt alla	Alt üles
Plussid	- Loomulik järgmine samm täisläbivaatusega lahendusest. - Leiab alamvastused ainult siis, kui see on vajalik (mõnedes olukordades kiirem).	- Kiirem juhul kui paljusid alamprobleeme „külastatakse“ korduvalt - Võimaldab mälu kokkuhoidu, kui kasutame „kokkuhoidliku DP“ trikki.
Miinused	- Rekursiivsel funktsioonide väljakutsumisel on lisakulu. - Kõigi olekute meelespidamine võib põhjustada mäluprobleeme.	- Lugejale, kes pole DP-ga tuttav, on koodi intuitiivne mõistmine raskem. - Leiab kõik alamvastused ka juhul, kui see pole otseselt vajalik.

5.9.2 Kidunud DP

Vahel on võimalik leida DP ka olukordades, kus seda esmapilgul ei ole. Olgu ülesandeks näiteks:

> Leida arvude jadas maksimaalne element.

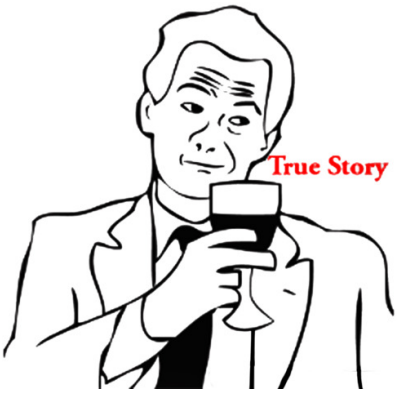
Loogiline ja intuitiivne lahendus on meeles pidada, mis on seni leitud maksimaalne element, võrrelda seda kõigi järgmiste elementidega ning kui leiame suurema, siis meeles peetud element selle suurema väärtuse vastu vahetada. Samas võib ka sellest algoritmist mõelda kui DP algoritmist, kus „tabel“ koosnebki sellestsamast seni leitud maksimaalsest elemendist. Alternatiivne „jõumeetodiga“ lahendus samal ülesandele oleks iga elemendi jaoks proovida, kas ta on kõigist teistest suurem või ei: $O(N^2)$ ajalise keerukusega algoritm.

5.10 DP PRAKTILISED KASUTUSALAD

Me kasutame sageli tööriistu, mis sisaldavad endas ühte või teist DP-l põhinevat algoritmi.

Mõned huvitavad DP kasutusalad:

- diff (https://en.wikipedia.org/wiki/Diff_utility) ja tema sugulased, mis kasutavad pikima ühise osajada leidmist.
- Polünoomide interpoleerimine (https://en.wikipedia.org/wiki/Polynomial_interpolation) on kergesti realiseeritav DP abil. Polünoomide interpoleerimine ise leiab kasutust mitmel pool arvutigraafikas, kus meil on vaja keerulisi jooni lihtsa vaevaga kuvada. Näiteks fontide skaleerimist saab realiseerida, kasutades polünoomide interpolatsiooni.
- Spordiennustus ja modelleerimine, eriti mängudes, mis koosnevad diskreetsetest etappidest, näiteks kriket ja pesapall. Vt nt https://en.wikipedia.org/wiki/WASP_(cricket_calculation_tool)
- Teksti joondamise ja poolitamise algoritmid (kuidas lõpetada ridu nii, et raisataks võimalikult vähe ruumi), mida kasutavad TeX ja teised küljendusprogrammid.
- Plagiaadi leidmise tööriistad kasutavad teisenduskauguse (i.k *edit distance*) leidmise algoritmi, mis põhineb DP-l. Teisenduskaugusest tuleb rohkem juttu peatükis „DP edasijõudnutele“.
- Lühima tee leidmise algoritmid kaardirakendustes.
- Logistika planeerimine.
- Markovi ahelad DNA-s mustrite leidmiseks.
- Soovituste andmine otsingumootorites.
- Võimsuse optimaalne planeerimine elektrijaamades.

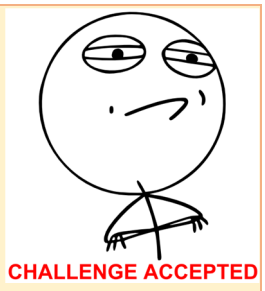


Üks päriselu olukord, kus mul isiklikult DP-laadsest lähenemisest kõige rohkem kasu on olnud, oli seotud andmebaasidega. Tegu oli suure, paljude klientidega olmeettevõttega. Kõigil klientidel oli palju mitmesuguseid tarbimisi. Iga kuu lõpus oli ärianalüütikutel vaja teha musttuhat raportit ja päringut erinevate tarbimisliikide, kliendisegmentide jm kohta. Täiendavalt tuli teha simulatsioonid, et mis juhtuks, kui hinnad muutuksid. Algne lahendus sisaldas hulka andmebaasipäringuid, mis töötasid aga terve nädalavahetuse, enne kui raportid valmis said. Kui asja uurisin, siis selgus, et üsna paljusid andmeid päriti kogu selles protsessis palju kordi. Et seda vältida, ehitasin eelpool kirjeldatud DP-tabelite analoogina hulga ajutisi tabeleid, milles jupphaaval lõplikku statistikat koostati. Kokkuvõttes valmisid raportid nüüd päevade asemel minutitega.

Selline lähenemine, kus andmebaasis andmeid teatud päringute huvides denormaliseeritakse, on küllaltki levinud (vt nt https://en.wikipedia.org/wiki/Denormalization), aga selle kombineerimine DP mõtteviisiga lisab su tööriistakasti veel mõned huvitavad võimalused.

5.11 KONTROLLÜLESANDED

5.11.1 Aktsiaturg



Kui siin õpikus oleks kirjas kindel retsept aktsiaturul raha teenimiseks, võiks selle raamatu iga eksemplari eest küsida tuhandeid eurosid. Kuna aga raamatutest pole võimalik kindlaid meetodeid leida, luuravad paljud kauplejad üksteise järel, püüdes jälile saada võitvatele strateegiatele.
Sul on antud võistleva kaupleja tehingute nimekiri, mille igal real on üks number tehingu kohta – täisarvuline kasum või kahjum.
Eesmärgiks on leida sellest jadast maksimaalse summaga alamjada. Kui kasumlikke tehinguid ei ole, väljastada 0.
Sisendi esimesel real on tehingute arv N (1 ≤ N ≤ 10000). Teisel real on igast tehingust saadud kasum või kahjum.

NÄIDE 1:

```
5
12 -4 -10 4 9
```

Vastus: 13

NÄIDE 2:

```
20
-896 995 -588 -23 648 -980 51 -705 -620 -640 915 663 444 307 -207 81 -763 -993
24 665
```

Vastus: 2329

5.11.2 Protsessori planeerimine

Operatsioonisüsteem peab järjekorda seadma hulga tegevusi, mille sooritamiseks on vaja protsessoriaega. Tegevusi on kaht sorti, esimeste täitmine võtab `n` millisekundit ja teiste täitmine `m` millisekundit.
Eesmärgid (prioriteedijärjekorras) on esiteks minimeerida aega, mil protsessor on jõude (ideaalis null, kui null pole võimalik, siis nii väike kui saab) ning teiseks maksimeerida täidetud ülesannete arvu. Töö tegemiseks on aega `t` millisekundit. Tööd ei tohi pooleli jääda, see oleks sama hea kui jõudeolek.
Kui antud on arvud `m`, `n` ja `t` (täisarvud lõigust 0 - 10000), leida mitu ülesannet on nende reeglite alusel võimalik maksimaalselt lõpetada.

NÄIDE 1:

```
3 5 54
```

Vastus: 18 (kõik kolmesed ülesanded)

NÄIDE 2:

```
3 5 55
```

Vastus: 17 (kaks viiest ja 15 kolmest ülesannet).

NÄIDE 3:

```
886 340 6177
```

Vastus: 10

5.11.3 Maksmise võimalused

Soomes ei ole ühe- ja kahesendised euromündid üldiselt kasutusel. Ignoreerides ühe- ja kahesendiseid (kõiki teisi europangatähti ja münte võib kasutada, hulk ei ole piiratud), leida, mitu võimalust on konkreetse summa maksmiseks. Maksimaalne uuritav summa on 500 eurot. Sisendiks on summa sentides.

NÄIDE 1:

`20`

Vastus: 4 (1x20, 2x10, 1x10 + 2x5 ning 4x5 senti)

NÄIDE 2:

`79`

Vastus: 24

5.11.4 Statistika manipuleerimine

Hoolimata sellest, et statistika on täppisteadus ega saa kuidagi „valetada“, nimetatakse statistikat siiski sageli valelikuks. Tegelikult on valelikud muidugi hoopis need, kes statistikat väärkasutavad, valides välja ainult osad andmed, võrreldes omavahel asjaolusid, mis on tegelikult erinevad, või visualiseerides andmeid eksitavalt.

Vaatame käesolevas ülesandes üht sellist statistika manipuleerimise viisi. Oletame, et keegi tahab näidata, et kallite autodega juhtub vähem õnnetusi. Selleks oleks efektne meetod luua nimekiri automarkidest, mis on üheaegselt hinna poolest kasvavas, aga õnnetuste arvu poolest kahanevas järjekorras.

Antud on hulk automarke koos nende hindade (tuhandetes eurodes) ja õnnetuste arvuga (miljoni auto kohta).

Ülesandeks on leida nende markide seast maksimaalse pikkusega jada, kus automargid on hinna poolest kasvavas, aga õnnetuste poolest kahanevas järjekorras. Jada väljastada esialgse jada järjekorranumbritena. Kõik võrratused on ranged, s.t hinnad peavad olema rangelt kasvavas ja õnnetused rangelt kahanevas järjekorras.

Sisendi esimesel real on automarkide arv `N` (`1 ≤ N ≤ 1000`). Järgmisel `N` real on täisarvupaarid, kus esimene arv tähistab hinda ja teine õnnetuste arvu `A` (`1 ≤ A ≤ 10000`).

NÄIDE:

```
9
13 6008
21 6000
20 500
40 1000
30 1100
20 6000
14 8000
12 6000
19 2000
```



Vastus:

```
7 2 5 4
```

PS! Kui sul tegelikult kästakse kunagi statistikat võltsida, siis intellektuaalselt aus oleks töökohta vahetada ☺

5.11.5 Lennukikütus

Reisilennukid püüavad teatavasti oma kütusekulu võimalikult hästi optimeerida. Oluliseks faktoriks on siin tuule kiirus. Ilmateatest on teada, milline tuul eri paikades ja kõrgustel puhub, ülesandeks on koostada optimaalne lennutrajektoor.
Iga 10 lennukilomeetri jaoks on kolm võimalust: hoida olemasolevat kõrgust (kulub 30 liitrit kütust), tõusta kilomeetri võrra (kulub 60 liitrit kütust) ja laskuda kilomeetri võrra (kulub 20 liitrit kütust).
Tuule kiirus erinevatel kõrgustel on antud 10 km/h täpsusega ja tuule kiirus võib olla -100 km/h (vastutuul) kuni 100 km/h (pärituul). Iga 10 km/h tuult suurendab või vähendab kütusekulu ühe liitri võrra. Tuult arvestatakse algkõrgusest lähtuvalt.
Lubatud on lennukõrgused 1 km kuni 9 km.
Sisendi esimesel real on antud soovitud lennukaugus L (kümnega jaguv positiivne täisarv kuni 10000). Järgmisel 10 real on igaühel $\frac{L}{10}$ täisarvu, mis tähistavad tuulekiirusi vastaval kõrgusel ja teelõigul (kõrgemad enne). Esimene rida tähistab tuult kõrgusel 9 ja alumine rida tuult kõrgusel 0. Reisi algus ja lõpp on mõlemad kõrgusel 0, mis tähendab, et alguses ja lõpus tuleb kindlasti tõusta ja laskuda.

NÄIDE 1:

```
40
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 90 90 10
10 -90 -90 10
```

Vastus: 120 (Alustame +10 pärituulega, tõuseme +90 tuulega kõrgusele, hoiame kõrgust ja siis laskume).

NÄIDE 2:

```
100
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
70 70 70 70 70 70 70 70 70 70
-50 -50 -50 -50 -50 -50 -50 -50 -50 -50
-70 -30 -70 -70 -70 -70 -70 -70 -70 -70
-90 -90 -90 -90 -90 -90 -90 -90 -90 -90
```

Vastus: 354

5.11.6 Kahjuritõrje

Farmer Fredil on ristkülikukujuline põld, mis on jagatud 10x10m ruutudeks. Ruute on kokku `MxN`. Nüüd on tal vaja põldu taimekaitsevahenditega pritsida. Pritsimine toob nii kasu (hävitades kahjureid) kui kahju (suurendades saagi kemikaalisisaldust). Iga ruudu kohta on teada, kui palju kasu või kahju selle ruudu pritsimine kokku annab.
Fred saab tellida lennuki, mis pritsib üle parajasti mingi ühe konkreetse ristkülikukujulise osa põllust (võib ka terve). Leida, kui palju kasu on võimalik lennuki tellimisest saada. Formaalsemalt väljendudes: leida maksimaalse summaga alamristkülik.
Sisendi esimesel real on täisarvud `M` ja `N` (`1 ≤ M, N ≤ 100`). Järgmistel `M` real on igaühel `N` arvu, mis tähistavad pritsimisväärtusi (täisarvud -127 kuni 127).

NÄIDE 1:

```
4 4
0 -2 -7 0
9 2 -6 2
-4 1 -4 1
-1 8 0 -2
```

Vastus:

```
15
```

(maksimaalse summaarse väärtusega alamristkülik asub vasakus alumises nurgas:

```
9 2
-4 1
-1 8
```

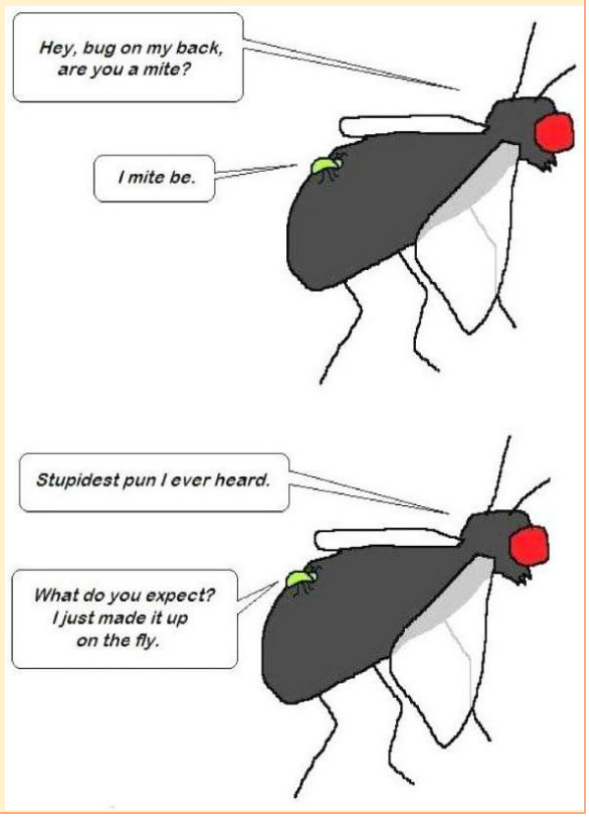
ja selle summa on 15)

NÄIDE 2:

```
5 5
125 85 -120 -114 63
-110 -77 49 27 47
13 -10 101 -116 -34
60 111 -1 61 -24
-120 110 80 69 107
```

Vastus:

```
513
```



5.11.7 Torni ehitamine

Väike Jaak ehitab klotsidest torni. Kõik klotsid on risttahukakujulised. Kuna Jaak on alles noor, suudab ta laduda klotse ainult nii, et asetab väiksema klotsi suurema peale. Täpsemalt tähendab see, et pealmise klotsi horisontaalsed mõõtmed (pikkus ja laius) peavad olema rangelt väiksemad kui alumise klotsi pikkus ja laius.
Samas võib klotse igat pidi pöörata, nii et nende pikkus, laius ja kõrgus on omavahel vahetatavad.
Jaagu vanem vend Ants soovib nooremat venda aidata ja annab talle kätte klotse sellises järjekorras ja niipidi pööratult, et Jaak saaks kokku võimalikult kõrge torni.
Kui vendadel on kokku `N` sorti klotse ja igat klotsi on võimalik kasutada piiramatul hulgal, leida, kui kõrge torni nad kokku saaksid.
Sisendis on antud arv `N` ning seejärel iga `N` klotsi kohta tema dimensioonid.

NÄIDE 1:

```
1
30 20 10
```

Vastus: 40 (kõigepealt klots lapiti, kõrgus 10, siis teine klots püsti esimese otsa, kõrgus 30, kokku 40)

NÄIDE 2:

```
2
6 8 10
5 5 5
```

Vastus: 21 (kõrgused 6, 10 ja 5)

NÄIDE 3:

```
7
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
6 6 6
7 7 7
```

Vastus: 28 (suuremast väiksemani üksteise otsas)



NÄIDE 4:

```
5
31 41 59
26 53 58
97 93 23
84 62 64
33 83 27
```

Vastus: 342

5.11.8 Putin sukeldumas

Vene Föderatsiooni president Vladimir Vladimirovitš Putin on teatavasti tuntud oma füüsiliste vägitükkide poolest, olgu siis tegu ratsutamise, lendamise või võitlemisega. Muuhulgas on ta paistnud silma oma sukeldumisoskusega, tuues Musta mere põhjast ära antiikseid vaase.
Vähem on aga teada, et sukeldumisele eelnes põhjalik planeerimine, kui palju ja millist varandust Putin ära saaks tuua.
Sul on unikaalne võimalus olla V.V. Putini sukeldumisülem ja planeerida võimalikult tulus sukeldumine. On teada, et merre on eelnevalt paigutatud `N` vaasi, sügavustele vastavalt $s_1,\ldots,s_N$ meetrit ning väärtustega $v_1,\ldots,v_N$. Putinil on hapnikuballoon, millest piisab, et olla vee all `t` sekundit. Vaasid tuleb ära tuua ükshaaval. Meres laskumine võtab aega `w` sekundit meetri kohta ning tõusmine `2*w` sekundit meetri kohta.
Sul tuleb leida parim strateegia, millises järjekorras peaks Putin vaase ära tooma, et nende koguväärtus oleks maksimaalne.
Sisendi esimesel real on arvud `t`, `w` ja `N`. Järgmisel `N` real on vaaside sügavused ning väärtused.
Väljastada iga vaasi sügavus ja väärtus, mille Putin saab ära tuua, kasutades parimat võimalikku strateegiat. `1 ≤ t ≤ 1000`, `N ≤ 30`.

NÄIDE 1:
```
210 4 3
10 5
10 1
7 2
```
Vastus:
```
10 5
7 2
```
NÄIDE 2:
```
656 2 5
79 74
97 75
37 81
21 99
38 25
```
Vastus:
```
37 81
21 99
38 25
```



5.11.9 Arvude liitmine

Antud on positiivsed täisarvud `K` ja `N` (`1 ≤ K, N ≤ 100`).
Mitu võimalust on arvu `N` esitamiseks K mittenegatiivse täisarvu summana? Sisendis on antud arvud `K` ja `N`, väljastada üks arv vastusena.

NÄIDE 1:
```
2 5
```
Vastus: 6 (0 + 5, 1 + 4, 2 + 3, 3 + 2, 4 + 1, 5 + 0)

NÄIDE 2:
```
3 4
```
Vastus: 15

NÄIDE 3:
```
100 3
```
Vastus: 5151

5.11.10 Templisambad

Antiikse templi restaureerimiseks on antud N marmorist silindrikujulist kivi, mis võivad olla erineva kõrgusega. Silindritest tuleb laduda K millimeetri kõrgune sammas. Kui täpselt K kõrgust sammast pole võimalik ehitada, tuleb mõni kivi madalamaks lihvida, mis tähendab, et kallist materjali läheb raisku. Eesmärgiks on minimeerida esiteks kaotsimineva materjali hulka ning teiseks vajaminevate kivide arvu. Sisendi esimesel real on arv K, teisel real arv N ($N \leq 100$). Järgmisel N real on kivide kõrgused (täisarvud kuni 2000). Väljastada vajaminevate kivide arv ja nende kõrgus enne madalamaks lihvimist.

NÄIDE 1:

```
1400
3
500 1000 2000
```

Vastus:

```
2 1500
```

NÄIDE 2:

```
2508
5
331 1585 1395 278 537
```

Vastus:

```
4 2541
```



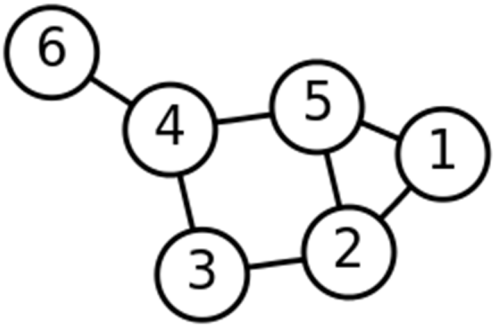
(Pildil on Apolloni templi varemed Delfis).

5.12 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk5 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
Fib.cpp, Fib.java, Fib.py	Fibonacci arvu leidmine rekursiooniga
Fib2.cpp, Fib2.java, Fib2.py	Fibonacci arvu leidmine mäluga rekursiooniga
Fib3.cpp, Fib3.java, Fib3.py	Fibonacci arvu leidmine alt-üles DP-ga
Fib4.cpp, Fib4.java, Fib4.py	Fibonacci arvu leidmine kokkuhoidliku DP-ga
Myndid1.cpp, Myndid1.java, Myndid1.py	Mündiülesande lahendus jõumeetodil
Myndid2.cpp, Myndid2.java, Myndid2.py	Mündiülesande lahendus DP-ga
Myndid3.cpp, Myndid3.java, Myndid3.py	Mündiülesande lahendus koos tagurdusmeetodiga (tagastab müntide väärtused)
LIS.cpp, LIS.java, LIS.py	Pikima kasvava osajada pikkuse leidmine jõumeetodil
LIS2.cpp, LIS2.java, LIS2.py	Pikima kasvava osajada pikkuse leidmine DP-ga
LIS3.cpp, LIS3.java, LIS3.py	Pikima kasvava osajada leidmine DP-ga
LCS.cpp, LCS.java, LCS.py	Pikima ühise osajada pikkuse leidmine DP-ga
Ristsumma.cpp, Ristsumma.java, Ristsumma.py	Ristsumma ülesande DP lahendus
ristsumma.xlsx	Ristsumma Excelis
Mortimer.cpp, Mortimer.java, Mortimer.py	Miljonäri ja vaeslaste ülesande lahendus
Vennad.cpp, Vennad.java, Vennad.py	Õiglase jagamise ülesande lahendus
Sallid.cpp, Sallid.java, Sallid.py	Moekunstniku ja koiliblika ülesande lahendus

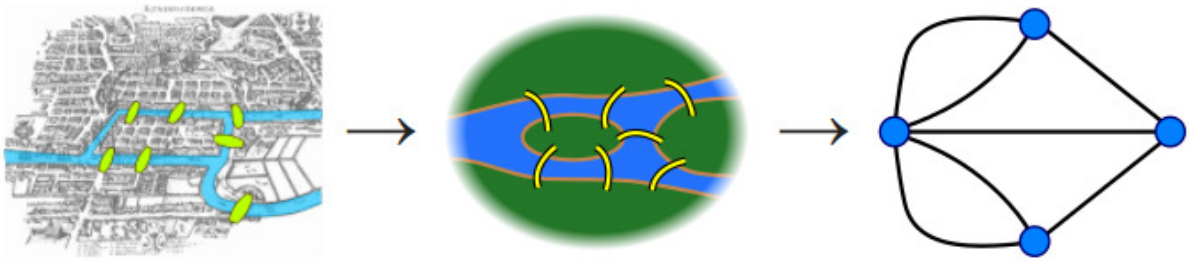
6 SISSEJUHATUS GRAAFITEOORIASSE



Graaf on struktuur, mida kasutatakse objektide vaheliste paarikaupa seoste kujutamiseks. Neid objekte nimetatakse graafi **tippudeks** (*vertex,* joonisel numbritega) ning seoseid graafi **servadeks** (*edge*, joonisel jooned tippude vahel). **Graafiteooria** on matemaatika haru, mis uurib graafe.

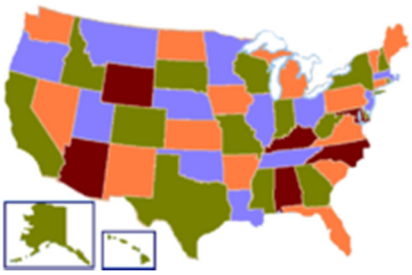
Graafe kasutatakse väga erinevate suhete ja protsesside modelleerimiseks, seda nii füüsilistes, bioloogilistes, sotsiaalsetes kui ka infosüsteemides. Näiteks võib graafina kujutada veebilehti ja nendevahelisi linke, sotsiaalseid „kes keda tunneb“ võrgustikke, aatomite vahelisi seoseid molekulis ja veel paljusid teisi struktuure.

Maailma esimeseks graafiteooria alaseks tööks peetakse 1736. aastal Leonhard Euleri avaldatud „Königsbergi sildade“ probleemi: Königsbergi linnas on seitse silda - kas on võimalik koostada jalutuskäiguteekond, mis ületab iga silla täpselt ühe korra?



Eitava vastuseni jõudmine pole raske, kuid Euleril tuli nullist alates luua ülesande lahendamiseks vajalik tehnika ja terminoloogia. Ülaltoodud joonis kujutab ka vastavat mõtlemisprotsessi: veekogude ja kallaste sisemised eripärad ei ole tegelikult tähtsad ja neid on võimalik abstraheerida sellisele tasemele, et alles jäävad vaid objektid ja seosed (tipud ja servad). Selline abstraktsioon ongi graafiteoreetilise mõtlemise tugevus - leitud tulemusi ja algoritme saab kasutada kõigis rakendustes veevärgist sotsiaalmeediani ja maakaartidest tuumauuringuteni.

Nagu diskreetse matemaatika puhul üldiselt, andis ka graafiteooriale hoo sisse arvutite areng ja antud juhul aitas see isegi täiesti otseselt uusi matemaatilisi resultaate saavutada.



Üks graafiteooria tuntumaid probleeme on **neljavärviprobleem**, mis küsib, kas mistahes tasapinnalise kaardi värvimiseks piisab neljast erinevast värvist, nii et iga riigi külg puutuks kokku vaid temast erinevat värvi naabriga. Tõestamaks, et selline värvimine on tõesti võimalik, leiti kõigepealt, et kõik erinevad värvimisvõimalused on taandatavad 1936 erinevale riikide asetuse kombinatsioonile. Nende kombinatsioonide läbivaatamiseks kasutasid Kenneth Appel ja Wolfgang Haken 1976. aastal arvuti abi – see oli ka esimene kord, kus olulise matemaatilise tulemuseni jõudmiseks arvutit kasutati.

6.1 GRAAFITEOORIA TERMINID

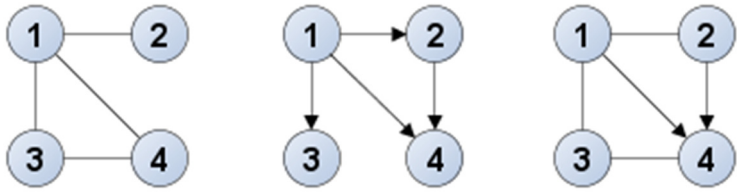
Matemaatiliselt tähistatakse graafi $G = (T, S)$, kus T on mittetühi tippude hulk ja S on paaride (x, y) hulk, kus $x, y \in T$.

Kui graafi servade hulk on tühi, nimetatakse sellist graafi **tühigraafiks**, kui serv on iga tipupaari vahel, nimetatakse sellist graafi **täielikuks**.

6.1.1 Suunatud ja suunamata servad

Graafi tippe x ja y ühendaval serv võib olla **suunatud** või **suunamata**. Kui suunda pole, siis see tähendab, et mööda serva saab liikuda nii tipust x tippu y kui ka vastupidi, sel juhul nimetatakse sellist serva suunamata servaks või lihtsalt servaks ja tähistatakse $s = \{x, y\}$. Kui aga serval on suund määratud, siis nimetatakse sellist serva suunatud servaks ehk **kaareks** ning tähistatakse $s = (x, y)$. Mööda sellist kaart saab liikuda tipust x tippu y, aga mitte vastupidi. Joonistel tähistatakse suunamata servi joontena ning kaari nooltena.

Kui graafi kõik servad on suunamata, siis sellist graafi nimetatakse **suunamata graafiks** *(undirected graph)* ehk lihtsalt graafiks. Graafi, milles kõik servad on suunatud, nimetatakse **suunatud graafiks** *(directed graph)*. Kui graafil on nii suunatud kui ka suunamata servi, siis on tegemist **segagraafiga**.

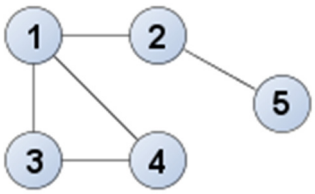


Esimene graaf joonisel on suunamata graaf, teine suunatud graaf ja kolmas segagraaf.

6.1.2 Teed graafis

Marsruudiks ehk **teeks** (*path*) graafis $G = (T, S)$ nimetatakse servade jada $s_1 = \{t_0, t_1\}$, $s_2 = \{t_1, t_2\}, \ldots, s_n = \{t_{n-1}, t_n\}$, milles iga serva lõpptipp on talle järgneva serva algustipp.

Ahel (*chain*) on selline marsruut, mille kõik servad on erinevad. Ahelat, mis ei läbi ühtki tippu rohkem kui üks kord, nimetatakse **lihtahelaks** (*simple chain*)**.** Järgneval joonisel moodustavad näiteks servad {1, 3}, {3, 4}, {4, 1}, {1, 2}, {2, 5} ahela ja servad {3, 4}, {4, 1}, {1, 2} ja {2, 5} lihtahela:



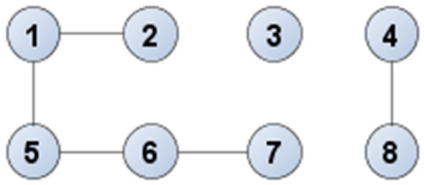
Tsükkel (*cycle*) on selline ahel, mille algus- ja lõpptipp on samad. Kui tsükli moodustab lihtahel, siis sellist tsüklit nimetatakse **lihttsükliks**. Ühest servast koosnevat tsüklit $s = \{t, t\}$ nimetatakse **silmuseks** (*loop*). Eelmisel joonisel on tsükkel näiteks {1, 3}, {3, 4} ja {4, 1}.

6.1.3 Sidususkomponendid

Graafi $G = (T,\ S)$ **alamgraafiks** (*subgraph*) nimetatakse mistahes graafi $G' = (T',\ S')$, mille korral kehtib $T' \subseteq T$ ja $S' \subseteq S$. See tähendab, et alamgraaf on saadav ülemgraafist mingi hulga tippude ja servade eemaldamise teel. Asjaolu, et G' on G alamgraaf, tähistatakse $G' \subseteq G$.

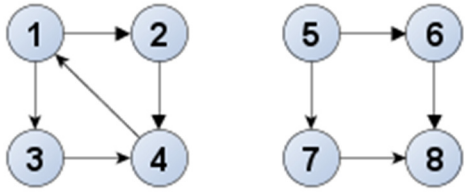
Öeldakse, et suunamata graaf on **sidus** (*connected*), kui iga tema tipupaari korral leidub marsruut ühest tipust teise. Seni toodud joonistel on olnud sidusad graafid.

Graafi G alamgraafi G' nimetatakse graafi G **sidususkomponendiks** (*connected component*), kui G' on sidus ja graafis G ei leidu sidusat alamgraafi G'', nii, et $G' \subset G''$. See tähendab, et sidususkomponent on maksimaalne sidus alamgraaf, seda ei saa kasvatada talle uute servade või tippude lisamisega ilma sidusust rikkumata.



Sellel graafil on kolm sidususkomponenti. Esimese moodustavad tipud 1, 2, 5, 6 ja 7 ning nendevahelised servad, teise tipp 3 ja kolmanda tipud 4 ja 8 koos nendevahelise servaga.

Sidusust ja sidususkomponente saab määrata ka suunatud graafidel. Suunatud graafi G nimetatakse sidusaks, kui kaarte asendamisel suunamata servadega saadud suunamata graaf G' on sidus. Kui suunatud graafis G leidub iga tipupaari korral marsruut esimesest tipust teise ja vastupidi, nimetatakse graafi G **tugevalt sidusaks** (*strongly connected*). Suunatud graafis saab eristada sidususkomponente ja **tugevalt sidusaid komponente**.



Joonisel on suunatud graaf, millel on kaks sidususkomponenti: esimene tippudega 1,2 3 ja 4 ning teine tippudega 5, 6, 7 ja 8 koos nendevaheliste kaartega. Vasakpoolne sidususkomponent on ka tugevalt sidus, parempoolne aga mitte.

6.2 GRAAFIDE ESITUSVIISID

Graafiteooriat õpetades on väga mugav graafe visuaalselt kujutada, programmi kirjutades on aga sellest vähe kasu. Kuidas siis graafe kujutada? Enim levinud on kolm põhimõttelist erinevat viisi, mida omakorda saab erinevaid andmestruktuure kasutades realiseerida. See, millist esitust valida, sõltub peamiselt sellest, mida graafiga edasi vaja teha on.

6.2.1 Naabrusmaatriks

Naabrusmaatriks (*adjacency matrix*) on kõige universaalsem ja lihtsam esitusviis. Naabrusmaatriks on maatriks, milles on üks rida ja üks veerg iga graafi tipu kohta. Tipule t_1 vastava rea ja tipule t_2 vastava veeru ristumiskohal on info kaare (t_1, t_2) kohta.

Naabrusmaatriksit on mugav hoida *n* x *n* massiivis M, kus n on graafi tippude arv. Minimaalne variant on see, kui tegemist on kaalumata graafiga – sellisel juhul piisab, kui massiivis hoida tõeväärtusi (või arve 0 ja 1), vastavalt sellele, kas antud kaar on olemas või mitte. Selle peatüki esimesel pildil olev graaf naabrusmaatriksina:

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Kaalutud graafi korral võib maatriksis hoida serva kaalu, sellisel juhul on oluline valida serva puudumise märkimiseks sobiv arv, mis ei läheks sassi serva kaaluga. Kui serva kohta on vaja hoida rohkem informatsiooni, siis võib massiivis hoida ka keerulisemaid kirjeid või kasutada 3-mõõtmelist massiivi.

Täielikult suunamata graafi korral on naabrusmaatriks peadiagonaali suhtes sümmeetriline ning sellisel juhul piisab, kui täita vaid pool tabelit.

Naabrusmaatriksi eelised:

- Servade lisamine ja eemaldamine on lihtne ja kiire.
- Kontroll, kas mingi tipupaari vahel on serv, on lihtne ja kiire.

Naabrusmaatriksi puudused:

- Kui tippe on palju, aga servi vähe, siis on selline esitus küllaltki mälu raiskav. Kui tippe on juba tuhandeid, on tõenäoliselt kasulikum mõnd muud esitusviisi kasutada.
- Tipu kõigi naabrite leidmiseks – protseduur, mida tuleb graafialgoritmides sageli ette – on naabrusmaatriksis vaja läbi käia terve maatriksi rida. Operatsiooni keerukus on $O(n)$, kus n on tippude arv graafis.

6.2.2 Tippude loend

Tippude loendis (*adjacency list*) on iga tipu kohta loend, kuhu sellest tipust serv läheb:

1: 2, 5
2: 1, 3, 5
3: 2, 4
4: 3, 5, 6
5: 1, 2, 4
6: 4

Tipust väljuvate kaarte loendeid on mugav hoida kas lihtahelas või dünaamilises massiivis, kuna nende pikkused on muutlikud. Loendeid endid võib hoida samuti ahelas või tavalises massiivis

pikkusega n, kus n on tippude arv graafis. Kui on vaja säilitada lisainformatsiooni tipu kohta, võib kasutada 2-mõõtmelist massiivi:

	1	2	3	4	5	6
Tipu nimi	karu	rebane	hiir	elevant	siga	kass
Naabrid	[2, 5]	[1, 3, 5]	[2, 4]	[3, 5, 6]	[1, 2, 4]	[4]

Kui on vaja hoida infot kaarte kohta, võib naabrite ahelas/massiivis kasutada keerulisemaid kirjeid, näiteks kaalutud graafis võib kasutada paare siht-tipu indeksist ja serva kaalust.

Tippude loendi eelised:

- vajadusel on tippude kohta mugav säilitada lisainfot,
- tipu naabrite leidmine ja töötlemine on kiire, see on vajalik paljudes graafitöötlusalgoritmides.

6.2.3 Servade loend

Servade loend (*edge list*) on loend graafi servadest. Siin on joonisel 1 toodud graafi esitus loeteluna servadest:

{1, 2}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {4, 5}, {4, 6}

Üks võimalus on selle realiseerimiseks kasutada 2-mõõtmelist massiivi, eriti juhul, kui servade arv on ette teada:

Esimene tipp	1	1	2	2	3	4	4
Teine tipp	2	5	3	5	4	5	6
Kaal	76	87	73	62	14	36	43

Sageli on aga mugavam säilitada serv eraldi struktuurina (näiteks kolmikuna) ning graaf on siis servade massiiv või ahel.

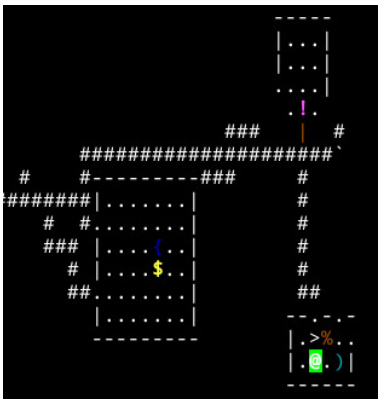
Servade loendi eelised:

- Mugav kasutada, kui servade kohta on palju lisainformatsiooni.
- Lihtsam kasutada, kui operatsioone on vaja sooritada just servadel. Näiteks on algoritme, kus on kasulik servi kaalu järgi jooksvalt sorteerida, seda on servade loendi abil mugav teha.

Servade loendi puudused:

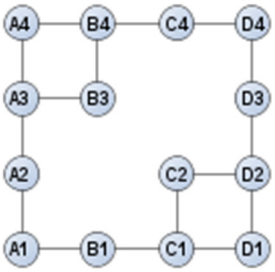
- Kui graafis on suunatud servi, tuleb kõik servad topelt salvestada.
- Tipu naabrite leidmiseks tuleb läbi käia kõik servad. Selle puuduse tõttu kasutatakse kõnealust esitlusviisi siiski harva.

6.2.4 Regulaarsete servadega graafid



Mõnel juhul on lihtne määrata, kas mingi kahe tipu vahel on serv või mitte. Nimetame siin selliseid graafe regulaarsete servadega graafideks. Üks lihtsamaid ja levinumaid regulaarsete servadega graafe on kahemõõtmelised kaardid:

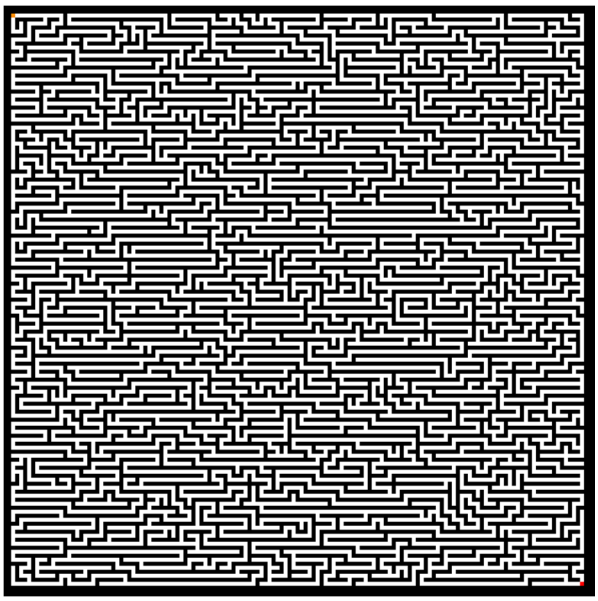
```
....
..#.
.#..
....
```



Kõikvõimalikke kaarte, labürinte jms saab sel moel graafina esitada. Iga märk selles ruudustikus on graafi tipuks ja servad on harilikult kõrvuti paiknevate märkide vahel. Kuna servad on nii regulaarsed, siis pole vaja luua eraldi andmestruktuuri nende hoidmiseks, piisab, kui hoida tippe kahemõõtmelises massiivis. Iga tipu naabertippude indeksid on kergesti arvutatavad:

(i-1, j-1)	(i-1, j)	(i-1, j+1)
(i, j-1)	(i, j)	(i, j+1)
(i+1, j-1)	(i+1, j)	(i+1, j+1)

6.3 GRAAFI LÄBIMINE



Graafi algoritmide juures on enamasti kõige olulisemaks protseduuriks graafi tippude töötlemine liikudes mööda servi tipust tippu. Sellisel moel graafi läbivaatamist nimetataksegi graafi läbimiseks. Graafi läbimiseks on kaks põhilist viisi: sügavuti läbimine ja laiuti läbimine.

6.3.1 Sügavuti läbimine

Sügavuti läbimise korral alustatakse graafi läbimist mingist tipust A ja liigutakse edasi selle uuele naabrile, siis omakorda järgmisele veel külastamata naabrile jne. Kui tipul on mitu naabrit, valitakse üks ja liigutakse mööda seda haru edasi. Kui kogu haru on läbitud, võetakse järgmine naaber ja vaadeldakse kogu see haru ja jätkatakse nii, kuni kõik harud on läbi vaadatud.

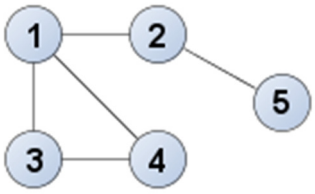
Kõige lihtsam on seda teha rekursiivselt. Lisainfona on vaja meeles pidada, millised tipud on juba külastatud, et vältida lõputusse tsüklisse minekut.

```
vector<int> graaf[tippude_arv]; // graaf tippude loendina
bool kaidud[tippude_arv] = { 0 }; // alguses kõik tipud on käimata

void otsi(int tipp)
{
    kaidud[tipp] = true; // siin me oleme
    tootle(tipp); // teeme midagi selle tipuga
    for (int i = 0; i < graaf[tipp].size(); i++) {
        int naaber = graaf[tipp][i]; // iga naabertipuga
        if (kaidud[naaber] == false) // kus ei ole veel käinud
            otsi(naaber); // kordame sama protseduuri
    }
}
```

Olenevalt sellest, kus asub töötlemise protseduur, eristatakse ees- ja lõppjärjestuses graafi töötlemist. Ülaltoodud näites töödeldakse tipp enne naabrite kallale asumist ja seega on tegu eesjärjestuses läbimisega. Kui töötlemine toimub pärast naabreid, on tegemist lõppjärjestuses läbimisega.

See, millises järjekorras tipud täpselt läbitakse, sõltub nii alguskohast kui sellest, kuidas naabreid valitakse (naabrite järjestusest). Selles mõttes ei ole graafil üht kindlat sügavuti läbimisel saadavat järjestust, isegi juhul, kui alustada läbimist samast tipust. Näiteks graafil:



on neli võimalikku tippude eesjärjestuses töötlemise järjestust sügavuti läbimisel, kui alustada tipust 1: need on (läbitavate tippude järjekorras) 1, 2, 5, 4, 3; 1, 2, 5, 3, 4; 1, 4, 3, 2, 5 ja 1, 3, 4, 2, 5. Tipust 5 alustades aga on võimalikke eesjärjestuses sügavuti läbimise järjestusi vaid 2: 5, 2, 1, 3, 4 ja 5, 2, 1, 4 3.

Sügavuti läbimise keerukus sõltub valitud andmestruktuurist. Oletades, et töötlemisoperatsioon on konstantse keerukusega, siis juhul, kui kasutatakse naabrusmaatriksit, tuleb kogu maatriks läbi käia ja keerukuseks on $O(T^2)$, kus T on tippude arv. Kui valitud on tippude loend, siis on keerukuseks $O(T + S)$, kus S on servade arv.

Sügavuti läbimist on hea kasutada ka tsüklite otsimiseks. Kui mõni vaadeldava tipu naabritest on juba vaadeldud, on graafis tsükkel:

```
bool on_tsykkel(int tipp, int eelmine)
{
    kaidud[tipp] = true;
    for (int i = 0; i < graaf[tipp].size(); i++) {
        int naaber = graaf[tipp][i];
        if (kaidud[naaber] == false) {
            if (on_tsykkel(naaber, tipp))
                return true;
        }
        else if (naaber != eelmine)
           return true;
    }
    return false;
}
```

6.3.2 Ülesanne: Ratsu teekond

Graafi läbimise illustreerimiseks kasutatakse sageli ülesandeid malelaual:



Leia selline ratsu teekond malelaual, mille puhul ratsu külastaks igat ruutu täpselt ühe korra. Antud on ratsu algpositsioon malelaual, väljastada ratsu teekond.
Sisendi esimesel real on üks number - ruudukujulise malelaua pikkus ruutudes. Teisel real on antud ratsu algpositsioon, mis koosneb veergu tähistavast tähemärgist ning numbrist, mis tähistab rida.
Väljastada ratsu teekond kasutades sama notatsiooni. Kui teed ei leidu, väljastada `EI SAA`.

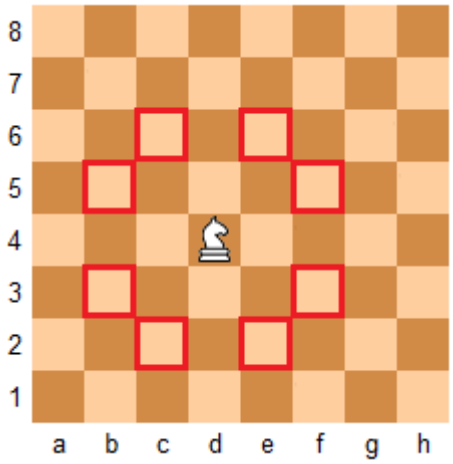
NÄIDE 1:

5
c3
Vastus:
`a2 c1 e2 d4 b5 a3 b1 d2 e4 c5 a4 b2 d1 e3 d5 b4 d3 e5 c4 a5 b3 a1 c2 e1`

NÄIDE 2:

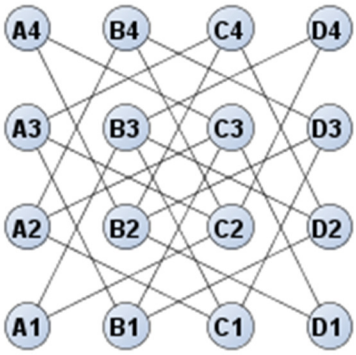
4
a1
Vastus:
`EI SAA`

Ratsu saab liikuda malelaual ratsukäikudega, st liikuda ühes suunas korraga kaks ja ristuvas suunas ühe ruudu. Malelaua keskel saab ratsu liikuda igast ruudust ühe käiguga kaheksasse teise ruutu:



Ratsu käigud. Punase raamiga on tähistatud ruudud, millele ratsu saab käia ühe käiguga.

Ratsu käike malelaual saab kujutada graafina, mille tippudeks on malelaua ruudud ning servadeks ratsu käigud. Kahe tipu vahel on serv parajasti siis, kui ratsu saab käia ühe käiguga ühele tipule vastavalt ruudult teisele tipule vastavale ruudule. Tippe on sellises graafis sama palju kui malelaual ruute.



Ratsu käigud 4X4 laual

Selles ülesandes võib graafi luua näiteks järgmiselt:

```
// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
vector<int> graaf[64];

void loo_graaf(int dim)
{
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            for (int k = 0; k < 8; k++){
                int naaber_x = i + deltad[k][0];
                int naaber_y = j + deltad[k][1];
                if ((naaber_x >= 0) && (naaber_x < dim) &&
                    (naaber_y >= 0) && (naaber_y < dim))
                        graaf[i*dim + j].push_back(dim*naaber_x + naaber_y);
            }
}
```

Kuna aga ratsu võimalikud käigud on lihtsasti arvutatavad, siis on tegemist regulaarsete servadega graafiga ning graafi luua ei ole ilmtingimata vajalik. Järgnevas lahenduses arvutatakse naabrid jooksvalt.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

const int MAX_DIM = 8;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = { 0, 0 };
stack<string> vastus;
bool Otsi(int x, int y, int kaigu_nr);

int main()
{
    cin >> dim;
    ruutude_arv = dim*dim;
    string algus;
    cin >> algus;
    if (Otsi(algus[0]-'a', algus[1]-'1', 1)) {
        while (vastus.size()) {
            cout << vastus.top() << " ";
            vastus.pop();
        }
    }
    else
        cout << "EI SAA" << endl;
}
```

```
bool Otsi(int x, int y, int kaigu_nr)
{
    if (kaigu_nr == ruutude_arv) return true; // kõik ruudud on käidud, tee on leitud
    kaidud[x][y] = true; // siin me oleme

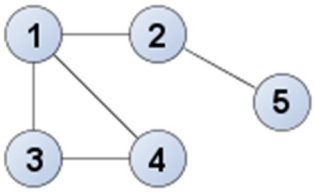
    for (int k = 0; k < 8; k++) { // iga võimaliku käigu jaoks
        int naaber_x = x + deltad[k][0];
        int naaber_y = y + deltad[k][1];
        if ((naaber_x < 0) || (naaber_x >= dim) || // kui on väljaspool..
            (naaber_y < 0) || (naaber_y >= dim) || //..mängulauda
            kaidud[naaber_x][naaber_y]) { // või on juba käidud,
            continue; // pole sobiv sihtkoht, jätkame
        }
        if (Otsi(naaber_x, naaber_y, kaigu_nr + 1)) { // Proovime järgmist käiku
            vastus.push(string() + (char)(naaber_x + 'a') + (char)(naaber_y + '1'));
            return true;
        }
    }
    // ei leidnud teed, mis läbiks kõik ruudud
    kaidud[x][y] = false; // võtame viimase käigu tagasi
    return false;
}
```

Selle ülesande näol on tegemist väga tuntud matemaatilise probleemiga, mille kohta võib rohkem lugeda siit: https://en.wikipedia.org/wiki/Knight's_tour

6.3.3 Laiuti läbimine

Laiuti läbimisel (*breadth-first search*, *BFS*) alustatakse ühest tipust A ja selle järel liigutakse järjekorras kõigile tipu A naabertippudele. Kui A kõik naabrid on töödeldud, liigutakse A ühe naabri kõikidele naabertippudele, seejärel teise naabri naabritele jne. See tähendab, et tipud lähevad töötlemisele nendeni jõudmise järjekorras. Seetõttu on algoritmiski hea kasutada tippude meelespidamiseks järjekorda. Järjekorra jaoks on muidugi vaja lisamälu, mis ongi üldjuhul kõige suuremaks miinuseks, võrreldes graafi sügavuti läbimisega.

Graafis



laiuti läbimisel alates tipust 1, töödeldakse järgmiseks tipud 2, 3 ja 4 (töötlemise järjekord oleneb sellest, kuidas meil servad järjestatud on) ning seejärel tipp 5. Esimesel ringil lisatakse järjekorda tipud, mis on lähtetipust ühe serva kaugusel, teisel need, mis on kahe serva kaugusel jne. Seetõttu laiuti läbimine sobib lühimate teede leidmiseks servade arvu mõttes (tee, mis läbib kõige vähem servi).

Lihtne laiuti läbimise algoritm on järgmine:

```
vector<int> graaf[tippude_arv];
bool lisatud[tippude_arv] = { 0 }; // alguses on kõik tipud lisamata

void labi_laiuti()
{
    queue<int> jarjekord;
    jarjekord.push(0); // lisame esimese tipu järjekorda
    lisatud[0] = true; // ja märgime, et on järjekorras

    while (!jarjekord.empty()) {
        int tipp = jarjekord.front();
        jarjekord.pop();
        tootle(tipp);
        for (int i = 0; i < graaf[tipp].size(); i++) {
            int naaber = graaf[tipp][i]; // iga naabertipu,
            if (!lisatud[naaber]) { // mis ei ole veel järjekorras olnud,
                jarjekord.push(naaber); // lisame järjekorda
                lisatud[naaber] = true;
            }
        }
    }
}
```

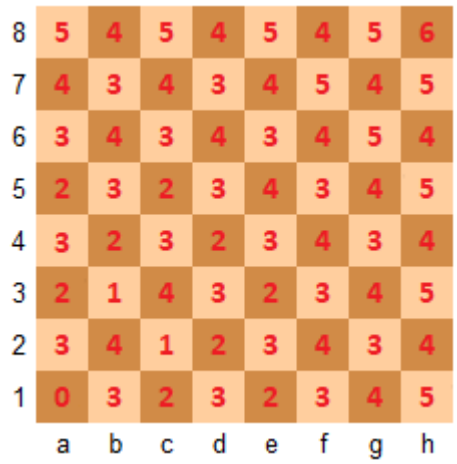
Vaatame järgmist ülesannet laiuti läbimise praktilisemast kasutusest:

6.3.4 Ülesanne: Ratsu teekond 2

Leida ratsu teekond ühelt malelaua ruudult teisele vähima võimaliku käikude arvuga. Sisendi esimesel real on üks number – malelaua ridade ja veergude arv, teisel real on kaks sõna: ratsu positsioon malelaual ja sihtkoht, väljastada üks võimalik ratsu teekond.
NÄIDE:
8
a1 h8
Vastus:
a1 c2 a3 c4 e5 g6 h8

Selle ülesande aluseks olev graaf on identne eelmise ratsu ülesandega ja seetõttu pikemat selgitust ei vaja. Kasutame siin taas selle graafi regulaarsust ning arvutame iga ruudu naabrid jooksvalt.

Vähima käikude arvuga tee leidmiseks saame kasutada laiuti läbimist. Alustades ratsu algkohast, asuvad kõik selle tipu naabrid algusest ühe käigu kaugusel. Naabrite naabrid on kahe käigu kaugusel jne:



Malelaua igas ruudus on kirjas, kui mitme käiguga ratsu sinna ruutu jõuab, kui ratsu alustab ruudult a1.

Seda, millise naabri kaudu ratsu kõige vähemate käikudega lõppu jõuab, ei ole ette teada. Selleks on kasulik meelde jätta kõige lühem tee iga ruudu jaoks, kuhu satume.

```
#include <iostream>
#include <string>
#include <stack>
#include <queue>
#include <map>
using namespace std;

const int MAX_DIM = 100;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = {0, 0};
void otsi(pair<int, int> algus, pair<int, int> lopp);

int main()
{
    cin >> dim;
    string salgus, slopp;
    cin >> salgus >> slopp;
    pair<int,int> algus = make_pair(salgus[0] - 'a', salgus[1] - '1');
    pair<int, int> lopp = make_pair(slopp[0] - 'a', slopp[1] - '1');
    otsi(algus, lopp);
}
```

```
void otsi(pair<int, int> algus, pair<int, int> lopp)
{
    queue<pair<int, int>> jarjekord;
    // Siin hoiame iga ruudu jaoks meeles lühima tee, kuidas sinna algusest sai.
    map<pair<int, int>, string> teed;

    jarjekord.push(algus);
    kaidud[algus.first][algus.second] = true;
    teed[algus] = string() + char(algus.first + 'a') + char(algus.second + '1');

    while (!jarjekord.empty()) {
        pair<int, int> ruut = jarjekord.front();
        jarjekord.pop();
        if (lopp == ruut) { // oleme jõudnud sihtkohta
            cout << teed[ruut] << endl; // väljastame tee
            return; // rohkem otsida pole vaja
        }

        for (int k = 0; k < 8; k++) { // iga võimaliku käigu jaoks
            int x = ruut.first + deltad[k][0];
            int y = ruut.second + deltad[k][1];
            if ((x >= 0) && (x < dim) && (y >= 0) && (y < dim) && //mängulaual
                !kaidud[x][y]) { // ja käimata
                jarjekord.push(make_pair(x, y));
                kaidud[x][y] = true;
               // Laiuti läbimise puhul kohe kui uuele ruudule jõuame, on see lühim
               // võimalik tee servade arvu mõttes. Jätame selle tee meelde
                teed[make_pair(x, y)] =
                    teed[ruut] + " " + char(x + 'a') + char(y + '1');
            }
        }
    }
   cout << "EI SAA" << endl; // ei ole teed leidnud
}
```

6.4 SIDUSUSKOMPONENTIDE LEIDMINE

Kui nüüd vaadata hoolega eelnevaid graafi läbimise algoritme, siis need läbivad terve graafi vaid siis, kui see on sidus. Õnneks on neis kasutuses massiiv, mis peab meeles, millised tipud on töödeldud. Seega saab pärast iga otsingut käia selle massiivi läbi ja alustada uut läbimist järgmisest veel vaatlemata tipust. Selline lähenemine võimaldab ka loendada ja leida graafi sidususkomponendid. Järgmises ülesandes on justnimelt seda vaja teha.

6.4.1 Ülesanne: Ratsud

Normaalselt saab ratsu liikuda kõigi malelaua ruutude vahel. Aga praegu on seal ka hulk ettureid – nendele ruutudele ratsu käia ei saa. Leida, mitu ratsut saab malelauale paigutada nii, et nad ei saa üksteise ruutudele käia (ükskõik, mitme käiguga). Sisendi esimesel real on antud kaks arvu: `N` – ruudukujulise malelaua pikkus ruutudes, ja `E` – etturite arv . Teisel real on `E` sõna – etturite asukohad. Väljastada üks arv – ratsude arv, mida saab malelauale paigutada.

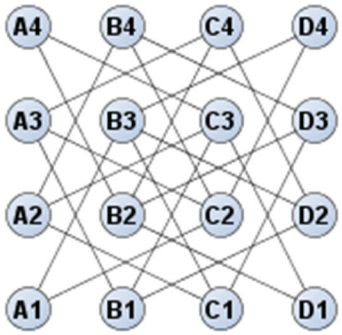
NÄIDE:

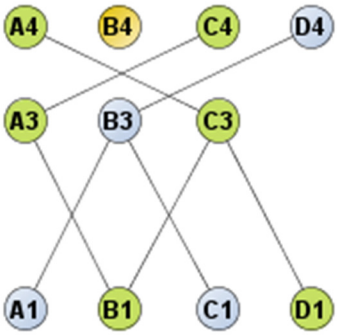
```
4 5
a2 b2 c2 d2 d3
```

Vastus:

```
3
```

Vaatame jällegi väiksemat, 4x4 lauda. Kui kõik ruudud on vabad, siis on tekkiv ratsu tee graaf sidus ning malelauale saab paigutada vaid ühe ratsu. Kui aga lauale on paigutatud mõned etturid, siis nendesse ruutudesse ratsu enam käia ei saa ning need võib graafist eemaldada.





Vasakpoolne graaf vastab etturiteta lauale, parempoolne aga olukorrale, kus etturid on asetatud ruutudesse A2, B2, C2, D2 ja D3. Jooniselt on hea näha, et nii tekib kolm sidususkomponenti (joonisel eri värvi tippudega) ja seega saab asetada malelauale kolm ratsut nii, et nende teed ei kattuks.

Nii on selleski ülesandes esiteks vaja tekitada korrektne graaf. Et mitte kaotada servade regulaarsust, on mugav tähistada hõivatud ruudud – kui käik viib ratsu asukoharuudult hõivatud ruudule, siis nende ruutude vahel serva ei eksisteeri. Kuna nagunii kontrollime, kas tipp on varem külastamata, siis võib antud ülesandes lihtsalt etturite asukohad märkida külastatuteks.

```
#include <iostream>
#include <string>
using namespace std;

const int MAX_DIM = 100;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
// leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = { 0, 0 };
void otsi(int x, int y);

int main()
{
    int etturid, ratsud=0;
    cin >> dim >> etturid;
    for (int i = 0; i < etturid; i++) {
        string ettur;
        cin >> ettur;
        kaidud[ettur[0] - 'a'][ettur[1] - '1'] = true;
    }

    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++) {
            if (!kaidud[i][j]){
                ratsud++;
                otsi(i, j);
            }
        }
    cout << ratsud << endl;
    return 0;
}
```

```
void otsi(int x, int y)
{
    kaidud[x][y] = true;

    for (int k = 0; k < 8; k++) {
        int naaber_x = x + deltad[k][0];
        int naaber_y = y + deltad[k][1];
        if ((naaber_x >= 0) && (naaber_x < dim) && // mängulaual
            (naaber_y >= 0) && (naaber_y < dim) &&
            !kaidud[naaber_x][naaber_y]) { // ja käimata
                otsi(naaber_x, naaber_y);
        }
    }
}
```

6.5 ÜLEUJUTAMINE



Veel üheks küllalki sagedaseks graafi läbimise ülesandeks on nn üleujutamise (*flood fill*) ülesanded. Oma olemuselt on need väga sarnased sidususkomponentide leidmisele. Samas on tegemist küllaltki sageli esineva ülesandetüübiga, seega vaatame siin üht sellist lähemalt:

6.5.1 Ülesanne: Veekogud

On antud kahevärviline kaart, kus sinine tähistab vett ja roheline metsa. Leia, mitu veekogu on kaardil ja kui suur on neist suurim. Erinevateks loetakse veekogud siis, kui nende sinised pikslid ei puutu kuidagi kokku, ka nurkapidi mitte.
Sisendi esimesel real on kaks arvu m ja n ($1 \le m,\ n \le 100$), mis näitavad kaardi piksliridade ja -veergude arvu. Igal järgneval m real on n-täheline sõna, mis koosneb S ja R tähtedest. S tähistab sinist ja R rohelist pikslit. Väljastada kaks rida, millest esimesel on eraldatud veekogude arv ja teisel suurima veekogu pikslite arv.
NÄIDE:

```
5 5
RRRRR
RSSRS
RSRRS
SSSRS
SSRRR
```

Vastus:

```
2
8
```

Selleski ülesandes on tegemist regulaarsete servadega graafiga ning võib graafi hoidmiseks kasutada vahetult kahemõõtmelist märkide massiivi. Tuleb meeles pidada, et serv eksisteerib ainult sama tähemärgiga tippude vahel.

Peamiseks eesmärgiks ülesandes on leida sidususkomponent ning sageli ka selle suurus (mitut tippu sidususkomponent sisaldab). Selleks, et eristada juba vaadeldud ja vaatlemata tippe, on lisamassiivi

asemel kavalam tipud nii-öelda „üle värvida“ – siis on teada, milliseid tippe on juba arvestatud. Ühe komponendi leidmise ja üle värvimise võib lahendada järgmiselt:

```
int deltad[8][2] =
    {{1, 0}, {1, 1}, {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -1}, {1, -1}};
int ridu, veerge;
char** kaart;

int taida(int r, int v, char v1, char v2) {
    if (r < 0 || r >= ridu || v < 0 || v >= veerge)  // Kui on väljaspool kaarti
        return 0;
    if (kaart[r][v] != v1) return 0; // kui ei ole esimest värvi
    int vastus = 1; // lisab vastusele ühe
    kaart[r][v] = v2; // värvime teist värvi, et ei läheks tsüklisse
    for (int d = 0; d < 8; d++) {
        vastus += taida(r + deltad[d][0], v + deltad[d][1], v1, v2);
    }
    return vastus;
}
```

Kogu lahenduse juures peame jälle kogu graafi läbi käima ja leidma võimalikud komponentide alguskohad:

```
int main()
{
    cin >> ridu >> veerge;
    kaart = new char*[ridu];
    for (int i = 0; i < ridu; i++) {
        kaart[i] = new char[veerge];
        for (int j = 0; j < veerge; j++) {
            cin >> kaart[i][j];
        }
    }

    int veekogusid = 0;
    int suurim = 0;

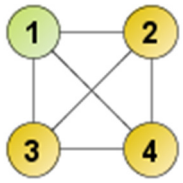
    for (int i = 0; i < ridu; i++) {
        for (int j = 0; j < veerge; j++) {
            if (kaart[i][j] == 'S') {
                veekogusid++;
                suurim = max(suurim, taida(i, j, 'S', '*'));
            }
        }
    }
    cout << veekogusid << endl;
    cout << suurim << endl;
    return 0;
}
```

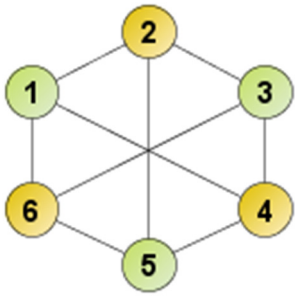
6.6 KAHEALUSELINE GRAAF

Kahealuseliseks (*bipartite*) graafiks nimetatakse sellist graafi, mille tipud jagunevad mingi tunnuse alusel kahte hulka ning servad saavad olla ainult erinevatesse hulkadesse kuuluvate tippude vahel.

Kõige lihtsam on ette kujutada graafi, mille tipud on värvitud kahe erineva värviga, näiteks on iga tipp värvitud kas punaseks või siniseks. Selline graaf on kahealuseline parajasti siis, kui kõik selle graafi servad on kahe erinevat värvi tipu vahel.

Graafiülesannetes tõstatub vahel küsimus, kas antud graaf saab olla kahealuseline? Oletame näiteks, et tantsuõpetaja soovib moodustada ühtlasi ringikujulisi formatsioone n lapsest nii, et iga poisi kõrval seisab tüdruk ja iga poisi vastas seisab tüdruk ning vastupidi – iga tüdruku kõrval ja vastas on poiss. Näiteks nelja last nii paigutada ei saa, kuus last aga küll:





Seda tüüpi ülesandeid võib esineda ka programmeerimisvõistlustel. Kui taibata ära, milline graaf tekib, ja see realiseerida, siis tippude „värvimine“ on taas lihtne graafi läbimise protseduur. Ka siin pole tegelikult oluline, kas läbida graafi laiuti või sügavuti – õige vastuse annavad mõlemad. Laiuti läbimine annab aga sageli kiiremini vastuolu.

Siin on funktsioon, mis etteantud graafi tipud kaheks jaotab kasutades graafi laiuti läbimist:

```
vector<int> graaf[tippude_arv];
vector<int> jaotus(tippude_arv, -1);

bool jaota()
{
    queue<int> jrkd;
    jrkd.push(0);
    jaotus[0] = 0;

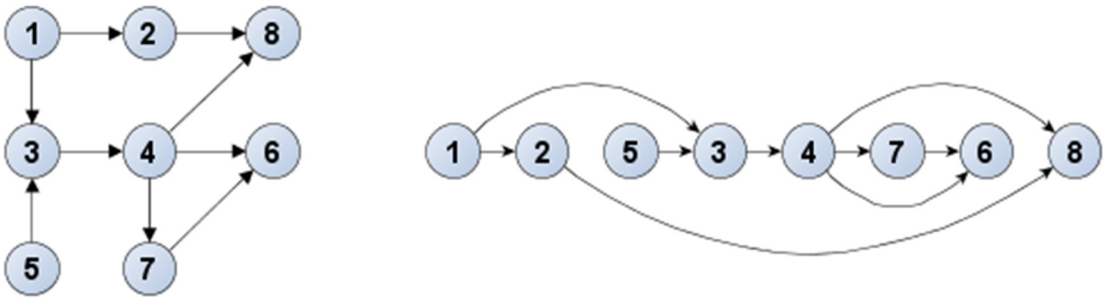
    while (!jrkd.empty()) {
        int u = jrkd.front();
        jrkd.pop();
        for (int i = 0; i < (int)graaf[u].size(); i++) {
            int v = graaf[u][i];
            if (jaotus[v] == -1) {// kui naaber ei ole veel jaotatud
                jaotus[v] = 1 - jaotus[u]; // paneme talle teise väärtuse
                jrkd.push(v); // ja lisame järjekorda
            }
            else if (jaotus[v] == jaotus[u]) { // kui serv on kahe sama omadusega tipu
                return false; // vahel, siis järelikult graaf ei ole kahealuseline
            }
       // juhul, kui naaber on juba jaotatud, aga teist värvi, on kõik korras, jätkame
        }
    }

    return true;
}
```

6.7 TOPOLOOGILINE SORTEERIMINE

Üheks huvitavaks graafi erijuhuks on suunatud tsükliteta graaf, millega tutvusime põgusalt eelmises peatükis. Sellises graafis ei saa liikuda tagasi tipule, mis on juba läbitud. Joonisel saab esitada sellise graafi lineaarse tippude reana, nii et servad on suunatud kõik ühes suunas (vasakult paremale). Sellist tippude järjestust nimetatakse **topoloogiliseks järjestuseks**.

Ühel graafil võib olla mitu topoloogilist järjestust, ainsaks tingimuseks on, et iga serva (t_i, t_j) korral $i < j$, kus i, j on tipu indeksid topoloogilises järjestuses.



Suunatud graaf ja selle topoloogiliselt sorteeritud esitus

Järgmiseks vaatame üht lihtsat topoloogilise sorteerimise ülesannet. Puhtalt topoloogilise sorteerimise ülesandeid esineb pigem lihtsamatel võistlustel, kuid sageli on topoloogiline sorteerimine mõne keerukama ülesande alamosaks.

6.7.1 Ülesanne: Loomad

Antud on hulk loomade paare, kus esimene loom on teisest tugevam. Sorteerida kõik loomad tugevuse järjekorda.
Sisendi esimesel real on loomapaaride arv `N`. Järgneval `N` real on igaühel kaks looma, esimene tugevam kui teine. Väljastada tugevuse järjekorras sorteeritud loomade nimekiri kõige tugevamast alates.
NÄIDE:

```
4
rebane jänes
karu hunt
hunt rebane
jänes hiir
```

Vastus:

```
karu hunt rebane jänes hiir
```

Taas tuleb alustada sellest, et milline graaf antud ülesandes tekib. Sisendina on antud loomapaarid, mille kohta on teada, et esimene on tugevam kui teine. Loomad on graafi tippudeks ja seos „on tugevam“ servaks – sisendi näol on meil tegemist juba servade loendiga ja nii ei ole graafi moodustamine keeruline. Kuna siin iseloomustab tippu looma nimi, siis tavalise massiivi asemel on graafi mugavam hoida andmestruktuuris, mis võimaldab kiiret ligipääsu nime järgi, näiteks `map`'is

```
#include <iostream>
#include <list>
#include <queue>
#include <map>
#include <string>

using namespace std;

typedef struct Loom
{
    string Nimi;
    map<string, Loom*> Sisse;
    map<string, Loom*> Valja;
} Loom;
```

```
map<string, Loom> graaf;

Loom* leia_loom(string nimi)
{
    if (!graaf.count(nimi)) { // kui selle nimega looma veel ei ole
        Loom n;
        n.Nimi = nimi;
        graaf[nimi] = n; // lisame selle
    }
    return &graaf[nimi];
}

int main()
{
    list<string> valjund;
    queue<Loom*> jrkd;
    int servi;
    cin >> servi;
    for (int i = 0; i < servi; i++) {
        string yks, kaks;
        cin >> yks >> kaks;
        Loom* esimene = leia_loom(yks);
        Loom* teine = leia_loom(kaks);
        esimene->Valja[teine->Nimi] = teine;
        teine->Sisse[esimene->Nimi] = esimene;
    }
    // kõigepealt leiame kõik tipud, millel sissetulevaid kaari ei ole. Need loomad
    // saavad olla kõige tugevamad teadaoleva info põhjal.
    for (map<string, Loom>::iterator it = graaf.begin(); it != graaf.end(); ++it) {
        if (it->second.Sisse.empty())
            jrkd.push(&it->second);
    }

    while (!jrkd.empty()) { // kuni on veel töötlemata tippe
        Loom* n = jrkd.front();
        jrkd.pop();
        valjund.push_back(n->Nimi); // lisame vastusesse

        for (map<string, Loom*>::iterator it2 = n->Valja.begin();
                 it2 != n->Valja.end(); ++it2) {
            Loom* jrgm = it2->second;
            // eemaldame naabrilt kõik vaadeldavast tipust sissetulevad kaared
            jrgm->Sisse.erase(n->Nimi);
            if (jrgm->Sisse.empty()) // kui naabril rohkem sissetulevaid kaari pole
                jrkd.push(jrgm); // lisame selle järjekorda
        }
        n->Valja.clear(); // eemaldame tipu kõik väljuvad kaared
    }

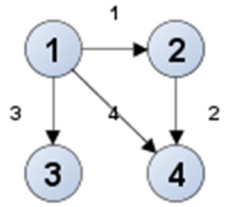
    for (map<string, Loom>::iterator it = graaf.begin(); it != graaf.end(); ++it) {
        if (it->second.Sisse.empty()) continue;
        cout << "EI SAA" << endl;
        return 0;
    }

    for (list<string>::iterator it = valjund.begin(); it != valjund.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    return 0;
}
```

Topoloogilist sorteerimist nõuavad harilikult igasugu planeerimisülesanded, kus objektidel on omavahel range järjestus – näiteks tuleb mingid tööd sooritada enne teisi.

6.8 KAALUTUD SERVADEGA GRAAFID

Programmeerimise seisukohalt on oluline eristada ka kaalutud graafe. Kaalutud graafis on igal serval kaal ehk väärtus. Oluline erinevus on näiteks lühima tee leidmisel graafis – kui kõik servad on võrdsed, taandub lühima tee leidmine vähima servade arvuga tee leidmisele, kaalutud graafis tuleb kasutada keerulisemaid algoritme.



Sellel graafil lühim tee tipust 1 tippu 4 läbi tipu 2, kuna 1+2 < 4.

6.9 DIJKSTRA LÜHIMA TEE LEIDMISE ALGORITM

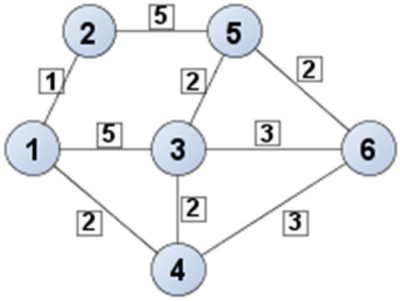
1956. aastal leiutas hollandi arvutiteadlane Edsger Dijkstra (1930-2002) efektiivse algoritmi, kuidas leida lühim tee graafi ühest tipust teise.

Dijkstra puhul tekitab tema tööst arusaamisest suurema hulgal segadust see, kuidas tema perekonnanime hääldama peaks. Nime esimeses silbis on diftong, mille esimene pool on a, e ja ä vahepealne ning teine pool on i. Kuula näidet siit: https://upload.wikimedia.org/wikipedia/commons/8/85/Dijkstra.ogg

Dijkstra algoritmi võib vaadelda kui graafi laiuti läbimise edasiarendust. Laiuti läbimise puhul on meil järjekord vaadeldavatest tippudest, kus kõigepealt tulevad tipud, mis on algtipust kaugusel 1, siis kaugusel 2 jne. Kui kõigi servade kaal on 1, siis on Dijkstra algoritm ekvivalentne tavalise laiuti läbimisega. Kui servadel võivad olla ka teiste väärtustega kaalud, muutub algoritm järgmiseks:

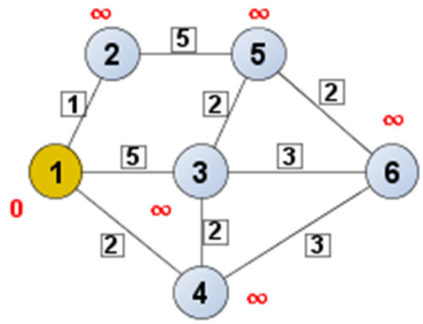
1. Sea algtipu kauguseks 0 ja kõigi teiste tippude kauguseks lõpmatus.
2. Märgi algtipp **aktiivseks** ja kõik ülejäänud tipud **vaatamata** tippudeks.
3. Olgu aktiivne tipp A. Vaata kõiki tema naabreid B ja:
    a. Arvuta B-sse jõudmise kaugus A kaudu (kaugus tippu A pluss A ja B vahelise serva kaal).
    b. Kui see kaugus on väiksem kui seni leitud vähim kaugus tippu B, siis paranda tipu B kaugus selleks väiksemaks väärtuseks.
4. Märgi tipp A **vaadatuks.** (Siia pole vaja enam tagasi tulla!)
5. Vali vaatamata tippude seast **vähima senise kaugusega** tipp uueks aktiivseks tipuks.
    a. Kui kõigi vaatamata tippude kaugus on lõpmatus, pole nõutud teed võimalik leida.
    b. Kui uus aktiivne tipp on otsitav sihtkoht, siis on lühim tee leitud – see on võrdne parima seni leitud kaugusega.
    c. Vastasel korral jätka uuesti alates sammust 3.

Vaatame, kuidas leida järgneval graafil lühim tee tipust 1 tippu 6:

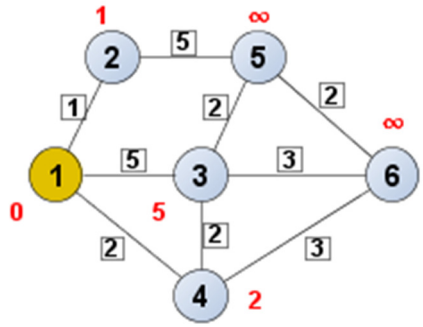




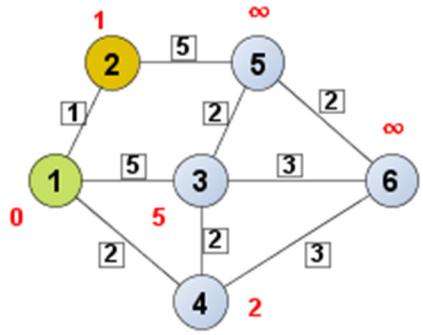
Kõigepealt märgime algtipu kauguseks 0 ja teiste tippude kauguseks lõpmatuse (joonisel punasega) ning seame tipu 1 aktiivseks (joonisel kollane):



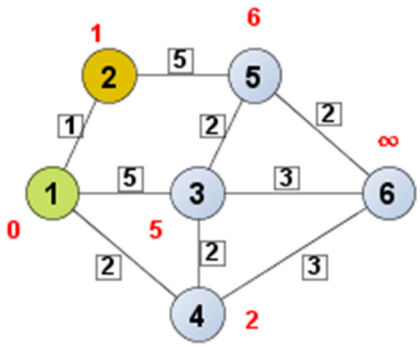
Leiame kõigi tipu 1 naabertippude kaugused algtipust: tipu 2 kaugus on 0 + 1 = 1, tipu 3 kauguseks saab 5 ja tipu 4 kauguseks 2:



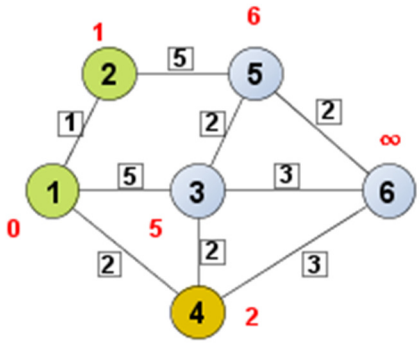
Nüüd on tipp 1 ja selle märgime vaadatuks (joonisel rohelisega), et seda uuesti mitte vaadelda. Valime mitteaktiivsetest vaatlemata tippudest kõige väiksema kaugusega tipu, milleks on tipp 2 kaugusega 1. Märgime selle aktiivseks:

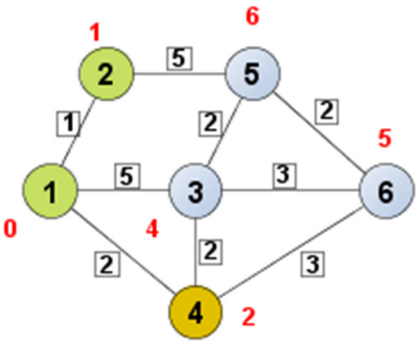


Vaatame tipu 2 naabreid: 1 on juba vaadatud ja sellega ei tee midagi. Leiame tipu 5 kauguse, milleks saab 1 + 5 = 6:

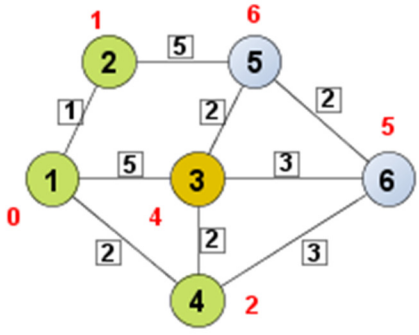


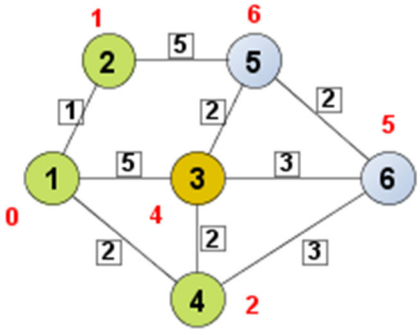
Nüüd on ka tipp 2 töödeldud ja valime uue vähima kaugusega tipu, milleks nüüd on tipp 4 ja arvutame selle kaudu kaugused. Kuigi tipul 3 on juba leitud kaugus, milleks on 5, on tipu 4 kaudu kaugus tippu 3 2 + 2 = 4. Parandame tipu 3 kauguse ning leiame ka lõpptipu 6 kauguse:



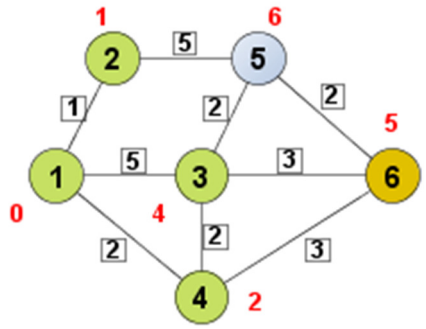


Kuigi oleme leidnud mingi kauguse lõpptippu, ei pruugi see olla veel lõplik vastus, seega jätkame valides uueks aktiivseks tipuks tipu 3 kaugusega 4. Siitkaudu kaugused siiski ei parane:





Nüüd on vähima kaugusega vaatlemata tipuks tipp 6 kaugusega 5. Märgime selle aktiivseks:



Kuna tegemist on lõpptipuga, siis oleme leidnud lühima teepikkuse tipust 1 tippu 6 ja selleks on 5. Tippu 5 meil enam vaadata ei ole vaja, sealtkaudu enam teepikkus lõpp-punkti parandada ei ole võimalik.

Dijkstra algoritmi kirjelduses on üks koht, mille realisatsioon pole täiesti ilmne: see on vähima kaugusega tipu leidmine. Üks võimalus on kõik vaatamata tipud ükshaaval läbi käia, mis võtab $O(T)$ aega, kus T on graafi tippude koguarv. Algoritmi kogukeerukuseks tuleb siis $O(S + T^2)$, kus S on graafi servade arv. Seda kasutas Dijkstra ka oma esialgses töös.

Parem võimalus on aga kasutada andmestruktuuri, mis võimaldab pidevalt vähima väärtusega elementi kiiresti kätte saada, näiteks peatükist 3.13 tuttaval kuhjal põhinevat eelistusjärjekorda. Selle abil saab senise vähima kauguse leida ajaga $log(T)$, nii et algoritmi kogukeerukuseks tuleb $O(S + T \cdot log(T))$.

Algoritmi näitlikustamiseks vaatame järgmist ülesannet:

6.9.1 Ülesanne: sõiduaeg

N asulat on omavahel ühendatud M teega, kuid mõned neist teedest on head kiirteed, teised aga peaaegu läbimatud kruusateed. Seetõttu on vahel otsetee asemel kiirem minna teiste asulate kaudu ringi. Leida kiireim tee asulast A asulasse B.
Sisendi esimesel real on asulate arv N, teede arv M, alguspunkti indeks A ja sihtkoha indeks B (0≤A,B<N). Igal järgneval M real on kolm arvu: ühendatud asulate indeksid ja esimesest asulast teise jõudmise aeg minutites.
Leida, mitu minutit kulub selleks, et jõuda asulast A asulasse B. Kui selline teekond on võimatu, väljastada -1.
NÄIDE:

```
6 9 1 6
1 2 1
1 3 5
1 4 2
2 5 5
3 5 2
3 6 3
3 4 2
4 6 3
5 6 2
```

Vastus:

```
5
```

See on klassikaline lühima tee pikkuse leidmise algoritm. Graafi tippudeks on asulad, servadeks teed nende vahel ning kaaludeks minutid, kui kaua tee läbimiseks kulub. Kuna kuluv aeg ei saa olla negatiivne, siis saab lahendamiseks kasutada eespool põhjalikult kirjeldatud Dijkstra algoritmi:

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;
#define INF INT_MAX // Lõpmatus

const int tippude_arv = 10001; // Suurim võimalik tippude arv.
vector<pair<int, int> > Naabrid[tippude_arv];
int Min_kaugused[tippude_arv];// Siin hoiame vähimat leitud kaugust algtipust tippu i.
bool Vaadatud[tippude_arv] = { 0 }; // Siin hoiame infot, kas tipp on juba töödeldud

int leia_teepikkus(int start, int siht, int n)
{
    for (int i = 0; i < tippude_arv; i++)
        Min_kaugused[i] = INF; // alguses oletame, et kõik tipud on lõpmata kaugel
    class greater { public: bool operator ()(pair<int, int>&p1, pair<int, int>&p2)
        { return p1.second>p2.second; } };
    priority_queue<pair<int, int>, vector<pair< int, int >>, greater > kaugused;
    // Lähtekoha kaugus iseendast on 0
    kaugused.push(make_pair(start, Min_kaugused[start] = 0));
    while (!kaugused.empty())
    {
        pair<int, int> lahim_tipp = kaugused.top();
        kaugused.pop();
        int tipp = lahim_tipp.first, vahim_kaugus = lahim_tipp.second;
        if(tipp == siht) { // otsitud tee on leitud
            return vahim_kaugus; //tagastame tee pikkuse
        }
        if (Vaadatud[tipp]) // tipp vaadatud, võtame järgmise
            continue;
        Vaadatud[tipp] = true; // Siin me oleme
        for (int i = 0; i < Naabrid[tipp].size(); i++) // Iga naabri jaoks
            if (!Vaadatud[Naabrid[tipp][i].first] && // kui naaber on veel käimata
                Naabrid[tipp][i].second + vahim_kaugus <
                Min_kaugused[Naabrid[tipp][i].first])
                kaugused.push(make_pair(Naabrid[tipp][i].first,
                 (Min_kaugused[Naabrid[tipp][i].first] =
                     Naabrid[tipp][i].second + vahim_kaugus)));
    }
    return -1; //ei leidnud teed
}

int main()
{
    int n, m, a, b, x, y, t;
    cin >> n >> m >> a >> b;

    for (int i = 0; i < m; i++) { // Koostame graafi
        cin >> x >> y >> t;
        Naabrid[x].push_back(make_pair(y, t));
        Naabrid[y].push_back(make_pair(x, t));
    }

    std::cout << leia_teepikkus(a, b, n);
    return 0;
}
```

6.10 PUUD

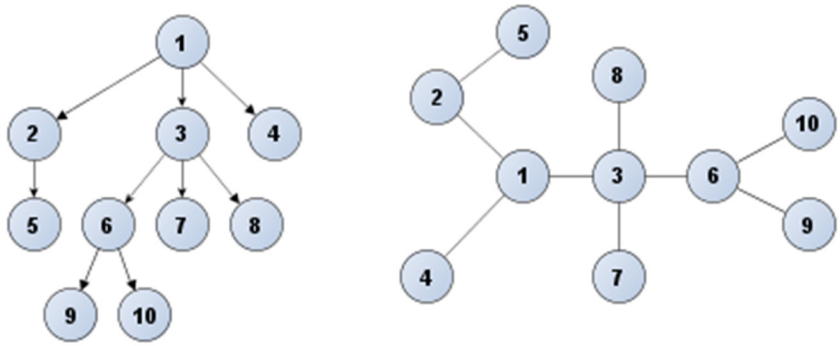
6.10.1 Puud kui graafid

Kolmandas peatükis alapeatükis „3.10 Kahendpuu“ oli juttu puudest andmestruktuurina, lähemalt just kahendpuust ja selle esitusest.

Kahendpuud (eriti kompaktset kahendpuud) on lihtne hoida massiivis, aga kui tegu on üldistatud puuga, mille tippudel võib olla kuitahes palju alluvaid, on kasulikum mõelda sellest kui graafi erijuhust. Puu hoidmiseks on tavaliselt kõige loomulikum esitusviis tippude loend.

6.10.2 Puu definitsioon

Pilti vaadates on üsna selge, et tegu on puuga:



Vasakul ja paremal on esitatud tegelikult sama puu. Vasakpoolsel joonise on tipp 1 välja toodud **juurtipuna** ja servad on suunatud juurest lehtedeni, see aitab algoritmi sageli paremini korrastada, kuid see pole kohustuslik. Parempoolsel joonisel olev graaf on samuti puu, kuigi see pole otseselt nii joonistatud.

Formaalselt saab seda, et *n* tipuga graaf *G* on puu, sõnastada mitmel viisil:

- *G* on sidus ja tsükliteta;
- *G* on tsükliteta ja mistahes uue serva lisamine loob graafile tsükli;
- *G* on sidus, kuid mistahes serva eemaldamine muudab selle mittesidusaks;
- Suvaline tipupaar graafis *G* on ühendatav lihtahelaga täpselt ühel viisil;
- *G* on sidus ja selles on *n* – 1 serva;
- *G* on tsükliteta ja selles on *n* – 1 serva.

Ülesannete sõnastuses on sageli kasutatud üht neist omadustest selle asemel, et lihtsalt öelda „tegemist on puuga“. Näiteks võib ülesandes olla juttu teedevõrgust, kus „igast linnast igasse linna saab liikuda unikaalsel viisil“, mille all on tegelikult mõeldud lihtsalt seda, et teedevõrk moodustab puu.

6.10.3 Graafitöötlusalgoritmid puul

Kuna puu on graaf, saab sellel kasutada ka graafitöötlusalgoritme. Samas puu küllaltki range struktuur võimaldab kasutada lihtsamaid lahendusi.

Kuna T tipuga puus on $T-1$ serva, siis graafitöötlusalgorimid keerukusega $O(T+S)$ on puude puhul keerukusega $O(T)$. Realiseerides graafitöötlusalgoritme puudel tasub meeles pidada, et puudes ei esine tsükleid ja puu on sidus - seega ei ole põhjust puust tsükleid ega sidususkomponente

otsida. Samuti on puus iga kahe tipu vahel üheselt määratud tee, mis teeb lühima tee leidmise oluliselt lihtsamaks – triviaalseks puu läbimise ülesandeks. Samuti on puu kahealuseline graaf.

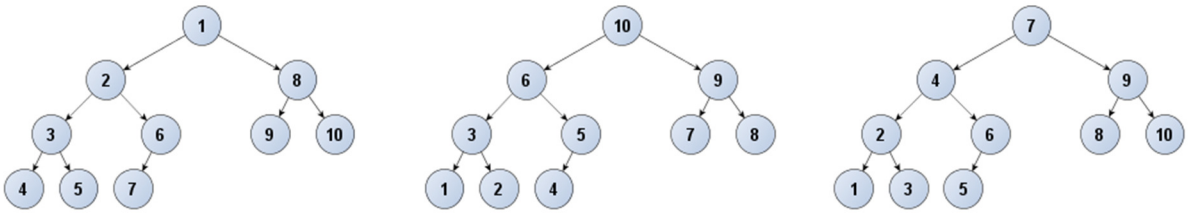
6.10.4 Kahendpuu sügavuti läbimine

Kui üldjuhul võib graafi läbimist alustada ükskõik millisest tipust, siis juurtega puudel on loomulik alguspunkt – puu juurtipp. Kahendpuus on määratud ka alluvate järjekord (eristatakse vasakut ja paremat alluvat), mistõttu saab eristada spetsiifilisemaid sügavuti läbimise võimalusi, kusjuures iga läbimise viis annab üheselt määratud tippude töötlemise järjekorra. Kahendpuu sügavuti läbimise viisid on järgmised:

Eesjärjestuses läbimise (*pre-order traversal*) korral töödeldakse kõigepealt tipp, seejärel selle vasak alampuu ja seejärel parem alampuu.

Lõppjärjestuses läbimise (*post-order traversal*) korral töödeldakse kõigepealt vasakpoolne alampuu, seejärel parempoolne alampuu ja alles siis tipp ise.

Keskjärjestuses (*in-order*) läbimise korral töödeldakse kõigepealt vasak alampuu, seejärel tipp ise ja siis parem alampuu.



Puu läbimine ees-, lõpp- ja keskjärjestuses. Numbrid tippudes tähistavad läbimise järjekorda.

Programmi kirjutamise seisukohalt ongi ainus erinevus selles, kus asub funktsioon tipu töötlemiseks:

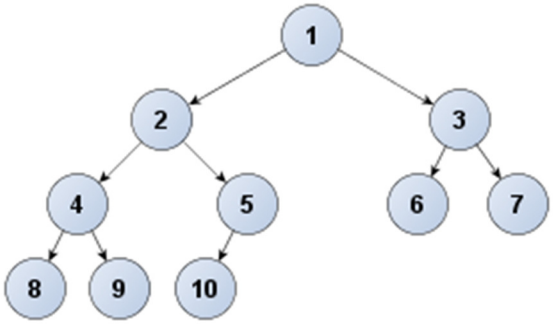
```
void labi_eesjarjestuses(int tipp)
{
    int vasak = puu[2*tipp + 1];
    int parem = vasak + 1;
    tootle(tipp);
    labi_eesjarjestuses(vasak);
    labi_eesjarjestuses(parem);
}

void labi_keskjarjestuses(int tipp)
{
    int vasak = puu[2 * tipp + 1];
    int parem = vasak + 1;
    labi_keskjarjestuses(vasak);
    tootle(tipp);
    labi_keskjarjestuses(parem);
}

void labi_loppjarjestuses(int tipp)
{
    int vasak = puu[2 * tipp + 1];
    int parem = vasak + 1;
    labi_loppjarjestuses(vasak);
    labi_loppjarjestuses(parem);
    tootle(tipp);
}
```

6.10.5 Puu laiuti läbimine

Ka puud saab läbida laiuti, mis tähendab tippude tasemete kaupa töötlemist – kõigepealt juurtipp, seejärel teise taseme tipud, siis kolmanda jne. Kõige viimasena töödeldakse puu lehed. Kui alluvad on järjestatud (nagu näiteks kahendpuul) on laiuti läbimise järjekord üheselt määratud:



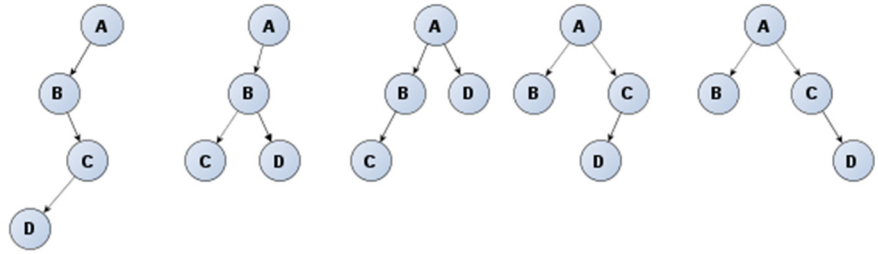
Kahendpuu tippude läbimise järjekord puu laiuti läbimisel

Kui kahendpuu on esitatud massiivina, nagu alampeatükis 3.10.1 kirjeldatud, siis sellise kahendpuu laiuti läbimine tähendab lihtsalt massiivi läbimist algusest lõpuni.

6.10.6 Ülesanne: Kahendpuud

Hannes magas hommikul sisse ja jõudis loengusse üsna lõpupoole. Loengu teemaks oli kahendpuude läbimine ja tahvlile oli joonistatud hulk erinevaid kahendpuid, mille tipud olid eristatud erinevate tähtedega. Kuna parajasti oli käsil puude erinevad läbimisviisid, siis Hannes ei viitsinud puid kohe üles joonistada, vaid kirjutas ruttu üles tekstistringi, mis saadi puud eesjärjestuses läbides. Õnneks Hannes taipas, et eesjärjestusest ilmselt ei piisa puude taastamiseks, ja lisas iga puu kohta ka selle keskjärjestuses läbides saadud järjestuse. Kodus asus ta puid taastama, kuid käsitsi oli see siiski tüütu ettevõtmine. Kuna kodutööks jäi nagunii leida nende puude tippude läbimise järjekord lõppjärjestuses, siis aita kirjutada Hannesel programm, mis leiab etteantud ees ja keskjärjestuse järgi antud puu läbimise lõppjärjestuses.
Sisendi esimesel real on puu tippude arv ja teisel real kaks stringi- antud puu tippude töötlemise järjekord eesjärjestuses ja keskjärjestuses. Väljastada üks string: antud puu tippude töötlemise järjekord lõppjärjestuses.
NÄIDE:
4
ABCD BACD
Vastus:
BDCA

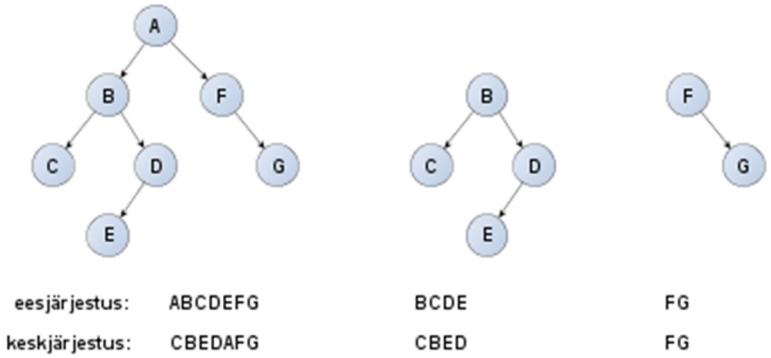
Tõesti, Hannesel on õigus. Kuigi konkreetse kahendpuu tippude töötlemise järjekord on puu eesjärjestuses (nagu ka kesk ja lõppjärjestuses) läbimisel üheselt määratud, ei kehti vastupidine: mitmel erineval kahendpuul võib olla sama tippude töötlemise järjekord.



Erinevaid kahendpuid, mille tippude töötlemise järjekord eesjärjestuses on ABCD

Keskjärjestus annab olulist lisainformatsiooni. Näiteks ülaltoodud graafide keskjärjestuses läbimise tulemused on BDCA, CBDA, CBAD, BADC, BACD, mis kõik on erinevad.

Eesjärjestuses töötlemisel töödeldakse esimesena alati puu juur, nii on juureks see tipp, mis esimeses stringis esimene. Juure leidmine on seega lihtne. Aga teise tipuga jääb juba hätta, kuna see võib olla nii parem kui ka vasak alluv. Keskjärjestuses läbimisel töödeldakse juurtipp vahetult pärast vasaku alampuu töötlemist ja enne parema alampuu töötlemist. Nii jagades keskjärjestuses saadud stringi juure kohalt kaheks, tekib kaks stringi: vasakpoolse alampuu keskjärjestuses läbimise järjekord ja parempoolse alampuu keskjärjestuses läbimise järjekord. Kuna ka eesjärjestuses läbitakse enne parema alampuu kallale asumist vasak, on meil olemas kogu info juure kummagi alampuu ees- ja keskjärjestuse kohta.



Vasakul on puu juurega A, keskel selle vasakpoolne alampuu ja paremal parempoolne alampuu

Nii võiks taastamise algoritm olla midagi sellist:

```
taasta_puu(eesjärjestus, keskjärjestus) {
    leia_alampuude_järjestused();
    taasta_puu(vasaku_alampuu_eesjärjestus, vasaku_alampuu_keskjärjestus);
    taasta_puu(parema_alampuu_eesjärjestus, parema_alampuu_keskjärjestus);
}
```

Jääb veel leida lõppjärjestus. Üheks võimaluseks on muidugi puu reaalselt tekitada ning seejärel läbida see lõppjärjestuses. Kindlasti ei ole see paha mõte, kuid saab veel kavalamini. `Taasta_puu` funktsiooni ülesehitus on sarnane kahendpuu sügavuti läbimise algoritmidele ning lõppjärjestuse saamiseks ei pea tegema midagi muud, kui funktsiooni lõpus parasjagu töötlemisel oleva tipu nimi vastusele lisada või miks mitte ka kohe väljastada.

Samuti võib loobuda stringide tegelikust tükeldamisest ja leida ainult alg- ja lõppindeksi, mille vahel olevale lõigule vastavat alampuud töötlema hakata. Kuna järjestused leitakse enne alampuude töötlema asumist ehk eesjärjestuses (kõigepealt leiame juurtipu, seejärel selle vasaku alampuu juure, mille järel omakorda selle vasaku alampuu juure jne), siis eesjärjestuse stringis piisab ainult jooksva indeksi meelespidamisest.

```cpp
#include<iostream>
#include<string>
using namespace std;

int ees_indeks;
string eesjarjestus, keskjarjestus;

void taasta_puu(int algus, int lopp) {
    if (algus>lopp) {
        return;
    }
    int i, kesk_indeks;

    char nimi = eesjarjestus[ees_indeks++];
    for (i = algus; i <= lopp; i++) {
        if (keskjarjestus[i] == nimi)
            kesk_indeks = i;
    }
    taasta_puu(algus, kesk_indeks - 1);
    taasta_puu(kesk_indeks + 1, lopp);
    cout << nimi;
    return;
}

int main() {

    int tippe;
    cin >> tippe;
    cin >> eesjarjestus >> keskjarjestus;
    ees_indeks = 0;
    taasta_puu(0, tippe - 1);
    cout << endl;

    return 0;
}
```

6.11 KONTROLLÜLESANDED

6.11.1 Robotivõistlus

Tänapäeval muutuvad järjest populaarsemaks mitmesugused robotivõistlused. Selles ülesandes on robotil vaja ristkülikukujulisel võistlusväljakul läbida hulk kontrollpunkte. Selleks on tal instruktsioon, mis koosneb märkidest 'V' (pööra 90 kraadi vasakule), 'P' (pööra 90 kraadi paremale) ja 'O' (liigu otse). Võistlusväljakul on ka takistused, millele robot sõita ei saa. Leida, mitu erinevat kontrollpunkti jõuab robot läbida.

Sisendi esimesel real on kolm täisarvu: `N` ja `M` (`1≤N, M≤100`) tähistavad väljaku mõõtmeid, `S` (`1≤S≤10 000`) tähistab instruktsiooni pikkust.

Järgmisel `N` real on igaühel `M` märgist koosnev string, kus:

- '`.`' tähistab tühja ruutu,
- '`*`' tähistab kontrollpunkti,
- '`#`' tähistab takistust,
- '`N`', '`S`', '`W`' või '`E`' tähistab roboti algset asukohta, kus konkreetne märk tähistab seda, millises suunas robot alguses liikuma hakkaks.

Viimasel real on `S` märgist koosne string, milles on ainult märgid 'V', 'P' ja 'O'.

Väljundisse kirjutada täpselt üks täisarv: mitu erinevat kontrollpunkti robot läbib.

NÄIDE 1:

```
3 3 2
***
*N*
***
PV
```

Vastus:

`0` (robot keerutab koha peal ega liigu kusagile)

NÄIDE 2:

```
4 4 5
...#
*#W.
*.*.
*.#.
OOVOO
```

Vastus:

`1`

NÄIDE 3:

```
10 10 20
....*.....
.......*..
.....*....
..*.#.....
...#N.*..*
...*......
..........
..........
..........
..........
OPOOOOOOVVOOOOOOVOPO
```

Vastus:

`3`

6.11.2 Civilization

Arvutimängus Civilization võistlevad erinevad riigid omavahel selle pärast, kes saab maailmas edukaimaks. Mängus koosneb maailma kaart hulgast kontinentidest ja saartest ning sageli juhtub, et erinevad riigid saavad erinevatel kontinentidel domineerivaks ja haaravad endale kõik selle kontinendi ressursid. Et seejärel erinevate riikide võimsust hinnata, on kasulik teada, kui suured erinevad kontinendid on. Ülesandeks on leida maailma kaardi põhjal suurima kontinendi pindala.
Sisendi esimesel real on kaks arvu `M` ja `N` (`1≤M, N≤1000`), mis tähistavad maailmakaardi suurust. Järgmisel `M` real on igaühel `N` märgist koosnev string, mis koosneb kas 'v' (vesi) või 'm' (maa) märkidest. Kaks märki on osa samast kontinendist, kui neil on ühine külg. Kaart moodustab „silindri", s.t läänepoolne ja idapoolne serv puutuvad tegelikult omavahel kokku. Väljundisse kirjutada üks täisarv: suurima kontinendi pindala.



NÄIDE:

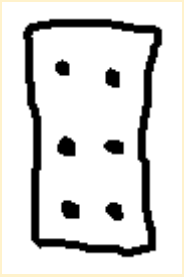
```
5 6
vvvvvv
vmmmvv
vvvvvv
mmvvmm
vvvvvv
```

Vastus:

4

(eelviimasel real on ainult üks kontinent, kuigi ta on kaardil kahes osas).

6.11.3 Doominod



Väike Martin joonistab graafikaprogrammis doominoklotse, näiteks selliseid nagu näha juuresoleval pildil.
On teada, et iga joonis on piiratud pideva joonega. Joone sees olevad silmad võivad olla ebakorrapärase kujuga, kuid on kõik ühes tükis ning ei puutu omavahel kokku.
Kirjutada programm, mis etteantud joonise põhjal leiab, mitu silma klotsil on.
Sisendi esimesel real on kaks täisarvu: `H (1≤H≤100)` ja `W (1≤W≤50)`. Järgmisel `H` real on igaühel string pikkusega `W`, mis koosneb märkidest `0-9` ja `A-F`. Neid märke tuleb tõlgendada 16-süsteemi arvudena ja teisendada bittideks (nt `A=1010`). Bitid väärtusega 1 tähistavad musti piksleid ja bitid väärtusega `0` tähistavad valgeid.
Väljundisse kirjutada täpselt üks arv: mitu silma on doominoklotsil.

NÄIDE:

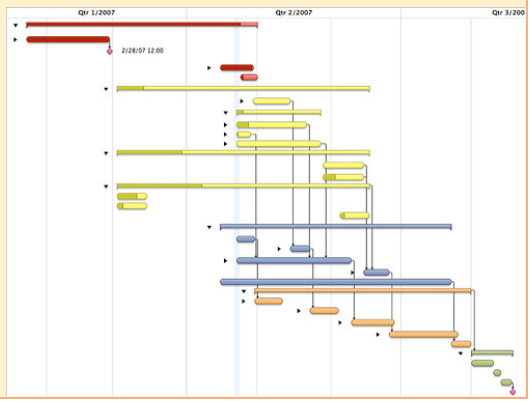
```
8 2
7C
86
92
8A
A2
AA
82
7E
```

Vastus:

3

6.11.4 Projektiplaan

Mitmesugused projektid, olgu nad siis ehituse, tarkvara või laulupeokorralduse valdkonnast, koosnevad paljudest tegevustest, mis tuleb ära teha teatud järjekorras. Mõnesid tegevusi saab teha paralleelselt, kuid sageli on need tegevused omavahelises sõltuvuses, s.t üks asi tuleb enne ära teha kui teise juurde saab asuda. Tavaline küsimus, mida suured ülemused projektijuhilt küsivad, on: „millal valmis saab?“ Eeldusel, et üksteisest mittesõltuvaid tegevusi võib teha paralleelselt, leida projekti minimaalne kestus.
Sisendi esimesel real on kaks täisarvu: tegevuste arv `N (1≤N≤1000)` ja sõltuvuste arv `M (1≤M≤10000)`
Teisel real on `N` positiivset täisarvu, mis tähistavad üksikutele tegevustele kuluvat aega.
Lõpuks tuleb `M` rida, igaühel kaks täisarvu, mis tähistavad tegevuste järjenumbreid: esimene tegevus peab lõppema enne, kui teine algab. Väljundisse kirjutada üks täisarv: projekti minimaalne kestus.



NÄIDE:

```
5 5
3 4 2 5 6
1 2
2 3
1 4
3 5
4 5
```

Vastus:

```
15
```

6.11.5 Sõnateisendused

Lapsed mängivad sõnamängu, kus ühest sõnast tuleb sammhaaval saada teine, kusjuures igal sammul tohib muuta ära ühe tähe. Näiteks võime teha teisenduse `kala→pala→palk→pauk→puuk`.
Kui antud on lubatud sõnade hulk, algne sõna ja lõpuks saadav sõna, siis leida, mitu sammu on vastavaks teisenduseks tarvis teha. On teada, et teisendamine on alati võimalik.
Sisendi esimesel real on sõnade arv `N (1≤N≤200)`. Järgmisel `N` real on igaühel üks sõna. Sisendi viimasel real on kaks sõna, mille vahelist teisendust tuleb mõõta.

NÄIDE:

```
8
kada
kaja
kala
pada
paha
raba
raha
sada
kala raha
```

Vastus:

```
4
```

6.11.6 Kahevärviprobleem

Nagu peatüki sissejuhatuses kirjutatud, on üks graafiteooria tuntumaid probleeme neljavärviteoreem, kus uuritakse, kas iga kaart on värvitav nelja värviga. Teoreem on tuntud ka selle poolest, et selle tõestamiseks kasutati arvutit. Siin ülesandes vaadeldakse sarnast, aga palju lihtsamat küsimust: kas etteantud kaarti on võimalik värvida kahe erineva värviga?
Sisendi esimesel real on kaks täisarvu: riikide arv `N` (`1≤N≤200`) ning piiride arv `M`. Järgmisel `M` real on igaühel kaks täisarvu, mis näitavad, et neil kahel riigil on ühine piir. Väljundisse kirjutada `JAH`, kui kaart on võimalik värvida ainult kahe värviga ning `EI`, kui see ei ole võimalik.

NÄIDE 1:

```
3 3
0 1
1 2
2 0
```

Vastus:

```
EI
```

NÄIDE 2:

```
3 2
0 1
1 2
```

Vastus:

```
JAH
```

NÄIDE 3:

```
9 8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
```

Vastus:

```
JAH
```



6.11.7 Joonejälgija robot

Robotexil ja teistel sarnastel võistlustel on üheks võistlusalaks sageli sellise roboti programmeerimine, mis järgib maha joonistatud joont. Antud ülesandes on maha joonistatud terve ruudustik, millel sõidab ümmarguse kujuga Roomba-suurune robot (diameeter 35 cm). Robot oskab sõita ainult nii, et jälgitav joon läbib kogu aeg roboti keskkohta. Jooned on üksteisest 20 cm kaugusel, nii et kõrvalolevate joonteni jääb alati pisut ruumi. Robotile on võimalik anda järgmisi käske: 20 cm edasi, 40 cm edasi, 60 cm edasi, 90° vasakule, 90° paremale. Iga käsu täitmine võtab ühe sekundi. Järgmiseks on robotile ehitatud takistusrada, kus ruudustikule joonte vahele on paigutatud kuubikud, mida robot peab vältima. Ülesandeks on leida kiireim tee, kuidas robot saab jõuda ühest etteantud punktist teise.

Lihtsuse mõttes kasutame edaspidi ühikuna ruutude enda suurust. Sisendi esimesel real on joonte vahele jääva ruudustiku pikkus `M (1≤M≤50)` ja laius `N (1≤N≤50)`. Järgmisel `M` real on igaühel `N` arvu väärtustega kas `0` või `1`. `0` tähistab, et ruut on tühi, `1` tähistab takistust.

Viimasel real on 4 täisarvu ja üks tähemärk, mis tähistavad roboti alguspunkti koordinaate, sihtkoha koordinaate ja roboti algset suunda. Koordinaadid algavad ruudustiku vasakust ülemisest nurgast. Roboti suund on üks märkidest `N`, `E`, `S` või `W`.

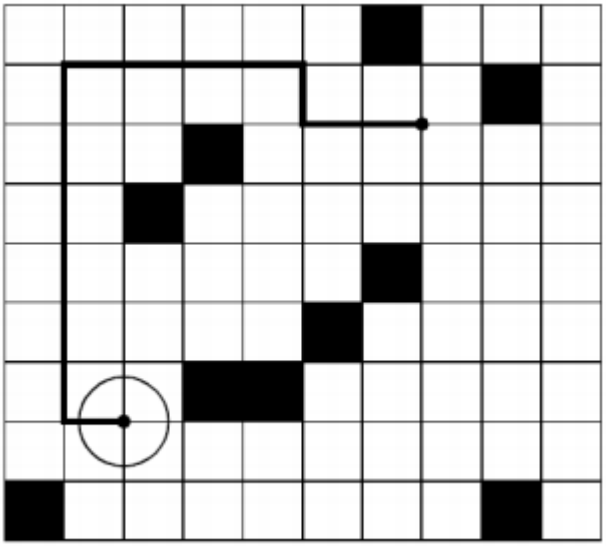
Väljundisse kirjutada, mitme sekundiga on robotil võimalik lõpp-punkti jõuda. Kui soovitud teekond pole võimalik, väljastada `-1`.

NÄIDE:

```
9 10
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0
7 2 2 7 S
```

Vastus:

12 (vt joonist)



6.11.8 Numbriruudud

Ruudustik on täidetud numbritega 0-9 (vt näidet joonisel). Vasakus ülemises ruudus on mängunupp, eesmärgiks on jõuda paremasse alumisse ruutu. Igal käigul võib nupuga astuda sammu vasakule, paremale, üles või alla, aga iga kord liidetakse sinu skoorile vastavas ruudus oleva numbri väärtus. Eesmärgiks on jõuda lõppu võimalikult väikese skooriga.
Sisendi esimesel real on ruudustiku kõrgus `M` ja laius `N` (`1≤M, N≤999`). Järgmisel `M` real on igaühel `N` numbrit, millest moodustubki ruudustik. Väljundisse kirjutada üks täisarv: minimaalne skoor, millega on võimalik algusest lõppu jõuda.

NÄIDE 1:

```
4 5
0 3 1 2 9
7 3 4 9 9
1 7 5 5 3
2 3 4 2 5
```

Vastus:

24 (vastab joonisele)

0	3	1	2	9
7	3	4	9	9
1	7	5	5	3
2	3	4	2	5

NÄIDE 2:

```
1 6
0 1 2 3 4 5
```

Vastus:

15

6.11.9 Optimaalne ruuting

IP (*Internet Protocol)* on peamine reeglistik, mille abil arvutid Internetis andmeid edastavad. IP oluline omadus on see, et ta defineerib **ruutingu** ehk meetodi, kuidas andmepakette on võimalik saata läbi paljude arvutite. Tavaliselt on igas vahepealses arvutis (ruuteris) tabel, mis ütleb, millisesse järgmisse punkti antud sihtaadressiga pakett tuleb saata. Pakett rändab ühest ruuterist teise kuni jõuab sihtkohta. Ühenduse kogukiirus sõltub seega vahepealsete etappide kiirusest.
Antud on rida ruutereid nendevaheliste ühenduste ja kiiruste infoga. Leida minimaalne võimalik aeg, mis kulub info ühest arvutist teise saatmiseks.
Sisendi esimesel real on neli täisarvu: arvutite arv `M` (`2≤M≤20000`), ühenduste arv `N` (`0≤N≤50000`), algse arvuti indeks `A` (`0≤A<M`) ja sihtarvuti indeks `O` (`0≤O<M`). Järgmisel `N` real on ühenduste info, mis on väljendatud kolme täisarvuna: ühendatud arvutite indeksid ning ühendusele kuluv aeg millisekundites. Väljundisse kirjutada minimaalne aeg, mis kulub arvutist `A` arvutisse `O` paketi saatmiseks. Kui ühendust luua pole võimalik, väljastada `-1`.

NÄIDE 1:

```
2 1 0 1
0 1 100
```

Vastus:

```
100
```

NÄIDE 2:

```
3 3 2 0
0 1 100
0 2 200
1 2 50
```

Vastus:

```
150
```

NÄIDE 3:

```
2 0 0 1
```

Vastus:

```
-1
```



Joonisel on Eestit katva lairibavõrgu ESTWin kaart.

6.11.10 Liftid

Suurtes pilvelõhkujates on sageli erineva kiirusega liftid, mis teenindavad erinevaid korruseid. Näiteks võib seal olla tavaline lift, mis liigub korruste 41 ja 50 vahel ning ekspresslift, mis peatub igal kümnendal korrusel. Sa oled ühe sellise pilvelõhkuja fuajees (0-korrusel), ülesandeks on leida, kuidas võimalikult kiiresti jõuda ettenähtud korrusele. Liikuda võib ainult liftidega ja ühelt liftilt teise minek võtab alati ühe minuti.

Sisendi esimesel real on kaks täisarvu: liftide arv `N` (`1≤N≤5`) ja korrus `K`, kuhu on vaja jõuda (`1≤K≤100`). Teisel real on `N` täisarvu, mis tähistavad liftide kiirusi: sekundite arvud, mis kulub ühe korrusevahe läbimiseks. Järgmistel `N` real on igaühel rida täisarve, mis tähistavad, millistel korrustel vastav lift peatub. Väljundisse kirjutada üks täisarv: sekundite arv, mis kulub soovitud korrusele jõudmiseks. Kui sihtkohta jõudmine pole võimalik, kirjutada väljundisse `-1`.

NÄIDE 1:

```
2 30
10 5
0 1 3 5 7 9 11 13 15 20 99
4 13 15 19 20 25 30
```

Vastus:

275 - Esimese liftiga 13. korrusele (130 sekundit), oodata teist lifti (60 sekundit), teise liftiga 30. korrusele (85 sekundit).

NÄIDE 2:

```
2 30
10 1
0 5 10 12 14 20 25 30
2 4 6 8 10 12 14 22 25 28 29
```

Vastus:

285 – Esimese liftiga 10. korrusele, siis teisega 25. korrusele, siis jälle esimesega 30. korrusele, aeg kokku 10*10+60+15*1+60+5*10=285 sekundit.

NÄIDE 3:

```
3 50
10 50 100
0 10 30 40
0 20 30
0 20 50
```

Vastus:

3920 – Esimese liftiga 30. korrusele, teisega 20. korrusele, siis kolmandaga 50. korrusele.



NÄIDE 4:

```
1 1
2
0 2 4 6 8 10
```

Vastus:

```
-1
```

6.12 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk6 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
SGL.cpp, SGL.java, SGL.py	Graafi sügavuti läbimine, tsüklilisuse kontroll
Ratsu1.cpp, Ratsu1.java, Ratsu1.py	Ratsu teekond (sügavuti läbimine).
LGL.cpp, LGL.java, LGL.py	Graafi laiuti läbimine.
Ratsu2.cpp, Ratsu2.java, Ratsu2.py	Ratsu lühim tee ühelt ruudult teisele (laiuti läbimine).
Ratsu3.cpp, Ratsu3.java, Ratsu3.py	Ratsude asetamine malelauale (sidususkomponendid).
Veekogud.cpp, Veekogud.java, Veekogud.py	Veekogude loendamine kaardil (üleujutamine).
Alused.cpp, Alused.java, Alused.py	Graafi kahealuseliseks jaotamine.
Loomad.cpp, Loomad.java, Loomad.py	Tugevuse järgi järjestamine (topoloogiline sorteerimine).
Soiduaeg.cpp, Soiduaeg.java, Soiduaeg.py	Asulatevahelised sõiduajad (Dijkstra algoritm).
Kahendpuud.cpp, Kahendpuud.java, Kahendpuud.py	Kahendpuu taastamine ees- ja keskjärjestuse põhjal ning lõppjärjestuses töötlemine.

TEINE OSA - EDASIJÕUDNUTELE

7 EFEKTIIVNE PROGRAMMEERIMISTEHNIKA



Algaja programmeerija jaoks on suur asi, kui tema programm tööle hakkab ja enamvähem õiget asja teeb. Tõepoolest, see on päris uhke tunne ja selle tunde pärast armastavadki paljud inimesed programmeerimist: „Ma olen jumal, kes tegi mitte millestki midagi ja andis masinale elu!“

Need lugejad, kes on jõudnud siia, õpiku teise osani, pole aga enam algajad, vaid arvestatava kogemusega programmeerijad. Kui alguses oli ainsaks edukriteeriumiks see, kas programm töötab, siis nüüd tuleb esile veel palju teisi kriteeriume, millele üks hea programm vastama peaks:

- Jõudlus – käesoleva raamatu põhiteema – kas minu programm töötab piisavalt kiiresti?
- Skaleeruvus – leiab samuti siin raamatus kajastamist – kuidas teha nii, et programm ka suuremate andmehulkade puhul hästi töötaks.
- Kasutusmugavus – kui kergesti ja efektiivselt saab kasutaja antud programmiga hakkama?
- Hallatavus – kui kerge on programmis muudatusi teha? Kas muudatus ühes kohas võib midagi muud katki teha?
- Töökindlus – kui hästi saab programm hakkama erijuhtumite ja ootamatute olukordadega?
- Laiendatavus – kui lihtne on programmile lisada täiendavat funktsionaalsust?
- Turvalisus – kas programm hoiab ära „mitte ettenähtud tagajärgede“ tekitamise?

Kui me soovime kirjutada head „päris elu“ programmi, siis on kõik need punktid sama tähtsad kui see, kas programm teeb õiget asja, vahel isegi tähtsamad!

Võrdleme näiteks kaht olukorda:

1. Programm töötab valesti, aga on kergesti modifitseeritav.
2. Programm töötab õigesti, aga on kirjutatud segaselt ja keeruliselt, nii et sellesse pole võimalik mingeid muudatusi teha.

Esimesel juhul saab vead ära parandada ja kõik on korras. Kui aga teisel juhul on tarvis programmi mingeid muudatusi teha, on vahel on ainsaks väljapääsuks programm algusest peale uuesti kirjutada. Seega kasvab koos programmi elueaga ka hallatavuse ja laiendatavuse väärtus.

Võistlusprogrammeerimise seisukohalt on ülaltoodud loetelust tähtsaimad mõistagi **jõudlus** ja **skaleeruvus** ning enamik käesolevast raamatust tegelebki nende teemadega. Võistlusel kasutatavate programmide eluiga on tavaliselt ülilühike ja teised aspektid jäävad tavaeluga võrreldes tagaplaanile, kuid ka siin ei saa ignoreerida **hallatavust** (kui programmis on viga või vaja midagi ümber kirjutada, siis kas seda on lihtne teha), **töökindlust** (igasuguste vigade vältimiseks) ja **laiendatavust** (ülesanded koosnevad sageli mitmest osast – kui üks osa on lahendatud, siis kui mugavalt on võimalik programmi täiendada).

Tänapäeva tarkvara on oma olemuselt ülikeeruline, koosnedes paljudest komponentidest, mis kõik peavad harmooniliselt koos töötama, kuid samas ka ülihabras, sest vahel võib üheainsa biti muutmine tulemuse valeks muuta või siis programmi üldse ära lõhkuda. Võime kujutleda, et tegu on tuhandete hammasratastega, mis peavad kõik sünkroonis pöörlema ja niipea kui üks neist kinni kiilub, siis programm enam ei tööta.



Siin peatükis uurime, kuidas seda kinnikiilumist võimalikult hästi vältida ja kui see siiski juhtub, siis kuidas kinni jäänud ratast leida ja taas valla päästa.

7.1 KOODISTIIL

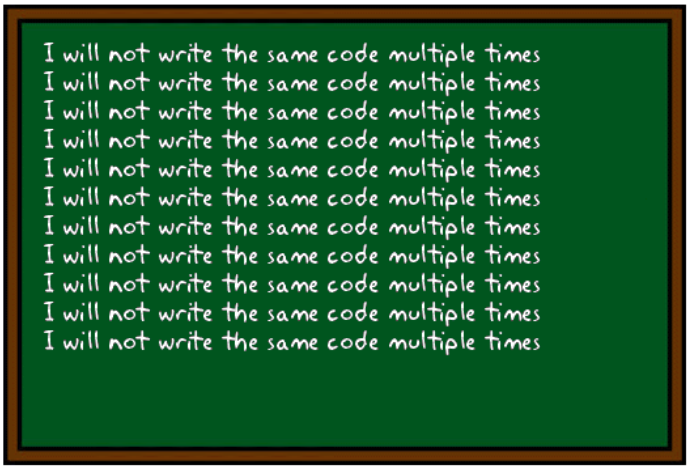
7.1.1 Pikaajaline ja lühiajaline lähenemine

Mida saab programmiga teha? Intuitiivne vastus on „valmis kirjutada ja käivitada“. Hea küll, kirjutasime ja käivitasime, aga nüüd on selles vaja midagi muuta või parandada. Kui algne kood polnud kirjutatud just hetk tagasi, tuleb enne täiendamist teha veel midagi olulist: senist koodi **lugeda**, et aru saada, mida see tegi, muidu pole efektiivne täiendamine võimalik. On programme, mida ei loeta pärast kirjutamist mitte kunagi, kuid on ka programme, mida loetakse tuhandeid kordi – enamasti juhul, kui tegu on tiimitöö ja pikaajalise projektiga. Vastavalt sellele, kui palju me eeldame, et programmi on vaja lugeda, muutub ka **loetavusele optimiseerimine** järjest tähtsamaks: et lugemise aega säästa, on sageli vaja kirjutamisele rohkem aega kulutada.

Programmeerimisvõistlustel kiiresti ja hooletult koodi kirjutamine tekitab kergesti halbu harjumusi, sest võistlusel kasutatava programmi koodi on harva pärast võistlust tarvis. Siiski kulub hea stiil ka juba võistluse jooksul ära, hoides kokku vigade leidmisele ja parandamisele kuluvat aega.

7.1.2 Alamprogrammid

Programmeerimisvõistlustel on inimestel sageli kogu programm ühes põhifunktsioonis. Kui lahendus on lihtne ja vähem kui tunniga realiseeritav ning sa oled ka kindel, et tegemist on õige lahendusega, pole sellest iseenesest midagi katki. On aga palju võistlusi (päriselu ülesannetest rääkimata), kus ülesande lahendamiseks on aega mitu päeva ning töö ka selle võrra mahukam. Sel juhul tuleks programm kindlasti jagada väiksemateks funktsioonideks, kuna see teeb koodi lihtsamini loetavamaks ja paremini hallatavaks.



Täpsemalt on alamprogrammide kasutamisel järgmised olulised eelised:

- **Ülevaatlikkus**. Kui vaadeldav blokk on pikem kui inimene korraga haarata suudab, läheb lõppu jõudmise ajaks kergesti meelest, mis alguses toimus. Jagades koodi väiksemateks osadeks lööme ühe hoobiga kaks kärbest: esiteks on iga osa puhul koha arusaadav, mis toimub, teiseks saab igale osale anda ülevaatliku, kirjeldava **nime**, mis hõlbustab mõistmist veelgi.
- **Probleemi jagamine alamosadeks**. Paljud ülesanded on võimalik jagada loogiliselt eristuvateks alamosadeks, sel juhul on hea, kui lahendus järgib sama struktuuri. Esiteks võimaldab see probleemi osade kaupa lahendamist, kuid teise, veel tähtsama võidu saavutame läbi parema **testitavuse**. Igale funktsioonile, mis probleemist teatud osa lahendab, on võimalik koostada eraldi testid, mis selle osa korrektsust kontrollivad – selline lähenemine hoiab sageli vigade otsimise aega kokku.
- **Koodi taaskasutuse võimaldamine.** Sageli tuleb ülesande raames sama või sarnast protseduuri mitu korda korrata. „Laisk“ meetod oleks seda koodi lihtsalt kopeerida, aga kui selgub, et selles koodis on iga või tarvis midagi täiendada, maksab kopeeritud koodi mitmekordne parandamine peagi kätte, ühelt poolt läheb rohkem aega ja teiselt poolt on kerge uusi vigu teha.

Kui pikk tohiks üks alamprogramm olla? Tuntud tarkvarakonsultant ja agiilse arenduse kontseptsiooni üks loojaid Robert „Uncle Bob“ Martin (https://cleancoders.com/) soovitab, et alamprogrammi pikkus võiks olla umbes neli rida ja kümme on juba liiga palju. Enamik programmeerijaid peab nii madalat piiri siiski liiga radikaalseks, kuid nõustuvad, et liiga pikad moodulid on halvad. Mina isiklikult kirjutasin oma esimesed programmid tekstimoodis kuvaril, mis näitas 25 rida ja 80 märki real, see oli üldine standard. Ka praegu avanevad paljud virtuaalterminalid vaikimisi just sellistes mõõtudes. Kui vaadeldav alamprogramm on suurem kui sellisesse aknasse mahub, tuleb seda kerida ning lugemine on juba raskendatud. Samuti on **25 rida** ka enamvähem piir, kui palju infot inimene korraga haarata suudab, mistõttu kasutatakse koodistandardites sageli just seda suurust.



Uncle Bob


Targo esimene programmeerimisarvuti oli samasugune

Taaskasutusest tuleb lähemalt juttu järgmises näites:

7.1.3 Näide: Tankid



M1A1 Abrams tank

2016/17 olümpiaadihooaja raames korraldati ka Tankide Turniir, kus iga osaleja sai programmeerida tanki, mis vastaval võistlusväljakul teistega võitlema peab. Sellise programmi kirjutamisel tekib üsna palju korduvaid osi, mida omaette funktsioonidesse koondada.

Näiteks vaatame siin üht võistlusel esinenud väga pikka funktsiooni (umbes 400 rida) `MineNurka`, mis proovib mängija tanki liigutada laual lähimasse nurka. Liikumise käigus tulistatakse ettejäävaid tanke ning vajadusel muudetakse liikumissuunda ja proovitakse minna ümber takistuse. Kogu funktsioon on liiga pikk, et siin ära tuua, aga siin on sellest esimene veerand:

```
int mineNurka()
{
    // mängulaud on m x n; x, y on tanki koordinaadid
    if (m - x > x) {  // mis pooles tank on, mis nurk on lähemal
        if (n - y > y) { // mis veerandis
            bool ol = 1; // on liikunud
            while (!ok[0][0] && (x != 0 || y != 0)) { // kuni nurk on vaba
                ol = 0;
                while (x != 0 && y != 0 && !ok[y - 1][x - 1]) { //liigu nurga suunas
                    ol = 1;
                    cout << "M5" << endl; // M5 liigu suunas alla-vasakule
                    lts(); // loeb info, sh tanki uued koordinaadid nx ja ny
                    if (nx == x && ny == y) { // ei liikunud (nx, ny - uued koordinaadid)
                        ok[y - 1][x - 1] = 1; // seal on keegi ees
                        fire(5, 6); // tulista sinna
                    }
                    xnx(); // uuendab koordinaadid
                }
                bool fm = 1; // diagonaalis liikuda ei saa
                while (x > 0 && !ok[y][x - 1] && (x > y || fm)) { // x telge mööda
                    ol = 1;
                    fm = 0;
                    cout << "M6" << endl; // liigu vasakule
                    lts();
                    if (nx == x && ny == y) {
                        ok[y][x - 1] = 1;
                        fire(6, 6);
                    }
                    xnx();
                }
                fm = 1;
                while (y > 0 && !ok[y - 1][x] && (x < y || fm)) { // alla
                    ol = 1;
                    fm = 0;
                    cout << "M4" << endl; // alla
                    lts();
                    if (nx == x && ny == y) {
                        ok[y - 1][x] = 1;
                        fire(4, 6);
                    }
                    xnx();
                }
```

```
            if (!ol) { // kui ei ole üldse liikunud
                if (x < y) { // üleval vasakul võrreldes õige diagonaaliga
                    cout << "M3" << endl; // liigub alla paremale
                    lts();
                    if (nx == x && ny == y) { // seal ka keegi ees
                        ok[y - 1][x + 1] = 1;
                        fire(3, 6); // tulista
                        cout << "M2" << endl; // liigu paremale
                        lts();
                        if (nx == x && ny == y) {
                            ok[y][x + 1] = 1;
                            fire(2, 6); // tulista
                            cout << "M1" << endl; // liigu üles paremale
                            lwf(); // loe ilma tulistamata (lts, xnx järjest)
                            cout << "M3" << endl; // liigu alla paremale
                            lwf();
                            continue; // jätkab nurka minekut algusest (oma veerandis)
                        }
                        xnx(); // kui sai paremale, uued koordinaadid
                        cout << "M3" << endl; // alla paremale
                        lwf(); // lts; xnx
                        continue;
                    }
                    xnx();
                    continue; // jätka alla vasakule
                }
                cout << "M7" << endl; // liigu üles vasakule
                lts();
                if (nx == x && ny == y) { // kui ei õnnestunud
                    ok[y + 1][x - 1] = 1; // seal on keegi ees
                    fire(7, 6); // tulista
                    cout << "M0" << endl; // liigu üles
                    lts();
                    if (nx == x && ny == y) { // kui ei õnnestunud
                        ok[y + 1][x] = 1; // seal on keegi ees
                        fire(0, 6); // tulista
                        cout << "M1" << endl; // liigu üles paremale
                        lwf();
                        cout << "M7" << endl; // liigu üles vasakule
                        lwf();
                        continue; // jätka
                    }
                    xnx();
                    cout << "M7" << endl; // liigu üles vasakule
                    lwf();
                    continue;
                }
                xnx();
                continue;
            }
        }
        return 1;
    }
```

Edasi järgneb praktiliselt sama kood ülejäänud kolme nurga jaoks, erinevus on vaid liikumissuundades. Ilmselgelt on sellist koodi raske lugeda ja veelgi raskem on selles parandusi teha. Vaatame, kuidas saaks seda koodi lühematesse funktsioonidesse jagada. Esiteks soovitakse valida lähim nurk, kuhu minna. Põhifunktsioon `MineNurka` võiks olla siis järgmine:

```
int mineNurka() {
    int nurkX = (m - x > x) ? 0 : m - 1;
    int nurkY = (n - y > y) ? 0 : n - 1;
    return mineNurka(nurkX, nurkY);
}
```

Siit kutsutakse välja funktsiooni `MineNurka` juba konkreetse nurga koordinaatidega:

```
int mineNurka(int nurkX, int nurkY) {
    int suund = leiaSuund(nurkX, nurkY);
    while (!ok[nurkY][nurkX] && (x != nurkX || y != nurkY)) {
        if (!liiguNurgaSuunas(nurkX, nurkY, suund)) { // kui ei ole üldse liikunud
            poikle(nurkX, nurkY, suund); // põikleme
        }
    }
    return leiaVastassuund(suund);
}
```

Funktsioonid `leiaSuund` ja `leiaVastassuund` on lihtsad funktsioonid, mis parandavad eelkõige loetavust:

```
int leiaSuund(int x, int y) {
    if (x == 0 && y == 0) return ALLA_VASAKULE;
    if (x == 0) return YLES_VASAKULE;
    if (y == 0) return ALLA_PAREMALE;
    return YLES_PAREMALE;
}

int leiaVastassuund(int suund) {
    return (suund + 4) % 8;
}
```

Olulisemad on siin funktsioonid `liiguNurgaSuunas` ja `põikle`. Kõigepealt esimene neist:

```
bool liiguNurgaSuunas(int nurkX, int nurkY, int suund){
    bool ol = 0;
    while (x != nurkX && y != nurkY && onVaba(suund)) {
        ol = liiguTulega(suund); //liigu diagonaalis
    }
    int xsuund = nurkX ? PAREMALE : VASAKULE;
    int ysuund = nurkY ? YLES : ALLA;
    bool fm = 1; // forsseeritud liikumine
    while (onLaual() && onVaba(xsuund) && (nurkX - x > nurkY - y || fm)) {
        ol = liiguTulega(xsuund); //liigu horisontaalselt
        fm = 0;
    }

    fm = 1;
    while (onLaual() && onVaba(ysuund) && (nurkX - x < nurkY - y || fm)) {
        ol = liiguTulega(ysuund); // liigu vertikaalselt
        fm = 0;
    }
    return ol;
}
```

Siingi on abifunktsioonid loetavuse parandamiseks `onLaual ja onVaba`:

```
bool onLaual() {
    return (x > -1) && (x < m) && (y > -1) && (y < n);
}

bool onVaba(int suund){
    return !ok[y + dy[suund]][x + dx[suund]];
}
```

Liikumise funktsioon on järgmine:

```
bool liigu(int suund, bool tulista) {
    cout << "M" << suund << endl;
    lts(); // info sisse, siin loetakse
    if (tulista && nx == x && ny == y) { // kui tulistamisega
        ok[y + dy[suund]][x + dx[suund]] = 1; // seal on keegi ees
        fire(suund, 6); // tulista sinna
    }
    xnx(); // muudab x ja y väärtused
    return true;
}
```

Funktsioonid `liiguTulega` ja `liiguTuleta` on taas loetavuse parandamiseks:

```
void liiguTuleta(int suund) {
    liigu(suund, false);
}

bool liiguTulega(int suund) {
    return liigu(suund, true);
}
```

Vaatamata on veel funktsioon `poikle`, milles pannakse peamiselt paika põiklemise suunad:

```
void poikle(int nurkX, int nurkY, int suund){
    int pSuund, d;
    if (!nurkX != !nurkY){
        pSuund = (suund + 2) % 8;
        d == 1;
    }
    else {
        pSuund = (suund + 6) % 8;
        d = -1;
    }

    if (nurkX - x < nurkY - y) {
        liiguPoikle(pSuund, d);
    }
    else {
        liiguPoikle(leiaVastassuund(pSuund), -1*d);
    }
}
```

Kõige lõpuks funktsioon `liiguPoikle`:

```
void liiguPoikle(int pSuund, int d) {
    cout << "M" << pSuund << endl; // liigub uues suunas
    lts();
    if (nx == x && ny == y) { //seal ka keegi ees
        ok[y + dy[pSuund]][x + dx[pSuund]] = 1;
        fire(pSuund, 6); //tulista
        int vSuund = (pSuund + d + 8) % 8;
        cout << "M" << vSuund << endl; //liigu jälle uues suunas
        lts();
        if (nx == x && ny == y) { // ikka keegi ees
            ok[y + dy[vSuund]][x + dx[vSuund]] = 1;
            fire(vSuund, 6); // tulista
            liiguTuleta((vSuund + d + 8) % 8); // liigu korraks vastassuunas
            liiguTuleta(pSuund); // ja tagasi põiklemise suunas
            return; // jätka nurka minekut algusest peale (oma veerandis)
        }
        xnx();
        liiguTuleta(pSuund);
        return;
    }
    xnx();
    return;
}
```

Kokku õnnestus kahandada üks umbes 400-realine funktsioon vähem kui 150le reale, mis on omakorda jagatud üsna mõistlikult arusaadavateks kirjeldava nimega osadeks.

7.1.4 Kommentaarid

Sageli võib programmeerimisõpikutest lugeda, et kood tuleb kommenteerida. Üheks soovituseks on kirjutada enne kommentaarid, mida kood võiks teha ning seejärel alles hakata koodi kirjutama. See on hea praktika, mis aitab oma mõtteid koondada ning ülesandest tervikpilti saada. Võib juhtuda, et esialgne idee oli üldse vildakas ja siis on palju parem, kui saad sellest aru pärast paari rea kommentaaride kirjutamist, mitte aga siis, kui oled kirjutanud hulga koodi.

Pahatihti minnakse aga kommenteerimise soovitustega liiale, näiteks paljudes ettevõtetes on kohustuslik kommenteerida iga meetod, parameeter, muutuja jne. Tulemuseks on nonsenss:

```
/*
** Tagastab kõrguse.
*/
double getHeight()
```

Sellistel kommentaaridel on ka suur oht vananeda. Kui nõuded muutuvad, võib ka kirjeldatava objekti olemus muutuda, kuid kommentaar jääb ikka samaks ning hakkab kasu asemel tooma kahju.

Näiteks leidsin ma koodist kord sellised read:

```
// ISO 3166 two letter country code
string threeLetterCode;
```

Uh-oh. Ilmselgelt oli millalgi kahetähelistelt koodidelt mindud üle kolmetähelistele ning kommentaar, mis enne oli olnud lihtsalt kasutu, muutus nüüd aktiivselt kahjulikuks.

Kommentaar, mis ütleb midagi, mis koodis niigi kirjas on, ei lisa mingit väärtust.

Kommentaarid võivad olla vajalikud, et kirjeldada **miks** kood on mingil viisil kirjutatud. Kui aga juhtub, et me peame kommenteerima, **mida** kood teeb, saab selle asemel tavaliselt parandada koodi loetavust.

Iga kommentaar on märgiks sellest, et kood ei kirjelda iseennast perfektselt ja seda peaks käsitlema kui võimalust koodi parandada.

Näiteks sellise koodi

```
// konverteerime eurod sentideks
a = x * 100;
```

asemel võib anda muutujatele paremad nimed:

```
sendid = eurod * 100;
```

Järgmine halb näide (kood on lihtne, aga mis maagiat siin tehakse?)

```
double r = n / 2;
while (abs(r - (n / r)) > t) {
    r = 0.5 * (r + (n / r));
}
```

Parem variant:

```
// leiame ruutjuure Newton-Raphsoni meetodiga
double r = n / 2;
while (abs(r - (n / r)) > t) {
    r = 0.5 * (r + (n / r));
}
```

Veel parem:

```
double LigikaudneRuutjuur(double n, double tapsus) {
    double r = n / 2;
    while (abs(r - (n / r)) > tapsus) {
        r = 0.5 * (r + (n / r));
    }
    return r;
}
```

Kommentaari asemel eraldi funktsiooni loomisega saavutasime kaks eesmärki:

- Eraldasime kasuliku funktsionaalsuse taaskasutatavaks alamprogrammiks.
- Andsime sellele kasuliku, iseennast kirjeldava nime, mis võimaldab koodist arusaamist ka seal, kus vastavat funktsiooni välja kutsutakse.

Samuti on tavapärane (aga kahjulik) nõuanne kirjutada iga muutuja juurde, mis on selle sisu. Selmet kirjutada iga muutuja juurde, milleks seda kasutatakse, tuleb nimetada muutuja kohe nii, et selle otstarve on selge.

7.1.5 Nimed



Phil Karlton

Kunagisele Netscape'i tarkvaraarhitektile Phil Karltonile omistatakse järgmist tsitaati:

„Tarkvaraarenduses on ainult kaks suurt probleemi:

1. Puhvrite invalideerimine
2. Asjade nimetamine
3. Ühe võrra eksimused"

Originaalis: „There are only two hard things in software engineering: cache invalidation, naming things, and off-by-one errors."

Siin siis veidi näpunäiteid, kuidas teist probleemi paremini lahendada.

- Mida pikem on kood, kus muutujat kasutatakse, seda raskem on järge pidada, mida muutuja teeb. Sellest järeldub **skoobi reegel**: mida pikem on skoop, kus muutujat kasutatakse, seda pikem peaks olema muutuja nimi. Tsüklimuutujatena on sobilik kasutada `i` ja `j`, samuti vaid ühes lühikeses tsüklis kasutatav abimuutuja võib olla ühetähelise nimega. Globaalsete muutujate puhul seevastu on hea, kui neil on ilusad kirjeldavad nimed. Koodi kerimine selleks, et uurida, mis see t või p täpselt oli, on selgelt ebaotstarbekas.
- Muutujad vastavad tavaliselt mingitele asjadele või objektidele ning neid võiks nimetada nimisõnadega, nt `int[][] laud`.
- Veidi erandlikuks on `boolean` tüüpi muutujad, mida sobib nimetada pigem omadus- või määrsõnadega, soovi korral võib lisada ette ka tegusõna 'on' nt `valmis` või `onValmis`; `kiire` või `onKiire`.

Järgneb koodinäide Lemmingute ülesandest, kus esimeses tulbas on kirjeldavate nimedega kood, teises lühikestega:

```
while (!tramm.empty() &&
  ((trammisabad[peatus].size() < jk_pikkus) ||
  tramm.top() == peatus)) {
    if (tramm.top() != peatus) {
        trammisabad[peatus].push(tramm.top());
    }
    tramm.pop();
    aeg++;
}
```

```
while (!s.empty() &&
  ((q[p].size() < l) ||
  s.top() == p)) {
    if (s.top() != p) {
        q[p].push(s.top());
    }
    s.pop();
    t++;
}
```

Vastuväide oleks, et pikemate nimede kirjutamine võtab rohkem aega, kuid esiteks on tänapäeva programmeerimiskeskkondades abistavad vahendid nimede automaatseks lõpetamiseks (*auto-complete*) ning teiseks on natuke pikema nime kirjutamisele kuluv aeg üsna väike võrreldes probleemi sisulisele lahendamisele ja programmi struktuuri loomisele kuluva ajaga. Võrreldes ajaga, mis kuluks vigade otsimisele, sest programm on muutunud ka autorile endale arusaamatuks, on need ekstra sekundid aga täiesti tühised.

Võistlusprogrammeerimise kontekstis võib ja on isegi soovitatav kasutada ülesande tekstis antud muutujate nimetusi (tavaliselt mingid N, M, K jne), see tähendab ka ühetähelisi tähistusi.

7.1.6 Funktsioonide nimetamine

Funktsiooni nimetused peaks ütlema, mida funktsioon teeb. Kui muutujatele sobivad nimed olid nimisõnad, siis funktsioonid on tegevused ja nende nimed võiks olla tegusõnad. Väga hästi sobivad näiteks `vaheta`, `otsi`, `looGraaf`, `leiaLyhimTee` ja muud sarnased nimed. Erandiks on funktsioonid, mille peamiseks otstarbeks on tõeväärtuse tagastamine, neid võiks nimetada sarnaselt `boolean`-tüüpi muutujatega, näiteks

```
bool onLaual(int x, int y, int pikkus) {
    return ((x >= 0) && (x < pikkus) && (y >= 0) && (y < pikkus))
}
```

Seda laadi nimetused teevad koodi hästi loetavaks:

```
if (onLaual(naaber_x, naaber_y, dim) {
    teeMidagi();
}
```

Ka funktsiooni parameetrid võiks olla funktsiooni päises mõistlikult nimetatud. Nt funktsioon päisega

```
long long liida(int a, int b);
```

võiks eeldatavasti liita kokku kaks argumendiks saadavat arvu a ja b, aga funktsioon päisega

```
long long liida(int start, int end);
```

liidab pigem mingit arvude vahemikku või mingit vahemikku jadas. Ainus erinevus on argumentide nimetustes! Või kui ülaltoodud funktsioon oleks päisega

```
bool onLaual(int a, int b, int c);
```

võib endalgi sassi minna, kas enne peaks ette andma koordinaadid ja siis laua mõõdu või vastupidi.

7.2 TESTIMINE

Võistlustel kasutatakse programmide hindamisel tavaliselt järgmisi meetodeid:

- **Testide kaupa** hindamine, kus punkte saab iga läbitud testi eest, mida rohkem teste ära teed, seda rohkem punkte. Enamik Eesti madalama taseme võistlusi kasutab seda lähenemist.
- „**Pakikaupa**“ testimine, kus lahendus peab punktide saamiseks läbima kõik testid. Õige lahendus annab 100% ja kui kasvõi üks test ei lähe läbi, on tulemuseks 0. Seda skeemi kasutatakse enamikel *online* võistlustel (CodeForces, TopCoder jne).
- Testid on jagatud **alamülesanneteks**, iga konkreetset alamülesannet testitakse pakikaupa. Seda kasutatakse enamikul rahvusvahelistel võistlustel nagu IOI või BOI.
- **Avatud testid**, kus esitada tuleb testide vastused, mitte programm, mis ülesannet lahendab. Inglise keeles nimetatakse neid „*output-only*“ ülesanneteks.

Loomulikult ei ole testimine ainult võistluste teema – seda on vaja ka päris elus, sest kes soovib kasutada programmi, mis töötab ainult teatud juhtudel.

Kuidas siis ikkagi testida? Selleks on muidugi vaja programmi, mida testida, teste ehk andmestikku, millega testida, ning teada õigeid vastuseid, millega testitava programmi tulemusi võrrelda. Testide koostamise jaoks on vaja kõigepealt mõelda välja, millised testid võiks olla, ning juhuslike testide tegemiseks kasutada testigeneraatorit. Õige vastuse saamiseks sobib võimalusel nii käsitsi kontrollimine, aga ka mõni lihtsam programm, nt selline, mis ülesande jõumeetodil lahendab, kuid

pole võistlusel esitamiseks piisavalt kiire. Veel üheks võimaluseks on kasutada lahenduse leidmiseks testigeneraatorit – st koostamise käigus saab leida vastuse või mõnikord on kavalam genereerida testid oodatavast vastusest lähtuvalt.

Selleks, et testimist veelgi efektiivsemalt läbi viia, tulevad kasuks skriptid, mis teste käivitavad ja tulemusi kontrollivad.

7.2.1 Testide koostamine

Milliseid teste üldse koostada? Üldiselt võib testid jagada kolme erinevasse liiki:

- Ülesande teksti analüüs. Siia kuuluvad maksimaalsete ja minimaalsete väärtustega testimine, testid tühjade väärtuste ja nullidega, samuti testid, mis kontrollivad väljundi korrektset formaatimist. Oluline on ka kontrollida, kas piirangutest on õigesti aru saadud, näiteks kas võrratus on range või mitterange.
- Ülesande sisu analüüs, millest võiks tulla testid erijuhtude ja piirjuhtude jaoks.
- Oma programmi teksti analüüs, nulliga jagamised, ületäitumised, ujukomaarvude täpsus.

Järgmine ülesanne on avatud testidega ülesanne Eesti Lahtiselt Võistluselt aastast 2015/16:

7.2.2 Ümbermõõt

Informaatikaolümpiaadi ettevalmistuslaagris anti osalejatele lahendada järgmine ülesanne:

Ristkülikute ühendi ümbermõõt

Tasandil on antud `N` ristkülikut, mille servad on koordinaattelgedega paralleelsed. Leida nende ristkülikute ühendi perimeeter ehk välise kontuuri pikkus. Tekkinud kujundi sees olevaid auke ei ole vaja arvesse võtta, kuid ühe ristkülikutest koosneva kujundi augus asuvat teist riskülikutest koosnevat kujundit, mis väliskujundit ei puuduta üheski punktis, tuleb lugeda ikkagi omaette kujundiks.

Sisend: Faili esimesel real on antud ristkülikute arv `N (1≤N≤1000)`. Järgmisel `N` real on igaühel neli reaalarvu `LX`, `LY`, `UX` ja `UY`, kus `LX` ja `LY` on ristküliku alumise vasaku ning `UX` ja `UY` ülemise parema nurga koordinaadid. Koordinaatide väärtused on `0` kuni `1 000 000`, mille esituseks piisab 32-bitisest `float` arvutüübist.

Väljund: Faili ainsale reale väljastada leitud ristkülikute ühendi perimeeter täpsusega kuni kolm kohta pärast koma.

Näide: Sisendfail

```
3
7.0 170.0 99.5 190.0
0.5 100.0 12.0 225.0
10.0 80.0 50.0 110.0
```

Väljundfail:

```
564.0
```

Teie ülesandeks on koostada testid, mis kontrolliksid esitatud lahenduste korrektsust.
Kõik testandmed koondada ühte tekstifaili järgnevalt kirjeldatud formaadis. Faili esimesel real on testide arv `K` (`1≤K≤20`). Järgnevas `K` blokis on bloki esimesel real oodatav tulemus ning selle järel sisendandmed samal kujul kui ülesande kirjelduses. Lahendusena tuleb esitada see tekstifail.

NÄIDE:

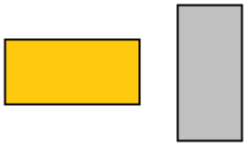
```
2
4.0
1
1.0 1.0 2.0 2.0
564.0
3
7.0 170.0 99.5 190.0
0.5 100.0 12.0 225.0
10.0 80.0 50.0 110.0
```

Ülesande sisu analüüsides tuleks kindlasti teha testid, mis kasutavad minimaalset ja maksimaalset lubatavat ristkülikute arvu, mis tähendab, et võiks olla test, milles on vaid üks ristkülik, ja test, milles ristkülikuid on täpselt tuhat.

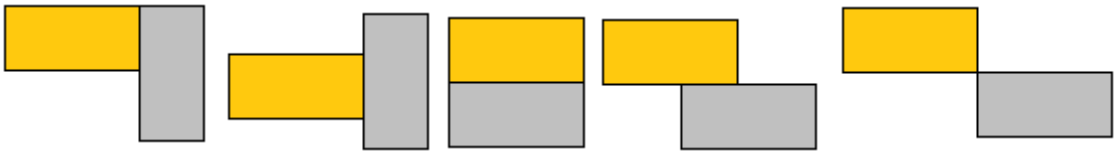
Kuna koordinaatideks võivad olla positiivsed reaalarvud, siis ei piisa testimisel kindlasti ainult täisarvulistest koordinaatidest, vaid tuleks kasutada nii väga väikeseid arve, paljude kohtadega arve, aga ka loomulikult maksimaalset lubatud koordinaadi väärtust.

Väljundi formaadi juures saaks siin kontrollida, kas vastus antakse ikka ettenähtud täpsusega kuni kolm kohta pärast koma. Seda küll raamülesandes ettekirjutatud formaat testida ei võimalda, kuid ise sellist ülesannet lahendades tuleks küll sellele tähelepanu pöörata.

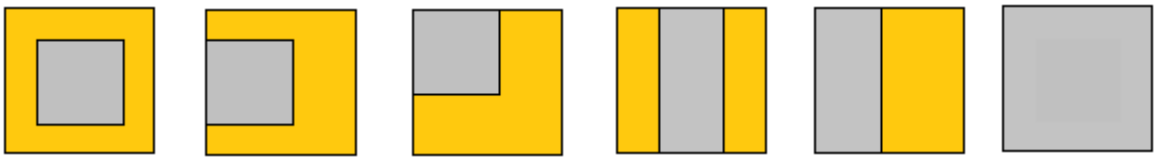
Asume nüüd ülesande sisulise analüüsi juurde. Vaatame kõigepealt, kuidas saavad 2 ristkülikut teineteise suhtes paikneda. Esimene võimalus on, et ristkülikud üldse kokku ei puutu:



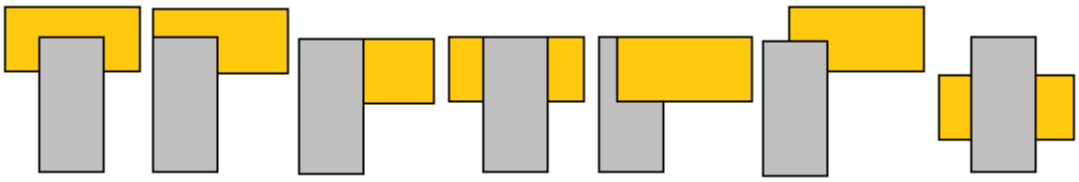
Puutuvatest, kuid mitte kattuvatest ristkülikutest koosnevad erinevad variandid:



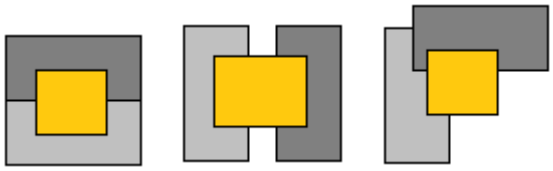
Üks ristkülik võib asuda ka täielikult teise sees. Kui arvestada ka võimalike puutumistega, saame järgmised võimalused (hall ristkülik on tervenisti oranži ristküliku sees):



Lõikuvad ristkülikud:

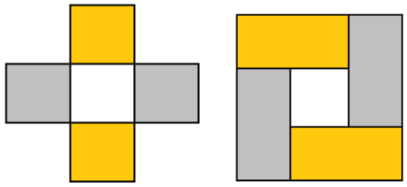


Nii on juba kahe ristküliku omavahelise paiknemise jaoks paarkümmend erinevat võimalust, kolmanda lisamine lisab veel hulga võimalikke kombinatsioone. Kõiki neid ei jõua siin ära tuua. Kolme ristkülikuga võimalustest huvitavamad on järgmised paiknemised:



Vasakpoolsel paikneb kolmas ristkülik tervenisti kahe teise ristküliku sees (aga mitte kummagi sees), parempoolsel on kattumine mõlema teise ristkülikuga, kõik kolm osalevad ümbermõõdu arvestamises, kolmandal on kõigil paarikaupa kattuvaid osi kui ka osa, mille katavad kõik kolm ristkülikut.

Kindlasti ei tohi ära unustada, et ristkülikud võivad moodustada tühje alasid kujundi sees, mis tuleb ümbermõõdu arvestamisel arvesse võtta. Neljast ristkülikust saab näiteks järgmised kujundid:



Kui lisada veel ristkülikuid, saab tekitada ühe kujundi sees ka mitu tühja ala. Kindlasti tasuks testida ka rohkemate kujunditega tekkivaid formatsioone, näiteks spiraali.

Nüüd on siin terve hulk ideid välja käidud, ülesandes lubatud testide arv on aga vaid 20. Mida sellisel juhul teha? Kuna eesmärgiks on ainult viga välja meelitada, siis võib mitu erinevat kombinatsiooni ühte testi panna. Loomulikult tuleb hoolega läbi mõelda, millised on mõttekas ühte testi panna, millised aga võivad nii mõne vea hoopis ära varjata. Ilmselge on, et ei saa ühes testis testida, kas programm töötab nii minimaalse kui ka maksimaalse ristkülikute arvu korral. Samuti enda programmi jaoks teste tehes on parem erinevad juhud eraldi testides hoida, kuna nii saab vea korral kiiremini jälile, millise juhtumiga programm hakkama ei saa.

See näide oli selleks, et tuua välja erinevaid aspekte testide koostamisel ja näidata, kuidas pealtnäha küllaltki lihtsal ülesandel võib olla üsna palju erinevaid juhte, millega on võimalik eksida. Järgevalt on toodud, milles siis ülesande programm tegelikult eksis, mis oleks tulnud lahendajal tuvastada:

- Lahendus ei erista välimist ja sisemist kontuuri.
- Lahendus töötab valesti, kui arvutuste täpsus on suurem kui 1 koht pärast koma.
- Lahendus ei leia kõigi väliste kontuuride perimeetrit, kui väliseid kontuure on rohkem kui üks.
- Lahendus töötab valesti, kui ristkülikute arv on 1000 ja viimane ristkülik osaleb perimeetri moodustamises.
- Lahendus töötab valesti, kui perimeetri pikkus ületab 16-bitilise arvu maksimumväärtust.
- Lahendus töötab valesti, kui leidub vähemalt 2 täpselt samasugust ristkülikut.

- Lahendus töötab valesti, kui leidub vähemalt 3 ristkülikut, mille vertikaalsed küljed on samal x-koordinaadil.
- Lahendus töötab valesti, kui leidub vähemalt 3 ristkülikut, mille horisontaalsed küljed on samal y-koordinaadil.
- Lahendus töötab valesti, kui kontuuris leidub külgede jada, kus on järjest 10 või enam ühesuunalist pööret (spiraal).
- Lahendus töötab valesti, kui üks välimine kontuur asetseb täielikult teise välimise kontuuri sees.

7.2.3 Testide genereerimine

Enamik ülesandeid õnneks nii põhjalikku läbimõtlemist ei vaja, kuid erineva andmestikuga testimine on siiski vajalik. Kui mõned spetsiifilisemad juhud on vaadatud, on mõttekas genereerida ka juhuslikke teste.

Viiendas peatükis oli üheks näiteks ülesanne „Miljonär ja vaeslapsed“:

Dickensi-aegsel Inglismaal elas miljonär Mortimer. Temaga samas linnas asusid kolm lastekodu, kus elasid vaeslapsed, kellele Mortimer tavatses jõulukinke teha. Kinkide jagamise protseduur oli järgmine:
1. Iga vaeslaps saab oma lastekodust korvi, millega ta kingi järele läheb.
2. Mortimer viskab kingitusi järjest laste sekka, mida nood oma korvidega püüavad.
3. Iga kingitus püütakse alati kinni.
4. Lapsed saavad kingitusi kätte juhuslikult, kuid tõenäosus, et konkreetne laps kingituse kätte saab, on võrdeline tema korvisuu pindalaga.
5. Sama lastekodu lastel on sama suurusega korvid.
6. Kui mõni laps saab kingituse kätte, läheb ta sellega kohe lastekodusse tagasi ja rohkem püüdmises ei osale.
7. Lapsi võib olla rohkem kui kingitusi :(
Igal kingitusel on väärtus. Leida iga lastekodu kohta, milline on selle kodu laste poolt saadud kingituste väärtuste keskmine eeldatav summa.
Sisendi esimesel kolmel real on igaühel kaks täisarvu L_i $(0 \le L_i \le 100)$ ja K_i $(1 \le K_i \le 100)$, mis tähistavad reanumbrile vastava lastekodu laste arvu ning korvi pindala. Sisendi neljandal real on kingituste arv N $(1 \le N \le L_1+L_2+L_3)$. Viimasel real on N $(1 \le N \le 1000)$ täisarvu: esimesel neist esimese kingituse väärtus, teisel teise jne.
Väljastada kolm reaalarvu (täpsusega vähemalt $0{,}0001$) kolmel real: iga lastekodu kohta, milline on selle kodu laste poolt saadud kinkide väärtuste keskmine eeldatav summa.

NÄIDE:

```
1 1
1 2
1 2
2
10 20
```

Vastus:

```
6,666667
11,333333
12
```



Selle ülesande juhuslike testide generaator võib olla näiteks selline:

```
int main()
{
    const string NIMI = "test";
    const int TESTE = 20;

    for (int i = 0; i < TESTE; i++) {
        ofstream fail;
        string fnimi = NIMI + to_string(i) + ".txt";
        fail.open(fnimi);
        int lapsi = 0;
        for (int i = 0; i < 3; i++) {
            int la = rand() % 101;
            lapsi += la;
            fail << la << " ";
            fail << 1 + rand() % 100 << endl;
        }

        int N = 1 + rand() % lapsi; // kinkide arv
        fail << N << endl;
        for (int i = 0; i < N - 1; i++) {
            fail << 1 + rand() % 1000 <<" "; //kinkide väärtused
        }
        fail << 1 + rand() % 1000 << endl;
        fail.close();
    }
    return 0;
}
```

Selle programmi väljundiks on 20 faili nimedega test0...test19 juhuslike andmetega, mis vastavad sisendi formaadile. Kui proovida need testid läbi lahendusega, siis saame teada ainult seda, kas me programm töötab etteantud ajalimiidi piires. Selleks, et kontrollida, kas saadud lahendus on ka korrektne, oleks vaja teada õiget lahendust. Siin tuleb appi jõumeetod ehk kõikide variantide läbivaatus:

```
int N;
int* kingid;
int K[3];
int L[3];
double vastus[3] = { 0 };

void jagaKingid(int kingiNr, int korvidePindala, double Tn)
{
    if(kingiNr == N) { //kõik on jagatud
        return;
    }

    for (int i = 0; i < 3; i++) {
        if (L[i] > 0) {
            double tn = Tn * L[i] * K[i] / korvidePindala;
            vastus[i] += kingid[kingiNr] * tn;
            L[i]--;
            jagaKingid(kingiNr + 1, korvidePindala - K[i], tn);
            L[i]++;
        }
    }
}
```

Kui meil just aastaid aega ei ole, siis tuleb aga sisendi limiite oluliselt kärpida, sest selle jõumeetodi keerukus on 3^N ja seega ei ole mõtet ette anda kuigi suurt N-i. Samas, kui me saame identsed

vastused nii DP lahendusega kui ka jõumeetodiga kümnekonnale juhuslikule testile, siis on üldjuhul normaalne eeldada, et testitav DP lahendus annab korrektse tulemuse ka suuremate testide puhul. Eelpool toodud testide genereerimise algoritmis tuleks muuta kinkide arvu leidmise rida vastavalt:

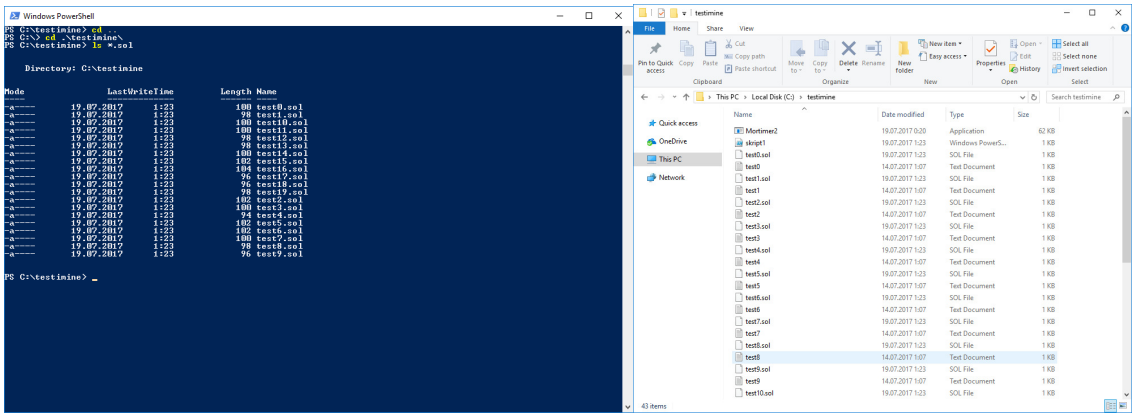
```
int N = 1 + rand() % min(lapsi, 15); // kinkide arv
```

Pane tähele, et laste arvu (ja ammugi mitte kinkide väärtuste) piiramine ei ole vajalik.

7.2.4 Käsuinterpretaatorid ja skriptimine

Kui testid valmis, tuleb need ka läbi jooksutada. Üldine stsenaarium on selline, et iga testi jaoks tuleb käivitada programmid vastava sisendiga ja võrrelda väljundit etaloniga. Kui teste on palju, on kõigi nende käivitamine käsitsi päris aeganõudev ning kui tulemusena tuleb programmis välja mõni viga, mis parandamist tahab, tuleb programm pärast parandust taas üle testida.

Õnneks saab neidki tegevusi automatiseerida, kasutades oskuslikult mõnd **käsuinterpretaatorit**. Käsuinterpretaator on tekstipõhine käsutöötlusprogramm, mille abil saab suhelda operatsioonisüsteemiga. Levinud käsuinterpretaatorid on näiteks Bash (Linux) ja PowerShell (Windows).



PowerShell (käsurealiides) *File Explorer (graafiline liides)*

*Shell*iks ehk **kestprogrammiks** nimetatakse programmi, mis võimaldab inimesel operatsioonisüsteemi funktsionaalsust kasutada (nt programme käivitada). Enamasti peetakse *shell*idest rääkides silmas tekstimoodis käsuinterpretaatoreid, kuid tegelikult on selleks ehitatud graafilised programmid (nt Windows Explorer) samuti kestprogrammid.

Operatsioonisüsteemiga suhtlemiseks kasutatakse **käske** (*commands*, PowerShellis *cmdlets*). Siin on tabel mõningate testide jooksutamiseks põhiliselt vajaminevate käskudega:

Käsk	Bash	PowerShell cmdlet	PowerShelli aliased
Kataloogi sisu	ls	Get-ChildItem	ls, dir, gci
Kataloogi muutmine	cd	Set-Location	cd, chdir, sl
Faili liigutamine	mv	Move-item	mv, move, mi
Faili kopeerimine	cp	Copy-Item	cp, copy, cpi
Faili väljastamine	cat	Get-content	cat, type, gc
Standardvoosse kirjutamine	Echo	Write-Output	echo, write
Failide võrdlemine	diff	Compare-Object	diff
Kellaaeg	time	Get-Date	

Näiteks

```
PS C:\testimine> cp test2.txt sisend.in
```
Kopeerib faili `test2.txt` sisu uude faili nimega `sisend.in`

Käske täidetakse järjest ja igal käsul võib olla sisend ja väljund, mis vaikimisi loetakse ja kirjutatakse standardvoogudesse. Eelmise käsu väljundi saab suunata järgmise käsu sisendiks kasutades käskude vahel märki „|“. Sel moel suunamist nimetatakse inglise keeles ***piping***uks ja nii suunatakse edasi ainult standardväljund.

Vooge on võimalik suunata ka üksteise vahel. Standardväljundi saab suunata faili: ’>’ kirjutab standardväljundi voo faili juurde ja ’>>’ kirjutab faili üle.

Näiteks

```
PS C:\testimine> cat sisend.in | programm.exe >> vastus.out
```
väljastab faili `sisend.in` sisu ja suunab selle programmi `programm` sisendiks ning lõpuks suunab programmi standardväljundi faili `vastus.out`. Kui selle nimega fail on olemas, kirjutatakse selle sisu üle.
Nii Bash kui ka PowerShell toetavad skriptimist. Skript on faili valmis kirjutatud käsk või käskude jada, mida käsuinterpretaator teostab. Lisaks käskudele saab kasutada ka muutujaid, tingimuslauseid ja tsükleid. PowerShelli skriptide laiendiks on .ps1 ja Bashi skriptidel kasutatakse laiendit .sh

PowerShellis muutujate nimed algavad dollari märgiga $, Bashis deklareerimisel $ märki ei kasutata, küll aga muutuja väärtuse kasutamisel, näiteks:

PowerShell	Bash
`$nimi = "Siim Susi"`	`nimi = "Siim Susi"`
`$vanus = 14`	`vanus = 14`
`echo $nimi $vanus`	`echo $nimi $vanus`

Kui Bashis on vaja muutujasse salvestada **käsu tulemus**, tuleb omistamisel käsk panna kas tagurpidi ülakomade vahele või kasutada süntaksit `muutuja = $(käsk)`.

PowerShell	Bash
`if` *`tingimus`* `{`*`teeMidagi`*`}`	`if` *`tingimus`* `then` *`teeMidagi`* `fi`

`foreach ($i in` *`mingiHulk`*`) {`
`  `*`teeMidagi`*
`}`
	`for i in` *`mingiHulk`*
	`do`
	`teeMidagi`
	`done`

`for ($i=0; $i -le 10; $i++) {`
`  `*`teeMidagi`*
`}`
	`for ((i=0; i < 10; i++))`
	`do`
	`teeMidagi`
	`done`

Rohkem infot Bashi ja PowerShelli käskude ja süntaksi kohta leiab näiteks lehelt https://ss64.com/.

Vaatame siin veel paari kõige tõenäolisemat testimise stsenaariumit.

Oletame, et testid on `.in` failides ja nende vastused `.out` failides, näiteks on testfailid `test1.in`, `test2.in` jne, ja vastavate testide vastused on failides `test1.out`, `test2.out` jne. Siis on Bashi testijooksutamisskript järgmine:

```
for f in *.in # iga .in laiendiga failiga
  do bn=`basename $f .in` # bn on ilma laiendita failinimi
  # käivita programmi 'myprog' sisendiga failist f ja suunab väljundi faili,
  # millel on sama nimi, mis failil f ja laiendiks on '.sol'
  cat $f | myprog > $bn.sol
  diff $bn.out $bn.sol # võrdleb näidisvastusega faili programmi lahendusega
done
```

Ja sama PowerShellis:

```
$failid=@(Get-ChildItem .\*.in)
Foreach ($f in $failid) { # iga in laiendiga failiga
  $bn= $f.basename # bn on ilma laiendita failinimi
  # käivita programmi 'myprog' sisendiga failist f ja suunab väljundi faili,
  # millel on sama nimi, mis failil f ja laiendiks on '.sol'
  $uusnimi = $bn + ".sol"
  cat $f | .\myprog.exe > $uusnimi
  # võrdleb näidisvastusega faili programmi lahendusega
  diff (cat $bn.out) (cat $bn.sol) #kuna PowerShell võrdleb vaikimisi objekte,
  # siis faili sisu võrdlemiseks tuleb anda argumentideks failide sisud
}
```

Kui sisend ja väljund tuleb lugeda/kirjutata etteantud nimega faili, olgu need näiteks `ylesanne.sis` ja `ylesanne.val` on skript järgmine:

```
for f in *.in
do
  cp $f ylesanne.in # kopeerib faili f faili 'ylesanne.in'
  myprog # käivitab programmi. Õige sisend on olemas eelmisest käsust,
  # tekib väljund 'ylesanne.val'
  bn=`basename $f .in`
  diff $bn.out ylesanne.val
done
```

Ja sama PowerShellis:

```
$failid=@(Get-ChildItem .\*.in)
Foreach ($f in $failid) {
  cp $f ylesanne.in # kopeerib faili f faili 'ylesanne.in'
  ./myprog.exe # käivitab programmi. Õige sisend on olemas eelmisest käsust,
  # tekib väljund 'ylesanne.val'
  $un= $f.basename + ".out"
  diff (cat $un) (cat $ ylesanne.val)
```

7.2.5 Valideerimine

Kuna `diff` võrdleb faile märk-märgilt, siis mõnel juhul on kasulikum kirjutada oma validaator, mis vastuseid võrdleb. Üheks selliseks juhuks on muidugi see, kui vastuseks tuleb anda reaalarv, nagu „Miljonäri ja vaeslaste“ ülesandes.

```
int main(int argc, char* argv[])
{
    ifstream fail1, fail2;
    fail1.open(argv[1], ifstream::in);
    fail2.open(argv[2], ifstream::in);
    double delta = 0.0001;
    double vastus1, vastus2=0;
    for (int i = 0; i < 3; i++){
        fail1 >> vastus1;
        fail2 >> vastus2;
        if (abs(vastus1 - vastus2) > delta) {
            cout << "vale vastus" << endl;
            break;
        }
    }

    fail1.close();
    fail2.close();
    return 0;
}
```

Reaalarvud ja nende täpsus ei ole aga sugugi ainus koht, kus oma validaatorit vaja võib minna. Siin on veel mõned juhtumid:

- Vastus koosneb mitmest täisarvust või stringist, aga nende järjestus ei ole üheselt määratud. Sellisel juhul peaks validaator enne võrdlemist mõlemad vastused sorteerima sama kriteeriumi järgi.
- Ülesandel võib olla mitu õiget vastust. Üheks võimaluseks on etalonvastusena anda kõik võimalikud vastused ja validaator kontrollib, kas programmi vastus sobib ühega neist.

7.3 TUNNE OMA KEELT

Hoolimata sellest, et programmeerimiskeeltel on palju sarnaseid omadusi ja võimalusi, on konkreetsetel keeltel ka spetsiifilisi võimalusi, mille tundmine suureks kasuks võib tulla – just nagu osav käsitöömeister suudab oma tööriistadega rohkem korda saata kui algaja.

7.3.1 Teegid

Suur osa programmeerijale suurepärastest abivahenditest asub keele standardteekides, nende võimaluste tundmine teeb programmi kirjutamise kiiremaks ja efektiivsemaks, kuna sageli päästab jalgratta leiutamisest või hoiab juhendi lugemisele kuluvat aega kokku.

Keelte võimalused on siinkohal erinevad ning igal keelel on omad tugevused ja nõrkused, mistõttu on hea tunda rohkem kui üht keelt. Kuigi üldjuhul on C++ võistlustel eelistatud, on ülesandeid, kus Java või Pythoni kasutamine on optimaalsem. Arvuteooria peatükis tuli suurte arvudega arvutamise juures C++ keeles kirjutada hulk funktsioone, mis Java `BigInteger` klassis olemas on, ning isegi Python suudab kiiresti arvutada suurte arvudega ilma, et programmeerija omalt poolt midagi erilist tegema peaks. Samuti võib C++ tekstitöötlus olla ebamõistlikult keeruline.

7.3.2 Andmestruktuurid

Andmestruktuurid on väga võimsad asjad, aga seda siis, kui neid õigel ajal õiges kohas kasutada. Klasside ja kirjete loomine on tore ning sageli aitab kaasa koodi loetavusele, samas ei maksa unustada ka võimalikke lihtsamaid alternatiive. Enne kirje loomist aga tasub läbi mõelda, kas see tegelikult ka midagi juurde annab võrreldes sama info hoidmisega mitmemõõtmelises või mitmes

massiivis, seda muidugi eriti C/C++ kasutades. Näiteks kui ülesandes on vaja töödelda punktihulka, siis esimese mõttena võib tulla pähe luua kirje `Punkt` muutujatega `X` ja `Y`, kuid see ei pruugi anda eelist kahe massiivi `X` ja `Y` ees.

7.3.3 Makrod

Võrreldes enamiku tänapäeval kasutatavate keeltega on C/C++ keelel järgmine huvitav omadus: teksti **eeltöötlus** ehk preprotsessimine. Java, Python ja enamik teisi tänapäeva keeli võimaldavad koodi pakendamist moodulitesse, mille kohta on teada, milline funktsionaalsus selles realiseeritud on. Kui nüüd teises programmis on vaja seda moodulit kasutada, viidatakse sellele nime alusel, `import` käsu abil. C/C++ puhul on selle aseme **päisfailid**, mis sisaldavad kasutatavate funktsioonide deklaratsioone ja mille tekst lisatakse `#include` käsu abil otse kirjutatava programmi sisse - preprotsessor lihtsalt **asendab** `#include` käsu vastava faili sisuga. Selliseid tekstioperatsioone nimetataksegi eeltöötluseks – selle tulemusena valmib lõplik programmi tekst, mida kompilaator tegelikult kompileerima hakkab.

Eeltöötluse käigus saab teha ka mitmeid muid tekstioperatsioone, näiteks defineerida, et mõni lekseem peab tähendama hoopis midagi muud. Selliseid asendusi nimetatakse **makrodeks.**

Mõned lihtsad makrod:

```
#define PI 3.14159
```

asendab kõik kohad koodis, kus kasutatakse konstanti `PI` vastava arvuga.

```
#define ll long long
```

võimaldab selle asemel, et igal pool kirjutada välja tüübinimi `long long` kasutada lühemat vormi „`ll`“.

Makrod võimaldavad ka avaldiste kasutamist, näiteks nii:

```
#define ruut(x) x*x
```

Selles näites on aga ohtlik viga, mis illustreerib makro ja tavalise funktsiooni erinevust! Mis juhtub, kui kirjutada näiteks:

```
int a=2, b=3;
cout << ruut(a+b);
```

Võiks arvata, et vastus on 25, kuid tegelikult asendatakse `ruut(a+b)` avaldisega `a+b*a+b`, mis annab tulemuseks hoopis 2+3*2+3 = 11. Sellise probleemi vältimiseks tuleb argumendid panna sulgudesse:

```
#define ruut(x) (x)*(x)
```

Makrod on võimas vahend, mida on kerge kuritarvitada, sest põhimõtteliselt võib kirjutada ka

```
#define 1 0
```

ja teisi hullumeelsusi. Suurem oht seisneb siiski selles, et makrode abil võib kirjutada väga keerulist ja arusaamatut koodi, mida on raske hallata ning debugida. Pole juhus, et „segase koodi võistlusi“ (*obfuscated code contest,* http://www.ioccc.org/) korraldatakse peamiselt C keeles, kus võistlustööd kasutavad ohtralt makrosid.

Võistlustel seisneb makrode kasutamise peamine väärtus võimaluses korduvaid asju lühemalt kirja panna, siin on mõned praktilised näited:

```
#define PB push_back
#define MP make_pair
#define sz(v) (in((v).size()))
#define forn(i,n) for(in i=0;i<(n);++i)
#define forv(i,v) forn(i,sz(v))
#define fors(i,s) for(auto i=(s).begin();i!=(s).end();++i)
#define all(v) (v).begin(),(v).end()
```

7.4 VÕISTLUSTE STRATEEGIA

Tänapäeval toimub palju erinevas formaadis võistlusi mitmesuguste raskusastmete, ajapiirangute ja testimissüsteemidega, mistõttu on ka võistlemise strateegiad erinevad.

Siin on siiski mõned üldised näpunäited. Peamine reegel on „parem pronks peos kui kuld katusel“ – liiga ambitsioonikas kohe raskete ülesannete kallale asumine on nii mõnelegi võistlejale tulemuse maksnud.

- Kui ülesannete eeldatav raskusjärjestus pole teada (nt IOI, kus kõik ülesanded annavad sama palju punkte), **loe enne lahendama asumist kõik ülesanded korralikult läbi**. Võib juhtuda, et kuigi esimene ülesanne tundub huvitav või lahendatav, on järgmine ülesanne veel kasulikum.
- **Maanda riske** ehk parem varblane peos kui tuvi katusel. Ära võta esimeseks ülesannet, mille osas on risk, et lahendus võib olla ühelt poolt keeruline kirjutada ja teiselt poolt kätkeb endas riski, et lahendus pole ka üldse õige. Halvimal juhul kulutad kogu võistluse aja sellele ülesandele, saad selle eest lõpuks nulli ning ei saa ka teiste ülesannete eest midagi.
- Kui sa arvad, et oskad mingit ülesannet lahendada, siis **ära jäta seda viimasele tunnile**, võttes enne ette mõne raskema ülesande, vaid lahenda pigem kohe ära. Sageli juhtub, et su esialgne idee polnud päris korrektne ja tegelikult kulub lahendusele oluliselt rohkem aega.
- Kui tegu on alamülesannetega võistlusega (nt IOI) ja tundub, et väiksema alamülesande lahendus võiks viia suurema lahenduseni, tee väiksem kiiresti ära ja esita see. Sellel on mitu eelist:
    - annab kinnituse, et oled algoritmiliselt õigel teel,
    - kui su lahenduses on vigu, mille tõttu mõni test läbi ei lähe, saad sellest kohe teada,
    - nüüd on kindlalt mingid punktid käes ja pole riski üldse ilma jääda.
- Kasuta aega efektiivselt. Kuigi võistlustel on enamasti 4-5 tundi aega, kulub see üllatavalt kiirelt. Tee ka lihtsamad ülesanded ära võimalikult kiiresti, säästetud kümme minutit on tihti just see, mis jääb puudu viimase vea leidmisest võistluse lõpus.

7.5 AHNED ALGORITMID



Et peatükk liiga teoreetiline poleks, uurime mõnd praktilist ülesannet. Enamik selle raamatu peatükke käsitlevad spetsiifilisi algoritme ja ülesandeid, mille lahendamiseks vastavad algoritmid hädavajalikud on. Samas on palju ülesandeid, mida on võimalik lahendada „ahnelt“, lihtsalt ühest spetsiifilisest kohast pihta hakates.

Ahne algoritm on selline, kus igal sammul tehakse parasjagu kõige optimaalsem valik, lootuses, et see viib optimaalse lõpptulemuseni. Kui selline lähenemine töötab, siis lahendus on kiire ja lühike, seega

väga efektiivne. Keerulisem osa seisneb pigem selliste ülesannete ära tundmises ja tõestamises, et ahne algoritm annab tõepoolest korrektse tulemuse. Ahne algoritmiga lahenduva probleemi ära tundmisel on abiks veel järgmine omadus: sellisel probleemil on optimaalsed alamstruktuurid. See tähendab, et optimaalne lahendus kogu ülesandele sisaldab optimaalseid lahendusi alamülesannetele. Eelmises peatükis käsitletud Dijkstra algoritm on tegelikult ahne algoritm – iga tippu vaadeldakse täpselt üks kord ja valitakse sellesse minemiseks hetkel parim tee.

Järgmiseks on toodud kolm klassikalist ahne algoritmiga lahenduvat ülesannet.

7.5.1 Mündid

Sul on vaja programmeerida müügiautomaadi jaoks raha tagastamise funktsioon. Automaat võtab vastu sularaha maksimaalselt kuni 10 eurot ja tagastab ainult münte (1-, 2-, 5-, 10-, 20- ja 50-sendiseid ja 1- ja 2- euroseid). Klientidele meeldib, kui nad saavad tagasi võimalikult vähe münte. Sinu funktsioon peab tagastama automaadi väljastatavad mündid. Võib eeldada, et münte on automaadis alati piisavalt.
Sisendi ühel ja ainsal real on üks täisarv: tagastatav summa. Väljastada 8 tühikuga eraldatud täisarvu, milles esimene näitab, mitu 1-sendist automaat peab tagastama, teine mitu 2-sendist jne. Viimane arv näitab, mitu kahe eurost tuleb tagastada.
NÄIDE 1:
`33`
Vastus:
`0 0 0 1 1 0 1 1` (20-, 10-, 2- ja 1-sendine)
NÄIDE 2:
`999`
Vastus:
`4 1 1 2 0 1 2 0`

Sellest ülesandest oli juba põgusalt juttu 5. peatükis punktis 5.1.2 Ahne algoritm. Siin siis ahne lahendus, mis on nii lihtne, et vaevalt sobib siia, edasijõudnute osasse ning ei vaja seetõttu ka pikemat selgitust:

```
int main()
{
    int myndid[8] = { 200, 100, 50, 20, 10, 5, 2, 1 };
    int vastus[8] = { 0 };
    int summa;
    cin >> summa;

    for (int i = 0; i < 8; i++) {
        while (summa >= myndid[i]){
            summa -= myndid[i];
            vastus[i]++;
        }
        if (summa == 0) break;
    }

    for (int i = 0; i < 8; i++) {
        cout << vastus[i] << ' ';
    }
    cout << endl;
    return 0;
}
```

Oluline on siiski tähele panna, et müntide väärtused on kahanevas järjekorras, kuna kõigepealt tuleb väljastada kõige suuremat münti nii palju, kui antud summasse mahub, seejärel suuruselt järgmist jne. Seega, kui müntide väärtused oleks juhuslikult ette antud, tuleb need kindlasti sorteerida (või otsida iga kord suuruselt järgmist väärtust).

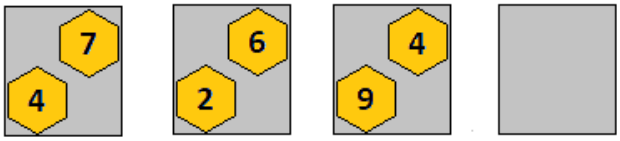
7.5.2 Kingid

Lastekodule on annetatud jõuludeks hulk mänguasju, millest tuleb lastele jõulupakid kokku panna. Igas pakis võib olla kuni kaks mänguasja ning need peavad olema komplekteeritud nii, et pakkide väärtused oleks võimalikult sarnased, st et summa iga paki väärtuse ja pakkide keskmise väärtuse vahel oleks võimalikult väike. Pakk võib olla ka tühi.
Sisendi esimesel real on kaks arvu: laste arv lastekodus `L` ja mänguasjade arv `M` (`L≤M≤2L`). Teisel real on `M` positiivset täisarvu: esimene on esimese mänguasja väärtus, teine teise oma jne. Väljastada üks arv: pakkide väärtuste keskmisest erinevuse minimaalne summa.
NÄIDE:

```
4 6
4 7 2 6 9 4
```

Vastus:
2 (pakid on asjadega väärtustega 4 ja 4 (vahe 0); 2 ja 6 (vahe 0); 9 (vahe 1)ja 7 (vahe 1))

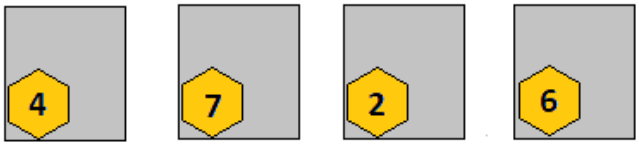
Selle ülesande puhul on veidi keerulisem läbi närida, kuidas ahne lähenemine töötada võiks. Vaatame esimest näidet nelja paki ja 6 mänguasjaga. Proovime kõigepealt lihtsalt jagada esimesse pakki 2 asja, teise 2 jne. Saame jaotuse:



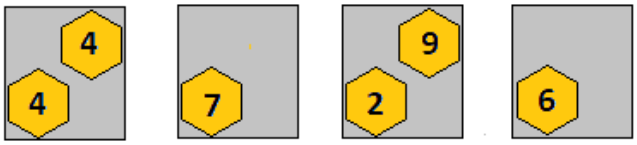
Esimese paki väärtus on 11, teise oma 8, kolmandal 13 ja viimasel 0. Kõigi pakkide keskmine väärtus on $(11 + 8 + 13 + 0) : 4 = 32 : 4 = 8$. Seega on praegu väärtuste vahede summa $|11-8| + |8-8| + |13-8| + |0-8| = 3 + 0 + 5 + 8 = 16$.

Kui tõsta mõnest pakist, kus on 2 asja, üks ese tühja pakki, siis on kaks võimalust: kui tõsta ese sellisest pakist, mille väärtus on kõrgem kui keskmine, siis väärtuste vahede summa paraneb (näiteks tõstes esimesest pakist eseme väärtusega 7 viimasesse pakki); kui tõsta ese ümber sellisest pakist, mille väärtus on võrdne või madalam keskmisest, siis väärtuste vahede summa jääb samaks (näiteks tõstes teisest kastist eseme väärtusega 6 viimasesse kasti). Veendu selles ise!

Kokkuvõtvalt: mõne kasti tühjaks jätmisega pakkide väärtuste vahede summat ei paranda, pigem vastupidi. Proovime siis jaotada kõige pealt igasse kasti täpselt ühe kingituse:

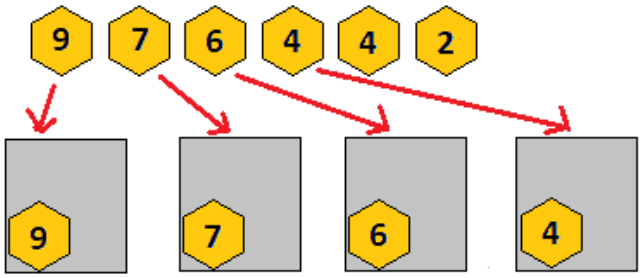


Vaja on veel lisada kingitused väärtustega 9 ja 4. Parima jaotuse saamiseks tuleks lisada kõige suurema väärtusega kingitus kõige väiksema väärtusega pakki, saame jaotuse:

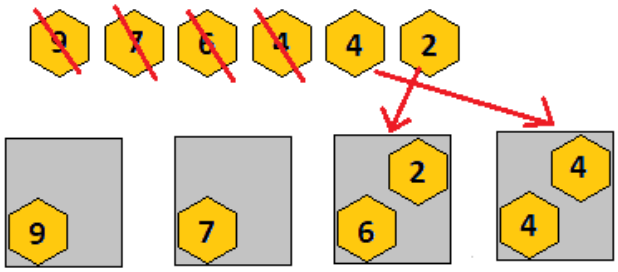


Siin on pakkide väärtuste keskmisest erinevuste summa $|8-8|+|7-8|+|11-8|+|6-8|=6$. Aga seegi ei ole veel vähim võimalik.

Ma arvan, et nii mõnigi lugeja on juba taibanud, et esimeste asjade pakkidesse jaotus võiks juba olla ühtlasem, st esimesena tuleks panna kõige suurema väärtustega esemed ning seejärel lisada ülejäänud esemed, nagu kirjeldatud. Tõesti, kui sorteerime mänguasjad väärtuse järgi, läheb pilt kohe selgemaks:



Ning viimased tuleb jaotada täpselt vastupidises järjekorras, st alustades viimasest pakist:



Nii on vastuseks $|9-8|+|7-8|+|8-8|+|8-8|=2$. See ongi parim jaotus.

Programmis on sageli lihtsam oletada, et igasse pakki läheb täpselt 2 kingitust. Selleks lisame nii palju 0-väärtusega asju, et mänguasjade arv oleks täpselt kahekordne pakkide arv. Nii saame lihtsa vaevaga pakid kohe komplekteerida, pannes esimesse pakki kõige suurema ja väiksema väärtusega eseme, teise teise kõige suurema ja teise kõige väiksema eseme jne, kuni viimasesse pakki jääb kaks keskmise väärtusega eset. Siin on ülesande lahendus:

```
int main()
{
    int L, M, summa = 0;
    cin >> L >> M;

    int* vaartused = new int[2*L];
    int i = 0;
    for (i; i < M; i++) {
        summa += vaartused[i];
    }

    for (i; i < 2*L; i++) {
        vaartused[i] = 0;
    }

    sort(vaartused, vaartused + 2*L);
    double keskmine = double(summa / L);
    double vahe = 0;
    for (i = 0; i < L; i++){
        vahe += abs(keskmine - vaartused[i] - vaartused[2 * L - i - 1]);
    }

    cout << vahe;
    return 0;
}
```

Seda tüüpi ülesandeid nimetatakse **tasakaalustamise** (*load balancing*) ülesanneteks. Järgnevaks veel üks ahne algoritmi tüüpülesanne: **lõikude katmine** (*interval covering*).

7.5.3 Majakavahid

Kaugel-kaugel merel asub üksik laid. Laiul on majakas ja seal peab olema kohal majakavaht. Iga aasta detsembris annavad potentsiaalsed majakavahid teada perioodid, millal nemad saaksid majakas olla. Kuna inimeste viimine laiule ja sealt tagasi on kallis, siis soovitakse, et vahetusi oleks võimalikult vähe. Kirjuta programm, mis leiab minimaalse vahetuste arvu. Esimene vahetus alustab alati 1. jaanuaril ja viimane lõpetab 31. detsembril. Aastas on alati 365 päeva. Kui üks vaht saab kohal olla kuni 3. aprill ja teine saab alustada 4. aprillil, siis saab need vahid järjest vahetada, ilma, et majakas valveta jääks.
Sisendi esimesel real on üks arv: majakavahtide arv kokku. Järgnevalt on rida majakavahi numbriga ja tema vahetuste arvuga V. Järgneval V real on vahetuse algusaeg ja lõppaeg kujul pp.kk, eraldatud tühikuga. Seejärel on järgneva majakavahi number ja tema vahetuste kuupäevad, jne.
Väljastada üks number: vahetuste arv. Kui jääb katmata päevi, kus ükski vaht majakas olla ei saa, väljastada „EI SAA“.

NÄIDE 1:

```
4
1 3
04.04 08.07
09.09 10.10
11.11 31.12
2 3
01.01 20.02
03.03 05.05
11.10 10.11
3 2
01.01 03.03
01.07 01.10
4 2
01.08 10.10
12.11 31.12
```

Vastus:

```
7
```

(1.1-3.3, 3.3-5.5, 4.4-8.7, 1.7-1.10, 1.8-10.10, 11.10-10.11, 11.11-31.12)

NÄIDE 2:

```
2
1 1
01.01 01.06
2 1
01.06 30.12
```

Vastus:

```
EI SAA
```



Paldiski tuletorn

Kindel on see, et esimese vahetuse majakavaht peab alustama 1. jaanuaril. Kui sel päeval saab alustada vaid üks majakavaht, tuleb tema esimesena laiule toimetada. Kui aga see päev sobib mitmele vahile, siis milline neist valida? Kuna meid huvitab ainult minimaalne vahetuste arv, siis valime ahnelt selle, kelle vahetus lõpeb kõige hiljem. Ülejäänud vahetused, mis saaksid alustada 1. jaanuaril, võime täiesti kõrvale heita, kuna meie valik katab need täielikult. Mis aga valida järgmiseks? Teame, et algusperiood peab kattuma või vahetult järgnema esimese vahetuse lõpule. Taas on hea valida sobiva algusega vahetuste seast see, mis lõppeb kõige hiljem. Nüüd taas võib kõik sobiva algusperioodiga vahetused, mille seast enne valisime, kõrvale jätta. Ja nii edasi, kuni aasta

lõpp on käes. Kui mingil etapil sobiva algusajaga vahetust polegi, siis kattumata vahetusi tekitada ei saa.

```
#include <iostream>
#include <vector>
#include <sstream>
#include <string>
#include <algorithm>
using namespace std;

// antud kuule eelnev päevade arv. Jaanuarile eelneb 0 päeva, veebruarile 31, märtsile
// 31+28 jne
const int kuud[12] = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 };

int leiaPaevaNr(string kuupaev) {
    int k = kuupaev.find('.');
    int kuu, paev;
    paev = atoi(kuupaev.substr(0, k).c_str());
    kuu = atoi(kuupaev.substr(k+1, kuupaev.length()).c_str());
    return kuud[kuu-1] + paev;
}

int main()
{
    int M, vahetusi = 0;
    vector<pair<int, int>> vahetused;
    cin >> M;
    while (M--){
        int N, V;
        cin >> N >> V;
        while (V--){
            string algus, lopp;
            cin >> algus >> lopp;
            vahetused.push_back(make_pair(leiaPaevaNr(algus), leiaPaevaNr(lopp)));
        }
    }

    sort(vahetused.begin(), vahetused.end()); // sorteerime alguse järgi kasvavalt
    int esimenePaev = 1, viimanePaev = 0, vahetusiKokku = 0;
    int i = 0;

    while (viimanePaev < 365) { // Vahetused on leitud
        // ei ole jõudnud aasta lõppu, aga rohkem vahetusi pole või
        // järgmise vaatamata vahetuse esimene päev on liiga kaugel
        if (i == vahetused.size() || vahetused[i].first > esimenePaev) {
            cout << "EI SAA" << endl;
            return 0;
        }
        for (i; i < vahetused.size() && vahetused[i].first <= esimenePaev; i++) {
            if (vahetused[i].second > viimanePaev) { // parem viimane päev
                viimanePaev = vahetused[i].second;
            }
        }
        vahetusiKokku++; // oleme leidnud parima lõpp-päeva
        esimenePaev = viimanePaev + 1;
    }

    cout << vahetusiKokku << endl;
    return 0;
}
```

7.5.4 Veel ahnetest algoritmidest

Nagu juba mainitud, siis ahne lähenemise juures kõige raskem on tõestada, et see viib tõesti korrektse tulemuseni. Need kolm klassikalist näidet tasub siiski meelde jätta ja harjutada nende ära tundmist.

Ahned algoritmid üldiselt eeldavad, et andmed on sorteeritud. Mündiülesandes olid mündid antud juba sorteeritud järjekorras, kinkide ja majakavahtide ülesannetes tuli andmed ise sorteerida. Sageli aitab andmete eelnev sorteerimine kaasa sobiva ahne algoritmi leidmisele. Seega võistlusel, kui hea lahendus kohe pähe ei tule, on kasulik andmed sorteerida ning vaadata, kas sellest võiks mingit abi olla.

Kuna need lahendused on ülilihtsad programmeerida ja töötavad imekiiresti, siis võistlustel selliseid klassikalisi ahneid ülesandeid sageli ei kohta. Tundmatu ülesande lahendamine ahne meetodiga on küllaltki riskantne, tõestamine, et ahne algoritm korrektse tulemuse annab, on ajamahukas. Seega, kui ülesande andmete maht on selline, et variantide läbivaatus või dünaamiline planeerimine leiab vastuse etteantud ajalimiiti jäädes, on need lähenemised kindlamad, kuna leiavad kindlasti korrektse vastuse. Teisalt aga võivad kavalamad ülesande koostajad panna meelega madalamad piirid andmetele, et lahendajad kohe ahne algoritmi kasutamise võimalusele ei mõtleks.

Ahned on ka mitmed kavalad nimelised algoritmid. Eespool oli juba juttu, et Dijkstra algoritm lühima tee leidmiseks graafis on oma olemuselt ahne. Üheksandas peatükis tutvustatakse ka Kruskali ja Primi algoritme minimaalse toesepuu leidmiseks, mis on samuti ahned. Tuntud Huffmani pakkimisalgoritm kasutab samuti ahnet lähenemist.

Järgmiseks aga üks keerukam ülesanne Rahvusvaheliselt informaatikaolümpiaadilt aastast 2015 (http://ioi2015.kz/).

7.5.5 Suveniirid

Käimas on IOI 2015 avatseremoonia lõpuakt. Tseremoonia käigus pidi iga võistkond saama endale suveniiri. Kahjuks olid aga kõik korraldajad tseremoonia poolt nii võlutud, et unustasid need välja jagada. Ainus, kes suveniiridest midagi mäletab, on Aman. Tema on entusiastlik korraldaja ning tahab, et IOI-l oleks kõik ideaalne, seega üritab ta jagada kõik suveniirid minimaalse ajaga.
Avatseremoonia toimumiskohaks on ringikujuline hoone, mis on jagatud `L` sektsiooniks. Sektsioonid on nummerdatud järjest `0` kuni `L-1`-ni. Seega, iga `0≤i≤L-2` puhul on sektsioonid `i` ja `i+1` naabrid ning sektsioonid `L-1` ja `0` omavahel naabrid. Kokku on `N` võistkonda. Iga võistkond istub ühes sektsioonidest. Igas sektsioonis võib asuda suvaline arv võistkondi. Mõned sektsioonid võivad olla ka tühjad.
Jaotamiseks on `N` identset suveniiri. Algselt asub nii Aman kui ka kõik suveniirid sektsioonis `0`. Amanil tuleb igale võistkonnale kätte toimetada üks suveniir ning pärast viimase suveniiri äraandmist peab ta tagasi jõudma sektsiooni `0`. Pange tähele, et mõned võistkonnad võivad istuda ka sektsioonis `0`.
Igal hetkel saab Aman kaasas kanda maksimaalselt `K` suveniiri. Aman võtab suveniirid kaasa sektsioonis `0`, ning selleks ei kulu tal aega. Igat suveniiri tuleb kaasas kanda, kuni ta on mõnele võistkonnale kättetoimetatud. Kui Amanil on käes vähemalt üks suveniir ning ta jõuab mõne sektsioonini, kus istub võistkond, kes pole veel suveniiri saanud, võib ta sellele võistkonnale ühe kaasaskantud suveniiridest üle anda. Üleandmine juhtub ka momentaalselt ning pole keelatud ühes sektsioonis mitmele võistkonnale korraga suveniire jagada. Ainus asi, mille peale aeg kulub, on liikumine. Aman võib liikuda mööda ringikujulist hoonet mõlemas suunas. Liikumine kõrvalasuvasse sektsiooni (nii päri- kui ka vastupäeva suunas) võtab täpselt ühe sekundi, sõltumata sellest, kui palju suveniire on tal parajasti kaasas.

Teie ülesanne on leida minimaalne sekundite arv, mille jooksul Aman jõuab ära jagada kõik suveniirid ning naasta tagasi algsesse positsiooni.
Sisendi esimesel real on kolm arvu `N` ($1 \le N \le 10^7$), `K` ($1 \le K \le N$) ja `L` ($1 \le L \le 10^9$). Teisel real on `N` arvu: esimesena esimese meeskonna sektor, teisena teise meeskonna oma jne.

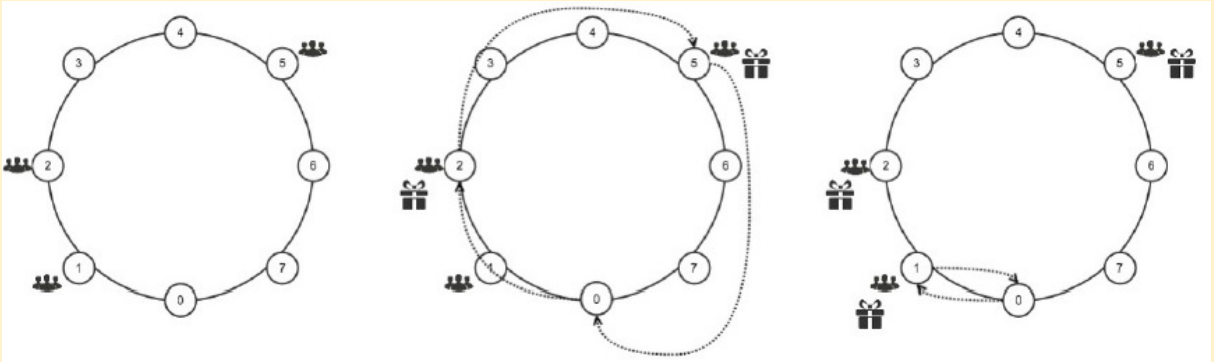
NÄIDE:

```
3 2 8
1 2 5
```

Vastus:

```
10
```

Üks võimalik optimaalne lahendus (näidatud pildil) on järgmine. Alguses võtab Aman kaasa kaks suveniiri, toimetab ühe sektsioonis `2` asuvale võistkonnale, teise sektsioonis `5` asuvale võistkonnale ning tuleb tagasi sektsiooni `0`. See käik võtab tal `8` sekundit. Teisel käigul toob Aman alles jäänud suveniiri sektsioonis `1` istuvale võistkonnale ning tuleb tagasi sektsiooni `0`. Tal kulub selle peale veel `2` sekundit. Koguaeg on seega `10` sekundit

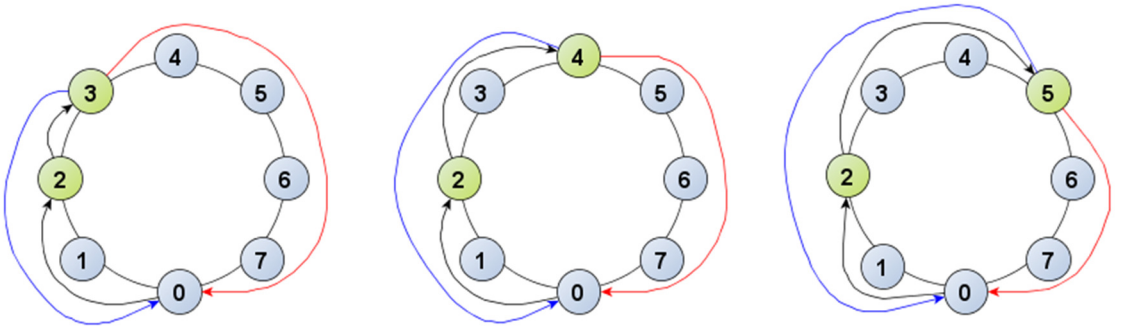


0-sektorist on võimalik suveniire viia teistesse sektoritesse liikudes kas päripäeva või vastupäeva. Ühe sületäie suveniiride jaotamiseks on kolm võimalust:

1. Viime suveniirid päripäeva liikudes ja tuleme tagasi vastupäeva
2. Viime suveniirid vastupäeva liikudes ja tuleme tagasi päripäeva
3. Viime suveniirid, tehes terve ringi. Tee pikkuse mõttes ei ole vahet, kas ring teha päri- või vastupäeva.

Ilmne on, et ühel käigul ei ole mõttekas teha rohkem kui üks täisring. Edasi-tagasi liikudes on mõtet minna maksimaalselt sektsioonini, kus suveniiri jagamine toimub. Edasi-tagasi liikudes ei ole tegelikult oluline, kas suveniiride jagamine soovitud sektsioonides toimub minnes või tulles, seega võib eeldada, et jagamine toimub alati minnes, samuti nagu ringiratast jagades.

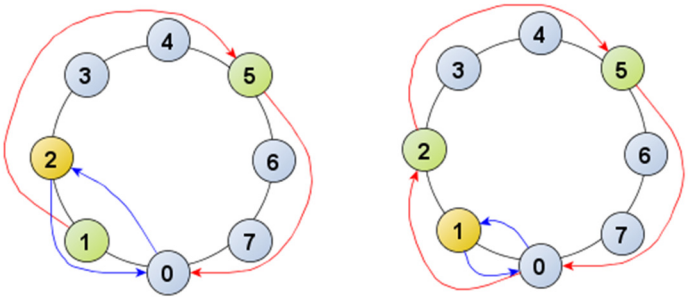
Oluline on tähele panna ka, et optimaalse jagamise korral ühe sületäie jagamisele ei saa kuluda rohkem kui L sekundit. Kui liigume ühe ringi, siis kulub selleks täpselt L sekundit. St kui me liigume päri- või vastupäeva jagades kaugemale kui poole ringi peale, on optimaalsem tagasi minemise asemel teha täisring. Seega päripäeva jagades on mõtet viia suveniire kuni sektsioonini L/2. Sama kehtib ka vastupäeva jagades:



Vasakpoolsel joonisel viiakse kaks pakki sektsioonidesse 2 ja 3. Seejärel on tagasi 0-sektsiooni liikumiseks 2 võimalust: päripäeva (punane nool) ja vastupäeva (sinine nool). Kasulikum on tagasi liikuda vastupäeva, nii kulub 2 paki jagamiseks 6 sekundit, vastupäeva kuluks 8 sekundit. Keskmisel joonisel viiakse kaks pakki sektsioonidesse 2 ja 4 ning seejärel liigutakse tagasi 0-sektsiooni kas päri- või vastupäeva. Mõlemad teekonnad on sama pikad ja võtavad 8 sekundit. Parempoolsel joonisel viiakse pakid sektoritesse 2 ja 5 ning liigutakse samuti edasi kas päri – või vastupäeva. Sellel juhul kulub päripäeva jätkates vähem aega: 8 sekundit (vs 10 sekundit).

Järgmiseks tähelepanekuks on, et optimaalsel lahendusel saame jagada suveniirid järjestikustes sektsioonides olevatele meeskondadele. Näites on mõttekas jagada ühel käigul suveniirid meeskondadele sektsioonides 1 ja 2 või 2 ja 5, aga mitte 1 ja 8 (liikudes teisest sektsioonist jagamata üle). Tõesti, ringiratast liikudes, kui jätame mõne sektsiooni vahele, siis saame selle vahetada sektsiooni vastu, mis on sektsioonile 0 lähemal. Ringiratast jagamise ajakulu see ei muuda, kuid 0-sektsioonile lähemal olevasse sektorisse on kiirem kui edasi-tagasi viimine.

Samuti edasi-tagasi käikudel tasub kogu sületäis viia korraga järjestikustesse sektsioonidesse, sest muidu saame samamoodi vahetada mõne kaugemal asuva meeskonna ja lähemal asuva meeskonna suveniiride jaotuse ja seega potentsiaalselt ainult vähendada jagamisele kuluvat aega.



Vasakpoolsel joonisel viiakse esmalt pakid sektsioonidesse 1 ja 5 (punase noolega näidatud teed) ja seejärel viimane pakk sektorisse 2 (sinised nooled). Kokku kulub 8 + 4 = 12 sekundit. Parempoolsel joonisel viiakse pakid esmalt sektsioonidesse 2 ja 5 ning seejärel viimane pakk sektsiooni 1. Kokku kulub 8 + 2 = 10 sekundit.

Veelgi huvitavam: meil ei ole optimaalse lahenduse korral vaja teha rohkem kui üht täisringi ning täisringil tuleks jagada võimalikult palju suveniire, st K suveniiri või kõik suveniirid, kui K ≥ N.

Teine pool sellest väitest on lihtsasti mõistetav. Kuna täisringi tegemine võtab L minutit sõltumata sellest, kui palju suveniire sellel jagatakse, siis on alati võimalik mõni edasi-tagasi käigul jagatav suveniir jagada hoopis ringiratast liikudes, ilma, et kogu ajakulu sellest suureneks.

Esimene osa on ehk veidi keerulisem taibata, aga kuna kindlasti ei ole aeglasem üht sületäit jagada järjestikuste mehitatud sektsioonide vahel, järjestikused sektsioonid kahel ringi poolel saavad aga olla ainult ühes lõigus. Muudel juhtudel me aga täisringi tehes ei võida ajas võrreldes edasi-tagasi käimisega.

```
long long delivery(int n, int k, int l, int p[])
{
    ll* ppAeg = new ll[n]; // meeskonnani paki viimise aeg ainult päripäeva liikudes
    ll* vpAeg = new ll[n]; // meeskonnani paki viimise aeg ainult vastupäeva liikudes
    // leiame ajad iga meeskonnani ainult päripäeva ja ainult vastupäeva liikudes,
    //arvestades, et eelnevad meeskonnad on suveniirid kätte saanud.
        for (int i = 0; i < n; i++) {
        if (i < k) { // esimesed k meeskonda mõlemast otsast
            ppAeg[i] = p[i] * 2; // käime päripäeva i-nda meeskonnani ja tagasi
            vpAeg[n-1-i] = (l - p[n-1-i]) * 2; // sama vastupäeva
        }
        else {
            int j = i-k; // meeskonna indeks, mis jääb kindlasti eelmisesse jaotusesse
            ppAeg[i] = p[i] * 2 + ppAeg[j];// eelmised jaotused + praeguse meeskonnani
            vpAeg[n-1-i] = (l - p[n-1-i]) * 2 + vpAeg[n-1-j]; //sama vastupäeva
        }
    }
    ll vastus = 1e18; // vastuseks suur arv, otsime miinimumi

    if (n == k) { // suveniirid saab korraga kaasa võtta ja ühe ringiga jaotada.
        vastus = l; // sel moel on üheks võimalikuks vastuseks sektorite arv.
    }

    // parim saab olla variant, kus jaotame kõik vastupäeva
    vastus = min(vastus, vpAeg[0]);
    // või üks täisring ja kõik ülejäänud vastupäeva
    vastus = min(vastus, vpAeg[k] + l);
    for (int i = 0; i < n; i++) {
        if (i == n - 1) {
            vastus = min(vastus, ppAeg[i]); // Kõik meeskonnad päripäeva jaotades
            continue;
        }
        // ainult edasi-tagasi: i. meeskonnani päripäeva ja i+1. alates vastupäeva.
        vastus = min(vastus, ppAeg[i] + vpAeg[i+1]);
        if (i + k + 1 == n) { // kui on ainult k meeskonda lõpuni
            vastus = min(vastus, ppAeg[i] + l); // üks täisring + ülejäänud päripäeva
        }
        else if (i + k + 1 < n) {
            // päripäeva i-nda meeskonnani, siis täisring (k pakki/meeskonda) ja siis
            // edasi alates meeskonnast i+k+1 vastupäeva.
            vastus = min(vastus, ppAeg[i] + vpAeg[i+k+1] + l);
        }
    }
    return vastus;
}
```

7.6 KONTROLLÜLESANDED

Nagu ka esimese osa esimeses peatükis, on käesoleva peatüki ülesanded suhteliselt lihtsad. Muretsemiseks pole põhjust, edasi tuleb ka raskemaid ülesandeid.

7.6.1 Onu Robert

Piilupart Donaldi lugudest tuntud Onu Robert külastab riiki, kus on kasutusel `N` erineva väärtusega rahatähte (`1≤N≤1000`). Kohale jõudes läheb ta kõigepealt panka, et kohalike kulutuste jaoks raha välja võtta. Pank väljastab raha „ahnelt“:
1. anda välja suurim kupüür, mis on väljastada jäänud summast väiksem,
2. korrata, kuni maksta jääv summa on null.



Väikseima kupüüri väärtus on alati 1, nii et väljastamine on võimalik. Kuna Onu Robert armastab erinevaid kupüüre, soovib ta saada võimalikult mitu erineva väärtusega raha-tähte. On teada, et ta on põhjatult rikas ja saab küsida mistahes summa. Leida, mitme erineva väärtusega kupüüre on tal võimalik ühe väljavõtmisega saada.
Sisendi esimesel real on arv `N`. Teisel real on `N` täisarvu rangelt kasvavas järjestuses, mis tähistavad kupüüride väärtusi. Väljastada üks täisarv: erineva väärtusega raha-tähtede arv, mida on võimalik pangast korraga välja võtta.

NÄIDE 1:
```
6
1 2 4 8 16 32
```
Vastus:
```
6
```
NÄIDE 2:
```
6
1 3 6 8 15 20
```
Vastus:
```
4
```

7.6.2 Bussijuhid

Ühes linnas töötab N bussijuhti (1≤N≤100). Samuti on seal linnas täpselt N hommikust ja N õhtust bussimarsruuti. Iga juht peab läbi sõitma ühe hommikuse ja ühe õhtuse marsruudi. Kui juhi poolt sõidetud tee pikkus on suurem kui etteantud arv D (1≤D≤10000), peab talle maksma lisatasu R tugrikut iga kilomeetri eest (1≤R≤5).
Ülesandeks on paigutada bussijuhid marsruutidele nii, et makstav lisatasu oleks minimaalne.
Sisendi esimesel real on arvud N, D ja R. Teisel real on N tühikutega eraldatud positiivset täisarvu, mis tähistavad hommikuste marsruutide pikkusi ning kolmandal real samuti N täisarvu, mis vastavad õhtustele marsruutidele.
Väljundisse kirjutada minimaalne võimalik lisatasu, mida bussijuhtidele tuleb maksta.



NÄIDE 1:
```
2 20 5
10 15
10 15
```
Vastus:
```
50
```
NÄIDE 2:
```
2 20 5
10 10
10 10
```
Vastus:
```
0
```
Pildil on dalla-dalla nimeline buss, millega ma Tansaanias sõitsin.

7.6.3 Hernehirmutised

Farmer Fredil on pikk peenar, mida ohustavad varesed. Peenar on jagatav N lõiguks (1≤N≤100), millest osadele on midagi külvatud, osadele mitte. Peenardele saab panna ka hernehirmutisi, mis kaitsevad vareste eest seda lõiku, millel nad ise seisavad, ning lisaks ühte naaberlõiku kummalgi pool. Leida, mitut hernehirmutist on vaja, et kaitsta ära terve peenar.
Sisendi esimesel real on arv N. Teisel real on string pikkusega N, mis koosneb märkidest '.' (tähistab külvatud lõiku) ja '#' (tähistab külvamata lõiku). Väljundisse kirjutada täpselt üks arv: mitut hernehirmutist on vaja.



NÄIDE 1:
```
3
.#.
```
Vastus:
```
1
```
NÄIDE 2
```
11
...##....##
```
Vastus:
```
3
```

7.6.4 Kolimine

Artur kolis just uude majja ja nüüd on tal üle N kuubikujulist pappkasti ($1 \le N \le 10\ 000$). Et ruumi kokku hoida, katsub Artur panna need kastid üksteise sisse. Et üht kasti teise sisse panna, peab see olema teisest rangelt väiksem. Leida strateegia, mis muudaks nähtavate kastide arvu võimalikult väikeseks. Sisendi esimesel real on kastide arv N ja teisel real N positiivset täisarvu, mis tähistavad kastide suurusi. Väljundi esimesele reale kirjutada minimaalne võimalik näha jäävate kastide arv M. Järgmisele M reale kirjutada igaühele info nähtava kasti ja selle sees olevate kastide kohta. Kui lahendusi on mitu, väljastada ükskõik milline neist.

NÄIDE:

```
6
1 1 2 2 2 3
```

Vastus:

```
3
1 2
1 2
2 3
```

Sobib ka:

```
3
1 2 3
1 2
2
```



7.6.5 Krokodillid

Indiana Jones seisab jõe vasakul kaldal ja tal tuleb teiselt kaldalt ära tuua kullast iidolikuju. Jões on rida kive ja krokodille, mille otsa Jones saab jõe ületamisel hüpata. Krokodille peale hüppamisel ujub too minema ja tagasitulekul ei saa enam seda krokodilli kasutada. Kivid jäävad alati paigale. Kuna pikad hüpped on ohtlikumad, katsub Indiana Jones jõge ületada nii, et tema pikim hüpe jääks minimaalseks. Leida, mis on pikim hüpe, mille ta siiski peab tegema.

Sisendi esimesel real on kivide arv N, krokodillide arv M ja jõe laius D ($1 \le D \le 10^9$). Teisel real on N täisarvu, mis tähistavad kivide kaugust jõe vasakust kaldast ning kolmandal real M täisarvu, mis tähistavad krokodillide kaugusi. Väljundisse kirjutada üks arv, mis tähistab Jonesi pikima vajaliku hüppe pikkust.

NÄIDE 1:

```
1 1 10
5
7
```

Vastus:

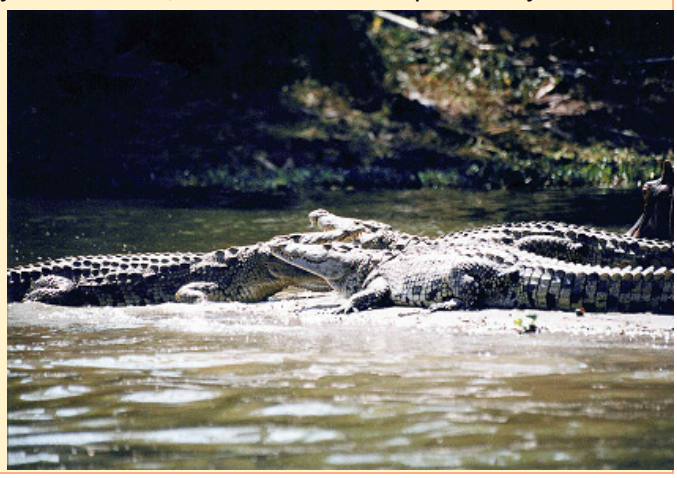
```
5
```

NÄIDE 2:

```
1 1 10
3
6
```

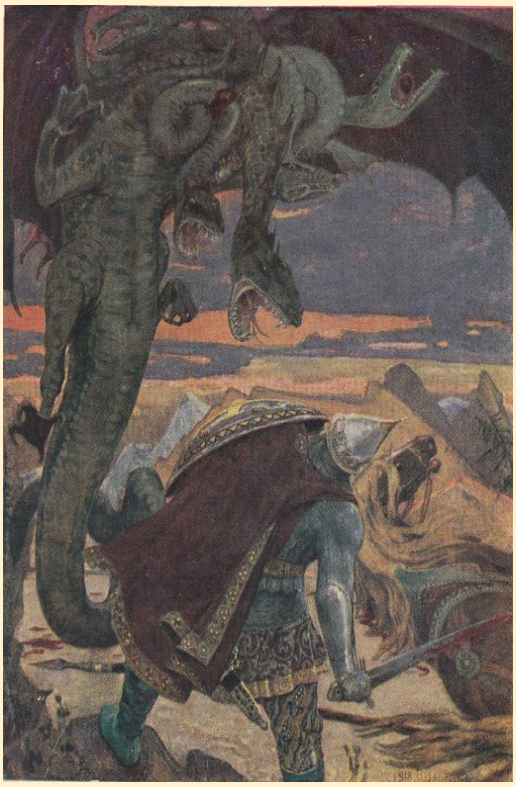
Vastus:

```
7
```



7.6.6 Lohe Gorõnitš

Vanal Kiievi-Venemaal oli suureks nuhtluseks lohe Gorõnitš. Bõliinadest tuntud vägilane Dobrõnja Nikititš palkab endale parajasti abiks teisi kangelasi, et lohe vastu võitlusse minna. Lohel on hulk erineva suurusega päid ja tema tapmiseks on vaja kõik need pead maha raiuda. Iga kangelane jaksab raiuda ülimalt ühe pea, aga ainult sellise, mis ei ole temast endast suurem. Samas tuleb kangelastele maksta tasu vastavalt nende enda suurusele. Leida minimaalne tasu, mis tuleb kangelastele lohe võitmise eest maksta.



Sisendi esimesel real on kaks täisarvu: lohe peade arv `N` ja riigis elavate kangelaste arv `M` (`1≤M, N≤20 000`). Teisel real on `N` täisarvu, mis tähistavad lohe peade suurusi ja kolmandal real `M` täisarvu, mis kangelaste suurused. Väljundisse kirjutada minimaalne makstav palk. Kui kangelastel pole võimalik lohet võita, sööb lohe nad kõik ära ja palka ei maksta. Sel juhul tuleb väljundisse kirjutada `0`.

NÄIDE 1:

```
2 3
5 4
7 8 4
```

Vastus:

11 (palkame esimese ja kolmanda kangelase)

NÄIDE 2:

```
2 1
5 5
10
```

Vastus:

```
0
```

7.6.7 Fraktsioonid

N-liikmeline (5≤N≤1000) parlament jaguneb fraktsioonideks, mis peavad olema erineva suurusega. Iga fraktsioon saadab kord päevas ühe liikme fraktsioonidevahelisele nõupidamisele. Et saaks arutada erinevaid asju, peab nõupidamisel iga kord olema erinev koosseis, vastasel korral arvab rahvas, et parlament ei tee midagi kasulikku ning korraldatakse uued valimised. Kuna parlamendi liikmed ei soovi oma kohast loobuda, katsuvad nad jaotada end fraktsioonideks nii, et nõupidamisi saaks korraldada võimalikult kaua. Kui antud on parlamendi suurus, siis leida, milline fraktsioonideks jaotus võimaldaks parlamendil võimalikult kaua püsida.
Sisendi ainsal real on täpselt üks arv N. Väljundi esimesele reale kirjutada fraktsioonide arv M ja teisele reale kasvavas järjestuses M tühikutega eraldatud täisarvu, mis tähistavad fraktsioonide suurusi.

NÄIDE 1:
7
Vastus:
2
3 4 (parlament töötab 3*4=12 päeva)

NÄIDE 2:
31
Vastus:
6
2 3 5 6 7 8 (parlament töötab 10080 päeva)
Pildil on Kreeka parlament.



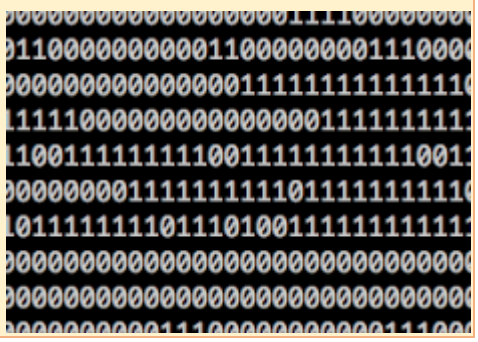
7.6.8 Bitimask

Raamatu esimeses osas oli muuhulgas juttu bitioperatsioonidest ja bitimaskidest. Bitimaskid võimaldavad kergesti manipuleerida mõne arvu üksikuid bitte. Kui meil on näiteks vaja 32-bitise arvu alumised neli bitti ära nullida, tuleb sooritada bitikaupa AND operatsioon bitimaskiga 0xFFFFFFF0 (kõik bitid peale alumise nelja on ühed).
Siin ülesandes tuleb leida bitimask M, mis:
1. asub kindlas vahemikus (L≤M≤U),
2. annab etteantud arvuga N bitikaupa OR operatsiooni järel maksimaalse tulemuse.
Sisendi ainsal real on kolm 32-bitist täisarvu N, L ja U, kus L≤U. Väljundisse kirjutada arv M. Kui võimalikke vastuseid on mitu, väljastada neist väikseim.

NÄIDE 1:
100 50 60
Vastus:
59 (59 | 100 = 127)

NÄIDE 2:
100 0 100
Vastus:
27 (27 | 100 = 127)

NÄIDE 3:
1 0 100
Vastus:
100



7.6.9 LED-lambid

Hullul elektroonikul Romanil on ristkülikukujuline paneel, mis on täidetud LED-lampidega. Alguses on kõik lambid välja lülitatud. Paneeli juhtimiseks on Roman koostanud mikroskeemi, mis võimaldab etteantud suurusega alamristkülikul lampe ümber lülitada – need, mis olid enne väljas, on nüüd sees ja vastupidi.
Ülesande sisendis on toodud pilt, mis on vaja lõpuks saavutada, leida, mitu lülitamist on selleks vaja. Sisendi esimesel real on neli täisarvu: paneeli kõrgus `N` ja laius `M` ning ümberlülitatava piirkonna kõrgus `A` ja laius `B` (`1≤A≤N≤100, 1≤B≤M≤100`). Järgmisel `N` real on igaühel `M` märgist koosnev string, mis näitab, millised lambis on soovitud pildil sisse lülitatud – `0` tähistab väljalülistatud ja `1` sisselülitatud lampi. Väljundisse kirjutada vajalike lülituste arv. Kui nõutud lülitamine pole võimalik, väljastada `-1`.

NÄIDE 1:

```
3 3 1 1
010
101
010
```

Vastus:

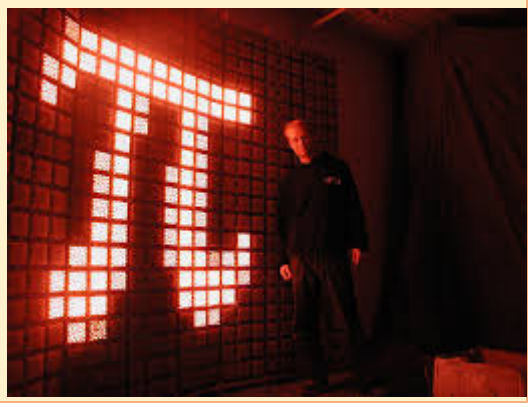
```
4
```

NÄIDE 2:

```
4 3 2 1
011
110
011
110
```

Vastus:

```
6
```



NÄIDE 3:

```
3 4 2 2
0110
0111
0000
```

Vastus:

```
-1
```

7.6.10 Antimonotoonne jada

Käesoleva raamatu neljandas peatükis uuriti muuhulgas etteantud arvude seast pikima kasvava või kahaneva alamjada (nn monotoonse alamjada) leidmist. Siin ülesandes tuleb leida jada A pikim antimonotoonne alamjada B, s.t selline jada, kus B[0]>B[1]<B[2]<B[3]…
Sisendi esimesel real on jada A pikkus N (1≤N≤30 000). Teisel real on N positiivset täisarvu.
Väljundisse kirjutada pikima võimaliku antimonotoonse alamjada pikkus.

NÄIDE 1:

```
5
1 2 3 4 5
```

Vastus:

1

NÄIDE 2:

```
5
5 4 3 2 1
```

Vastus:

2

NÄIDE 3:

```
5
5 1 4 2 3
```

Vastus:

5

NÄIDE 4:

```
5
2 4 1 3 5
```

Vastus:

3



7.7 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk7 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
MinenurkaPikk.cpp	Tankiülesande funktsioon mineNurka esialgne variant
MinenurkaLyhem.cpp	Nurka minemise meetodid ümber kirjutatuna
Mortimer2.cpp, Mortimer2.java, Mortimer2.py	Ülesande “Miljonär ja vaeslapsed” jõumeetodiga lahendus (testimiseks)
MortimerGen.cpp, MortimerGen.java, MortimerGen.py	Testigeneraator ülesande “Miljonär ja vaeslapsed” testide koostamiseks
MortimerVal.cpp, MortimerVal.java, MortimerVal.py	Validaator ujukomaarvude võrdlemiseks testimisel
Myndid.cpp, Myndid.java, Myndid.py	Mündiülesanne ahne algoritmiga
Kingid.cpp, Kingid.java, Kingid.py	Jaotamise ülesanne ahne algoritmiga
Majakavahid.cpp, Majakavahid.java, Majakavahid.py	Vahemike kattuvuse ülesanne ahne algoritmiga
Suveniirid.cpp, Suveniirid.java, Suveniirid.py	Ahne algoritmiga lahenduv keerukam ülesanne

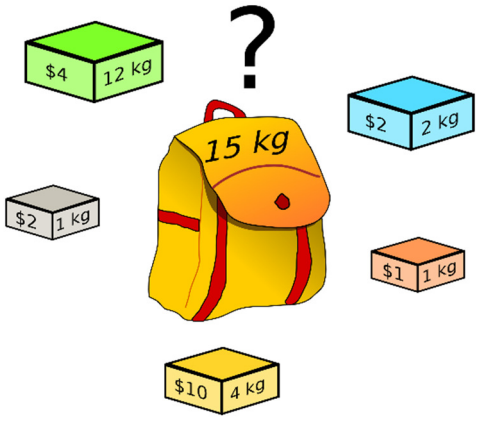
8 DÜNAAMILINE PLANEERIMINE EDASIJÕUDNUTELE

Raamatu esimeses osas oli juttu lihtsamatest dünaamilise planeerimise ülesannetest. Selles peatükis pöördutakse teema juurde tagasi ja tutvustatakse mitmeid klassikalisi dünaamilise planeerimise algoritme ja nende kasutust. Lisaks on siin mõned keerulisemad ülesanded, mille lahendamiseks tuleb tavapärasest rohkem pead murda.

> Those who cannot remember the past are condemned to repeat it.
>
> - Dynamic programming

8.1 SELJAKOTI PAKKIMINE

Seljakoti pakkimise probleem (*knapsack problem*) on ilmselt arvutiteaduse tuntumate ülesannete seas. Tegemist on optimiseerimisülesandega ja seda kasutatakse paljudes optimiseerimisvaldkondades. Probleemi on uuritud juba vähemalt aastast 1897 ning oma nime on see saanud Ameerikasse kolinud Baltisaksa matemaatiku Tobias Dantzigi töödest.



Seljakotiprobleem on NP-keeruline ja seda ei saa deterministliku algoritmiga polünomiaalse ajaga lahendada. Ülesandeklassi sagedast esinemist arvestades on aga kasulik teada, et paljude tavapäraste sisendite jaoks eksisteerib lihtne dünaamilist planeerimist kasutav algoritm. Vaatame, kuidas seda rakendada järgneva ülesande näitel.

8.1.1 Ülesanne: Kohver

Kalle on minemas IOI-le ja ta saab võtta kaasa ühe kohvri, mille kogukaal ei ületa `m` kilogrammi. Kalle tahab kaasa võtta muidugi kõige olulisemad esemed. Aita tal teha parim valik.
Sisendi esimesel real on üks täisarv `m` – kohvri maksimaalne kaal kilogrammides. Teisel real on üks täisarv `n` – Kalle kõigi esemete arv. Järgmisel `n` real on iga eseme kohta tema kaal kilogrammides ja väärtus.
Väljastada parim võimalik seljakoti väärtus ja vähemalt üks sellele vastav sisu (kotti valitavate esemete järjekorranumbrid sisendis).

NÄIDE:

```
10
8
5 7
3 9
1 5
2 2
6 4
4 5
7 8
8 9
```

Vastus:

```
21
```



Selle ülesande juures tuleb kohe tähele panna, et iga eset on täpselt üks ja on täpselt kaks võimalust – kas ese läheb kotti või mitte. Seda tüüpi seljakotiülesannet nimetatakse **0/1** seljakotiülesandeks. Jõumeetodil lahendades tuleks moodustada kõikvõimalikud kombinatsioonid esemetest ning leida iga võimaliku kombinatsiooni väärtus. Vastuseks on kõige suurem väärtus ja sellele vastav esemete kombinatsioon. Kuna iga kord tuleb teha binaarne valik (ese kuulub hulka või mitte), on erinevate kombinatsioonide arv 2^n . Muidugi saab puud kärpida selle võrra, et kui mingis harus on asjade kogukaal suurem kui lubatud maksimumkaal, siis seda haru mööda pole mõtet edasi liikuda, aga nagu teada, siis keerukusklass sellest siiski ei muutu.

8.1.2 Dünaamiline planeerimine

Oletame, et oleme juba viimase eseme juures ja kaalume, kas seda kohvrisse pakkida või mitte. Kui me viimast eset kohvrisse ei paki, siis jääb kohvri kaal ja seal olevate esemete koosseis muutumatuks ja meid huvitab hoopis, kuidas nende $n-1$ eseme hulgast optimaalne valik on saadud – ülesanne taandub veidi väiksemaks. Kui aga soovime viimase eseme lisada, siis peame arvestama selle kaaluga: see tähendab, et varem lisatud esemete kaal peab olema selle viimase eseme kaalu võrra väiksem kui lubatud kogukaal. Näiteks kui võime kohvrisse pakkida 9kg esemeid ja viimane ese kaalub 2kg, siis eelnevalt pakitud esemete kogukaal ei tohi ületada $9-2=7$ kg. Nii sõltub viimase, n-nda eseme valik sellest, kas suurema väärtusega on selline kohver, mis sai pakitud $n-1$ esemest või selline, kus eelneva $n-1$ eseme kogukaal ei ületa seitset kilogrammi. Nii on meil vaja lahendada hoopis kaks väiksemat alamülesannet. Kuna viimase eseme lisamise seisukohalt ei ole oluline, millised esemed on varem valitud (huvitab ainult nende kogukaal), siis saame ehitada oma algoritmi alt üles.

8.1.3 Rekursioonivalem

Olgu esemete kaalud positiivsed täisarvud $m_1, m_2 \dots m_n$ ja väärtused $v_1, v_2 \dots v_n$ Olgu $mv(i,m)$ suurim kohvri väärtus, mida saab moodustada kuni i-st esimesest esemest ja mille kaal ei ületa m-i. Siis:

$$mv(i,m) = \begin{cases} 0, kui\ i = 0 \\ mv(i-1,m), kui\ m_i > m \\ \max(mv(i-1,m), mv(i-1,m-m_i) + v_i, kui\ m_i \leq m \end{cases}$$

See tähendab, et ridades on meil esemed, mille lisamist kaalume. Esimeses reas esimene, teises teine jne. Veergudes on kogukaalud $0 \dots m$.

8.1.4 Tabeli täitmine

Iga lahtri täitmisel peame kõigepealt vaatama, kas vaadeldava kaaluga ese mahub vaadeldava kogukaaluga kohvrisse (kui eseme kaal on suurem kui vaadeldava kohvri kogukaal, siis seda lisada ei saa) ja väärtus jääb samaks, mis rida varem (`mv[i-1][j]`).

Kui ese mahuks kohvrisse, siis tuleb vaadata, kas seda lisades saame parema väärtuse kui lisamata jättes. Viimasel juhul on väärtuseks muidugi `mv[i-1][j]`, lisamise korral aga eseme kaalu ja selle kaalu võrra väiksema kohvri parim väärtus ehk siis, kui eseme väärtus on v ja kaal `m`, siis on selleks `mv([i-1], [j-m]) + v`.

Tabeli veergudes on kohvri kogukaalud ja ridades esemete kaalud ja väärtused. Tabeli lahtris `[i, j]` on maksimaalne väärtus kohvrist, mille mass on kuni `i` kg ja mis on komplekteeritud esemete hulgast ridadel `1…j`.

	0 kg	1 kg	2 kg	3 kg	4 kg	5 kg	6 kg	7 kg	8 kg	9 kg	10 kg
5kg (7)	0	0	0	0	0	7	7	7	7	7	7
3kg (9)	0	0	0	9	9	9	9	9	16	16	16
1kg (5)	0	5	5	9	14	14	14	14	16	21	21
2kg (2)	0	5	5	9	14	14	16	16	16	21	21
6kg (4)	0	5	5	9	14	14	16	16	16	21	21
4kg (5)	0	5	5	9	14	14	16	16	19	21	21
7kg (8)	0	5	5	9	14	14	16	16	19	21	21
8kg (9)	0	5	5	9	14	14	16	16	19	21	21

8.1.5 Kood

Siin on funktsioon, mis analoogse tabeli täidab ja ülesandele vastuse leiab:

```
int pakiKott(int maksKaal, int kogus, int* kaalud, int* vaartused)
{
    int** parimad = new int*[maksKaal + 1];
    for (int i = 0; i <= maksKaal; i++){
        parimad[i] = new int[kogus];
    }

    for (int i = 0; i < kogus; i++){
        parimad[0][i] = 0;
    }

    for (int i = 0; i <= maksKaal; i++){
        parimad[i][0] = 0;
        if (kaalud[0] <= i){
            parimad[i][0] = std::max(parimad[i][0], vaartused[0]);
        }
    }

    int vastus = 0;
    for (int i = 1; i <= maksKaal; i++){
        for (int j = 1; j < kogus; j++){
            parimad[i][j] = parimad[i][j - 1];
            if (kaalud[j] <= i){
                parimad[i][j] =
                    max(parimad[i][j], parimad[i - kaalud[j]][j - 1] + vaartused[j]);
            }
            vastus = max(vastus, parimad[i][j]);
        }
    }
    return vastus;
}
```

8.1.6 Tee taastamine

Mis esemed siis ikkagi kohvrisse pakiti? Seda on tervest DP tabelist päris kerge välja lugeda: kui viimase ruudu väärtus on sama, mis väärtus samal kohal eelmises reas (`mv[i-1, j] == mv[i,j]`), siis seda eset kotti ei lisatud (või vähemalt eksisteerib optimaalne lahendus ilma selle eseme lisamiseta). Kui väärtus eelmises reas on erinev, siis ese lisati ja vaatame samamoodi kotti, mille kaal on kogukaalu ja lisatud eseme kaalu `m` vahe (`mv[i-1, j-m]`).

Näidet vaadates saab vastava kombinatsiooni kõige lihtsamini leida liikudes viimast veergu mööda üles, kuni viimase väärtuseni 21 ehk 3. reale. Kuna teisel real on viimases veerus väiksem väärtus (16), siis 3. ese kuulub kohvrisse. Nüüd peame selle kaalu maha lahutama: $10 - 1 = 9$, seega

vaatame eelviimast tulpa. Siin on 2. reas väärtus 16 ja eelmises reas 7, järelikult tuleb lisada ka teine ese. Lahutame selle kaalu: $9 - 3 = 6$. Kuna esimese rea 6. veerus on väärtus 7, lisati ka esimene ese. Seega on üheks lahendiks esemed 1, 2 ja 3, kohvri kogukaal on $5 + 3 + 1 = 9$ kg ja väärtus $7 + 9 + 5 = 21$.

8.1.7 Keerukus

Dünaamilise planeerimise juures on mahulist keerukust enamasti küllaltki kerge hinnata – sellele vastab DP tabeli suurus. Seljakotiülesandeks on tabeli üheks mõõtmeks esemete arv, teine mõõde aga sõltub sellest, kui suur on lubatud kaal.

Ka ajaline keerukus sõltub selle lahenduse puhul lubatud kaalu väärtusest ja on on $O(nM)$, kus M on koti lubatud kaal.

Seega antud algoritm, lahendab seljakotiülesande **pseudopolünomiaalse** keerukusega. See tähendab, et keerukus on polünomiaalne sisendi arvulise väärtuse suhtes (seljakoti maksimaalse kaalu M suhtes), aga on siiski eksponentsiaalne sisendi mahu (bittide arvu) suhtes. Kui seljakoti maksimaalne kaal kasvab ühe biti võrra, kasvab algoritmi tööaeg 2 korda. Näiteks kui seljakoti lubatud maksimaalne kaal on 8 asemel 256, siis kasvab kuluv aeg 32 korda. Seega on keerukus täpsemalt lahti kirjutades $O(n * 2^{\log_2 M})$.

8.1.8 Seljakoti erinevad variatsioonid

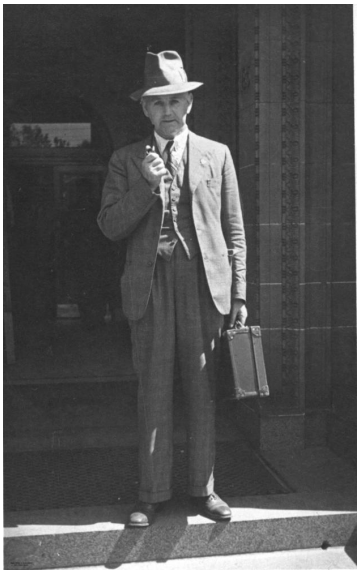
Sellist tüüpi seljakoti ülesannet, kus igat eset saab lisada maksimaalselt ühe korra, nimetatakse ka **0/1** seljakotiülesandeks. Sageli kohtab aga ka **piiramata** (*unbounded*) seljakotiülesannet – sel juhul saab lisada iga eset nii palju, kui soovi on. Sellisel juhul:

$$mv(m) = \begin{cases} 0, kui\ m = 0 \\ \max\limits_{m_i \leq m}(v_i + mv(m - m_i)) \end{cases}$$

Oma olemuselt on seljakoti ülesanne väga lähedane 5. peatükis tutvustatud mündiülesandele. Müntide väärtused vastavad kaaludele ja kogusumma kogukaalule. Optimeerisime sellele, et müntide arv oleks kõige väiksem, seega võiks anda igale mündile väärtuseks -1.

8.2 RÄNDKAUPMEHE ÜLESANNE

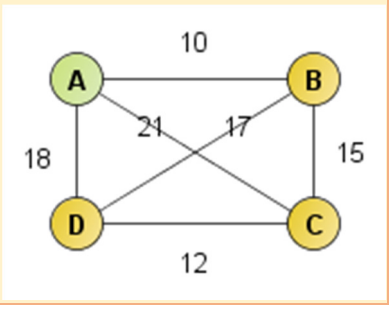
Teine klassikaline NP-keeruline optimiseerimisülesanne on rändkaupmehe ülesanne. Ülesande sõnastus on lihtne: antud on hulk linnasid koos nendevaheliste kaugustega. Leida lühim tee, mis võimaldab kõik need linnad läbi käia ja alguspunkti tagasi jõuda. Ülesanne sõnastati esmakordselt 1930. aastatel ja see on üks kõige rohkem uuritud optimiseerimisülesandeid. Hoolimata üldkujul lahendamise keerukusest on ülesandele leitud mitmeid heuristikuid, mis võimaldavad paljudel erijuhtudel leida mõistliku ajaga lahendeid, mis on ideaalsest vaid protsendi või paari võrra halvemad.



Praeguses kontekstis uurime rändkaupmehe ülesandele täpse lahendi leidmist dünaamilise planeerimise abil.

8.2.1 Ülesanne: kaupmees

Antud on linnade nimekiri, alguspunkt ja iga kahe linna vahelised kaugused. Leida lühim tee, mis läbib iga linna täpselt üks kord ning lõppeb alguspunktis.
Sisendi esimesel real on üks täisarv `n` - linnade arv. Järgneval `n-1` real on igaühel `n-i` arvu, kus `i` on reanumber: `i`-nda linna kaugus linnadest `i+1`, `i+2`…`n`
Väljundisse kirjutada üks täisarv: lühima tee pikkus, kui tee algab ja lõppeb esimeses linnas.



NÄIDE:

```
4
10 21 18
15 17
12
```

Vastus:

55

Jõumeetodit kasutades tuleks proovida kõikvõimalikud linnade järjestused läbi, neid on $(n-1)!$. Samas pole raske märgata, et kui vaatame teid `A-B-C-D-ülejäänud linnad` ja `A-C-B-D-ülejäänud linnad`, siis arvutame nende ülejäänud linnade erinevaid võimalikke järjekordi iga kord uuesti. Samas sõltub kokkulangevus kahest asjast: viimati külastatud linnast ja kõikide seni külastatud linnade hulgast.

Teame, et peame lõpetama linnas A. Enne linna A jõudmist tuleb meil läbida kõik ülejäänud linnad mingis järjekorras. Tähistame seda olukorda (A, {B,C,D}). Meid huvitab selline linnade B, C ja D järjestus, mille korral läbitakse kõige väiksem maa. On kolm erinevat võimalust vastavalt sellele, kas linna A tullakse linnast B, linnast C või linnast D.

Kui linna A tullakse linnast B, siis tee pikkuseks on tee linnast B linna A ja parima tee (B, {C,D}) pikkus. See, millisel moel C ja D kaudu linna B jõuti, ei ole sellel sammul enam oluline, meid huvitab ainult, et see oleks lühim. Lühima tee iga alamtee on ise ka lühim tee.

Analoogselt, kui linna A tullakse linnast C, siis tee pikkus on (A,C)+(C,{B,D}) ja kui linnast D, siis (A,D) + (D,{B,C}). Vastuseks on muidugi neist teedest lühim!

8.2.2 Rekursioonivalem

Olgu meil linnad $L_0..L_{n-1}$ ja mingi linnade hulk $V \in \{L_0,..,L_{n-1}\}$. (L_i, L_j) tähistab kaugust linnast L_i linna L_j ja $mk(L_i, V)$ minimaalset teed linna L_i läbi kõigi linnade hulgas V. Siis saame järgmise valemi:

$$mk(L_i, V) = \begin{cases} (L_0, L_i), kui\ V = \emptyset \\ \min_{\forall L_j \in V}((L_j, L_i) + mk(L_j, V \backslash \{L_j\})) \end{cases}$$

8.2.3 Valemi kasutus

Kõigepealt leiame baasjuhud ehk teed lähtelinnast A otse igasse teise linna. Need saame kauguste tabelist:

	A	B	C	D
A	0	10	21	18
B	10	0	15	17
C	21	15	0	12
D	18	17	12	0

$$mk(B,\{\}) = (A,B) = 10$$

$$mk(C,\{\}) = (A,C) = 21$$

$$mk(D,\{\}) = (A,D) = 18$$

Järgmiseks saab arvutada minimaalsed teepikkused läbi ühe linna:

$$mk(B,\{C\}) = (C,B) + mk(C,\{\}) = 15 + 21 = 36$$

$$mk(B,\{D\}) = (D,B) + mk(D,\{\}) = 17 + 18 = 35$$

$$mk(C,\{B\}) = (B,C) + mk(B,\{\}) = 15 + 10 = 25$$

$$mk(C,\{D\}) = (D,C) + mk(D,\{\}) = 12 + 18 = 30$$

$$mk(D,\{B\}) = (B,D) + mk(B,\{\}) = 17 + 10 = 27$$

$$mk(D,\{C\}) = (C,D) + mk(C,\{\}) = 12 + 21 = 33$$

Teepikkused läbi kahe linna on juba huvitavamad:

$$mk(B,\{C,D\}) = min((C,B) + mk(C,\{D\}),(D,B) + mk(D,\{C\})) = min(15 + 30, 17 + 33) = 45$$

$$mk(C,\{B,D\}) = min((B,C) + mk(B,\{D\}),(D,C) + mk(D,\{B\})) = min(15 + 35, 12 + 27) = 39$$

$$mk(D,\{B,C\}) = min((B,D) + mk(B,\{C\}),(C,D) + mk(C,\{B\})) = min(17 + 36, 12 + 25) = 37$$

Lühim tee, mis linnast A tagasi linna A viib läbi kõigi linnade:

$$mk(A,\{B,C,D\}) =$$

$$= min\big((B,A) + mk(B,\{C,D\}),(C,A) + mk(C,\{B,D\}),(D,A) + mk(D,\{B,C\})\big) =$$
$$= min(10 + 45, 21 + 39, 18 + 37) = 55$$

8.2.4 DP tabel

Aga kuidas neid väärtusi tabelisse panna? Funktsioonil mk on 2 parameetrit – vaadeldav linn ja linnade hulk, mis on juba vaadeldud. Linnaga on lihtne, aga mida teha hulgaga? Siin tuleb kasuks meelde tuletada ülesanne „Moekunstnik ja koiliblikas“ (I osa, ptk 5.8) – hulki on hea mugav edasi anda bitimaskidena.

	0	1	2	3	4	5	6	7
	{}	{B}	{C}	{BC}	{D}	{BD}	{CD}	{BCD}
B	10	-	36	-	35	-	45	-
C	21	25	-	-	30	39	-	-
D	18	27	30	37	-	-	-	-
A	-	-	-	-	-	-	-	55

8.2.5 Kood

Siin on funktsioon, mis selle ülesande lahendab:

```
vector<int> kaupmees(int n, int ap, int** tabel)
{
    int** dp;
    int n2 = 1 << n;
    dp = new int*[n2];
    for (int i = 0; i < n2; i++) {
        dp[i] = new int[n];
        for (int j = 0; j < n; j++) {
            dp[i][j] = -1;
        }
    }
    for (int i = 0; i < n2; i++) {
        if (1 << ap & i > 0) {
            continue;
        }
        if (i == 0) {
            continue;
        }
        vector<int> cv;
        for (int j = 0; j < n; j++) {
            int linn = 1 << j;
            if (i & linn > 0) {
                cv.push_back(j);
            }
        }
        if (cv.size() == 1) {
            dp[i][cv[0]] = tabel[ap][cv[0]];
            continue;
        }
        for (int j = 0; j < cv.size(); j++) {
            int J = cv[j];
            dp[i][J] = 1000000000;
            int pi = i - (1 << J);
            for (int k = 0; k < cv.size(); k++) {
                int K = cv[k];
                if (K == J || dp[pi][K] == -1 || tabel[K][J] == -1) {
                    continue;
                }
                dp[i][J] = min(dp[i][J], dp[pi][K] + tabel[K][J]);
            }
        }
    }
    n2--;
    n2 -= (1 << ap);
    int vas = 1000000000;
    for (int i = 0; i < n; i++) {
        if (tabel[i][ap] == -1) {
            continue;
        }
        vas = min(vas, tabel[i][ap] + dp[n2][i]);
    }
```

```
    vector<int> vasvec;
    vasvec.push_back(ap);
    int rd = vas;
    int ni = ap;
    while (n2 != 0) {
        for (int i = 0; i < n; i++) {
            if (dp[n2][i] == -1 || tabel[i][ni] == -1) {
                continue;
            }
            if (dp[n2][i] + tabel[i][ni] == rd) {
                rd -= tabel[i][ni];
                vasvec.push_back(i);
                ni = i;
                n2 -= (1 << i);
                break;
            }
        }
    }
    return vasvec;
}
```

8.2.6 Tee taastamine

Tee taastamiseks on vaja teada, milline avaldis miinimumi andis. Kuna aga seda miinimumi on vaja leida päris mitme lahtri seast (nii mitme, kui mitu linna meil parasjagu vaadeldavas hulgas on), siis on taastamise vajaduse korral targem meeles hoida see linn, mille kaudu antud miinimum leiti. Nii teame taastades, et antud sammul läbiti see linn ja eelnevalt läbiti linnad hulgast $V \backslash \{L_i\}$

8.2.7 Keerukus

Kavalast lahendusest olenemata ei tohi linnade arv olla kuigi suur. DP tabeli üheks mõõtmeks on kombinatsioonide arv, mis on juba 2^{n-1}. Teiseks mõõtmeks on linnade arv n ja lisaks kasvab koos linnade arvuga ka see, mitmest väärtusest miinimumi võtta tuleb. Kokku on Held-Karpi algoritmi ajaliseks keerukuseks $O(n^2 2^n)$ ja mahuliseks keerukuseks $O(n2^n)$

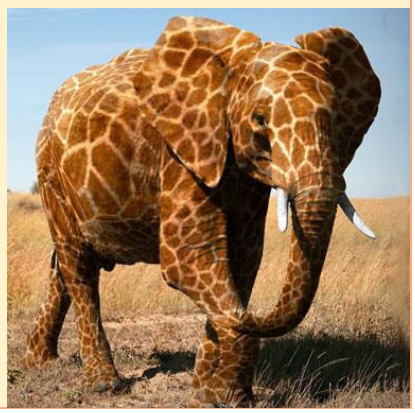
8.3 LEVENSHTEINI KAUGUS

Paljudes ülesannetes on vaja hinnata kahe stringi omavahelist „sarnasust“. Näiteks „jalalaba“ ja „ali baba“ võiks olla omavahel sarnasemad kui „kartulipõld“ ja „veoauto“. Sellise sarnasuse hindamiseks kasutatakse sellist **Levenshteini kauguse** nimelist mõõdikut (nimetatud vene arvutiteadlase Vladimir Levenshteini järgi, kes tegeles vastava probleemiga juba aastal 1965). Sageli nimetatakse seda kaugust ka **toimetamiskauguseks** (i.k. *edit distance*).



8.3.1 Ülesanne: Levenshteini kaugus

Olgu defineeritud järgmised operatsioonid:
asendamine: üks märk asendatakse teisega, näiteks uxv → uyv
kustutamine: üks märk eemaldatakse: uxv → uv
lisamine: üks märk lisatakse: uv → uxv
Leia, mitu operatsiooni on vaja minimaalselt teha, et saada ühest stringist teine.
Sisendi esimesel real on kaks täisarvu – esimese stringi ja teise stringi pikkus. Teisel real on esimene string ja kolmandal real teine string.
Väljastada üks täisarv – vajalike operatsioonide arv, et saada ühest stringist teine.



NÄIDE:

```
7 10
elevant
kaelkirjak
```

Vastus:

```
8
```

Esimeses osas 5. peatükis oli pikima ühise osajada leidmise ülesanne. Seal meid huvitas just kokkulangevate tähtede arv, antud ülesandes on vaja loendada kohti, kus kokkulangevus puudub (märk tuleb kas lisada, eemaldada või asendada). Oma olemuselt on antud ülesanne väga sarnane pikim osajada leidmisele ning tegelikult on viimane Levenshteini kauguse leidmise erijuht.

Pikima ühise osajada leidmise ülesandes loendasime tähtede kokkulangevusi ja leidsime maksimumi, selles ülesandes on kokkulangevuse hinnaks aga 0 ja otsitakse miinimumi. Üldine idee aga jääb samaks.

Võrdleme jadades viimaseid märke a_m ja b_n. Seekord on meil neli võimalust:

- märgi a_m kustutamine esimesest stringist;
- märgi b_n lisamine esimesse stringi;
- märgi a_m asendamine märgiga b_n;
- juht, kus $a_m = b_n$.

Viimane neist on kõige lihtsam: kui on kokkulangevus, siis teisenduskauguseks on stringide $a_1 \dots a_{m-1}$ ja $b_1 \dots b_{n-1}$ teisenduskaugus. Samuti asendamise puhul on teisenduskauguseks on stringide $a_1 \dots a_{m-1}$ ja $b_1 \dots b_{n-1}$ teisenduskaugus, millele on juurde liidetud 1 toimunud asendusoperatsiooni eest. Samuti on vaja liita 1 toimunud operatsiooni eest lisamise/kustutamise korral ning võrdlemiseks jäävad stringid $a_1 \dots a_{m-1}$ ja $b_1 \dots b_n$ (kustutamisel) või $a_1 \dots a_m$ ja $b_1 \dots b_{n-1}$ (lisamisel). Kuna vaja leida on vähim võimalik toimetamiskaugus, tuleb võtta nendest neljast võimalusest see, mis annab kõige väiksema toimetamiskauguse.

Tuleb veel tähele panna, et kui üks stringidest on tühi (pikkusega 0), siis minimaalseks toimetamiskauguseks on teise stringi pikkus – tuleb teha nii mitu lisamise/kustutamise operatsiooni, kui pikemas stringis märke on.

NB! Selleks, et leida kahe stringi vahelist Levenshteini kaugust, ei ole vaja sooritada tegelikult ühtki neist operatsioonidest, sest selle ülesande käigus ei muudeta kumbagi sisendstringi!

8.3.2 Rekursioonivalem

Olgu $LK_{a,b}(i,j)$ stringide $a_1 \dots a_i$ ja $b_1 \dots b_j$ toimetamiskaugus ehk Levenshteini kaugus. Siis saame järgmise rekursioonivalemi:

$$LK_{a,b}(i,j) = \begin{cases} \max(i,j), kui \min(i,j) = 0 \\ \min \begin{cases} LK_{a,b}(i-1,j) + 1 \\ LK_{a,b}(i,j-1) + 1 \\ LK_{a,b}(i-1,j-1) + 0, kui\ a_i = b_j \\ LK_{a,b}(i-1,j-1) + 1, kui\ a_i \neq b_j \end{cases} \end{cases}$$

8.3.3 DP tabel

Järgnev tabel on täidetud seda valemit kasutades:

	[]	E	L	E	V	A	N	T
[]	0	1	2	3	4	5	6	7
K	1	1	2	3	4	5	6	7
A	2	2	2	3	4	4	5	6
E	3	2	3	2	3	4	5	6
L	4	3	2	3	3	4	5	6
K	5	4	3	3	4	4	5	6
I	6	5	4	4	4	5	5	6
R	7	6	5	5	5	5	6	6
J	8	7	6	6	6	6	6	7
A	9	8	7	7	7	6	7	7
K	10	9	8	8	8	7	7	**8**

Paneme tähele, et esimeses reas ja veerus on olukorrad, kus üks stringidest on tühi, sellisel juhul on kauguseks teise stringi pikkus. See on ka loogiline – selleks, et saada tühjast stringist näiteks string „elevant“, tuleb lisada kõik selle stringi tähed ehk toimetamiskauguseks on stringi pikkus.

Mööda diagonaali toimuvad asendamised(või kokkulangemised – kulu on 0), paremale või alla liikudes eemaldamised ja lisamised. Vastus asub tabeli alumises parempooles nurgas.

Funktsioon, mis vastava tabeli koostab, on siin:

```
int LevenshteiniKaugus(int n, int m, string s, string t)
{
    int** dp = new int*[n + 1];
    for (int i = 0; i <= n; i++) {
        dp[i] = new int[m + 1];
        for (int j = 0; j <= m; j++) {
            dp[i][j] = 0;
        }
    }
    for (int i = 0; i <= n; i++) {
        dp[i][0] = i;
    }
    for (int i = 0; i <= m; i++) {
        dp[0][i] = i;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int sc = 1;
            if (s[i-1] == t[j-1]) {
                sc = 0;
            }
            dp[i][j] = min(min(dp[i-1][j] + 1, dp[i][j-1] + 1), dp[i-1][j-1] + sc);
        }
    }

    return dp[n][m];
}
```

8.3.4 Teisenduse taastamine tabelist

Tabelist on võimalik taastada ka tehtud teisendused, mis vähima kauguseni viisid. Lahendusi võib olla muidugi mitu.

Taastamine käib loomisega võrreldes tagurpidi, alustame kohalt `dp[m,n]` ja lõpetame kohal `dp[0,0]`. Olles lahtris `dp[i,j]`, vaatame milline väärtustest lahtrites `dp[i-1,j]`, `dp[i-1,j-1]` ja `dp[i,j-1]` on vähim.

Siin on mõned lahendused:

```
--ELEVANT- --EL--EVANT     --ELEVA-NT
KAELKIRJAK      KAELKIRJAK-     KAELKIRJAK
```

8.3.5 Ajaline ja mahuline keerukus

Selle algoritmi ajaline ja mahuline keerukus on $O(nm)$, kus n ja m on stringide pikkused. Kui vaja on leida vaid arvuline kaugus ja tee taastamine oluline ei ole, siis saab aga kasutada kokkuhoidlikku lahendust – uue lahtri arvutamiseks on vaja teada ainult eelmise rea infot ning käesoleva rea eelmist väärtust – seega piisab kauguse leidmiseks vaid eelmise ja käesoleva rea meelespidamisest.

8.3.6 Levenshteini kauguse omadused

Levenshteini kaugusel on järgmised omadused:

- See on vähemalt nii suur, kui stringide pikkuste erinevus (puuduvad tähed tuleb lisada või üleliigsed eemaldada) : $LK(a,b) \geq |\bar{a} - \bar{b}|$, kus $\bar{x}$ on stringi x pikkus.
- Ei ole kunagi suurem, kui pikema stringi pikkus (saab asendada iga märgi): $LK(a,b) \leq max(\bar{a},\bar{b})$
- See on 0 siis ja ainult siis, kui stringid on täpselt samad: $LK(a,b) = 0 \Leftrightarrow a = b$
- Esimese sõna kaugus teisest on sama, mis teise sõna kaugus esimesest: $LK(a,b) = LK(b,a)$
- Kahe stringi Levenshteini kaugus ei ole kunagi suurem kui esimese stringi kauguse mingist kolmandast stringist ja teise stringi kauguse sellest kolmandast stringist summa: $LK(a,b) \leq LK(a,c) + LK(b,c)$

8.3.7 Levenshteini kauguse variatsioone

Nagu juba mainitud, saab pikima ühise osajada leidmist vaadata kui Levenshteini kauguse erijuhtu, kus lubatud on ainult lisamised ja kustutamised (maksumusega 0), kattuvuse hinnaks on 1. Kui lubatud on ainut asendamised (stringid peavad siis muidugi olema sama pikkusega), nimetatakse sellist kaugust **Hammingi** kauguseks ning selle leidmine on triviaalne.

Mõnikord lisatakse operatsioonide hulgale veel **transpositsioon** ehk kahe kõrvuti asuva tähe vahetamine, uxyv→uyxv. Sellisel juhul omandab rekursioonivalem järgmise kuju:

$$LK_{a,b}(i,j) = \begin{cases} \max(i,j), kui \min(i,j) = 0 \\ \min \begin{cases} LK_{a,b}(i-1,j) + 1 \\ LK_{a,b}(i,j-1) + 1 \\ LK_{a,b}(i-1,j-1) + 0, kui\ a_i = b_j \\ LK_{a,b}(i-1,j-1) + 1, kui\ a_i \neq b_j \\ LK_{a,b}(i-2,j-2) + 1, kui\ a_{i-1}a_i = b_j b_{j-1} \end{cases} \end{cases}$$

8.4 GEENIJÄRJESTUSTE VÕRDLEMINE



1970. aastal kasutasid Saul B. Needleman ja Christian D. Wunsch esmakordselt dünaamilist planeerimist kasutavat algoritmi bioinformaatikas nukleotiidijärjestuste vastavusse seadmiseks ehk joondamiseks.

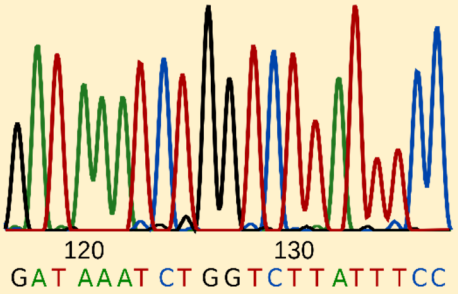
8.4.1 Ülesanne: Sarnasus

On antud kaks nukleotiidide järjestust ning need tuleb seada üksteisega vastavusse, nii et võimalikult palju järjenditest langeks kokku. Järjendite vastavusse seades on kolm võimalust:
1. Kattuvus – mõlema järjendi nukleotiid on sama
2. Mittekattuvus – kummaski järjendis on erinev nukleotiid
3. Kustutamine – ühes järjendis puudub vastav nukleotiid

Igal juhtumil on oma hind: iga kattuvus annab 1 punkti, iga mittekattuvus -1 ning iga kustutamine samuti -1. Leia nende järjendite sarnasus ehk maksimaalne hind.
Sisendi esimesel real on esimese järjestuse pikkus ja teisel real sellele vastav märgijada, mis koosneb märkidest A, C, G ja T ja U. Kolmandal real on teise järjestuse pikkus ja sellele vastav märgijada.
Väljastada üks täisarv: maksimaalne hind.

NÄIDE:

7
GCATGCU
7
GATTACA
Vastus:
0
Näiteks sobib järgmine joondus:
GCAT-GCU
G-ATTACA



Kuigi nimetused on teised, pole vast raske taibata, et mittekattuvus on sama, mis eelmises ülesandes asendamine. Erinevuseks on veel operatsioonide hind: originaalses Needlemani ja Wunschi töös kasutati sellist süsteemi nagu ülesandes – kattuvus 1 punkt, mittekattuvus ja kustutamine -1. Põhimõtteliselt on võimalik kasutada ükskõik millist punktisüsteemi, kuna sellest ei muutu algoritmi ega dp-tabeli täitmise loogika. Kasutades eelnevalt kirjeldatud süsteemi, saab hinnata kattuvust 0 punktiga ja mittekattuvust ning lisamist/kustutamist ühe punktiga. Nii saame tulemuseks Levenshteini kauguse. Seetõttu algoritm ise pikemat selgitamist ei vaja:

```
int NeedlemanWunsch(int n, int m, string s, string t)
{
    int** dp = new int*[n+1];
    for (int i = 0; i <= n; i++) {
        dp[i] = new int[m+1];
        for (int j = 0; j <= m; j++) {
            dp[i][j] = 0;
        }
    }

    int vas = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int hind = -1;
            if (s[i-1] == t[j-1]) {
                hind = 1;
            }
            dp[i][j] = max(max(dp[i-1][j] - 1, dp[i][j-1] - 1), dp[i-1][j-1] + hind);
        }
    }
    return dp[n][m];
}
```

8.4.2 Ülesanne: Mis on ühist?

On antud kaks geenijärjestust. Leida parima hinnaga joondatava alamosa hind, kui operatsioonide hinnad on järgmised:
kattuvus 3
mittekattuvus -3
kustutamine -2
Sisendi esimesel real on esimese järjestuse pikkus ja teisel real sellele vastav märgijada, mis koosneb märkidest A, C, G ja T ja U. Kolmandal real on teise järjestuse pikkus ja sellele vastav märgijada. Väljastada üks täisarv: maksimaalne lõigu hind.
NÄIDE:
9
GGTTGACTA
8
TGTTACGG
Vastus:
13

Kui Needleman-Wunschi algoritm leidis kahe järjendi parima joonduse, siis antud ülesandes on vaja leida kahest geenijärjestusest parima kattuvusega lõigud. Seetõttu nimetatakse esimest ka globaalseks joondamiseks ja teist, antud ülesannet, lokaalseks joondamiseks.

Lokaalse joondamise probleemiga tegelesid Temple F. Smith ja Michael S. Waterman, kes pakkusid 1981. aastal lahenduseks algoritmi, mis kannabki nüüd Smith-Watermani nime. Nagu aimata võid, on taas tegemist dünaamilise planeerimisega, täpsemalt jällegi Levenshteini kauguse modifikatsiooniga. Siin on oluline, et mitte soositud operatsioonid on negatiivse hinnaga, soositud (kokkulangemine) aga positiivsega. Kui mingil hetkel läheb tehingu väärtus nulli, siis seda enam ei vähendata, vaid hakatakse otsima uut parimat kokkulangevat juppi.

8.4.3 Rekursioonivalem

Tähistame kustutamise/lisamise eest lahutatavat hinda LH-ga (antud ülesandes -2) ning kattuvuse hinda KH-ga (3) ja mittekattuvuse hinda MH-ga (-3), siis:

$$SW_{a,b}(i,j) = \begin{cases} 0, kui\ i = 0\ V\ j = 0; \\ \max \begin{cases} SW_{a,b}(i-1,j) + LH \\ SW_{a,b}(i,j-1) + LH \\ SW_{a,b}(i-1,j-1) + KH, kui\ a_i = b_j \\ SW_{a,b}(i-1,j-1) + MH, kui\ a_i \neq b_j \\ 0 \end{cases} \end{cases}$$

8.4.4 Tabel

Näiteandmete põhjal tekib järgmine dp tabel:

	[]	G	G	T	T	G	A	C	T	A
[]	0	0	0	0	0	0	0	0	0	0
T	0	**0**	0	3	3	1	0	0	3	1
G	0	3	**3**	1	1	6	4	2	1	0
T	0	1	1	**6**	4	4	3	1	5	3
T	0	0	0	4	**9**	**7**	5	3	4	2
A	0	0	0	2	7	6	**10**	8	6	7
C	0	0	0	0	5	4	8	**13**	11	9
G	0	3	3	1	3	8	6	11	10	8
G	0	3	6	4	2	6	5	9	8	7

8.4.5 Kood

```
int SmithWaterman(int n, int m, string s, string t)
{
    int** dp = new int*[n+1];
    for (int i = 0; i <= n; i++) {
        dp[i] = new int[m+1];
        for (int j = 0; j <= m; j++) {
            dp[i][j] = 0;
        }
    }

    int vas = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            int sc = -3; // ei lange kokku
            if (s[i-1] == t[j-1]) {
                sc = 3; // langeb kokku
            }
            dp[i][j] =
                max(max(dp[i-1][j] - 2, dp[i][j-1] - 2), max(0, dp[i-1][j-1] + sc));
            vas = max(vas, dp[i][j]);
        }
    }
    return vas;
}
```

8.4.6 Tee taastamine

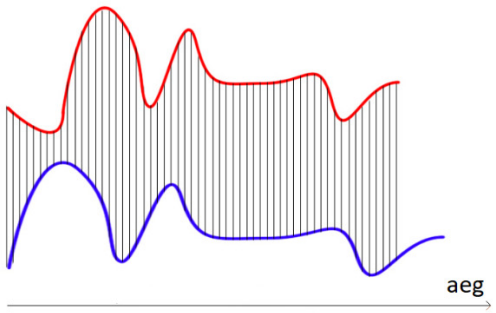
Tee taastamiseks tuleb leida suurima väärtusega lahter tabelist (seda on kaval jooksvalt meeles pidada) ning sealt saab juba tagurpidi tagasi tulles leida operatsioonid, mida kasutati. Kui jõutakse lahtrisse, kus on `0`, siis on algus leitud ja rohkem kuhugi liikuda pole vaja. Seega vastus ülesandele võib asuda suvalises tabeli lahtris ning taastamisteegi ei pea jooksma ilusti tagasi tabeli algusesse, nagu klassikalisele toimetamiskauguse algoritmile omane. Ülesande lahenduseks on näiteks järgmine joondus (parim kokkulangev lõik on märgitud rohelisega):

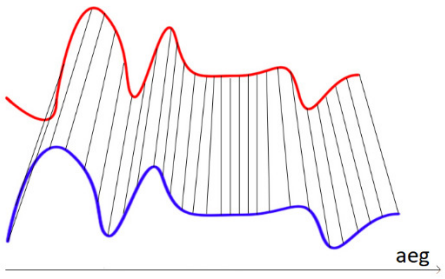
```
GGTTGACTA
TGTT-ACGG
```

8.5 Dünaamiline Aja painutamine (Dynamic Time Warping)

Dünaamiline aja painutamine (i.k. *dynamic time warping, DTW*) on algoritm kahe aegrea sarnasuse hindamiseks. DTW algoritmi kasutati algselt just automaatses kõnetuvastuses, et hakkama saada erinevatel kiirustel rääkimisega.

Näiteks, kui sarnased on järgmistel joonistel toodud punane ja sinine funktsioon? Kui proovida võrrelda omavahel punke, mis ajas täpsel kokku langevad (esimene joonis), siis tundub, et üsna erinevad.





Parempoolsel joonisel aga võrreldakse ajas veidi üksteise suhtes nihkes olevaid punkte, mis seataksegi vastavusse nii, et erinevus oleks võimalikult väike. See saavutatakse nii, et vahepeal on aeg justkui kokkusurutud või väljavenitatud – ühele kitsale lõigule ajas vastab teisest funktsioonist hoopis laiem lõik.

8.5.1 Ülesanne: helid

On antud kaks heli aegridadena – uuritav rida ja etalon. Leia minimaalne summaarne erinevus nende ridade vahel.
Sisendi esimesel real on arv `m` – uuritava rea pikkus millisekundites ja teisel real on m arvu – igale millisekundile vastav kõrgus. Kolmandal real on arv `n` – etalonrea pikkus - ja neljandal real `n` arvu. Kõik andmed on täisarvudena.
NÄIDE:

```
6
1 1 2 3 2 0
7
0 1 1 2 3 2 1
```

Vastus:

```
2
```

Ka selle ülesande lahendamiseks saab kasutada dünaamilist planeerimist ning täpsemalt Levenshteini kauguse modifikatsiooni. Kirjeldatud kokkusurumistele ja venitamistele vastavad kustutamised ja lisamised ning asendamisi ei toimu. Operatsiooni hinnaks on punktide väärtuste kaugus.

8.5.2 Rekursioonivalem

Olgu $d(a_i, b_j) = |a_i - b_j|$, siis

$$T(i,j) = \begin{cases} 0, & kui\ i = 0\ ja\ j = 0 \\ \infty, kui\ i = 0\ või\ j = 0 \\ min \begin{cases} T(i-1,j) + d(a_i, b_j) \\ T(i-1,j-1) + d(a_i, b_j) \\ T(i,j-1) + d(a_i, b_j) \end{cases} \end{cases}$$

8.5.3 Tabel

Ülesande näide annab järgmise DP-tabeli:

	0	1	1	2	3	2	0
0	**0**	∞	∞	∞	∞	∞	∞
0	∞	**1**	2	4	7	9	9
1	∞	**1**	1	2	4	5	6
1	∞	1	**1**	2	4	5	6
2	∞	2	2	**1**	2	2	4
3	∞	3	4	2	**1**	2	5
2	∞	4	4	2	2	**1**	3
1	∞	4	4	3	4	2	**2**

8.5.4 Kood

```
int DTW(int n, int m, vector<int>, vector<int> t)
{
    int** dp = new int*[n];
    for (int i = 0; i < n; i++)
    {
        dp[i] = new int[m];
        for (int j = 0; j < m; j++) {
            dp[i][j] = 0;
        }
    }
    for (int i = 0; i < n; i++) {
        dp[i][0] = MAXINT;
    }
    for (int i = 0; i < m; i++) {
        dp[0][i] = MAXINT;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]);
            dp[i][j] = min(dp[i][j], dp[i - 1][j - 1]);
            dp[i][j] += kaugus(i, j);
        }
    }
    return dp[n][m];
}
```

8.6 LÕIKUDEGA ÜLESANDED

8.6.1 Ülesanne: palgi lõikamine

Töökoda pakub teenust, et kliendi palgist lõigatakse talle sobivast kohast sobiva pikkusega jupid. Kuna töökojas on vaid üks ühe teraga saag, saab palki lõigata korraga ühest kohast. Palgi lõikamise kulu sõltub sellest, kui pikk on parasjagu lõigatav palk. Seega, kui kliendil on 10m pikkune palk ja ta soovib saada lõiked vasakult 1 m kauguselt, 3 m kauguselt ja 5m kauguselt, siis esimene kord on vaja lõigata 10-meetrist palki, teise lõike tegemisel aga sõltub, millisest kohast lõigati esimene kord. Kui lõigati 1m kauguselt, tuleb järgmine lõige teha 9-meetrisesse palki, kui aga näiteks 5m kauguselt, siis 5-meetrisesse palki. Töökoda küsib iga meetri lõigatava palgi eest 1 ühiku raha. Leia minimaalne summa, mis etteantud palgi lõikamiseks kulub.
Sisendi esimesel real on kaks täisarvu palgi pikkus p ja lõigete arv n. Teisel real on n täisarvu – lõikekohtade kaugused palgi vasakust servast.
Väljundisse kirjutada üks arv – minimaalne hind, mis tuleb palgi jupitamise eest tasuda.
NÄIDE:

```
10 4
4 5 7 8
```

Vastus:

```
22
```

8.6.2 Tee DP-ni

Alustame taas jõumeetodile mõtlemisega. Võimalikke lõikekohti on n ja tuleks läbi proovida kõik võimalikud järjestused. Iga lõige jaotab palgi täpselt kaheks jupiks, mida tuleb siis samal moel edasi tükeldada, kuni kõik lõiked on tehtud. Kuigi see lähenemine töötab, on selle keerukuseks $O(n!)$.

Siingi hakkavad alamprobleemid kattuma, sest kui mingil hetkel on järel jupp lõikekohast i kuni lõikekohani j, siis pole tegelikult vahet, mil moel selle seisuni jõuti – optimaalse lahenduse saamiseks on vaja lõigata see tekkinud jupp võimalikult väikese kuluga.

8.6.3 Rekursioonivalem

Tähistame $h(i,j)$ parimat hinda lõikekohast i kuni lõikekohani j, kus $0 \le i < j \le n+1$. 0 on palgi vasak ja $n+1$ parem otspunkt. Otspunktide vahel on lõikekohad $1 \dots n$, järjestatud rangelt vasakult paremale. Sellisel juhul

$$h(i,j) = \begin{cases} 0, kui\ i+1=j \\ p(i,j) + \min\limits_{\forall k \in \{i+1,\dots,j-1\}} \big(h(i,k)+h(k,j)\big) \end{cases}$$

kus $p(i,j)$ on palgi pikkus i. lõikepunktist j. lõikepunktini.

8.6.4 Tabeli täitmine

Kasutame rekursioonivalemit DP tabeli täitmiseks. Siin tuleb tähele panna, et alustada tuleb lühematest lõikudest, kuna neid kasutatakse pikemate lõikude arvutamiseks. See tähendab, et tabelit täidetakse sisuliselt mööda diagonaali – lühemad lõigud asuvad diagonaali lähedal, pikemad sellest eemal. Praktikas on lihtsam täita veerg-haaval, täites iga veeru diagonaalist ülespoole liikudes. Ka vastus ei ole tavapärases kohas ehk tabeli lõpus, vaid hoopis esimese rea viimases veerus:

	1 (4)	2 (5)	3 (7)	4 (8)	5 (10)
0 (0)	0	5 + min([0,1],[1,2]) = 5 + min(0, 0) = **5** L=1	7 + min([0,1]+[1,3], [0,2]+[2,3]) = 7+ min(0+3, 5+0) = **10** L=1	8 + min([0,1]+[1,4], [0,2]+[2,4], [0,3]+[3,4]) = 8 + min(0+7, 5+3, 10+0) = **15** L=1	10 + min([0,1]+[1,5], [0,2]+[2,5], [0,3]+[3,5], [0,4]+[4,5]) = 10 + min(0+12, 5+8, 10+3, 15+0) = **22** **L=1**
1 (4)		0	3 + min([1,2]+[2,3]) = 3 + min(0+0) = **3** **L=2**	4 + min([1,2]+[2,4], [1,3]+[3,4]) = 4 + min(0+3, 3+0) = **7** **L={2,3}**	6 + min([1,2]+[2,5], [1,3]+[3,5], [1,4]+[4,5]) = 6 + min(0+8, 3+3, 7+0) = **12** **L=3**
2 (5)			0	3 + min([2,3]+[3,4]) = 3 + min(0+0) = **3** **L=3**	5 + min([2,3]+[3,5], [2,4]+[4,5]) = 5 + min(0+3, 3+0) = **8** **L={3,4}**
3 (7)				0	3+ min([3,4]+[4,5]) = 3 + min(0+0) = **3** **L=4**
4 (8)					0

Iga arvutatud lahtri viimasel real on toodud ka lõikekoht (või lõikekohad), mis parima tulemuse andis. Nii on lihtne leida, mis järjekorras tuleb palki tegelikult tükeldada.

8.6.5 Kood

```
int loika(int pikkus, int loikeid, int* loikekohad) {

    int** dp = new int*[loikeid + 4];
    for (int i = 0; i < loikeid + 4; i++) {
        dp[i] = new int[loikeid + 4];
    }
    for (int i = 0; i < loikeid + 4; i++) {
        for (int j = 0; j < loikeid + 4; j++) {
            dp[i][j] = MAXINT;
        }
    }

    for (int i = 0; i <= loikeid + 1; i++) {
        dp[i][i] = 0;
        dp[i][i + 1] = 0;
        dp[i][i + 2] = loikekohad[i + 2] - loikekohad[i];
    }

    for (int k = 3; k <= loikeid + 1; k++) {
        for (int i = 0; i <= loikeid + 1 - k; i++) {
            for (int j = i + 1; j <= i + k - 1; j++) {
                if ((dp[i][j]+dp[j][i+k]+loikekohad[i+k]-loikekohad[i])<dp[i][i+k]) {
                    dp[i][i+k] = dp[i][j]+dp[j][i+k]+loikekohad[i+k]-loikekohad[i];
                }
            }
        }
    }

    return dp[0][loikeid+1];
}
```

8.6.6 Tee taastamine

Tee taastamiseks on vaja teada, milline lõige andis vaadeldava miinimumi. Kui vaatame tabeli ülemist parempoolset lahtrit ehk seda, kus on terve palgi jupitamine ja seega kogu ülesande vastus, siis see jaotus, mis andis tol sammul miinimumi, ongi esimeseks lõikeks optimaalse lahenduse puhul. Järgmiste lõigete jaoks tuleb siis asuda järelejäänud juppe uurima.
Kuigi on võimalik see miinimum tagurpidi taastada (proovime taas kõik lõiked läbi ja vaatame, milline neist andis miinimumi), siis juhul, kus taastamine on vajalik, on sageli kasulikum meeles pidada see üks arv – optimaalse lõike asukoht.

Antud näites siis on optimaalseks hinnaks 22 ja see saadi siis, kui viimane lõige tehti 1. lõikekohta. Seega järgmiseks lõikeks jäävad järele jupid 0-1 ja 1-5. Mis järjekorras neid edasi tükeldada, ei oma tähtsust. Antud juhul esimest juppi rohkem tükeldada pole vajagi. Teise tüki jaoks vaatame tabelis rea 1 veergu 5. Seal on näha, et minimaalse hinna (12) andis lõige 3. lõikekohast. Seega jääb vaadata juppe 1-3 ja 3-5. Neil on nagunii vaid täpselt üks lõikekoht ja nagu öeldud – järjekord ei ole oluline. Seega sobivad lahenduseks lõiked järjestuses 1, 3, 2, 4 ja 1, 3, 4, 2.

Kontrollime, kas esimene järjestus annab oodatud tulemuse: Alguses on palgi pikkus 10 ja peame arvestama kuluga 10. Lõigates 1. lõikekohast, saame palgid pikkusega 4 ja 6. Esimesel neist lõikekohti rohkem pole. Teise palgi lõikame lõikekohast 3 ja saame kaks palgijuppi pikkusega 3. Kuna lõigatava palgi pikkus oli 6, teeb see kuluks 10+6=16. Nüüd on vaja jupitada veel mõlemad 3-se pikkusega palgitükid. Täpne lõikekoht polegi enam oluline, kuna lõikamise kulu on igal juhul 3, seega on kogukulu 16+3+3 =22.

8.6.7 Keerukus

Mahuline keerukus on $O(n^2)$. Ajalise keerukuse leidmiseks tuleb tähele panna, et ühe tabeli lahtri arvutamiseks võib vaja minna kuni n operatsiooni, seega on ajaliseks keerukuseks $O(n^3)$.

8.7 MAATRIKSITE KORRUTAMINE

Paljud ülesanded füüsikast majandusteooriani taanduvad lineaaralgebrale, kus omakorda on sagedaseks ülesandeks erinevate maatriksite korrutamine. Seetõttu on ka vastavaid algoritme palju uuritud ja optimiseeritud.

8.7.1 Ülesanne: maatriksid

On antud n maatriksit A_1, A_2…A_n. i. maatriksi suurus on P_{i-1}*P_i. Leida vähim operatsioonide arv, mida on vaja teha, et maatriksid omavahel korrutada. Kui maatriks A on mõõtmetega n*m ja maatriks B on mõõtmetega m*k, siis AxB on mõõtmetega n*k ning korrutamiseks vajalike operatsioonide arv on n*m*k.

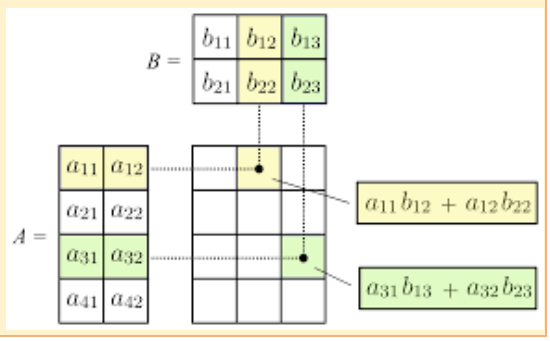
Sisendi esimesel real on antud maatriksite arv n.
Järgmisel real on antud n+1 tühikuga eraldatud arvu P_0…P_n, kus P_{i-1} ja P_i tähistavad i. maatriksi mõõte.
Väljastada vähim operatsioonide arv.

NÄIDE:

```
5
5 2 7 4 3 5
```

Vastus:

```
160
```



8.7.2 Võimaluste arv

Esimese hooga tuleb ehk tahtmine proovida kõikvõimalikke variante. Näiteks nelja maatriksi korrutamiseks on 5 varianti:
`((ab)c)d, (a(bc)d), (ab)(cd), a((bc)d)` ja `a(b(cd))`.

Võimalike erinevate variantide arv on C_{n-1}, kus C_i tähistab i. Catalani arvu ja n maatriksite arvu. Catalani arv avaldub

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k}$$

Ja esimesed 20 Catalani arvu on:
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190.

See tähendab, et kõigi võimaluste läbivaatamine tuleb kõne alla vaid siis, kui korrutatavate maatriksite arv on väga väike.

8.7.3 Tee DPni

Vaatame mingit teatud pikkusega alamosa maatriksite jadast. Olgu esimene maatriks selles A_i ja viimane A_j. Tähistame korrutise $A_i \times A_{i+1} \times \ldots \times A_j$ leidmiseks vajalike tehete arvu $op(i,j)$. Siis

$$op(i,j) = \begin{cases} 0, kui\ i = j \\ \min\limits_{\forall k \in \{i,\ldots,j\}} (op(i,k) + op(k+1,j) + P_{i-1} \cdot P_k \cdot P_j) \end{cases}$$

8.7.4 Tabeli täitmine

Selle ülesande juures tuleb tähele panna, et tabeli peadiagonaalil, kus i=j, asuvad 0-väärtused. Tabeli täitmisel liigutakse peadiagonaalist eemale ning vastus on hoopis kohal [1, n], mis loogiliselt vastabki maatriksite jadale $A_i \ldots A_n$.

	A1	A2	A3	A4	A5
A1	0				
A2	[A1,A1] + [A2,A2] + P0*P1*P2 = **70**	0			
A3	min([A1,A2]+[A3,A3]+P0*P2*P3; **[A1,A1]+[A2,A3] + P0*P1*P3**) = min(70+140, **56+40**) = **96**	[A2,A2] + [A3,A3] +P1*P2*P3= **56**	0		
A4	min([A1,A3]+[A4,A4]+P0*P3*P4; [A1,A2]+[A3,A4]+P0*P2*P4; **[A1,A1]+[A2,A4]+P0*P1*P4**)=min(96+60, 70+84+105, **80+30**) =**110**	min(**[A2,A3]+[A4,A4]]+ P1*P3*P4**; [A2,A2]+[A3,A4]+P1*P2*P4)=min(**56+24**,84+42)=**80**	[A3,A3] + [A4,A4] + P2*P3*P4 = **84**	0	
A5	min(**[A1,A1]+[A2,A5]+P0*P1*P5**; [A1,A2]+[A3,A5]+P0*P2*P5; [A1,A3]+[A4,A5]+P0*P3*P5; [A1,A4]+[A5,A5]+P0*P4*P5)= min(**110+50**; 70+189+175; 96+60+100; 110+75)=**160**	min(**[A2,A4]+[A5,A5]+P1*P4*P5**; [A2,A3]+[A4,A5]+P1*P3*P5; [A2,A2]+[A3,A5]+P1*P2*P5)= min(**80+30**, 56+60+40, 189+70)= **110**	min(**[A3,A4]+[A5,A5]+ P2*P4*P5**; [A3,A3]+[A4,A5]+P2*P3*P5) = min(**84+105**,60+140)= **189**	[A4,A4] + [A5,A5] + P3*P4*P5 = **60**	0

Kui huvitab, milline korrutis siis sellise tulemuse andis, siis tabelis on märgitud rasvaselt ka see summa, mis miinimumini viis. Lahtrist [A1,A5] näeb, et miinimum tuli summast [A1,A1]+[A2,A5]+P0*P1*P5 ehk siis maatriksi A1 korrutisest maatriksite A2…A5 korrutisega. A2 ja A5 parim korrutis tuli lahti [A2,A5] põhjal maatriksi A5 korrutisest maatriksite A2..A4 korrutisega. A2…A4 saab vähimate operatsioonide arvuga leida, korrutades maatriksi A4 maatriksite A2 ja A3 korrutisega. Seega parima tulemuse andis korrutis

$$A_1 \times \Big(\big((A_2 \times A_3) \times A_4 \big) \times A_5 \Big).$$

8.7.5 Kood

Siin on funktsioon, mis antud tabeli täidab:

```
int matMul(int n, vector<int> ml)
{
    int** dp = new int*[n+1];
    for (int i = 0; i <= n; i++) {
        dp[i] = new int[n+1];
    }

    for (int vh = 0; vh <= n; vh++) { // vh - vahe ehk kui pikki lõike täidetakse
        for (int i = 1; i + vh <= n; i++) {
            int j = i + vh;
            dp[i][j] = (i == j) ? 0 : 1000000000;
            for (int k = i; k < j; k++) { // iga lõikekoha k jaoks
                dp[i][j] = min(dp[i][j], dp[i][k]+dp[k+1][j]+(ml[i-1]*ml[k]*ml[j]));
            }
        }
    }
            return dp[1][n];
}
```

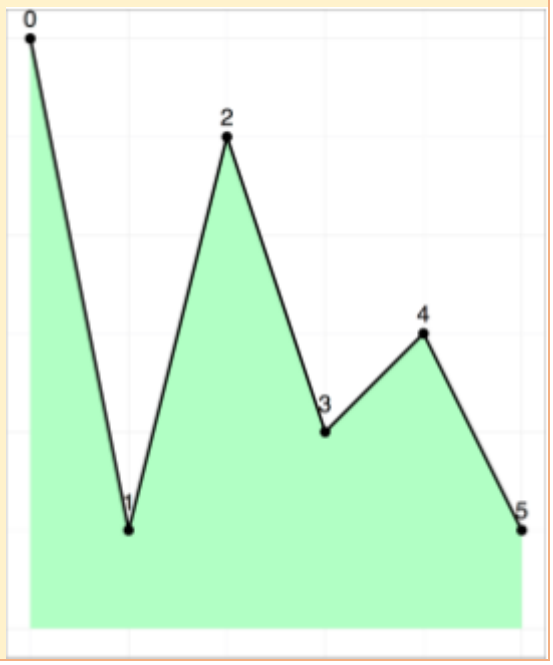
8.7.6 Keerukus

Mahuline keerukus on $O(n^2)$. Ajalise keerukuse leidmiseks tuleb tähele panna, et ühe tabeli lahtri arvutamiseks võib vaja minna kuni n operatsiooni, seega on ajaliseks keerukuseks $O(n^3)$.

8.8 Ülesanne: mäed

Järgnev ülesanne oli 2017. aastal Teheranis toimunud rahvusvahelise informaatikaolümpiaadi proovivoorus:

Kolmas Pärsia kuningas Tahmuras on võitnud lahingu suure deevade (deemonite) armeega. Ta soovib vangistada nii palju deevasid kui võimalik Alborzi mägedesse ja lasta ülejäänud vabaks. Alborz on mäeahelik, mis näeb välja nagu n-tipuga ($1 \le n \le 2000$) hulknurkne ahel. i. tipu koordinaadid on `(i, y[i])`, kus `i` on kaugus aheliku algusest ja `y[i]` ($0 \le y[i] \le 10^9$) tipu kõrgus.
Deevasid saab vangistada erinevatesse tippudesse. Ainus tingimus on, et deevad ei tohi üksteist näha, sest siis saavad nad teha põgenemisplaane. Kaks deevat ei näe teineteist, kui nende vahel on vähemalt üks tipp, mis on rangelt kõrgemal kui nende deevade asukoha tippude vahele tõmmatud joon.
Järgneval joonisel deeva tipus 0 näeb deevasid tippudes 1 ja 2, aga ei näe neid, kes asuvad tippudes 3, 4 ja 5, kuna tipp 2 on kõrgemal ja varjab vaate tipust 0 tippudesse 3, 4 ja 5.
Aita Tahmurasel leida, kui mitu deevat saab ta Alborzi mäeahelikku vangistada. Sisendi esimesel real on mäetippude arv N. Järgmisel N real on tippude kõrgused.



***NÄIDE 1** (joonisel):*
```
6
6 1 5 2 3 1
```
Vastus:
3 (Näiteks tippudele 1, 3 ja 5)
NÄIDE 2:
```
3
0 1 2
```
Vastus:
1

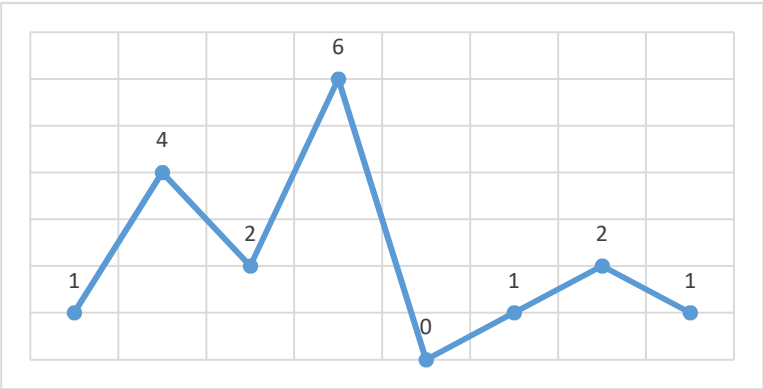
Esimese asjana on meil vaja vastust küsimusele kas tipp i paistab tipust j? Näitame hiljem ülesande lahenduses, kuidas seda teha, esialgu lihtsalt eeldame, et meil on nxn maatriks, mis vastab antud küsimusele. Näiteandmete puhul on see siis järgmine:

	0	1	2	3	4	5
0	1	1	1	0	0	0
1	1	1	1	0	0	0
2	1	1	1	1	1	0
3	0	0	1	1	1	0
4	0	0	1	1	1	1
5	0	0	0	0	1	1

Pealtnäha ei ole võimalik seda ülesannet lahendada muud moodi, kui jõumeetodiga proovides. Kuid tuleb tähele panna, et kui proovime deeva panna mingisse tippu k, ei saa teisi deevasid olla neis tippudes, mis on näha tipust k, aga deevad saavad olla nende nähtavate tippude vahelistel lõikudel.

Eraldi lõigud tekivad siis, kui nende vahel on nii kõrge punkt, et kumbki pool ei saa teist näha, mistõttu need kaks alamosa ei mõjuta üksteist. Seega, kui me paneme ühe deeva tippu k, siis maksimaalselt saame paigutada mäestikusse nii mitu deevat, kui on igale tipust k nähtamatule

tippude lõigule maksimaalselt paigutatavate deevade summa pluss veel see deeva, mille just paigutasime tippu k.



Näiteks kui panna deeva joonisel viimasesse, 8. tippu (kõrgusega 1), siis on sealt näha 4. tipp kõrgusega 6 ja 7. tipp kõrgusega 2. Deevasid saab paigutada veel lõikudele [1,3] ja [4,5], kuna need ei ole nähtavad viimasest tipust. Nagu jooniselt näha, saab esimesele neist [1,3] paigutada 2 deevat.

8.8.1 Rekursioonivalem

Tähistagu $D(i,j)$ seda, mitu deevat saab panna tippude lõigule i-st j-ni vastavalt ülesande tingimustele. On lihtne taibata, et ühetipulisele lõigule ($i = j$) saab panna täpselt ühe deeva. Tegelikult ka kahetipulisele lõigule saab panna täpselt ühe deeva, kuna teine tipp on esimesest ja esimene tipp teisest tipust näha.

Olgu k tipp, mis asub lõigul $[i,j]$ ehk $i \leq k \leq j$. Tähistagu LK_{ijk} selliste maksimaalse pikkusega mittekattuvate lõikude hulka, kus iga lõik $[h,g] \in [i,j]$ ja ükski lõigu $[h,g]$, $h \leq g$ punktidest ei ole tipust k nähtav. Saame järgmise rekursioonivalemi:

$$D(i,j) = \begin{cases} 1, kui\ i = j \\ 1 + \max\limits_{\forall k \in \{i \ldots j\}} \sum\limits_{\forall [h,g] \in LK_{ijk}} D(h,g) \end{cases}$$

8.8.2 Tabeli täitmine

Tabeli täitmine sarnaneb kahele eelmisele ülesandele:

	0	1	2	3	4	5
0	**1**	1+max(0,0) = **1**	1+max(0,0,0) =**1**	1+max([3,3], [3,3], 1, [0,1]) =**2**	1+max([3,4], [3,4], 0, [0,1] [0,1]) =**2**	1+ max([3,5], [3,5], [5,5], [0,1]+[5,5], [0,1], [0,3]) =**3**
1		**1**	1+max(0,0) = **1**	1+max([3,3], 0, [1,1]) = **2**	1+max([3,4], 1, [1,1], [1,1]) =**2**	1+max([3,5], [5,5], [1,1]+[5,5], [1,1], [1,3]) = **3**
2			**1**	1+max(0,0) = **1**	1+max(0,0,0) = **1**	1+max([5,5], [5,5], 0, [2,3]) = **2**
3				**1**	1+max(0,0) = **1**	1+max([5,5], 0, [3,3]) =**2**
4					**1**	1+max(0,0) = **1**
5						**1**

8.8.3 Kood

Koodi juures tuleb juhtida tähelepanu kahele asjale: esiteks ei ole antud maagilist nähtavuse tabelit ja see, mis tipud mingist tipust näha on, tuleb ise arvutada. Teiseks, kuna tabeli täitmisel tuleks päris palju kordi järgi vaadata, millised järjestikused tipud ei ole nähtavad, siis on lihtsam hoida iga tipu kohta meeles, millised lõigud sellest tipust nähtavad ei ole.

```
int maximum_deevs(std::vector<int> y)
{
    int n = y.size();
    vector<pair<int, int> >* nl = new vector<pair<int, int> >[n];
    for (int i = 0; i < n; i++) {
        vector<pair<int, int> > vi;
        nl[i] = vi;
    }
    int** dp = new int*[n];
```

```
    for (int i = 0; i < n; i++)
    {
        dp[i] = new int[n];
        double bs = -3; // (-pi)
        int ch = y[i]; // vaadeldava tipu kõrgus
        for (int j = i + 1; j < n; j++) {
            int nh = y[j]; // järgmine tipp
            int cy = nh - ch; // kõrguste vahe
            int cx = j - i; // kauguste vahe
            double cs = atan2(cy, cx); // tippude vaheliste sirgete nurk radiaanides
            if (cs >= bs) { // nurk on suurem, kui seni leitud
                bs = cs;
                int nljs = nl[j].size();
                if (nljs > 0 && nl[j][nljs - 1].second == i - 1) {
                    nl[j][nljs - 1].second = i;
                    continue;
                }
                nl[j].push_back(make_pair(i, i));
            }
        }
    }

    for (int i = 0; i < n; i++) { // esimesest tipust viimaseni
        for (int j = i; j >= 0; j--) { // vaatame eelmisi tippe
            if (i == j) {
                dp[j][i] = 1; // ühetipulisele lõigule saab panna 1 deemoni
                continue;
            }
            //saame panna lõigule j..i vähemalt sama palju deemone, kui lõigule j..i-1
            dp[j][i] = dp[j][i - 1];
            int ov = 1;
            int si = j;
            for (int k = 0; k < nl[i].size(); k++) {
                if (nl[i][k].first < si) { // kui algus on enne vaadeldavat indeksit
                    continue;
                }
            // kui on esimene lõik või eelmise lõigu lõpp on enne vaadeldavat tippu
                if (k == 0 || nl[i][k - 1].second < si) {
                     if (nl[i][k].first <= si && nl[i][k].second >= si) {
                        continue; // oleme lõigu sees, ei muutu midagi
                    }
                    // liidame need lõigud juurde, mis tuleb arvesse võtta
                    ov += dp[si][nl[i][k].first - 1];
                    continue;
                }
                if (nl[i][k].first - 1 == nl[i][k - 1].second) {
                    continue;
                }
                ov += dp[nl[i][k - 1].second + 1][nl[i][k].first - 1];
            }
            dp[j][i] = max(dp[j][i], ov);
        }
    }
    int vas = dp[0][n - 1];
    return vas;
}
```

8.8.4 Keerukus

Mahuline keerukus on muidugi taas tabeli suurus ehk $O(n^2)$. Lahtri täitmiseks on vaja vaadata kõiki eelnevaid tippe ehk teha kuni n arvutust, kogu ajaline keerukus on seega $O(n^3)$.

8.9 KONTROLLÜLESANDED

8.9.1 Torn

Külma sõja ajal üritasid USA ning NSVL üksteist igas asjas üle trumbata. Seega, kui Leonid Brežnev kuulis, et üks Ameerika koolipoiss on ehitanud maailma kõrgeima pappkastidest koosneva torni, teatas ta kohe, et Nõukogude Liit ei saa jääda teiseks. Oma pappkasttorni ehitamiseks said nõukogude teadlased `N(1≤N≤1000)` erinevat sorti kasti, mis on nummerdatud arvudega `1...N` ning mida on lõputul hulgal saadaval. Iga kasti kohta on teada selle kaal ning maksimaalne kaal, mida see kannatada suudab. Kastide paigutamiseks on järgmised reeglid: väiksema numbriga kasti ei tohi suurema numbriga kasti peale panna ning ühegi kasti peal ei tohi kokku olla rohkem raskust, kui see kannatada suudab. Leida maksimaalne arv kaste, mida on võimalik üksteise peale laduda.
Sisendi esimesel real on arv `N`. Järgmisel `N` real on kaks tühikutega eraldatud täisarvu: kasti kaal ning kandmisvõime.
Väljastada üks täisarv: maksimaalne arv kaste, mida on võimalik üksteise peale laduda.



NÄIDE:

```
5
19 15
7 13
5 7
6 8
1 2
```

Vastus:

```
4
```

8.9.2 Bussireisid

Bussifirma tahab saata ühe bussi mööda teedevõrku sõitma, nii et see jõuab lõpuks ka bussijaama tagasi. Teedevõrgus on $n(1 \le n \le 50)$ ristmikku, $m(1 \le m \le 1000)$ kahesuunalist teed kahe ristmiku vahel, mille kohta on teada sõidukulu teed mööda sõitmiseks ning $p(1 \le p \le 12)$ peatust, mis asuvad ristmikutel ning mille kohta on teada, kui palju tulu saab firma, kui buss sinna sõidab. Buss alustab teekonda ristmikult nr 0. Leida suurim kasum, mida firma saab teenida.
Sisendi esimesel real on kaks tühikuga eraldatud täisarvu: n ja m. Järgmisel m real on kolm tühikutega eraldatud täisarvu: tee vasak otspunkt, tee parem otspunkt ning tee sõidukulu sentides. Siis on üksikul real täisarv p. Järgmisel p real on kaks tühikuga eraldatud täisarvu: peatuse asukoht ning peatuses peatumise eest saadav tulu sentides.
Väljundisse kirjutada üks täisarv: firma maksimaalne kasum sentides.

NÄIDE 1:

```
4 5
0 1 100
1 2 300
1 3 200
2 4 400
3 4 325
3
2 150
3 700
4 900
```

Vastus:

```
350
```



NÄIDE 2:

```
1 1
0 1 150
1
1 299
```

Vastus:

```
0
```

8.9.3 Sillad

Pärast üht looduskatastroofi on Räniorg jagunenud kaheks osaks, mille vahel on kuristik. Kummalegi poole kuristikku on jäänud firmad, millest igaühel on oma tootlus ning usk mingisse IT suurtegijasse. Otsustati, et oleks hea mõte ehitada kuristikule firmade vahele sildasid selleks, et nad saaksid ühineda ning oma tootlust kahekordistada. Samas on teada, et ei ole hea mõte ühendada erineva usuga firmasid ning sillad üksteist ületada ei saa. Aidake firmajuhtidel leida, kuidas maksimeerida tootlikkuse kasvu ning mis on vähim vajalik sildade arv, et nõutud tootlikkust saada.
Sisendi esimesel real on arv $v(1 \le v \le 1000)$, vasakul äärel olevate firmade arv. Järgmised v rida sisaldavad kolm tühikutega stringi: firma nimi, firma kirikupea ning firma tootlikkus. Firmad on antud nende paiknemise järjekorras serval.
Nendele järgneb arv $p(1 \le p \le 1000)$ - paremal äärel olevate firmade arv. Järgmised p rida sisaldavad nende firmade informatsiooni samas stiilis, kui vasakul äärel olevate firmade kohta.
Väljastada kaks tühikutega eraldatud täisarvu: maksimaalne võimalik tootlikkuse kasv ning selleks minimaalne sildade arv.

NÄIDE :

```
3
Apple Jobs 1000000
Lääne-Google Lee 1000
Micro Gates 400
4
Googol Lee 5000
Pear Jobs 1200
Soft Gates 100
Ida-Google Lee 50
```

Vastus:

```
1002250 2
```



8.9.4 Mäng

4*4 laual mängitakse järgmist mängu: mängija saab kas katta ühe vabalt valitud ruudu või valida kaks või kolm järjestikust tühja ruutu, millest moodustatud rida on ühe otsaga vastu ruudustiku serva, ning kõik selles reas olevad ruudud katta. Kaotab see, kes katab kogu ruudustiku. Kirjutada programm, mis vaatab juba kaetud ruute, ning ütleb, kas antud seis on selline, et parima strateegia korral võidab praegu käija või teisena käija.
Sisendi esimesel real on arv n(1≤n≤100 000), erinevate laudade arv. Järgmistel ridadel on n 4*4 lauda, kus 'X' tähistab kaetud ruutu ning '.' katmata ruutu. Iga laua ees on tühi rida.
Väljastada n rida. i-ndal real on kirjas üks sõna: ALUSTAJA, kui praegu käija võidab või TEINE, kui teisena käija võidab.
NÄIDE:

```
3

XXX.
XXX.
.XXX
.XXX

XXXX
…X
XX.X
XX.X

….
….
….
….
```

Vastus:

TEINE
ALUSTAJA
TEINE

8.9.5 Uuring

KAPO on oma korruptsiooniuuringutega saanud kätte ühe märkmiku, kuhu on kirja pandud kõik tehingud, mida kahtlusalune organisatsioon on viimase kuu jooksul teinud. Iga tehing on mingi arv, mis näitab, kas organisatsioon on kulutanud tehinguga x eurot ära või saanud x eurot juurde. Iga lehekülje lõpus on märgitud kõigi tehingute summa ehk voog. Näiteks kui on leheküljel olevate tehingutega saadud 7€, saadud veel 2€, kulutatud 3€, saadud 1€ ning kulutatud 11€, on lehekülje lõpus arv 7 + 2 – 3 + 1 – 11 = -4. Samas ei ole iga tehingu juurde märgitud, kas see oli kulu või tulu. Kuna KAPO tahab seda teada, on vaja kirjutada programm, mis leiab kas mingi tehing oli kulu, tulu või ei ole teada, kumb tehing see on.

Sisendi esimesel real on kaks tühikuga eraldatud arvu: `N (2≤N≤40)` ning `F (-16000≤F≤16000)`, leheküljel märgitud voog. Järgmisel `N` real on arv `T (1≤T≤400)`, tehingu absoluutväärtus.

Väljastada `N`-märgiline string. `i`-s märk peab olema '+', kui on teada, et `i`-s tehing on tulu, '-', kui `i`-s tehing on kulu ning '?', kui ei ole võimalik seda informatsiooni teada. Kui ei ole võimalik märgitud voogu nende tehingutega saada, kirjutada väljundi ainsale reale üks märk '*'.

NÄIDE 1:

```
5 7
1
2
3
4
5
```

Vastus:

```
?+??+
```

NÄIDE 2:

```
4 15
3
5
7
11
```

Vastus:

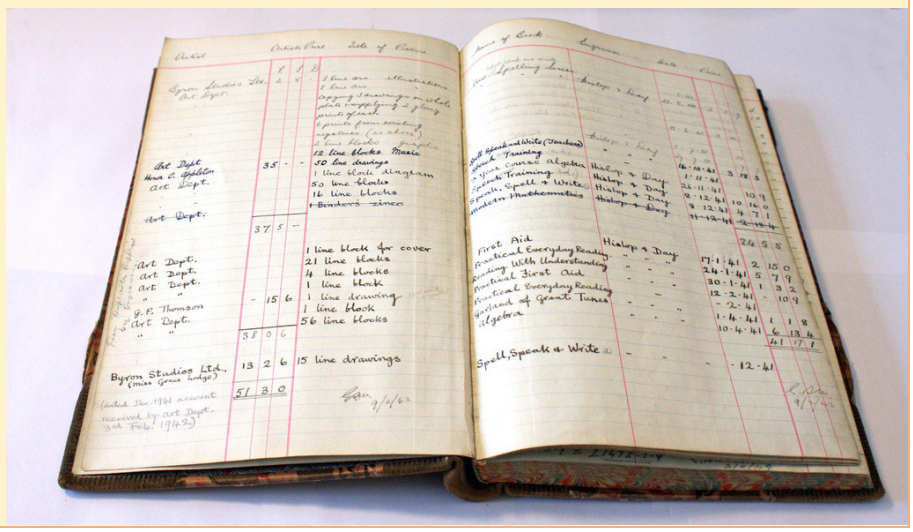
```
*
```

NÄIDE 3:

```
5 12
6
7
7
1
7
```

Vastus:

```
+??-?
```



8.9.6 Alkeemik

Alkeemialaboris on hulk erinevaid kemikaale. Alkeemik tahab need kahekaupa omavahel kokku segada, et alles jääks vaid üks kemikaal. Iga kahe kemikaali reageerimisel tekib mingisugune kolmas kemikaal ning eraldub teatud hulk soojust, mida saab tabelist järgi vaadata:

Kemikaalid	1		2		3	
	Uus kemikaal	Eraldunud soojus	Uus kemikaal	Eraldunud soojus	Uus kemikaal	Eraldunud soojus
1	1	0	3	-10	3	3000
2	3	-10	2	0	1	-500
3	3	3000	1	-500	3	0

Ülalolevas tabelis on kolm erinevat kemikaali: 1, 2 ja 3. Kui kokku segada kemikaalid 1 ja 3, tekib 3000 ühikut soojust ning uus segu on kemikaal 3. Vahel võib soojus neelduda. Näiteks tekib kemikaalide 2 ja 3 segamisel kemikaal 1 ning temperatuur langeb 500 soojusühiku võrra. Alkeemik tahab tekitada võimalikult kõrget temperatuuri. Seega, kui laboris on kemikaalid 1, 2, 2 ja 3, on tal võimalik nendest saada erinevaid temperatuure. Kui valida järjekord (2(1(23))), on temperatuuri muut kokku -510, kui see on aga ((12)(23)), on temperatuuri muut 2490.

Sisendi esimesel real on kemikaalide arv `n` (`1≤n≤6`). Järgmisel `n` real on `n` tühikutega eraldatud arvupaari, kus `i`-nda rea `j`-nda paari esimene arv on kemikaalide `i` ja `j` segamisel tekkiv uus kemikaal ning teine arv on sellel reaktsioonil toimuv temperatuuri muut. Järgmisel real on katseklaaside arv `k` (`2≤k≤10`). Sellele järgneb rida, kus on `k` arvu: igas katseklaasis olev kemikaali number.

Väljastada üks arv: maksimaalne võimalik temperatuuri tõus, mida on võimalik tekitada.

***NÄIDE 1*:**

```
3
1 0 3 -10 3 3000
3 -10 2 0 1 -500
3 3000 1 -500 3 0
4
1 2 2 3
```

Vastus:



***NÄIDE 2*:**

```
3
1 0 3 500 3 -250
3 500 2 0 1 10
3 -250 1 00 3 0
6
1 1 1 2 2 3
```

Vastus:

8.9.7 Puhkus

Lõbus Albert otsustas minna mõneks päevaks puhkusele. Puhkusel on tal võimalik teha erinevaid tegevusi, mille hind ja lõbuväärtus on teada. Albert teeb iga päev täpselt üht tegevust, kuid kui teha üht tegevust teist päeva järjest, saab Albert sellest 50% esialgsest lõbuväärtusest ning kolmandal järjestikusel päeval pole see tegevus enam lõbus. Seega, kui mingi tegevuse lõbuväärtus on v ning Albert teeb seda kolm päeva järjest, saab ta kokku lõbuväärtuse `1,5v`. Aita Albertil oma reisi planeerida, nii et tal oleks võimalikult lõbus ning ta ei ületaks oma eelarvet.
Sisendi esimesel real on kolm tühikutega eraldatud arvu: reisil olemise päevade arv `k (1≤k≤21)`, erinevate tegevuste arv `n(1≤n≤50)` ning Alberti eelarve `m (0≤m≤100)`.
Järgneval n real on kaks tühikuga eraldatud täisarvu: `c (1≤c≤50)`, tegevuse hind ning `v (1≤v≤10000)`, tegevuse lõbuväärtus.
Väljundisse kirjutada kaks tühikuga eraldatud arvu: maksimaalne lõbuväärtus, mida Albert saab endale lubada ning minimaalne hind, millega ta saab seda endale lubada. Kui ükski võimalik tegevuskava ei mahu eelarvesse sisse, väljastada arvupaar `0 0`.

***NÄIDE 1*:**

```
2 1 5
3 5
```

Vastus:

```
0 0
```



***NÄIDE 2*:**

```
3 5 20
2 5
18 6
1 1
3 3
2 3
```

Vastus:

```
13 6
```

8.9.8 Tango

Sel ajal, kui muusika algab, on peol võrdselt poisse ning tüdrukuid. Nad moodustavad omavahel poiss-tüdruk paarid, nii et keegi ei jää tantsust kõrvale. On teada, et mõned paarid on sellised, et paarilised vihkavad teineteist (vihkav paar) ning teised paarid on sellised, et paarilised armastavad üksteist (armastav paar). Ülejäänud paarides on paariliste omavaheline suhe neutraalne (neutraalne paar). Leida, mitu erinevat võimalust on paaride moodustamiseks, kus kas ei ole ühtegi vihkavat paari või on vähemalt üks armastav paar.

Sisendi esimesel real on arv `n(1≤n≤15)`, mis näitab poiste (ja seega ka tüdrukute) arvu. Järgmisel `n` real on `n` tühikutega eraldatud arvu (`0`, `1` või `2`). real `i` veerus `j` tähistab `0`, et `i`-s poiss ning `j`-s tüdruk moodustavad vihkava paari, `1`, et moodustub neutraalne paar ning `2` tähistab armastavat paari.

Väljundisse kirjutada üks arv: võimalike paarikomplektide koguarv.



***NÄIDE 1*:**

```
3
0 1 1
1 1 0
1 0 1
```

Vastus:

```
2
```

NÄIDE 2:

```
3
1 1 2
2 1 0
1 1 2
```

Vastus:

```
4
```

8.9.9 Valimised

FIPA (*Federation Internationale de Programmation Association*) korraldab hääletuse, et teada saada, mis riik korraldab tulevast programmeerimise MM-i ning EIO žürii on nõus tegema kõik, mis võimalik, et saada Eestit korraldajaks. Selleks teevad nad rahalisi annetusi erinevatele riikidele. Erinevate meetodite kaudu on teada, mitu eurot on vaja mingile riigile annetada, et nad hääle Eestile annaksid. Samas on teada, et hääle saamiseks ei pea igale riigile raha annetama, sest väiksemad ja vaesemad riigid on suuremate mõju all ning hääletavad nii, nagu nende mõjutaja. Seega, kui annetada mingile riigile, saab Eesti selle riigi hääle, kõigi sellele alluvate riikide hääled jne. Näiteks kui on teada, et Moldova on Rumeenia mõju all ning Rumeenia on Venemaa mõju all, on võimalik kõigi kolme riigi hääled saada siis, kui annetada vaid Venemaale. Ühelgi riigil pole mitut mõjutajat ning ükski mõjuahel ei moodusta tsüklit. EIO žürii palub abi, et teada saada, mis on vähim hulk raha, mida nad peavad kulutama, et Eesti võidaks. Kandidaadina ei ole Eestil hääleõigust.

Sisendi esimesel real on kaks arvu: n (`1≤n≤200`) ning `m (0≤m≤n)`, mis on hääletusel osalevate riikide arv ning Eesti valimiseks vajaminev häälte arv. Järgmisel `n` real on info iga hääleõigusliku riigi kohta. Esimesena on riigi nimi, siis on hääle muutmiseks vajaminev raha hulk, siis on number `0` või `1`, mis näitab, kas antud riigil on ülemus: 0 tähistab, et ülemust pole ja 1, et ülemus on. Riigi ülemuse nimi (kui see on olemas) on neljas string reas. Iga riigi ülemus on sisendis eelnevalt mainitud.

Väljundisse kirjutada üks arv: Mitu eurot on vaja kulutada, et Eesti saaks MM-i korraldajaks.

NÄIDE:

```
3 2
Läti 15 0
USA 20 0
Kanada 10 1 USA
```

Vastus:

```
20
```



8.9.10 Plaan

Kuulus programmeerija ning kolmekordne IOI kuldmedalist Martin Pettai töötab nüüd väikeses idufirmas Pisiraamat, kus tal tuleb tegeleda kahe projektiga: „Pehme“ ja „Nägu“. Martin peab kummagi projekti kallal töötama D (D ≤ 33) päeva. Ta ei saa ühel päeval mitme projekti kallal töötada ning ta peab varem alustatud projekti ka varem lõpetama. Samuti on Martin ainus inimene, kes projekte töös hoiab, mistõttu ta ei tohi rohkem kui G (G ≤ 33) päeva kummagi projekti kallal mitte töötada. Seega, kui D = 3 ja G = 2, on tal võimalik järgneva kuue päeva jaoks kümme erinevat töögraafikut koostada:

	1. päev	2. päev	3. päev	4. päev	5. päev	6. päev
1.	„Pehme“	„Pehme“	„Nägu“	„Pehme“	„Nägu“	„Nägu“
2.	„Nägu“	„Nägu“	„Pehme“	„Nägu“	„Pehme“	„Pehme“
3.	„Pehme“	„Pehme“	„Nägu“	„Nägu“	„Pehme“	„Nägu“
4.	„Nägu“	„Nägu“	„Pehme“	„Pehme“	„Nägu“	„Pehme“
5.	„Pehme“	„Nägu“	„Pehme“	„Pehme“	„Nägu“	„Nägu“
6.	„Nägu“	„Pehme“	„Nägu“	„Nägu“	„Pehme“	„Pehme“
7.	„Pehme“	„Nägu“	„Pehme“	„Nägu“	„Pehme“	„Nägu“
8.	„Nägu“	„Pehme“	„Nägu“	„Pehme“	„Nägu“	„Pehme“
9.	„Pehme“	„Nägu“	„Nägu“	„Pehme“	„Pehme“	„Nägu“
10.	„Nägu“	„Pehme“	„Pehme“	„Nägu“	„Nägu“	„Pehme“

Sisendi ainsal real on arvud D ja G.
Väljundisse kirjutada üks arv: mitu erinevat võimalust on Martinil töögraafiku koostamiseks.
NÄIDE:

```
3 2
```

Vastus:

```
10
```

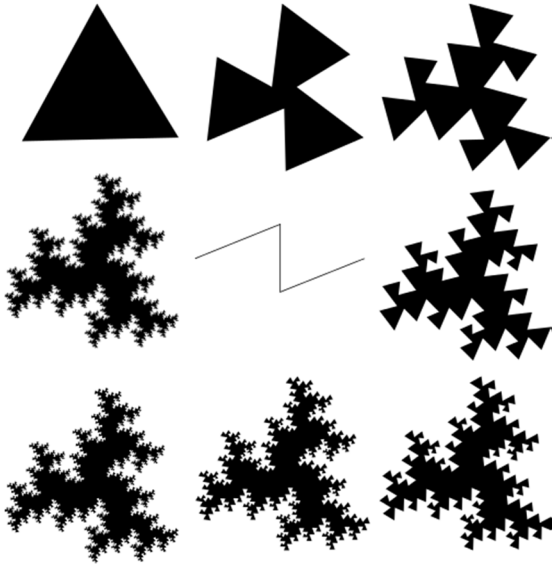
8.10 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk8 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
Kohver.cpp, Kohver.java, Kohver.py	0/1 seljakoti ülesnne
Kaupmees.cpp, Kaupmees.java, Kaupmees.py	Rändkaupmees Held-Karpi algoritmiga
Levenshtein.cpp, Levenshtein.java, Levenshtein.py	Levenshteini kaugus ehk toimetamiskaugus
Geenid1.cpp, Geenid1.java, Geenid1.py	Needleman-Wunchi algoritm
Geenid2.cpp, Geenid2.java, Geenid2.py	Smith-Watermani algoritm
DTW.cpp, DTW.java, DTW.py	Dünaamiline aja painutamine
Palk.cpp, Palk.java, Palk.py	Lõikudega dp, palgi saagimine
Maatriksid.cpp, Maatriksid.java, Maatriksid.py	Maatrikisite korrutamine (lõikudega dp)
Maed.cpp, Maed.java, Maed.py	Ülesanne “Mäed” (lõikudega dp)

9 ERIOTSTARBELISED PUUD

Raamatu esimeses osas uuriti mitmesuguseid üldotstarbelisi andmestruktuure nagu massiivid, pinud ja järjekorrad. Enamasti võimaldavad need andmestruktuurid tööd üksikute elementidega ja nende tavalisteks operatsioonideks ongi elemendi lisamine, muutmine, lugemine või eemaldamine.



Nii arvutiteaduses üldiselt kui ka võistlusprogrammeerimises konkreetsemalt on aga sageli vaja andmestruktuure, mis võimaldavad mitte ainult üksikväärtuste, vaid ka suuremate väärtuste hulkadega manipuleerimist. Näitena võib tuua operatsioonid nagu „mis on suurima väärtusega element 25. ja 137. elemendi vahel?“ või „Suurenda kõiki väärtusi 33. elemendist 99. elemendini“.

Sellist laadi ülesannete lahendamiseks sobivad hästi mitmesugused puukujulised andmestruktuurid. Nende efektiivset esitamist ja erinevate operatsioonide realiseerimise võimalusi uurimegi käesolevas peatükis.

9.1 LÕIKUDE PUU

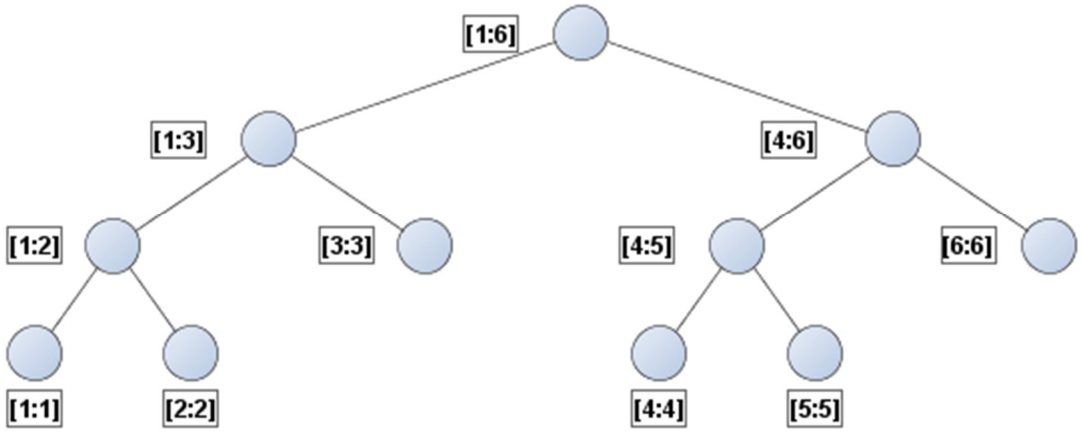
Vaatame ülesannet, kus on antud jada ja on vaja leida minimaalne element mingil suvalisel lõigul selles jadas. Sellist ülesannet tuntakse kui *Range Minimum Query (RMQ)*.

Kui sellist miinimumi leidmist on vaja teha vaid ühe korra, siis on loomulikult optimaalne käia antud lõik läbi ja leida minimaalne väärtus selles. Selle keerukus on lõigu pikkuse suhtes lineaarne: $O(n)$, kus n on jada pikkus. Kui aga päringuid on palju, siis muutub selline lähenemine liiga aeglaseks: m päringu korral on keerukuseks $O(m * n)$. Kui m on väike ja/või kui andmed muutuvad väga sagedasti, siis võib selline lihtne lähenemine olla optimaalne – kuna mingit uut struktuuri luua pole vaja, siis initsialiseerimisaeg puudub, samuti on andmete muutmine keerukusega $O(1)$.

Kui $m >> n$, siis on kasulikum täita tabel, kus kirjas kõikide lõikude miinimumid. Selle koostamise keerukuseks on $O(n^2)$, kuid igale päringule vastuse leidmise keerukus on $O(1)$. Aga ka see lähenemine võib osutuda liiga aeglaseks, eriti juhul, kui andmed päringute vahel muutuda võivad.

Võtame appi kahendpuu, mille igas tipus säilitatakse teatud lõigu miinimumi. Puu juurtipus on kogu jada $[1,\ n]$ miinimum, lehtedes jada üksikud väärtused, sest lõigu $[i,\ i]$ miinimum asub jadas kohal i.

Vahetippudes on vasaku alluva ja parema alluva lõikude ühend ehk lõik vasaku alluva vasakpoolsest indeksist parema alluva parempoolse indeksini:



Lõikude puu kuueelemendilise jada jaoks. Iga tipu juures on näidatud, millise lõigu kohta selles tipus informatsiooni säilitatakse.

Sellist puud nimetatakse **lõikude puuks** (i.k *segment tree*) ja selle mõtles välja Carnegie-Melloni ülikoolis töötanud ameerika arvutiteadlane Jon Louis Bentley aastal 1977.

9.1.1 Puu loomine jadast

Võtame näitejadaks järgmise jada:

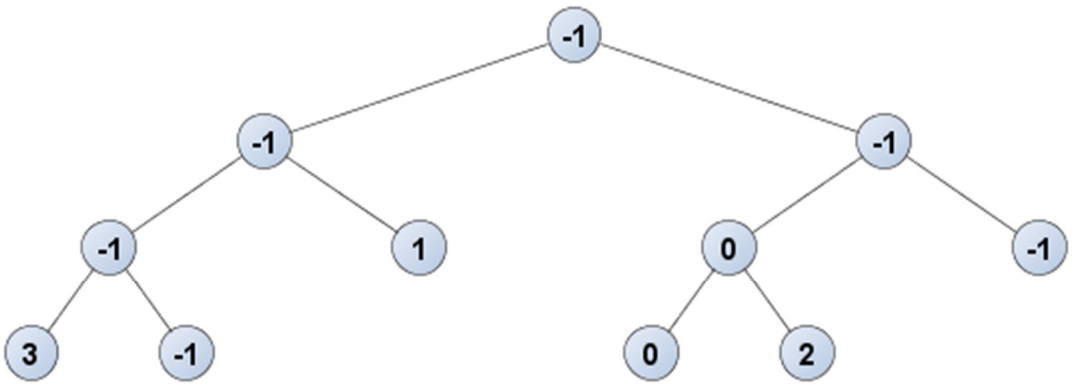
1	2	3	4	5	6
3	-1	1	0	2	-1

Esimeseks ülesandeks on algandmetest puu koostamine. Selleks on vaja kõigepealt otsustada, millist informatsiooni tippudes vaja hoida on. Antud ülesande puhul on mõttekas hoida iga lõigu kohta minimaalset väärtust sellel lõigul või uuritavas lõigus minimaalse väärtusega elemendi indeksit. Kasutame siin näidetes esimest.

Meeldetuletuseks teisest peatükist: kahendpuud, saab seda efektiivselt hoida massiivis. Kui alustada puud indeksilt 1, siis tipu k vasak alluv on massivis kohal $2 * k$ ja parem alluv kohal $2 * k + 1$. Igas tipus hoiame tema alluvate miinimumi indeksit.

Üheks sellise puu loomise võimaluseks on teha seda rekursiivselt:

```
void loo_puu(int tipp, int algus, int lopp)
{
    if (algus == lopp)
        LP[tipp] = A[algus]; //LP - lõikude puu
    else {
        //täida vasak ja parem alampuu
        loo_puu(2 * tipp, algus, (algus + lopp) / 2);
        loo_puu(2 * tipp + 1, (algus + lopp) / 2 + 1, lopp);
        //vasaku ja parema alampuu miinimum
        int k = 2 * tipp;
        LP[tipp] = min(LP[k], LP[k + 1]); //A - algandmed
    }
}
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	1	0	-1	3	-1	∞	∞	0	2	∞	∞

Eelmisel joonisel toodud jadale vastav miinimumide indekseid sisaldav lõikude puu abstraktselt ja massiivina.

Puu loomise keerukuseks on $O(n)$, kuid tasub meeles pidada, et see on ühekordne protseduur.

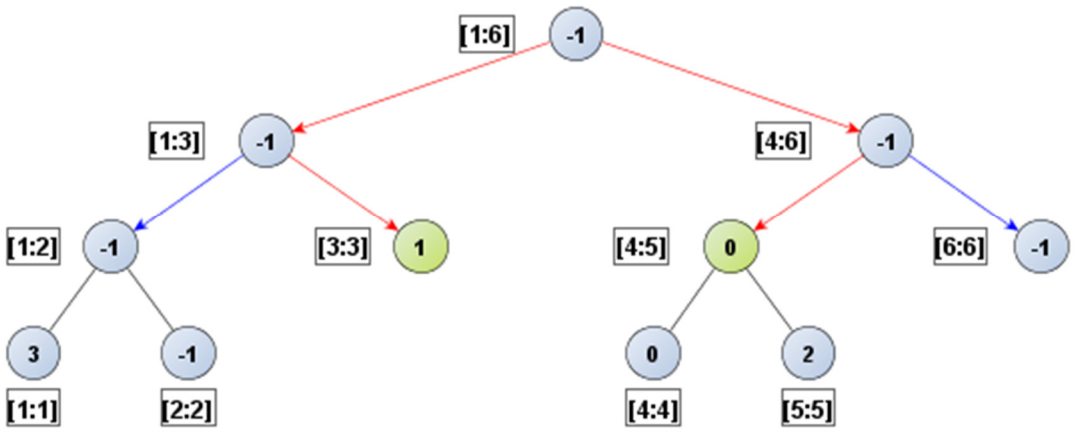
9.1.2 Lõigul väärtuse leidmine

Kui puu on valmis, siis järgmiseks protseduuriks on päringutele vastamine ehk väärtuse leidmine mingil suvalisel lõigul $[i,\ j]$.

Väärtuse leidmist alustame juurest, kus on olemas kogu jada miinimumi indeks. Kui aga lõik $[i,\ j]$ ei kata kogu jada, siis kogu jada miinimumiga ei ole midagi kasulikku peale hakata. Edasi otsimiseks on kolm võimalust:

1. Otspunkt j asub vasakus alampuus ($j \leq \frac{n}{2}$). Sellisel juhul asub kogu uuritav lõik vasakus alampuus ja võime sealt jätkata.
2. Otspunkt i asub paremas alampuus ($i > \frac{n}{2}$). Sellisel juhul asub kogu otsitav lõik paremas alampuus ja võime sealt jätkata.
3. Otspunkt i asub vasakus alampuus ja otspunkt j paremas alampuus. Sellisel juhul jagame lõigu kaheks $[i,\ \frac{n}{2}]$ ja $[\frac{n}{2}+1,\ j]$ ja otsime kummastki alampuust vastavat lõiku. Lõpuks on vaja leida miinimum vasakust ja paremast puust leitud miinimumidest.

Kui jõuame tippu, kus on säilitatud otsitava lõigu miinimum, siis tagastame kohe selle.



Lõigu [3:5] väärtuse leidmine. Rohelisega on märgitud need tipud, mille väärtus on oluline miinimumi leidmiseks uuritaval lõigul.

Otsime näiteks minimaalset väärtust lõigul [3, 5]. Alustame juurest. Esimene otspunkt 3 asub vasakus alampuus ja lõigu teine otspunkt 5 paremas alampuus. Vasak alampuu katab lõigu [1, 3]. Kuna ka see ei kuulu täielikult otsitavasse lõiku [3, 5], tuleb vaadata selle lõigu alampuid. Vasakul on lõik [1, 2], mis ei kattu üldse otsitava lõiguga. Selle väärtusega ei ole vaja arvestada ega siit edasi otsida. Lõigu [1, 3] parem alampuu [3:3] sisaldub tervenisti otsitavas lõigus [2, 5] ja seega tuleb selle väärtusega arvestada.

Juure parempoolses alampuus on lõik [4, 6], mis ei kuulu tervenisti otsitavasse lõiku, seega on taas vaja vaadata selle alampuid. Vasakpoolne alampuu [4, 5] aga kuulub tervenisti otsitavasse lõiku, saame kasutata siin salvestatud väärtust ilma selle alampuudele liikumata. Lõik [6, 6], mis on lõigu [4, 6] paremaks alampuuks, ei oma ühisosa lõiguga [3, 5], sellega pole vaja arvestada. Vastuse saab seega kokku tippudes [2, 2] ja [3, 4] salvestatud väärtustest. Miinimumi leidmise ülesandes on selleks vähim nendest.

Funktsioon, mis seda teeb:

```
int otsi(int tipp, int algus, int lopp, int i, int j)
{
    int vv, pv;

    if (i > lopp || j < algus)
        return maxInt;    // seda intervalli ei kasuta

    if (algus >= i && lopp <= j) // Kui kogu intervall kattub,
        return LP[tipp]; // tagasta väärtus selles sõlmes.

    vv = otsi(2*tipp, algus, (algus+lopp) / 2, i, j); //leia sobiv intervall vasakust
    pv = otsi(2*tipp + 1, (algus+lopp) / 2 + 1, lopp, i, j); // ja paremast alampuust

    // tagastame sobiva

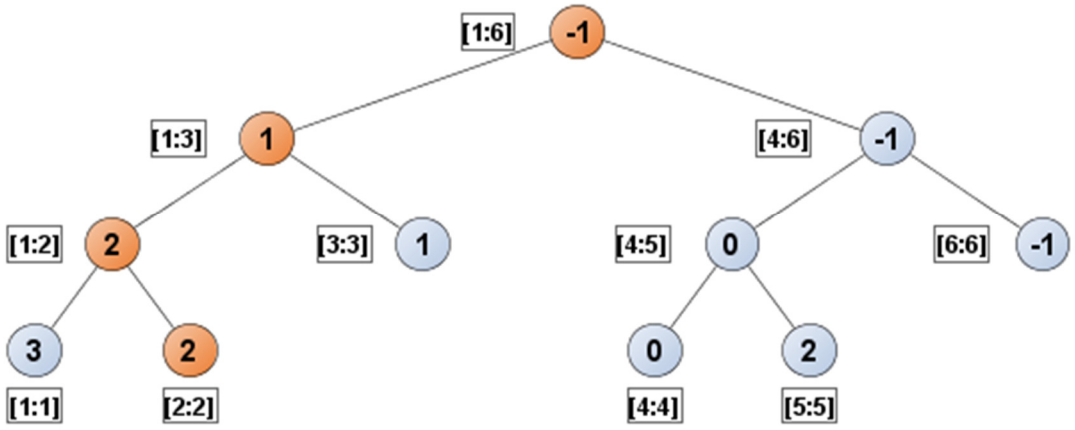
    if (vv <= pv)
        return vv;
    return pv;
}
```

Üksikoperatsiooni keerukuseks on $O(log\ n)$.

9.1.3 Väärtuse muutmine

Eriti oluline on lõikude puu siis, kui väärtused jadas võivad muutuda. Väärtuse muutmine on samuti keerukusega $O(log\ n)$, kuna peame uuendama ainult lehe ja selle eellaste väärtusi.

Näiteks soovime elemendile indeksil 2 anda väärtuseks -1 asemel 2. Kindlasti võib selline muudatus mõjutada juurt. Kuna element 2 kuulub juure vasakusse alampuusse, tuleb muuta ka seda ehk lõigu [1, 3] väärtust, parem alampuu [4, 6] jääb aga samaks. Samuti tuleb muuta lõikude [1, 2] ja [2, 2] väärtusi.



Väärtuse indeksil 2 muutmine. Punasega on märgitud need tipud, mille väärtused tuleb üle arvutada.

Klassikaline võte on taas tulla puus ülevalt alla ja rekursiivselt muuta nende lõikude väärust, mis sisaldavad muudetavat elementi:

```
void uuenda(int tipp, int algus, int lopp, int id)
{
    if (algus == lopp) { //Leht
        LP[tipp] = A[id]
        return;
    }
    else {
        int kk = (algus + lopp) / 2;
        if (algus <= id && id <= kk) {
            uuenda(2 * tipp, algus, kk, id); // vasak alampuu
        }
        else {
            uuenda(2 * tipp + 1, kk + 1, lopp, id); //parem alampuu
        }
        // vasak ja parem kokku
        LP[tipp] = min(LP[2*tipp], LP[2*tipp + 1]);
    }
}
```

9.1.4 Väärtuste muutmine tervel lõigul

Mõnikord on vaja korraga uuendada väärtusi terves mingis lõigus. Loomulikult on võimalik väärtustada kõik lehed ükshaaval. Veidi parem on kirjutada eraldi funktsioon:

```
void uuenda_loik(int tipp, int algus, int lopp, int algusID, int loppID)
{
    // ei ole piirides ja see haru ei vaja seega uuendamist
    if (algus>lopp || algus>loppID || lopp<algusID)
        return;

    // leht
    if (algus == lopp) {
         LP[tipp] = A[id]
         return;
    }

    // Kui ei ole leht, töötleme alluvad
    int kk = (algus + lopp) / 2;
    uuenda_loik(tipp * 2, algus, kk, algusID, loppID);
    uuenda_loik(tipp * 2 + 1, kk + 1, lopp, algusID, loppID);

    // alluvate tulemustest miinimum
    LP[tipp] = min(LP[2*tipp], LP[2*tipp + 1]);
}
```

9.2 LAISK VÄÄRTUSTAMINE LÕIKUDE PUUDES

Iga tipp lõikude puus hoiab endas infot mingi vahemiku kohta. Kui seda tippu on vaja uuendada, siis on vaja uuendada ka selle tipu alluvaid. Programmeerimisülesannetes tuleb sageli ette, et väärtusi muudetakse tervel lõigul sarnaselt, st kas antakse kogu lõigule üks ja sama väärtus või suurendatakse/vähendatakse väärtusi mingi väärtuse x võrra või korda. Sellistel juhtudel saab kergesti leida, mis on terve lõigu väärtuseks (näiteks miinimumiks või summaks), ilma et oleks vaja teada alamlõikude väärtusi.

Sellisel juhul on võimalus jätta nende alamlõikude väärtustamised lõikude puus tegemata ja teha need alles siis ja ainult siis, kui tekib vajadus kasutada konkreetseid (alam)lõike. Sellist meetodit nimetatakse **laisaks väärtustamiseks** (i.k *lazy propagation*).

Kuidagi on vaja ära tunda, millised tipud on uuendamata, et neid siis vajadusel õigete väärtustega varustada. Selleks kasutatakse eraldi massiivi, milles on iga tipu kohta vajalik informatsioon, kas ja kuidas selle tipu väärtust muuta tuleb. Algselt on iga elemendi väärtus selline, mis tähistab, et mingit muutust teha ei ole vaja.

Oletame, et on lõikude puu LP ja uuenduste jaoks massiiv U. Kui tipp t muutmist ei vaja, siis $U[t] = nil$. Muutmisfunktsiooniks on $F(U[t])$.

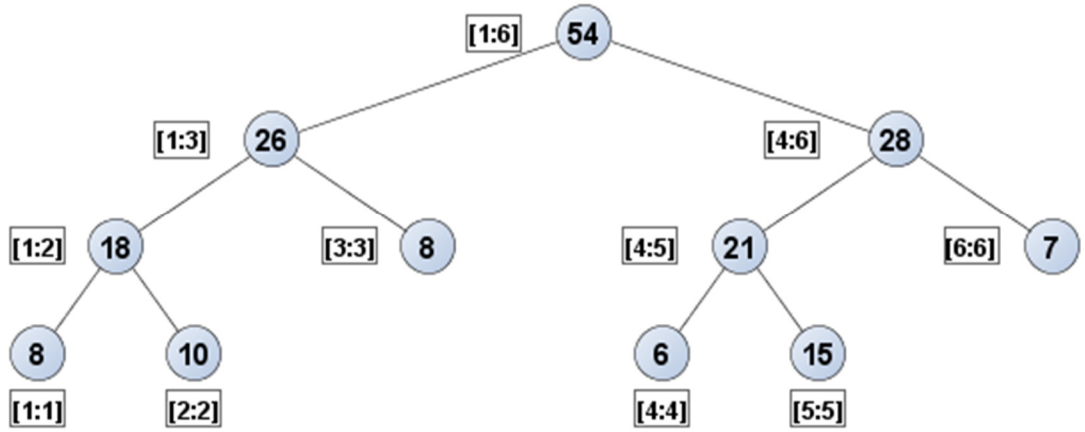
Lõikude puu elemendi muutmise algoritm on järgmine:

- Kui vaadeldaval tipp t ootab uuendamist, st $U[t] \neq nil$, muuda selle väärtus: $LP[t] = F(U[t])$.
- Kui vaadeldav vahemik sisaldub tervenisti päringus, uuenda vaadeldav tipp ja sea selle alluvatele ootel muutused.
- Kui vahemik kattub osaliselt, siis väärtusta vasak ja parem alampuu ja uuenda vastavalt saadud väärtustele.

Vaatame sellist lõikude puud, milles on igas tipus vastava lõigu elementide summa. Näiteks jadale

1	2	3	4	5	6
8	10	8	6	15	7

Vastab järgmine lõikude puu:

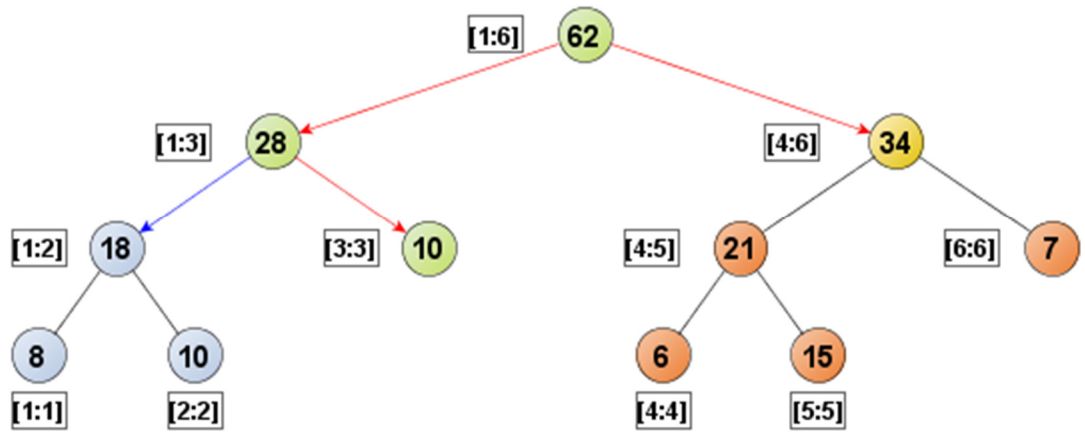


Lõikude puu, mille tippudes on vastava lõigu elementide summa. Juurtipus on kogu jada summa, lehtedes üksikud elemendid.

Edasi on meil vaja suurendada või vähendada iga elemendi väärtust teatud arvu võrra mingil etteantud lõigul. Oletame, et igale väärtusele lõigul [3, 6] liidetakse juurde 2.

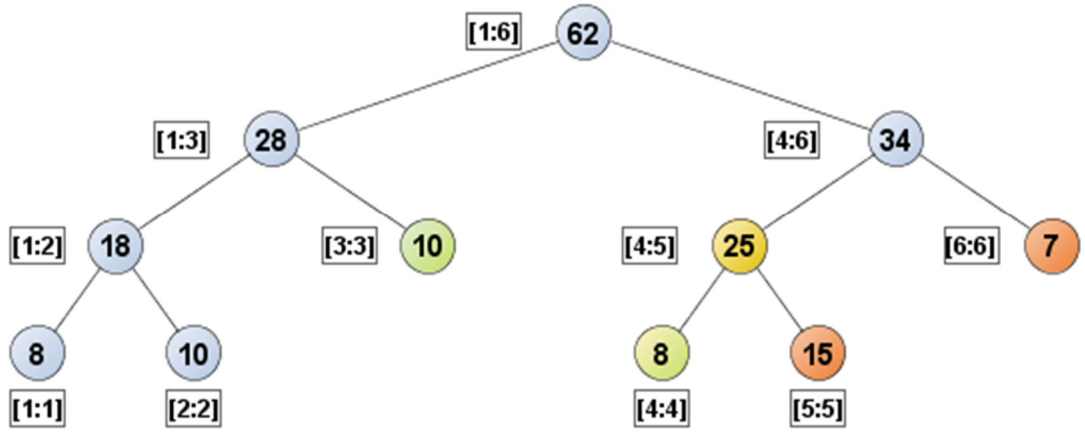
Alguses on meil uuenduste jada U elemendid määramata. Juurtipp ei asu tervenisti selles vahemikus, samuti kuna U[1] on määramata, ei ole sellel vaja teha uuendusi. Liigume juure alampuudele ning kontrollime, ega need uuendamist ei vaja. Praegu muidugi veel ei vaja. Vasak alampuu [1, 3] ei asu samuti tervenisti muudetavas vahemikus, liigume omakorda selle alampuudele. Vasak alampuu on tervenisti uuendatavast vahemikust väljas ega vaja uuendamist, sellega ei tee midagi. Parem alampuu aga on tervenisti muudetavas vahemikus, uuendame selle väärtuse ja samuti siis vahemiku [1, 3] väärtuse. Siiamaani toimub uuendamine tavapäraselt.

Vaatame aga paremat alampuud. Vahemik [4, 6] on tervenisti uuendatavas vahemikus, muudame selle väärtust ja seame alampuud muutuse ootele. Selleks U[6] = U[7] = 2. Seejärel uuendame tavapäraselt juurtipu.



Laisk väärtustamine. Igale tipule lõigus [3:6] lisatakse 2. Rohelisega on tipud, mis muudetakse tavapäraselt, kollasega on tipp, mille väärtus muudetakse (+6), aga mille alluvate väärtusi (punased) ei muudeta.

Oletame, et nüüd esitatakse päring lõigu [3, 4] kohta. Lõigu [3, 3] väärtuse saame tavapäraselt vasakust alampuust, paremas alampuus aga kui jõutakse lõiguni [4, 5], siis on kõigepealt vaja uuendada selle väärtust (+4) ning seada lõigud [4, 4] ja [5, 5] muutmist vajavateks (U[12]=U[13]=2). Seejärel liigume lõigule [4, 4], uuendame selle väärtuse (+2) ning saamegi kombineerida vastuse. Lõik [5, 5] jääb ikka muutmise ootele.



Laisk väärtustamine. Puu pärast päringut lõigu [3:4] kohta.

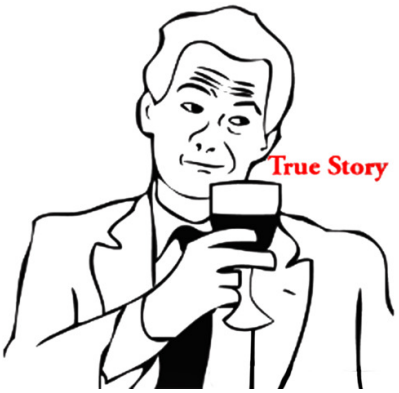
Oluline on meeles pidada, et info muudatuste kohta tuleb hoida värskena. Seega, kui muudatus on tehtud, tuleb see eemaldada muudatuste massiivist. Samuti tasub ka tähele panna, et sageli võivad muudatused kumuleeruda, enne kui neid tegelikele väärtustele rakendatakse. Näiteks kui nüüd soovime vähendada iga elemendi väärtust 3 võrra lõigul [4, 6], siis lõigu [6, 6] väärtust pole vaja vahepeal välja arvutada, lihtsalt väärtus massiivis U muutub 3 võrra väiksemaks. See tähendab, et kui enne oli U[7]=2, siis pärast on U[7] = 2-3=-1. Seega kui varem oli vaja selle tipu väärtust suurendada 2 võrra, siis pärast on seda hoopis vaja vähendada 1 võrra. Vahepealset väärtust (9) polnudki vaja välja arvutada.

Ühe mugavusena koodi kirjutamisel saab antud näite juures ära kasutada ka seda, et nulli võib vabalt liita ja lahutada, ilma et summa sellest muutuks, see tähendab, et võime alguses väärtustada muutuste massiivi nullidega ning igale tipule liita massiivis oleva väärtuse ilma eraldi kontrollimata, kas seda tegelikult vaja teha on.

Hetkel võib veel jääda segaseks, miks sellist keerukat süsteemi üldse vaja on. Päris mitmed võistlusülesanded on sellised, kus tehakse kõigepealt palju muudatusi lõikudel ning alles lõpuks esitatakse päring vastuse saamiseks. Nii muutuvad väärtused tervel lõigul mitmeid kordi, enne kui selle lõigu alampuid üldse vaja kasutada on. Vaatame selleks lähemalt kahte ülesannet toimunud võistlustelt.

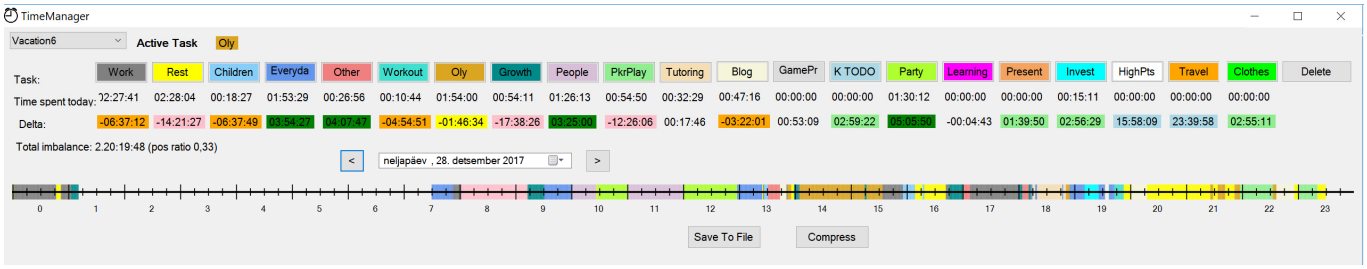
9.2.1 Paberiribade kleepimine

Selle ülesande panin ma 2014. aasta Eesti Informaatika Lahtiselt võistlusele. Originaalis on see aga hoopis päriselust, illustreerides asjaolu, et algoritmide tundmine aitab nii mõndagi „tavalist“ programmeerimisülesannet paremini lahendada.



Olen oma iseloomult alati olnud inimene, kes endale hulgaliselt erinevaid kohustusi, tegemisi, hobisid ja sebimisi teha võtab, mis viib kergesti ohuni, et mõni asi võtab liiga palju aega ning miski muu jääb seetõttu unarusse. Selle ennetamiseks kirjutasin endale programmi, kus saab erinevaid tegevusi kergesti ajajoonele märkida, näiteks hiirega algusest vastava ajalõigu algusest lõpuni vedades. Sageli on olukordi, kus mõne pikemaajalise tegevuse (nt tööpäeva) vahele tuleb märkida muid väikesi tegevusi (nt puhkepause), nii et kogutulemus on üpris fragmenteeritud. Andmete pealt saab teha igasugust huvitavat statistikat, nt uurida, millele erinevatel ajaperioodidel rohkem või vähem aega kulub.

Sisuliselt on tegu hulga ajaintervallidega, millega on vaja erinevaid lisamise, muutmise ja pärimise operatsioone sooritada. Kuni andmeid on vähe, pole väga suurt vahet, millist algoritmi kasutada, aga aastatega koguneb selliseid ajaintervalle palju tuhandeid ja ülesande lahendamiseks sobiski hästi lõikude puu.



Järgneb aga juba võistluse jaoks kohandatud sõnastus:

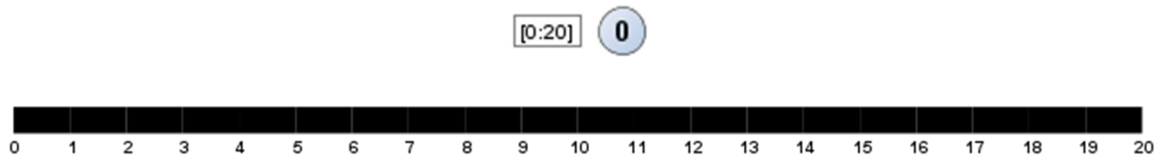
Jukul on pikk must pabeririba pikkusega L cm, millele ta kleebib teisi, värvilisi paberiribasid. Kleebitavad ribad mahuvad alati esialgse riba peale ära. Kõik ribad on sama laiusega. Iga järgmine riba varjab ära tema all olevad värvid. Leida, millised värvid ja milliste pikkustega jäävad lõppkokkuvõttes näha.
Sisendi esimesel real on antud musta riba pikkus L ($1 \le L \le 10^9$) ning kleebitavate ribade arv N ($0 \le N \le 500\ 000$). Järgmisel N real on igaühel kolm täisarvu: ühe riba värvikood K ($1 \le K \le 100$) ning selle alguse A ja lõpu B kaugus musta riba algusest ($0 \le A < B \le L$). Musta värvi kood on 0. Ainult alumine riba on must, teised on kõik värvilised.
Väljundisse väljastada lõpuks näha jäävate erinevat värvi lõikude värvid ja pikkused alates algse musta riba algusest kuni lõpuni. Järjestikused sama värvi ribad väljastada ühe lõiguna.
NÄIDE:

```
20 4
1 2 10
2 5 8
3 3 6
3 5 7
```

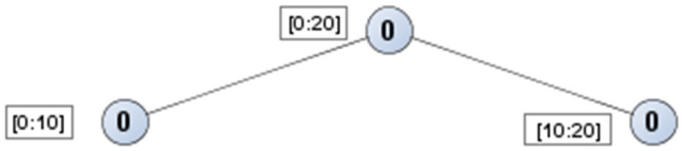
Vastus:

```
0 2
1 1
3 4
2 1
1 2
0 10
```

Iga pabeririba katab ära mingi teadaoleva lõigu ehk muudab värvi sellel lõigul. Siin toimub meil kõigepealt hulk kleepimisi, millest igaüks muudab värvi mingil lõigul. Meil on vaja kuidagi järge pidada nende muutuste üle, et saaksime väljastada lõppseisu. Selleks sobib suurepäraselt lõikude puu. Kuna väärtusi muudetakse tervete lõikude kaupa ja võib olla ka päris palju üle kleepimisi, siis on kaval kasutada laiska väärtustamist – konkreetsete ühiklõikude väärtused ei pruugigi kunagi kasutuses olla. Seetõttu pole vaja isegi luua tervet puud vaid võime luua mingi tipu t alampuud alles siis, kui neid on vaja. Ülevaatlikkuse mõttes ongi joonistel toodud ainult vajalikud tipud. Alustame ainult juurtipuga, mis on must (kood 0):

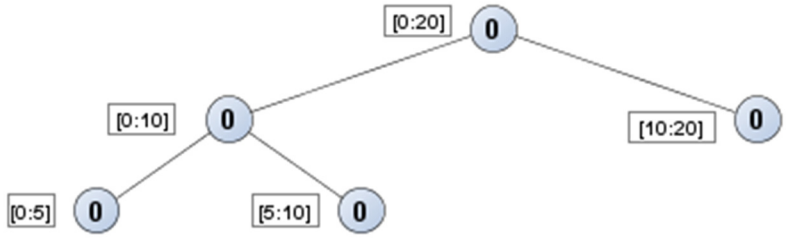


Nüüd on vaja lisada esimene riba värviga 1 (oletame, et see on punane). Kuna riba [2, 10] ei kata kogu musta riba, on vaja liikuda selle alampuudele. Alampuud pärivad esialgu sama värvi, mis vanemal:

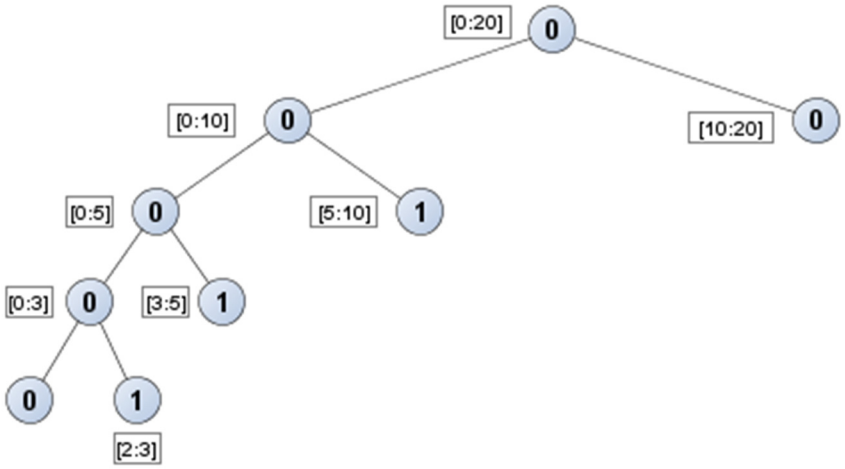


Vaatame kõigepealt, mis juhtub parempoolse alampuuga, mis vastab lõigule [10, 20]. Kuna see lõik jääb tervenisti kleebitavast ribast mõjutamata, siis jääb selle värviks must (0) ning sealt edasi liikuda

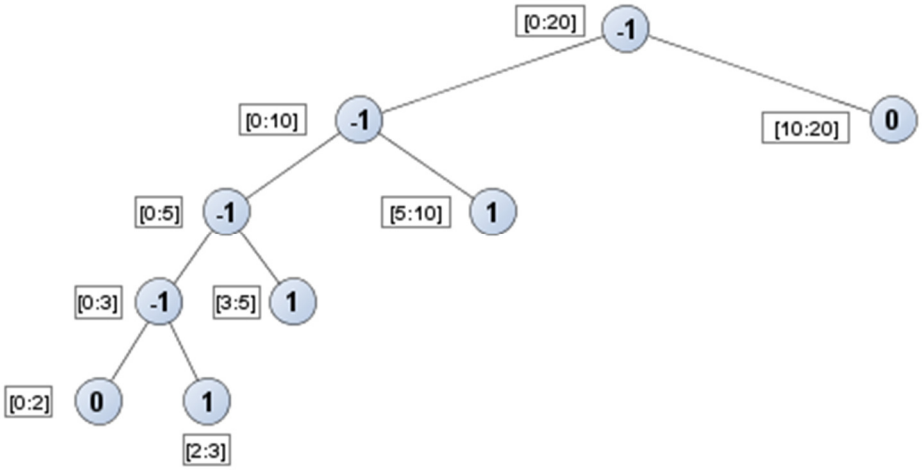
ei ole vaja. Vasakpoolsele alampuule vastav lõik [0, 10] on osaliselt kleebitava ribaga kaetud, seega tuleb liikuda selle alampuudele:



Vaatame taas enne lõigu [0, 10] parempoolset alampuud, mis vastab lõigule [5, 10]. See läheb tervenisti katmisele ja saamegi sellele määrata uue värvi ehk väärtuse 1. Siit edasi ei ole vaja liikuda. Vasakpoolne lõik [0, 5] aga on osaliselt kaetud ja seda tuleb taas poolitada ning pärast seda veel kord:

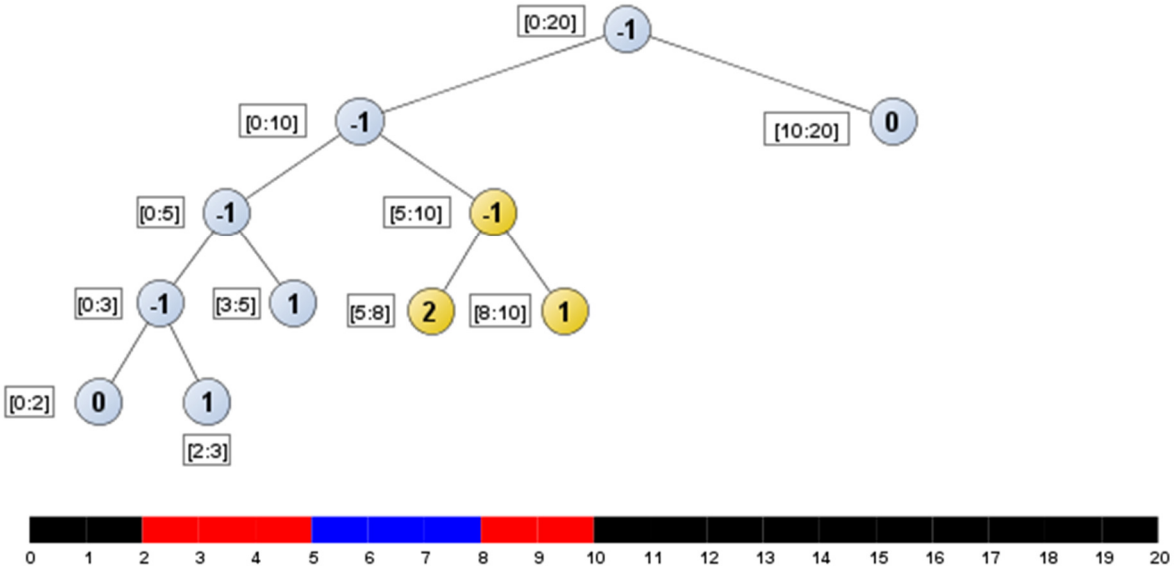


Nüüd on probleem selles, et mitmetel vanematel on värv erinev alluvate omast. Seega paneme selliste vanemate, millel on eri värvi alluvaid, värvi väärtuseks -1. Nii on puud läbi käies lihtne aru saada, mis toimub – kui lõigu värviks on -1, siis selle lõigu tegelikku värvi me ei tea ja peame uurima tema alamlõike. Kui aga lõigul on seatud mingi muu värv, siis on kogu see lõik justnimelt seda värvi ja edasi ei ole vaja seda lõiku jupiti uurida:

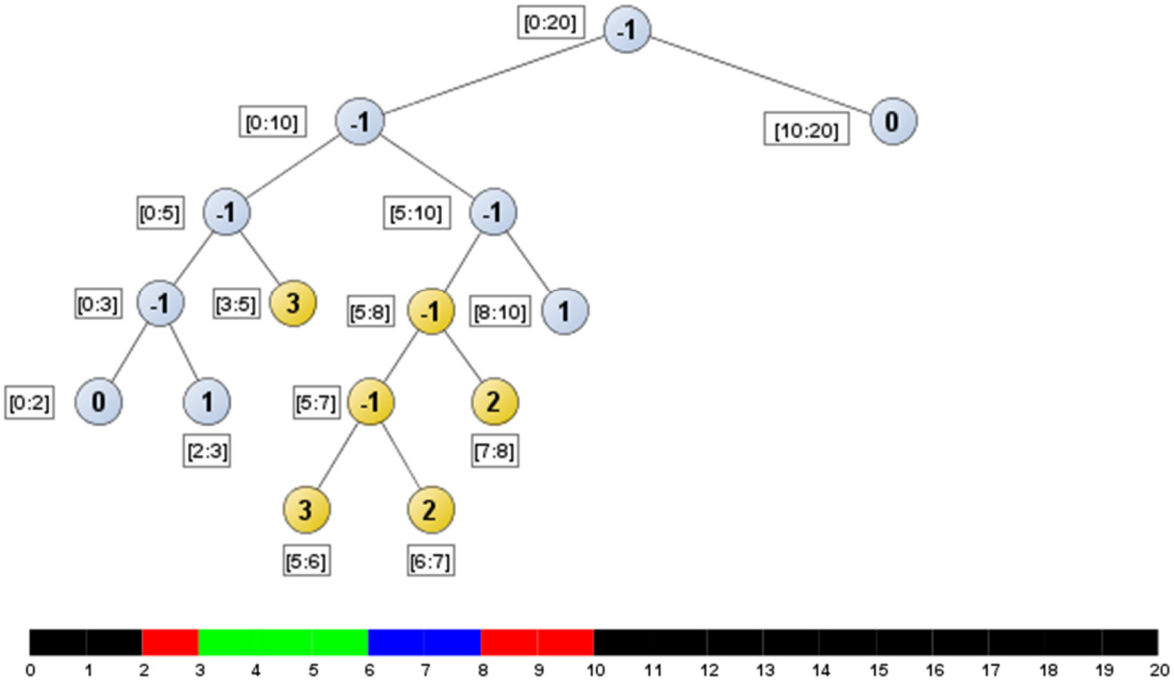




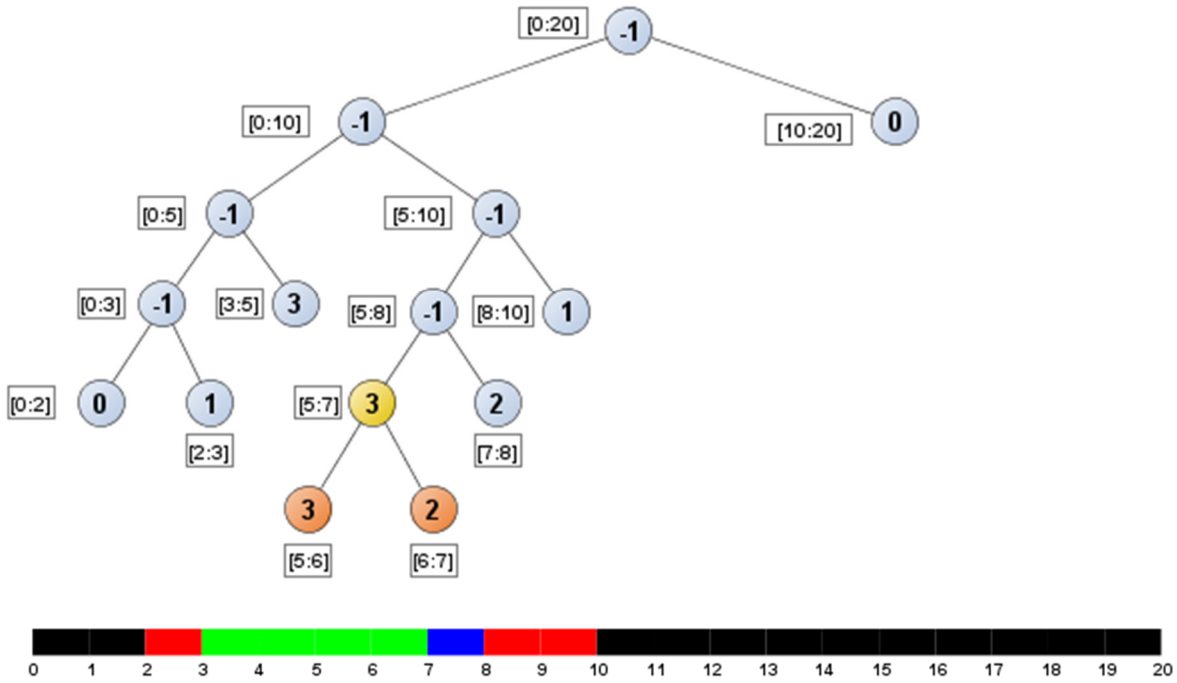
Teise riba (sinine) lisamise järgselt on pilt selline (kollased on muutunud väärtustega tipud):



Kolmas riba (roheline) annab seisu:



Pärast neljanda riba lisamist (samuti roheline):



Siin tekkis selline huvitavam olukord, kus sai üle värvida lõigu, mis oli enne mitmevärviline (värvi väärtus oli -1). See tipp on märgitud joonisel kollasega. Sellest jäävad ripakile tema alluvad (joonisel punased). Kas on vaja nendega midagi ette võtta? Esialgu puudub selleks vajadus, aga tuleb meeles pidada, et kui liigume ühevärviliselt tipult tema alluvatele, siis tuleb nende väärtust uuendada – nad pärivad vanemalt värvi väärtuse. Nii et kui see veel enne ilmne ei tundunud, siis nüüd on vast aru saada, et tegeleme laisa väärtustamisega.

Pane tähele, et kui puu kohe alguses valmis ehitada, tekib see probleem tegelikult juba esimese riba lisamisel, kuna aga joonisel neid alluvaid ei kujutatud, siis ei paistnud see veel välja.

9.2.2 Kood

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;
vector<int> _pointList;
vector<Ribbon> _result;
static TreeNode* MakeTree(int startIx, int endIx);
static void SetRibbon(TreeNode node, Ribbon ribbon);
static void WriteTree(TreeNode node);

class Ribbon
{
public:
    int Start;
    int End;
    int Color;
    void init(int start, int end, int color);
};
```

```
void Ribbon::init(int a, int l, int varv) {
    Start = a;
    End = l;
    Color = varv;
}

class TreeNode
{
public:
    int Color;

    int LeftPoint;
    int RightPoint;

    int LeftEnd;
    int RightEnd;

    bool CoversLeft;
    bool CoversRight;

    TreeNode* LeftNode;
    TreeNode* RightNode;
};

int main()
{
    set<int> pointSet;
    vector<Ribbon> ribbons;

    int l, n;
    cin >> l >> n;

    pointSet.insert(0);
    pointSet.insert(l);

    for (int i = 0; i < n; i++) {
        int varv, algus, lopp;
        cin >> varv >> algus >> lopp;
        Ribbon newRibbon;
        newRibbon.init(algus, lopp, varv);
        ribbons.push_back(newRibbon);
        pointSet.insert(newRibbon.Start);
        pointSet.insert(newRibbon.End);
    }

    _pointList.assign(pointSet.begin(), pointSet.end());
    sort(_pointList.begin(), _pointList.end());

    TreeNode* root = MakeTree(0, _pointList.size() - 1);

    for (int i = ribbons.size(); i > 0; --i) {
        Ribbon ribbon = ribbons[i];
        SetRibbon(root, ribbon);
    }

    WriteTree(root);

    for (int i = 0; i < _result.size(); i++) {
        Ribbon ribbon = _result[i];
        cout << ribbon.Color << " " << ribbon.End - ribbon.Start << endl;
    }
}
```

```
static void WriteTree(TreeNode node)
{
    if (node.LeftNode && !node.CoversLeft) {
        WriteTree(node.LeftNode);
    }

    Ribbon ribbon;
    ribbon.init(node.CoversLeft ? node.LeftEnd : node.LeftPoint,
        node.CoversRight ? node.RightEnd : node.RightPoint, node.Color);
    if (ribbon.Color == -1) {
        ribbon.Color = 0;
    }

    if (_result.size() > 0 && _result[_result.size() - 1].Color == ribbon.Color) {
        _result[_result.size() - 1].End = ribbon.End;
    }
    else {
        _result.push_back(ribbon);
    }

    if (node.RightNode && !node.CoversRight) {
        WriteTree(node.RightNode);
    }
}

static void SetRibbon(TreeNode node, Ribbon ribbon)
{
    // 0 == täiesti värvimata haru, -1 = osaliselt värvitud, värvimata juurtipuga haru
    if (node.Color <= 0) {
        if (node.Color==0&&ribbon.Start<=node.LeftEnd&&ribbon.End>=node.RightPoint) {
            node.CoversLeft = true;
        }
        if (node.Color==0&&ribbon.Start<=node.LeftPoint&&ribbon.End>=node.RightEnd) {
            node.CoversRight = true;
        }
        if (ribbon.Start <= node.LeftPoint && ribbon.End >= node.RightPoint) {
            node.Color = ribbon.Color;
        }
        else {
            node.Color = -1;
        }
    }

    if (ribbon.Start < node.LeftPoint && !node.CoversLeft && node.LeftNode) {
        SetRibbon(node.LeftNode, ribbon);
    }
    if (ribbon.End > node.RightPoint && !node.CoversRight && node.RightNode) {
        SetRibbon(node.RightNode, ribbon);
    }
}
```

```
static TreeNode* MakeTree(int startIx, int endIx)
{
    int medIx = (startIx + endIx) / 2;
    TreeNode node;
    node.LeftEnd = _pointList[startIx];
    node.RightEnd = _pointList[endIx];
    node.LeftPoint = _pointList[medIx];
    node.RightPoint = _pointList[medIx + 1];
    if (startIx < medIx) {
        node.LeftNode = MakeTree(startIx, medIx);
    }
    if (endIx > medIx + 1) {
        node.RightNode = MakeTree(medIx + 1, endIx);
    }
    return &node;
}
```

9.2.3 Ülesanne: Sein

See ülesanne on Taipeis toimunud 2014. aasta rahvusvaheliselt informaatikaolümpiaadilt.

Jian-Jia ehitab seina, ladudes üksteise otsa ühesuuruseid kive. Sein koosneb tulbast kividest, mis on nummerdatud vasakult paremale. Tulpadel kõrgused võivad olla erinevad. Tulba kõrguseks on kivide arv selles tulbas.
Jian-Jia ehitab seina järgneval viisil: alguses pole üheski tulbas ühtki kivi. Seejärel läbib Jian-Jia K kivide *lisamise* või *eemaldamise* etappi. Ehitamine lõpeb, kui kõik K etappi on läbitud. Igas etapis saab Jian-Jia lõigu järjestikuseid kivitulpasid ja kõrguse H ning sooritab järgmise protseduuri:

- *Lisamise* etapis lisab Jian-Jia etteantud lõigus kive nendesse tulpadesse, kus on vähem kui H kivi, nii et neis oleks täpselt H kivi. Tulpadega, kus juba on H või rohkem kivi, ei tee ta midagi.
- *Eemaldamise* etapis eemaldab Jian-Jia etteantud lõigus kive nendest tulpadest, kus on rohkem kui H kivi, nii et alles jääb H kivi. Tulpadega, kus juba on H või vähem kivi, ei tee ta midagi.

Sinu ülesanne on leida seina lõplik kuju.
Sisendi esimesel real on kaks täisarvu: n ($0 \le n \le 2\ 000\ 000$) – tulpade arv seinas ja k ($0 \le k \le 500\ 000$) – etappide arv. Järgmisel k real on igaühel neli täisarvu, mis kirjeldavad igat etappi: operatsiooni tüüp t (1-lisamine, 2-eemaldamine), vasak otspunkt v, parem otspunkt p, kusjuures $v \le p$, ja h ($0 \le h \le 100\ 000$) – seina kõrgus sellel etapil.
Väljundi ainsale reale kirjutada tühikutega eraldatult n täisarvu – iga tulba lõppkõrgus.
***NÄIDE*:**

```
10 6
1 1 8 4
2 4 9 1
2 3 6 5
1 0 5 3
1 2 2 5
2 6 7 0
```

Vastus:

```
3 4 5 4 3 3 0 0 1 0
```

Sellegi ülesande lahendamiseks saab kasutada lõikude puud. Siin aga tuleb väga hoolega silmas pidada, milline on ehitamise mehhanism: me ei pruugi muuta lõigul kõiki väärtusi, vaid ainult teatud osa nendest. Suurt probleemi see kohe ei tekita, sest lisamisel peame võtma maksimumi senisest kõrgusest ja uuest kõrgusest, eemaldamisel aga neist miinimumi.

Ühe võimalusena on hoida lisamise ja eemaldamise protseduuride jaoks eraldi massiive.

```
#include <algorithm>
#include <iostream>

using namespace std;

const int LEHED = (1 << 21);
const int PIKKUS = (1 << 22) + 5;

int D[PIKKUS], U[PIKKUS];
int N, K;

void yhenda(int n, int d, int u)
{
    D[n] = min(D[n], d);
    D[n] = max(D[n], u);
    U[n] = max(U[n], u);
    U[n] = min(U[n], d);
}

void tootle(int n, char t, int h)
{
    if (t == 2) {
        D[n] = min(D[n], h);
        U[n] = min(U[n], h);
    }
    if (t == 1) {
        D[n] = max(D[n], h);
        U[n] = max(U[n], h);
    }
}

void ehita(int v, int p, int t, int h, int n, int a, int b)
{
    if (a > p || b < v) return; // ei ole lõigus
    if (a >= v && b <= p) { // on tervenisti lõigus
        tootle(n, t, h);
        return;
    }

    yhenda(2 * n, D[n], U[n]);
    yhenda(2 * n + 1, D[n], U[n]);
    D[n] = PIKKUS;
    U[n] = 0;

    int mid = (a + b) / 2;
    ehita(v, p, t, h, 2 * n, a, mid);
    ehita(v, p, t, h, 2 * n + 1, mid + 1, b);
}

void lopeta()
{
    for(int i = 1; i <= LEHED - 1; i++) {
        yhenda(2 * i, D[i], U[i]);
        yhenda(2 * i + 1, D[i], U[i]);
    }
}
```

```
int main()
{
    cin >> N >> K;

    for (int i = 1; i <= K; i++) {
        int t, v, p, h;
        cin >> t >> v >> p >> h;
        //1-lisamine, 2-eemaldamine
        ehita(v + 1, p + 1, t, h, 1, 1, LEHED);
    }

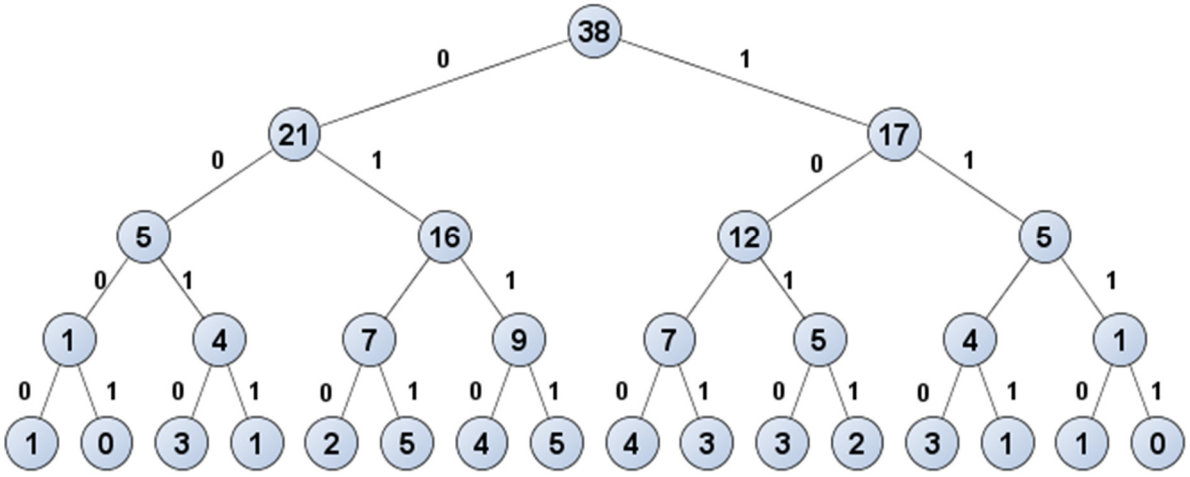
    lopeta();
    for (int i = LEHED; i < LEHED + N; i++)
        cout << min(D[i], U[i]) << " ";
    cin >> K;
}
```

9.3 FENWICKI PUU

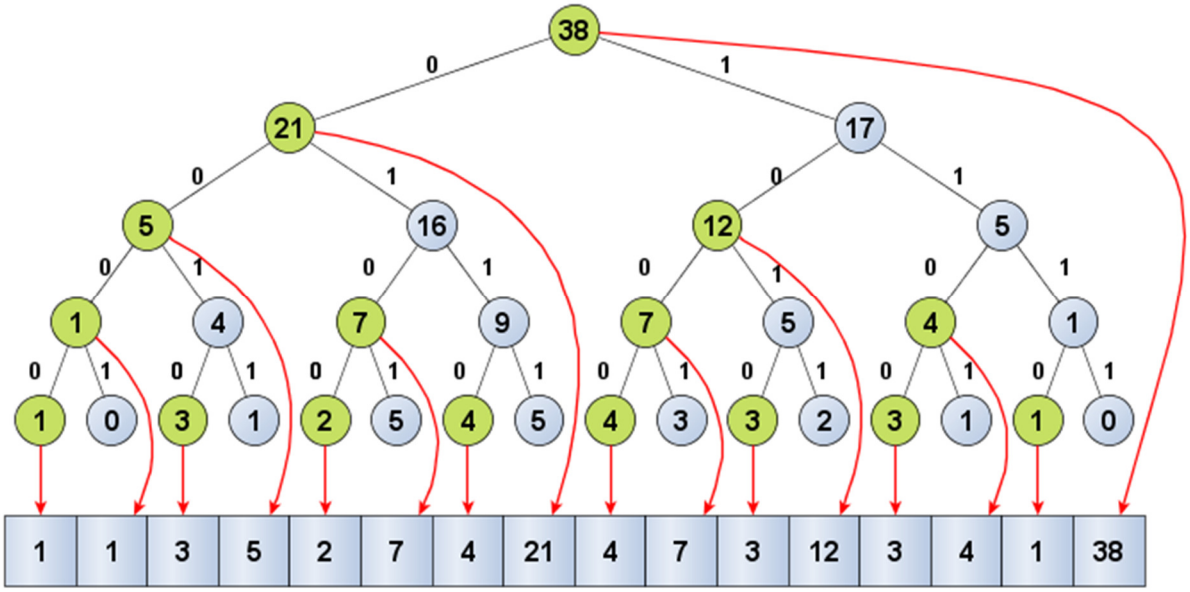
Fenwicki puu ehk **kahendindekseeritud puu** leiutas 1994. aastal Uus-Meremaa arvutiteadlane Peter Fenwick, kes uuris võimalusi andmepakkimisalgoritmide efektiivsuse suurendamiseks.

Vaatame ülesannet, kus tuleb leida kumulatiivne summa mingil lõigul. Jällegi – kui seda on vaja teha ainult üks kord, siis piisab sellest, kui liidamegi väärtused antud lõigul kokku. Kui päringuid on palju, kuid väärtused ei muutu, siis võib jadas hoida kumulatiivseid summasid: jada esimesel kohal on esimese elemendi väärtus, teisel kohal esimese ja teise summa, kolmandal esimese kolme summa jne. Kui on vaja leida summa lõigul $[a, b]$, siis selleks on lõikude $[0, b]$ ja $[0, a-1]$ vahe. Seega on sellise massiivi ülesseadmine keerukusega $O(n)$ ning sellest summa leidmine keerukusega $O(1)$.

Kolmandaks võimaluseks on kasutada lõikude puud, mille tippudes hoitakse alluvate summasid. Näiteks:



Lehtedeks on kõikide arvude 0-15 (0000-1111) sagedused. Igas tipus on tema alluvate summa. Paneme tähele, et summa leidmiseks on vaja vaadata ainult mõningaid tippe:



Kumulatiivsete summade leidmiseks on vaja hoida meeles ainult roheliste tippude väärtused.

9.3.1 Summa leidmine

Vaatame, kuidas leida esimese 13 elemendi summat. Iga summat saab esitada mingi mittekattuvate alamlõikude summana.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	3	1	2	5	4	5	4	3	3	2	3	1	1	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	3	5	2	7	4	21	4	7	3	12	3	4	1	38

Ülemisel joonisel on algses massiivis summeeritavad väärtused. Erinevate värvidega on tähistatud erinevad lõigud. Teises massiivis on Fenwicki puu ja rohelisega on tähistatud esimese 8 elemendi summa, kollasega 8.-12. elementide summa ja punasega 13. element. 13 esimese elemendi kumulatiivne summa on $21 + 12 + 3 = 36$.

Näiteks kui on vaja leida 13 esimese elemendi summat, siis kuna $13 = 2^0 + 2^2 + 2^3$, siis on meil vaja leida `FP[8] + FP[8+4] + FP[8+4+1]`, sest `FP[8]` katab lõigu 1…8, `FP[12]` lõigu 9..12 ja `FP[13]` üheelemendilise lõigu ehk elemendi kohal 13.

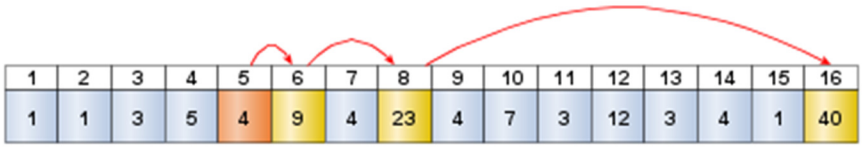
Järgnev funktsioon leiab esimese i elemendi summa:

```
int KumSumma(int i) {
    int sum = 0; // siin hoiame otsitavat summat
    while (i > 0) {
        sum += FP[i]; // Fenwicki puu
        i = i - (i & -i); // lahutame viimase biti
    }
    return sum;
}
```

Siin on kasutatud veidi kavalust bittidega. Negatiivsete arvude esitusest tulenevalt annab `i & -i` tulemuseks viimase seatud biti arvus. Näiteks 7 & -7 = 1, 6 & -6 = 2, 8 & -8 = 8 jne. `i - (i & -i)` nullib viimase seatud biti.

9.3.2 Elemendi lisamine ja muutmine

Iga kord, kui mingi elemendi väärtust muudetakse, tuleb muuta ka vastavad summad. Muutmise algoritm on sarnane kumulatiivse summa leidmise omale, aga liigume siin vastupidises suunas – muutmisel tuleb muuta järgnevaid elemente, mida muudatus mõjutab. Näiteks kui suurendada 5. elemendi väärtust 2 võrra, siis suurendame kõigepealt elementi FP[5], seejärel ka FP[6], kuna selles hoitakse FP[4] ja FP[5] summat. Edasi tuleb suurendada FP[8], kus hoitakse esimese kaheksa elemendi summat ja loomulikult ka FP[16], kus on kogu jada summa.



Kasutades taas viimase biti leidmist, saame kiirelt arvutada, millised väärtused tuleb muuta:

```
void lisaVaartus(int i, int v) {
    while (i <= N) {
        FP[i] += v;
        i = i + (i & -i);
    }
}
```

Lisamist võib ka vaadata kui muutmise operatsiooni, seega saab nii lihtsa vaevaga Fenwicki puu ehitada.

9.3.3 Üksiku elemendi leidmine

Kui mäluruumi jagub, siis võib muidugi iga elemendi tegeliku väärtuse hoida harilikus massiivis ja üksiku elemendi väärtuse leidmine on sel juhul keerukusega $O(1)$. Üksiku elemendi väärtust on võimalik aga taastada ka Fenwicki puust. Pole raske märgata, et $A[k] = FP[k] - FP[k-1]$, kus A on algandmetega massiiv.

```
int leiaVaartus(int i) {
    return KumSumma(i) - KumSumma(i - 1);
}
```

Vajadusel saab veidi kavalamalt ka, kasutades ära teadmist, et FP[i] väärtuseks on kas i. elemendi väärtus või i. elemendi ja mingite eelnevate elementide summa:

```
int leiaVaartus2(int i) {
    int sum = FP[i]; // summat hakkame vähendama (vajadusel)
    if (i > 0) { // erijuht kui i=0
        int z = i - (i & -i); // lahutame viimase biti maha
        i--;
        while (i != z) { // mingil hetkel z saab võrdseks indeksiga
            sum -= FP[i]; //vähendame summat
            i -= (i & -i); //lahutame viimase biti
        }
    }
    return sum;
}
```

9.3.4 Indeksi leidmine kumulatiivse summa järgi

Sageli on vaja ka leida etteantud kumulatiivsele summale vastavat indeksit. Üheks võimaluseks on proovida lihtsalt iga indeksi jaoks arvutada kumulatiivne summa ja võrrelda seda soovitud summaga. Kui väärtused võivad olla ka negatiivsed, siis see on tegelikult ainus moodus. Kui aga väärtused on kõik positiivsed, saab kasutada järgmist kahendotsimise algoritmi:

```
int otsiSumma(int summa) {
    int i = 0; // vastus
    int bitMask = esimeneBitt(N);
    while ((bitMask != 0) && (i < N)) {
        int mid = i + bitMask; // keskpunkt lõigul
        if (summa == FP[mid])
            return mid; // leidsime õige summa
        else if (summa > FP[mid]) { // kui mahub otsitavasse summasse
            i = mid;
            summa -= FP[mid]; // otsime, mis veel mahub
        }
        bitMask >>= 1; // poolitame otsimislõigu
    }
    if (summa != 0)
        return -1; // otsitavat summat ei olegi
    else
        return i;
}
```

9.3.5 Teatud lõigu summa leidmine

Kui on vaja leida summa teatud lõigul, näiteks elemendist 2 kuni elemendini 7, siis kõige lihtsamaks lahenduseks on

```
int loiguSumma(int i, int j) {
    return KumSumma(j) - KumSumma(i - 1);
}
```



9.4 2-MÕÕTMELINE FENWICKI PUU

Vaatame järgmiseks ülesannet, kus on vaja punkte tasandile lisada ja eemaldada ning leida parasjagu mingisse ristkülikukujulisse alasse jäävate punktide arv.

Selle lahendamiseks saab mugavalt kasutada kahemõõtmelist Fenwicki puud, mis sisuliselt on Fenwicki puu, mille tippudeks on omakorda Fenwicki puud.

9.4.1 Lisamine

Lisamise protseduur näeb välja järgmine:

```
void lisa(int x, int y, int val) {
    int y1;
    while (x <= max_x) {
        y1 = y;
        while (y1 <= max_y) {
            FP2D[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
```

Antud ülesandes võib siis kasutada uue punkti lisamisel väärtust 1 ja olemasoleva punkti eemaldamisel väärtust -1.

9.4.2 Loendamine

Analoogselt saab laiendada (kumulatiivse summa) leidmise algoritmi kahemõõtmelisele puule:

```
int loenda(int i, int j) {
    int j1;
    int sum = 0;
    while (i > 0) {
        j1 = j;
        while (j1 > 0) {
            sum += FP2D[i][j1];
            j1 = j1 - (j1 & -j1);
        }
        i = i - (i & -i);
    }
    return sum;
}
```

Kui on vaja loendada punkte suvalisest ristkülikust:

```
int loenda(int x1, int y1, int x2, int y2) {
    return loenda(x2, y2) - loenda(x1-1, y2) - loenda(y1-1, x2) + loenda(x1-1, y1-1);
}
```

Sama põhimõtte järgi saab koostada ka n-mõõtmelisi Fenwicki puid.

9.5 AJALOOGA ANDMESTRUKTUURID

Vahel on meil vaja andmestruktuuri, mis lisaks oma tavalisele funktsionaalsusele „mäletab“ ka neid andmeid, mis selles varem olid. Sel juhul ütleme, et andmestruktuur on **ajalooga** (i.k. *persistent data structure*). Andmestruktuur võib olla **täieliku** ajalooga (kui kõiki selle varasemaid versioone on võimalik muuta) või siis **osalise** ajalooga (kui muuta saab ainult viimast seisu).

Lihtsaim meetod andmestruktuuri ajaloo hoidmiseks on mõistagi sellest iga muudatuse puhul koopia tegemine, kuid see on jõudluse mõttes raiskav. Oluliselt parem on muutumatuks jäävaid andmeid versioonide vahel jagada.

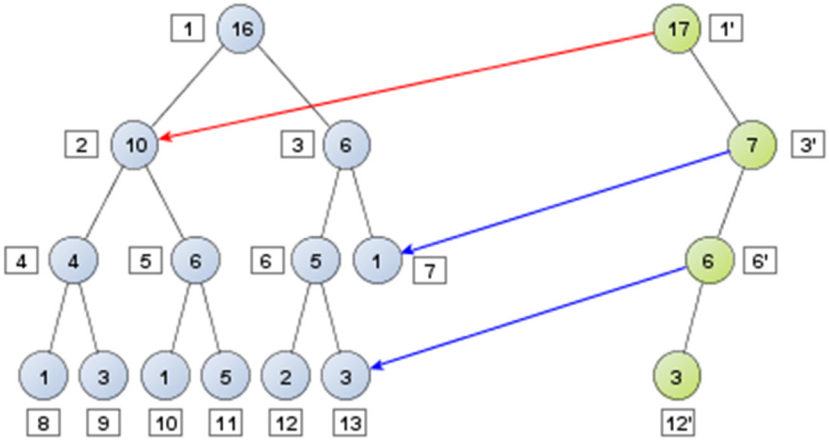
Üks võimalus selleks on nn **paksu elemendi** meetod – iga üksikelemendi asemel hoiame andmestruktuuris isekasvavat massiivi (nt vectorit või listi). Kui vastavat väärtust muudetakse, lisatakse massiivi järgmine element. Paksu elemendi meetod sobib näiteks ajalooga massiivi loomiseks.

Teine oluline võimalus on **teede kopeerimine**. Seda saab kasutada juhul, kui struktuur on ehitatud üksteisele viitavatest elementidest (nt ahel või puu). Selle meetodi korral:

1. Muuda ära viited muutunud elemendile.
2. Muuda ära viited eelmises sammus muutunud elementidele.
3. Korda sammu 2 kuni struktuuri alguseni.

9.5.1 Ajalooga lõikude puu

Ajalooga lõikude puu (i.k. *persistent segment tree*) tähendab seda, et säilitame kõik tehtud muudatused, kasutades teede kopeerimise meetodit. Kõige lihtsam on sellest mõelda kui versioneerimisest – iga muudatuse kohta tekib uus puu. Samas on terve uue puu loomine on nii mälu- kui ka ajamahukas. Teede kopeerimise meetod võimaldab aga teha muudatusi nii, et uuendatakse maksimaalselt $log(n)$ tippu. Põhimõtteliselt hoitakse meeles vaid see muudatuste puu – ülejäänud tipud on samad, mis eelmises verisoonis.



Ajalooga lõikude puu. Vasakul on eelmine versioon (hallid tipud) ja paremal uus versioon (rohelised tipud). Punane nool tähistab vasakut alluvat ja sinine paremat alluvat juhul, kui alluv on eelmises versioonipuus (ei ole muutunud).

Ülaltoodud joonisel on tippudes alluvate väärtuste summad. Elemendi asukohas 12 väärtust suurendatakse ühe võrra. Selle asemel, et see väärtus üle kirjutada, tekitatakse uus tipp 12'. Kuna see muudatus muudab ka tippude 6, 3 ja 1 väärtusi, tuleb nende jaoks luua samuti uued tipud 6', 3' ja 1'. Tipu 6' vasakuks alluvaks on tipp 12', kuid vasakuks alluvaks tipp 13. Samamoodi käitub ka tipp 3'. Juurtipu 1' vasakuks alluvaks saab tipp 2, paremaks 3'.

Alustades päringut tipust 1, saame seisu, mis oli enne muudatust, alustades aga tipust 1', läbime muudetud puud.

Peamiselt kerkib üles küsimus, kuidas pidada arvet versioonide üle. Piisab, kui teada versioonipuude juuri – neid võib meeles pidada eraldi massiivis. Pane tähele, et iga muudatus puus kandub edasi juureni, seega on igal versioonipuul oma versioon algse puu juurtipust.

Vaatame järgmise ülesande näitel, kuidas ajalooga lõikude puud rakendada. Ülesanne on pärit Balti Informaatikaolümpiaadilt (BOI) aastast 2015.

9.5.2 Ülesanne: Tekstiredaktor

Programmeerija Baitasar töötab uue revolutsioonilise tekstiredaktori kallal. Redaktoril on kaht tüüpi operatsioone: üks võimaldab redaktoris teksti *redigeerida*, teine võimaldab eelnevat operatsiooni *tagasi võtta* (undo). Üks redaktori innovatiivsetest võimalustest on *mitmetasemeline tagasivõtt*. See töötab järgmiselt: me ütleme, et teksti redigeerimine on *0. taseme* operatsioon. *i. taseme tagasivõtuoperatsioon* (kus `i = 1, 2,...`) võtab tagasi viimase ülimalt `i−1` taseme operatsiooni, mis pole veel tagasi võetud. Näiteks 1. taseme tagasivõtt saab tagasi võtta ainult redigeerimis-operatsioone, aga 2. taseme tagasivõtt saab tagasi võtta nii redigeerimisoperatsioone kui ka esimese taseme tagasivõtte (aga mitte kõrgemate tasemete tagasivõtte).
Formaalsemalt saab iga juba sooritatud operatsioon olla kahes olekus: `aktiivne` või `tagasivõetud`. Olgu `X` üks neist operatsioonidest. Kohe pärast operatsiooni `X` sooritamist on ta olekus `aktiivne`. Kui `X` on `i.` taseme tagasivõtuoperatsioon, siis leiame viimase ülimalt `i−1` taseme (tähistame selle `X1`) operatsiooni seisus `aktiivne` ning muudame operatsiooni `X1` seisu `tagasivõetud`. Kui `X1` on samuti tagasivõtuoperatsioon, siis tuleb meil ka see operatsioon, mille `X1` tagasi võttis (tähistame selle `X2`), muuta seisu `aktiivne`. Jätkame samal viisil: kui tagasivõtuoperatsiooni X_j olek, mis oli enne tagasi võtnud operatsiooni X_j+1, muutub, siis peame muutma ka operatsiooni X_j+1 olekut (mis võib omakorda muuta järgmiste operatsioonide olekut). Kogu muutmiste jada lõpeb, kui jõutakse redigeerimisoperatsioonini.
Lihtsuse mõttes on tekstiredaktori hetkeseis antud üksiku täisarvuga `s`, mida nimetatakse redaktori olekuks (alguses väärtusega `0`). Iga redigeerimisoperatsioon annab meile uue redaktori oleku. Redaktori olek sõltub viimasest redigeerimisoperatsioonist, mille olek on `aktiivne`. Aita Baitasaril kirjutada programm, mis jälgib tekstiredaktori olekut.
Sisendi esimesel real on positiivne täisarv `n`, mis tähistab Baitasari poolt sooritatavate operatsioonide arvu. Järgmised `n` rida sisaldavad operatsioone, üks igal real. Iga operatsioon on täisarv a_i ($-n \le a_i \le n,\ a_i \ne 0$). Kui $a_i > 0$, siis see tähistab redigeerimisoperatsiooni, mis viib redaktori olekusse a_i. Kui $a_i < 0$, siis see tähistab tagasivõtuoperatsiooni tasemega $-a_i$. Võib eeldada, et iga tagasivõtuoperatsiooni jaoks on alati mõni eelnev madalama tasemega operatsioon, mille seis on aktiivne, mida saab tagasi võtta.
Väljasta `n` rida. Rida `i` peaks sisaldama ühe täisarvu, millel on redaktori seis pärast esimese `i` operatsiooni sooritamist.

NÄIDE:

```
11
1
2
5
-1
-1
-3
4
-2
-1
-1
1
```

Vastus:

```
1
2
5
2
1
2
4
2
1
0
1
```

Vaatame järgmist näidet: järgnev tabel näitab mõnesid operatsioone, mida Baitasar saab sooritada ning redaktori olekut pärast nende sooritamist. Sümbol E_s tähistab redigeerimisopratsiooni, mis viib redaktori olekusse s ja sümbol U_i tähistab i. taseme tagasivõtuoperatsiooni:

	1	2	3	4	5	6	7	8	9	10	11
Operatsioon	E_1	E_2	E_5	U_1	U_1	U_3	E_4	U_2	U_1	U_1	E_1
Olek pärast	1	2	5	2	1	2	4	2	1	0	1

Alguses sooritas Baitasar kolm redigeerimisoperatsiooni. Redaktor läks olekust 0 olekusse 1, siis 2 ja lõpuks 5. Järgmiseks sooritas ta kaks 1. taseme tagasivõtuoperatsiooni, mis võtsid tagasi operatsioonid E_5 ja E_2 (muutes nad olekusse tagasivõetud). Sellega oli redaktori seis läinud tagasi olekusse 1. Järgnev 3. taseme tagasivõtuoperatsioon võttis tagasi viimase U_1 operatsiooni (muutes selle olekusse tagasivõetud). Tagajärjena taastatakse operatsioon E_2 (see läheb olekusse aktiivne) ning redaktor läheb uuesti seisu 2. Operatsioon U_2 võtab tagasi operatsiooni E_4, operatsioon U_1 võtab jälle tagasi taastatud operatsiooni E_2, siis võtab viimane U_1 operatsioon tagasi operatsiooni E_1. Viimane operatsioon on E_1.

9.5.3 Kood

```
#include <cstdio>

int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
int max(int x, int y, int z) {
    return max(x, max(y, z));
}

struct PersTree {
    PersTree *l, *r;
    int key;
    int value;
    int max_value;

    int find(int) const;
    PersTree *add(int, int) const;

    PersTree(PersTree *_l, PersTree *_r, int _key, int _value, int _max_value) :
l(_l), r(_r), key(_key), value(_value), max_value(_max_value) {}

};

int safe_max_value(PersTree *x) {
    if (x == NULL)
        return 0;
    return x->max_value;
}

int PersTree::find(int mxkey) const {
    if (mxkey < key)
        return l->find(mxkey);
    else if (mxkey == key)
        return max(safe_max_value(l), value);
    else // if (mxkey > key)
        return max(safe_max_value(l), value, r->find(mxkey));
}

PersTree *PersTree::add(int _key, int nvalue) const {
    PersTree *_l = l;
    PersTree *_r = r;
    int _value = value;

    if (_key < key)
        _l = l->add(_key, nvalue);
    else if (_key == key)
        _value = nvalue;
    else // if (_key > key)
        _r = r->add(_key, nvalue);

    return new PersTree(_l, _r, key, _value, max(safe_max_value(_l), _value,
safe_max_value(_r)));
}
```

```
PersTree *create_pers_tree(int left, int right) {
    if (left > right)
        return NULL;
    int s = (left + right) / 2;
    return new PersTree(create_pers_tree(left, s - 1), create_pers_tree(s + 1, right),
s, 0, 0);
}

int n;
const int M = 3e5 + 5;
PersTree *akt[M];
int wyn[M];
int main() {
    scanf("%d", &n);
    akt[0] = create_pers_tree(0, n);
    wyn[0] = 0;

    for (int i = 1; i <= n; ++i) {
        int op;
        scanf("%d", &op);
        if (op > 0) {
            wyn[i] = op;
            akt[i] = akt[i - 1]->add(0, i);
        }
        else {
            int cof = akt[i - 1]->find(-op - 1) - 1;
            wyn[i] = wyn[cof];
            akt[i] = akt[cof]->add(-op, i);
        }
        printf("%d\n", wyn[i]);
    }
}
```

9.6 HÕRE TABEL

Kui andmed ei muutu, siis on võimalik lõikude puu asemel kasutada „hõredat tabelit“ (i.k. *sparse table*). Tabeli veergudeks on lõigu alguspunktid ning tabeli veergudeks lõikude pikkuseid, nii et reas indeksiga j on lõikudele pikkusega 2^j vastavad väärtused. Esimeses reas, mille indeksiks on 0, on seega üheelemendise pikkusega lõigud ehk töödeldava jada algväärtused.

Siin on tabel, mis vastab kumulatiivse summa ülesandele:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	3	1	2	5	4	5	4	3	3	2	3	1	1	0
1	1	3	4	3	7	9	9	9	7	6	5	5	4	2	1	
2	5	6	11	12	16	18	16	15	12	11	9	7	5			
3	21	24	27	27	28	29	25	22	17							
4	38															

9.6.1 Tabeli koostamine

Tabeli koostamise üldreegel:

$$T[i,j] = F_{k=i\ldots i+2^j-1}\{Jada[k]\}, kus\ 1 \le i \le n, 1 \le j \le \lfloor \log n \rfloor$$

Kiiremaks koostamiseks saab kasutada dünaamilist programmeerimist. Näiteks minimaalse elemendi leidmise korral on rekursioonivalem järgmine:

$$T[i,j] = F(T[i,j-1], T\left[i+2^{j-1}, j-1\right]$$

Tabeli ehitamiseks kuluv aeg on $O(n\ log\ n)$.

9.6.2 Tabelist lugemine

Tabelist lugemine sõltub mõnevõrra ülesande spetsiifikast.

Näiteks minimaalse väärtuse leidmiseks lõigul [i, j] on vaja leida kaks lõiku, mis tervenisti katavad otsitava lõigu [i, j]. Pikim tabeli lõik, mis mahub lõiku [i, j] on pikkusega $2^{\lfloor \log(j-i+1) \rfloor}$.

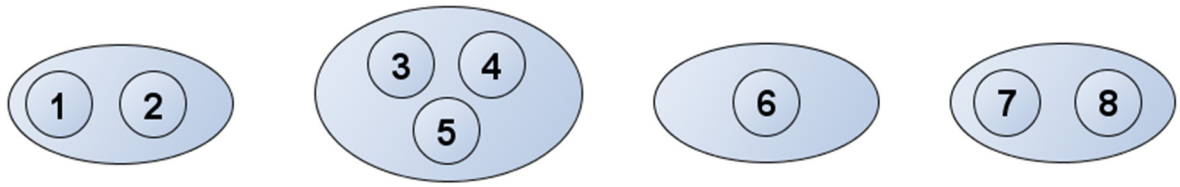
$$RMQ[i,j] = \begin{cases} T[i,k], & kui\ Jada[T[i,k]] \leq Jada[T[j-2^k+1,k]] \\ T[T[j-2^k+1,k] \end{cases}$$

Samas näiteks kumulatiivse summa ülesandes on oluline, et arvestatavad lõigud ei kattuks, vaid järgneksid vahetult üksteisele.

9.7 ÜHISOSATA HULGAD

Hulk (i.k *set*) on andmestruktuur, mille kõige olulisemaks tunnuseks on, et iga element saab esineda selles maksimaalselt ühe korra.

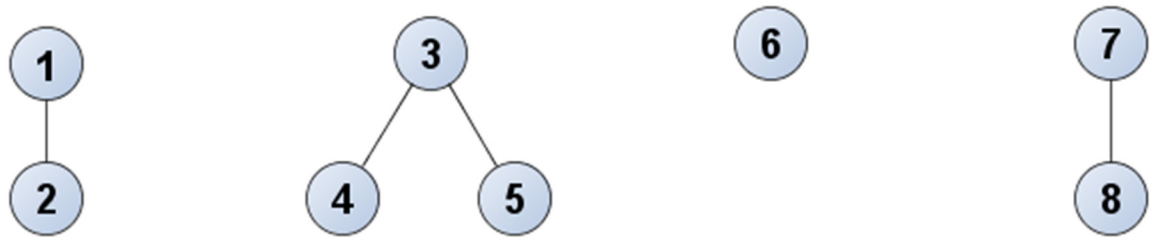
Mõnikord on aga vaja tervet hulkade komplekti nii, et iga element esineb mitte rohkem kui ühes hulgas. Sellist struktuuri võib nimetada ühisosata hulkadeks (i. k. *disjoint sets*).



Peamised operatsioonid, mida on vaja sooritada, on hulga loomine, kahe hulga ühendamine ning leidmine, millisesse hulka element kuulub.

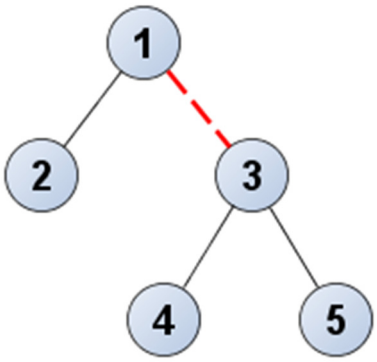
Abstraktselt on tegemist küllaltki lihtsa struktuuriga, küsimus on, et kuidas hoida hulki ja nende elemente nii, et mainitud operatsioonid oleksid võimalikult efektiivsed.

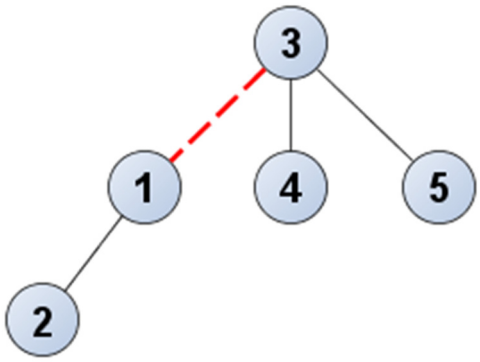
Tavaliselt esitatakse iga hulk elementide puuna ning kogu ühisosata hulkade struktuur on seega mets.



Peamiselt on meil vaja teada, kes on kelle vanem, seega võib kogu struktuuri hoida loendis, kus kohal i on i-nda elemendi vanem. Juurelemendi vanemaks loetakse enamasti element ise.

Kahe puu ühendamiseks piisab, kui muuta ühe puu vanemat:





Elemendi kuuluvuse leidmine on samuti lihtne:

```
int leia(int x) {
      if (hulgad[x] == x) return x;
      return leia(hulgad[x]);
}
```

Probleem on see, et halvimal juhul on selle keerukuseks $O(n)$ – seda küll juhul, kui kogu mets koosneb ühest puust sügavusega n ja soovime leida elemendi n hulka. Seetõttu on kasulik hoida puud võimalikult madalad.

Elemendi lisamisel on seda kerge teha, kuna meil ei ole piiratud, mitu alluvat tipul olla võib, ja seega saab uue elemendi lisada alati otse juure alla. Puu kõrgus kasvab seega just puude ühendamise operatsiooni käigus. Siingi on esmaseks lihtsaks lahenduseks, et madalam puu läheb alati kõrgema alla, mitte vastupidi. Nii jääb ka ühendatud puu kõrgus muutumatuks. Muutus kõrguses toimub vaid siis, kui ühendada tuleb kaks sama kõrgusega puud.

Puu kõrgust iga kord leida on tülikas, selle asemel peetakse seda meeles. Selleks antakse igale tipule veel üks väärtus – **aste** (i.k *rank*). Kahe puu ühendamisel lisame väiksema astmega puu suurema astmega puu alla ja nii jääb uue puu aste muutmatuks. Kui aga ühendatavad puud on võrdse kõrgusega, siis ühendame puud ja suurendame juure astet ühe võrra. Nii ei saa puu aste kasvada kunagi suuremaks kui log n.

Puus, mille juure aste on n, on vähemalt 2^n tippu. Näitame seda induktsiooniga. Baasjuhuks on ühetipuline puu, mille asteks on 0: $2^0 = 1$. Puu aste muutub ühe võrra, kui liidame kaks sama astmega puud. Olgu selleks astmeks k ja hüpoteesi järgi on siis kummaski puus vähemalt 2^k tippu. Tippude arv uues puus on siis vähemalt $2^k + 2^k = 2^{k+1}$.

Teiseks võtteks, et veelgi kiirendada elementide leidmiseks on nn **tee pakkimine** (i.k. *path compression*). Nagu juba lisamisel sai mainitud, siis parim juht on, kui puu on võimalikult madal ehk kõik hulga liikmed on puus vahetult juurtipu all. Kui me hakkaks seda ühendamisel tegema, siis kaotaksime oluliselt ühendamise operatsiooni efektiivsuses. Samas hulga leidmise operatsiooni ajal saame lihtsalt kõikide teele jäävate tippude vanemat muuta:

```
int leia2(int x) {
      if (hulgad[x] != hulgad[hulgad[x]]) hulgad[x] = leia2(hulgad[x]);
      return hulgad[x];
}
```

Pane tähele, et me ei vaevu muutma astmeid, mistõttu need ei pruugi enam vastata puu tegelikele kõrgustele. Sellele ei ole lihtsalt praktiline aega kulutada. Kuna tee pakkimine ei muuda tippude arvu

puus, siis jääb kehtima ka varem tõestatud väide, et puus astmega n on vähemalt 2^n tippu. Puu tegelik kõrgus ei saa kunagi olla suurem kui puu aste.

Koos tee pakkimisega m ühendamise või leidmise operatsiooni on keerukusega $O((m+n)log^*n)$, kus log^*n on pöördfunktsioon funktsioonist $2^{2^{2^{\cdots 2}}}$, kus siis astendatakse n korda. Selle funktsiooni väärtus kasvab väga kiiresti: 2, 4, 16, 65 536, $2*10^{19728}$. Vastava pöördfunktsiooni väärtus on jällegi väga väike ja ei lähe praktilises kasutuses üle viie, mis tähendab, et lisamise ja ühendamise operatsioonide keerukus on m operatsiooni korral $O(m+n)$.

```
bool yhenda(int x, int y) {
     int xx = leia2(x), yy = leia2(y);
     if (xx == yy) return false;
     if (rank[xx] > rank[yy]) { int t = xx; xx = yy; yy = t; }
     if (rank[xx] == rank[yy]) rank[yy]++;
     hulgad[xx] = yy;
     return true;
}
```

9.7.1 Kasutamine

Ühisosata hulki kasutatakse peamiselt graafiülesannete juures, nimelt minimaalse toese leidmiseks. Sellest tuleb täpsemalt juttu järgmises peatükis.

9.8 KONTROLLÜLESANDED

9.8.1 Laoseisud

Reas on n ($1 \le n \le 200000$) laohoonet. i-ndas laohoones on r ($0 \le r \le 1000$) võistlusprogrammeerimise õpikut. Vahetevahel lisatakse õpikuid laohoonesse juurde või võetakse neid sealt ära. Samuti tahetakse aeg-ajalt teada, mitu õpikut on mingites järjestikku asuvates laohoonetes kokku.

Sisendi esimesel real on täisarv n. Järgmisel n real on laohoonetes olevate raamatute arv. Sellele järgneb ühel real arv q ($1 \le q \le 200000$) – reas tehtavate operatsioonide arv. Järgmised q rida on kas kujul „L x r“ ($1 \le x \le n$; $1 \le r \le 1000$) – tehakse operatsioon, mille tagajärjel on laohoones x r raamatut, või „U x y“ ($1 \le x \le y \le n$) – uuritakse, mitu õpikut on laohoonetes vahemikus x…y.

Väljundisse kirjutada mingi hulk ridu. Igale reale on vaja väljastada üks täisarv: vastava teist tüüpi operatsiooni vastus.

NÄIDE 1:

```
3
100
100
100
6
U 1 1
U 1 3
L 2 200
U 1 2
L 3 0
U 2 3
```

Vastus:

```
100
300
300
200
```

NÄIDE 2:

```
10
1
2
3
4
5
6
7
8
9
10
1
U 1 10
```

Vastus:

```
55
```



9.8.2 Korrutamismäng

Sa mängid sõpradega mängu, kus sul on arvujada pikkusega N ning kus sulle antakse K käsku. Iga käsk on kas muutmise käsk, kus sa pead üht arvu jadas muutma, või korrutamise käsk, kus sa pead ütlema, kas mingis vahemikus olevate arvude korrutis on positiivne, negatiivne või 0. Seega otsustasid teha programmi, mis seda mängu mängib.

Sisendi esimesel real on kaks täisarvu N ja K ($1 \le N, K \le 10^5$). Teisel real on N täisarvu – esialgse jada väärtused x ($-100 \le x \le 100$). Järgmisel K real on käsud, mis on kas muutmise või korrutamise käsud. Muutmise käsk antakse ette kujul M x y ($1 \le x \le N$; $-100 \le y \le 100$), mis tähendab, et arv kohal x on nüüd väärtusega y. Korrutamise käsk antakse ette kujul K x y ($1 \le x \le y \le N$), mis tähendab, et on vaja uurida väärtust vahemikus x…y.

Väljundisse kirjutada ühele reale üks string: iga korrutamise käsu kohta peab olema kas +, kui korrutis on positiivne, -, kui korrutis on negatiivne, või 0, kui korrutis on 0.

NÄIDE 1:

```
4 6
-2 6 0 -1
M 1 10
K 1 4
M 3 7
K 2 2
M 4 -5
K 1 4
```

Vastus:

```
0+-
```

NÄIDE 2:

```
5 9
1 5 -2 4 3
K 1 2
K 1 5
M 4 -5
K 1 5
K 4 5
M 3 0
K 1 5
M 4 -5
M 4 -5
```

Vastus:

```
+-+-0
```



9.8.3 Külaskäigud

Mäe nõlval on tee, mille ääres on n maja. Igal majal on kõrgus merepinnast ning on teada, et majade kõrgused moodustavad mittekahaneva rea (Järgmine maja ei ole kunagi eelmisest majast madalamal). Samuti teame, et iga maja laius on 1 ning kahe erineva kõrgusega maja vahel on aste. Majades elab q paari lapsi, kes käivad üksteisel külas. Lapsed tahavad teada, kui pikk on pikim teelõik, mis on ühel kõrgusel, mida nad külla minnes läbivad. Laps läbib külla minnes ka oma maja pikkuse ning ka oma sõbra maja pikkuse.

Sisendi esimesel real on kaks täisarvu: n ja q ($1 \le n,q \le 100000$). Järgmisel real on n täisarvu vahemikus $-100000 \ldots 100000$ – majade kõrgused merepinnast. Järgmisel q real on kaks täisarvu: x ja y ($1 \le x \le y \le n$) – laste elukohad.

Väljundisse kirjutada q rida: Pikima ühel kõrgusel oleva teelõigu pikkus, mida laps peab külla minnes läbima.



NÄIDE:

```
10 3
-1 -1 1 1 1 1 3 10 10 10
2 3
1 10
5 10
```

Vastus:

```
1
4
3
```

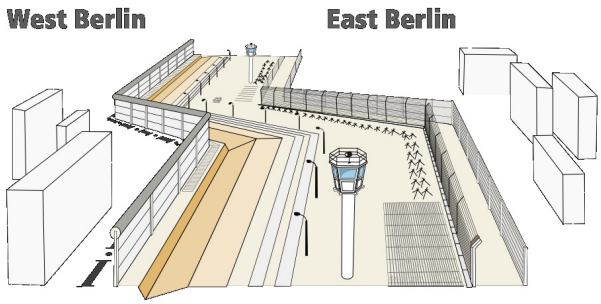
9.8.4 Berliini müür

Tulevane Berliini müür jagati enne ehitamist tulpadeks. Müüri otsustati ehitada osade kaupa, see tähendab, et valiti mingid lõigud, mille kohta oli teada, et müür peab ilmtingimata olema sellel lõigul teatud kõrgusel. Selle jaoks alustati tööd selles lõigus olevate tulpade, mis olid lühemad nõutud kõrgusest, kõrgendamisega. Iga sellise lõigu ehitamise järel tehti viieminutiline paus. Sinu eesmärk on teada, mitu korda alustati kokku tööd mingisuguse tulba üles ehitamisega.

Sisendi esimesel real on arv n ($1 \le n \le 100000$) – vajalike müürilõikude arv.

Järgmisel n real on kolm täisarvu: l, r ja h ($1 \le l < r \le 100000$; $1 \le h \le 10^9$) – müürijupi otspunktid ning kõrgus.

Väljundisse kirjutada üks täisarv: töö alustamiste arv.



NÄIDE:

```
3
5 11 3
1 10 1
3 13 2
```

Vastus:

```
14
```

9.8.5 Lambipirnid

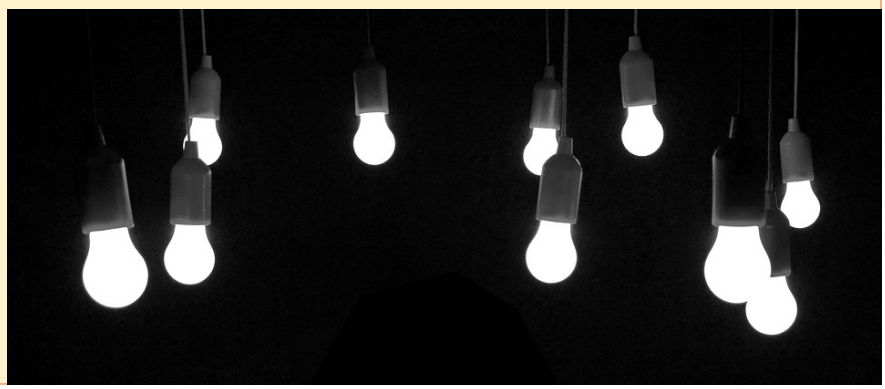
Reas on N (1 ≤ N ≤ 1000000) lambipirni. Iga pirn kas põleb või ei põle. Vahetevahel pannakse kõik mingis vahemikus olevad pirnid põlema, vahetevahel kustutatakse kõik mingis vahemikus olevad pirnid, vahel muudetakse kõigi mingis vahemikus olevate pirnide olekut ning vahel küsitakse sinu käest, mitu pirni mingis vahemikus põlevad.
Sisendi esimesel real on arv n. Järgmisel real on n märgist koosnev sõne. i-s märk sõnes on 0, kui vastav pirn ei põle, ning 1, kui vastav pirn põleb. Järgneb üksikul real arv q (1 ≤ q ≤ 1000) – päringute arv, mis tehakse. Järgmisel q real on tähemärk ning kaks arvu kujul c a b (0 ≤ a ≤ b < n). Kui c = 'F', pannakse põlema kõik pirnid vahemikus a…b. Kui c = 'E', kustutatakse kõik pirnid vahemikus a…b. Kui c = 'I', muudetakse kõikide pirnide olek vahemikus a…b. Kui c = 'S', küsitakse sinu käest, mitu pirni põlevad vahemikus a…b.
Väljundisse kirjutada mingi hulk ridu. Iga korra kohta, kui küsitakse, mitu pirni mingis vahemikus põlevad, väljastada ühele reale vastav pirnide arv.

NÄIDE 1:

```
18
101010101010001000
5
F 0 17
I 0 5
S 1 10
E 4 9
S 2 10
```

Vastus:

```
5
1
```



NÄIDE 2:

```
9
111000000
2
I 0 2
S 0 8
```

Vastus:

```
0
```

9.8.6 Taani valimised

Taanis on valimiste eel kogu maa plakateid täis. Üks kohtadest, kuhu plakateid riputati, on plakatikõrgune sein Holbækvej tänaval Sorøs. See nn Sorø müür on $l = 10^7$ millimeetrit pikk, kus iga millimeetri järel on tehtud märge. Plakat, mis seinale riputatakse, peab olema riputatud täpselt märkmest märkmeni ning selle pikkus peab olema vähemalt kaks millimeetrit. Valimiste eel riputatakse Sorø müürile n ($1 \le n \le 40000$) plakatit. Kuna plakat või osaliselt või täielikult katta ka teist plakatit, leida, mitu plakatit on valimispäeval osaliselt või täielikult nähtaval.
Sisendi esimesel real on täisarv n. Järgneval n real on kaks täisarvu:
v ja p ($1 \le v < p \le l$) – plakati esimese ning viimase kaetud millimeetri järjekorranumber. Plakatid on ette antud nende riputamise järjekorras.
Väljundisse kirjutada üksikule reale üks täisarv: Plakatite arv, mis on kas osaliselt või täielikult katmata.



NÄIDE:

```
5
1 4
2 6
8 10
3 4
7 10
```

Vastus:

```
4
```

9.8.7 Rivistus

n last seisavad reas. Sul on vaja m korda vastata küsimusele, kui pikk on k-s kõige lühem laps vahemikus i…j.
Sisendi esimesel real on kaks arvu: n ja m ($1 \le n \le 100000$, $1 \le m \le 5000$). Sisendi teisel real on jada pikkusega n – reas seisvate laste pikkused. Järgneval m real on kolm arvu:
i, j ja k ($1 \le i \le j \le n$, $1 \le k \le j - i + 1$).
Väljundisse kirjutada eraldi ridadele m täisarvu: vastused päringutele.



NÄIDE:

```
7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
```

Vastus:

```
5
6
3
```

9.8.8 Madalad majad

Linnas on n hoonet ning n − 1 teed hoonete vahel, nii et iga hoone juurest saab iga hoone juurde sõita. Sinult küsitakse m korda, mis on k-nda kõige lühema hoone kõrgus, millest möödutakse teel hoonest u hoonesse v.
Esimesel real on kaks täisarvu: N ja M (N,M ≤ 100000). Järgmisel real on N täisarvu: hoonete kõrgused. Järgneval N − 1 real on kaks täisarvu: u ja v – hoonete u ja v vahel on tee. Järgneval M real on kolm tühikutega eraldatud täisarvu: u, v ja k – küsitakse k kõige lühemat hoonet teel hoonest u hoonesse v.
Väljundisse kirjutada erinevatele ridadele M täisarvu: vastused päringutele.

NÄIDE:

```
8 5
105 2 9 3 8 5 7 7
1 2
1 3
1 4
3 5
3 6
3 7
4 8
2 5 1
2 5 2
2 5 3
2 5 4
7 8 2
```



Vastus:

```
2
8
9
105
7
```

9.8.9

Ruudukujulisel toal, mille küljepikkus on N meetrit, on põrand jaotatud N^2 ruutmeetriseks platvormiks, mille kõrgus võib olla vabalt valitud täisarv millimeetrites. Sulle antakse ette selle ruumi algseis ning sinu eesmärk on öelda, mis on selle toa mingi alamristküliku kõige kõrgema ning kõige madalama platvormi kõrgus teatud ajahetkedel. Selliste küsimuste vahel võidakse muuta mingite üksikplatvormide kõrgusi.

Sisendi esimesel real on täisarv N ($0 \le N \le 500$) – toa küljepikkus. Järgneval N real on igal real N tühikutega eraldatud täisarvu – platvormide esialgsed kõrgused. Järgmisel real on täisarv Q ($0 \le Q \le 40000$). Järgmised Q rida on antud ette kas kujul q x_1 y_1 x_2 y_2, mis tähendab, et tahetakse teada, mis on suurima ning vähima kõrgusega platvorm ristkülikus, mille ülemine vasak nurk on (x_1, y_1) ning alumine parem nurk on (x_2, y_2), või siis c x y v, mis tähendab, et platvormi, mille koordinaadid on (x, y), on nüüd kõrgusega v.

Väljundisse kirjutada mingi hulk ridu. Iga esimest tüüpi päringu kohta väljastada kaks tühikutega eraldatud täisarvu: suurima ning vähima kõrgusega platvorm nõutud vahemikus.

NÄIDE:

```
5
1 2 3 4 5
0 9 2 1 3
0 2 3 4 1
0 1 2 4 5
8 5 3 1 4
4
q 1 1 2 3
c 2 3 10
q 1 1 5 5
q 1 2 2 2
```



Vastus:

```
9 0
10 0
9 2
```

9.8.10 Paberi lõikamine

Sa lõikad üht pikka värvilist pabeririba. Pabeririba on n millimeetrit pikk ning iga tükk, mida sa selle küljest lõikad, peab olema pikkusega, mis on täisarv millimeetreid. Samuti vaheldub riba värv ainult täisarvu millimeetrite järel. Sinu ülemused on öelnud, et ükski sinu lõigatud tükk ei tohi sisaldada rohkem kui k erinevat värvi, aga nad unustasid sulle k väärtust öelda. Leia iga võimaliku k väärtuse jaoks, kus $1 \leq k \leq n$, mis on minimaalne arv lõikeid, mida sa pead ribasse tekitama.
Sisendi esimesel real on täisarv n ($1 \leq n \leq 100000$). Järgmisel real on n tühikutega eraldatud täisarvu a ($1 \leq a \leq n$) – riba iga millimeetri värv.
Väljundisse kirjutada n tühikutega eraldatud täisarvu: iga k (väiksemast suuremani) väärtuse jaoks minimaalne lõikuste arv.

NÄIDE 1:

```
5
1 3 4 3 3
```

Vastus:

```
3 1 0 0 0
```

NÄIDE 2:

```
8
1 5 7 8 1 7 6 1
```

Vastus:

```
7 3 2 1 0 0 0 0
```



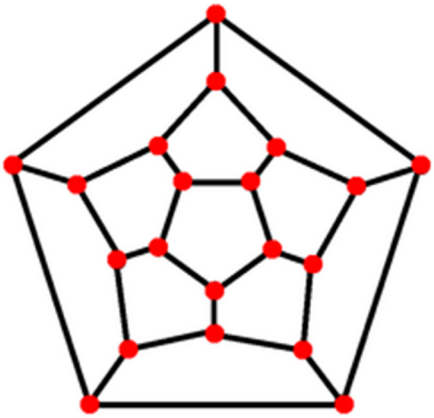
9.9 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk10 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
RMQ.cpp, RMQ.java, RMQ.py	Lõikude puu
Ribad.cpp, Ribad.java, Ribad.py	Lõikude puu laisa väärtustamisega
Sein.cpp, Sein.java, Sein.py	Lõikude puu laisa väärtustamisega
RCS.cpp, RCS.java, RCS.py	Kumulatiivne summa (Fenwicki puu)
Punktid.cpp, Punktid.java, Punktid.py	2-mõõtmeline Fenwicki puu
Redaktor.cpp, Redaktor.java, Redaktor.py	Ajalooga lõikude puu
UF.cpp, UF.java, UF.py	Ühisosata hulgad

10 GRAAFITEOORIA

Raamatu esimeses osas oli juttu põhilistest graafiteooria algoritmidest, kuid see andis valdkonnast ainult väikese maitse suhu. Tegelikult moodustab graafiteooria nii arvutiteadusest üldiselt kui ka olümpiaadidel esinevates ülesannetes märksa suurema osa. Viimastel aastatel on sageli tulnud ette võistlusi, kus graafiülesanded moodustavad ülesandekomplektist enam kui poole. Seega tuleme nüüd teema juurde tõsisemalt tagasi.



Kõrvaltoodud joonis illustreerib veel üht graafiteooria uurimisteemat – planaarseid graafe. Graafi nimetatakse planaarseks, kui selle tippe on võimalik joonistada nii, et ükski servade paar ei tarvitse omavahel lõikuda. Iga hulktahukas on teisendatav planaarseks graafiks ja joonisel on toodud korrapärase dodekaeedri selline teisendus nn Schlegeli diagrammi kujul.

10.1 FLOYD-WARSHALLI ALGORITM

Kuuendast peatükist tuttava Dijkstra algoritmiga saab leida lühima tee kahe tipu vahel graafis (kui selles ei esine negatiivse kaaluga tsükleid ega servi). Dijkstra algoritmi kasutades kogu graafi läbi käies saame lühimad teed mingist konkreetsest tipust A selle graafi igasse teise tippu. Mõnikord on aga vaja leida graafi iga tipupaari jaoks lühim tee nende tippude vahel, näiteks linnadevaheliste kauguste tabeli loomiseks.

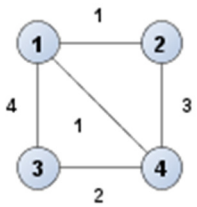
Üheks võimaluseks on muidugi kasutada Dijkstra algoritmi, valides alguspunktiks kõik graafi tipud. Dijkstra algoritmi keerukus on $O(T + S * logS)$,kus T on tippude ja S servade arv. Läbides seda algoritmi T korda, on keerukuseks $O(T^2 + T * S * logS)$. Kui graafis on vähe servi, on selline lahendus tõhus.

Teiseks võimaluseks on kasutada Floyd-Warshalli algoritmi. 1962. aastal avaldas tollal 26 aastane Illinois' Tehnoloogiainstituudis töötanud Robert Floyd algoritmi, mis kasutab dünaamilist planeerimist graafis kõigi lühimate teede leidmiseks. Sarnase algoritmini oli samal aastal jõudnud ka Stephen Warshall. Nagu ka Dijkstra algoritmis, ei tohi Floyd-Warshalli algoritmi puhul graafis olla negatiivseid tsükleid, negatiivsed servad on lubatud.

10.1.1 Ülesanne: tuleb hiljem

NB: Ülesanne lisandub hiljem

Vaatame, kuidas töötab Floyd-Warshalli algoritm järgmise graafi näitel:



10.1.2 Rekursioonivalem

Nagu mainitud, kasutab Floyd-Warshalli algoritm dünaamilist planeerimist. Üldine idee seisneb selles, et vaatame teed tipust i tippu j. Valime suvalise tipu k ning proovime seda lisada teesse tipus i tippu j. Kui teepikkus tipust i tippu j väheneb, siis oleme leidnud lühema tee tipust i tippu j läbi tipu k.

Kõigi teede leidmiseks tuleb seda teha iga tipupaari (i, j) ja iga tipu k jaoks. Saame järgneva rekursioonivalemi:

$$T(i,j,k) = \begin{cases} P(i,j), kui\ k = 0 \\ \min(T(i,j,k-1), T(i,k,k-1) + T(k,j,k-1)) \end{cases}$$

Siit on näha, et kõigepealt tuleks leida kõik teepikkused $k = 0$ korral, seejärel $k = 1$ korral jne, kuni $k = n$, kus n on tippude arv graafis.

10.1.3 Tabeli täitmine

Nagu ikka – kus on dünaamiline planeerimine, seal on tabel. Graafide juures on juba tuttavaks mõisteks kauguste maatriks kui graafi üks võimalikke esitusviise. Sellisel juhul on kauguste maatriksis vaid otseteed iga tipupaari (i, j) vahel. Sama tabelit pisut laiendades saamegi minimaalsete kauguste tabeli.

Esimesel sammul, kui $k = 0$, ongi valemi kohaselt tabelis ainult servade pikkused. Kui serv kahe tipu vahel puudub, on kauguseks lõpmatus:

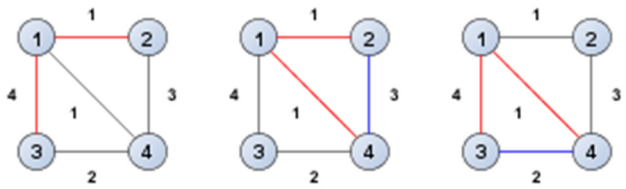
K=0	1	2	3	4
1	0	1	4	1
2		0	∞	3
3			0	2
4				0

Järgmisel sammul $k = 1$ proovime minna igast tipust igasse tippu läbi tipu 1.

$$T(2,3,1) = min\big(T(2,3,0), T\ (2,1,0) + T(1,3,0)\big) = min(\infty, 1+4) = \mathbf{5}$$

$$T(2,4,1) = min(T(2,4,0), T(2,1,0) + T(1,4,0)) = min(3, 1+1) = \mathbf{2}$$

$$T(3,4,1) = min(T(3,4,0), T(3,1,0) + T(1,4,0)) = min(2, 4+1) = 2$$



Sinisega on märgitud otsetee tippude vahel (k=0), punasega tee läbi tipu 1 (k=1).

K=1	1	2	3	4
1	0	1	4	1
2		0	**5**	**2**
3			0	2
4				0

Läbi tipu 2:

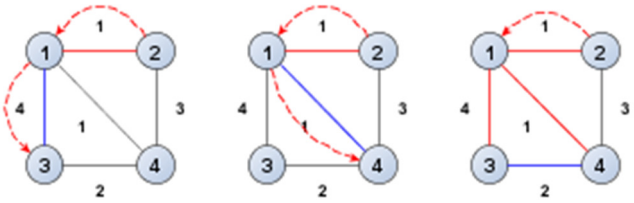
$$T(1,3,2) = min(T(1,3,1), T(1,2,1) + T(2,3,1)) = min(4, 1+5) = 4$$

$$T(1,4,2) = min(T(1,4,1), T(1,2,1) + T(2,4,1)) = \min(1, 1+2) = 1$$

$$T(3,4,2) = \min(T(3,4,1), T(3,2,1) + (2,4,1) = \min(2, 5+2) = 2$$

Siin on oluline silmas pidada, et kõige lühemad teed läbi tipu 1 ehk juhul, kui k=1 on meil juba leitud ja eelnevas tabelis kirjas! See, kas need tegelikult tippu 1 kasutasid või mitte, pole minimaalse tee kaalu leidmise juures enam oluline.

K=2	1	2	3	4
1	0	1	4	1
2		0	5	2
3			0	2
4				0



Sinisega on seni leitud parim tee ühest tipust teise. Kuna eelmisel etapil (k=1) ei parandanud ühtki teed (1,3), (1,4) ega (3,4) siis on need kõik juhtumisi üheservalised. Punasega on märgitud lühim tee läbi tipu 2, mis praeguseks teada. Näiteks tipust 3 tippu 4 minemiseks tuleb minna kõigepealt tippu 2 läbi tipu 1 ja seejärel tipust 2 tippu 4 (taas läbi tipu 1, kuna eelmisel sammul leidsime, et see on lühem kui otsetee tippude 2 ja 4 vahel).

Läbi tipu 3:

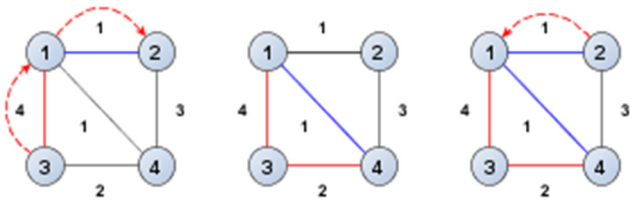
$$T(1,2,3) = \min\big(T(1,2,2), T(1,3,2) + T(3,2,2)\big) = \min(1, 4+5) = 1$$

$$T(1,4,3) = \min(T(1,4,2), T(1,3,2) + T(3,4,2)) = \min(1,\ 4+2) = 1$$

$$T(2,4,3) = \min\big(T(2,4,2), T(2,3,2) + T(3,4,2)\big) = \min(2,\ 5+2) = 2$$

Seekordki midagi ei parandata ja tabel jääb muutumatuks:

K=3	1	2	3	4
1	0	1	4	1
2		0	5	2
3			0	2
4				0



Kaugused tipu 3 kaudu. Kolmandal graafil on näha, et kasutame (üle) eelmisel sammul (k=1) leitud teed tipust 2 tippu 4 saamiseks.

Läbi tipu 4:

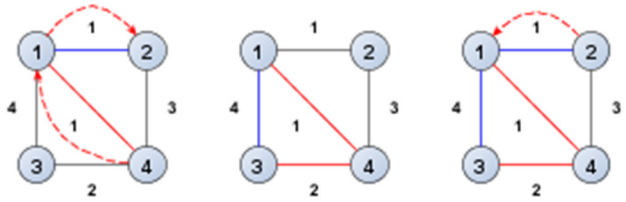
$$T(1,2,4) = \min(T(1,2,3), T(1,4,3) + T(4,2,3)) = \min(1, 1+2) = 1$$

$$T(1,3,4) = \min(T(1,3,3), T(1,4,3) + T(4,3,3)) = \min(4, 1+2) = \mathbf{3}$$

$$T(2,3,4) = \min(T(2,3,3), T(2,4,3) + T(4,3,3)) = \min(5, 2+2) = \mathbf{4}$$

Leidsime lühemad teed tippude 1 ja 3 ning 2 ja 3 vahel:

K=4	1	2	3	4
1	0	1	**3**	1
2		0	**4**	2
3			0	2
4				0



Funktsioon, mis seda teeb, on tõesti väga lihtne ja lühike:

```
int** maatriks;

void LeiaKaugused(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (maatriks[j][i] + maatriks[i][k] < maatriks[j][k]) {
                    maatriks[j][k] = maatriks[j][i] + maatriks[i][k];
                }
            }
        }
    }
}
```

Nagu siit näha, on üks oluline eelis Dijkstra algoritmi mitmekordse rakendamise ees just realisatsiooni kompaktsus.

10.1.4 Floyd-Warshalli algoritmi kasutusi

Väikese tipuhulgaga graafide korral on mõnikord kiirem kasutada Floyd-Warshalli algoritmi Dijkstra algoritmi asemel lühima tee leidmiseks kahe konkreetse tipu vahel. Keerukus on oluliselt suurem, kuid aeg koodi kirjutamiseks lühem ning kood ise lihtsam.

Küllaltki lihtsalt saab Floyd-Warshalli algoritmi muuta nii, et minimaalse tee asemel leiame maksimaalse. Praktikas enamasti just pikimat kaugust ei otsita, kuid kaaluks võib olla hoopis nt läbilaskevõime ja tahetakse leida suurima läbilaskevõimega teid/kanaleid. Sellisel juhul (teede pikkuse) summa asemel leida miinimum (kõige kehvema läbilaskevõimega koht ehk pudelikael otsustab, kui suur on kogu tee läbilaskevõime).

Lisaks saab selle algoritmiga tuvastada negatiivseid tsükleid ja leida kõige odavama (lühema) tsükli graafis. Selleks tuleks diagonaalil asuvad väärtused `[i,i]` seada väga suureks. Kui pärast algoritmi

jooksutamist ei ole mõne `i` väärtuse puhul diagonaalil asuv `[i,i]` väärtus enam lõpmatu, siis leidub graafis tsükkel, mis läbib tippu `i`. Kõige väiksema väärtusega selline tsükkel ongi minimaalne tsükkel graafis. Kui mõni väärtus `[i,i]` on negatiivne, siis on graafis negatiivne tsükkel.

Floyd-Warshalli algoritmi abil saab leida ka graafi **diameetri** ehk suurima võimaliku minimaalse kauguse kahe graafi tipu vahel. Selleks on loomulikult leitud minimaalsete kauguste tabelis maksimaalne väärtus. Eelpool vaadeldud graafi diameeter on 4.

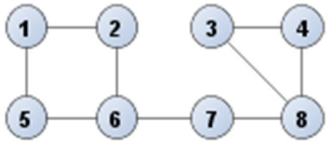
Lahendiks olevast kauguste tabelist saab lihtsalt näha, kas kahe punkti vahel üldse leidub tee või mitte. Kui ainult seda infot vaja on, võimegi tabelis meeles pidada vaid tee leidumist (või mitteleidumist). Taas on see praktiline ainult siis, kui tippude hulk on väike.

Sel moel aga saab tuvastada ka tugevalt sidusad komponendid suunatud graafis. Pärast algoritmi jooksutamist kontrollime tulemuseks saadud tabelit. Näiteks mingi tipuga i kuuluvad samasse sidususkomponenti need tipud `j`, kus `[i,j] = [j,i]`.

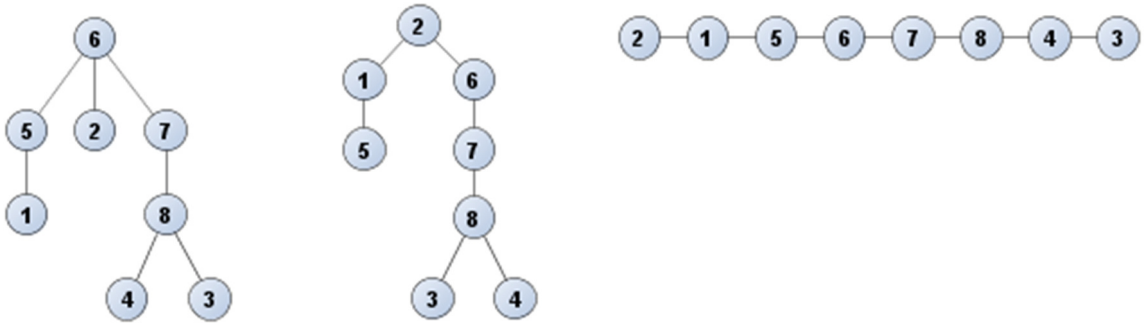
10.2 TOESE MÕISTE

Tuletame meelde, et puud saab vaadelda kui graafi erijuhtu. Graaf G on puu, kui G on sidus ja tsükliteta.

Sidusa graafi G **toeseks** ehk **aluspuuks** nimetatakse tema alamgraafi T, mis on puu ja mis sisaldab graafi G kõiki tippe. Ühel graafil võib olla mitu erinevat toesepuud. Näiteks graafi



toesepuudeks võivad olla järgmised graafid:



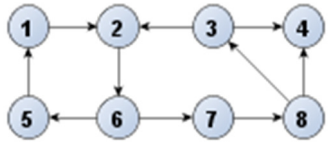
10.2.1 Servade liigitamine

Graafi sügavuti läbides tekib nn sügavuti otsimise puu, mis on graafi aluspuu. Milline see puu täpselt on, sõltub sellest, millisest tipust sügavuti läbimist alustatakse ning mis järjekorras naabreid valitakse. Sügavuti otsimisel saab suunatud graafis eristada nelja tüüpi servi:

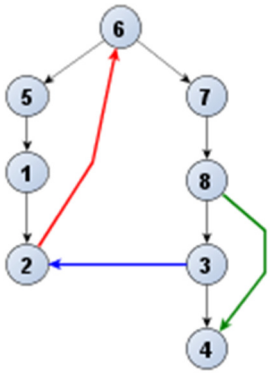
- Puu servad (*tree edges*) ehk servad, mis antud valiku juures moodustavad graafi toesepuu.
- Tagasiviivad servad (*back edges*) ehk servad, mis ühendavad tippu mõne ülema tipuga.

- Edasiviivad servad (*forward edges*) ehk servad, mis ühendavad tippu mõne alama tipuga, kuid ei ole puu servad.
- Ristuvad servad (*cross edges*) ehk servad, mis ei kuulu ühtegi eelnevasse liiki.

Alustades järgmise suunatud graafi läbimist tipust numbriga 6 ja töödeldes alati enne väiksema numbriga naabri



saame järgneva toesepuu koos lisaservadega:



Joonisel mustad servad on puu servad, punane serv on tagasiviiv serv, roheline edasiviiv serv ning sinine on ristuv serv.

Suunamata graafis räägime harilikult kahest esimesest tüübist, kuna muud tüüpi servad tegelikult langevad kokku.

10.2.2 DFS koos servade liigitamisega

Võtame tavalisel sügavuti läbimisel abiks veel ühe tipu oleku, eristades vaatlemata, leitud ja töödeldud tippe. Peame meeles ka tipu vanema toesepuus ehk tipu, mille kaudu antud tipp esmalt leiti. Nii saame lihtsa algoritmi, kuidas eristada loodavas toesepuus puu servad, tagasiviivad servad ja edasiviivad/ristuvad servad. Algoritm ise on järgmine:

1. Vaata graafi tippu u, märgi see leituks.
2. Vaatle tipu u naabrit v
   a. Kui seda naabrit pole veel leitud, pane u selle vanemaks ja jätka punktist 1 tipu v kohta. Serv (u, v) on puu serv.
   b. Kui naaber on varem leitud ja u ei ole v vanem, siis on serv (u, v) tagasiviiv serv.
   c. Kui naaber on juba uuritud, siis on serv (u, v) edasiviiv või ristuv serv.
3. Kui kõik u naabrid on töödeldud, märgi tipp u töödelduks.

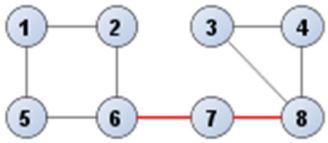
Funktsioon, mis liigitab servad:

```
void liigitaServ(int u) {
    tipud[u] = LEITUD;
    for (int j = 0; j < (int)graaf[u].size(); j++) {
        int v = graaf[u][j];
        if (tipud[v] == UUS) {
            vanem[v] = u;
            cout << "Puu serv " << u+1 << " " << v+1 << endl;
            liigitaServ(v);
        }
        else if (tipud[v] == LEITUD) {
            if (v != vanem[u]) {
                cout << "Tagasiviiv serv " << u+1 << " " << v+1 << endl;
            }
        }
        else if (tipud[v] == UURITUD) {
            cout << "Edasiviiv või ristuv serv " << u+1 << " " << v+1 << endl;
        }
    }
    tipud[u] = UURITUD;
}
```

10.3 Sillad ja Artikulatsioonipunktid

Sidusa graafi $G = (T, S)$ serva $s \in S$ nimetatakse **sillaks**, kui selle serva eemaldamisel graafis G muutub graaf mittesidusaks. Võib öelda ka, et sild on graafi serv, mille eemaldamisel kasvab graafi sidususkomponentide arv.

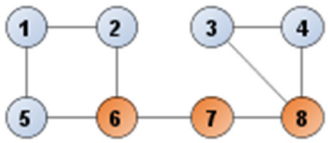
Näiteks graafis:



on sildadeks servad (6,7) ja (7,8) (joonisel märgitud punasega).

Sidusa graafi $G = (T, S)$ tippu $t \in T$ nimetatakse **artikulatsioonipunktiks**, kui selle tipu eemaldamisel graafist *koos sellesse suunduvate ja sellest väljuvate servadega* muutub graaf mittesidusaks. Võib öelda ka, et artikulatsioonipunkt on graafi tipp, mille eemaldamisel kasvab graafi sidususkomponentide arv.

Näiteks graafis



on tipud 6, 7 ja 8 (joonisel punased) artikulatsioonipunktid.

Sillad ja artikulatsioonipunktid on omavahel üsna tihedalt seotud. Silla otsatipud on artikulatsioonipunktid, välja arvatud juhul, kui sild on ainus otspunkti sisenev või sealt väljuv serv. Samas iga kahe artikulatsioonipunkti vahel asuv serv ei pruugi olla sild.

10.3.1 Ülesanne: Muinastulede öö

Juba viikingiajal oli kombeks meie esiisadel ohust märku anda süüdatud lõketega. Siis toimis rannikul laevateede läheduses juba ühtne valvesüsteem, mis ulatus ka sisemaale. Sellesse pidi iga kogukond andma lõkke süütamisel ja hoidmisel oma kindlaksmääratud panuse ja kandis küllalt ranget vastutust teate mitteedastamise korral. Süsteem oli ise lihtne – kui lõket nähti, tuli süüdata ise teine, et see paistaks järgmiste elupaikadeni jne. On arusaadav, et mõned lõkked on sõnumi levimiseks igasse maanurka olulisemad kui teised. Sinu ülesanne on leida, milliseid lõkkeid ei tohi kindlasti süütamata jätta. Ülesandes eeldatakse, et kui lõke kohas A paistab kohta B, siis lõke kohas B paistab kohta A.



Sisendi esimesel real on üks arv n – lõkkekohtade arv. Järgneval n real on igaühel 0 – n-1 arvu – n.dast lõkkest paistvate lõkete järjekorranumbrid. Väljastada nende lõkkekohtade järjekorranumbrid, mida ei tohi kindlasti jätta süütamata. Järjekorraumbrid väljastada kasvavas järjestuses.

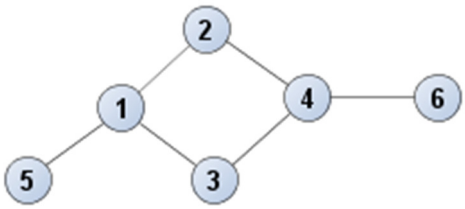
NÄIDE:

```
6
2 3 5
4
4
6
```

Vastus:

```
1 4
```

Selles ülesandes on lõkked graafi tippudeks ja kahe lõkke vahel on serv, kui ühest lõkkekohast paistab teine lõke. Kui graaf loodud, tuleb leida need lõkked, mis ei tohi kindlasti süütamata jääda ehk mille väljalülitamise korral ei ole moodustunud graaf enam sidus, st need lõkked, mis on artikulatsioonipunktideks.



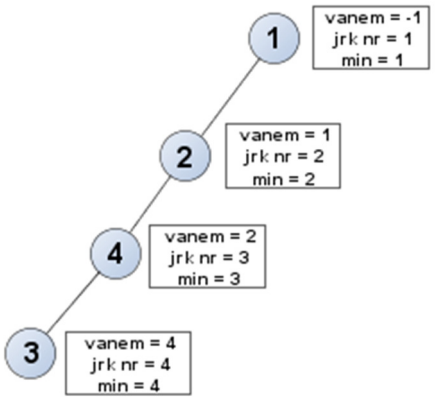
Ülesande näiteandmetele vastav graaf

Vahetu lähenemine oleks selline, et eemaldame graafist iga tipu ning kontrollime seejärel sidususkomponentide arvu. Kui sidususkomponentide arv suureneb, oleme leidnud artikulatsioonipunkti, kui mitte, siis antud punkt ei ole artikulatsioonipunkt. Selle keerukuseks oleks $O(V * (V + E))$.

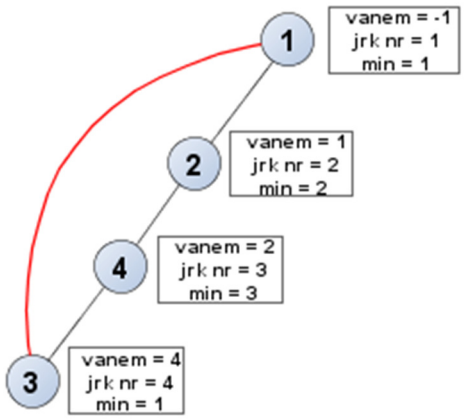
Saab ka paremini. Kasutades sügavuti läbimist, nimetame tipu u tipu v eellaseks, kui tippu v jõudmiseks läbisime tipu u. Tipp u on artikulatsioonipunkt kui

- u on juur toeses ja tal on vähemalt 2 alluvat.
- u ei ole juur ja tal on selline alluv v, millest algavas alampuus ei ole ühtki serva mõne u eellasega.

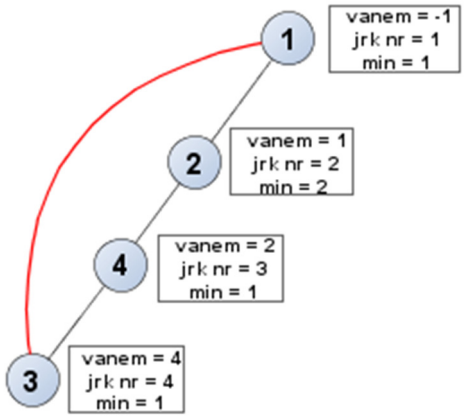
Nii on artikulatsioonipunkti leidmise algoritmi aluseks toesepuu. Hakkame seda looma, aga iga tipu juures peame meeles lisaks kahte numbrit – tipu avastamise järjekorranumbrit ehk leidmise aega ja üht teist numbrit, mida nimetame minimaalseks leidmise ajaks – see on vähima järjekorranumbriga tipu, millesse tema alampuust jõuda võib, number. Lisaks peame meeles, kes on tipu vanem. Alustades tipust 1 ja võttes tippe järjekorras, saame esialgu järgmise toesepuu alguse:



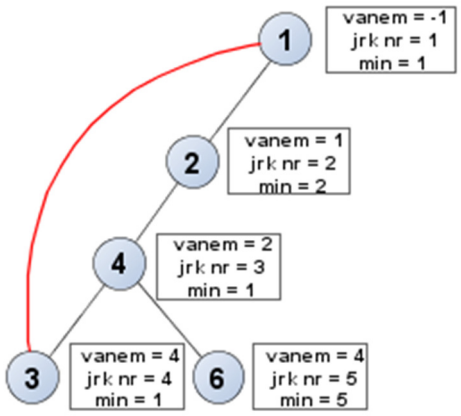
Nüüd tipust 3 läheb tagasiviiv serv tippu 1. Uuendame tipu 3 minimaalset leidmise aega milleks on kas tipu 3 senine minimaalne leidmise aeg või tipu 3 naabri 1 järjekorranumber – see, mis on neist kahest väiksem. Saame:



Kuna tipu 3 naabrid on nüüd kõik töödeldud, läheme tagasi tipu 4 juurde ja uuendame selle minimaalset aega, milleks saab tema alluva, tipu 3 minimaalne aeg:

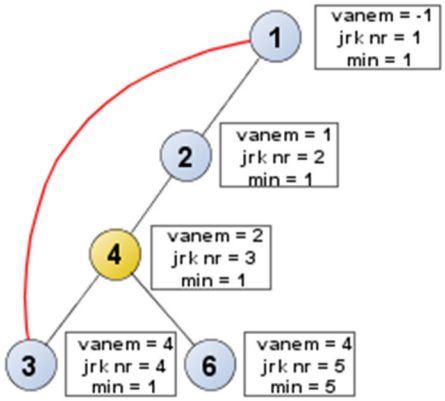


Hetkel tipp 4 ei kvalifitseeru artikulatsioonipunktiks, kuid sel tipul on veel naabreid, nimelt tipp 6:

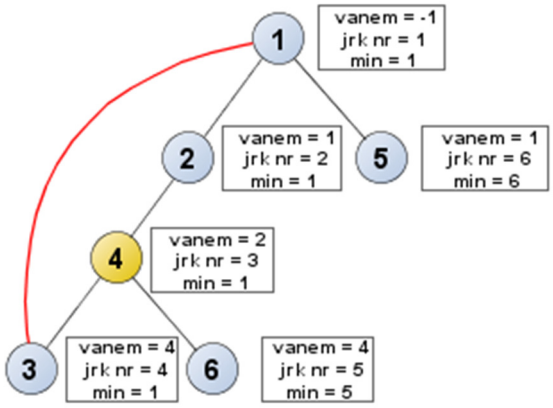


Kuna tipul 6 naabreid ei ole, siis sellega midagi teha ei ole vaja. Selline tipp ei saa olla ka artikulatsioonipunktiks. Küll aga on artikulatsioonipunktiks tipp 4, sest see on ainus tipp, mille kaudu saab tippu 6. Märgime tipu 4 artikulatsioonipunktiks (joonisel edaspidi kollane).

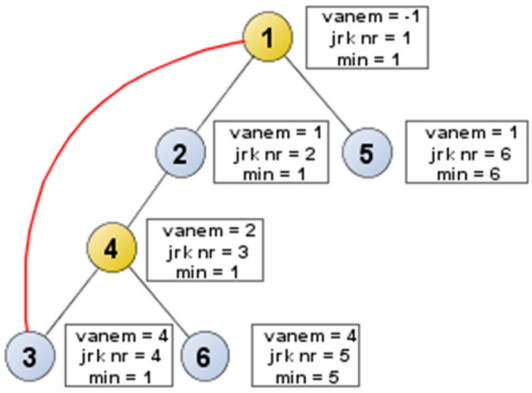
Tipu 4 naabrid on nüüd kõik üle vaadatud ja läheme tagasi tipu 2 juurde. Siin saame taas parandada madalaimat aega tipu 4 andmete põhjal ning otsustada, et 2 hetkel ei ole artikulatsioonipunktiks:



Ka tipul 2 ei ole rohkem alluvaid ja liigume tagasi tipu 1 juurde. Veel ei muutu seal midagi, jätkame tipu 1 naabriga, tipp 5-ga. Lisame selle puusse:



Kuna tipul 5 naabreid pole peale tema vanema, siis analoogselt tipuga 6 see mingit töötlemist ei vaja. Nüüd aga, kuna tipp 1 on puu juur ja tal on kaks alluvat, on ka tipp 1 artikulatsioonipuntiks.



Selle algoritmi autoriteks on John Hopcroft Cornelli Ülikoolist ja Robert Tarjan Princetoni Ülikoolist ning see pärineb aastast 1973. See algoritm on lineaarse keerukusega - $O(V + E)$.

```
int n // tippude arv
int jrk_nr = 0; // mitmendana tipp leitakse
bool ap[n] = { 0 }; // leitud artikulatsioonipunktid
bool kylastatud[n] = {0}; // kas tipp on külastatud
int aeg[n] = {0}; // külastamise järjekorranumber ehk „aeg“
int madalaim[n] = {0}; // vähim sellise tipu külastusaeg, millel on antud tipuga serv.
int vanemad[n]; // hoiame iga tipu jaoks meeles tema vanema

void leiaAPd(int u)
{
    int alluvaid = 0; // loendame alluvaid alampuus
    kylastatud[u] = true; // siin me oleme
    aeg[u] = madalaim[u] = ++jrk_nr; // alguses on madalaim avastamise aeg

    vector<int>::iterator i;
    for (i = graaf[u].begin(); i != graaf[u].end(); ++i) {
        int v = *i;  // v on u naaber

        if (!kylastatud[v]) { //Kui v ei ole veel külastatud, siis on see u alluv puus
            alluvaid++; // loeme selle u alluvaks
            vanemad[v] = u; // ja paneme u v vanemaks
            leiaAPd(v); // jätkame v-ga

            madalaim[u] = min(madalaim[u], madalaim[v]);

            // u on artikulatsioonipunkt, kui:
            if (vanemad[u] == -1 && alluvaid > 1) // u on juur ja tal on mitu alluvat
                ap[u] = true;

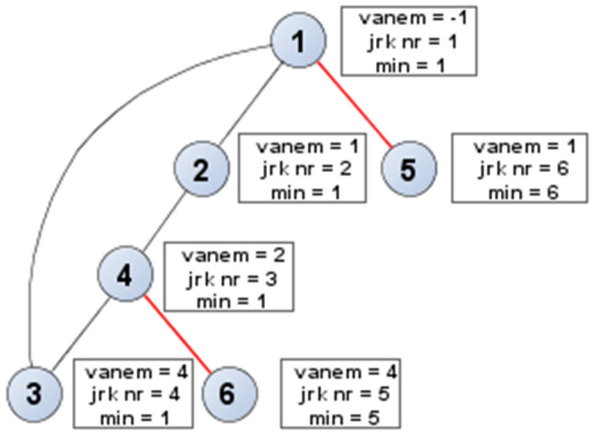
            if (vanemad[u] != -1 && madalaim[v] >= aeg[u]) // u alt ei ole tagasiserva
                ap[u] = true;
        }

        else if (v != vanemad[u]) // kui v ei ole u vanem
            madalaim[u] = min(madalaim[u], aeg[v]);
    }
}
```

10.3.2 Sildade leidmine

Sildade leidmise algoritm on sarnane artikulatsioonipunktide leidmisele. Tipp u on artikulatsioonipunkt siis, kui `madalaim[v] >= aeg[u]`, serv (u, v) on sild siis, kui `madalaim[v] > aeg[u]`. Kuna otsime servi, siis mingid erandid juurtipu jaoks ei ole sildade leidmisel vajalikud.

Eelmises ülesandes saadud andmete põhjal on sildadeks servad (1, 5) ja (4, 6):



Siin on protseduur, mis leiab sillad:

```
void leiaSild(int u)
{
    kylastatud[u] = true; // siin me oleme
    aeg[u] = madalaim[u] = ++jrk_nr; // leidmise jrk

    // vaatame kõiki naabreid
    vector<int>::iterator i;
    for (i = graaf[u].begin(); i != graaf[u].end(); ++i) {
        int v = *i; // v  on u naaber
        if (!kylastatud[v]) { //kui ei ole enne leitud
            vanemad[v] = u; // paneme u vanemaks
            leiaSild(v); // ja töötleme graafi sealt edasi

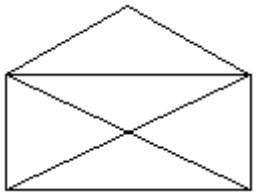
            madalaim[u] = min(madalaim[u], madalaim[v]);

            //Kas v-l on ühendus mõne u eellasega? Kui mitte, siis on u-v sild
            if (madalaim[v] > aeg[u])
                cout << u << " " << v << endl;
        }

        // kui oleme enne külastanud, muudame madalamat aega
        else if (v != vanemad[u])
            madalaim[u] = min(madalaim[u], aeg[v]);
    }
}
```

10.4 EULERI GRAAF

Kunagi oli populaarne ajaviitemäng, kus tuli joonistada paberile lahtine kirjaümbrik ilma pliiatsit paberilt tõstmata ja ühtki joont topelt tõmbamata. Selleks on rohkem kui üks võimalus, aga neid kõiki ühendab see, et alustada tuleb ühest alumistest punktidest ning lõpetada teises alumises punktis. Graafiteoorias nimetatakse graafis sellist teed ehk ahelat, mis läbib iga selle graafi serva täpselt üks kord, **Euleri ahelaks.**



Euleri tsükliks nimetatakse sellist Euleri ahelat, mis algab ja lõppeb samas tipus. Sellist graafi, milles leidub Euleri tsükkel, nimetatakse **Euleri graafiks**, graafi, milles leidub Euleri ahel aga **Euleri semigraafiks.**

Selle kindlaks tegemine, kas graaf on Euleri graaf, on väga lihtne. Graaf peab olema sidus ja selles peavad olema kõik tipud sellised, et sinna viivate teede arv on paarisarv. See tingimus on nii tarvilik (kehtib igal Euleri graafil) kui ka piisav (kehtib ainult Euleri graafidel). Euleri semigraafis tohib olla täpselt kaks tippu, mille korral see ei kehti. Sel juhul algab Euleri ahel ühes nendest tippudest ja lõppeb teises, nagu ka kirjaümbriku näites. Kontrolli, kas graaf on Euleri graaf või semigraaf, saab teha üsna lihtsalt ajaga $O(V + E)$ ning seda tehakse sageli juba graafi sisselugemisel.

10.4.1 Euleri tsükli leidmine

Keerulisem on aga leida tegelik tee või tsükkel. Selleks on kaks tuntud algoritmi: Fleury algoritm aastast 1883 ja Hierholzeri algoritm aastast 1873.

Fleury põhiideeks on „ära põleta sildu!“:

1. Kui graafis on 2 paaritut tippu, alusta ühest neist. Kui selliseid tippe pole, alusta, kust tahad.
2. Vaata tipust väljuvaid servi: kui nende seas on selline, mis ei ole sild, vali see esimesena.

Fleury algoritm on küll väga elegantne, kuid kuna selle jaoks on vaja leida graafist sillad, on selle keerukuseks $O(S^2)$, kus S on servade arv graafis. Samuti ei saa Fleury algoritmi kasutada suunatud graafidel.

Hierholzeri idee põhineb tsüklite eemaldamisel – kui eemaldada Euleri graafist kõik mingi tsükli servad, siis allesjäänud alamgraafid on samuti Euleri graafid. Algoritm ise on järgmine:

1. Alusta suvalisest tipust u.
2. Mine suvaliselt mööda servasid, kuni oled tagasi tipus u (kui graafis on Euleri tsükkel, pole kinnijäämine võimalik). Tulemuseks on tsükkel.
3. Niikaua, kui meie leitud tsüklis on mõni tipp v, millest väljub mõni serv, mis pole leitud tsükli osa:
   a. Koosta uus tsükkel alustades tipust v.
   b. Lisa uus tsükkel esialgsele.

Kui hoida meeles, kus on ülejäävaid servi, töötleme iga serva vaid ühe korra ja sellisel juhul on kogu algoritmi keerukus $O(E)$.

Hierholzeri algoritmi saab kasutada ka suunatud graafidel.

Vaatame järgmist ülesannet:

10.4.2 Loomanimed

Taas üks tuntud laste ajaviitemäng on selline, et öeldakse järjestikku kordamööda loomanimesid, aga nii, et iga järgmise looma nimi peab algama selle tähega, millega eelmine lõppes. Näiteks ütleb esimene mängija „rebane“, teine peab ütlema e-tähega algava looma, nt „eesel“, kolmas selle peale l-iga algava looma, nt „lõvi“ ning seejärel esimene i-ga algava, nt „ilves“ jne.
Sulle on antud hulk tähestiku järjekorras loomanimesid ja sa pead leidma, kas neist saab moodustuda terve mäng ja kui saab, siis leidma vähemalt ühe sobiva järjestuse.
Sisendi esimesel real on loomade arv `n` ja järgneval `n` real igaühel üks loomanimi. Loomanimed koosnevad kõik väikestest ladina tähtedest.
Väljastada loomanimed selles järjekorras, mis moodustaks mängu, iga nimi eraldi real. Kui kõikidest nimedest mängu moodustamine pole võimalik, väljastada ühele reale „`ei saa`“.

NÄIDE 1:

```
8
eesel
lehm
lammas
mutt
rebane
siil
tigu
tiiger
```

Vastus:

```
tiiger
rebane
eesel
lammas
siil
lehm
mutt
tigu
```

NÄIDE 2:

```
4
karu
koer
rott
makaak
```

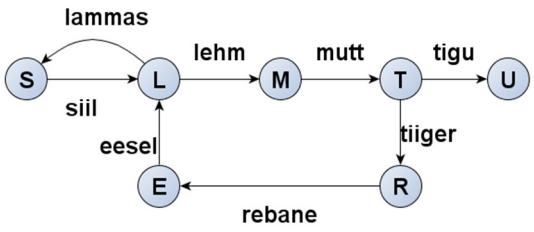
Vastus:

```
ei saa
```



Pildil skulptuur Breemeni linna moosekantidest.
Kui pole veel lugenud, siis loe nende lugu!

Selle ülesande lahendamiseks tuleb kõigepealt analüüsida, milline graaf tekib. Üks võimalus ongi esitada graaf kujul, kus tippudeks on kõikvõimalikud erinevad algus ja lõpptähed ning loomanimed on suunatud servad nende tippude vahel:

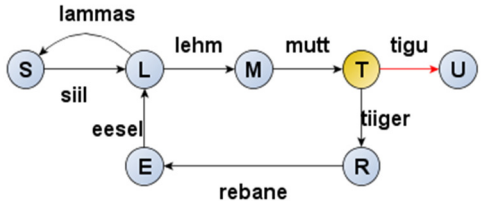


Lahendiks on Euleri tee selles graafis, kui selline leidub. Seega teiseks ülesandeks on teha kindlaks, kas see tee leidub ning ühes sellega, milline on tee alguspunkt. Seda teeb järgnev funktsioon:

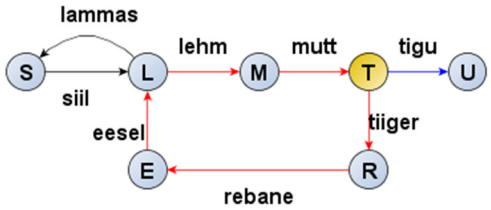
```
bool onTee()
{
    int alguseid = 0;
    int loppe = 0;
    for (int i = 0; i < 26; i++) {
        int valja = graaf[i].size();
        if (sisse[i] > valja + 1 || sisse[i] + 1 < valja)
            return false; //sisse ja väljatulevate vahe on rohkem kui 1
        if (sisse[i] == valja + 1) {
            loppe++; // võimalik lõpp
        }
        if (sisse[i] + 1 == valja) {
            alguseid++; // võimalik algus
            algus = i;
        }
        if (valja == 0 && algus == i)
            algus++; // tsükli jaoks esimese tipp, kust on servi;
    }
    if (alguseid < 2 && alguseid == loppe) { // üks lõpp ja üks algus või ei kumbagi
        return true;
    }
    return false;
}
```

Kui on kindlaks tehtud, et Euleri tee eksisteerib ning ühes sellega selle algustipp (tsükli korral sobib milline tahes, alustame tähestikus esimesest tähest, kust servi väljub). Antud näites väljub tipust T 2 serva, siseneb aga 1, seega peab tipp T olema Euleri tee alguseks. Tee lõpuks on tipp U, millest väljuvaid servi pole.

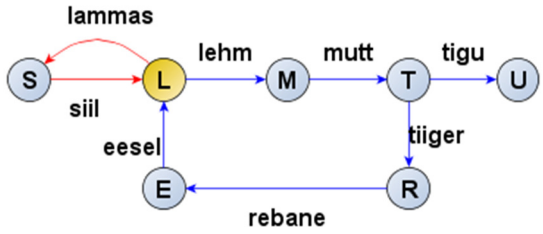
Alustame tipust T ja võtame suvalise serva, näiteks „tigu“:



Eemaldades selle serva, proovime edasi liikuda tipust U. Kuna aga sealt väljuvaid servi (rohkem) pole, siis oleme jõudnud tee lõppu. Praegu on meil kogu teeks „tigu“, aga graafis on veel servi. Läheme tagasi algusserva T juurde ja valime sealt teise väljuva serva „tiiger“. Sellest omakorda väljub „rebane“ jne:



Nendest kahest käigust saame kokku tee `tiiger-rebane-eesel-lehm-mutt-tigu`. T-st rohkem väljuvaid servi ei ole, M-ist ka mitte, aga L-ist on. Alustame järgmist teejuppi sealt:



Saame lisada olemasolevale teele veel kaks looma ja tee on nüüd `tiiger-rebane-eesel-lammas-siil-lehm-mutt-tigu`. Nüüd on ka kasutatud kõik loomanimed (servad) ja vastus on käes.

Funktsioon, mis leiab Euleri tee ja väljastab selle:

```
void leiaTee()
{
    tipud.push(algus);
    int u = algus; // vaadeldav tipp

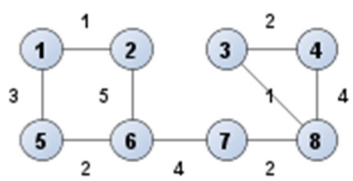
    while (!tipud.empty()) {
        if (!graaf[u].empty()) { // on veel väljuvaid servi
            tipud.push(u);
            int v = graaf[u].top();
            graaf[u].pop();
            u = v;
        }
        else { // jõudsime tippu, kus pole väljuvaid servi
            tee.push_back(u); // u on viimane tipp, mis pole veel tees
            u = tipud.top(); // võtame järgmise (eelmise) tipu
            tipud.pop();
        }
    }

    // tee väljastamine
    for (int i = tee.size() - 1; i > 0; i--) {
        cout << servad[tee[i]][tee[i - 1]].top() << endl;
        servad[tee[i]][tee[i - 1]].pop();
    }
}
```

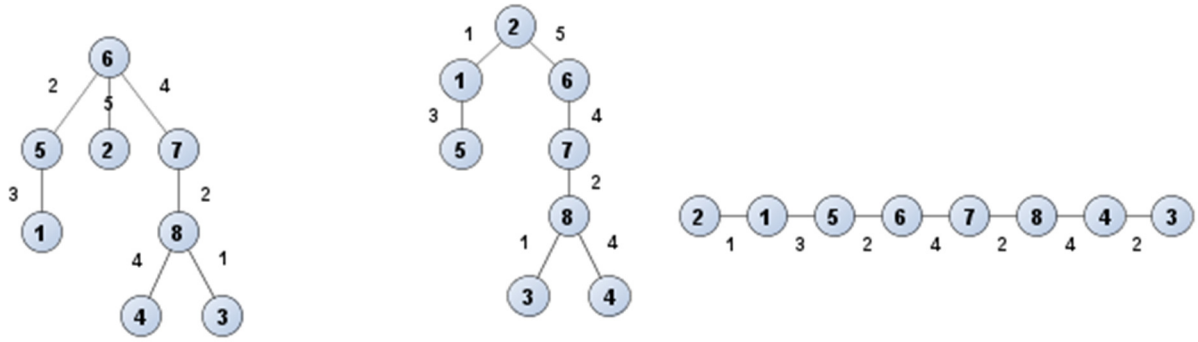
10.5 Minimaalne Toes

Minimaalsest toesest saab rääkida siis, kui tegemist on kaalutud graafiga. Nimetame toesepuu kaaluks kõigi tema servade kaalude summat. Erinevate toesepuude kaalud võivad olla erinevad.

Näiteks graafis

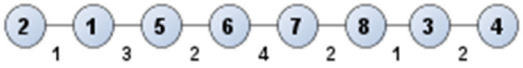


on eelpool toodud toesepuude kaalud



esimesel puul 21, teisel 20 ja kolmandal 18.

Probleemide lahendamisel tuleb ette, et on vaja leida just graafi kõige väiksema ehk minimaalse kaaluga toes. Antud näitepuudest pole ükski minimaalse kaaluga, selleks on hoopis puu kogukaaluga 15:



10.5.1 Minimaalse toese leidmine

Minimaalse toese leidmiseks on kaks tuntud algoritmi. Esimene neist on Kruskali algoritm, mille idee on järgmine:

1. Alusta tühjast puust.
2. Kuni lisatud on vähem kui $n - 1$ serva, lisa lühim serv nende hulgast, mis ei tekita eelnevatega koos tsüklit.
3. Abiks on, kui sorteerime alguses kõik servad kaalude järgi
4. Meil tekib hulk sidususkomponente – nende ühendamiseks saab kasutada ühisosata hulki (*disjoint sets, union-find*)

Primi algoritmi töötab aga järgmiselt:

1. Vali suvaline tipp toesepuu alguseks.
2. Kuni leidub veel tippe, mis puusse pole lisatud, leia lühim serv, mis seob mõnda puu tippu seal mitteolevaga,
3. Lisa see serv puusse.

10.5.2 Ülesanne: Elektrikatkestus

Meie linnakeses oli suur torm ja seetõttu langes rivist välja hulk elektriliine ning praktiliselt kogu linn on elektrita. Kuna varasem elektrivõrk oli üles ehitatud eri aegadel tõmmatud liinidest, siis ei pruukinud iga tarbija olla ühendatud võrguga just kõige lühemat teed pidi ning kokku sarnanes kogu süsteem nagunii rohkem taldrikutäie spagettidega. Igatahes tuleb nüüd pea kogu elektrisüsteem üles ehitada nii kiiresti ja odavalt kui võimalik. Sulle on antud tarbijate nimekiri ja nendevahelised kaugused, kus ühendus tõmmata võimalik on. Väljasta uute liinide minimaalne kogupikkus, mis varustab kõik tarbijad elektriga.

Sisendi esimesel real on kaks täisarvu: tarbijate arv m ja ühenduste arv n. Järgneval n real on igaühel 3 täisarvu x, y ja k, kus x ja y on tarbijate numbrid, kelle vahele ühenduse tekitada saaks ja k on nende tarbijate vaheline kaugus.

Väljastada üks täisarv minimaalne liinide kogupikkus, millega saab kõik tarbijad omavahel ühendada.



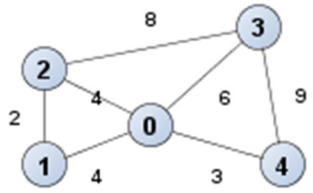
NÄIDE:

```
5 7
1 0 4
1 2 2
0 2 4
0 4 3
0 3 6
2 3 8
3 4 9
```

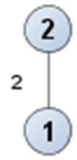
Vastus:

```
15
```

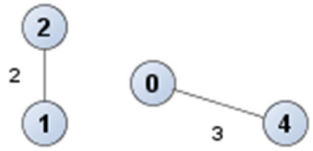
Esiteks, kuna tegu on graafiülesandega, tuleb luua graaf. Visuaalselt näeb näiteandmetel põhinev graaf välja selline:



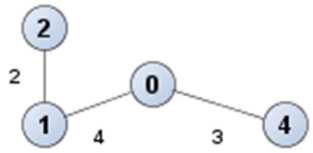
See on küllaltki lihtne ja läbipaistev minimaalse toese ülesanne. Kasutame ülesande lahendamiseks Kruskali algoritmi. Hakkame looma graafi aluspuud ja kõigepealt valime sinna kõige kasulikuma ehk kõige väiksema kaaluga serva, milleks on (1,2):



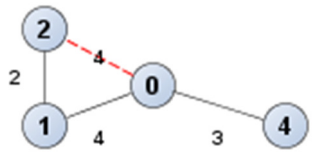
Järgmiseks vaatame kaalult järgmist serva (0,4). See ei tekita eelmise servaga tsüklit, lisame selle:



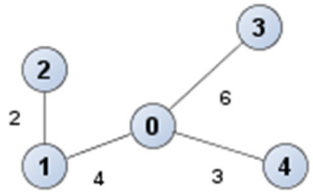
Nüüd on meil valida kahe serva vahel, mis on võrdse kaaluga: (0,1) ja (0,2). Proovime lisada neist esimese, mis ka õnnestub, kuna tsüklit ei teki:



Siin tasub tähele panna, et uus serv ühendas kaks enne iseseisvat komponenti. Seejärel proovime lisada serva (0,2). Seda aga toesesse lisada ei saa, kuna tipud 0 ja 2 on juba samas komponendis ja nii tekiks tsükkel:

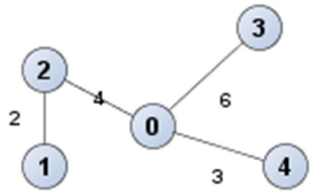


Seega serv (0,2) ei kuulu minimaalsesse toesesse, jätame selle kõrvale ja jätkame servaga (0,3). Selle saab taas lisada olemasolevale komponendile:



Kindlasti ei tasu ka ära unustada, et toesepuus, nagu igas puus, on täpselt $n-1$ serva, kus n on tippude arv. Kuna graafis on 5 tippu ja oleme lisanud juba 4 serva, siis ülejäänud servi enam vaatama ei pea – minimaalne toesepuu on leitud.

Kui oleksime valinud serva (0,2) enne serva (0,1), saanuksime järgmise minimaalse toesepuu:



Seega ka minimaalseid toesepuid võib olla mitu.

Konkreetse ülesande teeb lihtsamaks asjaolu, et tegelikku toest leida pole vajagi – piisab selle kaalust. Seega toesesse kuuluvate servade meelespidamise asemel piisab, kui toesesse kuuluva serva leidmisel lisame selle serva kaalu summale. Kuna tegelikke minimaalseid toesepuid võib olla mitu, siis tegelikult selline kogukaalu leidmine on küllaltki tavaline.

Siin on funktsioon, mis leiab minimaalse liinide pikkuse, kasutades Kruskali algoritmi:

```
int leiaPikkus()
{
    int servi_puus = 0;
    int jrgm_serv = 0;
    int vastus = 0;

    sort(servad.begin(), servad.end()); // sorteeritud kauguse (kaalu) järgi

    UF hulgad = UF(m); // iga tipp on eraldi hulk
    while (servi_puus < m - 1) // puus on m-1 serva
    {
        int kaugus = servad[jrgm_serv].first;
        pair<int, int> otsad = servad[jrgm_serv].second;

        //kui ei teki tsüklit, lisame serva puusse
        if (!hulgad.connected(otsad.first, otsad.second)) {
            vastus += kaugus;
            servi_puus++;
            hulgad.merge(otsad.first, otsad.second);
        }
        jrgm_serv++;
    }

    return vastus;
}
```

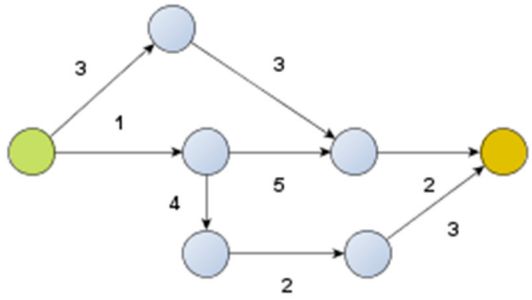
10.6 MAKSIMAALNE VOOG JA MINIMAALNE LÕIGE

Probleem sellest, kui suur on mingi võrgustiku (milleks võib olla veevärk, teedevõrk, kommunikatsioonistruktuur vms) summaarne läbilaskevõime, oli aktuaalne juba enne arvutite laialdast kasutuselevõttu. Probleemiga seonduvad kaks peamist mõistet on maksimaalne voog ja minimaalne lõige. 1956. aastal panid Ameerika matemaatikud Lester Ford ja Delbert Fulkerson tähele, et need mõisted on omavahel tihedalt seotud ning leiutasid sellele tähelepanekule toetudes Ford-Fulkersoni algoritmi. Enne algoritmi kirjelduseni jõudmist tutvume veel mõningate graafiteooria mõistete formaalsemate definitsioonidega.

10.6.1 Võrk

Võrguks nimetatakse suunatud sidusat graafi, mille igale kaarele on vastavusse seatud mittenegatiivne reaalarv ehk kaare **läbilaskevõime (***i.k. capacity*).

Võrku, millel on täpselt üks tipp, kust kaared ainult väljuvad ja täpselt üks tipp, kuhu kaared ainult sisenevad, nimetatakse **transpordivõrguks** (i.k. *flow network*, *transportation network*). Tippu, kust kaared väljuvad, nimetatakse **lähteks** (i.k. *source*) ja tippu, kuhu kaared suunduvad, **suudmeks** (i.k. *sink*). Kuna edaspidi käsitleme peamiselt transpordivõrke, siis edaspidi mõeldakse sõna „võrk“ tähenduses justnimelt transpordivõrku.

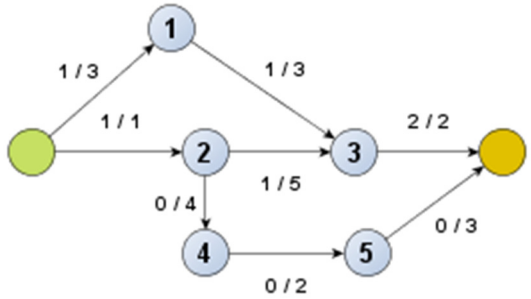


Transpordivõrk. Roheline tipp on lähe ja kollane tipp suue. Kaartel on läbilaskevõimed.

10.6.2 Voog

Transpordivõrgu kaartele seatakse sageli vastavusse veel üks arv: kaare **voog** (i.k *flow*). Läbilaskevõimest ja voost arusaamiseks on hea ette kujutada näiteks teedevõrgustikku. Teedel on läbilaskevõime ehk mitu autot ajaühikus seda maksimaalselt läbida saab. Samas on arusaadav, et alati nii palju autosid seal ei sõida. See, kui palju autosid tegelikult ajaühikus teelõiku läbib, ongi voog. Voog ei saa olla suurem kui läbilaskevõime.

Vool on veel selline omadus, et tippu sisenevate kaarte voogude summa on võrdne sellest tipust väljuvate voogude summaga. Erandiks on lähe ja suue. Samas lähtest väljuvate kaarte voogude summa on võrdne suudmesse jõudvate voogude summaga. Ideaalses maailmas ei lähe midagi teel kaduma, kõik, mis lähtest väljub, jõuab suudmesse.

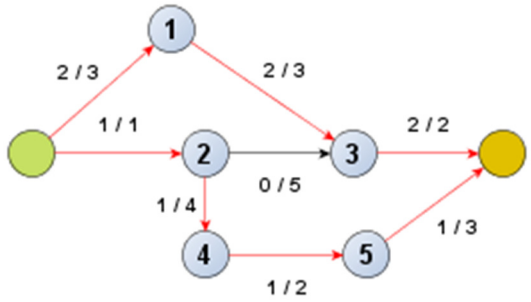


Voogudega võrk. Esimene arv kaare juures on voog ja teine läbilaskevõime.

Võime ette kujutada, et joonisel alustab teed lähtest kaks sõidukit: üks liigub lähtest suudmesse läbi tippude 1 ja 3 ning teine läbi tippude 2 ja 3. Kaart tipus 3 suudmesse läbivad mõlemad sõidukid.

10.6.3 Maksimaalne voog

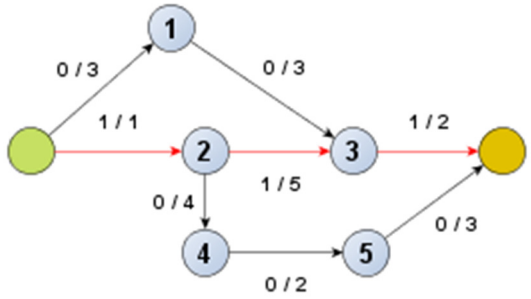
Võrkude puhul on enamasti kõige huvitavamaks küsimuseks, kui palju maksimaalselt saab midagi võrku läbides jõuda lähtest suudmesse ehk leida võrgu **maksimaalne voog.**



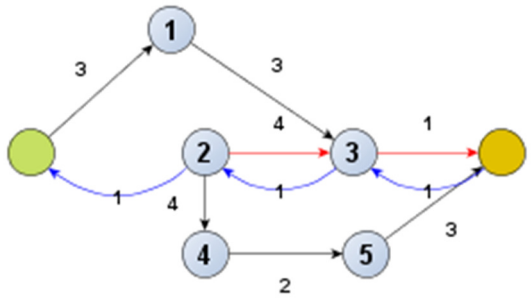
Võrgu maksimaalne voog. Maksimaalse voo väärtus on 3 – nii palju jõuab lähtest suudmesse.

10.6.4 Maksimaalse voo leidmine

Vaatame, kuidas leida maksimaalset voogu. Selleks kõigepealt leiame suvalise tee lähtest suudmesse, näiteks läbi tippude 2 ja 3. Mööda seda teed saab liikuda voog suurusega 1, kuna sel teel on serv, mille läbilaskevõime on 1



Kaarel lähtest tippu 2 on voog võrdne läbilaskevõimega ehk see kaar on küllastunud ja sel voogu enam suurendada ei saa, kuid paneme tähele, et kaartel (2,3) ja (3,S) on vood küllastamata ja neid kaari saaks kasutada võrgu voo suurendamiseks. Järgmise sammuna loome **jääkvõrgu** – samade tippude ja sarnaste servadega graafi, kus iga kaare läbilaskevõimeks on vooga graafi vastava kaare läbilaskevõime ja voo vahe. Seda vahet nimetatakse jääk-läbilaskevõimeks (i.k. *residual capacity*), lühidalt lihtsalt jäägiks. Kaared, mis on esialges graafis küllastunud (mille jääk on 0), jätame lisamata. Lisaks lisame tagasikaared nende tippude vahele, kus voog on positiivne. Nende läbilaskevõimeks on voo väärtus algses graafis:

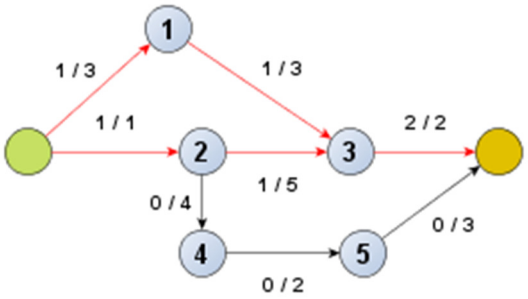


Eelmisele joonisele vastav jääkvõrk. Sinisega on tähistatud tagasikaared.

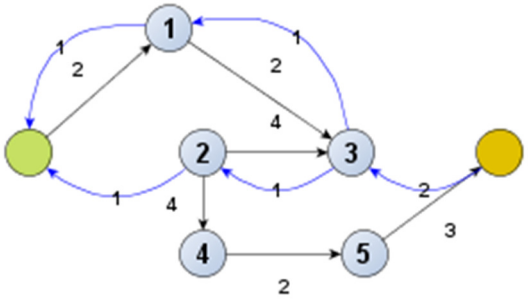
Kui vooga võrgu jääkvõrgus leidub selline tee, mis algab lähtest ja lõpeb suudmes ning mida mööda saab veel ressursse liigutada ehk iga kaare läbilaskevõime on positiivne, saame algse graafi voogu suurendada selle tee läbilaskevõime võrra. Sellist teed jääkvõrgus nimetatakse **täiendavaks teeks** (i.k. *augmenting path*).

Võrgul on leitud maksimaalne voog siis ja ainult siis, kui selle vooga võrgu jääkvõrgus ei leidu täiendavat teed.

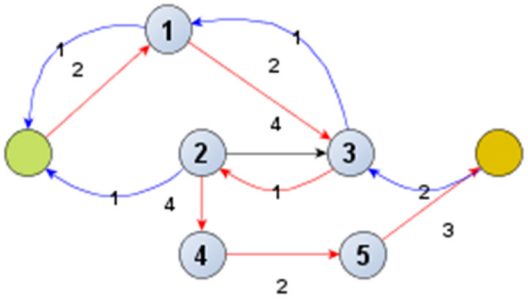
Ülaltoodud jääkvõrgus selline tee leidub, näiteks L-1-3-S. Kuna serva (3,S) läbilaskevõime on vaid 1, saame nii suurendada algset voogu ühe võrra. Selleks liidame algse graafi vastavate servade voogudele ühe juurde:



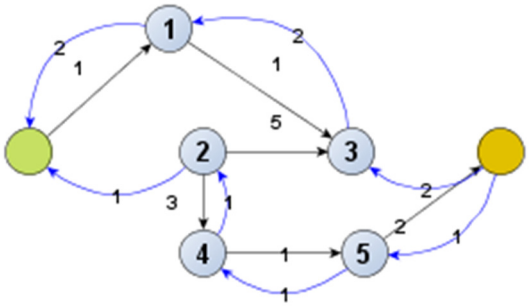
ja leiame tekkinud võrgu jääkvõrgu:



Nüüd tuleb jääkvõrgu tegelik olemus ja vajadus paremini ilmsiks, sest kui eelmisel sammul oli tee lähtest suudmesse olemas ka algsel võrgul, siis praegu enam ei leidu selles ühtki teed, mida mööda saaks voogu suurendada. Küll aga leidud tee algusest lõppu jääkvõrgus:



See tee kasutab tagasikaart (3,2). Kuna taas on minimaalse läbilaskevõimega serv selles tees väärtusega 1, saame suurendada voogu ühe võrra. Kanname selle tee algsele võrgule, pidades silmas, et tagasiviiva tee puhul tuleb voog esialgse graafi vastava serva voost liitmise asemel lahutada. Tulemuseks on eelpool toodud maksimaalse vooga võrk. Selle jääkvõrguks on



ning pole keeruline veenduda, et selles ei leidu enam ühtki teed lähtest suudmesse.

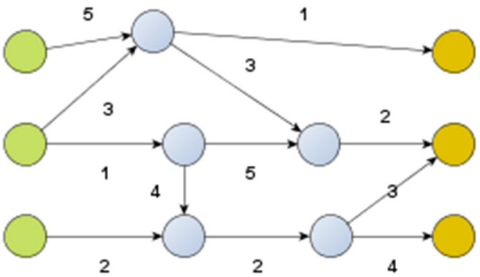
Kirjeldatud juhis ongi Ford-Fulkersoni algoritm. Mõnikord nimetatakse seda ka Ford-Fulkersoni meetodiks, kuna puudub täpne eeskiri, kuidas leida täiendavat teed jääkgraafis, viimasest aga võib sõltuda algoritmi keerukus.

Täisarvuliste läbilaskevõimete korral on selle keerukuseks $O(Ef)$, kus f on maksimaalse voo väärtus. Reaalarvuliste läbilaskevõimete korral ei pruugi antud algoritm tööd lõpetada.

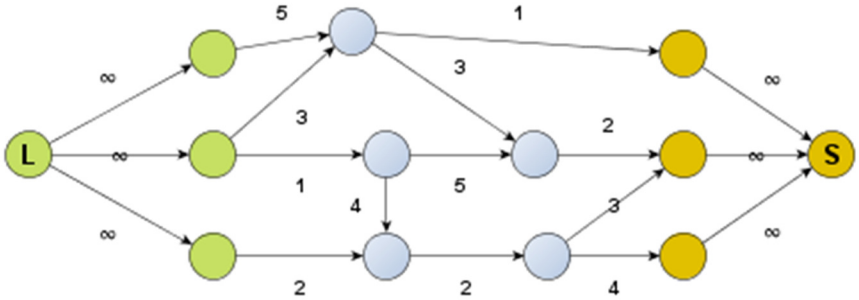
Kõige tuntum Ford-Fulkersoni meetodi implementatsioon on **Edmonds-Karpi algoritm**. See kasutab täiendava tee leidmiseks laiuti läbimist ning on keerukusega $O(VE^2)$.

10.6.5 Mitme lähte ja suudmega ülesanded

Mõnikord võib olla, et lähteülesandes on mitu lähet ja/või mitu suuet, näiteks võivad rongid väljuda mitmest depoost või kaup liikuda välja mitmest tehasest.



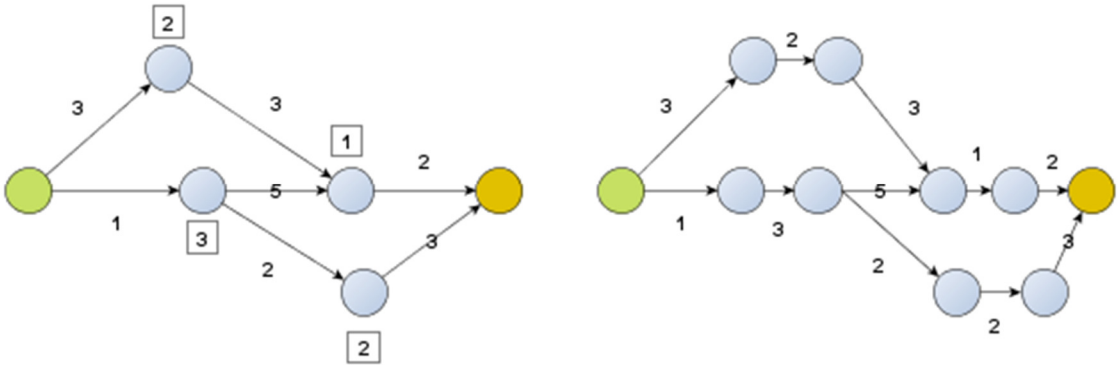
Sellisel juhul täiendatakse võrku kahe tipuga, nn superlähte ja –suudmega ∞:



10.6.6 Kaaludega tipud võrgus

Teiseks sagedaseks olukorraks on, kus ka tippudel on kaalud ehk läbilaskevõimed. Näiteks kauba vedamisel ladude vahel võib olla igal laol laadimisvõimsus, mis piirab kauba liikumist sellest laost

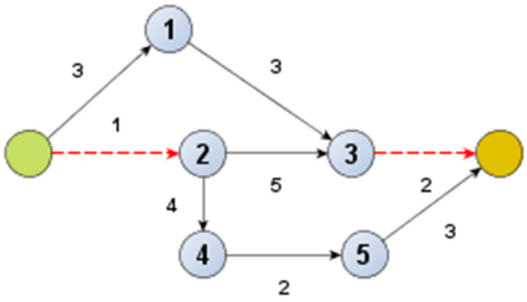
edasi. See lahendatakse nii, et tipp dubleeritakse ja sinna vahele lisatakse ekstra kaal tipu läbilaskevõimega:



10.6.7 Lõige

Vooga seotud on veel üks mõiste, nimelt **lõige** (i.k *cut*). Võrgu $G(T,S)$ lõikeks nimetatakse graafi G kaarte sellist hulka $S_L \subseteq S$, millesse kuuluvate kaarte eemaldamisel ei sisaldaks G enam ühtegi (suunatud) ahelat lähtest suudmesse. Lõike G_L läbilaskevõimeks nimetatakse tema kaarte läbilaskevõimete summat. Minimaalse võimaliku läbilaskevõimega lõikeid nimetatakse **minimaalseteks lõigeteks.**

Näiteks järgmise graafi



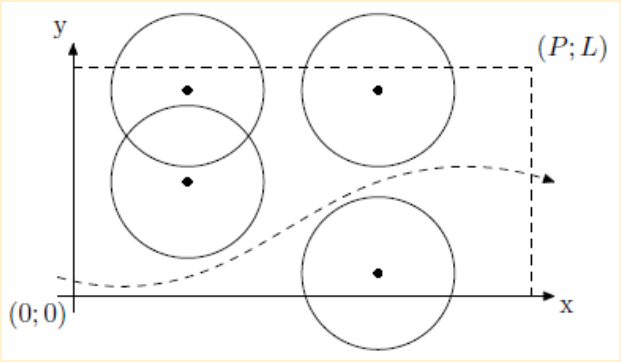
Minimaalseks lõikeks on kaared (L,2) ja (3,S) ning lõike väärtuseks 1+2=3.

Ford ja Fulkerson panid tähele, et võrgu maksimaalsete voogude väärtused on võrdsed selle võrgu minimaalsete lõigete läbilaskevõimetega. Seega saab maksimaalse voo leidmise algoritmi kasutada ka minimaalse lõike leidmiseks.

Seda kasutame ka järgmise ülesande lahendamisel. See ülesanne oli 2017. aastal Eesti Lahtisel Programmeerimisvõistlusel.

10.6.8 Vangid

Sõjavangide grupp plaanib põgenemist. Ainus tee laagrist välja viib läbi P meetri pikkuse ja L meetri laiuse kanjoni. Kanjonis on N valvurit, kes seisavad igaüks oma postil ja kelle nägemisraadius on täpselt 100 meetrit. Vahelejäämise vältimiseks tuleks läbi kanjoni hiilida nii, et kaugus lähima valvurini on alati rangelt suurem kui 100 meetrit, nagu näha alloleval joonisel.



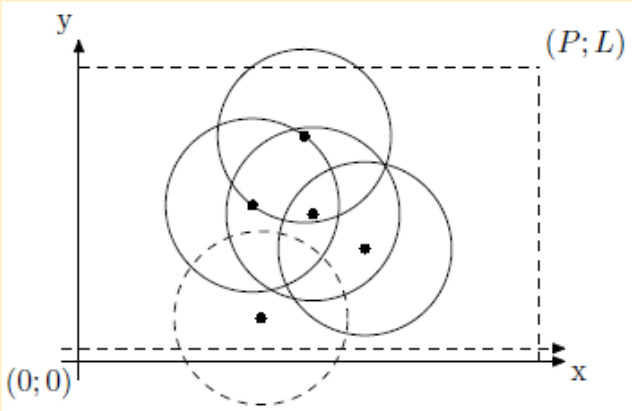
Vangid, kellel on vägivallast juba kõrini, tahavad põgenemistee vabastamiseks kõrvaldada minimaalse võimaliku arvu valvureid. Kirjutada programm, mis neile selle arvu leiab. Võib eeldada, et vangid on võimelised kõrvaldama ükskõik milliseid valvureid (isegi neid, keda mõni teine valvur näeb).
Sisendi esimesel real on kolm täisarvu P (1≤P≤50 000), L (1≤L≤50 000) ja N (1≤N≤250). Faili järgmisel N real on igaühel ühe valvuri täisarvulised koordinaadid X_i ja Y_i ($0 \le X_i \le P$, $0 \le Y_i \le L$). Kanjoni edelanurga koordinaadid on (0; 0) ja kirdenurga koordinaadid (P; L). Vangid võivad kanjonisse siseneda mistahes punktis (0; Y_s), kus $0 \le Y_s \le L$, ja väljuda mistahes punktis (P; Y_v), kus $0 \le Y_v \le L$. Seejuures ei pea Y_s ja Y_v olema täisarvud.
Väljundi ainsale reale väljastada mittenegatiivne täisarv, mis näitab vähimat võimalikku kõrvaldatavate valvurite arvu.

NÄIDE:

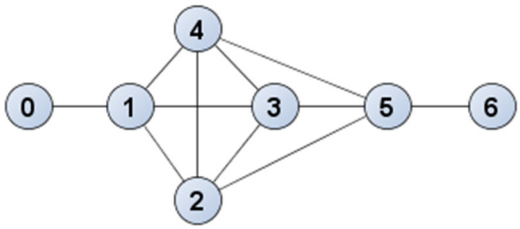
```
530 340 5
210 50
330 130
270 170
200 180
260 260
```

Vastus:

1

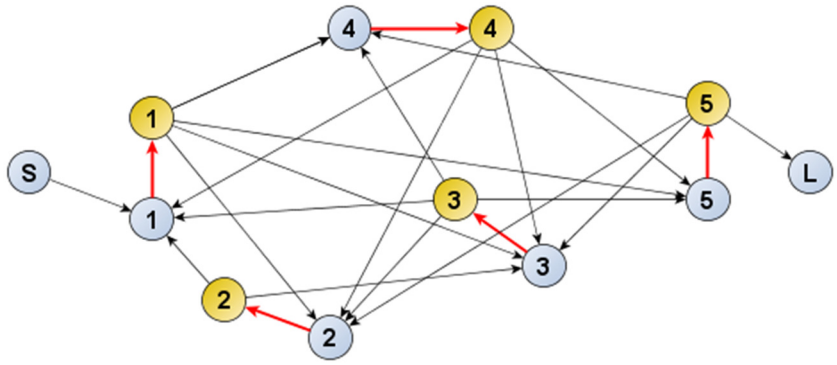


Esimeseks ülesandeks on jällegi koostada graaf. Võtame kõigepealt graafi tippudeks valvurid ja lisaks kaks tippu - kanjoni servad. Serv on kahe tipu vahel siis, kui nende valvealad puutuvad või kattuvad.



On küllaltki ilmne, et kui leidub tee tipust 0 tippu 6 ehk kanjoni seinte vahel, on põgenemine võimatu. Tuleb leida vähim arv tippe, mis tuleb graafist eemaldada, nii et tipp 0 ja tipp 6 ei ole enam ühendatud. Aga minimaalse lõike leidmiseks on tippude asemel vaja servi!

Selleks jagame iga valvurit tähistava tipu i kaheks eraldi tipuks i_s ja i_v ja lisame suunatud serva (i_s, i_v). Iga algse graafi suunamata serva (i, j) jaoks tekitame suunatud servad (j_v, i_s) ja (i_v, i_s). Loeme ühe kanjoni seina lähteks ja teise suudmeks. Kui tipp i puudutab lähteseina, lisame serva (L, i_s), kui suudmeseina, siis analoogselt serva (i_v, S), kus L ja S on vastavalt lähe ja suue. Saame:



Ülesande näiteandmetele vastav võrk.

Kuna küsitakse, mitu valvurit on vaja minimaalselt eemaldada, on iga valvuri eemaldamise hind sama ehk üks. Seega graafil punasega märgitud servadel on läbilaskevõimeks üks, teiste servade läbilaskevõime on piiramata – neid servi me nagunii eemaldada ei soovi. Samas võib arvestada ka iga serva läbilaskevõimeks ühe, sest graafi ülesehitus on selline, et rohkem nagunii ühestki tipupaarist edasi ei pääse.

Edasi leiamse sellel graafil maksimaalse voo, mis teoreemile vastavalt on ka minimaalseks lõikeks selles graafis ehk minimaalseks valvurite arvuks, keda on vaja vangidel kõrvaldada.

Siin on kood, mis ülesande lahendab:

```
#include <iostream>
#include <vector>
using namespace std;

int p, l, n;
int** kaarte_maht; // jääk-läbilaskevõimed
bool* vaadatud; // kas tippu on juba vaadatud
vector<int>* naabrid;  // tippude naabrid
int* naabrite_arv; // hoiame meeles iga tipu jaoks naabrite arvu

bool dfs(int tipp) // sügavuti läbimine
{
    if (tipp == n - 1) { // jõudsime seinani e suudmesse
        return 1; // maksimaalne voog leitud teel on alati 1
    }
    vaadatud[tipp] = 1; // siin me oleme
    for (int i = 0; i < naabrite_arv[tipp]; i++) {
        int naaber = naabrid[tipp][i];
        // kui naabertippu saab minna (läbilaskevõime on positiivne)
        // ja me ei ole seal veel olnud, otsime teed naabrist edasi
        if (kaarte_maht[tipp][naaber] > 0 && !vaadatud[naaber] && dfs(naaber)) {
            // kui see tee viib lõppu
            kaarte_maht[tipp][naabrid[tipp][i]]--; // kaare läbilaskevõime
            kaarte_maht[naabrid[tipp][i]][tipp]++; // tagasikaare läbilaskevõime
            return 1;
        }
    }
    return 0; // teed ei leidunud
}
```

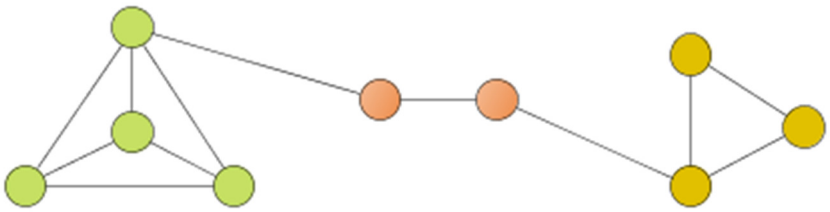
```
int main()
{
    cin >> p >> l >> n;
    int* x = new int[n];
    int* y = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }
    int valvureid = n;
    n *= 2; // iga valvuri jaoks on vaja 2 tippu graafis
    n += 2; // lisaks on vaja seinte jaoks tipud. need on lähe ja suue
    vaadatud = new bool[n];
    kaarte_maht = new int*[n];
    naabrid = new vector<int>[n];
    naabrite_arv = new int[n];
    for (int i = 0; i < n; i++) {
        naabrite_arv[i] = 0;
        vector<int> vi;
        naabrid[i] = vi;
        vaadatud[i] = 0;
        kaarte_maht[i] = new int[n] {0};
    }

    for (int i = 0; i < valvureid; i++) {
        if (y[i] <= 100) { // kui on alumise serva lähedal
            kaarte_maht[0][i * 2 + 1]++; // lisame serva
            naabrid[0].push_back(i * 2 + 1);
            naabrid[i * 2 + 1].push_back(0); // tagasiserv
            naabrite_arv[0]++;
            naabrite_arv[i * 2 + 1]++;
        }
        if (y[i] + 100 >= l) { // kui on ülemise serva lähedal
            kaarte_maht[i * 2 + 2][n - 1]++;
            naabrid[n - 1].push_back(i * 2 + 2);
            naabrid[i * 2 + 2].push_back(n - 1);
            naabrite_arv[n - 1]++;
            naabrite_arv[i * 2 + 2]++;
        }
        for (int j = 0; j < valvureid; j++) {
            if (((x[i]-x[j])*(x[i]-x[j]))+((y[i]-y[j])*(y[i]-y[j])) <= 40000) {
                if (j == i)
                    kaarte_maht[i * 2 + 1][i* 2 + 2]++; // kaar iseendaga
                else
                    kaarte_maht[i * 2 + 2][j * 2 + 1]++;
                naabrid[j * 2 + 1].push_back(i * 2 + 2);
                naabrid[i * 2 + 2].push_back(j * 2 + 1);
                naabrite_arv[j * 2 + 1]++;
                naabrite_arv[i * 2 + 2]++;
            }
        }
    }

    int vastus = 0;
    while (dfs(0)) {
        vastus ++;
        for (int i = 0; i < n; i++) {
            vaadatud[i] = 0; // nullime järgmiseks läbimiseks
        }
    }
    cout << vastus << endl;
}
```

10.7 KLIKID

Klikiks (i.k *clique*) nimetatakse graafi täielikku alamgraafi. Formaalsemalt on lihtgraafi G tippude alamhulk $S \subseteq V(G)$ klikk, kui iga kahe tipu $u, v \in S$ jaoks, kus $u \neq v$, leidub graafis G serv nende kahe tipu vahel. Alamhulka $S \subseteq V(G)$ nimetatakse sõltumatuks hulgaks siis, kui ühegi kahe hulka S kuuluva tipu vahel ei ole graafis G serva.



Klikke nimetatakse selle järgi, mitu tippu klikis on. Ülaloleval joonisel on näiteks 9 ühetipulist klikki (iga tipp on klikk), 12 kahetipulist klikki (iga tipupaar, mille vahel on serv), 5 kolmetipulist klikki (tekkivad kolmnurgad roheliste tippudega alas ja kollaste tippudega kolmnurk) ning üks 4-tipuline klikk (rohelised tipud).

Maksimaalseks klikiks (i.k. *maximal clique*) nimetatakse sellist klikki, millele ei saa lisada ühtki naabertippu, st sellist klikki, mis ei ole ühegi suurema kliki alamgraaf. Joonisel on viis maksimaalset klikki: rohelise tipuhulgaga klikk, kollase tipuhulgaga klikk ja kolm kahetipulist klikki, servaga rohelise ja punase tipu vahel, kahe punase tipu vahel ning punase ja kollase tipu vahel.

Mõnikord mõeldakse ka lihtsalt kliki all maksimaalset klikki.

Maksimumklikiks (i.k. *maximum clique*) nimetatakse kõige suurema tipuhulgaga klikki graafis. Joonisel oleva graafis on üks maksimumklikk – rohelise tipuhulgaga klikk.

Klikkide leidmise ülesanne on NP-keeruline.

10.7.1 Bron-Kerboschi algoritm

Bron-Kerboschi algoritm on nime saanud Hollandi teadlaste Coenraad Broni ja Joep Kerboschi järgi, kes avaldasid selle algoritmi kirjelduse aastal 1973.

Algoritmi näol on tegemist rekursiivse läbivaatusalgoritmiga, mis leiab kõik maksimaalsed klikid graafis. Täpsemalt, kui on antud kolm hulka R, P, ja X, leiab see maksimaalsed klikid, millesse kuuluvad kõik tipud hulgast R, mõned tipud hulgast P ja mitte ükski tipp hulgast X. Hulkade P ja X ühend on selliste tippude hulk, mis saavad klikki R laiendada. Kui P ja X on tühjad, on R maksimaalne klikk.

Algoritm ise on järgmine:

```
BK(R,P,X):
    kui P=∅ ja X=∅:
        R on maksimaalne klikk
    ∀v∈ P:
        BK(R∪{v}, P∩N(v), X∩N(v)))
        P = P \ {v}
        X = X ∪ {v}
```

10.7.2 Sõbrad

Keskkoolis on kõige tähtsam olla vinges sõprade grupis. Sõbrad korraldavad omavahel pidusid, kus kõik kohalolijad on omavahel sõbrad ja millest ükski õpilane, kes on kõigi kohalolijate sõber, ei puudu. Loomulikult võib iga õpilane osaleda erineva kosseisuga pidudel, kui eelnev tingimus on täidetud. Pidu aga pole päris see, kui kellelgi on mõned sõbrad peolt puudu.
Leia, mitu erineva koosseisu saab moodustada pidude tarbeks, nii et ühegi peo ühelgi osalejal ei oleks korraga rohkem kui p sõpra, kes sellel peol ei osale. Sõprus on sümmeetriline, st kui A on B sõber, siis B on ka A sõber.
Sisendi esimesel real on kaks mittenegatiivset täisarvu `n` ja `p`. Järgneval `n` real on info iga õpilase kohta alates õpilasest `i=0`. Iga rida algab täisarvuga m_i – `i.nda` õpilase väidetav sõprade arv. Selle järel on mi täisarvu `0..n-1` – tema sõprade numbrid. On teada, et õpilane ei nimeta kunagi ennast oma sõbraks.
Väljundi ainsale reale väljasta üks täisarv – ülesande tingimustele vastavate peokosseisude arv.

NÄIDE:

```
8 1
2 1 7
2 0 2
4 1 3 4 5
2 2 4
2 2 3
3 2 6 7
2 5 7
3 0 5 6
```

Vastus:

```
2
```



Selles ülesandes on võimalikud peokosseisud kõik graafis leiduvad klikid, seega on vaja kõigepealt leida kõik klikid sõprade graafis. Selleks saab kasutada eelpool kirjeldatud Bron-Kerboschi algoritmi:

```
vector<int>* graaf;
vector<vector<int>> klikid;

vector<int> yhend(vector<int> a, vector<int> b)
{
    vector<int> vastus;
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    int bi = 0;
    int ai = 0;
    while (ai < a.size()) {
        while (bi < b.size() && b[bi] < a[ai]) {
            bi++;
        }
        if (bi == b.size() || b[bi] > a[ai]) {
            vastus.push_back(a[ai]);
        }
        ai++;
    }
    return vastus;
}
```

```
vector<int> yhisosa(vector<int> a, vector<int> b)
{
    vector<int> vastus;
    sort(b.begin(), b.end());
    sort(a.begin(), a.end());
    int bi = 0;
    int ai = 0;
    while (ai < a.size()) {
        while (bi < b.size() && b[bi] < a[ai]) {
            bi++;
        }
        if (bi == b.size()) break;

        if (b[bi] == a[ai]) {
            vastus.push_back(a[ai]);
        }
        ai++;
    }
    return vastus;
}

vector<int> lisa_element(vector<int> a, int b)
{
    vector<int> vastus;
    vastus = a;
    vastus.push_back(b);
    return vastus;
}

void BronKerbosch(vector<int> r, vector<int> p, vector<int> x)
{
    if (p.size() == 0 && x.size() == 0) {
        klikid.push_back(r);
        return;
    }
    if (p.size() == 0) return;
    int rp = p[0];
    vector<int> pr = yhend(p, graaf[rp]);
    for (int i = 0; i < pr.size(); i++) {
        BronKerbosch(
          lisa_element(r, pr[i]), yhisosa(p, graaf[pr[i]]), yhisosa(x, graaf[pr[i]]));
        vector<int> dmp;
        dmp.push_back(pr[i]);
        p = yhend(p, dmp);
        x.push_back(pr[i]);
    }
}
```

Nüüd, kui kõik klikid on leitud, tuleb leida nende seast selliseid, mille ühelgi tipul ei ole rohkem kui p naabrit, mis ei kuulu antud klikki:

```
int leia_vastus()
{
    int vastus = klikid.size();
    for (int i = 0; i < klikid.size(); i++) {
        int kliki_suurus = klikid[i].size();
        for (int j = 0; j < kliki_suurus; j++) {
            int v = klikid[i][j];
            if (graaf[v].size() - kliki_suurus + 1 > q) {
               //lahutame need klikid, kus on mõnel tipul rohkem klikist väljuvaid
               //servi kui lubatud
                vastus--;
                break;
            }
        }
    }
    return vastus;
}
```

10.8 KONTROLLÜLESANDED

10.8.1 Laadimispunktid

Uus Euroopa Liidu seadus näeb ette, et iga linn, mida peab kindlasti läbima, et saada mingist teisest linnast kolmandasse linna, oleks varustatud elektriautode laadimispunktiga. Seega pöördus valitsus sinu poole, et teada saada, mis linnades peavad uue seaduse kohaselt laadimispunktid olema.
Sisendi esimesel real on arv `n` (`2≤n≤100`): võrgus olevate linnade arv. Järgmisel `n` real on väikeste ladina tähtedega linna nimi. Nendele järgneb eraldi real arv `m`: teede arv. Järgneval `m` real on kaks sõnet: tee alguslinn ning lõpplinn.
Väljundi esimesele reale kirjutada täisarv `d`: mitmes linnas peavad laadimispunktid olema. Järgmisele `d` reale kirjutada tähestiku järjekorras vastavate linnade nimed.

NÄIDE:

```
6
tallinn
tartu
narva
parnu
kuressaare
kardla
7
parnu narva
narva tallinn
parnu tallinn
tartu kardla
tallinn tartu
kuressaare tallinn
kardla kuressaare
```

Vastus:

```
1
tallinn
```



10.8.2 Internetiühendus

Kord ennemuistsetel aegadel polnud igal Eesti koolil ligipääsu internetile ning valitsus otsustas probleemi lahendada. Kuna internet oli sel ajal veel väga haruldane asi, oleks igale koolile otseühenduse kindlustamine väga kallis olnud, seega otsustati selle asemel ühendada koole omavahel. Koolil on ligipääs Internetile juhul, kui sellel on kas otseühendus või on see ühendatud kooliga, millel on ligipääs Internetile. Leida minimaalne hind, millega kindlustada igale koolile ligipääs Internetile ning mitu otseühendust on selle jaoks vaja. Kuna riigile meeldib hea välja näha, väljastada samahinnalistest plaanidest see, mis sisaldab rohkem otseühendusi.

Sisendi esimesel real on kolm tühikutega eraldatud täisarvu: `n`, `m` ja `a` ($1 \le n \le 10000$, $1 \le m \le 100000$, $0 < a \le 10000$) – koolide arv, võimalike koolide ühendamise viiside arv ning otseühenduse loomise hind. Järgmisel `m` real on info ühenduste kohta. Igal real on kolm tühikutega eraldatud täisarvu: `x`, `y` ja `c` ($1 \le x, y \le n$, $0 < c \le 10000$) – ühenduses esimene kool, teine kool ning ühenduse hind. Tuleb tähele panna, et kahe kooli ühendamiseks võib olla mitu erihinnalist plaani.

Väljundi ainsale reale väljastada kaks tühikutega eraldatud täisarvu: a ja b – projekti läbiviimiseks vajalik minimaalne hind ning selle jaoks vajalik internetiga otseühenduste arv.

NÄIDE 1:

```
4 4 100
1 2 10
4 3 12
4 1 41
2 3 23
```

Vastus:

```
145 1
```

NÄIDE 2:

```
5 3 1000
1 2 20
4 5 40
3 2 30
```

Vastus:

```
2090 2
```



10.8.3 Kiviheitemasin

Keskajal oli ainus praktiline viis ühe sõnumi ühest kohast teise saamiseks kiviheitemasina kasutamine*. Kiviheitemasin saab visata teateid igasse linna, mis on selle viskekauguse ulatuses. Konfliktide vältimiseks otsustati, et igal linnal on kiviheitemasin võimeline täpselt sama kaugele laskma ning et see kaugus on minimaalne vajalik, et kõik linnad oleksid omavahel kuidagi ühendatud (A ja B on ühendatud parajasti siis, kui A saab B-ga otse ühendust pidada või kui A saab mingi B-ga ühendatud linnaga ühendust pidada). Leida see minimaalne kaugus.
Sisendi esimesel real on täisarv `p` (`0<p≤500`) – linnade koguarv. Järgmisel `p` real on kaks tühikuga eraldatud täisarvu: x ja y (`0≤x,y≤10000`) linna x ja y koordinaat.

Väljundisse kirjutada üks reaalarv: minimaalne kaugus, mis kiviheitemasinal peab olema, et kõik linnad oleksid ühendatud.

NÄIDE:

```
4
0 100
0 300
0 600
150 750
```

Vastus:

```
300
```

*fakt!



10.8.4 Uus tee

Riigis on n (1≤n≤50) linna, mis on ühendatud omavahel m (1≤m≤1225) teega. Riigil läheb hästi ning seega otsustatakse kahe linna, millel puudub otseühendus, vahele luua otseühendus. Selle jaoks võetakse arvesse parima kasumi valemit: kasum(u,v)=Sum(pc(i,j)−cc(i,j)), kus iga linnapaari i,j jaoks on pc(i,j) lühima tee pikkus linnast i linna j, kui tee u ja v vahel puudub, ning cc(i,j) on lühima tee pikkus linnast i linna j, kui tee u ja v vahel eksisteerib. Leida, milliste kahe linna vahel on selle valemi järgi kõige parem teed ehitada. Juhul, kui selliseid maksimume on mitu, väljastada nendest teedest lühim. Juhul kui selliseid teid on ikka mitu, väljastada vähimate otspunktidega tee. Sisendi esimesel real on kaks arvu: n ja m. Järgmisel n real on kaks tühikuga eraldatud täisarvu: x ja y (-1000≤x,y≤ 1000), linna koordinaadid. Järgmisel m real on kaks tühikuga eraldatud täisarvu: a ja b (1≤a,b≤n), tee otspunktid. Väljundi ainsale reale kirjutada kaks tühikuga eraldatud täisarvu: uue tee algus- ja lõpp-punkt. Juhul, kui parim kasum ≤ 1 või pole võimalik uut teed luua, väljastada arvupaar 0, 0.

NÄIDE 1:

```
4 6
0 0
0 2
2 0
2 2
1 2
1 3
1 4
2 3
2 4
3 4
```

Vastus:

```
0 0
```

NÄIDE 2:

```
4 4
0 0
0 2
2 0
2 2
1 2
2 3
3 4
4 1
```

Vastus:

```
1 3
```

NÄIDE 3:

```
4 3
0 0
0 2
2 0
2 2
1 2
2 3
3 4
```

Vastus:

```
1 3
```



10.8.5 Mõjuvõim

Kuritegelikud organisatsioonid uurivad, kui palju neil mõjuvõimu on. Iga organisatsioon mõjutab mingit hulka teisi organisatsioone ning on ka teada, et kui organisatsioon A mõjutab organisatsiooni B ning organisatsioon B mõjutab organisatsiooni C, mõjutab organisatsioon A kaudselt organisatsiooni C, aga organisatsioon A ei mõjuta otseselt organisatsiooni C. Samuti ei ole tsükleid, nii et organisatsioon A ei mõjuta kaudselt iseennast. Me teame iga organisatsiooni kohta, mitu organisatsiooni see otseselt ja kaudselt mõjutab. Leida selle info põhjal, mis organisatsioone mõjutab iga organisatsioon otseselt.

Sisendi esimesel real on täisarv `n` ($1 \le n \le 120$), organisatsioonide arv. Järgmisel `n` real on üks sõne – organisatsiooni nimetus. Seejärel on eraldi real täisarv `m` ($1 \le m \le n$), organisatsioonide arv, mis mõjutavad mingeid teisi organisatsioone. Järgmisel `m` real on tühikutega eraldatud sõned. Esimene on mõjutava organisatsiooni nimi. Seejärel on arv `k` – mõjutatavate organisatsioonide arv. Siis on `k` sõnet – mõjutatava organisatsiooni nimi.

Väljundisse kirjutada tähestiku järjekorras `m` organisatsiooni nime, millele järgneb tühikuga eraldatuna selle organisatsiooni poolt otseselt mõjutatavate organisatsioonide arv ning nende organisatsioonide nimetused tähestiku järjekorras.

NÄIDE:

```
5
maranzano
profaci
mangano
luciano
gagliano
3
mangano 4 gagliano luciano maranzano profaci
gagliano 3 luciano maranzano profaci
luciano 1 profaci
```

Vastus:

```
gagliano 2 maranzano luciano
luciano 1 profaci
mangano 1 gagliano
```



10.8.6 Civilization

Arvutimängus „Side Meier's Civilization VI" kontrollivad mängijad riike, kellel on maavarad. Mängus on korraga n $(2 \le n \le 10)$ riiki ning m $(5 \le m \le 25)$ erinevat maavara. Igal riigil on igast maavarast mingi arv koopiat (koopiate arv võib olla ka 0). Riigid saavad omavahel vahetada maavarasid kursiga 1:1, aga enamus riikidest vahetavad ära ainult varasid, millest neil on rohkem kui üks koopia, varade vastu, millest neil on null koopiat. Juku taipas, et kui eesmärk on saada võimalikult palju erinevaid maavarasid, on vahepeal kasulik teha selliseid tehinguid, kus ta annab ära olemasoleva maavara teise maavara vastu, mida tal juba on. Aita Jukul leida, mis on maksimaalne erinevate maavarade arv, mis tal saab olla, kui kõik teised mängijad vahetavad ainult eelnevalt mainitud reeglite põhjal.
Sisendi esimesel real on kaks tühikutega eraldatud täisarvu: n ja m. (n sisaldab ka Jukut). Järgmisel n real on kirjeldus iga mängija maavarade kohta. Esimene nendest kirjeldab Juku maavarasid. Igal real on mingi arv tühikutega eraldatud täisarve, millest esimene on k_i $(0 \le k_i \le 50)$, sellele riigile kuuluvate maavarade arv ning sellele järgneb k_i arvu, riigile kuuluvate maavarade indeksid.
Väljundisse kirjutada üks täisarv: Maksimaalne võimalik erinevate maavarade arv, mida Juku võib saavutada.

NÄIDE 1:

```
2 5
6 1 1 1 1 1 1
3 1 2 2
```

Vastus:

```
1
```

NÄIDE 2:

```
3 5
4 1 2 1 1
3 2 2 2
5 1 3 4 4 3
```

Vastus:

```
3
```



10.8.7 Põgenemine

Indiana Jones on kogemata käivitanud lõksu, mille tagajärjel variseb enamus toast, kus on ka palju teisi inimesi, kokku. Tuba on mõõtmetega `N * M (1≤N,M≤30)` ning koosneb viiest erinevast ruudust:
`*` – inimene ohtlikul platvormil. Inimene tahab siit võimalikult ruttu ära saada, sest lõpuks kukub see ka kokku. Inimene saab liikuda ükskõik missuguse põhiilmakaare poole ning platvorm kukub dramaatilisuse huvides alla kohe sel hetkel, kui inimene selle pealt maha astub.
`~` – kuristik. Kui inimene satub kuristikku, kukub ta alla ning hukkub.
`.` – ohtlik platvorm. Kui inimene astub sellele peale, käitub see nagu „inimene ohtlikul platvormil“ tüüpi ruut.
`@` - vähem ohtlik platvorm. See on platvorm, mis küll kukub lõpuks alla, aga see ei taha olla dramaatiline ning kukub alla alles siis, kui aeg läbi saab. Tuleb tähele panna, et ohtlikul ning vähem ohtlikul platvormil tohib korraga peal olla maksimaalselt üks inimene.
`#` - ohutu platvorm. See platvorm ei kuku kunagi alla. Igal ohutul platvormil võib olla maksimaalselt `P (1≤P≤10)` inimest.
Sulle antakse ette toa kirjeldus. Sinu eesmärk on leida, mis on maksimaalne võimalik ellujääjate arv, kui eeldada, et aega on piisavalt palju, et vastavat strateegiat kasutada.
Sisendi esimesel real on kolm tühikutega eraldatud täisarvu arvud `N, M` ja `P`.
Järgmisel `n` real on `m` tähemärgist koosnev jada: toa kirjeldus.

NÄIDE 1:

```
3 4 2
*~~#
...@
.~.*
```

Vastus:

2

NÄIDE 2:

```
3 5 1
~~*~~
#.@.#
~~*~~
```

Vastus:

2

NÄIDE 3:

```
1 4 2
**#~
```

Vastus:

1



10.8.8 Teepuhastus

Tartu linnas on ainult üks lumepuhastusmasin. Leida minimaalne aeg, mis kulub, et see puhastaks kõik Tartu tänavad lumest ning jõuaks oma koju tagasi. Lumepuhastusmasina kiirus puhastamisel on 20km/h ning puhastatud teel sõitmine on 50km/h. Iga tänav on mõlemas suunas ühesuunaline ning alguses puhastamata olekus. Lumepuhastusmasin saab korraga ainult ühte sõidurida puhastada.
Sisendi esimesel real on kaks tühikuga eraldatud täisarvu: x ja y, lumepuhastusmasina jaama koordinaadid. Järgmisel real on tänavate arv n (1≤n≤100). Järgmisel n real on neli tühikutega eraldatud täisarvu: x_1, y_1, x_2, y_2 - tänava otspunktide koordinaadid. On teada, et jaamast saab igale tänavale sõita ainult tänavaid mööda. Iga koordinaat on antud meetrites ning on täisarv vahemikus -100000 kuni 100000.
Väljundi ainsale reale kirjutada aeg tundides ja minutites, mis on vaja, et kogu linn ära puhastada. Vastus väljastada täpsusega 1 minut.



NÄIDE:

```
0 0
3
0 0 10000 10000
5000 -10000 5000 10000
5000 10000 10000 10000
```

Vastus:

```
3:55
```

10.8.9 Doomino

Sul on n (1≤n≤1000) doominoklotsi, mida sa tahad järjest ringi panna, nii et kahel järjestikusel klotsil on ühendumiskohal sama number ning viimane klots on ühendatud esimesega. Leida, kas selline järjestus eksisteerib.
Sisendi esimesel real on täisarv n. Järgmisel n real on kaks tühikutega eraldatud täisarvu: a ja b (1≤a,b≤50), doominoklotsile kirjutatud arvud.
Väljundi ainsale reale väljastada üks sõna. Kui kõiki klotse on võimalik ringi panna, väljastada JAH, kui ei saa, väljastada EI.

NÄIDE 1:

```
5
1 2
2 3
3 4
4 5
5 6
```

Vastus:

```
EI
```



NÄIDE 2:

```
2 1
2 2
3 4
3 1
2 4
```

Vastus:

```
JAH
```

10.8.10 Valimised

USA-s on mängusaade, mida nimetatakse USA presidendivalimisteks, kus osaleb d demokraati ja v vabariiklast ($1 \le d, v \le 100$). Pärast saadet hääletavad n ($0 \le n \le 500$) vaatajat ühe inimese poolt, kellest nad tahavad, et jääks saatesse alles ning ühe inimese poolt, keda nad tahavad välja visata. Kuna iga ameeriklane on ise kas demokraat või vabariiklane, valivad demokraadid alati demokraadi, kes saatesse peaks jääma, ja vabariiklase, keda välja visata, ning vabariiklased alati vabariiklase, kes saatesse peaks jääma, ja demokraadi, keda välja visata. Saatejuhid otsustasid inimesi välja visata nii, et oleks võimalikult palju hääletajaid, kelle mõlemad hääled lähevad arvesse – see tähendab, et nende lemmik jääb saatesse ja vihatu lahkub. Leida see hääletajate arv.
Sisendi esimesel real on kolm tühikutega eraldatud täisarvu: d, v ja n. Järgmisel n real on kaks tühikuga eraldatud sõnet. Esimene on osaleja kood, keda vaataja tahab alles jätta ning teine on kandidaat, keda vaataja tahab välja visata. Osaleja kood koosneb tähest D või V – demokraat või vabariiklane ning ühest arvust, mis näitab, mitmenda demokraadi või vabariiklasega tegu on.
Väljundisse kirjutada 1 täisarv: maksimaalne arv rahulolevaid hääletajaid.

NÄIDE 1:

Sisend:

```
1 1 2
D1 V1
V1 D1
```

Vastus:

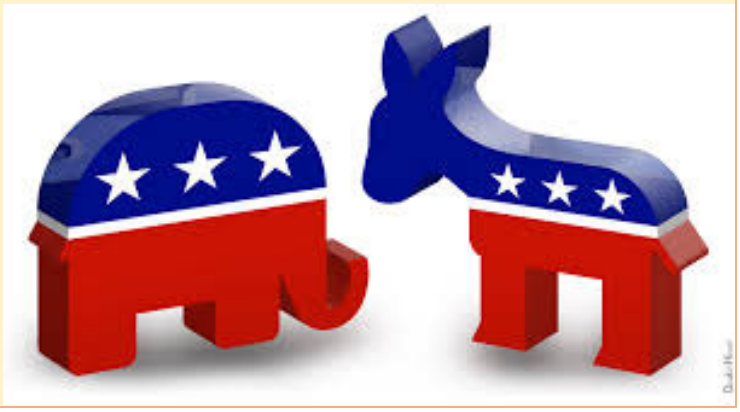
```
1
```

NÄIDE 2:

```
1 2 4
D1 V1
D1 V1
D1 V2
V2 D1
```

Vastus:

```
3
```



10.9 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk9 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
Tuleb hiljem	Floyd-Warshall
Servad.cpp, Servad.java, Servad.py	Graafi servade liigitamine
Sillad.cpp, Sillad.java, Sillad.py	Sildade leidmine graafis
Muinastuled.cpp, Muinastuled.java, Muinastuled.py	Artikulatsioonipunktide leidmine
Loomad.cpp, Loomad.java, Loomad.py	Euleri tsükkel (Hierholzer)
Elektrikatkestus.cpp, Elektrikatkestus.java, Elektrikatkestus.py	Minimaalne toes (Kruskal)
Vangid.cpp, Vangid.java, Vangid.py	Maksimaalne voog/minimaalne lõige
Sobrad.cpp, Sobrad.java, Sobrad.py	Klikid graafis

11 TEKSTIALGORITMID

Enamik seni käsitletud algoritme töötavad arvulistel andmetüüpidel, mis on oma olemuselt „lihtsad“: üks väärtus vastab üldjuhul ühele mälupesale ning protsessori käsud on optimeeritud nendega operatsioonide sooritamisele. Inimesed eelistavad aga enamasti tekstiga töötamist arvudele, seetõttu ei pääse ka arvutid teksti käsitlemise kohustusest.

Arvutustehnika levides ja abstraktsioonitaseme tõustes on väga levinud ka mitmesuguste andmete esitamine „inimloetaval“ kujul, näiteks XML või JSON formaadis, sealhulgas ka juhul, kui samade andmete kahendesituses käitlemine oleks tuhandeid kordi kiirem. See tähendab, et kaasaegne arvuti kulutab enamiku oma protsessoriajast just nimelt tekstitüüpi andmetega tegeledes.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="points">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="point">
          <xs:complexType>
            <xs:attribute name="x" type="xs:unsignedShort" use="required" />
            <xs:attribute name="y" type="xs:unsignedShort" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Kui arvudel on tavalisteks operatsioonideks liitmine, korrutamine, bitinihutamine ja muud aritmeetilised operatsioonid, siis teksti puhul on põhikohal erinevate tekstijuppide võrdlemine, kokkupanek või siis ühest tekstilõigust teise otsimine. Et aga neid operatsioone võimalikult efektiivselt sooritada, muutub väga tähtsaks see, kuidas me teksti arvuti mälus hoiame. Kui arvu liigutamine ühelt mäluaadressilt teisele võtab protsessoril vaid nanosekundi, siis pika teksti kopeerimine on palju pikem protsess.

Enamiku siin peatükis toodud algoritmide peamine idee seisnebki selles, kuidas selliseid liigutamisi ja kopeerimisi võimalikult palju vältida.

11.1 TEKSTI PARSIMINE

Enamik tekstist, millega inimesed tegelevad, on „proosatekstid“, mis koosnevad sõnadest ja lausetest, kuid millel pole kuigi palju täiendavaid reegleid. Peale selle on olemas aga väga palju tekstivormis andmeid, millel on keerulisem struktuur: programmeerimiskeeltes kirjutatud kood, algebralised avaldised, HTML-tekst jne.

$$\frac{\frac{1}{3a}-\frac{1}{3b}}{\frac{a}{b}-\frac{b}{a}} = \frac{\left(\frac{b-a}{3ab}\right)}{\left(\frac{a^2-b^2}{ab}\right)}$$

$$= \frac{b-a}{3ab}\cdot\frac{ab}{a^2-b^2} = \frac{(b-a)}{3ab}\cdot\frac{ab}{(a+b)(a-b)}$$

$$= \frac{(b-a)^{-1}}{3ab}\cdot\frac{ab}{(a+b)(a-b)} = -\frac{1}{3(a+b)}$$

Inimene näeb sellise teksti struktuuri tervikuna, kuid arvuti jaoks on tegu märgijadaga nagu iga teine. Et infoga midagi intelligentset teha, tuleb tekst teisendada mingiks andmestruktuuriks (näiteks puuks), millel saab juba täiendavaid operatsioone läbi viia. Sellist teksti paremini käsitletavale kujule viimist nimetatakse **parsimiseks**.

Vaatame järgmist klassikalist ülesannet:

11.1.1 Ülesanne: arvutustehe

On antud aritmeetiline avaldis, mis sisaldab positiivseid täisarve, liitmis-, lahutamis, korrutamis- ja jagamismärki (`+`, `-`, `*`, `:`) ning sulge (`(`, `)`). Leia tehte vastus.
NÄIDE:
`4*(5+3)-(2*6+3):15+11`
Vastus:
42

Tavapäraselt matemaatikas kasutatav tehtekuju, kus tehtemärk ehk operaator on arvude (argumentide) vahel, nimetatakse **infiks**-kujuks.



Infiks-kujul tuleb tehteid sooritada üsna suvalises järjekorras, näiteks siin ei saa esimest tehet (korrutamine) enne sooritada, kui oleme teinud sulgudes oleva liitmistehte. Sellele järgnev lahutamistehe jääb aga juba järgneva kolme tehte ootele.

Seetõttu on antud ülesandes mugavam viia tehe **postfiks**-kujule. Postfiks-kuju ehk pööratud Poola notatsioon on selline tehete esitus, kus operaator järgneb argumentidele. Näiteks tavapärane liitmistehe $5 + 6$ on postfiks-kujul $5\ 6\ +$. Näiteavaldisest esimene osa $4 * (5 + 3)$ saab kuju $5\ 3\ +\ 4\ *$ ning kogu tehe on postfiks-kujul järgmine:

$$4\ 5\ 3\ +\ *\ 2\ 6\ *\ 3\ +\ 15\ :\ -\ 11\ +$$

Üheks postfiks-kuju eeliseks on see, et puudub vajadus sulgude järele ning tehetel ei ole erinevat kaalu – nii võib „unustada“ algklassides õpitud reegli: „Enne korrutan ja jagan, siis liidan ja lahutan“.

Kuidas siis teisendada avaldis postfiks-kujule ja leida selle väärtus? Selleks on siiski vaja teada just eelpool nimetatud reeglit. Siin on väike abifunktsioon, mis tehetele prioriteedid annab:

```
int pref(char ch)
{
     switch (ch) {
          case '*':
          case ':': return 1;
          case '+':
          case '-': return 2;
          case ('('): return 3;
          default: return 0;
     }
}
```

Kui vaadata näidet, on näha, et arvud on tehtes ikka samas järjekorras, vaid operaatorid ja nende omavaheline järjestus muutub. Seega arvud saab kohe nö ühest stringist teise sõidutada, tehtemärgid tuleb aga vahepeal kõrvalteele (pinusse) ootele panna, et need siis õigel ajal arvude sekka tagasi lükata:

```
string postfix(string s) {
	string vastus = "";
	stack<char> op;

	int index = 0;
	while (index < s.length()) {
		while (index < s.length() && isdigit(s[index])) {
			vastus += s[index];
			index++;
		}
		vastus += ' ';

		if(index < s.length()) {
			char mark = s[index];
			if (mark == '+' || mark == '-' || mark == '*' || mark == ':') {
				while (!op.empty() && pref(op.top()) <= pref(mark)) {
					vastus += op.top();
					op.pop();
				}
				op.push(mark);
			}
			else if (mark == '(') op.push(mark);
			else if (mark == ')') {
				while (op.top() != '(') {
					vastus += op.top();
					op.pop();
				}
				op.pop();
			}
		}
		index++;
	}

	while (!op.empty()) {
		vastus += op.top();
		op.pop();
	}
	return vastus;
}
```

Postfiks-kujul avaldist on lihtne arvutada, kasutades taas pinu:

```
int arvuta(string s) {
	stack<int> vastus;
	int index = 0;
	while (index < s.length()) {
		if (isdigit(s[index])) {
			int d = s[index++] - '0';
			while (index < s.length() && isdigit(s[index])) {
				d = d * 10 + s[index++] - '0';
			}
			vastus.push(d);
		}
		if (s[index] == ' ') index++;
		else {
			int d2 = vastus.top();
			vastus.pop();
			int d1 = vastus.top();
			vastus.pop();
			switch (s[index++]) {
			case '+': vastus.push(d1 + d2); break;
			case '-': vastus.push(d1 - d2); break;
			case '*': vastus.push(d1 * d2); break;
			case ':': vastus.push(d1 / d2); break;
			}
		}
	}
	return vastus.top();
}
```

11.2 ALAMSTRINGI LEIDMINE

Nagu sissejuhatuses öeldud, on tekstitöötlusalgoritmide üheks peamiseks ülesandeks etteantud stringist mingi teise, samuti etteantud alamstringi leidmine.

11.2.1 Mõisted

Tuletame kõigepealt meelde põhimõisted. **String** on tervikuna käsitletav samalaadsete elementide, näiteks märkide, järjend. **Märk** on andmete esituseks, korralduseks või juhtimiseks kasutatava elemendihulga liige. **Märgistik** on määratud otstarbeks täielik erinevate märkide lõplik hulk.

Enamasti tähistame stringi $S = s_1 s_2 \dots s_n$, kus s_i on märk kasutatavas märgistikus M ehk $\forall s_i \in M$.

Stringi $S = s_1 s_2 \dots s_n$ **alamstringiks** nimetame sellist järjendit $A = a_1 a_2 \dots a_m$, kus $m \leq n$ ja leidub selline indeks i, et $\forall a_j \in A, a_j = s_{i+j}$ ja $s_{i+j} \in S$. Näiteks stringi `pesukaru` alamstringiks on string `suka`, aga mitte string `suru`.

Stringi S **sufiksiks** nimetatakse sellist stringi S alamstringi, mis lõpeb stringi S lõpus. Näiteks stringi `pesukaru` sufiksid on `pesukaru`, `esukaru`, `sukaru`, `ukaru`, `karu`, `aru`, `ru` ja `u`. Stringil pikkusega n on täpselt n sufiksit.

Stringi S **prefiksiks** nimetatakse sellist stringi S alamstringi, mis algab stringi S algusest. Näiteks stringi `pesukaru` prefiksid on `p`, `pe`, `pes`, `pesu`, `pesuk`, `pesuka`, `pesukar` ja `pesukaru`. Nii nagu sufikseidki, on stringil pikkusega n täpselt n prefiksit.

11.2.2 Lihtne alamstringi otsimise algoritm

On antud kaks stringi T ja S. Leia kõik stringi S esinemised stringis T.
Sisendi esimesel real on string T. Teisel real on string S.
Väljastada ühel real tühikutega eraldatuna kõik asukohad tekstis T, millest algab alamstring S. Kui string S ei sisaldu stringis T, väljastada -1.
NÄIDE 1:
`varvas`
`va`
Vastus:
`1 4`
NÄIDE 2:
`varvas`
`aas`
Vastus:
`-1`

Kõige vahetum lähenemine on muidugi hakata stringe märk-märgilt omavahel võrdlema: kui esimesed märgid langevad kokku, võrdleme teisi jne, kuni alamstring saab võrreldud (ja on leitud!) või kuni märgid ei lange kokku, millisel juhul otsime edasi kõrvutades alamstringi esimest märki stringi teise märgiga jne.

Siin on funktsioon, mis seda teeb:

```
string tekst, sona;

void lihtne_alamstring() {
    for (int i = 0; i <= tekst.length() - sona.length(); i++) {
        for (int j = 0; j < sona.length(); j++) {
            if (tekst[i + j] == sona[j]) {
                if (j == sona.length() - 1) {
                    cout << i + 1 << " ";
                };
            }
            else break;
        }
    }
}
```

Huvitav on see, et suvalise teksti korral on selline lähenemine päris hea ja keskmise keerukusega $O(n)$, kus n on stringi ehk teksti pikkus. Suvalises juhuslikus tekstis on meil enamasti kasutusel ca 200 erinevat tähemärki ja tõenäosus, et esimene võrreldav märk kokku langeb, on vaid $\frac{1}{200}$. Tõenäosus, et kaks esimest kokku langevad, aga juba vaid $\frac{1}{200*200}$ jne – seega tekib üliharva olukord, kus tuleb mitmeid märke võrrelda otsustamaks, et kokkulangevust tegelikult ei ole.

Samas isegi tavalised kirjatekstid ei ole täiesti juhuslikud. Esiteks ei esine kõik märgid sama sagedusega, teiseks on ka tähejärjenditel oma esinemissagedused. Näiteks, kui otsida siit raamatust alamstringi `ktsea`, siis olenemata sellest, et kõik kasutatud märgid on päris sagedased, ei tule ilmselt

teha väga palju võrdlusi, kuna ei leidu eriti sõnu, mis algaks täheühendiga `kt`. Samas, kui otsida stringi `arvustus`, siis tuleb teha märkimisväärselt võrdlusi, mis tulemust ei anna, kuna ilmselt on siin raamatus küllalt sõnu `arv`, `arvu`, `arvus` ja `arvust`.

Ka siis kasvab prefiksite kokkulangevuse osakaal, kui kasutusel on väiksem märgistik. Näiteks, kui otsime telefoninumbrite seast mingit konkreetset numbrit, siis on küllaltki suur tõenäosus, et otsitava numbri algus langeb kokku päris mitmete teiste telefoninumbrite osadega. Eriti muidugi algustega, kuna selleski osas ei ole tegemist täiesti suvaliste arvujärjenditega.

Võid kindel olla, et programmeerimisvõistlustel on testid nii koostatud, et tekstid ei ole sugugi juhuslikud ja sellisel juhul läheneb keerukus halvimale juhule ehk $O(mn)$, kus m on otsitava alamstringi pikkus ja n teskti pikkus.

11.3 Knuth-Morris-Pratti algoritm

1970. aastatel mõtlesid Stanfordi ülikooli professor (ja üldse üks kuulsamaid arvutiteadlasi) Donald Knuth ja tema õpilane Vaughan Pratt välja algoritmi, kuidas tekstist efektiivsemalt alamstringi leida. Enam-vähem samal ajal jõudis sarnase algoritmini ka kolmas mees, James H. Morris Carnegie-Melloni ülikoolist. Kolmekesi avaldasid nad vastavasisulise töö aastal 1977. Sellest ajast tuntakse seda kui Knuth-Morris-Pratti algoritmi (lühendatult tihti lihtsalt KMP-algoritm).



Algoritmi peamiseks mõtteks on see, et juba vaadatud teksti ei peaks uuesti üle vaatama. Oletame, et meil on tekst `pannkook kohupiimaga kook koorega pannkook kodujuustuga` ja otsime sellest alamstringi `kook kodu`. Tähistame parasjagu vaadeldava asukoha tekstis i-ga ja parasjagu vaadeldava asukoha alamstringis j-ga

Alguses ei erine algoritm sugugi eelnevalt tutvustatud lihtsast algoritmist, sest esimene täht juba kokku ei lange ja sellest oluliselt rohkem infot saada ei ole. Seisuks on `i=0`, `j=0` ja `s[i]≠a[j]`:

```
          1         2         3         4         5
01234567890123456789012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
kook kodu
```

Järgmiseks suurendame `i-d` ühe võrra ja `j` jääb samaks:

```
          1         2         3         4         5
01234567890123456789012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
 kook kodu
```

Kuna ka siin kokkulangevust pole, jätkame samamoodi, kuni `i=4`. Kuna `s[4]=a[0]`, siis suurendame edasi nii i-d kui ka j-i, kuni enam kokkulangevust pole. See juhtub kohal `j=7`:

```
          1         2         3         4         5
01234567890123456789012345678901234567890123456789012345678901234567890
```

```
pannkook kohupiimaga kook koorega kook kodujuustuga
    kook kodu
```

Nüüd lihtne algoritm alustaks taas kohast `i=5` ning `j=0`. KMP aga jätkab kohast `i=11 ja j=2`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
         kook kodu
```

Kuna siingi kokkulangevust pole, siis edasi on veel ükshaaval minekut kuni `i=21`, millele järgnevad taas kokkulangevused, kuni `i=28` ja `j=7`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                     kook kodu
```

Nüüd saame taas jätkata kohalt `i=28` ja `j=3` ja kuigi `s[28]=j[3]`, siis juba `s[29]≠a[4]`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                           kook kodu
```

Kuna ka siin kokkulangevust pole, liigume alates `i=29` ja `j=0` ükshaaval kuni `i=34`:

```
          1         2         3         4         5
012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
                                  kook kodu
```

Sellest näitest on ilusti näha, kuidas tekstis tagasi liikumist ei toimu, kuigi võrdusi võib sama positsiooni korral olla vaja teha rohkem kui ühe korra. Pigem aga on küsimus selles, et kuidas me teame, kui palju tuleb otsitavat alamstringi nihutada, et arvestaksime esiosa ehk prefiksi kokkulangemisega?

11.3.1 Kokkulangevuste tabel

Kokkulangevustega arvestamiseks kasutatakse osaliste kokkulangevuste tabelit. Selleks töödeldakse enne otsitavat stringi.

i	1	2	3	4	5	6	7	8	9
a[i]	K	o	o	k	’ ’	K	o	d	u
nihe	0	1	1	0	1	0	1	3	1

Vaatame näitena juba esimesest peatükist tuttavat ülesannet Eesti 2016. aasta informaatikaolümpiaadi eelvoorult.

11.3.2 Hulknurgad

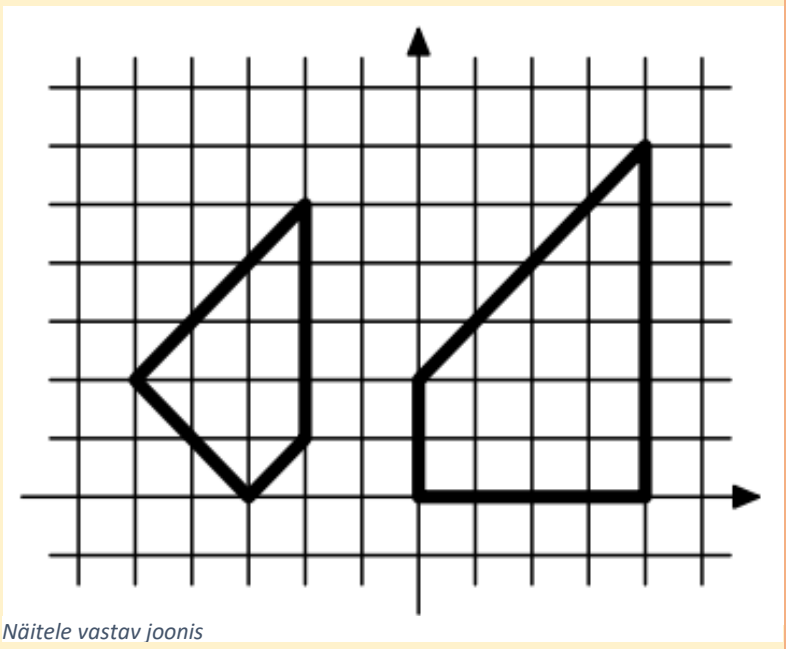
Juku õpib koolis hulknurkade sarnasust ja saab teada, et hulknurgad on sarnased, kui nende vastavate nurkade suurused on võrdsed ja vastavate külgede pikkused võrdelised. Sarnased hulknurgad võivad olla omavahel pööratud, peegeldatud ja nihutatud. Sarnaste hulknurkade vastavate külgede pikkuste jagatist nimetatakse nende sarnasusteguriks.
Kodutööna saab ta hulga hulknurki, mille sarnasustegureid on vaja määrata. Jukul on fanaatiline matemaatikaõpetaja, kes andis tööna väga paljude nurkadega hulknurki. Aita Juku hädast välja.
Sisend. Tekstifaili esimesel real on hulknurga tippude arv N (3<=N<=200 000).
Faili teisel real on 2 · N täisarvu lõigust −10^9 kuni 10^9: esimese hulknurga tippude x- ja y-koordinaadid. Kolmandal real on samuti 2·N arvu: teise hulknurga tippude koordinaadid. Tipud võivad olla antud nii päripäeva kui vastupäeva järjekorras. Antud punktid moodustavad alati hulknurga, milles pole ühtelangevaid punkte, sirgnurki, ega endaga lõikumisi.
Väljund. Kui hulknurgad on sarnased, siis kirjutada väljundi esimesele reale täpselt üks reaalarv, mis näitab, mitu korda on esimene hulknurk suurem kui teine (kui esimene hulknurk on väiksem, on ka vastus väiksem kui 1).
Teisele reale kirjutada täisarv, mis näitab, mitmes teise hulknurga tipp vastab esimese hulknurga esimesele tipule (mõlema hulknurga tipud on nummerdatud alates ühest nende failis esitamise järjekorras).
Kui hulknurgad ei ole sarnased, kirjutada väljundi ainsale reale −1.

NÄIDE:

```
4
0 0 4 0 4 6 0 2
-2 5 -2 1 -3 0 -5 2
```

Vastus:

```
1.414213
3
```



Näitele vastav joonis

Vastuse teine osa on 3, sest teise hulknurga kolmas punkt (−3; 0) vastab esimese hulknurga esimesele punktile (0; 0).

Sellel ülesandel oli kaks poolt: geomeetriline ja tekstitöötluse pool. Kui esimest käsitlesime raamatu alguses, siis nüüd uurime teksti osa.

Kui mõelda küljepikkuse ja nurgasuuruse kombinatsioonist kui tähemärgist, siis taandub ülesanne kontrollile, kas kaks stringi on saadud üksteise esimese ja tagumise jupi äravahetamise teel. Näiteks stringid CABAABBA ja BBACABAA on selles mõttes ekvivalentsed.

Üks võimalus sellise ekvivalentsuse kontrollimiseks oleks proovida stringi kõiki rotatsioone. Kavalam on aga tähele panna, et üks string on teise rotatsioon juhul, kui ta on teise stringi kahekordse koopia alamstring. Seega eeltoodud näites CABAABBA ja BBACABAA ekvivalentsus järeldub sellest, et CABAABBA on BBA**CABAABBA**CABAA alamstring (ja ka BBACABAA on CABAA**BBACABAA**BBA alamstring).

Nüüd aga on juba lihtne näha, et ekvivalentsuse kontroll taandub lihtsalt efektiivsele alamstringi leidmisele. Kasutame seekord Knuth-Morris-Pratti algoritmi.

Kõigepealt funktsioon, mis arvutab kokkulangevuste tabeli:

```
Segment** pat;
Segment** str;
int* lps;
int m;
int n;

void ComputeLpsArray()
{
     int len = 0;
     int i = 1;

     lps[0] = 0;

     while (i < m) {
          if (pat[i]->Equals(pat[len])) {
               len++;
               lps[i] = len;
               i++;
          }
          else {
               if (len != 0) {
                    len = lps[len - 1];
               }
               else {
                    lps[i] = 0;
                    i++;
               }
          }
     }
}
```

Stringide sobimist kontrolliv osa:

```
int KmpMatch() {
      int i = 0;
      int j = 0;
      lps = new int[m];

      ComputeLpsArray();

      while (i < 2 * n) {
            if (pat[j]->Equals(str[i % n])) {
                  j++;
                  i++;
            }

            if (j == m) {
                  int vas = i - j + 1;
                  return vas;
            }
            else if (i < 2 * n && !pat[j]->Equals(str[i % n])) {
                  if (j != 0)
                        j = lps[j - 1];
                  else
                        i++;
            }
      }

      return -1;
}
```

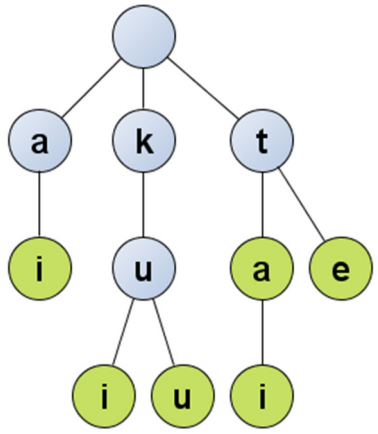
11.4 SUFIKSIPUUD

Mitmesugustes tekstiotsingualgoritmides on kasulik organiseerida stringid nii, et nende kattuvad osad on arvuti mälus samas kohas. Loomulik struktuur selliste kattuvuste kokkupanekuks on puu.

11.4.1 Prefiksipuu

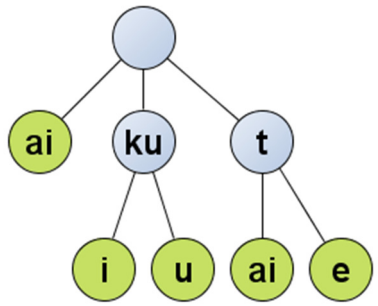
Prefikispuu (i.k *trie*, *digital tree, prefix tree*) ingliskeelne nimetus *trie* tuleb sõnast *re**trie**ve*, mis viitabki otsimisele. Tuletame meelde, et otsingupuu on puu, kus hoitakse võtme järgi väärtusi ning oluline ongi, et väärtus on võtme järgi kiiresti ja efektiivselt leitav, st võtmed on mingil moel sorteeritud. Kahendotsingupuus näiteks kehtis reegel, et tipu vasakus alampuus on selle tipu võtmest väiksemad võtmed, paremas aga suuremad võtmed. Igas tipus hoitakse üht võtit.

Juhul, kui võtmeteks on sõnad, saabki kasutada otsingupuuks prefiksipuud. Prefiksipuu vahetippudes hoitakse osa võtmest ning kogu võti moodustub läbitavates tippudes või servades oleva info põhjal. Iga tipu kõigil järglaste võtmel on sama prefiks.



Prefiksipuu. Rohelisega on tähistatud need tipud, milles on ka väärtused. Võtmed selles puus on ai, kui, kuu, ta, tai ja te.

Kompaktne prefiksipuu (i.k *compact prefix tree*, *radix tree*) on selline prefiksipuu, kus iga alluv, mis on oma vanema ainsaks alluvaks, ühendatakse vanemaga:



Kompaktne prefiksipuu

Mõned prefiksipuu eelised paisktabeli ees:

- Halvim juht info leidmiseks on O(m) , kus m on võtme pikkus
- Kokkupõrgete ehk kollisioonide puudumine
- Saab esitada võtmed kiiresti tähestikulises järjekorras

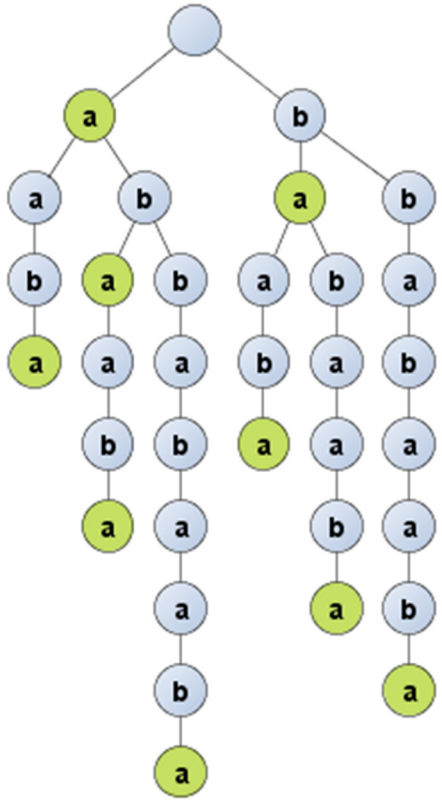
Prefiksipuid kasutatakse näiteks sõnade äraarvamisel (*autocomplete*) ja spellerites.

11.4.2 Sufiksipuu

Nimetame stringi S i-ndaks sufiksiks sellist stringi s sufiksit, mis algab positsioonilt i. Näiteks stringi S=’abbabaaba’ sufiksid on:

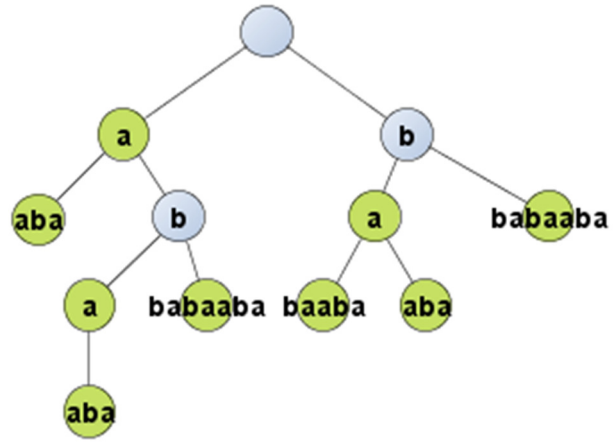
I	1	2	3	4	5	6	7	8	9
i-s sufiks	abbabaaba	bbabaaba	babaaba	abaaba	baaba	aaba	aba	ba	a

Sufiksipuuks (i.k *suffix trie*) nimetatakse stringi sufiksitest moodustatud prefiksipuud.



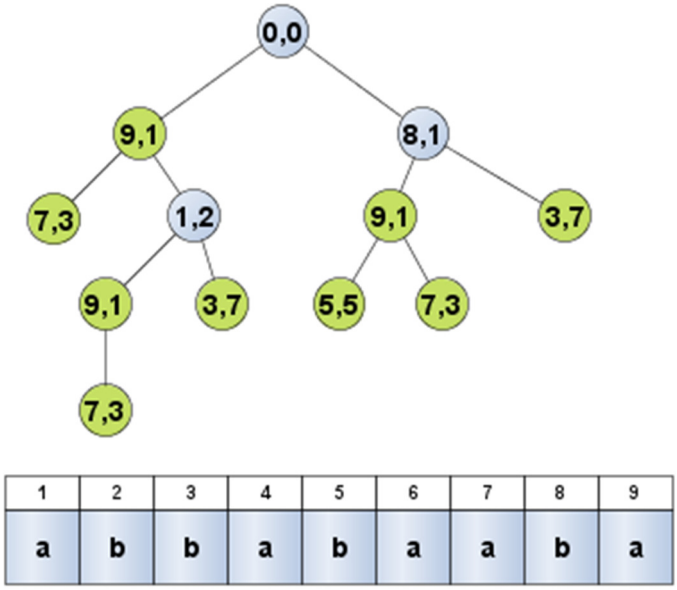
Stringi 'abbabaaba' sufiksipuu

Samuti nagu prefiksipuust sai teha kompaktse prefiksipuu, võime esitada ka sufiksipuu kompaktsel kujul:



Stringi 'abbabaaba' kompaktne sufiksipuu

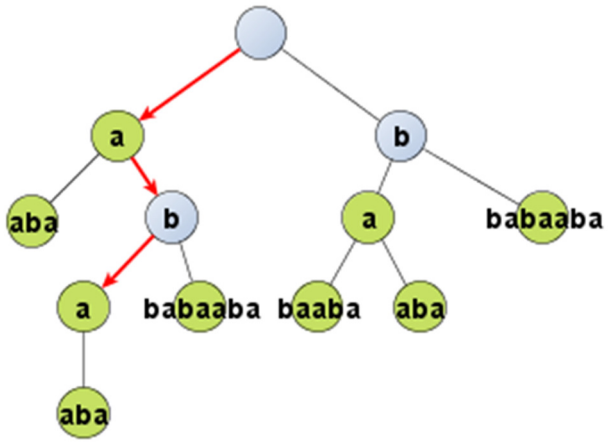
Veelgi enam: me võime puus hoida ainult algusindeksi ja pikkuse:



11.4.3 Sufiksipuu kasutamine

11.4.3.1 Alamstringide leidmine

Kui alamstring A leidub tekstis, siis see on mingi selle teksti sufiksi prefiksiks. Proovime näiteks leida alamstringi ’aba’ eelpool toodud stringist ’abbabaaba’. Kui vaadata kõiki selle stringi sufikseid, siis pole raske veenduda, et ’aba’ on täpselt kahe – 4. ja 7. sufiksi prefiksiks.



Alamstringi ’aba’ leidmine sufiksipuust.

Seda teeb ka järgnev kood:

```
using namespace std;
struct Tipp {
	char* voti;
	map<char, Tipp> alamad;
};

int leia_alamstring(Tipp *tipp, char* str, int indeks) {
	if (tipp == NULL) {
		return -1; // ei leidnud
	}
	int vas = -1;
	//kui tipp t ei ole juur
	if (tipp->voti != 0)
	{
		int vastus = kontrolli(str, indeks, tipp->voti);
		if (vastus != 0)
			return vastus;
	}
	//suurenda indeksit
	indeks += strlen(tipp->voti);
	//Kui on õige algusega väljuv serv
	if (tipp->alamad.find(str[indeks]) != tipp->alamad.end())
		return leia_alamstring(&(tipp->alamad[str[indeks]]), str, indeks);
	else
		return -1;  // ei leidnud
}

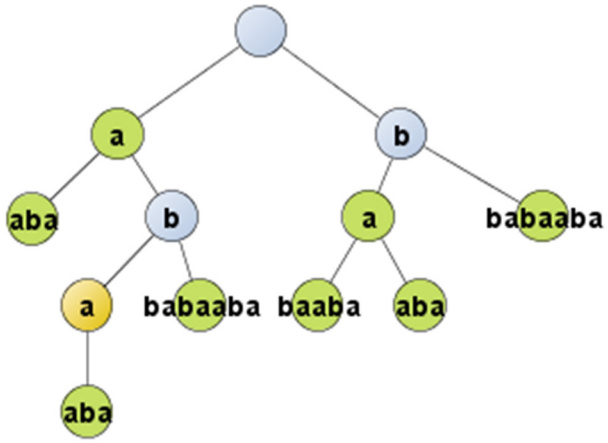

int kontrolli(char *str, int indeks, char *voti)
{
	if (str[indeks] == '\0')
		return 1;  //korras

	//kontrollime, kas on kattuvus
	for (int i = 0; i < strlen(voti); i++, indeks++)
	{
		if (voti[i] != str[indeks])
			return -1;  // ei kattu
	}
	return 0;  // siiani korras, vaatame edasi
}
```

Alamstringide leidmine on keerukusega $O(m + k)$, kus m on alamstringi pikkus ja k alamstringi esinemise arv tekstis. Pane tähele, et keerukus ei sõltu üldse teksti pikkusest!

11.4.3.2 Pikim korduv alamstring

Pikim korduv alamstring on selline alamstring, mis esineb tekstis vähemalt kaks korda. Kui meil on olemas sufiksipuu, siis tuleb leida kõige sügavamal asuv hargnemine:




```
void leia_pikim_korduv(Tipp *tipp, int sygavus, int* max_sygavus,
      int* algus)
{
      if (tipp == NULL) {
            return;
      }
      if (tipp->voti == '\0') { //sisemine tipp
            for (int i = 0; i < MAX_CHAR; i++) {
                  if (tipp->alamad.find(str[indeks]) != tipp->alamad.end()) {
                        otsi_pikim_korduv(tipp->alamad[i], sygavus +
                              strlen(tipp->voti), max_sygavus,
                              algus);
                  }
            }
      }
      else if (tipp->voti > 0 &&
            (*max_sygavus < sygavus - strlen(tipp->voti)))
      {
            *max_sygavus = sygavus - strlen(tipp->voti);
            *algus = tipp;
      }
}

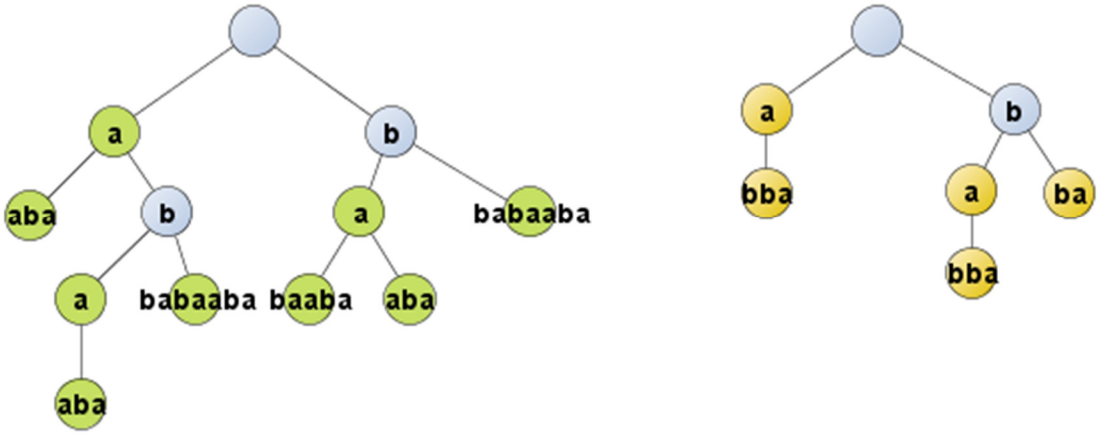
void otsi_pikim_korduv()
{
      int max_sygavus = 0;
      int indeks = 0;
      leia_pikim_korduv(puu, 0, &max_sygavus, &indeks);

      int k;
      for (k = 0; k < max_sygavus; k++)
            cout << tekst[k + indeks];
      cout << endl;
}
```

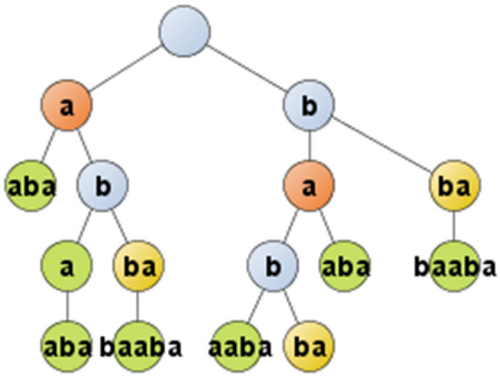
Selle lähenemise keerukus on $O(n)$.

11.4.3.3 Pikim ühine alamstring

Ehitame mõlema stringi ühise sufiksipuu. Ainsaks nipiks on, et tuleb eristada kuidagi, mis sufiksid millisesse puusse kuuluvad. Kõige sügavamal asuv hargnemine, mis on prefiksiks mõlemale (või kõigile) stringidele, ongi vastuseks:



Stringide 'abbabaaba' ja 'babba' sufiksipuud



Ja ühendatud sufiksipuu.

Pikim ühine alamstring on 'abba'.

Selle leidmise keerukus on $O(n)$.

11.4.4 Sufiksipuu loomine

Sufiksipuu on väga kasulik struktuur, kuid puu loomine ise ei ole just kõige lihtsam, kuigi see on teostatav keerukusega $O(n)$. Samas kuna võistlusel on oluline ka koodi kirjutamisele kuluva aja kokkuhoid, siis jõuame järgmise struktuuri juurde.

11.5 SUFIKSILOEND

Kuna sufiksipuu loomine ise on keeruline ning probleemiks on ka see, et puu on võrreldes tekstiga väga ruumimahukas, siis sageli kasutatakse selle asemel **sufiksiloendit** (i.k. *suffix array*).

Sufiksiloendi leiutasid Udi Manber ja Eugene Myers aastal 1990.

Vaatame taas tuttava stringi ’abbabaaba’ sufikseid:

Sufiksi number	Sufiks
1	abbabaaba
2	bbabaaba
3	babaaba
4	abaaba
5	baaba
6	aaba
7	aba
8	ba
9	a

Sorteerime need tähestiku järjekorras:

Sufiksi number	Sufiks
9	a
6	aaba
7	aba
4	abaaba
1	abbabaaba
8	ba
5	baaba
3	babaaba
2	bbabaaba

Sellist loendit nimetataksegi sufiksiloendiks.

11.5.1 Loomine

Sufiksiloendi saab konstrueerida otse sufiksipuust seda laiuti läbides. Ajakulu on vaid $O(n)$. Aga selle jaoks peab muidugi olemas olema sufiksipuu, mis võtab palju ruumi ning mille loomise algoritm on küllaltki keerukas. Seega ei ole selline sufiksiloendi koostamine kuigi praktiline.

Esimese ideena tuleb vast pähe, et leiame kõik sufiksid (lihtne ju!) ning sorteerime ära. Sorteerimise keerukus on ju $O(n\ log\ n)$. Siin tuleb aga tähele panna, et kuna kahe pika sufiksi võrdlemine ei võta mitte $O(1)$ vaid $O(n)$, siis selline triviaalne meetod on keerukusega $O(n^2\ log\ n)$.

Selle asemel sorteerime sufiksid ainult esimese tähe järgi, seejärel esimese kahe tähe järgi, siis esimese nelja tähe järgi jne. Kui meil on sorteeritud sufiksid 2^iesimese tähe järgi, siis saame ühe võrdlemisega paari kohta sorteerida need esimese 2^{i+1}tähe järgi.

Tavalist sorteerimist kasutades on sellise lähenemise keerukuseks sufiksipuu loomisel $O(n\ log^2 n)$. Enamasti on see võistlustel piisav ning koodikirjutamise ja testimise aja kokkuhoiu mõttes optimaalne. Kui aga on vaja veelgi kiiremini, siis saab kasutada kiiremat sorteerimisalgoritmi.

Siin on funktsioon sufiksiloendi loomiseks koos kiire sorteerimisega:

```
string tekst, sona;
int* sufiksid, *tmpS;
int* kohad, *tmpK;
int loendur[256];

void sorteeri(int k, int n) {
    memset(loendur, 0, sizeof(loendur)); //nullime loenduri
    for (int i = 0; i < tekst.length(); i++) {
        if(i + k < n) loendur[kohad[i + k]]++; //loendame
          else loendur[0]++;
     }
    int sum = 0;
    for (int i = 0; i < 256; i++) {
        int t = loendur[i];
        loendur[i] = sum; //kumulatiivsed algused
        sum += t;
    }

    for (int i = 0; i < tekst.length(); i++) {
          if (sufiksid[i] + k < n)
               tmpS[loendur[kohad[sufiksid[i] + k]]++] = sufiksid[i];
          else
               tmpS[loendur[0]++] = sufiksid[i];
     }

    for (int i = 0; i < tekst.length(); i++) {
        sufiksid[i] = tmpS[i];
    }
}
```

```
void loo_sufiksiloend(int n) {

    sufiksid = new int[n];
    kohad = new int[n] {};
    tmpS = new int[n] {};
    tmpK = new int[n] {};
     for (int i = 0; i < n; i++) {
        sufiksid[i] = i; //kõik sufiksid, igast indeksist alates
        kohad[i] = tekst[i]; //algsed kohad
    }
     for (int k = 1; k < n; k <<= 1) {
          sorteeri(k, n);
          sorteeri(0, n);
          tmpK[sufiksid[0]] = 0;
        int r = 0;
        for (int i = 1; i < n; i++) {
            if (kohad[sufiksid[i]] != kohad[sufiksid[i - 1]] ||
                kohad[sufiksid[i] +k] != kohad[sufiksid[i-1] + k]) {
                r++;
            }
            tmpK[sufiksid[i]] = r;
        }
        for (int i = 0; i < tekst.length(); i++) {
            kohad[i] = tmpK[i];
        }
        if (kohad[sufiksid[n - 1] == n - 1]) break;
    }
}
```

11.5.2 Sufiksiloendi kasutamine

11.5.2.1 Alamstringi leidmine

Alamstringi otsimisel sufiksiloendist kasutame sama omadust, et iga otsitav string, mis tekstis leidub, on mingi selle teksti sufiksi prefiksiks. Kuna sufiksiloendis on sufiksid sorteeritud, siis saame kasutada kahendotsingut. Selle keerukuseks on $O(m * log\ n)$, kus m on alamstringi pikkus ja n teksti (ja ka sufiksiloendi) pikkus.

```
int leia_prefiks(string str)
{
     int m = str.length();  // otsitava stringi pikkus

     //kahendotsing
     int vasak = 0, parem = tekst.length()-1;
     while (vasak <= parem)
     {
          int mid = vasak + (parem - vasak) / 2; // leiame keskmise
          char* prefiks = new char[m];

          int res = strncmp(str, tekst + sufiksid[mid], m);

          // leitud
          if (res == 0)
          {
               return suffArr[mid];
          }

          // Otsi vasakult
          if (res < 0) parem = mid - 1;

          // otsi paremalt
          else vasak = mid + 1;
     }

     return -1; ei leidnud;
}
```

Seegi on enamasti piisav, kuid vajadusel saab sedagi kiiremini teha, kui luua abitabel, kus on kirjas, mitu märki on võrreldavate sufiksite alguses samad. Siis saab võrdlemist alustada juba sellest positsioonist, kust vajalik.

11.6 Räsi

Veel üks stringitöötluses sageli vajaminev operatsioon on kahe stringi võrdsuse kontroll. Kui juba stringide esimesed märgid on erinevad, on see operatsioon kiire, aga mida teha juhul, kui esimene erinev märk on alles tuhandendal positsioonil? Sel juhul saab arvutada välja stringide räsid (i.k. *hash*) ja hoopis neid omavahel võrrelda.

Räsifunktsioon on funktsioon, mis seab mingitele andmetele, sageli suvalise pikkusega, vastavusse mingi väärtuse ehk räsi, mille pikkus ja formaat on fikseeritud.

Ideaalis saaksid samad andmed sama räsi ja vastupidi- ühele räsile vastavad samad andmed. Kuna aga räsid on üldjuhul kompaktsemad kui andmed ise, esineb olukordi, kus erinevatele andmetele võib vastata sama räsi. Sellist olukorda nimetatakse **kokkupõrkeks** ehk **kollisiooniks**.

Põhimõtteliselt võib räsifunktsioon olla milline iganes, aga hea räsifunktsioon peab olema kiiresti arvutatav ning ei tohi tekitada palju kollisioone.

Vaatame näiteks stringi $S = s_1 s_2 s_3 \dots s_n$. Milline võiks olla sellele vastav räsifunktsioon? Üheks lihtsaks funktsiooniks on kokku liita iga tähemärgi kood, kuid sellisel juhul tekib palju kokkupõrkeid, sest näiteks stringid ’OAR’ ja ’MAT’ saaksid sama räsi.

Enamasti on tekstitöötlusalgoritmide juures oluline ka märkide järjekord stringis, seega tuleks valida kindlasti selline räsifunktsioon, mis arvestab tähemärgi positsiooni.

Üheks heaks funktsiooniks on

$$f(S) = \sum_{i=0}^{n-1} s_i * p^i$$

kus p on vabalt valitud arvud.

Vältimaks kokkupõrkeid võiks p olla suurem kui märkide arv kasutatavas tähestikus.

11.6.1 Alamstringi leidmine räsi abil

Vaatame aga, mis kasu on räsist alamstringide leidmisel. Oletame, et meil on tekst $S = s_1 s_2 s_3 \dots s_n$ ja me soovime leida sellest alamstringi $A = a_1 a_2 a_3 \dots a_k$ Otsitava alamstringi räsi saame lihtsalt välja arvutada: $f(A) = \sum_{i=0}^{k-1} a_i * p^i$, kuid kuidas leida seda tekstist? Oletame, et otsitav alamstring algab tekstis positsioonilt u ja lõpeb positsioonil v. Sellisel juhul

$$f(A) = f(S, 0, v-u) = \sum_{i=0}^{v-u} s_{i+u} * p^i$$

Sellest üksi ei ole palju kasu. Aga pane tähele, et

$$f(S, 0, v) = \sum_{i=0}^{v} s_i * p^i$$

ja

$$f(S, 0, u-1) = \sum_{i=0}^{u-1} s_i * p^i$$

Ning nende vahe on:

$$f(S, 0, v) - f(S, 0, u-1) = \sum_{i=0}^{v} s_i * p^i - \sum_{i=0}^{u-1} s_i * p^i = \sum_{i=u}^{v} s_i * p^i = f(A) * p^u.$$

Sellest on muidugi kasu vaid siis, kui peame meeles kõigi tekstis esimesest positsioonist algavate alamstringide räsid. Samas teksti räsi arvutades on see vaid meelespidamise kulu. Vaatame ka pisikest näidet. Oletame arvutuste lihtsuse mõttes, et meie tähestik koosneb neljast märgist (a=1,b=2,c=3,d=4), sellisel juhul võime valida p=5. Oletame, et meil on tekst S = abcab.

$$f('abcab') = a * 5^0 + b * 5^1 + c * 5^2 + a * 5^3 + b * 5^4$$
$$= 1 * 1 + 2 * 5 + 3 * 25 + 1 * 125 + 2 * 625 = 1461$$

Arvutuste käigus saame ka kõik prefiksid:

A	Ab	abc	Abca	abcab
1	11	86	211	1461

Leiame nüüd 'ca' räsi kõigepealt tabelist

$$\frac{f('abca') - f('ab')}{5^2} = \frac{211 - 11}{25} = 8.$$

Ja kontrolliks otse:

$$f('ca') = c * 5^0 + a * 5^1 = 3 * 1 + 1 * 5 = 8$$

11.6.2 Räsi moodularvutusega

Kuna aga päris tekstid võivad olla pikad, siis selliselt arvutatud räsi läheb väga suureks. Selle vältimiseks võetakse tihti räsi leidmisel jääk mingi suure arvuga jagamisel ehk mooduli m järgi.

Mooduli m valikul tuleb silmas pidada, et see oleks küllaltki suur, et vähendada kokkupõrkeid, samuti võiks m olla algarv. Eelpool toodud räsifunktsioon omandab siis kuju

$$f(S) = \left(\sum_{i=0}^{n-1} s_i * p^i\right) mod\ m,$$

mis omakorda on samaväärne

$$f(S) = \left(\sum_{i=0}^{n-1} (s_i * p^i) mod\ \ m\right) mod\ m.$$

Arvutamine moodulitega:

1. $a\ mod\ m = a,\ \ kui\ 0 \le a < m$
2. $(a\ mod\ m)\ mod\ m\ =\ a\ mod\ m$
3. $(a + b)\ mod\ m\ =\ [(a\ mod\ m) +\ (b\ mod\ m)]\ mod\ m$
4. $(a - b)\ mod\ m\ =\ [(a\ mod\ m) - (b\ mod\ m) + m]\ mod\ m$
5. $(ab)\ mod\ m\ =\ [(a\ mod\ m)(b\ mod\ m)]\ mod\ m$
6. $a^b\ mod\ m\ =\ (a\ mod\ m)^b\ mod\ m$
7. $(a \div b)\ mod\ m = [(a\ mod\ m) \cdot (b^{-1}\ mod\ m)]\ mod\ m$, kus b^{-1} on arvu b modulaarne

Uurime, kuidas muutub meie eelpool toodud arvutuskäik, kui leiame räsi mooduli m = 101 järgi. (Nii väikese mooduli valime muidugi arvutuste näitlikustamiseks.)

Liitmine, lahutamine, korrutamine ja isegi astendamine eriti probleeme ei tekita, kuid mis saab jagamisega? Üheks võimaluseks on leida modulaarne pöördväärtus (vt meeldetuletuseks vajadusel arvuteooria peatükki) ehk selline arv p', mille korral $p * p' \equiv 1\ (mod\ m)$.Kui $p = 5$, nagu meie näites, siis sobib $p' = 81$, sest

$$5 * 81\ (mod\ 101)\ =\ 405(mod\ 101)\ =\ 1.$$

Sellisel juhul saab prefiksite tabel mooduli 101 järgi järgmise sisu:

A	Ab	Abc	Abca	abcab
1	11	86	9	47

Arvutame nüüd tabeli põhjal moodularvutuse reegleid kasutades stringi $'ca'$ räsi:

$$f('ca') = \big(f('abca') - f('ab')\big) * (p')^2\ (mod\ 101) = \big((9 - 11) * 81^2\big)(mod\ 101)$$
$$= (9 - 11) mod\ 101 * 81^2\ mod\ 101 = 99 * 97\ mod\ 101 = 8.$$

11.6.3 Jagamise vältimine

Kuna aga modulaarse vastandarvu leidmine pole triviaalne ja nõuab ka kas koodikirjutamist või kusagilt järele vaatamist, siis üheks võimaluseks on ehitada räsifunktsioon ja prefiksitabel üles nii, et jagamise asemel saame kasutada korrutamistehteid. Näiteks kasutades räsifunktsioonina

$$f(S) = (\sum_{i=0}^{n-1} s_i * p^{n-1-i}) \; mod \; m.$$

Prefiksite arvutamine läheb nüüd veidi põnevamaks, kuna iga prefiksi pikkus on ju erinev. Esimene prefiks a on ikka $a * 5^0 (mod\ 101) = 1$. Järgmine, $f('ab') = a * 5^1 + b * 5^0 (mod\ 101) = 7$ ja kolmas, $f('abc') = a * 5^2 + b * 5^1 + c * 5^0 (mod\ 101) = 38$. Selline ükshaaval algusest peale arvutamine on muiudgi liiga ajamahukas, aga taas saab arvutada järgmise prefiksi räsi eelmist väljaarvutatud räsi kasutades. Pane tähele, et

$$\begin{aligned} f('abc') = a * 5^2 + b * 5^1 + c * 5^0 (mod\ 101) &= 5(a * 5^1 + b * 5^0) + c * 5^0 \; (mod\ 101) \\ &= \; 5(a * 5^1 + b * 5^0) \; mod \; 101 + \; c * 5^0 \; (mod\ 101) \\ &= (5 * f('ab') + c) \; (mod\ 101) \end{aligned}$$

Siin on taas uus prefiksite räsiväärtuste tabel:

A	Ab	Abc	Abca	abcab
1	7	38	90	48

Arvutus tabelist:

$$f('ca') = f('abca') - f('ab') * p^k (mod\ 101) = 90 - 7 * 5^2 = 16.$$

Veendume, et sama tulemuse saame ka otse arvutades:

$$f('ca') = c * 5^1 + a * 5^0 (mod\ 101) = 16.$$

11.6.4 Rabin-Karpi algoritm

Ülesannetes on enamasti vaja leida tekstist S alamstring A. Vaatame nüüd, kuidas seda teha kasutades räsi.

Jõumeetodiga lahendades peaksime hakkama võrdlema algusest peale kõigepealt s1 ja a1. kui need langevad kokku, siis s2 ja a2 jne. Kui kõik langevad kokku, oleme leidnud ühe alamstringi esinemise tekstis. Kui mingil kohal i s_i <> a_i, siis hakkame otsima teksti järgmisest positsioonist ja võrdleme s2 ja a1 jne. Selle keerukuseks on O(n*m), kus n on teksti pikkus ja m ostitava alamstringi pikkus.

Räsi abil otsimine käib tegelikult samamoodi, aga selle asemel, et võrrelda alamstringi tähemärke ükshaaval, saab arvutada välja räsid ja võrrelda neid.

Esmalt arvutame alamstringi A räsi f(A). Seejärel arvutame teksti esimese m märgi räsi f(S,0,m-1). Kui need on võrdsed, oleme tõenäoliselt leidnud alamstringi tekstist. Kui mitte, siis kokkulangevust kindlasti ei ole ja meil on vaja leida järgmise n pikkusega alamstringi räsi tekstist ehk f(S, 1, m). Kui me peaksime kogu räsi uuesti välja arvutama, siis mingit võitu võrreldes jõumeetodiga ei tule. Parem on kasutada eelpool toodud prefiksite ettearvutamist või kui otsimine on ühekordne, siis libistada räsi mööda teksti: eemaldame eest esimese märgi väärtuse ja lisame lõppu uue märgi väärtuse. Nii saame rulluda läbi kogu teksti ja leida kõik kattuvused. Sellist meetodit alamstringi leidmiseks

nimetatakse **Rabin-Karp**i algoritmiks ja selle mõtlesid välja Richard M. Karp California Berkeley Ülikoolist ja Michael O. Rabin Harvardi Ülikoolist 1987. aastal.

Vaatame, kuidas leida stringi 'ca' tekstist 'abcab'. f('ca') = 16. Arvutame kõigepealt tavapärasel moel stringi 'ab' räsi: f(ab) = 7. Kuna $16 \neq 7$, siis kokkulangevust kindlasti pole ja leiab järgmise räsi:

$$f('bc') = \big((7 - 1 * 5 * \ mod\ 101) * 5 + \ 3\big)\ mod\ 101 = 13$$

Kuna ka $16 \neq 3$, saame asuda järgmist räsi arvutama:

$$f('ca') = \big((13 - 2 * 5 * \ mod\ 101) * 5 + \ 1\big)\ mod\ 101 = 16$$

Kokkulangevuse korral kontrollime üle, kas on tegelik kokkulangevus või räsi kollisioon. Antud juhul on tegelik kokkulangevus.

11.6.5 Ülesanne

Mõnikord on vaja otsida ja /või loendada etteantud tekstidest mingeid teatud pikkusega sõnu: näiteks leida lepingutest etteantud isikukoode või teatud telefoninumbreid vms.
Sulle on antud tekst T ja n stringi, igaüks pikkusega k. Leia iga stringi kohta, mitu korda see tekstis T esineb.
Sisendi esimesel real on kaks täisarvu n – otsitavate stringide arv ja k – otsitavate stringide pikkus.
Järgmisel n real on igaühel string pikkusega k. Viimasel real on tekst T.
Väljundisse kirjutada n rida, kus iga real on otsitav string ja selle esinemissagedus tekstis T.
NÄIDE:

```
5 4
koer
hiir
kass
kukk
kana
kass ronis puu otsa ja kukkus alla koer viisakana aitas kassi.
```

Vastus:

```
koer 1
hiir 0
kass 2
kukk 1
kana 1
```

Kasutame selle lahendamiseks Rabin-Karpi algoritmi. Kuna antud ülesandes on paring ühekordne ja sõnad ette teada, siis tegelikult ei ole vaja kõiki räsisid alles hoida: võime lihtsalt teksti algusest räsid arvutada ja neid kohe otsingusõnade räsidega võrrelda. Aega see oluliselt kokku ei hoida, kuid ruumi küll.

```
#define ull unsigned long long
using namespace std;

char* tekst; // pikk tekst
ull m = 1000000007; // suur algarv jäägi jaoks
ull p = 256; // märkide arv
int k, n; //k-otsisõne pikkus, n-otsisõnede arv

//hoiame otsisõnu dünaamilises massiivis paaridena, kus esimeseks paaris on sona
//esinemissagedus, teisel kohal paar, milles esimesel kohal on sona räsi ja teisel
//kohal lõpuks sõna ise.
vector<pair<int, pair<ull, char*>>> otsisonad;

ull arvuta_rasi(char *str) { //arvutab k pikkusega räsi
    ull rasi = 0LL;
    for (int i = 0; i < k; i++) {
        rasi = (p*rasi + str[i]) % m;
    }
    return rasi;
}

void loenda_sonad()
{
    int teksti_pikkus = strlen(tekst);
    int teksti_rasi = arvuta_rasi(tekst); // esimese räsi väärtus teksti algusest
    int h = 1;

    // h väärtus on pow(d, k-1)%q"
    for (int i = 0; i < k - 1; i++)
        h = (h*p) % m;

    // libistame räsi
    for (int i = 0; i <= teksti_pikkus - k; i++)
    {
        for (int ii = 0; ii < n; ii++) { //iga otsisõna jaoks
            if (otsisonad[ii].second.first == teksti_rasi) { //kui on kattuvus
                int j = 0;
                for (j; j < k; j++)                    {
                    if (tekst[i + j] != otsisonad[ii].second.second[j])
                        break; // kollisioon
                }

                if (j == k) //kui kollisiooni polnud ja oli tegelik kattuvus
                    otsisonad[ii].first++; //sona leitud!
            }
        }
        // arvutame järgmise räsi eemaldades esimese tähe ja lisades järgmise
        if (i < teksti_pikkus - k)
        {
            teksti_rasi = (p*(teksti_rasi - tekst[i] * h) + tekst[i + k]) % m;

            if (teksti_rasi < 0) //kui jääk on negatiivne
                teksti_rasi += m; //teeme positiivseks
        }
    }
}
```

11.7 KONTROLLÜLESANDED

11.7.1 Telefoninumbrid

Kunagi ammustel aegadel, kui loomad veel rääkisid ning inimesed taevas lendasid, kasutati selliseid riistapuid, mida nimetati klahvidega telefonideks. Nendel telefonidel oli 10 numbriklahvi, millest osad võisid ka tähti tähistada järgneva skeemi põhjal:

```
2-ABC
3-DEF
4-GHI
5-JKL
6-MNO
7-PQRS
8-TUV
9-WXYZ
```

Siis taibati, et mingeid telefoninumbreid saab meelde jätta, asendades võimalusel numbri ühe tähega nende hulgast, mis vastavad sellele numbrile. Näiteks võib numbrile 340833 vastata tähekombinatsioon EI0TEE. Sulle on nüüd antud üks sarnasel viisil kirjutatud telefoninumber, mille pikkus ei ületa 30 tähemärki ning mis koosneb vaid suurtähtedest ning numbritest 0 ja 1, ning sa pead väljastama sellele vastava telefoninumbri.



NÄIDE 1:

EI0TEE

Vastus:

340833

NÄIDE 2:

MUKODUKEONTILLUKE1

Vastus:

685638536684558531

NÄIDE 3:

MINUARMASKASS

Vastus:

6468276275277

11.7.2 Enigma

Teise maailmasõja ajal oli sakslaste Enigma kodeerimismasina murdmine väga oluline samm, mis lubas liitlastel lugeda Saksa sõjaväe teateid. Teatavasti oli iga Enigmaga saadetud sõnumi pikkusega `n` ees `n` fibonacci arvu, millele järgnes suvalise pikkusega tekst, mis koosnes erinevatest märkidest, kuid milles oli täpselt `n` suurtähte. Enigma dešifreerimiseks pidi alguses lihttekstist kõik sümbolid, mis ei olnud suurtähed, välja viskama. Originaalse teksti saamiseks tuli vaadata kohal `i` olevat fibonacci arvu, mis on fibonacci jadas kohal `j`. See tähendas, et täht kohal `i` on originaaltekstis kohal `j`. Antud krüpteering eeldab, et fibonacci jada esimene arv on `1`, teine on `2` ning `f(n+1) = f(n) + f(n-1)`, kus `f(x)` tähistab kohal `x` olevat fibonacci arvu. Kui mingile kohale `j` ei vasta ükski antud fibonacci arv, kuid eksisteerib koht `k`, kus `k > j` ning `k`-le vastab mingi fibonacci arv, siis oli sellel kohal originaaltekstis tühik. Näiteks sõnumit, mis koosnes jadast `317 987 1587 8 2 13 144 610 34 3 1 89 55` ning lausest „`Jah, MA OlEN RASkE SUU.`“, sai dekodeerida, eemaldades kõigepealt kõik märgid, mis ei ole suurtähed, et saada `JMAOENRASESUU.` Kuna `317` on 13. fibonacci arv, läheb täht `M` kohale 13. Protsessi jätkates, saame sõnumi `SEE ON SUUR JAMA.` Paneme tähele, et tühikud tekivad sellel põhjusel, et neile vastavaid arve (5, 21 ja 233) ei ole kasutusel. Sulle on ette antud üks Enigma abil saadetud sõnum, mille pikkus ei ületa 100 tähemärki. Leia, mis on originaaltekst.



Sisendi esimesel real on arv `n` - fibonaccide arvude hulk. Järgmisel real on `n` fibonacci arvu. Kolmandal real on saadetud tekst.
Väljundisse kirjutada dešifreeritud tekst.

NÄIDE 1:

```
13
317 987 1587 8 2 13 144 610 34 3 1 89 55
Jah, MA OlEN RASkE SUU.
```

Vastus:

```
SEE ON SUUR JAMA
```

NÄIDE 2:

```
14
55 13 144 89 5 1597 1 377 987 3 2 34 610 21
```

Vastus:

```
GLooBUs VALITsEs AIa TagaplaanIl
```

11.7.3 Universaalgeen

On olemas tulnukate rass, kelle genoomi kirjeldamiseks on vaja kasutada kõiki väikesi ladina tähti. Need tulnukad otsustasid otsida nn universaalgeeni, mis on defineeritud kui genoomis kõige tihedamini esinev alamstring pikkusega `n` `(0<n<11)`. Leia see universaalgeen.
Sisendi esimesel real on arv `n`. Järgmisel real on string, mille pikkus on maksimaalselt 1000 – tulnukate genoomi kirjeldus.



Väljundisse kirjutada üks string: kõige tihedamini esinev alamstring. Kui vastuseid on mitu, väljasta neist leksikograafiliselt vähim.

NÄIDE 1:

```
3
baababacb
```

Vastus:

```
aba
```

11.7.4 Arelofansi keel

Arelofansi keel on austroneesia keelkonda kuuluv keel, mida kõneldakse peamiselt Arelofansi saarel Indoneesias ning mida kõneleb umbes 100-150 inimest. Keele üks omapäradest on see, et täishäälikutel vahet ei tehta, mistõttu eurooplased, kes on varem üritanud keele omapärasid kirja panna, ei suutnud kokku leppida sõnade tõlgetes. Aidake neid.
Sisendis on antud kahel real kaks sõna arelofansi keeles. Sõnade pikkused ei ületa 100 tähemärki. Leida, kas antud sõnad on sama tähendusega või mitte. Antud juhul on kaks sõna sama tähendusega parajasti siis, kui arelofansi keele kõneleja teeb neil vahet.
Väljundisse kirjutada üks sõna. Kui sõnad on sama tähendusega, väljastada `SAMA`, kui erinev, siis `ERINEV`.

NÄIDE 1:

```
arelofansi
erilufenso
```

Vastus:

```
SAMA
```

NÄIDE 2:

```
ai
ei
```

Vastus:

```
SAMA
```

NÄIDE 3:

```
mina
nina
```

Vastus:

```
ERINEV
```

NÄIDE 4:

```
saen
san
```

Vastus:

```
ERINEV
```



11.7.5 Koodigeneraator

Tänapäeval on olemas mitmeid keskkondi, kus inimene saab programmeerimist harjutada, saates ülesandeid testerile testimiseks. Paljud saidid aga kontrollivad vaid näidistesti lahendamise õigsust, mistõttu töötab ka programm, mis tagastab vaid näidisväljundi. Sinu eesmärk on katsetada, kas selline programm läheb läbi, kuid kuna ülesandeid on mitu, otsustasid teha oma koodigeneraatori. Näidisväljund asub `n (0 < n < 100)` real. Sinu programm peab vastama kujule

```
#include<string.h>
#include<stdio.h>
int main()
{
<koodiread>
printf(„\n“);
return 0;
}
```

kus `<koodiread>` asemel peab olema n rida kujul `printf(„<rida>\n“);` kus `<rida>` asemel on vastav rida näidisväljundis.
Sisendi esimesel real on arv n. Järgmisel n real on tekst, mille pikkus ei ületa 1000 tähemärki - näidisväljund.
Väljundisse kirjutada näidisväljundit kirjutav programm.

NÄIDE:

```
2
Mina Olen Tubli Tudeng
M.O.T.T.
```

Vastus:

```
#include<string.h>
#include<stdio.h>
int main()
{
printf(„Mina Olen Tubli Tudeng \n“);
printf(„M.O.T.T.\n“);
printf(„\n“);
return 0;
}
```



11.7.6 Automaathelistaja

Uus äpp lubab sul valida telefoninumbri nii, et kui ekraanil olev number on sinu kontaktnimekirjas olemas, hakkab see kohe antud numbrile helistama. See võib aga probleeme tekitada, sest su sõbra Juku telefoninumber on 11234478, mistõttu ühendatakse sind häirekeskusega, enne kui Juku numbrini jõuad. Nüüd mõtled sa, kas Jukul oleks sellisest äpist kasu, või on tal ka olemas selline numbrite paar, millest ühe valimisel annab see äpp poole pealt teise.
Sisendi esimesel real on täisarv `n` (`0<n<10001`) – Juku telefoniraamatu suurus. Järgmisel `n` real on `n` kuni kümnekohalist telefoninumbrit.
Väljundisse kirjutada üks sõna. Kui Jukul oleks äpist kasu, väljastada JAH, kui mitte, väljastada `EI`.

NÄIDE 1:

```
3
911
97625999
91125426
```

Vastus:

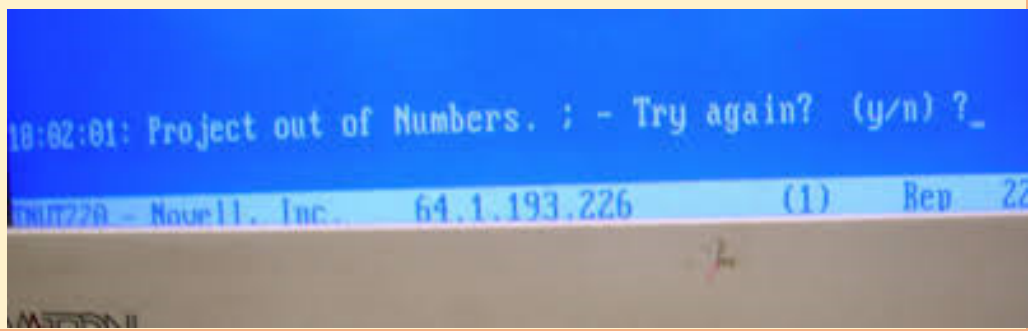
```
EI
```

NÄIDE 1:

```
5
113
12340
123440
12345
98346
```

Vastus:

```
JAH
```



11.7.7 Risti-rästi

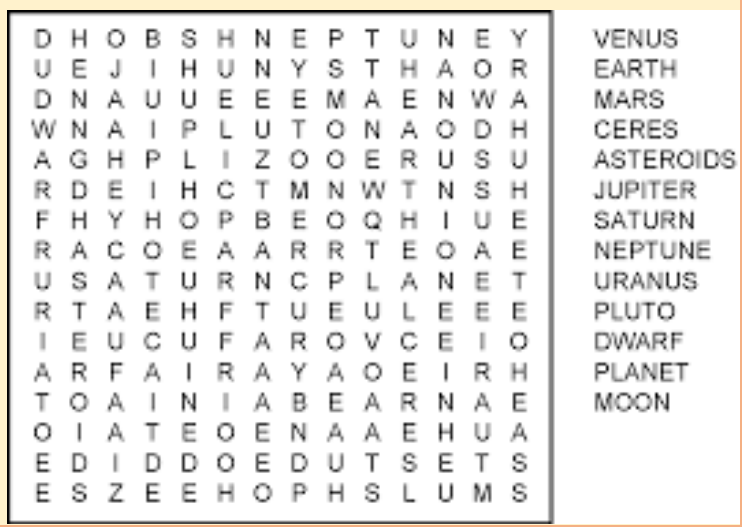
Juhal on `n*m` tabel (`0 < n, m < 51`), kus igas lahtris on mingi ladina täht. Ta peab sealt tabelist üles otsima `k` (`0 < k < 21`) sõna. Sõna võib olla tabelis kirjutatud sirgjooneliselt mistahes kaheksast võimalikust suunast. Suur- ja väiketähed ei mõjuta sõna korrektsust. Näiteks kui otsitav sõna on `LEGO`, aga tabelis on `lego`, on tabelis olev sõna õige. Leida kõikide otsitavate sõnade esimese tähe koordinaadid tabelis. Esimene koordinaat tähistab rea numbrit, teine tähistab märgi numbrit. Kui otsitavat sõna esineb tabelis mitu korda, tuleb valida võimalikest koordinaatidest kõige väiksema reanumbriga ning seejärel kõige väiksema veerunumbriga koordinaat.
Sisendi esimesel real on täisarvud `n` ja `m`. Järgmisel `n` real on sõned pikkusega `m` – tabeli read. Siis tuleb üksikul real arv `k`. Järgmisel `k` real on otsitavad sõnad.

NÄIDE 1:

```
8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

Vastus:

```
2 5
2 3
1 2
7 8
```



11.7.8 Ahvitembud

Kuulus zooloog Jane Goodall pani tähele, et päeva jooksul teevad ahvid maksimaalselt 26 erinevat tegevust, kuid iga ahv ei pea neid tegema samas järjekorras. Ta otsustas uurida, mis on maksimaalse pikkusega tegevuste nimekirja pikkus, mille kohta saab kindlalt väita, et mingi paar ahve teeb kõik nimekirjas antud tegevused järjekorras, st et tegevust loetakse tehtuks, kui ahv teeb parasjagu seda tegevust ning kõik varasemad tegevused nimekirjas on juba tehtud. Aidake teda. Sisendi esimesel real on kaks stringi, mille pikkused on kuni 100 tähemärki – kummagi ahvi tegevuskava.
Väljundisse kirjutada üks arv – pikima ühise tegevuskava pikkus.



NÄIDE 1:

```
abcd
acdb
```

Vastus:

```
3
```

NÄIDE 2:

```
abcd
dacb
```

Vastus:

```
2
```

11.7.9 Palindroomid

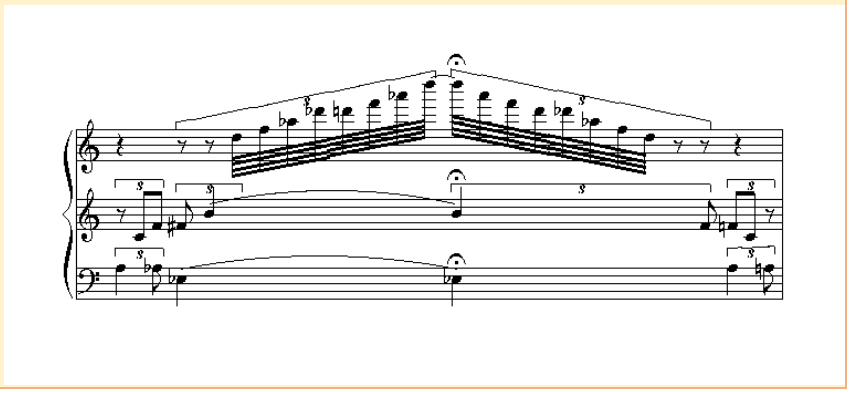
Rannarile meeldivad palindroomid. Tal on string pikkusega `n` `(0 < n < 1001)` ning ta tahab saada sellest palindroomi, lisades sellele võimalikult vähe tähti juurde. Aidake Rannarit ning leidke, mis on minimaalne vajalik tähtede arv.
Sisendi ainsal real on string pikkusega `n` – Rannari string.
Väljundisse kirjutada üks täisarv – minimaalne vajalik tähtede arv, mis on vaja lisada, et Rannaril oleks palindroom.

NÄIDE 1:
`siga`
Vastus:
`3`

NÄIDE 2:
`ees`
Vastus:
`1`

NÄIDE 3:
`kuulilennuteetunneliluuk`
Vastus:
`0`

NÄIDE 4:
`pqrsabcdpqrs`
Vastus:
`9`



11.7.10 Otsing

Andmebaasis on n märksõna (`0 < n < 20000`), mis on nummerdatud arvudega 1 kuni `n`. Iga märksõna on pikkusega kuni 100 tähemärki ning koosneb väikestest ladina tähtedest.
Otsingualgoritm peab võtma vastu stringi, mille pikkus on samuti kuni 100 tähemärki ning väljastama esimese kümne märksõna numbrid, milles on otsitav string märksõnas alamstringina. Märksõnad sorteeritakse alguses suuruse järjekorras – väiksemad enne. Sama pikkusega märksõnad sorteeritakse leksikograafiliselt ning kui andmebaasis on kaks ühesugust märksõna, tuleb enne väljastada väiksema numbriga märksõna number. Kui tingimusele vastavaid märksõnu on vähem kui kümme, tuleb väljastada kõikide märksõnade numbrid, mis vastavad antud tingimusele, ning kui tingimusele vastavaid märksõnu andmebaasis ei ole, tuleb väljastada arv `-1`. Kirjutage see otsingualgoritm.
Sisendi esimesel real on arv `n`. Järgmisel n real on andmebaasis hoitavad märksõnad. Märksõnad on antud nummerdamise järjekorras – esimene on numbriga `1`, viimane on numbriga `n`. Järgmisel real on arv `m (0 < m < 100001)` – päringute arv. Järgmisel `m` real on otsitavad stringid.
Väljundisse kirjutada `m` rida – iga päringu kohta õige arv tühikutega eraldatud numbreid.

NÄIDE 1:

```
17
tere
minu
nimi
on
kuuli
lennu
tee
tunneli
luuk
ja
ma
tulen
tartust
anne
linnast
anne
kanalist
5
a
an
jah
r
z
```

Vastus:

```
10 11 14 16 13 17
14 16 17
-1
1 13
-1
```

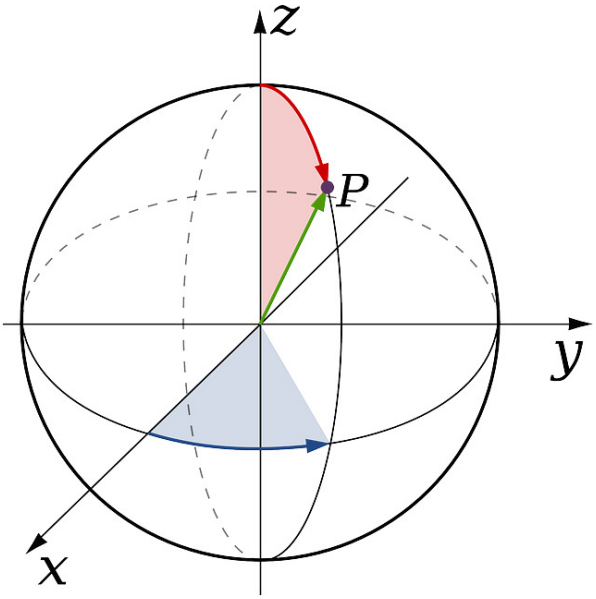


11.8 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk11 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
arvutus.cpp, arvutus.java, arvutus.py	Teksti parsimine
alamstring.cpp, alamstring.java, alamstring.py	Lihtne alamstringi leidmine
hulknurgad.cpp, hulknurgad.java, hulknurgad.py	Knuth-Morris-Pratti algoritm
spuu.cpp, spuu.java, spuu.py	Sufiksipuu
sloend.cpp, sloend.java, sloend.py	Sufiksiloend
rk.cpp, rk.java, rk.py	Rabin-Karpi algoritm

12 ARVUTUSGEOMEETRIA



Viimases peatükis tuleme mõnede teistega võrreldes väga vana valdkonna juurde, milleks on geomeetria. Geomeetrial on seoses maamõõtmise ja ehitusega väga praktilised väljundid ning sellele pandi alus juba vanas Kreekas, kuuendal sajandil enne meie aega. Kolmandal sajandil e.m.a lõi Eukleides oma teosega „Elemendid“ distsipliinile ka formaalsema raamistiku. Eukleidese töö seadis ka paljudeks järgnevateks sajanditeks standardi, millele kogu seonduv uurimistöö tugines.

Nagu ka teiste matemaatilise taustaga ülesannete puhul, annab arvuti kasutamine meile võimaluse käsitleda palju suuremaid andmehulki ja teha nendega rohkem arvutusi. Praeguses kontekstis tähendab see võimalust uurida suuremaid punktide, lõikude ja muude kujundite hulki kui see paberi ja pliiatsiga võimalik oleks.

Arvutiga lahendatavad geomeetriaülesanded võibki jagada laias laastus kaheks: ühed on sellised, mida on siiski realistlik enne paberi ja pliiatsiga lahendada ja edasi on ka küllaltki lihtne kirjutada programm, mis seejärel seda tüüpi ülesannet lahendada oskab. Teised on sellised ülesanded, mis nõuavad keerulisemaid algoritme – neid nimetataksegi arvutusgeomeetria ülesanneteks.

Arvutusgeomeetria ülesanded nõuavad enamasti nii head algoritmitundmist kui ka väga täpselt programmeerimist. Võrreldes paljude teiste ülesandeklassidega tuleb arvestada paljude erijuhtudega, mis teeb maksimumpunktidele lahendamise vaevalisemaks. Näiteks:

- Mis siis, kui jooned on paralleelsed?
- Mis siis, kui punktid kattuvad?
- Mis siis, kui hulknurk ei ole kumer?

Täiendavat keerukust lisab see, et tehteid tuleb enamasti teha ujukomaarvudega, mis toob kaasa vajaduse arvestada arvutuste täpsusest tulenevate probleemidega.

12.1 GEOMEETRILISED OBJEKTID

12.1.1 Punkt

Punkt on lihtsaim geomeetria objekt, sellel puuduvad mõõtmed ja seda iseloomustab ainult asukoht. Tasapinnal saab selle edasi anda kahe koordinaadiga, mida enamasti tähistatakse tähtedega x ja y. Seega on ka programmeerimises vaja, et säilitame iga punkti kohta tema asukoha ehk koordinaadid. Punkti tähistame $P(x, y)$.

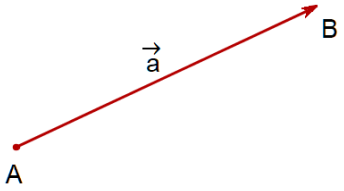
12.1.2 Lõik

Lõik (i.k. l*ine segment*) on kaht punkti ühendav sirgjoon. Neid punkte nimetatakse lõigu otspunktideks. Lõik on üheselt määratud oma otspunktidega. Lõigul on pikkus ja siht. Lõigud on näiteks kolmnurga küljed. Lõiku tähistame AB,kus A ja B on lõigu otspunktid. $AB = BA$.

12.1.3 Vektor

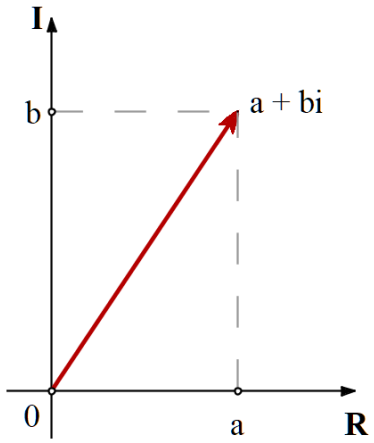
Vektor (i.k *vector*) on suunatud lõik, sellel on pikkus, siht ja suund. Kui kõik kolm parameetrit ühtivad, on vektorid võrdsed. Vektoreid, mis on samal sirgel, nimetatakse **kollineaarseteks**. Vektorit tähistame $\overrightarrow{AB}$. $\overrightarrow{AB} = -\overrightarrow{BA}$.

Vektori määrab üheselt selle algus ja lõpp-punkt. Abstraktsemalt saab aga kõik vektorid esitada ka ühtsest alguspunktist (enamasti siis koordinaatide alguspunktist). Sellisel juhul piisab vektori määramiseks ainult lõpp-punktist.



Vektorid on peamiselt huvitavad just seetõttu, et neil on suund ja nendega saab teha tehteid – neid saab liita, lahutada, korrutada arvuga ja teise vektoriga.

Punkti asukohta saab määrata ka tema kaugusega koordinaatide alguspunktist ja nurgaga x-telje suhtes. Sellisel moel on üles ehitatud kompleksarvud ja seega saab punktide hoidmiseks kasutada ka kompleksarvu klassi, kui selline on su kelle teegis defineeritud. Siin on väike ülevaade, mida saab teha keeles C++:



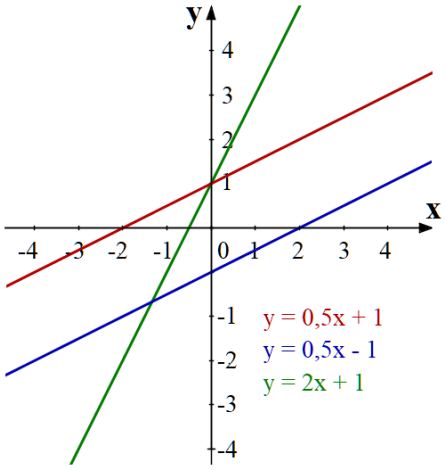
Kompleksarvu reaalosa a on punkti x koordinaadiks ja imaginaarosa kordaja b punkti y koordinaadiks. Kui meil on kaks punkti u ja v, siis $u + v$ on vektorite summa, mille alguspunkt on (0,0) ja lõpp-punktid vastavalt u ja v.
`abs(u)` on vektori u pikkus. Seda saab kasutada ka kahe punkti vahelise kauguse leidmiseks: punktide u ja v omavaheline kaugus on `abs(v-u)`.

`arg(u)` arvutab vektori u nurga x-telje suhtes radiaanides.

`polar(s, a)` loob vektori pikkusega s ja nurgaga a. Vektorit saab keerata a kraadi, korrutades selle vektoriga, mille pikkus on 1 ja nurk a: `u*polar(1,a)`, keerab vektorit u a kraadi.

12.1.4 Sirge

Sirge e sirgjoon on lihtsaim kahemõõtmeline objekt, see on kõverusteta joon, mis on lühim iga oma kahe punkti vahel. Sirge saab edasi anda sirge võrrandiga: $ax + by = c$. Iga punkt asukohaga (x, y), mis rahuldab seda võrrandit, asub antud sirgel ja ka vastupidi: iga sirgel asuv punkt rahuldab seda võrrandit.



Ühte punkti läbib lõpmata palju sirgeid, kuid kaks punkti määravad sirge üheselt. Kahe punkti järgi saab leida sirge võrrandi. Oletame, et meil on kaks punkti $P = (x_1, y_1)$ ja $Q = (x_2, y_2)$. Kuna need asuvad samal sirgel, siis $ax_1 + by_1 = c$ ja $ax_2 + by_2 = c$. Siit

$$ax_1 + by_1 = ax_2 + by_2 \Rightarrow a(x_1 - x_2) = b(y_2 - y_1).$$

See võrdus kehtib siis, kui valime a väärtuseks b kordaja ja vastupidi, st $a = (y_2 - y_1)$ ja $b = (x_1 - x_2)$.

Funktsioon, mis seda arvutab, on otsene:

```
void punktid_sirgeks(pair<double, double> P, pair<double, double> Q)
{
	double a = Q.second - P.second;
	double b = P.first - Q.first;
	double c = a*(P.first) + b*(P.second);

	if (b<0) {
		cout << a << "x " << b << "y = " << c << endl;
	}
	else{
		cout << a << "x + " << b << "y = " << c << endl;
	}
}
```

Sirget saab esitada ka ühe punkti ja **kaldenurga** kaudu.

12.1.5 Kiir

Kiir (i.k *ray*) on sirgjoon, millel on alguspunkt, kuid ei ole lõpp-punkti. Kiir tekib näiteks ühe lõigu pikendamisel lõpmatuseni üle ühe otspunkti. Programmeerimises tähendab see üldiselt lõiku, mille lõpp-punkt on piisavalt kaugel.

12.1.6 Murdjoon

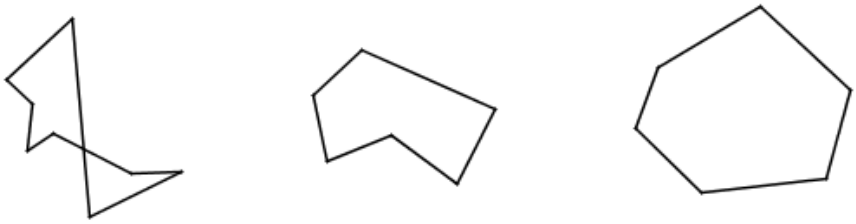
Murdjoon (i.k *polygonal chain*) on selline joon, mis koosneb järjestikustest, paarikaupa ühise otspunktiga lõikudest ehk lülidest $A_1A_2, A_2A_3, \ldots, A_{n-1}A_n$. Murdjoone lülide otspunktid $A_1 \ldots A_n$ on kõik erinevad. Kui $A_1 = A_n$, on tegemist kinnise murdjoonega. Murdjoont, mille ükski lüli ei lõiku, nimetatakse lihtsaks murdjooneks.

Murdjoont on mõistlik esitada punktiloendina.

12.1.7 Hulknurk

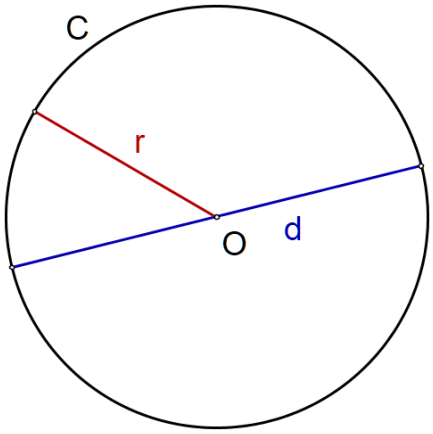
Hulknurk (i.k. *polygon*)on kinnine murdjoon. Lõike, millest murdjoon koosneb, nimetatakse külgedeks ja nende otspunkte tippudeks. Hulknurka, mille moodustab kinnine lihtne murdjoon, nimetatakse lihtsaks hulknurgaks. Kui hulknurga mistahes külje pikendamine mõlemas suunas lõpmatuseni ei lõika hulknurga ühtki külge, nimetatakse hulknurka **kumeraks**.

Järgmisel joonisel on ennast lõikav hulknurk, lihtne hulknurk ja kumer hulknurk:



12.1.8 Ringjoon

Ringjooneks (i.k. *circle*) nimetatakse sellist joont, mille iga punkt asub ühest punktist samal kaugusel. Seda punkti nimetatakse ringjoone **keskpunktiks** ja joone kaugust sellest nimetatakse ringjoone **raadiuseks**.



Keskpunkt ja raadius määravad ringjoone üheselt.

Ringjoone võrrand avaldub kujul

$$(x-a)^2+(y-b)^2=r^2,$$

Kus (a,b) on ringjoone keskpunkt ja r raadius.

Ringi ümbermõõdu ja pindala valemid on ehk kõigile tuttavad:

$$C=2\pi r$$

$$S=\pi r^2$$

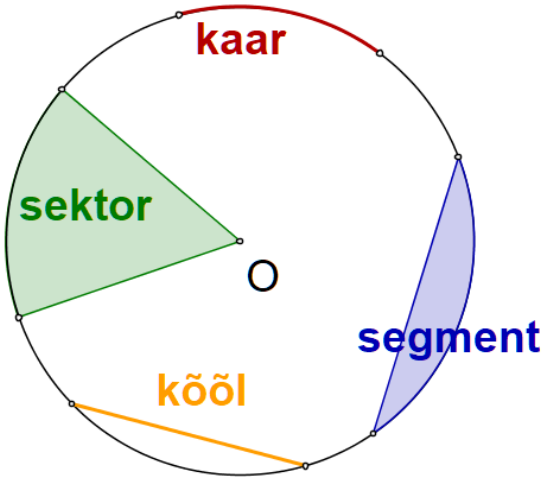
Programmeerimisülesannete lahendamisel on oluline mitte hakata π-d ise ümardama, vaid kasutada olemasolevat konstanti või määrata π trigonomeetria abil: $\pi=\mathrm{acos}(-1)$.

12.1.9 Kaar, kõõl, sektor ja segment

Kaareks (i.k. *arc*) nimetatakse üht pidevat osa ringjoonest. Kaarel on otspunktid, mis asuvad samuti ringjoonel. Kui tõmmata lõigud kaare kummastki otspunktist ringi keskpunkti, siis nende lõikude omavaheline nurk aitab leida kaare pikkuse:

$$P = \frac{\alpha}{360} \cdot C$$

kus α on eelpool kirjeldatud lõikude vaheline nurk ja C ringjoone pikkus.



Kõõluks (i.k *chord*) nimetatakse lõiku, mis ühendab kaks ringjoonel asuvat punkti omavahel.

Sektoriks nimetatakse ala ringist, mis on piiratud kaarega ja kaare otspunktidest tõmmatud raadiustega. Sektori pindala saab leida sarnaselt kaare pikkusele:

$$S = \frac{\alpha}{360} \cdot A$$

Kus A on ringi pindala.

Segment on osa ringist, mis on piiratud kõõluga ja kõõlu otspunktide vahel asuva kaarega. Segmendi pindala avaldub sektori pindala ja võrdhaarse kolmnurga, mille haaradeks on raadiused ning aluseks kõõl, vahena.

Ringjoone **puutujaks** (i.k *tangent*) nimetatakse sirget, mil on ringjoonega täpselt üks ühine punkt.

12.2 OPERATSIOONE PUNKTIDE JA SIRGETEGA

12.2.1 Punkti pööramine tasandil

Punti pööramiseks θ kraadi vastupäeva, saab kasutada seost:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

12.2.2 Kahe punkti vaheline kaugus

Kahe punkti vahelise kauguse tasandil saab leida Pythagorase teoreemi abil:

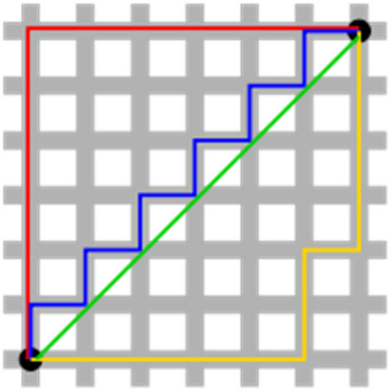
$$s = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

C++ on kõige mugavam kasutada funktsiooni hypot, mis just seda teebki:

```
int main()
{
	pair<double, double> p1(1,2), p2(3,4);
	double s = hypot(p2.first - p1.first, p2.second - p1.second);
	return 0;
}
```

Lisaks sellele kasutatakse ülesannetes vahetevahel nn **Manhattani kaugust**. Nimetus tuleb sellest, et Manhattanil on tänavad ilusti kas põhja-lõuna või ida-lääne suunalised ja ühest punktist teise saamiseks saab kasutada ainult neid tänavaid. Manhattani kaugus on vähim kaugus kahe punkti vahel ainult mööda horisontaalseid ja vertikaalseid lõike liikudes. Erinevalt tavalisest, Eukleidese kaugusest on lühimaks marsruudiks mitmeid võimalusi:





Kuigi võimalusi on palju, avaldub Manhattani kaugus väga lihtsalt, nimelt algus- ja lõpp-punkti koordinaatide vahede summaga:

$$MK = |x_2 - x_1| + |y_2 - y_1|$$

12.2.3 Paralleelsuse ja samasihilisuse kontroll

Sirget iseloomustab veel sirge **tõus** (i.k *slope*). Kui on antud kaks sirgel asuvat punkti $P = (x_1, y_1)$ ja $Q = (x_2, y_2)$, siis neid läbiva sirge tõus on $(x_1 - x_2)/(y_1 - y_2)$. Sirge võrrandist on tõus $-a/b$. Kaldenurga kaudu avaldub tõus kujul $tan(\alpha)$.

Horisontaalse sirge tõus on 0, vertikaalse sirge tõus on aga määramata (nulliga jagamine!). Programmeerides tuleb seda juhtu alati eraldi käsitleda.

Kaks sirget on paralleelsed siis ja ainult siis, kui neil on sama tõus. Kaks sirget ristuvad, kui nende tõusude korrutis on -1 või kui üks on horisontaalne (tõus 0) ja teine vertikaalne (tõus ∞).

12.2.4 Sirgete lõikepunkt

Sirgete lõikumist on kerge määrata, sest meil ongi tasandil 3 võimalust: sirged kas kattuvad, on paralleelsed või lõikuvad mingis punktis.

Kahe sirge lõikumispunkti saab määrata lihtsalt sirge võrrandist. Kui on antud sirged

$a_1x + b_1y = c_1$ ja $a_2x + b_2y = c_2$, siis nende lõikepunktiks on selline punkt, mis rahuldab samaaegselt mõlemat võrrandit. Korrutame esimest b_2 ja teist b_1-ga, saame

$$\begin{cases} a_1b_2x + b_1b_2y = b_2c_1 \\ a_2b_1x + b_1b_2y = b_1c_2 \end{cases}$$

Lahutades esimesest teise, saame

$$(a_1b_2 - a_2b_1)x = b_2c_1 - b_1c_2$$

millest

$$x = \frac{b_2c_1 - b_1c_2}{(a_1b_2 - a_2b_1)}$$

Analoogselt

$$y = \frac{a_2c_1 - a_1c_2}{(a_1b_2 - a_2b_1)}$$

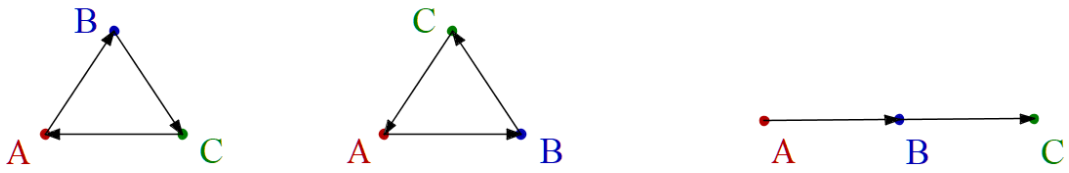
Funktsioon, mis leiab lõikepunktid:

```
pair<double,double> leia_loikepunkt(sirge s1, sirge s2)
{
	double determinant = s1.a*s2.b - s2.a*s1.b;

	if (determinant == 0) { //paralleelsed
		return make_pair(FLT_MAX, FLT_MAX);
	}
	else {
		double x = (s2.b*s1.c - s1.b*s2.c) / determinant;
		double y = (s1.a*s2.c - s2.a*s1.c) / determinant;
		return make_pair(x, y);
	}
}
```

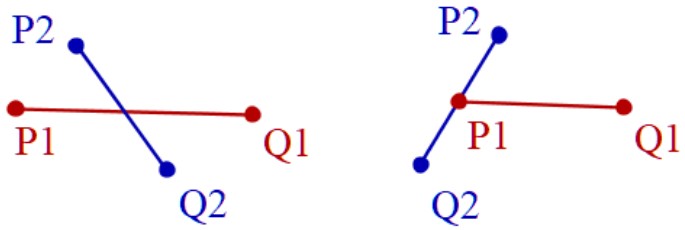
12.2.5 Lõikude lõikumine

Huvitavam ülesanne on aga määrata, kas kaks lõiku lõikuvad või kattuvad. Selleks vaatame kolme punkti tasandil ja nendevahelisi vektoreid. Eristame kolme võimalust:

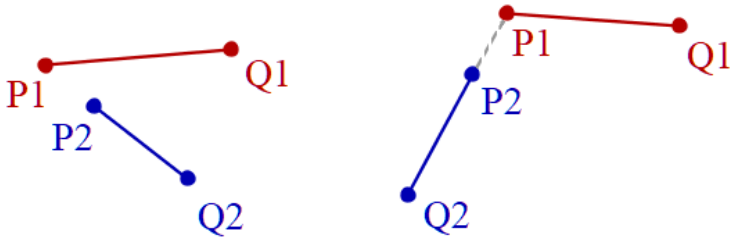


Esiteks, et punktid asuvad päripäeva, teiseks, et need asuvad vastupäeva ja kolmandaks, et need asuvad samal sirgel.

Vaatame nüüd, kuidas saavad tasandil asetseda omavahel kaks lõiku.



Kaks lõiku (p1,q1) ja (p2,q2) lõikuvad, kui p1,q1,p2 ja p1,q1,q2 on erineva suunaga ja p2,q2,p1 ja p2,q2, q1 on erineva suunaga.



Kui aga lõigud ei lõiku, siis p1,q1,p2 ja p1,q1,q2 on erineva suunaga, aga p2,q2,p1 ja p2,q2, q1 on sama suunaga.

Erijuhuks on osaliselt kattuvad lõigud:



Sellisel juhul vaatame projektsioone x ja y teljele. Kattumise korral kattuvad mõlemad projektsioonid. Kui üks projektsioonidest ei kattu, siis lõigud ei kattu.

Nende tähelepanekute põhjal saame defineerida järgmised funktsioonid:

```
bool on_loigul(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
    return false;
}

int suund(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
        (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0;  // kollineaarsed

    return (val > 0) ? 1 : 2; // suund: 1-päripäeva, 2-vastupäeva
}
```

```
bool loikuvad(Point p1, Point q1, Point p2, Point q2)
{
    int o1 = suund(p1, q1, p2);
    int o2 = suund(p1, q1, q2);
    int o3 = suund(p2, q2, p1);
    int o4 = suund(p2, q2, q1);

    // lõikuvad
    if (o1 != o2 && o3 != o4)
        return true;

    // p1, q1 ja p2 on kollineaarsed ja p2  asub lõigul (p1,q1)
    if (o1 == 0 && on_loigul(p1, p2, q1)) return true;

    // p1, q1 ja q2 on kollineaarsed ja q2 asub lõigul (p1,q1)
    if (o2 == 0 && on_loigul(p1, q2, q1)) return true;

    // p2, q2 ja p1 on kollineaarsed ja p1 asub lõigul (p2,q2)
    if (o3 == 0 && on_loigul(p2, p1, q2)) return true;

    // p2, q2 ja q1 on kollineaarsed ja q1 asub lõigul (p2,q2)
    if (o4 == 0 && on_loigul(p2, q1, q2)) return true;

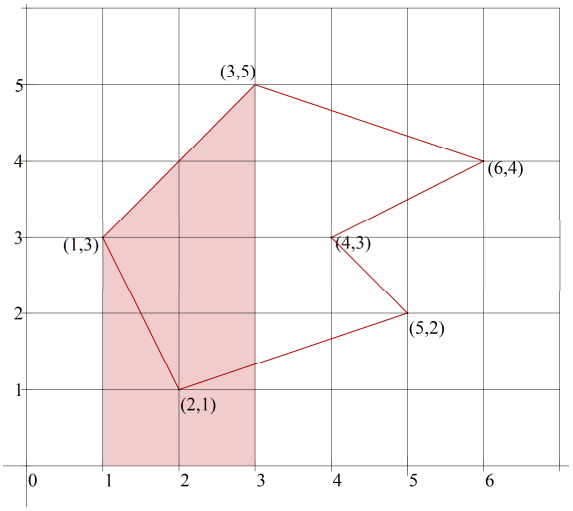
    return false; // ei kattu
}
```

12.3 HULKNURK

12.3.1 Hulknurga ümbermõõt ja pindala

Lihtsa hulknurga ümbermõõdu leidmine vast probleeme ei valmista – tuleb kokku liita hulknurka moodustavate lõikude ehk hulknurga servade pikkused. Lõigu pikkuseks on selle otspunktide vaheline kaugus.

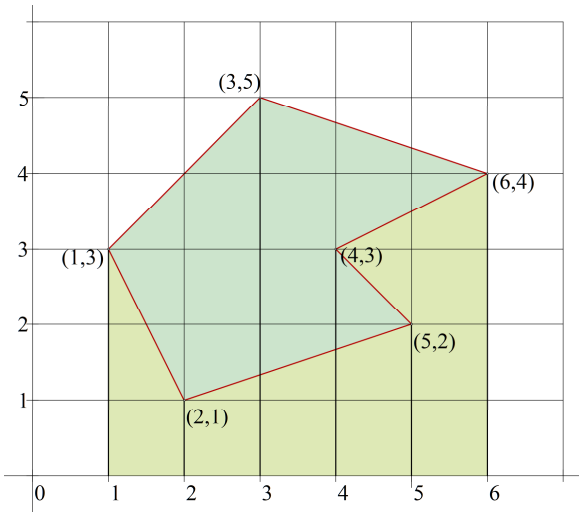
Pindalaga on veidi keerulisem, selle leidmiseks jagatakse hulknurk tavaliselt osadeks. Vaatame ühe külje ja x-telje vahele jäävat trapetsit:



Selle trapetsi pindala on ristküliku ja täisnurkse kolmnurga pindalade summa. Üheks kolmnurga kaatetiks on külje projektsioon x teljele ehk serva otspunktide x koordinaatide vahe. Teiseks kaatetiks on projektsioon y teljele, mis on võrdne otspunktide y koordinaatide vahega. Ristküliku küljed on samuti projektsioonid. Saame

$$\frac{(x_2-x_1)(y_2-y_1)}{2} + (x_2-x_1)*y_1 = (x_2-x_1)\left(\frac{y_2-y_1+2y_1}{2}\right) = \frac{(x_2-x_1)(y_2+y_1)}{2}.$$

Kordame sama iga külje jaoks:



Märke hoolega silmas pidades tulevad mõned pindalad negatiivsed, täpsemalt nende külgede omad, kus $x_2 < x_1$, mis joonisele vaadates tähendab, et heleroheline ala lahutatakse ära ja alles jääb tumerohelise ala pindala ehk just see, mis vaja:

$$\frac{(3-1)(5+3)}{2}+\frac{(6-3)(4+5)}{2}+\frac{(4-6)(3+4)}{2}+\frac{(5-4)(2+3)}{2}+\frac{(2-5)(1+2)}{2}+\frac{(1-2)(3+1)}{2}=$$

$$=\frac{2*8+3*9-2*7+5-3*3-4}{2}=\frac{21}{2}=10{,}5\ (ruut\ddot{u}hikut)$$

Funktsioon, mis sellel meetodil hulknurga pindala leiab:

```
double pindala(double X[], double Y[], int n)
{
      double S = 0.0;

      int j = n - 1;
      for (int i = 0; i < n; i++) {
            S += (X[j] + X[i]) * (Y[j] - Y[i]);
            j = i;  // j on i-le eelneva tipu nr
      }

      // absoluutväärtus!
      return abs(S / 2.0);
}
```

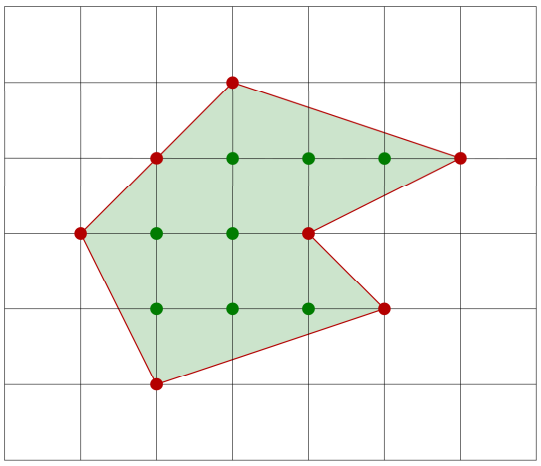
NB! See lähenemine eeldab, et tipud on sorteeritud järjekorras!

12.3.2 Picki teoreem

Kui hulknurga tippude koordinaadid on täisarvud, saab pindala leidmiseks kasutada ka **Picki teoreemi**. Nimelt avaldub hulknurga pindala valemiga

$$S = a + \frac{b}{2} - 1,$$

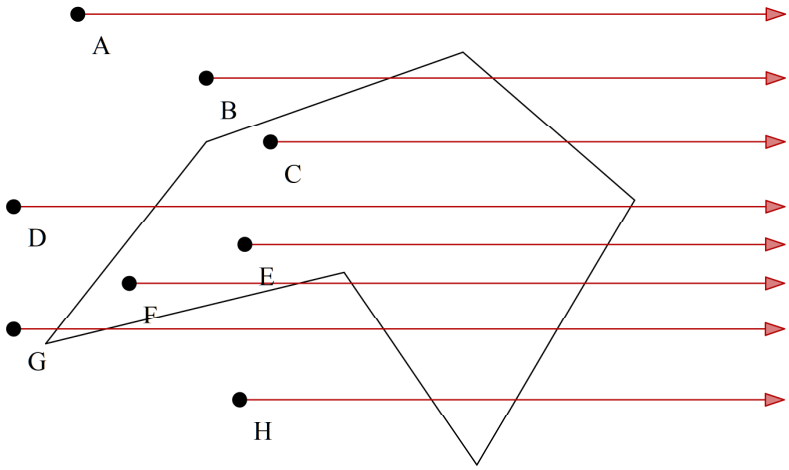
kus a on hulknurga sees asuvate täisarvuliste koordinaatidega punktide arv ja b on hulknurga servadel asuvate täisarvuliste koordinaatidega punktide arv.



Joonisel on $a = 8$ punkti hulknurga sees (rohelised) ja $b = 7$ punkti hulknurga servadel (punased), seega on selle hulknurga pindala $8 + 7/2 - 1 = 10{,}5$ ruutühikut.

12.3.3 Punkti sisaldumine hulknurgas

Uurime, millal sisaldub antud punkt $P(x, y)$ hulknurgas. Kui tõmmata kiir punktist P paralleelselt horisontaalse teljega, saab loendada, mitu korda selline kiir läbib mõnd hulknurga külge. Kui läbimisi on paaritu arv, asub punkt hulknurga sees, kui aga paarisarv, siis hulknurgast väljapool. Hulknurga servadel asuvad punktid loetakse hulknurga sees olevateks.



Tekitame punktist lõigu, mille alguspunktiks on punkt ise ja lõpp-punktiks punkt, mille y-koordinaat võrdub punkti y-koordinaadiga ja x-koordinaat on võimalikult kaugel (lõpmatuses). Tegelikult piisab, kui see on kaugemal kui kõige suurema x-koordinaadiga hulknurga tipu x-koordinaat. Edasi kontrollime iga hulknurga küljega, kas see lõikub punktist tõmmatud lõiguga ning loendame, mitme küljega lõikumine toimus.

```
bool on_sees(Point polygon[], int n, Point p)
{
	if (n < 3)  return false; //hulknurgal on vähemalt 3 tippu

	// p-st kiire lõpp-punkt(lõpmatuses)
	Point extreme = { INF, p.y };

	// loendame lõikumised külgedega
	int count = 0, i = 0;
	do {
		int next = (i + 1) % n;

		// kas kiir puntist p lõikab mõnd külge
		// 'polygon[i]' st kuni 'polygon[next]'-ini
		if (doIntersect(polygon[i], polygon[next], p, extreme)) {
			// Kui on kollineaarne, siis kas on serval?
			if (orientation(polygon[i], p, polygon[next]) == 0)
				return onSegment(polygon[i], p, polygon[next]);

			count++;
		}
		i = next;
	} while (i != 0);

	return count & 1;  // =(count%2 == 1)
}
```

12.4 KOORDINAATIDE PAKKIMINE

On antud NxN ruudustik ($1 \le N \le 10^9$). Ruudustikus on M ($1 \le M \le 100$) takistust, mille laius on 1 ruut ja pikkus 1 kuni K ruutu. Kõik takistused on asetatud kas vertikaalselt või horisontaalselt. Ruudustikus töötab robottolmuimeja, mis asub alguses ruudul S koordinaatidega x ja y. Leia, mitu ruutu saab robot puhastada. Robot takistusi ei suuda ületada.
Sisendi esimesel real on kaks arvu N ja M. Teisel real on kaks täisarvu: roboti stardipositsiooni koordinaadid. Järgmisel M real on neli täisarvu: iga takistuse alguse ja lõpu koordinaadid.
Väljastada üks täisarv: ruutude arv, mida robot saab puhastada (k.a ruut, kus ta alguses on).
NÄIDE:

```
4 2
0 0
1 2 1 3
1 2 3 2
```

Vastus:

```
10
```

Näite puhul piisab muidugi, kui käsitleda ruudustikku graafina ja läbida see laiuti. Kuid mida peale hakata, kui ruudustik on maksimaalse lubatud suurusega?

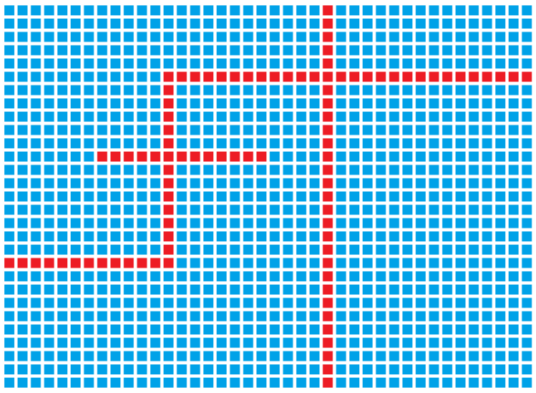
Paneme tähele, et kuigi ruudustik on suur, on takistusi seal hõredalt. Enamik read on täpselt samasugused, seega ei pea iga rida üksikult töötlema. Selle asemel saab kasutada meetodit, mida nimetatakse **koordinaatide pakkimiseks** (i.k. *coordinate compression*)

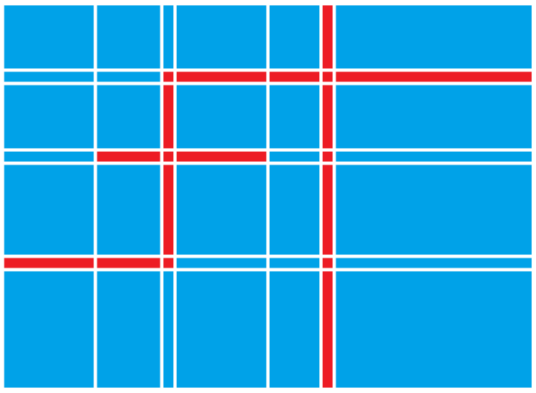
Selle meetodi peamine idee seisnebki selles, et enamik read (või veerud) on tegelikult samasugused, aga ülesande seisukohalt olulist infot sisaldavad ainult need read, mis on eelmisest erinevad.

Antud ülesandes on järgmine rida eelmisest erinev ainult siis, kui takistus algab või lõpeb vaadeldaval või sellele eelneval real. Kuna takistusi on vaid 100, on võimalikke algus ja lõpp-punkte vaid paarsada. Sama kehtib ka veergude jaoks.

Hoiamegi alles meid huvitavad read ja veerud, st esimese ja viimase ning iga takistuse otspunktile vastava rea ning samuti sellele eelneva ja järgneva rea. See teeb kokku maksimaalselt 602 rida. Iga rea kohta hoiame meeles ka, mitut rida see esindab (mitu samasugust rida sellele järgneb). Sama teeme ka veergudega ning saame 602x602 ruudustiku, mida on juba võimalik sügavuti või laiuti läbida.

Vastuse leidmiseks peame arvestama, et iga ruut (R, V) vastab nii mitmele ruudule, kui mitut rida R ja veergu V see ruut esindab: kui rida R esindab N_r ruutu ja veerg V N_v ruutu, siis (R, V) esindab $N_r \times N_v$ ruutu.





12.4.1 Ülesanne: Tikuruudud

See ülesanne on Eesti informaatikaolümpiaadi lõppvoorust aastast 2014 (autoriks Avo Muromägi).

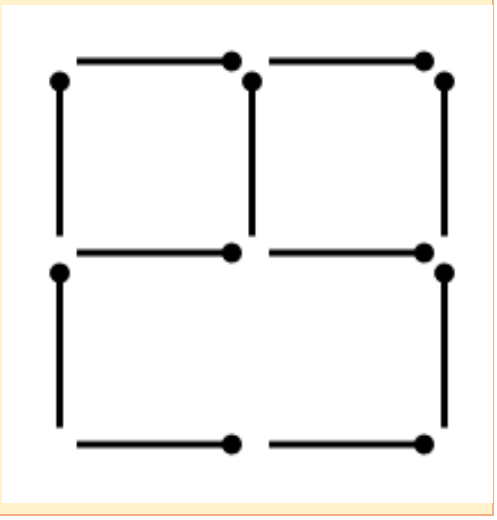
Kirjutada programm, mis saab tikkude paigutuse laual ja loendab, mitu ruutu need tikud kokku moodustavad. Näiteks alloleval joonisel kujutatud tikud moodustavad kokku 3 ruutu (kaks ruutu suurusega 1×1 ja ühe ruudu suurusega 2×2).
Sisendi esimesel real on tikkude arv `N` (`1≤N≤1000`). Järgmisel `N` real on igaühel ühe tiku asend kujul `X Y S`, kus `X` ja `Y` on tiku väävlita otsa täisarvulised koordinaadid (`0≤X≤1000, 0≤Y≤1000`) ja `S` selle suund. Suund võib olla kas `H`, mis tähendab, et tiku väävliga ots on suunatud paremale, või `V`, mis tähendab, et väävliga ots on suunatud üles. Võib eeldada, et joonisel pole kuskil mitu tikku üksteise peal.
Väljundi ainsale reale väljastada moodustuvate ruutude arv.

NÄIDE:

```
11
0 0 H
1 0 H
0 0 V
2 0 V
0 1 H
1 1 H
0 1 V
1 1 V
2 1 V
0 2 H
1 2 H
```

Vastus:

```
3
```



Kuna ruudustik on küllaltki suur, siis tasub tähele panna, et ruute ei tasu otsida koordinaatidelt, kus ei asugi tikke. Veelgi enam - iga ruudu vasakust alumises nurgas peab olema nii horisontaalse kui ka vertikaalse tiku väävlita ots.

Kaval ongi juba andmete sisse lugemisel jagada tikud kaheks - horisontaalsed ja vertikaalsed - ning hoida neid eraldi loendites. Sorteerime horisontaalsed tikud esmalt y-koordinaadi järgi, võrdsete y-koordinaatide korral x-koordinaadi järgi. Vertikaalsete tikkude puhul toimime vastupidi: sorteerime esmalt x-koordinaadi, teiseks y-koordinaadi järgi.

Nüüd saab alustada ruutude otsimisega. Selleks käime üle horisontaalsete tikkude loendi. Iga horisontaalse tiku korral vaatame, kas leidub vertikaalne tikk, mis asub samadel koordinaatidel. Kuna tikud on järjestatud, saab seda teha kahendotsinguga. Kui sellist vertikaalset tikku ei ole, siis võtame vaatluse alla järgmise horisontaalse tiku. Kui aga on olemas vastavate koordinaatidega vertikaalne tikk, siis võivad need kaks tikku olla ühe või mitme ruudu alumiseks vasakuks nurgaks. Sellisel juhul asume loendama tekkivaid ruute.

```
int loenda_ruudud() {
     int vastus = 0;
     for (int i = 0; i < hor_tikud.size(); i++)
          vastus += loenda_ruudud_tikust(i);
     return vastus;
}
```

Kui on olemas ka teised kaks tikku, nii et moodustub ruut küljepikkusega 1, siis suurendame vastust ühe võrra. Sõltumata eelmisest sammust kontrollime, kas horisontaalse tiku väävliga otsas asub horisontaalne tikk ning teeme sama ka vertikaalse tikuga. Mõlemal juhul peab see tikk olema vastavas loendis järgmine, see tähendab, et kui me hetkel vaatasime horisontaalset tikku indeksiga i, siis kontrollime, kas horisontaalne tikk indeksiga $i + 1$ on sama y-koordinaadi ja ühe võrra suurema x-koordinaadiga. Kui on olemas nii vertikaalne kui ka horisontaalne tikk, mis sobib külgi pikendama, siis kontrollime, kas moodustub ruut küljepikkusega 2. Sama protseduuri jätkame seni, kuni jõuame olukorda, kus horisontaalse või vertikaalse tiku väävliga otsa juurde pole sama suunaga tikku asetatud. Sellisel juhul võib loendamise katkestada, sest suuremaid ruute enam samast alguskohast moodustuda ei saa, ja edasi liikuda järgmise horisontaalse tiku juurde.

```
int loenda_ruudud_tikust(int tiku_nr) {
      int   kylje_pikkus;
      int vasak_indeks, alumine_indeks;
      pair<int, int> vasak_tikk, alumine_tikk;
      int vastus = 0;
      pair<int, int> tikk = hor_tikud[tiku_nr];
      kylje_pikkus = 1;
      alumine_indeks = tiku_nr;
      vasak_indeks = otsi_tikk(make_pair(tikk.second, tikk.first), ver_tikud);
      if (vasak_indeks == -1) return vastus; //Selle algkoordinaadiga vertikaalset
// tikku pole
      while ((alumine_indeks < hor_tikud.size()) && (vasak_indeks < ver_tikud.size()))
{
            alumine_tikk = hor_tikud[alumine_indeks];
            if ((alumine_tikk.second - kylje_pikkus + 1 != tikk.second) ||
(alumine_tikk.first != tikk.first)) break;
            vasak_tikk = ver_tikud[vasak_indeks];
            if ((vasak_tikk.first != tikk.second) || (vasak_tikk.second - kylje_pikkus
+ 1 != tikk.first)) break;

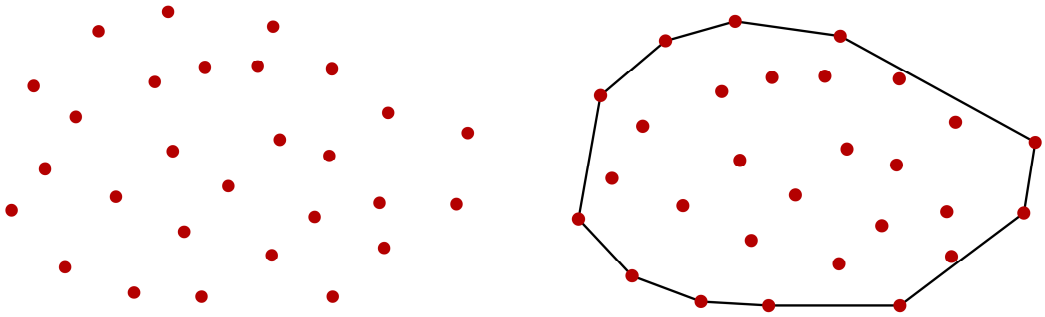
            if (on_hor_yhendus(tikk.second, tikk.second + kylje_pikkus - 1, tikk.first
+ kylje_pikkus) && // ülemine külg
                  on_ver_yhendus(tikk.second + kylje_pikkus, tikk.first, tikk.first +
kylje_pikkus - 1)) // parem külg
                  vastus++; //ruut leitud!
            kylje_pikkus++; //proovime suuremat
            alumine_indeks++;
            vasak_indeks++;
      }
      return vastus;
}
```

Praegu on veel kirjeldamata, kuidas teha kindlaks, et moodustub ruut küljepikkusega p? Tänu eelpool kirjeldatud viisile me teame, et (potentsiaalse) ruudu vasakus ja alumises küljes on kummaski p tikku olemas. Jääb veel ainult kontrollida, kas paremas ja ülemises küljes on kõik tikud olemas. Siin on oluline märgata, et iga tiku olemasolu kontroll polegi vajalik. Veendumaks, et eksisteerib p tikust moodustatud külg, piisab tänu järjestatud tikkude loendile kontrollist, kas on olemas külje esimene ja viimane tikk ning kui need on olemas, siis nende tikkude indeksite vahe peab olema $p - 1$. Indeksid saab leida taas kahendotsingut kasutades.

```
bool on_hor_yhendus(int X1, int X2, int Y) {
      int   indeks1, indeks2;
      indeks1 = otsi_tikk(make_pair(Y, X1), hor_tikud);
      indeks2 = otsi_tikk(make_pair(Y, X2), hor_tikud);
      return (indeks1 >= 0) && (indeks2 >= 0) && (indeks2 - indeks1 == X2 - X1);
}
```

12.5 KUMER KATE

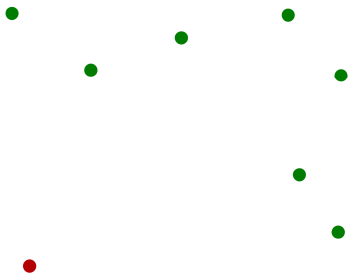
Kumer kate (i.k *convex hull*) on väikseim selline kumer hulknurk, mis sisaldab kõik etteantud punktid. Sisaldumine tähendab, et iga punkt asub kas hulknurga sees või mõnel selle küljel.



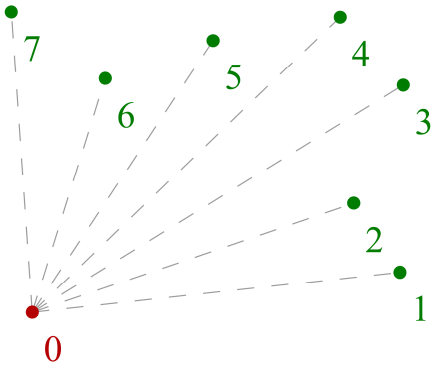
Jõumeetodil võiks käia selline otsimine nii, et kontrollime iga punktipaari korral, kas selle vahele tõmmatud lõik saab olla hulknurga küljel. Kui saab, siis peavad kõik ülejäänud punktid asuma sirgest, mis läbib vaadeldavat kahte punkti, ühel pool. n punktist saab moodustada n^2 paari ja iga paari jaoks tuleb kontrollida veel $n-2$ punkti asukohta, on sellise naiivse lähenemise keerukuseks $O(n^3)$.

Kumera katte leidmiseks on olemas mitmeid paremaid algoritme. Üks neist on **Grahami seire**. Selle idee on järgmine:

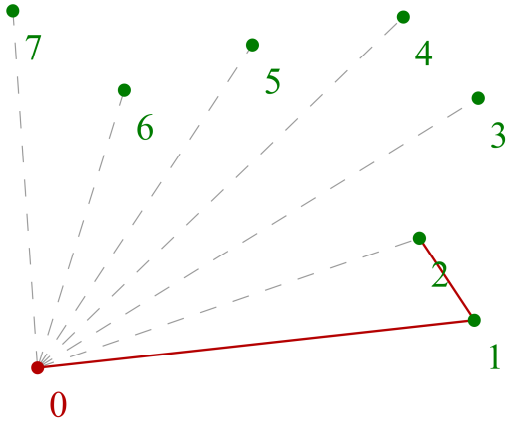
1. Leia kõige väiksema y-koordinaadiga punkt. Kui sama y-koordinaadiga on mitu punkti, vali neist see, mille x-koordinaat on kõige väiksem. Nimetame seda punkti vaatlus- ehk seirepunktiks (joonisel punane):

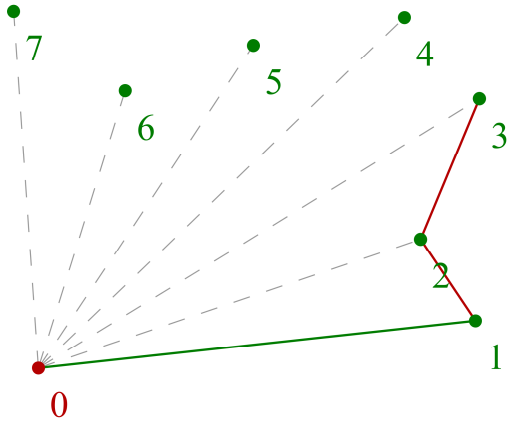


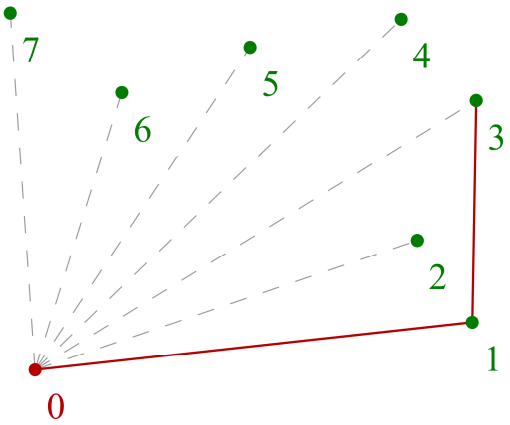
   Sea see punkt punktiloendis esimeseks ja kuulub ise kindlasti kumerasse kattesse.
2. Ülejäänud $n-1$ punkti jaoks leia seirepunkti ja vaadeldavat punkti läbiva sirge tõusunurk. Sorteeri punktid leitud tõusunurga järgi vastupäeva.

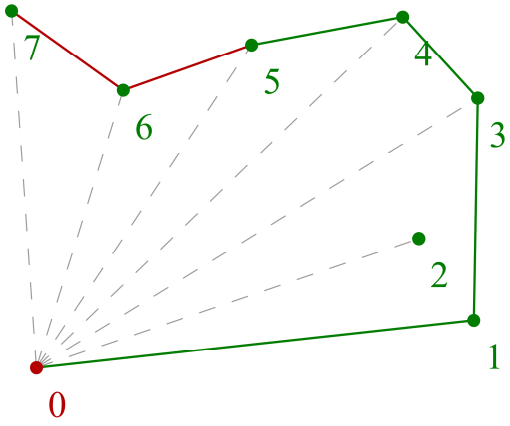


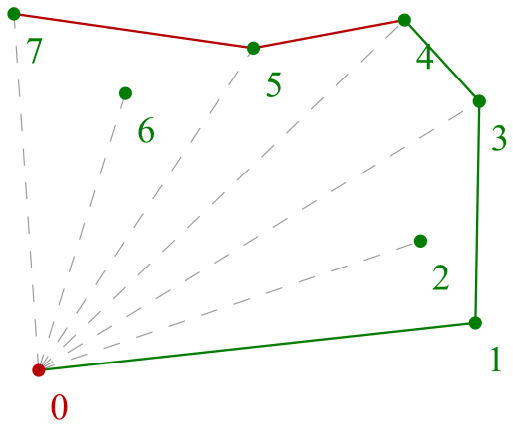
3. Kõigist sama tõusunurgaga punktidest jäta alles ainult see, mis on valitud punktist kõige kaugemal, sest vaid see saab kuuluda kumerasse kattesse.
4. Kui järele jääb vähem kui kolm punkti, siis kumerat katet ei eksisteeri (punktid asuvad samal sirgel).
5. Edasi tuleb iga punkti p_i jaoks otsustada, kas see kuulub kumerasse kattesse või mitte. Kumer hulknurk saab moodustuda ainult sellistest punktidest, mille vahele tõmmatud servad keeravad eelmise serva suhtes rangelt vastupäeva. Seega
    a. kui toimub pööre vastupäeva, jäta punkt p_i kumerasse kattesse ja korda 4. sammu punkti p_{i+1} jaoks.
    b. Vastasel juhul eemalda kattest **eelmine** punkt p_{i-1} ja korda 4. sammu.

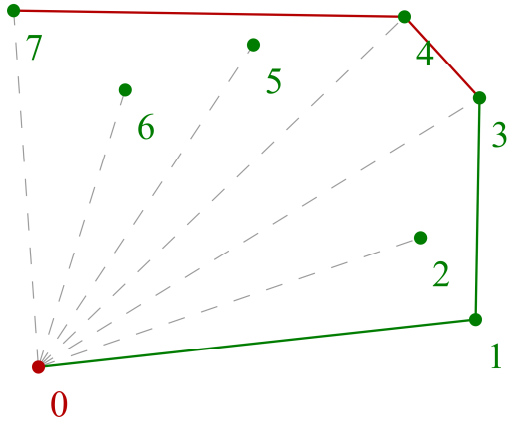




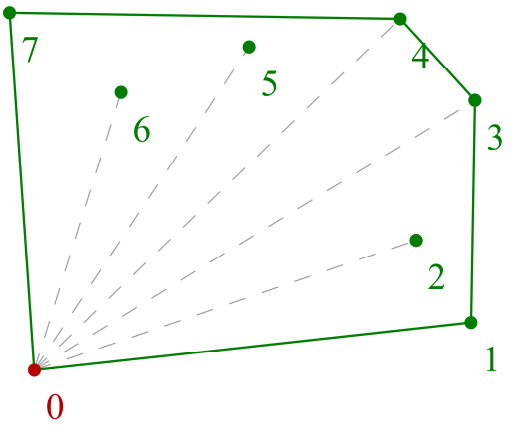








Joonisel esiteks vaatame pööret punktikolmikul (0,1,2). pööre on vastupäeva ja punktid jäävad kattesse. Edasi vaatame pööret (1,2,3), mis on aga päripäeva. Seega punkt 2 kattesse siiski ei kuulu, eemaldame selle kattest. Nüüd tuleb kontrollida kolmikut (0, 1, 3), mis on ilusti vastupäeva. Edasi on kenasti vastupäeva pöörded, kuni jõuame punktideni (5,6,7). Kuna seal on pööre päripäeva, eemaldame punkti 6 kattest. Aga ka pööre (4, 5, 7) on vales suunas, seega tuleb kattest eemaldada ka punkt 5. Kumera katte moodustavad punktid 0, 1, 3, 4 ja 7.



Siin on ka protseduur, mis leiab kumera katte:

```
void leia_kumer_kate(Point punktid[], int n)
{
    // vasak alumine punkt
    int ymin = punktid[0].y, min = 0;
    for (int i = 1; i < n; i++) {
        int y = punktid[i].y;
        if ((y < ymin) || (ymin == y && punktid[i].x < punktid[min].x))
            ymin = punktid[i].y, min = i;
    }

    //vaheta vasak alumine punkt esimeseks
    swap(punktid[0], punktid[min]);

    // sorteeri punktid tõusunurga järgi
    p0 = punktid[0];
    qsort(&punktid[1], n - 1, sizeof(Point), compare);

    // hoiame sobivaid punkte pinus. Alustame esimese kolme punktiga
    stack<Point> S;
    S.push(punktid[0]);
    S.push(punktid[1]);
    S.push(punktid[2]);

    for (int i = 3; i < n; i++) {
        // eemalda pealmine punkt,kuni õige suunani
        while (leia_suund(eelmine(S), S.top(), punktid[i]) != 2)
            S.pop();
        S.push(punktid[i]); //suund õige, lisa punkt
    }

    // nüüd on pinus kumera katte moodustavad punktid
    while (!S.empty()) {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}
```

Grahami seiremeetodi keerukuseks on $O(n\ log\ n)$.

12.6 Ülesanne: ring ja ruut

See ülesanne oli 1997. aastal Balti olümpiaadil. Sõnastuse poolest on ülesanne ülilihtne, seda saab kirjeldada kõigest paari lausega:

On antud ring ja ruut oma koordinaatidega. Leida nende kattuva osa pindala.
Sisendi esimesel real on kolm täisarvu: ringi keskpunkti koordinaadid ja ringi raadius. Teisel real on 8 täisarvu: ruudu tippude koordinaadid.
Väljundi ainsale reale kirjutada üks reaalarv: ringi ja ruudu kattuva osa pindala.
***NÄIDE 1*:**

```
0 0 1
0 0 0 2 2 0 2 2
```

Vastus:

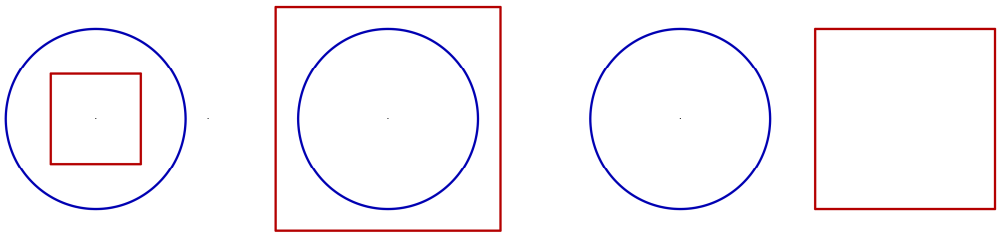
***NÄIDE 2*:**

```
0 0 1
0 -2 0 2 4 -2 4 2
```

Vastus:

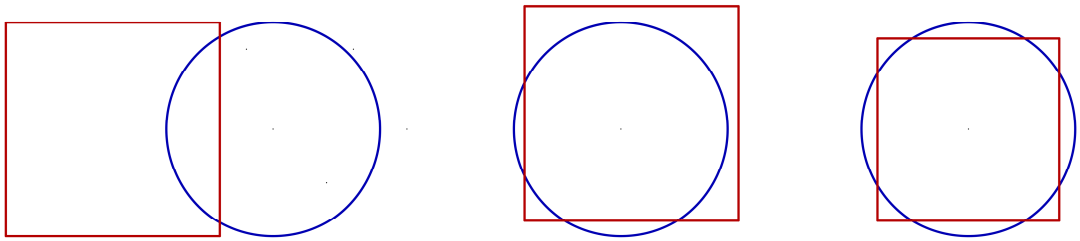
Algselt arvas ka ülesande väljamõtleja Indrek Jentson, et kui raske see ülesanne ikka olla saab, kuid erinevad testjuhud lähevad väga kiiresti käest ära. Uurime neid korraks, tegu on ka kasuliku testimisalase harjutusega.

Kõige lihtsamad ringi ja ruudu omavahelise paiknemise võimalused on järgmised kolm:



- Ruut paikneb tervenisti ringi sees. Kattuva osa pindala on ruudu pindala.
- Ring paikneb tervenisti ruudu sees. Kattuva osa pindala on ringi pindala.
- Ruudul ja ringil ühine osa puudub. Kattuva osa pindala on null.

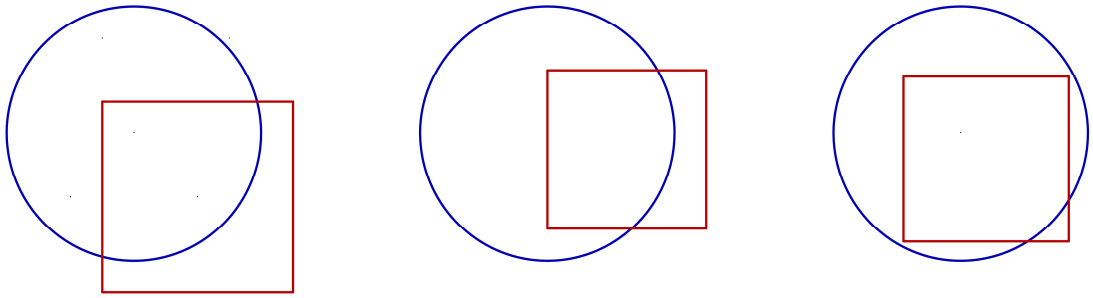
Lisaks on muidugi ka keerulisemaid võimalusi, näiteks asetused, kus ükski nurk ei ole ringi sees ja:



- Ring lõikab ruudu ühte külge
- Ring lõikab ruudu kaht külge
- Ring lõikab ruudu kõiki külgi

Neil juhtudel tuleb leida segmendi pindala ja otsustada, kas segment on ringi sees või sellest väljas.

Ruudu nurgad aga võivad asetseda ka ringi sees:



- Üks nurk ringi sees
- Kaks nurka ringi sees
- Kolm nurka ringi sees

Täiendavalt tekivad erijuhud, kui ring lihtsalt puudutab ruudu külge või tippu. Kiiresti saab selgeks, et **geomeetriline lähenemine** – proovida iga erijuhu jaoks leida vastavad geomeetrilised valemid – läheb õige pea käest ära: lahendamine on võimalik, aga lahenduse kirjapanek võtaks tõenäoliselt sadu ja sadu ridu koodi ning mitu päeva aega.

Kui jooniste üle natuke mõtiskleda, võib järgmiseks pähe tulla **analüütiline lähenemine**. Paneme tähele, et:

1. Geomeetrilise kõvera alla jääva ala pindala saab leida integraaliga.
2. Ruudu küljed annavad meile enamasti integreerimiseks sobilikud rajad.
3. Pöörates joonise kõigepealt sobivat pidi ning seejärel ringjoone valemit $y = \sqrt{r^2 - x^2}$ vastavate radadega integreerides saamegi vastuse. Integreerimiseks vajalikud vihjed võib leida näiteks siit: https://www.quora.com/How-can-I-integrate-to-find-the-area-of-a-circle

See lahendus on juba parem, kuid ka selle realisatsioon võttis mul endal pea terve päeva.

Selgub aga, et selliste ülesannete puhul on sageli võimalik ka **programmeerija lähenemine**, kus me paneme tähele, et üldjuhul pole vaja leida absoluutse täpsusega vastust ning piisab sellest, kui me leiame vastuse **lähendamise** teel.

Algoritmi idee on järgmine:

1. On väga lihtne kontrollida, kas ruut on **täielikult ringi sees** (sel juhul liidame vastusele ruudu pindala) või **täielikult ringist väljas** (sel juhul ei liida vastusele midagi).
2. Kui ruut kattub ringiga osaliselt, jagame selle **rekursiivselt neljaks osaks** ja kordame arutlust iga osa jaoks, kuni oleme saavutanud sobiva täpsuse.

Realisatsiooni juurde minnes on kõigepealt kasulik koordinaadid mugavamaks teisendada, siis on edasises arvutused lihtsamad. Esiteks võib nihutada koordinaate nii, et ringi keskpunkt oleks koordinaatidega (0, 0). Selleks ei tule teha midagi muud, kui sisse lugedes lahutada igast ruudu x-koordinaadist ringi x-koordinaat ja igast ruudu y-koordinaadist ringi y-koordinaat.

Teiseks on kasulik pöörata sisendit nii, et ruudu küljed on paralleelsed koordinaattelgedega. See on lihtne teisendus ja teeb edaspidi programmeerija elu palju kergemaks:

```
void keera(pair<double, double> ruut[4]) {
	if (ruut[0].second - ruut[2].second == 0) return; //on juba õige
	double tous = (ruut[0].first - ruut[2].first)/(ruut[0].second - ruut[2].second);
	tous = atan(tous);
	for (int i = 0; i < 4; i++) {
		double x = ruut[i].first;
		double y = ruut[i].second;
		ruut[i].first = x*cos(tous) - y*sin(tous);
		ruut[i].second = x*sin(tous) + y*cos(tous);
	}
	return;
}
```

Edasi vaatame läbi esimesed kolm triviaalset juhtu. Ring on ruudu sees, kui kaugus ringi keskpunktist iga servani on suurem või võrdne ringi raadiusega. Seda on lihtne kontrollida:

```
bool ring_ruudus(pair<double, double> ruut[4]) {
	if (r*(-1) >= ruut[0].first &&
		r <= ruut[3].first &&
		r*(-1) >= ruut[0].second &&
		r <= ruut[3].second)
		return true;
	else return false;
}
```

Ruut on ringi sees, kui ükski ruudu tipp ei ole ringi keskpunktist kaugemal kui ringi raadius. Selleks on kõige kindlam lugeda üle, mitu tippu on ringis:

```
int tipp_ringis(pair<double, double> tipp) {
	double s = hypot(tipp.first, tipp.second);
	if (s <= r) return 1;
	else return 0;
}

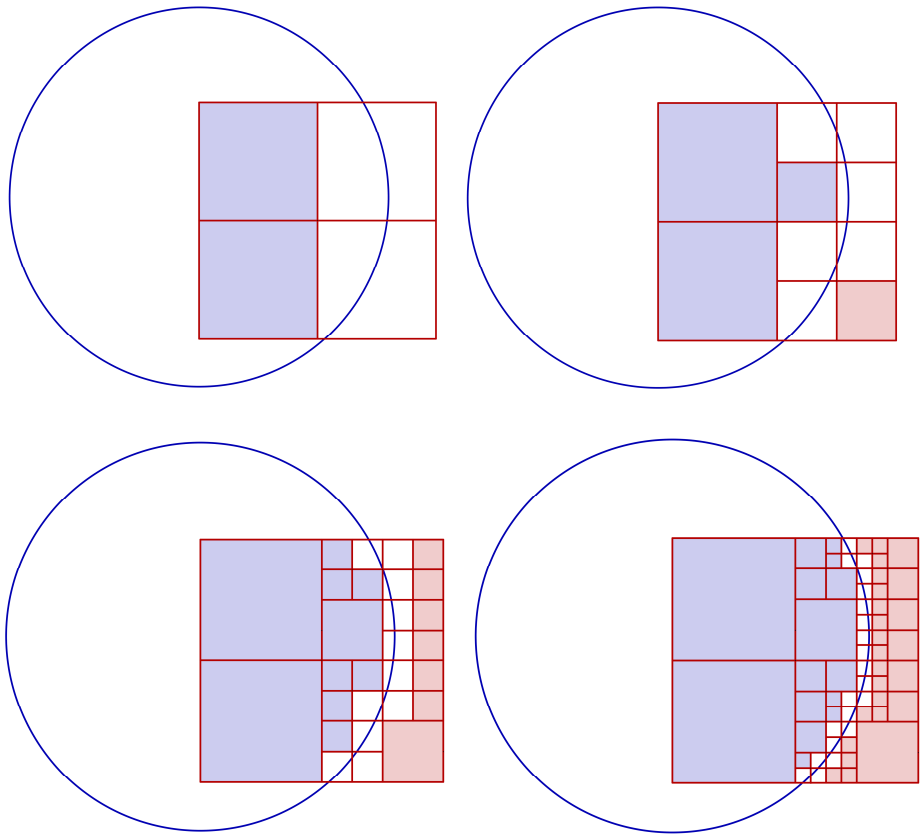
int tippe_ringis(pair<double, double> ruut[4]) {
	int summa = 0;
	for (int i = 0; i < 4; i++) {
		summa += tipp_ringis(ruut[i]);
	}
	return summa;
}
```

Kahjuks see, kui kõik tipud on ruudust väljas, ei garanteeri veel, et ringi ja ruudul kattuvat osa ei ole. Seega otsustamaks, et ruut on tervenisti väljaspool ringi, tuleb seda veel eraldi kontrollida.

```
bool on_valjas(pair<double, double> ruut[4])
{
	if (ruut[0].first >= r || ruut[3].first <= 0 - r ||
		ruut[0].second >= r || ruut[3].second <= 0 - r)
		return true;
	double sumx=0, sumy = 0, abssumx=0, abssumy = 0;
	for (int i = 0; i < 4; i++) {
		sumx += ruut[i].first;
		sumy += ruut[i].second;
		abssumx += abs(ruut[i].first);
		abssumy += abs(ruut[i].second);
	}
	if (abs(sumx) == abssumx && abs(sumy) == abssumy) return true;

	return false;
}
```

Nüüd järgneb rekursiivne osa: kui on teada, et osa ruutu on ringi sees, saame ruudu jagada neljaks väiksemaks ruuduks ja kontrollida samamoodi, kas need on täielikult ringi sees või täielikult ringist väljas või osaliselt ringis. Viimasel juhul saame jälle jagada need osaliselt sees olevad ruudud neljaks jne, kuni oleme saavutanud piisava täpsuse.



Lillad ruudud on täielikult ringi sees, roosad täielikult ringist väljas, valged need, mida vajadusel tuleb edasi jagada.

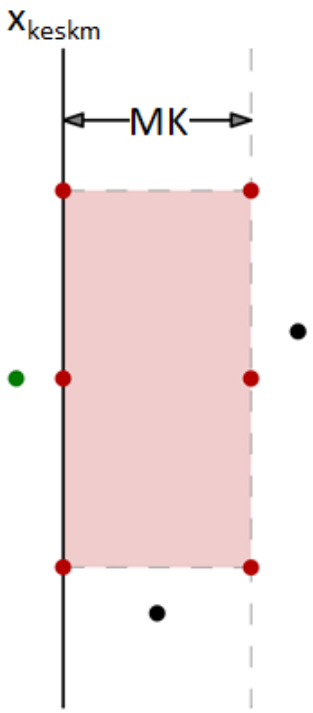
```
double leia_kattuv_pindala(pair<double, double>ruut[4])
{
	double serva_pikkus = abs(ruut[3].first - ruut[0].first);
	if (serva_pikkus < eps) return 0;
	double ruudu_pindala = serva_pikkus*serva_pikkus;
	int tipud_sees = tippe_ringis(ruut);
	if (tipud_sees == 4) return ruudu_pindala;
	if (tipud_sees == 0 && on_valjas(ruut)) return 0;
	double keskm_x = ruut[0].first + (serva_pikkus / 2);
	double keskm_y = ruut[0].second + (serva_pikkus / 2);
	pair<double, double> ruut1[4], ruut2[4], ruut3[4], ruut4[4];
	tee_ruut(ruut[0].first, ruut[0].second, keskm_x, keskm_y, ruut1);
	tee_ruut(ruut[0].first, keskm_y, keskm_x, ruut[3].second, ruut2);
	tee_ruut(keskm_x, ruut[0].second, ruut[3].first, keskm_y, ruut3);
	tee_ruut(keskm_x, keskm_y, ruut[3].first, ruut[3].second, ruut4);
	return leia_kattuv_pindala(ruut1) +
		leia_kattuv_pindala(ruut2) +
		leia_kattuv_pindala(ruut3) +
		leia_kattuv_pindala(ruut4);
}
```

12.7 LÄHIM PUNKTIPAAR

Üks klassikaline arvutusgeomeetria ülesanne on veel punktihulgast omavahel lähima punktipaari leidmine. Naiivne lähenemine oleks muidugi leida kõik kaugused kahe punki vahel. Selle keerukus aga on $O(n^2)$.

Kasutades aga "jaga ja valitse" tüüpi lähenemist, saame algoritmi, mis töötab $O(n\ log\ n)$ ajaga. Selleks

1. sorteerime punktid x-koordinaadi järgi
2. jagame punktid kahte võrdse suurusega alamhulka vertikaalse joone $x = x_{keskm}$
3. leia lähim paar kummaski alamhulgas kasutades sama võtet (rekursiivselt). Selle tulemusena saame vasaku ja parema poole minimaalse kauguse (MK_v ja MK_p).
4. leia vähim kaugus sellise punktipaari vahel, millest üks asub jaotusjoonest vasakul ja teine paremal
5. Vastuseks on minimaalne kaugus neist kolmest.



Tundub ilus, aga kas punkt 4 juures ei ole me sarnases lõksus, mis alguseski? Siin aga saab ära kasutada asjaolu, et kuna meid huvitab lõpuks vaid minimaalne kaugus, siis pole mõtet punkte kaugemalt vaatama hakata, kui seni leitud vähim kaugus ehk $\min(MK_v, MK_p) = MK$. Nii on vaja vaadelda vaid punkte, mis asuvad ristkülikus mõõtmetega $MK \times 2MK$. Lisaks ei saa see ristkülik sisaldada rohkem kui 6 punkti, sest muidu ei oleks MK meil vähim seni leitud kaugus! Endiselt tuleb kogu protseduur teha läbi iga punkti jaoks, aga maksimaalselt teeme n^2 võrdlemise asemel vaid $6n$ võrdlemist.

```
// Px on punktid sorteerituna x koordinaadi järgi ja
//Py punktid sorteerituna y koordinaadi järgi
float leia_lahim_paar(Point Px[], Point Py[], int n)
{
	// kuni 3 punktiga lihtsalt võrrelda:
	if (n <= 3)
		return leia_jouga(Px, n);

	// keskmine punkt x järgi
	int mid = n / 2;
	Point midPoint = Px[mid];

	Point Pyl[mid + 1];
	Point Pyr[n - mid - 1];
	int li = 0, ri = 0;
	for (int i = 0; i < n; i++)
	{
		if (Py[i].x <= midPoint.x)
			Pyl[li++] = Py[i];
		else
			Pyr[ri++] = Py[i];
	}

	float dl = leia_lahim_paar(Px, Pyl, mid);
	float dr = leia_lahim_paar(Px + mid, Pyr, n - mid);

	float d = min(dl, dr);

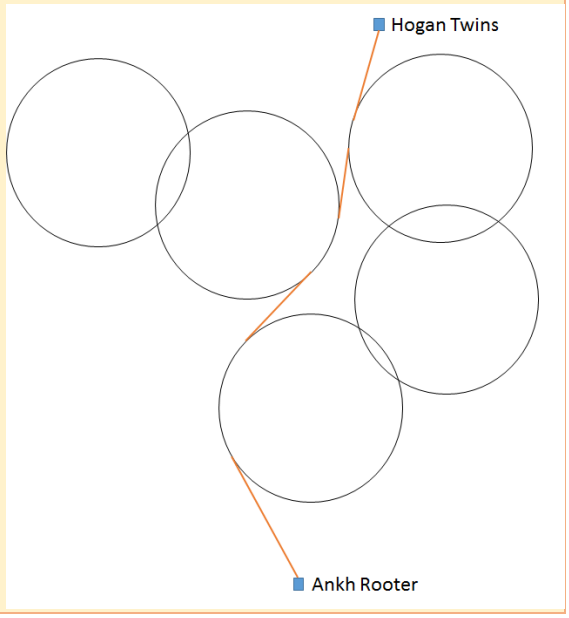
	Point strip[n];
	int j = 0;
	for (int i = 0; i < n; i++)
		if (abs(Py[i].x - midPoint.x) < d)
			strip[j] = Py[i], j++;

	return min(d, stripClosest(strip, j, d));
}
```

12.8 MAGIC MISSILE (ÜLESANNE)

Järgmise ülesande saatsime Balti olümpiaadile aastal 2016, kuid see jäi lõppvalikust välja kui liiga raske, s.t arvati, et enamik võistlejaid ei suudaks seda ettenähtud aja jooksul lahendada. Ülesande idee autor oli Oliver-Matis Lill, boonus läheb neile lugejaile, kes siit ajastukohase poliitilise olukorra viited üles leiavad.

Kettamaailm on tuntud kahe asja poolest: see asub lamedal tasapinnal, mitte kera pinnal, nagu meie maailm ja see jaguneb mitmeks võistlevaks riigiks.
Üks neist riikidest on Ankh Rooter, mida valitseb sõjakas diktaator Gin Junk Om, kes on ehitanud maagilise raketi, mida saab kasutada teiste hävitamiseks. Kõige rohkem vihkab Gin Junk Om võlur Kabob Maracat, kes elab linnas nimega Hogan Twins.
Kabob Maracal on mitmeid liitlasi Kettamaailmas ja ta on neile saatnud maagilise raketi vastase kilbi generaatorid. Kui maagiline rakett jõuab generaatorile lähemale kui R miili, kukub rakett ohutult alla. Gin Junk Om aga teab kilbigeneraatorite asukohti ning tahab, et sa arvutaks, kui mitu thaumi (võlujõu mõõtühik) võlujõudu peab raketti laadima. Selleks pead leidma lühima tee Ankh Rooter-ist Hogan Twinsi, mis väldib kõiki kilbigeneraatoreid.
Sisendi esimesel real on neli täisarvu: Gin Junk Omi võlutorni koordinaadid ja Kabob Maraca võlutorni koordinaadid. Teisel real on kaks täisarvu: generaatori ulatus R ja generaatorite arv N (1 <=N<=1000). Järgneval N real on igaühel kaks täisarvu: ühe generaatori koordinaadid.
Väljundisse kirjuta üks arv: vajalik võlujõud. Kui raketti lennutada ei saa, väljastada -1.
NÄIDE:

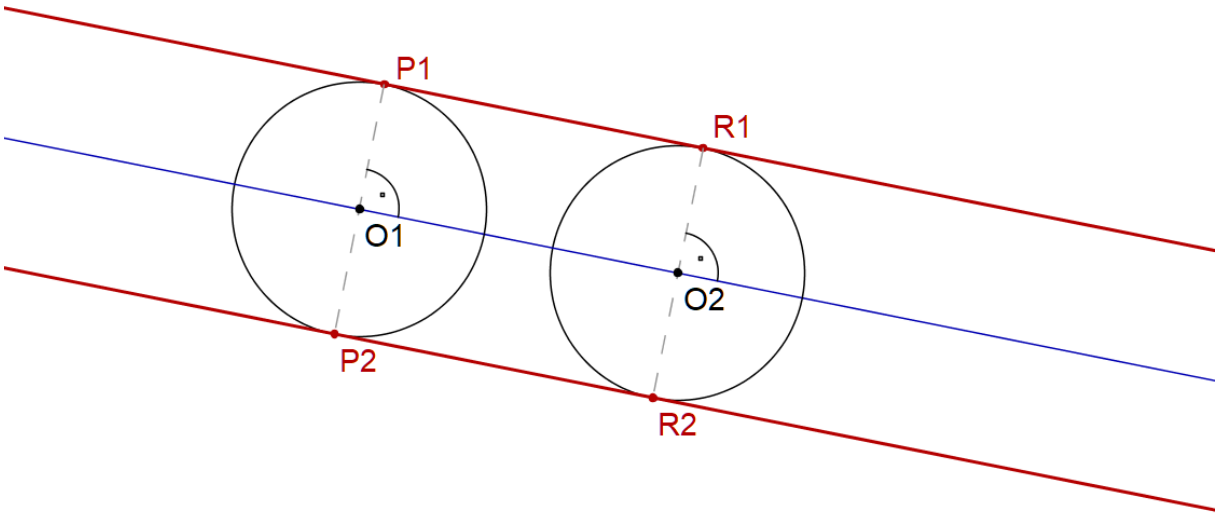


Selleks, et võimalikult otse lennata, peab rakett liikuma kas mööda kahe ringi ühist puutujat või mööda ringjoont. Algus ja lõppmarsruut on mööda puutujat, mis algab/lõpeb tornist.

Lühima tee leidmiseks tulebki leida kõik mõistlikud puutujad ja kaared ning moodustada nendest graaf, millest lühima tee leidmiseks saab kasutada Dijkstra algoritmi.

Kõlab väga lihtsalt, kuid tegelikult on sobivate puutujate ja kaarte leidmine omajagu nikerdamist vajav ettevõtmine.

Kahe ringi vahel võib olla kuni neli puutujat. Kui ringid puutuvad või lõikuvad, siis on puutujaid vaid kaks. Iga leitud puutuja juures tuleb kontrollida, ega see ei lõika mõnd muud ringi. Samuti ei tohi ära unustada algus ja lõpp-punkti ning võimalust, et need on omavahel otse ühendatavad.




```
void leia_puutujad()
{
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            double rdx = x[j] - x[i];
            double rdy = y[i] - y[j];
            double vn = atan2(rdy, rdx);
            double mp = sin(vn) * r;
            double mu = cos(vn) * r;
            lisa_nihked(0, mp, mu, i, j);
            lisa_nihked(1, mp, mu, i, j);

            double kk = hypot(rdx, rdy) / 2;
            double un = vn - asin(r / kk);
            double ma = cos(un) * r;
            double mv = sin(un) * r;
            lisa_nihked(3, mv, ma, i, j);
            lisa_nihked(4, mv, ma, i, j);

            bool on_sisemised = (r / kk) <= 1;
            bool sobib[4] = { 1, 1, on_sisemised, on_sisemised };

            //käib kõik ringid läbi ja kontrollib, ega ei lõika ühtki ringi
            for (int k = 0; k < n; k++) {             {
                for (int m = 0; m < 4; m++) {
                    if (sobib[m] && liglah(ex[m], tx[m], ey[m], ty[m], x[k], y[k])) {
                        sobib[m] = 0;
                    }
                }
            }
            for (int m = 0; m < 4; m++) {
                if (sobib[m]) {
                    pv.push_back(make_pair(make_pair(i, j), make_pair(make_pair(ex[m],
ey[m]), make_pair(tx[m], ty[m]))));
                    pv.push_back(make_pair(make_pair(j, i), make_pair(make_pair(tx[m],
ty[m]), make_pair(ex[m], ey[m]))));
                }
            }
        }
    }
```

```
    //algus ja lõpppunkti jaoks puutujad
    for (int i = 0; i < n; i++)
    {
        leia_punkt(gx, gy, 0, i);
        leia_punkt(kx, ky, 2, i);
        bool sobib[4] = { 1, 1, 1, 1 };
        for (int k = 0; k < n; k++)
        {
            for (int m = 0; m < 4; m++) {
                if (sobib[m] && liglah(ex[m], tx[m], ey[m], ty[m], x[k], y[k])) {
                    sobib[m] = 0;
                }
            }
        }
        double px[4] = { gx, gx, kx, kx };
        double py[4] = { gy, gy, ky, ky };
        int inda[4] = { -1, -1, n, n };
        for (int m = 0; m < 4; m++) {
            if (sobib[m]) {
                pv.push_back(make_pair(make_pair(i, inda[m]),
make_pair(make_pair(xp[m], yp[m]), make_pair(px[m], py[m]))));
                pv.push_back(make_pair(make_pair(inda[m], i),
make_pair(make_pair(px[m], py[m]), make_pair(xp[m], yp[m]))));
            }
        }
    }
    // omavaheline sirge iga ringjoone jaoks
    bool o = 1;
    for (int k = 0; k < n; k++) {
        if (o && liglah(kx, gx, ky, gy, x[k], y[k])) {
            o = 0;
            break;
        }
    }
    if (o) {
        pv.push_back(make_pair(make_pair(-1, n), make_pair(make_pair(gx, gy),
make_pair(kx, ky))));
        pv.push_back(make_pair(make_pair(n, -1), make_pair(make_pair(kx, ky),
make_pair(gx, gy))));
    }
    sort(pv.begin(), pv.end());
}
```

Kaarte leidmiseks sorteerime ringjoonel asuvad puutepunktid esinemise järjekorras. Nii on vaja vaadata ainult võimalikke kaari järjestikuste punktide vahel, sealjuures ei tohi unustada võimalikku kaart esimese ja viimase punkti vahel. Ka kaarte puhul on oluline, et uuritav kaar ei lõikaks mõnd teist ringi.

```
void yhenda_kaared()
{
	int pr = -1;
	int pi = 0;
	for (int i = 1; i < np; i++) {
		int cr = pl[i].first.first.first;
		if (i == 1) {
			pr = cr;
			pi = 1;
			continue;
		}
		if (cr > pr) {
			double n1 = pl[pi].first.first.second;
			double n2 = pl[i - 1].first.first.second;
			double nv = (4 * k90) - (n2 - n1);
			double tp = nv * r;
			bool ok = 1;
			for (int j = 0; j < n; j++) {
				if (j == cr || hypot(x[j] - x[cr], y[j] - y[cr]) >= 2 * r) {
					continue;
				}
				int xv = x[j] - x[cr];
				int yv = y[j] - y[cr];
				double vn = kraad(xv, yv);
				if (vn > n2 || vn < n1) {
					ok = 0;
					break;
				}
			}
			if (!ok) continue;
			dt[pi].push_back(make_pair(i - 1, tp));
			dt[i - 1].push_back(make_pair(pi, tp));
			pr = cr;
			pi = i;
			continue;
		}
		double n1 = pl[i].first.first.second;
		double n2 = pl[i - 1].first.first.second;
		double nv = n1 - n2;
		double tp = nv * r;
		bool ok = 1;
		for (int j = 0; j < n; j++) {
			if (j == cr || hypot(x[j] - x[cr], y[j] - y[cr]) >= r) {
				continue;
			}
			int xv = x[j] - x[cr];
			int yv = y[j] - y[cr];
			double vn = kraad(xv, yv);
			if (vn > n2 && vn < n1) {
				ok = 0;
				break;
			}
		}
		if (!ok) continue;
		dt[i].push_back(make_pair(i - 1, tp));
		dt[i - 1].push_back(make_pair(i, tp));
	}
}
```

Kui leida iga ringipaari ühine puutuja, on algoritmi keerukuseks $O(N^3)$, kui aga ringid eelnevalt kauguse järgi sorteerida, saab kahandada seda $O(N^2 \log N)$-ile.

12.9 KONTROLLÜLESANDED

12.9.1 Liitumine

Juku ehitas endale uue maja. Nagu ikka, tahab Juku oma majja ka elektrit saada. Selle jaoks on vaja aga luua uus elektriliin Juku maja ning pealiini vahele. Pealiin on N lõigust koosnev murdjoon, mis läbib metsa Juku maja läheduses. Nagu ikka, maksab iga meeter ning seega on vaja leida lühima sirglõigu, mis ühendab maja ning pealiini, pikkust.
Sisendi esimesel real on kaks täisarvu, Juku maja koordinaadid. Järgmisel real on täisarv N, pealiini lõikude arv. Järgmisel N+1 real on järjest iga liini lõigu otspunktid. Enne on x-koordinaat, seejärel on y-koordinaat.
Väljundisse kirjutada kaks arvu: lähima punkti x ja y koordinaat.

NÄIDE 1:

```
6 -3
3
0 1
5 5
9 -5
15 3
```

Vastus:

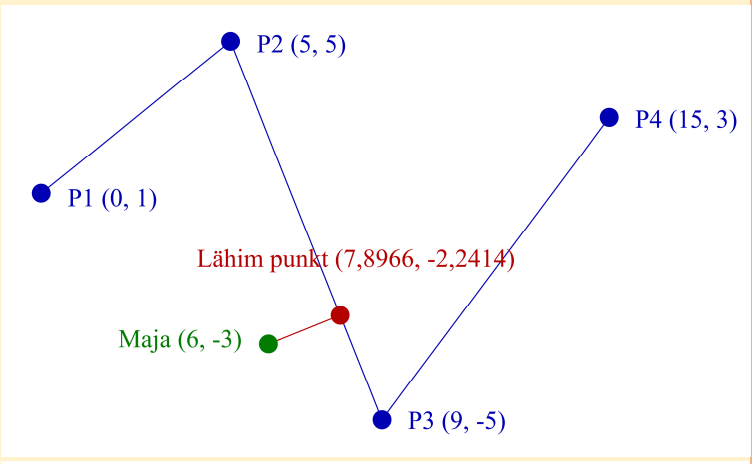
```
7,8966 -2,2414
```

NÄIDE 2:

```
0 0
1
1 0
2 0
```

Vastus:

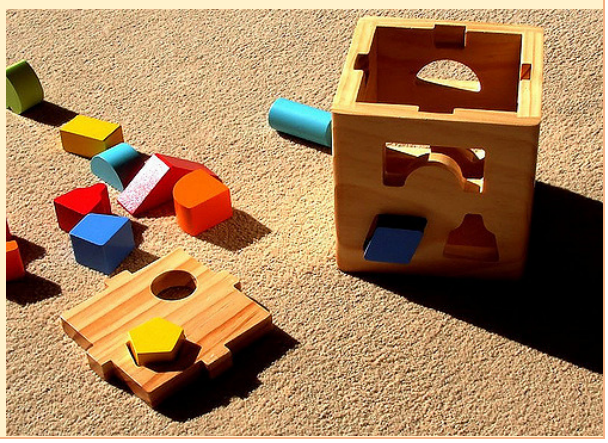
```
1.0000 0.0000
```



Esimesele näitele vastav joonis

12.9.2 Augumäng

Väike Kadi üritab kolmnurki toppida läbi ringikujuliste aukude. Kolmnurk on antud mõistes perfektne kolmnurkne püstprisma, mille pikkus on suurem kui mistahes ringi diameeter. Väikesel Kadil on N (1 ≤ N ≤ 26) kolmnurka ning M (1 ≤ M ≤ 100) ringi. Aita teda ning ütle, missuguseid kolmnurki saab läbi missuguste ringide toppida.
Sisendi esimesel real on täisarv M. Järgmisel real on M tühikutega eraldatud arvu, ringide diameetrid. Järgmisel real on täisarv N. Järgmisel N real on kolm tühikutega eraldatud arvu, vastava kolmnurga küljepikkused.
Väljundisse kirjutada N rida. Iga rea esimene arv on k, mitmesse ringi vastav kolmnurk mahub. Edasi on k tühikutega eraldatud täisarvu, ringide indeksid kasvavas järjekorras, kuhu kolmnurk mahub.



NÄIDE:

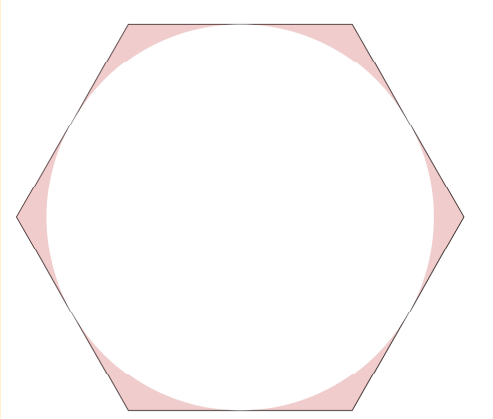
```
3
4.0 5.0 6.0
3
3.0 4.0 5.0
5.0 5.0 5.0
4.5 4.5 8.0
```

Vastus:

```
2 2 3
1 3
0
```

12.9.3 Pallimäng

Kettamaailmas tuntud mängu, N-nurk palli, mängitakse korrapärase N-nurga (3 ≤ N ≤ 50) siseringjoone sees, kohtunikud seisavad N-nurga alal, mis jääb väljapoole siseringjoont ning pealtvaatajad kogunevad kohtadesse, mis asuvad küll N-nurga ümberringjoone piirides, kuid mitte N-nurgas endas. Sulle on ette antud N ning N-nurga pindala S (0 ≤ S ≤ 30000). Leida kohtunike ning pealtvaatajate alade pindalad.
Sisendis on ette antud kaks tühikuga eraldatud arvu, N ja S.
Väljundisse kirjutada kaks tühikuga eraldatud arvu, pealtvaatajate ala pindala ning kohtunike ala pindala.



NÄIDE 1:

```
3 0.43301
```

Vastus:

```
0.61418 0.17121
```

NÄIDE 2:

```
6 2.59808
```

Vastus:

```
0.54352 0.24188
```

NÄIDE 3:

```
9 6.18182
```

Vastus:

```
0.53226 0.25314
```

12.9.4 Nelinurgad

Sulle antakse ette neli punkti, millest ükski kolm ei asu ühel sirgel. Leida, kas need neli punkti moodustavad, ruudu, ristküliku, rombi, rööpküliku, trapetsi või nelinurga. Antud juhul tuleb nimetada esimene punktinelikule vastav nimetus. Kui on ruut, siis tuleb väljastada „Ruut“, kui on nelinurk, mis ei vasta ühelegi teisele tingimusele, tuleb väljastada „Nelinurk“.
Sisendi esimesel real on täisarv N (1 ≤ N ≤ 50000), punktinelikute arv. Järgmisel 4N real on kaks tühikutega eraldatud täisarvu x ja y (-10000 ≤ x,y ≤ 10000), punkti x ja y koordinaat.
Väljundisse kirjutada N rida. N-nda punktineliku liik.

NÄIDE:

```
6
0 0
2 0
2 2
0 2
0 0
3 0
3 2
0 2
0 0
8 4
5 0
3 4
0 0
2 0
3 2
1 2
0 0
5 0
4 3
1 3
0 0
5 0
4 3
1 4
```

Vastus:

```
Ruut
Ristkylik
Romb
Roopkylik
Trapets
Nelinurk
```

12.9.5 Üleujutus

Suur üleujutus on tabanud Muumiorgu ning sinu ülesanne on nüüd aru saada, kui sügav on vesi ning kui suur osa Muumiorust on veega kaetud. Muumiorg koosneb 10*10 meetristest ruutudest, millel on teatud kõrgus. Samuti teame, et Muumiorus on kanalisatsioon nii hea, et vesi saab alati voolata ükskõik kust kõige madalamasse punkti, kus vett ei ole.
Sisendi esimesel real on kaks täisarvu N ja M (1 ≤ N,M ≤ 30), Muumioru mõõtmed ruutudes. Järgmisel N real on M tühikutega eraldatud täisarvu, ruudu kõrgus merepinnast. Järgmisel real on täisarv V, Muumiorgu voolanud vee hulk kuupmeetrites.
Väljundisse kirjutada eraldi ridadele kaks arvu, Muumioru veetaseme kõrgus merepinnast ning protsent, kui suur osa Muumioru ruutudest jääb täielikult vee alla.

NÄIDE:

```
3 3
25 37 45
51 12 34
94 83 27
10000
```

Vastus:

```
46,67
66,67
```

12.9.6 Park

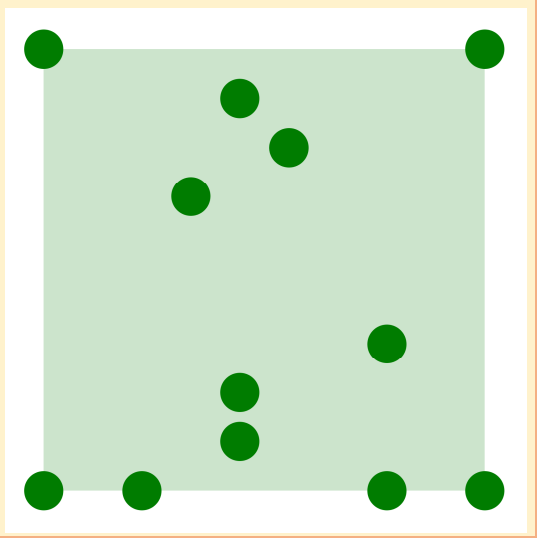
Linnavalitsus otsustas uut parki ehitama hakata. Selleks aga otsustati võtta mingisugune ala, kus kasvasid mõned üksikud puud ning võtta minimaalse ümbermõõduga hulknurk, mis sisaldab kõiki neid puid, ning muuta see ala pargiks. Leida tekkiva pargi pindala.
Sisendi esimesel real on täisarv N (1 ≤ N ≤ 100), pargis olevate puude arv. Järgmisel N real on kaks arvu, puu koordinaadid.
Väljundisse kirjutada üks arv, tekkiva pargi pindala.

NÄIDE:

```
12
1 1
5 3
4 7
1 10
6 8
3 1
10 1
8 1
5 9
10 10
5 2
8 4
```

Vastus:

```
81
```



12.9.7 Müür

Väike külake Süürias otsustas, et hea mõte oleks ümber küla müür ehitada. Selleks, et kulusid säästa, otsustati, et müür tuleb minimaalse ümbermõõduga, nii et kõik külas olevad majad oleksid müüriga piiratud. Leida see minimaalne ümbermõõt.
Sisendi esimesel real on täisarv N (1 ≤ N ≤ 100), külas olevate majade arv. Järgmisel N real on kaks arvu, maja koordinaadid.
Väljundisse kirjutada üks arv, ehitatava müüri ümbermõõt.

NÄIDE 1:

```
3
1 2
4 10
5 12,3
```

Vastus:

```
22,1
```

NÄIDE 2:

```
6
0 0
1 1
3,1 1,3
3 4,5
6 2,1
2 -3,2
```

Vastus:

```
19,66
```

NÄIDE 3:

```
7
1 0,5
5 0
4 1,5
3 -0,2
2,5 -1,5
0 0
2 2
```

Vastus:

```
12,52
```

12.9.8 Spioonid

Selleks, et tungida ülisuure valve all olevasse ettevõttesse ning sealt kohe signaale staapi saata, peab spioonifirma oma kommunikatsioonivarustust tükkidena hoidma, saates lisaks põhispioonidele ka tugispioone, kes kannavad endaga spetsialiseeritud tehnikat. Selleks, et välismaailmaga rääkida, peab põhispioon seisma tugispioonidest koosnevas kolmnurgas ning ta ongi ühenduses staabiga. Paraku on salatseval ettevõttel olemas *jammer*'id, mis blokeerivad põhispiooni signaali. Need *jammer*'id on peidetud pealtnäha tavaliste inimeste riietesse ning spioon, kes asub *jammer*'ite moodustatud kolmnurgas, ei saa staabiga suhelda hoolimata sellest, kas ta on sidekolmnurgas või mitte. Praegu loeme spiooni blokeerituks, kui tema signaali aktiivselt blokeeritakse, ühenduses, kui tal on ühendus staabiga ning levist väljas, kui teda aktiivselt ei blokeerita, kuid ta lihtsalt ei ole ühenduses. On antud *jammer*'ite, tugispioonide ning põhispioonide asukohad. Leida iga spiooni kohta, kas ta on blokeeritud, ühenduses või levist väljas.

Sisendi esimesel real on kolm tühikutega eraldatud täisarvu, J, T, P ($0 \le J,T,P \le 200$), *jammer*'ite, tugispioonide ning põhispioonide arv. Järgmisel J real on kaks tühikuga eraldatud täisarvu vahemikus -500 kuni 500, *jammer*'i koordinaadid. Järgmisel T real on kaks tühikuga eraldatud täisarvu vahemikus -500 kuni 500, tugispiooni koordinaadid. Järgmisel P real on kaks tühikuga eraldatud täisarvu vahemikus -500 kuni 500, põhispiooni koordinaadid.

Väljundisse kirjutada P stringi, millest igaüks on eraldi real. Kui vastav põhispioon on blokeeritud, kirjutada „Blokeeritud". Kui spioon on aga ühenduses, väljastada „Yhenduses". Kui ta on aga levist väljas, tagastada „Levist v2ljas".

NÄIDE 1:

```
3 3 2
0 0
10 0
0 10
20 20
20 0
0 20
5 5
15 15
```

Vastus:

```
Blokeeritud
Yhenduses
```

NÄIDE 2:

```
3 3 1
0 0
10 0
0 10
20 20
20 0
0 20
40 40
```

Vastus:

```
Levist v2ljas
```

12.9.9 Märklaud

Märklaud koosneb N (1 ≤ N ≤ 10) alast, mis on nummerdatud arvudega 0 kuni N-1. Iga ala märklaual on kas ristkülik, ring või kolmnurk ning annab 2^i punkti, kus i on ala number. Mängija viskab M (1 ≤ M ≤ 10) noolt vastu märklauda. Nool saab ala poolt antavaid punkte parajasti siis, kui see asub selle ala sees. Leida iga noole kohta, mitu punkti see saab.
Sisendi esimesel real on kaks täisarvu, N ja M. Järgmisel M real on mingi täht c, mis võib olla kas r, c või t. Kui see on r, on tegemist ristkülikuga ning samal real järgneb 4 tühikutega eraldatud arvu, ristküliku ülemise vasaku ning alumise parema nurga koordinaadid. Kui see on c, on tegemist ringiga ning järgmised 3 tühikutega eraldatud arvu on selle keskpunkti koordinaadid ning raadius. Kui see on t, on tegemist kolmnurgaga ning järgmised 6 arvu on selle iga tipu koordinaadid. Järgmisel M real on kaks arvu, mis näitavad noole asukohta. Kõik koordinaadid on vahemikus -1000 kuni 1000.
Väljundisse kirjutada M täisarvu, vastava noole poolt saadud punktisumma.
NÄIDE:

```
9 6
r 8.5 17.0 25.5 -8.5
c 20.2 7.3 5.8
t -1.0 -1.0 10.1 2.2 .4 1.4
r 0.0 10.3 5.5 0.0
c -5.0 -5.0 3.7
t 20.3 9.8 10.0 -3.2 17.5 -7.7
r 2.5 12.5 12.5 2.5
c 5.0 15.0 7.2
t -10.0 -10.0 10.0 25.0 30.0 -10.0
2.0 2.0
4.7 5.3
6.9 11.2
20.0 20.0
17.6 3.2
-5.2 -7.8
```

Vastus:

```
264
328
448
0
291
272
```

12.9.10 Viirus

Internet on ristkülik mõõtmetega W*H (1 ≤ W,H ≤ 1000), mida läbib N (0 ≤ N ≤ 100) sirgjoonelist tulemüüri. Iga tulemüür ulatub täpselt servast serva. Viirus sattus internetis ühte punkti koordinaatidega x,y (0 ≤ x ≤ W; 0 ≤ y ≤ H) ning nakatas kogu selle ala, mis ei olnud tema eest tulemüüriga kaitstud. Leida nakatunud interneti pindala.
Sisendi esimesel real on viis tühikutega eraldatud täisarvu, N, W, H, x ja y. Järgmisel N real on neli tühikutega eraldatud täisarvu, tulemüüri alg- ja lõpppunkt. Võib eeldada, et tulemüür algab ning lõpeb interneti servas.
Väljundisse kirjutada üks arv, nakatunud interneti pindala.

NÄIDE:

```
3 100 100 20 30
0 0 100 100
100 0 0 100
0 40 40 100
```

Vastus:

```
1780,000
```