

Võistlusprogrammeerimine

2. osa



Targo Tennisberg Katrin Gabrel

Tartu Ülikool 2018



TARTU ÜLIKOOL
teaduskool



TARTU ÜLIKOOL
arvutiteaduse instituut

Võistlusprogrammeerimise õpik on loodud Tartu Ülikooli teaduskooli ja arvutiteaduse instituudi koostöös. Õpiku loomine sai alguse soovist pakkuda kõigile motiveeritud Eesti kooliõpilastele parimaid võimalusi oma programmeerimisoskuste arendamisel ja olümpiaadideks valmistumisel - selleks on juba ka loodud õpikule toetuv teaduskooli kursus "Võistlusprogrammeerimine II". Õpik sobib ka lisamaterjalina kasutamiseks kõrgkoolide andmestruktuuride ja algoritmide kursusel üliõpilaste teadmiste ja oskuste arendamisel. Samuti leiavad siit õpikust tuge ka juba professionaalsed programmeerijad algoritmiliselt keerukate probleemide lahendamisel.

Lisaks TÜ teaduskoolile ja arvutiteaduse insituudile on õpiku loomist toetanud ka mitmed annetajad. Täname õpiku loomist toetanud eraisikuid:

Ahti Heinla, Ahti Kitsik, Neeme Praks

Täname õpiku loomist toetanud ettevõtteid:



Autoriõigused Targo Tennisberg, Katrin Gabrel ja Tartu Ülikool, 2018

Võistlusprogrammeerimine

ehk kuidas kirjutada kiiret koodi

2. osa

Targo Tennisberg, Katrin Gabrel

7	Efektne programmeerimistehnika.....	7
7.1	Koodistiil.....	8
7.1.1	Pikaajaline ja lühiajaline lähenemine	8
7.1.2	Alamprogrammid.....	8
7.1.3	Näide: Tankid.....	10
7.1.4	Kommentaarid.....	14
7.1.5	Nimed	16
7.1.6	Funktsioonide nimetamine.....	17
7.2	Testimine	17
7.2.1	Testide koostamine	18
7.2.2	Übermööd	18
7.2.3	Testide genereerimine	21
7.2.4	Käsuinterpretaatorid ja skriptimine	23
7.2.5	Valideerimine	25
7.3	Tunne oma keelt.....	26
7.3.1	Teegid	26
7.3.2	Andmestruktuurid	26
7.3.3	Makrod	27
7.4	Võistluste strateegia.....	28
7.5	Ahned algoritmid.....	28
7.5.1	Mündid	29
7.5.2	Kingid	30
7.5.3	Majakavahid	33
7.5.4	Veel ahnetest algoritmidest	35
7.5.5	Suveniirid.....	35
7.6	Kontrollülesanded	39
7.6.1	Onu Robert	39
7.6.2	Bussijuhid.....	40
7.6.3	Hernehirmutised	40
7.6.4	Kolimine.....	41
7.6.5	Krokodillid.....	41
7.6.6	Lohe Gorõnitš	42
7.6.7	Fraktsioonid.....	43
7.6.8	Bitimask	43
7.6.9	LED-lambid.....	44
7.6.10	Antimonotoonne jada	45
7.7	Viited lisamaterjalidele.....	45

8	Dünaamiline planeerimine edasijõudnutele	46
8.1	Seljakoti pakkimine.....	46
8.1.1	Ülesanne: Kohver	46
8.1.2	Dünaamiline planeerimine	47
8.1.3	Rekursioonivalem	47
8.1.4	Tabeli täitmine.....	47
8.1.5	Kood.....	48
8.1.6	Tee taastamine	48
8.1.7	Keerukus	49
8.1.8	Seljakoti erinevad variatsioonid	49
8.2	Rändkaupmehe ülesanne	49
8.2.1	Ülesanne: kaupmees	50
8.2.2	Rekursioonivalem	50
8.2.3	Valemi kasutus.....	50
8.2.4	DP tabel	51
8.2.5	Kood.....	52
8.2.6	Tee taastamine	53
8.2.7	Keerukus	53
8.3	Levenshteini kaugus	53
8.3.1	Ülesanne: Levenshteini kaugus	54
8.3.2	Rekursioonivalem	55
8.3.3	DP tabel	55
8.3.4	Teisenduse taastamine tabelist.....	56
8.3.5	Ajaline ja mahuline keerukus	56
8.3.6	Levenshteini kauguse omadused	57
8.3.7	Levenshteini kauguse variatsioone	57
8.4	Geenijärjestuste võrdlemine	57
8.4.1	Ülesanne: Sarnasus.....	58
8.4.2	Ülesanne: Mis on ühist?	59
8.4.3	Rekursioonivalem	59
8.4.4	Tabel	60
8.4.5	Kood.....	60
8.4.6	Tee taastamine	60
8.5	Dünaamiline Aja painutamine (Dynamic Time Warping)	61
8.5.1	Ülesanne: helid	61
8.5.2	Rekursioonivalem	62
8.5.3	Tabel	62
8.5.4	Kood.....	62
8.6	Lõikudega ülesanded.....	63
8.6.1	Ülesanne: palgi lõikamine	63
8.6.2	Tee DP-ni	63
8.6.3	Rekursioonivalem	63
8.6.4	Tabeli täitmine.....	63
8.6.5	Kood.....	65
8.6.6	Tee taastamine	65
8.6.7	Keerukus	66
8.7	Maatriksite korrutamine	66
8.7.1	Ülesanne: maatriksid	66

8.7.2	Võimaluste arv.....	66
8.7.3	Tee DPni.....	67
8.7.4	Tabeli täitmine.....	67
8.7.5	Kood.....	68
8.7.6	Keerukus.....	68
8.8	Ülesanne: mäed.....	69
8.8.1	Rekursioonivalem.....	70
8.8.2	Tabeli täitmine.....	71
8.8.3	Kood.....	71
8.8.4	Keerukus.....	72
8.9	Kontrollülesanded.....	73
8.9.1	Torn.....	73
8.9.2	Bussireisid.....	74
8.9.3	Sillad.....	75
8.9.4	Mäng.....	76
8.9.5	Uuring.....	77
8.9.6	Alkeemik.....	78
8.9.7	Puhkus.....	79
8.9.8	Tango.....	80
8.9.9	Valimised.....	81
8.9.10	Plaan.....	82
8.10	Viited lisamaterjalidele.....	82
9	Eriotstarbelised puud.....	83
9.1	Lõikude puu.....	83
9.1.1	Puu loomine jadast.....	84
9.1.2	Lõigul väärtuse leidmine.....	85
9.1.3	Väärtuse muutmine.....	87
9.1.4	Väärtuste muutmine tervel lõigul.....	87
9.2	Laisk väärtustamine lõikude puudes.....	88
9.2.1	Paberiribade kleepimine.....	91
9.2.2	Kood.....	95
9.2.3	Ülesanne: Sein.....	98
9.3	Fenwicki puu.....	100
9.3.1	Summa leidmine.....	101
9.3.2	Elemendi lisamine ja muutmine.....	102
9.3.3	Üksiku elemendi leidmine.....	102
9.3.4	Indeksi leidmine kumulatiivse summa järgi.....	103
9.3.5	Teatud lõigu summa leidmine.....	103
9.4	2-mõõtmeline Fenwicki puu.....	103
9.4.1	Lisamine.....	104
9.4.2	Loendamine.....	104
9.5	Ajalooga andmestruktuurid.....	104
9.5.1	Ajalooga lõikude puu.....	105
9.5.2	Ülesanne: Tekstiredaktor.....	106
9.5.3	Kood.....	108
9.6	Hõre tabel.....	109
9.6.1	Tabeli koostamine.....	109
9.6.2	Tabelist lugemine.....	110

9.7	Ühisosata hulgad	110
9.7.1	Kasutamine	112
9.8	Kontrollülesanded	113
9.8.1	Laoseisud	113
9.8.2	Korrutamismäng	114
9.8.3	Külaskäigud	115
9.8.4	Berliini müür	115
9.8.5	Lambipirnid	116
9.8.6	Taani valimised	117
9.8.7	Rivistus	117
9.8.8	Madalad majad	118
9.8.9	Platvormid	119
9.8.10	Paberi lõikamine	120
9.9	Viited lisamaterjalidele	120
10	Graafiteooria	121
10.1	Floyd-Warshalli algoritm	121
10.1.1	Ülesanne: Kauguste maatriks	122
10.1.2	Rekursioonivalem	122
10.1.3	Tabeli täitmine	122
10.1.4	Floyd-Warshalli algoritmi kasutusi	125
10.2	Toese mõiste	126
10.2.1	Servade liigitamine	126
10.2.2	DFS koos servade liigitamisega	127
10.3	Sillad ja Artikulatsioonipunktid	128
10.3.1	Ülesanne: Muinastulede öö	129
10.3.2	Sildade leidmine	133
10.4	Euleri Graaf	134
10.4.1	Euleri tsükli leidmine	134
10.4.2	Loomanimed	135
10.5	Minimaalne Toes	137
10.5.1	Minimaalse toese leidmine	138
10.5.2	Ülesanne: Elektrikatkestus	139
10.6	Maksimaalne voog ja minimaalne lõige	141
10.6.1	Võrk	141
10.6.2	Voog	142
10.6.3	Maksimaalne voog	142
10.6.4	Maksimaalse voo leidmine	143
10.6.5	Mitme lähte ja suudmega ülesanded	145
10.6.6	Kaaludega tipud võrgus	145
10.6.7	Lõige	146
10.6.8	Vangid	147
10.7	klikid	150
10.7.1	Bron-Kerboschi algoritm	150
10.7.2	Sõbrad	151
10.8	Kontrollülesanded	153
10.8.1	Laadimispunktid	153
10.8.2	Internetiühendus	154
10.8.3	Kiviheitemasin	155

10.8.4	Uus tee.....	156
10.8.5	Mõjuvõim	157
10.8.6	Civilization	158
10.8.7	Põgenemine.....	159
10.8.8	Teepuhastus	160
10.8.9	Doomino	160
10.8.10	Valimised	161
10.9	Viited lisamaterjalidele.....	161
11	Tekstialgoritmid.....	162
11.1	Teksti parsimine	162
11.1.1	Ülesanne: arvutustehe	163
11.2	Alamstringi leidmine.....	165
11.2.1	Mõisted.....	165
11.2.2	Lihtne alamstringi otsimise algoritm	166
11.3	Knuth-Morris-Pratti algoritm.....	167
11.3.1	Kokkulangevuste tabel	168
11.3.2	Hulknurgad	169
11.4	Sufiksipuud	171
11.4.1	Prefiksipuu.....	171
11.4.2	Sufiksipuu	172
11.4.3	Sufiksipuu kasutamine.....	174
11.4.4	Sufiksipuu loomine	177
11.5	Sufiksiloend	177
11.5.1	Loomine.....	178
11.5.2	Sufiksiloendi kasutamine.....	180
11.6	Räsi	181
11.6.1	Alamstringi leidmine räsi abil	182
11.6.2	Räsi moodularvutusega	183
11.6.3	Jagamise vältimine	184
11.6.4	Rabin-Karpi algoritm.....	184
11.6.5	Ülesanne	185
11.7	Kontrollülesanded	187
11.7.1	Telefoninumbrid.....	187
11.7.2	Enigma	188
11.7.3	Universaalgeen.....	188
11.7.4	Arelofansi keel	189
11.7.5	Koodigeneraator.....	190
11.7.6	Automaathelistaja	191
11.7.7	Risti-rästi.....	192
11.7.8	Ahvitembud	192
11.7.9	Palindroomid	193
11.7.10	Otsing.....	194
11.8	Viited lisamaterjalidele.....	195
12	Arvutusgeomeetria.....	196
12.1	Geomeetrilised objektid.....	196
12.1.1	Punkt.....	196
12.1.2	Lõik	196
12.1.3	Vektor	197

12.1.4	Sirge	197
12.1.5	Kiir.....	198
12.1.6	Murdjoon.....	198
12.1.7	Hulknurk	199
12.1.8	Ringjoon.....	199
12.1.9	Kaar, kõõl, sektor ja segment	200
12.2	operatsioone punktide ja sirgetega.....	200
12.2.1	Punkti pööramine tasandil	200
12.2.2	Kahe punkti vaheline kaugus.....	200
12.2.3	Paralleelsuse ja samasihilisuse kontroll	201
12.2.4	Sirgete lõikepunkt.....	201
12.2.5	Lõikude lõikumine	202
12.3	Hulknurk	204
12.3.1	Hulknurga ümbermõõd ja pindala	204
12.3.2	Picki teoreem.....	206
12.3.3	Punkti sisaldumine hulknurgas.....	206
12.4	Koordinaatide pakkimine	207
12.4.1	Ülesanne: Tikuruudud	209
12.5	Kumer kate	211
12.6	Ülesanne: ring ja ruut.....	215
12.7	Lähim punktipaar.....	219
12.8	Magic missile (ülesanne)	220
12.9	Kontrollülesanded	225
12.9.1	Liitumine.....	225
12.9.2	Augumäng.....	226
12.9.3	Pallimäng	226
12.9.4	Nelinurgad	227
12.9.5	Üleujutus	228
12.9.6	Park.....	228
12.9.7	Müür	229
12.9.8	Spioonid.....	230
12.9.9	Märklaud	231
12.9.10	Viirus.....	232

7 EFEKTIIVNE PROGRAMMEERIMISTEHNICA

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!

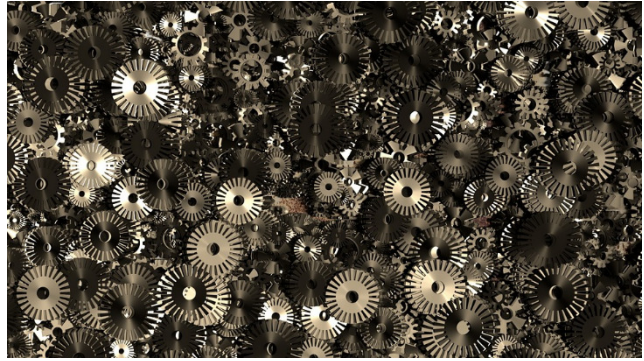
Võrdleme näiteks kaht olukorda:

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 KODISTIIL

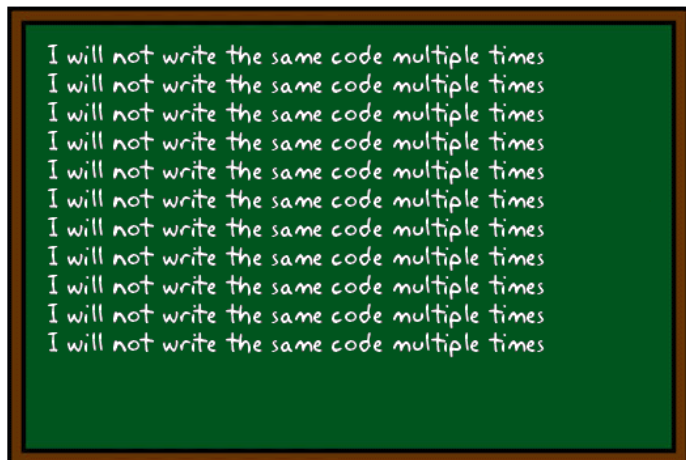
7.1.1 Pikaajaline ja lühiajaline lähenemine

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 optimeerimine** 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 kohe 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/>) soovib, 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.

Taaskasutusest tuleb lähemalt juttu järgmises näites:



Uncle Bob



Targo esimene programmeerimisarvuti oli samasugune

7.1.3 Näide: Tankid

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:



M1A1 Abrams tank

```
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; // jätkka 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; // jätkka
                }
                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. Ilmselt 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); // pöikleme
        }
    }
    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 `pöikle`. 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 (onLaua() && onVaba(xsuund) && (nurkX - x > nurkY - y || fm)) {
        ol = liiguTulega(xsuund); //liigu horisontaalselt
        fm = 0;
    }

    fm = 1;
    while (onLaua() && onVaba(ysuund) && (nurkX - x < nurkY - y || fm)) {
        ol = liiguTulega(ysuund); // liigu vertikaalselt
        fm = 0;
    }
    return ol;
}
```

Siingi on abifunktsioonid loetavuse parandamiseks `onLaua` ja `onVaba`:

```
bool onLaua() {
    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 soovitusena 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 nonsens:

```
/*
** 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 kahetäheliselt 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.

Kommentaariid 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) nõuande 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

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.



Phil Karlton

- Mida pikem on kood, kus muutujat kasutatakse, seda raskem on järele pidada, mida muutuja teeb. Sellest jäeldub **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 tsükli 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() < 1) ||
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 programmeerimiskeskondades 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 kulub vigade otsimisele, sest programm on muutunud ka autorile endale arusaamatuks, on need ekstra sekundid aega täiesti tühi.

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 onLaua1(int x, int y, int pikkus) {  
    return ((x >= 0) && (x < pikkus) && (y >= 0) && (y < pikkus))  
}
```

Seda laadi nimetused teevad koodi hästi loetavaks:

```
if (onLaua1(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 argumentiks 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 onLaua1(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 Übermõõt

Informaatikaolümpiaadi ettevalmistuslaagris anti osalejatele lahendada järgmine ülesanne:

Ristkülikute ühendi übermõõt

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 ristkülikutest koosnevat kujundit, mis väliskujundit ei puuduta üheski punktis, tuleb lugeda ikkagi omaette kujundiks.

Sisend: Faili esimesel real on antud ristkülikute arv N ($1 \leq N \leq 1000$). 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 ($1 \leq K \leq 20$). Järgnevas K blokis on bloki esimesel real oodatav tulemus ning selle järel sisendandmed samal kujul kui ülesande kirjelduses. Lahendusena tuleb esitada see tekstifail.

NÄIDE:

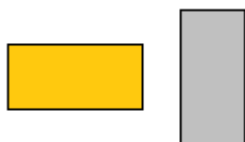
```
2
4.0
1
1.0 1.0 2.0 2.0
564.0
3
7.0 170.0 99.5 190.0
0.5 100.0 12.0 225.0
10.0 80.0 50.0 110.0
```

Ülesande sisu analüüsid tuleks kindlasti teha testid, mis kasutavad minimaalset ja maksimaalset lubatavat riskülikute arvu, mis tähendab, et võiks olla test, milles on vaid üks riskülik, ja test, milles riskülikuid on täpselt tuhat.

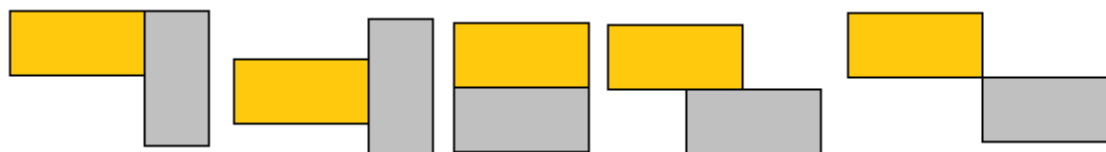
Kuna koordinaatideks võivad olla positiivsed reaalarvud, siis ei piisa testimisel kindlasti ainult täisarvulistest koordinaatidest, vaid tuleks kasutada nii väga väikeseid arve, paljude kohtadega arve, aga ka loomulikult maksimaalset lubatud koordinaadi väärtust.

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.

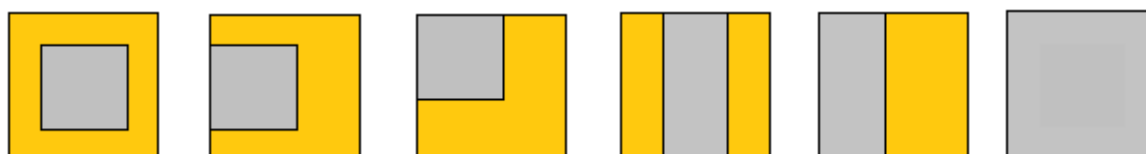
Asume nüüd ülesande sisulise analüüsi juurde. Vaatame kõigepealt, kuidas saavad 2 riskülikut teineteise suhtes paikneda. Esimene võimalus on, et riskülikud üldse kokku ei puutu:



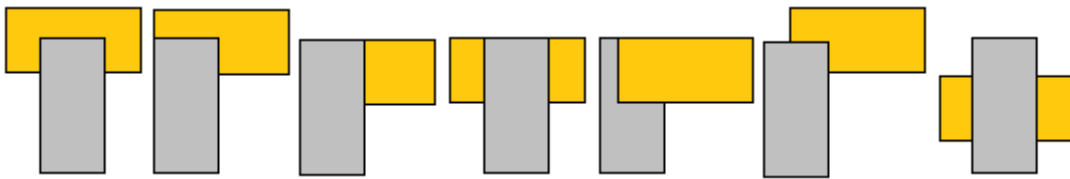
Puutuvatest, kuid mitte kattuvatest riskülikutest koosnevad erinevad variandid:



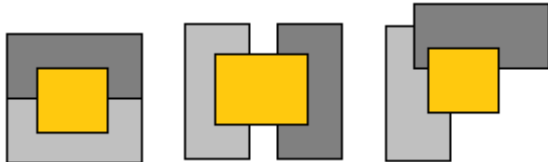
Üks riskülik võib asuda ka täielikult teise sees. Kui arvestada ka võimalike puutumistega, saame järgmised võimalused (hall riskülik on tervenisti oranži risküliku sees):



Lõikuvad riskülikud:

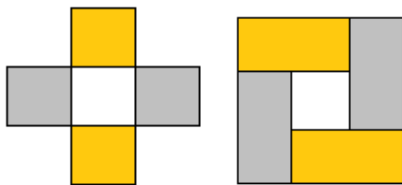


Nii on juba kahe risküliku omavahelise paiknemise jaoks paarkümmend erinevat võimalust, kolmanda lisamine lisab veel hulga võimalikke kombinatsioone. Kõiki neid ei jõua siin ära tuua. Kolme riskülikuga võimalustest huvitavamad on järgmised paiknemised:



Vasakpoolsel paikneb kolmas riskülik tervenisti kahe teise risküliku sees (aga mitte kummagi sees), parempoolsel on kattumine mõlema teise riskülikuga, kõik kolm osalevad übermõõdu arvestamises, kolmandal on kõigil paarikaupa kattuvaid osi kui ka osa, mille katavad kõik kolm riskülikut.

Kindlasti ei tohi ära unustada, et riskülikud võivad moodustada tühje alasid kujundi sees, mis tuleb übermõõdu arvestamisel arvesse võtta. Neljast riskülikust saab näiteks järgmised kujundid:



Kui lisada veel riskülikuid, saab tekitada ühe kujundi sees ka mitu tühja ala. Kindlasti tasuks testida ka rohkemate kujunditega tekkivaid formsioone, 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 riskülikute 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 riskülikute arv on 1000 ja viimane riskülik osaleb perimeetri moodustamises.
- Lahendus töötab valesti, kui perimeetri pikkus ületab 16-bitilise arvu maksimumväärtust.
- Lahendus töötab valesti, kui leidub vähemalt 2 täpselt samasugust riskülikut.

- Lahendus töötab valesti, kui leidub vähemalt 3 riskülikut, mille vertikaalsed küljed on samal x-koordinaadil.
- Lahendus töötab valesti, kui leidub vähemalt 3 riskülikut, 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 ($0 \leq L_i \leq 100$) ja K_i ($1 \leq K_i \leq 100$), mis tähistavad reanumbrile vastava lastekodu laste arvu ning korvi pindala. Sisendi neljandal real on kingituste arv N ($1 \leq N \leq L_1 + L_2 + L_3$). Viimasel real on N ($1 \leq N \leq 1000$) 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:

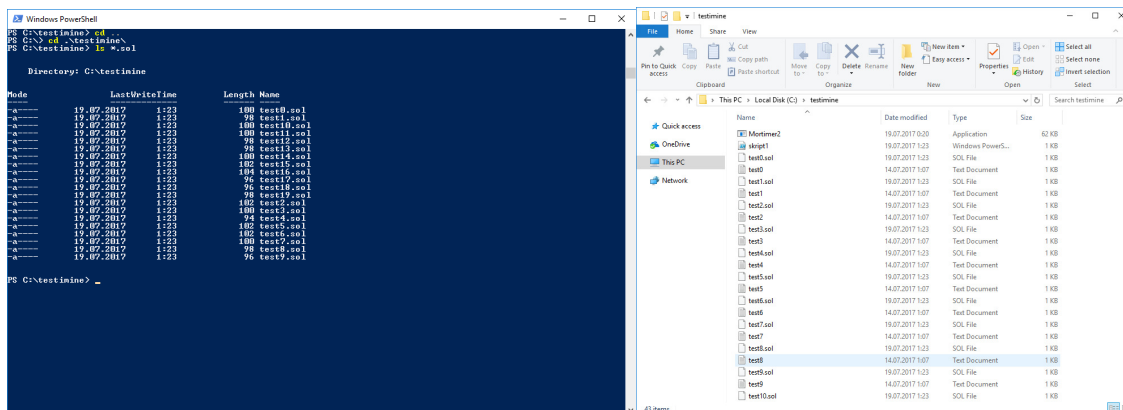
```
int N = 1 + rand() % min(lapsi, 15); // kinkide arv
```

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)

Shelliks 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 **käsk**e (*commands*, PowerShellis *cmdlets*). Siin on tabel mõningate testide jooksutamiseks põhiliselt vajaminevate käskudega:

Käsk	Bash	PowerShell cmdlet	PowerShell 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
Kellaage	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 *pipinguks* 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. PowerShell 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

```
$nimi = "Siim Susi"
$vanus = 14
echo $nimi $vanus
```

Bash

```
nimi = "Siim Susi"
vanus = 14
echo $nimi $vanus
```

Kui Bashis on vaja muutuja salvestada **käsu tulemus**, tuleb omistamisel käsk panna kas tagurpidi ülakomade vahele või kasutada süntaksit `muutuja = $(käsk)`.

PowerShell

```
if ($tingimus) {teeMidagi}
```

```
foreach ($i in $mingiHulk) {
    teeMidagi
}
```

```
for ($i=0; $i -le 10; $i++) {
    teeMidagi
}
```

Bash

```
if $tingimus
then teeMidagi
fi
```

```
for i in $mingiHulk
do
    teeMidagi
done
```

```
for ((i=0; i < 10; i++))
do
    teeMidagi
done
```

Rohkem infot Bashi ja PowerShell 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 programmeerimiskeeltele on palju sarnaseid omadusi ja võimalusi, on konkreetsetel keeltele 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 suurepärasest 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 kirjade 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 lahendamist 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:
 - o annab kinnituse, et oled algoritmiliselt õigel teel,
 - o kui su lahenduses on vigu, mille tõttu mõni test läbi ei lähe, saad sellest kohe teada,
 - o 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 „ahnel“, 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 ($L \leq M \leq 2L$). 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:

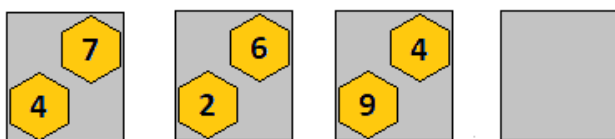
4 6

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))

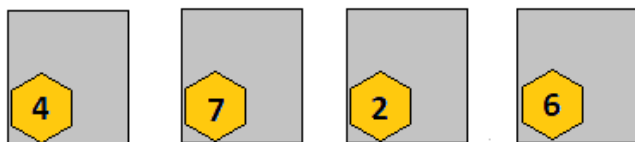
Selle ülesande puhul on veidi keerulisem läbi närida, kuidas ahne lähenemine töötada võiks. Vaatame esimest näidet nelja paki ja 6 mänguasjaga. Proovime kõigepealt lihtsalt jagada esimesse pakki 2 asja, teise 2 jne. Saame jaotuse:



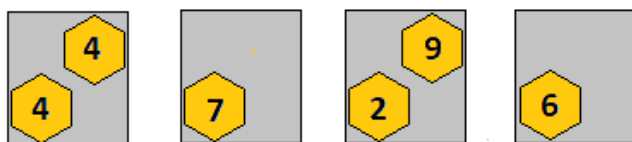
Esimese paki väärtus on 11, teise oma 8, kolmandal 13 ja viimasel 0. Kõigi pakkide keskmine väärtus on $(11 + 8 + 13 + 0) : 4 = 32 : 4 = 8$. Seega on praegu väärtuste vahede summa $|11 - 8| + |8 - 8| + |13 - 8| + |0 - 8| = 3 + 0 + 5 + 8 = 16$.

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:

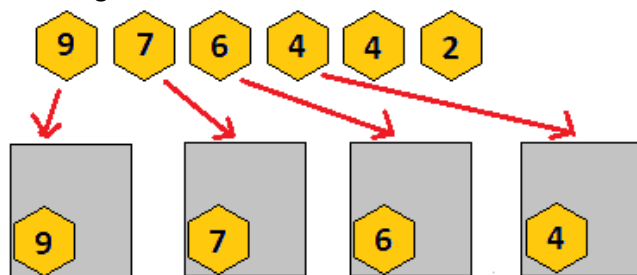


Vaja on veel lisada kingitused väärtustega 9 ja 4. Parima jaotuse saamiseks tuleks lisada kõige suurema väärtusega kingitus kõige väiksema väärtusega pakki, saame jaotuse:

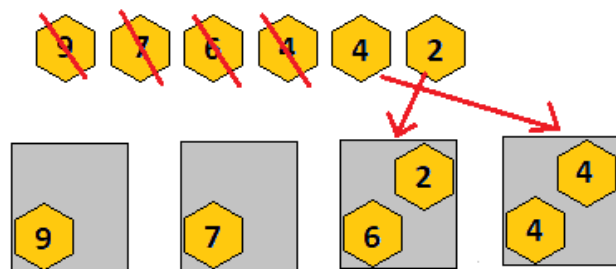


Siin on pakside väärtuste keskmisest erinevuste summa $|8-8| + |7-8| + |11-8| + |6-8| = 6$. Aga seegi ei ole veel vähim võimalik.

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 pakside 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 toeseppu 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 $0 \leq i \leq L-2$ 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 algseesse positsiooni.

Sisendi esimesel real on kolm arvu N ($1 \leq N \leq 10^7$), K ($1 \leq K \leq N$) ja L ($1 \leq L \leq 10^9$). Teisel real on N arvu: esimesena esimese meeskonna sektor, teisena teise meeskonna oma jne.

NÄIDE:

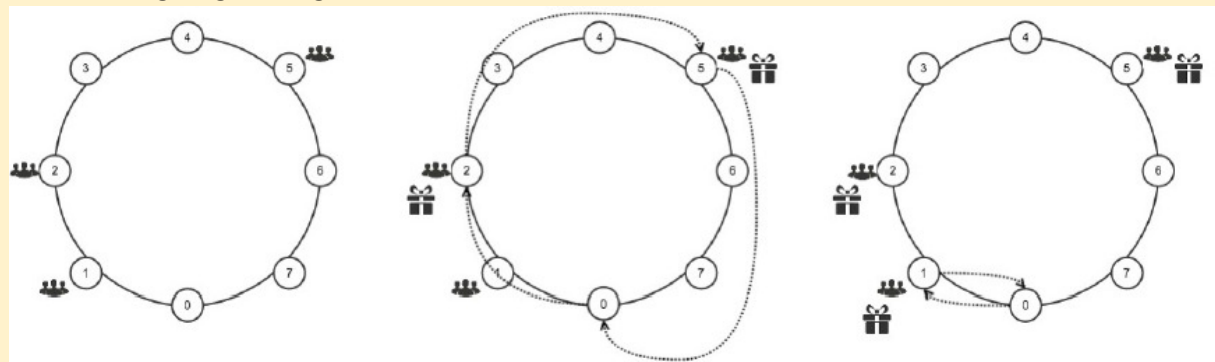
3 2 8

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

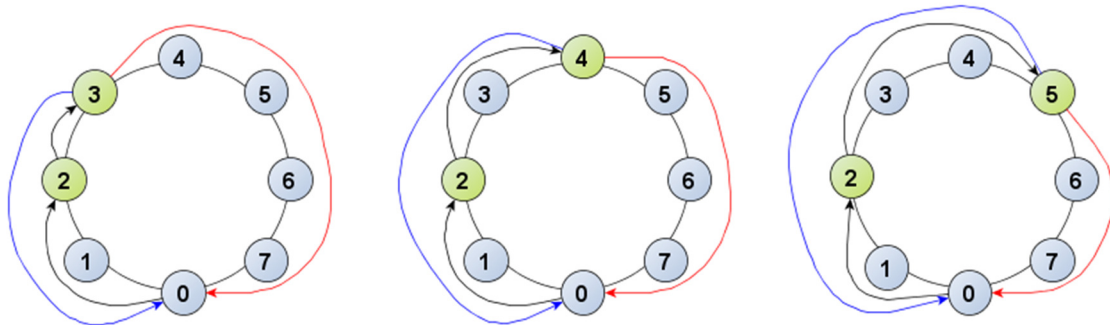


0-sektorist on võimalik suveniire viia teistesse sektoritesse liikudes kas päripäeva või vastupäeva. Ühe sületäie suveniiride jaotamiseks on kolm võimalust:

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.

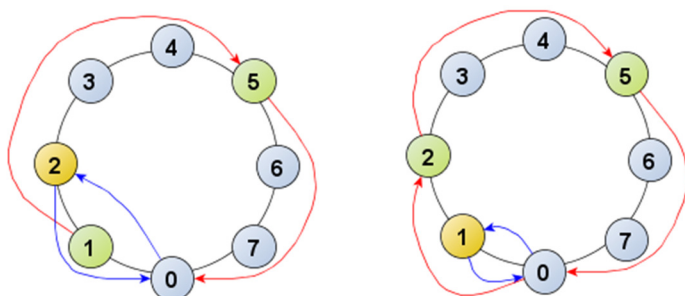
Oluline on tähele panna ka, et optimaalse jagamise korral ühe sületäie jagamisele ei saa kuluda rohkem kui L sekundit. Kui liigume ühe ringi, siis kulub selleks täpselt L sekundit. St kui me liigume päri- või vastupäeva jagades kaugemale kui poole ringi peale, on optimaalsem tagasi minemise asemel teha täisring. Seega päripäeva jagades on mõtet viia suveniire kuni sektsioonini $L/2$. Sama kehtib ka vastupäeva jagades:



Vasakpoolsel joonisel viiakse kaks pakki seksioonidesse 2 ja 3. Seejärel on tagasi 0-seksiooni liikumiseks 2 võimalust: päripäeva (punane nool) ja vastupäeva (sinine nool). Kasulikum on tagasi liikuda vastupäeva, nii kulub 2 paki jagamiseks 6 sekundit, vastupäeva kulub 8 sekundit. Keskmisel joonisel viiakse kaks pakki seksioonidesse 2 ja 4 ning seejärel liigutakse tagasi 0-seksiooni kas päri- või vastupäeva. Mõlemad teekonnad on sama pikad ja võtavad 8 sekundit. Parempoolisel 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 seksioonides olevatele meeskondadele. Näites on mõttekas jagada ühel käigul suveniirid meeskondadele seksioonides 1 ja 2 või 2 ja 5, aga mitte 1 ja 8 (liikudes teisest seksioonist jagamata üle). Tõesti, ringiratast liikudes, kui jätame mõne seksiooni vahele, siis saame selle vahetada seksiooni vastu, mis on seksioonile 0 lähemal. Ringiratast jagamise ajakulu see ei muuda, kuid 0-seksioonile lähemal olevasse sektorisse on kiirem kui edasi-tagasi viimine.

Samuti edasi-tagasi käikudel tasub kogu sületäis viia korraga järjestikustesse seksioonidesse, 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 seksioonidesse 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 seksioonidesse 2 ja 5 ning seejärel viimane pakk seksiooni 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 $K \geq N$.

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 seksioonide vahel, järjestikused seksioonid 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] = (1 - 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] = (1 - 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 = 1; // 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] + 1);
    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] + 1); // 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] + 1);
        }
    }
    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 ($1 \leq N \leq 1000$). Kohale jõudes läheb ta kõigepealt pank, 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 rahatähte. 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 rahatähtede 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 ($1 \leq N \leq 100$). 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 ($1 \leq D \leq 10000$), peab talle maksma lisatasu R tugrikut iga kilomeetri eest ($1 \leq R \leq 5$).

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 Tansaania sõitsin.



7.6.3 Hernehirmutised

Farmer Fredil on pikk peenar, mida ohustavad varesed. Peenar on jagatav N lõiguks ($1 \leq N \leq 100$), 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 kummaldi 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 ($1 \leq N \leq 10\,000$). 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 ($1 \leq D \leq 10^9$). 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. Böliinadest tuntud vägilane Dobrõnja Nikitič 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 ($1 \leq M, N \leq 20\ 000$). 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:

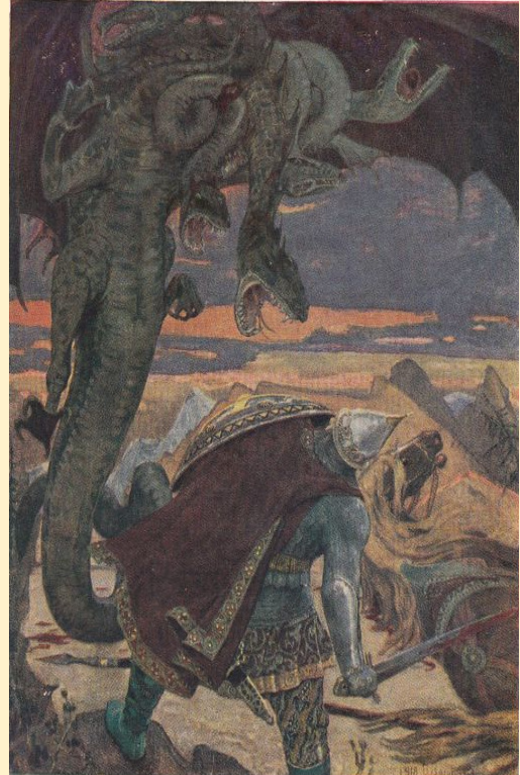
11 (palkame esimese ja kolmanda kangelase)

NÄIDE 2:

2 1
5 5
10

Vastus:

0



7.6.7 Fraktsioonid

N -liikmeline ($5 \leq N \leq 1000$) 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 $3 \cdot 4 = 12$ 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 bittide. Kui meil on näiteks vaja 32-bitise arvu alumised neli bitti ära nullida, tuleb sooritada bitikaupa AND operatsioon bitimaskiga $0xFFFFFFFF$ (kõik bitid peale alumise nelja on ühed).

Siin ülesandes tuleb leida bitimask M , mis:

1. asub kindlas vahemikus ($L \leq M \leq U$),
2. annab etteantud arvuga N bitikaupa OR operatsiooni järel maksimaalse tulemuse.

Sisendi ainsal real on kolm 32-bitist täisarvu N , L ja U , kus $L \leq U$. Väljundisse kirjutada arv M . Kui võimalikke vastuseid on mitu, väljastada neist väikseim.

NÄIDE 1:

100 50 60

Vastus:

59 ($59 \mid 100 = 127$)

NÄIDE 2:

100 0 100

Vastus:

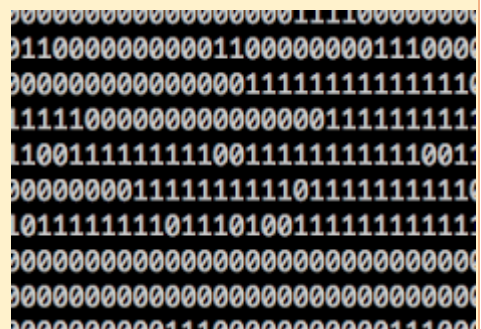
27 ($27 \mid 100 = 127$)

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 ($1 \leq A \leq N \leq 100$, $1 \leq B \leq M \leq 100$). Järgmisel N real on igaühel M märgist koosnev string, mis näitab, millised lambid on soovitud pildil sisse lülitatud – 0 tähistab väljalülitatud 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

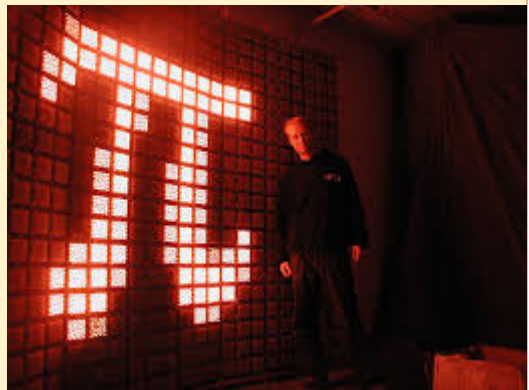
0110

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 $B[0] > B[1] < B[2] < B[3] \dots$

Sisendi esimesel real on jada A pikkus N ($1 \leq N \leq 30\,000$). 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 pööratakse 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.

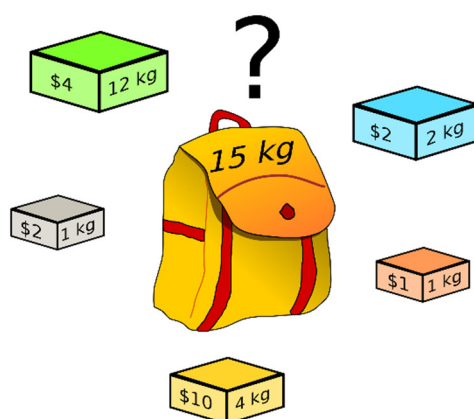
Those who cannot remember the past are condemned to repeat it.

- Dynamic programming

8.1 SELJAKOTI PAKKIMINE

Seljakoti pakkimise probleem (*knapsack problem*) on ilmselt arvutiteaduse tuntumate ülesannete seas. Tegemist on optimeerimisülesandega ja seda kasutatakse paljudes optimeerimisvaldkondades. 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 kohvrissi pakkida või mitte. Kui me viimast eset kohvrissi 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 kohvrissi 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:

$$mv(i, m) = \begin{cases} 0, & \text{kui } i = 0 \\ mv(i - 1, m), & \text{kui } m_i > m \\ \max(mv(i - 1, m), mv(i - 1, m - m_i) + v_i), & \text{kui } m_i \leq m \end{cases}$$

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 kohvrissi (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 kohvrissi, 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 kohvrissi, mille mass on kuni i kg ja mis on komplekteeritud esemete hulgast ridadel $1 \dots j$.

	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 kohvrissi 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 kohvrissi. 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 $O(nM)$, kus M on koti lubatud kaal.

Seega antud algoritm, lahendab seljakotiülesande **pseudopolünoomiaalse** keerukusega. See tähendab, et keerukus on polünoomiaalne 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:

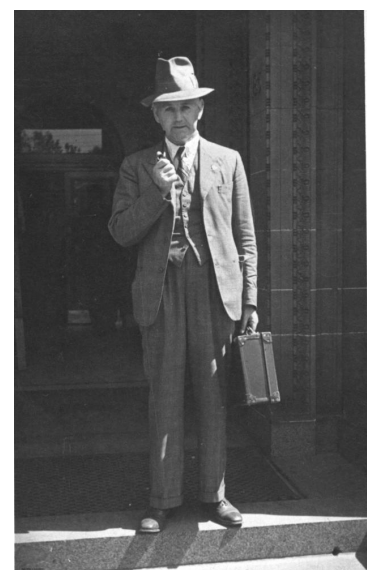
$$mv(m) = \begin{cases} 0, & \text{kui } m = 0 \\ \max_{m_i \leq m} (v_i + mv(m - m_i)) \end{cases}$$

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 optimeerimisülesanne 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 optimeerimisülesandeid. 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 lõpeb 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 $i+1, i+2, \dots, n$

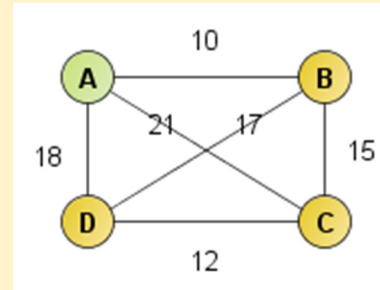
Väljundisse kirjutada üks täisarv: lühima tee pikkus, kui tee algab ja lõpeb esimeses linnas.

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-ülejäanud linnad ja A-C-B-D-ülejäanud 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 $L_0 \dots L_{n-1}$ ja mingi linnade hulk $V \in \{L_0, \dots, L_{n-1}\}$. (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:

$$mk(L_i, V) = \begin{cases} (L_0, L_i), & \text{kui } V = \emptyset \\ \min_{\forall L_j \in V} ((L_j, L_i) + mk(L_j, V \setminus \{L_j\})) \end{cases}$$

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:

$$\begin{aligned} &mk(A, \{B, C, D\}) = \\ &= \min((B, A) + mk(B, \{C, D\}), (C, A) + mk(C, \{B, D\}), (D, A) + mk(D, \{B, C\})) = \\ &= \min(10 + 45, 21 + 39, 18 + 37) = 55 \end{aligned}$$

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 $V \setminus \{L_i\}$

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 $O(n 2^n)$

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 $uxv \rightarrow uyv$

kustutamine: üks märk eemaldatakse: $uxv \rightarrow uv$

lisamine: üks märk lisatakse: $uv \rightarrow uxv$

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:

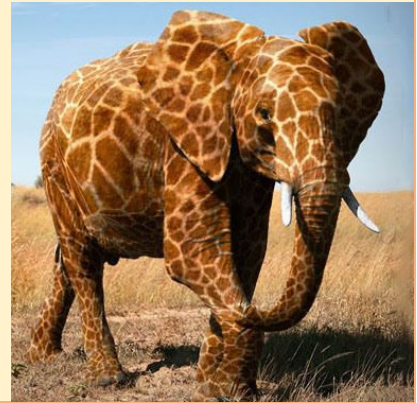
7 10

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:

$$LK_{a,b}(i, j) = \begin{cases} \max(i, j), & \text{kui } \min(i, j) = 0 \\ \min \begin{cases} LK_{a,b}(i-1, j) + 1 \\ LK_{a,b}(i, j-1) + 1 \\ LK_{a,b}(i-1, j-1) + 0, & \text{kui } a_i = b_j \\ LK_{a,b}(i-1, j-1) + 1, & \text{kui } a_i \neq b_j \end{cases} \end{cases}$$

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 asendamis(ed) (või kokkulangemised – kulu on 0), paremale või alla liikudes eemaldamis(ed) ja lisamis(ed). 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 (maksimumsega 0), kattuvuse hinnaks on 1. Kui lubatud on ainult asendamised (stringid peavad siis muidugi olema sama pikkusega), nimetatakse sellist kaugust **Hammingi** kauguseks ning selle leidmine on triviaalne.

Mõnikord lisatakse operatsioonide hulga veel **transpositsioon** ehk kahe kõrvuti asuva tähe vahetamine, $uxyv \rightarrow uy xv$. Sellisel juhul omandab rekursioonivalem järgmise kuju:

$$LK_{a,b}(i, j) = \begin{cases} \max(i, j), & \text{kui } \min(i, j) = 0 \\ \min \begin{cases} LK_{a,b}(i-1, j) + 1 \\ LK_{a,b}(i, j-1) + 1 \\ LK_{a,b}(i-1, j-1) + 0, & \text{kui } a_i = b_j \\ LK_{a,b}(i-1, j-1) + 1, & \text{kui } a_i \neq b_j \\ LK_{a,b}(i-2, j-2) + 1, & \text{kui } a_{i-1}a_i = b_jb_{j-1} \end{cases} \end{cases}$$

8.4 GEENIJÄRJESTUSTE VÕRDLEMINE

1970. aastal kasutasid Saul B. Needleman ja Christian D. Wunsch esmakordselt dünaamilist planeerimist kasutavat algoritmi bioinformaatikas nukleotiidjärjestuste vastavusse seadmiseks ehk joondamiseks.



8.4.1 Ülesanne: Sarnasus

On antud kaks nukleotiidide järjestust ning need tuleb seada üksteisega vastavusse, nii et võimalikult palju järjestitest 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 järjestite 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

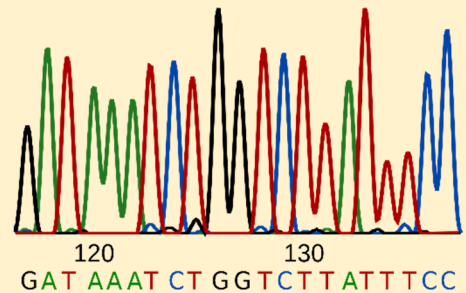
Vastus:

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 Wunshi 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 soovitud operatsioonid on negatiivse hinnaga, soovitud (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:

$$SW_{a,b}(i,j) = \begin{cases} 0, & \text{kui } i = 0 \vee j = 0; \\ \max \begin{cases} SW_{a,b}(i-1,j) + LH \\ SW_{a,b}(i,j-1) + LH \\ SW_{a,b}(i-1,j-1) + KH, & \text{kui } a_i = b_j \\ SW_{a,b}(i-1,j-1) + MH, & \text{kui } a_i \neq b_j \end{cases} \end{cases}$$

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

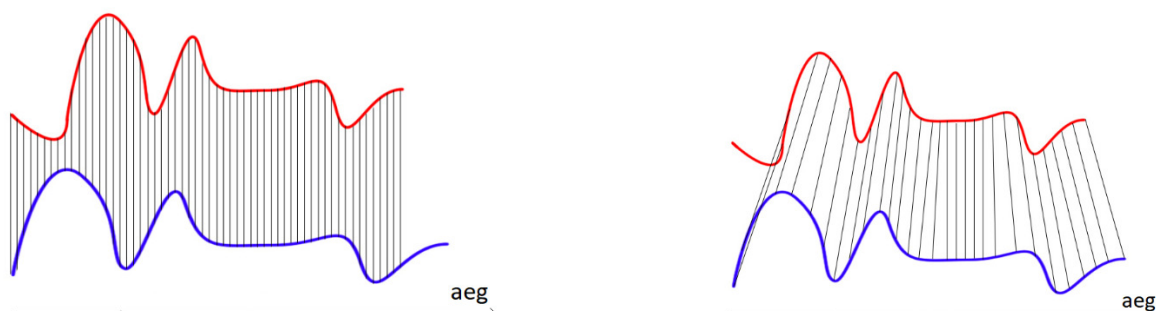
Tee taastamiseks tuleb leida suurima väärtusega lahter tabelist (seda on kaval jooksvalt meeles pidada) ning sealt saab juba tagurpidi tagasi tülles leida operatsioonid, mida kasutati. Kui jõutakse lahtrisse, kus on 0, siis on algus leitud ja rohkem kuhugi liikuda pole vaja. Seega vastus ülesandele võib asuda suvalises tabeli lahtris ning taastamisteegi ei pea jooksma ilusti tagasi tabeli algusesse, nagu klassikalisele toimetamiskauguse algoritmile omane. Ülesande lahenduseks on näiteks järgmine joendus (parim kokkulangev lõik on märgitud rohelisega):

GTTGACTA
TGT-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 punkte, 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

$$T(i, j) = \begin{cases} 0, & \text{kui } i = 0 \text{ ja } j = 0 \\ \infty, & \text{kui } i = 0 \text{ või } j = 0 \\ \min \begin{cases} T(i-1, j) + d(a_i, b_j) \\ T(i-1, j-1) + d(a_i, b_j) \\ T(i, j-1) + d(a_i, b_j) \end{cases} \end{cases}$$

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 lõigataks 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-meetrisse palki, kui aga näiteks 5m kauguselt, siis 5-meetrisse 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 $0 \leq i < j \leq n + 1$. 0 on palgi vasak ja $n + 1$ parem otspunkt. Otspunktide vahel on lõikekohad $1 \dots n$, järjestatud rangelt vasakult paremale. Sellisel juhul

$$h(i, j) = \begin{cases} 0, & \text{kui } i + 1 = j \\ p(i, j) + \min_{k \in \{i+1, \dots, j-1\}} (h(i, k) + h(k, j)) & \text{muud korrad} \end{cases}$$

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 tavapärase 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 jupitamise 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 optimeeritud.

8.7.1 Ülesanne: maatriksid

On antud n maatriksit $A_1, A_2, \dots, A_n$. 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 $A * B$ 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

$P_0 \dots P_n$, kus P_{i-1} ja P_i tähistavad i . maatriksi mõõde.

Väljastada vähim operatsioonide arv.

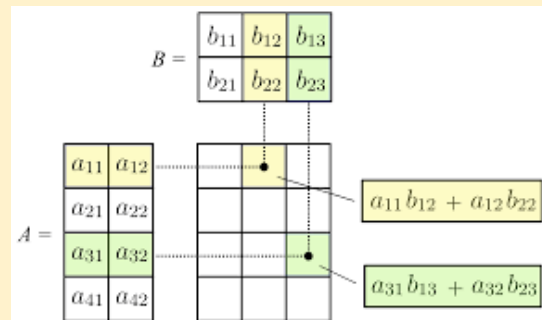
NÄIDE:

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 $A_i \times A_{i+1} \times \dots \times A_j$ leidmiseks vajalike tehete arvu $op(i, j)$. Siis

$$op(i, j) = \begin{cases} 0, & \text{kui } i = j \\ \min_{\forall k \in \{i, \dots, j\}} (op(i, k) + op(k + 1, j) + P_{i-1} \cdot P_k \cdot P_j) & \end{cases}$$

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 $A_i \dots A_n$.

	A1	A2	A3	A4	A5
A1	0				
A2	$[A1, A1] + [A2, A2] + P0 \cdot P1 \cdot P2 = 70$	0			
A3	$\min([A1, A2] + [A3, A3] + P0 \cdot P2 \cdot P3; [A1, A1] + [A2, A3] + P0 \cdot P1 \cdot P3) = \min(70 + 140, 56 + 40) = 96$	$[A2, A2] + [A3, A3] + P1 \cdot P2 \cdot P3 = 56$	0		
A4	$\min([A1, A3] + [A4, A4] + P0 \cdot P3 \cdot P4; [A1, A2] + [A3, A4] + P0 \cdot P2 \cdot P4; [A1, A1] + [A2, A4] + P0 \cdot P1 \cdot P4) = \min(96 + 60, 70 + 84 + 105, 80 + 30) = 110$	$\min([A2, A3] + [A4, A4] + P1 \cdot P3 \cdot P4; [A2, A2] + [A3, A4] + P1 \cdot P2 \cdot P4) = \min(56 + 24, 84 + 42) = 80$	$[A3, A3] + [A4, A4] + P2 \cdot P3 \cdot P4 = 84$	0	
A5	$\min([A1, A1] + [A2, A5] + P0 \cdot P1 \cdot P5; [A1, A2] + [A3, A5] + P0 \cdot P2 \cdot P5; [A1, A3] + [A4, A5] + P0 \cdot P3 \cdot P5; [A1, A4] + [A5, A5] + P0 \cdot P4 \cdot P5) = \min(110 + 50; 70 + 189 + 175; 96 + 60 + 100; 110 + 75) = 160$	$\min([A2, A4] + [A5, A5] + P1 \cdot P4 \cdot P5; [A2, A3] + [A4, A5] + P1 \cdot P3 \cdot P5; [A2, A2] + [A3, A5] + P1 \cdot P2 \cdot P5) = \min(80 + 30, 56 + 60 + 40, 189 + 70) = 110$	$\min([A3, A4] + [A5, A5] + P2 \cdot P4 \cdot P5; [A3, A3] + [A4, A5] + P2 \cdot P3 \cdot P5) = \min(84 + 105, 60 + 140) = 189$	$[A4, A4] + [A5, A5] + P3 \cdot P4 \cdot P5 = 60$	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 $[A1, A1] + [A2, A5] + P0 \cdot P1 \cdot P5$ 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

$$A_1 \times ((A_2 \times A_3) \times A_4) \times A_5).$$

8.7.5 Kood

Siin on funktsioon, mis antud tabeli täidab:

```
int matMul(int n, vector<int> m1)
{
    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]+(m1[i-1]*m1[k]*m1[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 ($1 \leq n \leq 2000$) hulknurkne ahel. i . tipu koordinaadid on $(i, y[i])$, kus i on kaugus aheliku algusest ja $y[i]$ ($0 \leq y[i] \leq 10^9$) tipu kõrgus.

Deevasid saab vangistada erinevatesse tippudes. Ainus tingimus on, et deevad ei tohi üksteist näha, sest siis saavad nad teha põgenemisplaanide. 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:

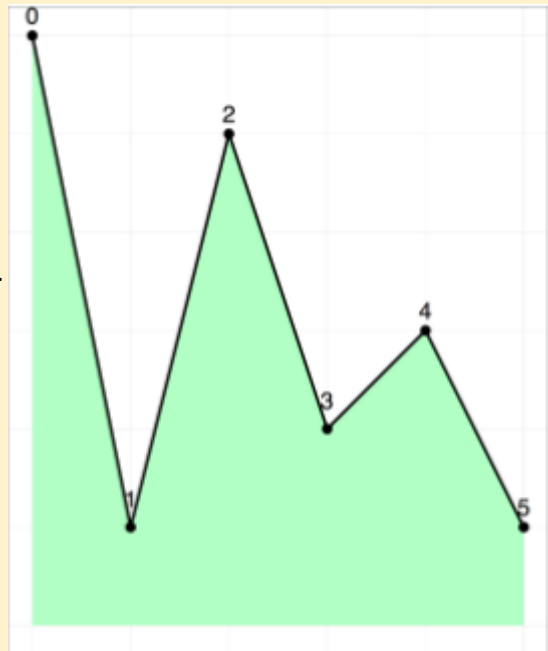
3 (Näiteks tippudele 1, 3 ja 5)

NÄIDE 2:

3
0 1 2

Vastus:

1



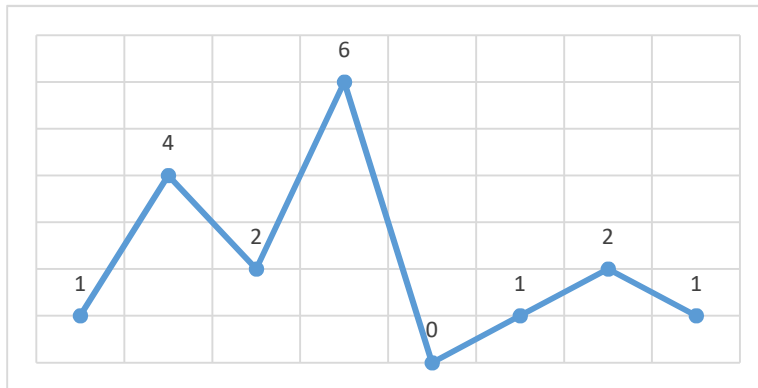
Esimese asjana on meil vaja vastust küsimusele kas tipp i paistab tipust j ? Näitame hiljem ülesande lahenduses, kuidas seda teha, esialgu lihtsalt eeldame, et meil on $n \times n$ maatriks, mis vastab antud küsimusele. Näiteandmete puhul on see siis järgmine:

	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 maskimaalselt 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. Deevaid 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 $[h, g]$, $h \leq g$ punktide ei ole tipust k nähtav. Saame järgmise rekursioonivalemi:

$$D(i, j) = \begin{cases} 1, & \text{kui } i = j \\ 1 + \max_{\forall k \in \{i \dots j\}} \sum_{\forall [h, g] \in LK_{ijk}} D(h, g) \end{cases}$$

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_deevis(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 $N(1 \leq N \leq 1000)$ erinevat sorti kasti, mis on nummerdatud arvudega $1 \dots N$ 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 ($1 \leq n \leq 50$) ristmikku, m ($1 \leq m \leq 1000$) kahe suunalist teed kahe ristmiku vahel, mille kohta on teada sõidukulu teed mööda sõitmiseks ning p ($1 \leq p \leq 12$) 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 mõtte ehitada kuristikule firmade vahele sildasid selleks, et nad saaksid ühineda ning oma tootlust kahekordistada. Samas on teada, et ei ole hea mõtte 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 ($1 \leq v \leq 1000$), 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 ($1 \leq p \leq 1000$) - 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 ($1 \leq n \leq 100\ 000$), 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 kahtluselise 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 ($2 \leq N \leq 40$) ning F ($-16000 \leq F \leq 16000$), leheküljel märgitud voog. Järgmisel N real on arv T ($1 \leq T \leq 400$), 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

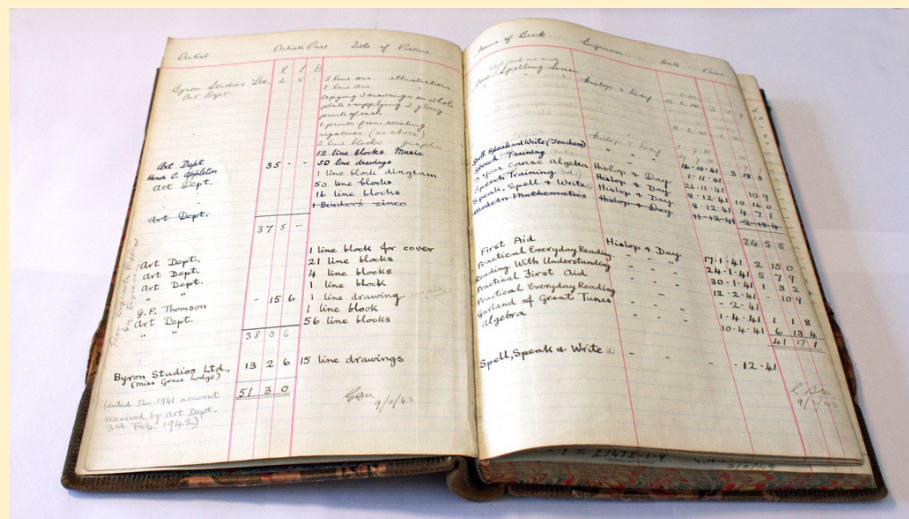
7

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 ($1 \leq n \leq 6$). 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 ($2 \leq k \leq 10$). 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: reisiril olemise päevade arv k ($1 \leq k \leq 21$), erinevate tegevuste arv n ($1 \leq n \leq 50$) ning Alberti eelarve m ($0 \leq m \leq 100$).

Järgneval n real on kaks tühikuga eraldatud täisarvu: c ($1 \leq c \leq 50$), tegevuse hind ning v ($1 \leq v \leq 10000$), 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 ($1 \leq n \leq 15$), 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 ($1 \leq n \leq 200$) ning m ($0 \leq m \leq n$), mis on hääletusel osalevate riikide arv ning Eesti valimiseks vajaminev hääle 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 ($D \leq 33$) 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 ($G \leq 33$) 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:

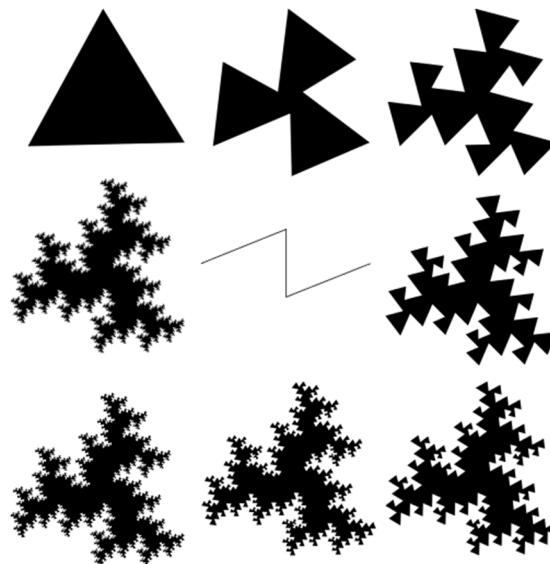
Fail	Kirjeldus
Kohver.cpp, Kohver.java, Kohver.py	0/1 seljakoti ülesanne
Kaupmees.cpp, Kaupmees.java, Kaupmees.py	Rändkaupmees Held-Karpi algoritmiga
Levenshtein.cpp, Levenshtein.java, Levenshtein.py	Levenshteini kaugus ehk toimetamiskaugus
Geenid1.cpp, Geenid1.java, Geenid1.py	Needleman-Wunchi algoritm
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	Maatriksite 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 võistlus-programmeerimises konkreetselt 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

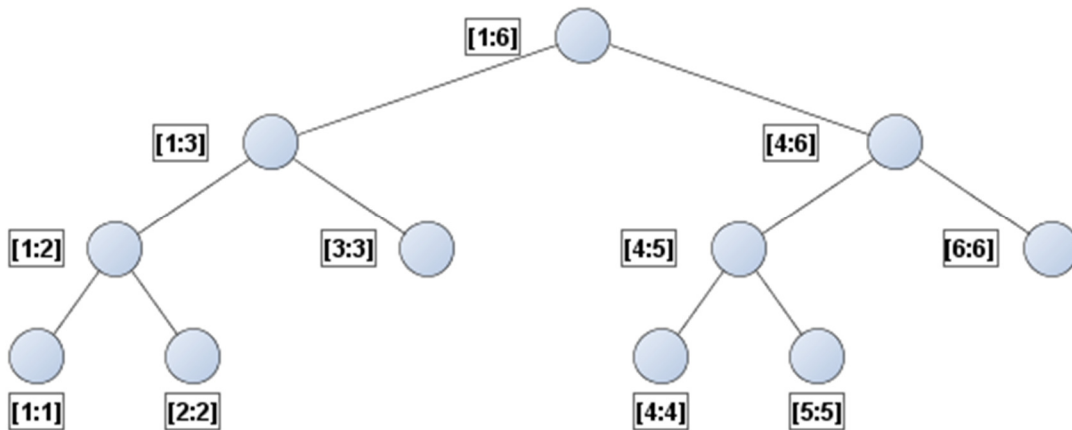
Vaatame ülesannet, kus on antud jada ja on vaja leida minimaalne element mingil suvalisel lõigul selles jadas. Sellist ülesannet tuntakse kui *Range Minimum Query (RMQ)*.

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 $m \gg n$, 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:

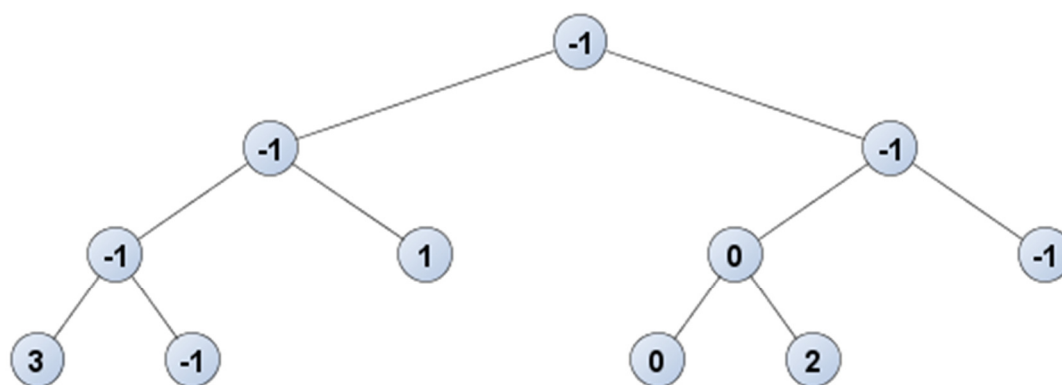
1	2	3	4	5	6
3	-1	1	0	2	-1

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 massiivis 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.

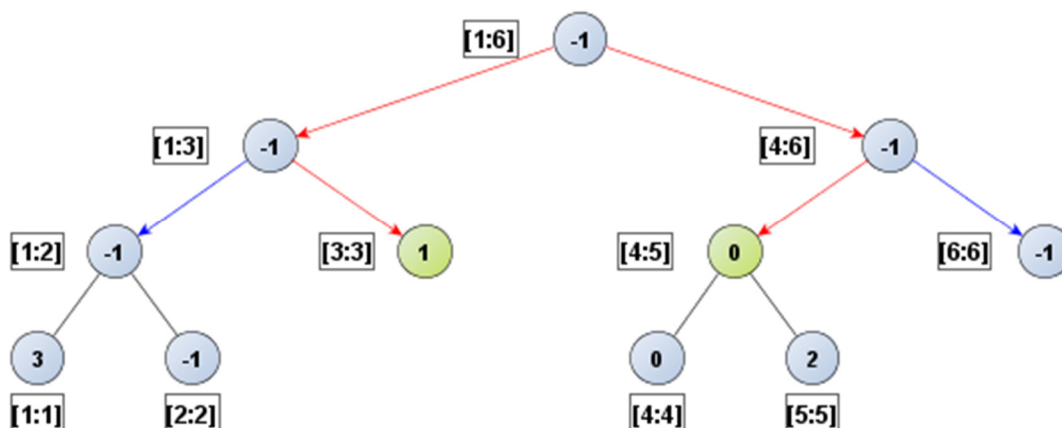
9.1.2 Lõigul väärtuse leidmine

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. Otsupunkt j asub vasakus alampuus ($j \leq \frac{n}{2}$). Sellisel juhul asub kogu uuritav lõik vasakus alampuus ja võime sealt jätkata.
2. Otsupunkt i asub paremas alampuus ($i > \frac{n}{2}$). Sellisel juhul asub kogu otsitav lõik paremas alampuus ja võime sealt jätkata.
3. Otsupunkt i asub vasakus alampuus ja otsupunkt 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 alampuu ja lõigu teine otspunkt 5 paremas alampuu. 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 alampuu 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 kasutada siin salvestatud väärtust ilma selle alampuudele liikumata. Lõik [6, 6], mis on lõigu [4, 6] paremaks alampuudeks, 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

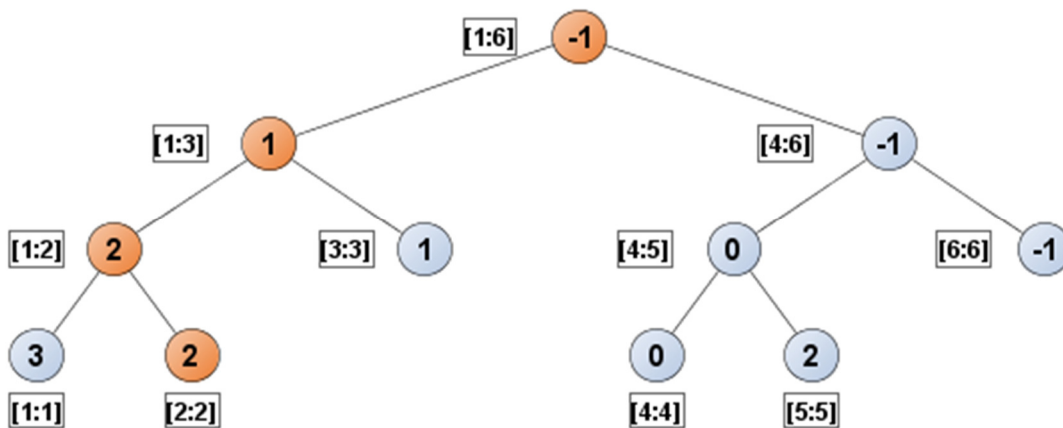
    if (vv <= pv)
        return vv;
    return pv;
}
```

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 väärtust, 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, töötlemine 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 $F(U[t])$.

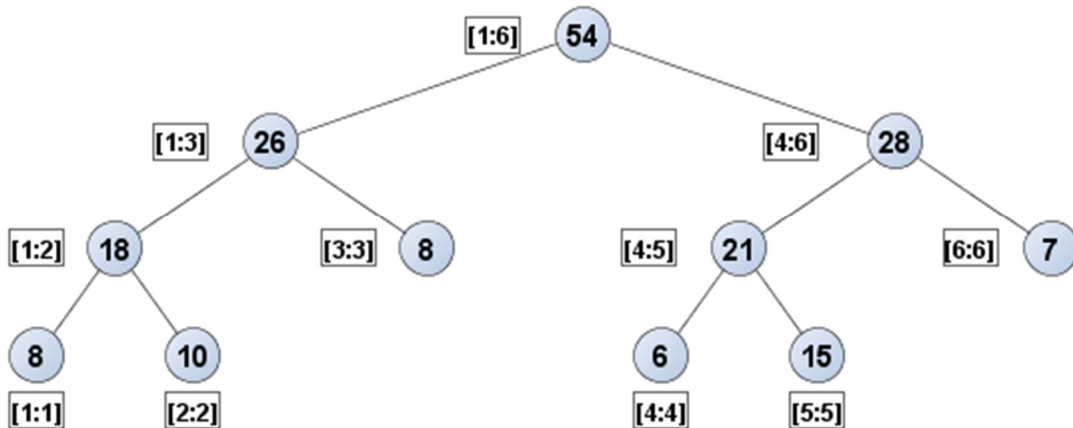
Lõikude puu elemendi muutmise algoritm on järgmine:

- 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:

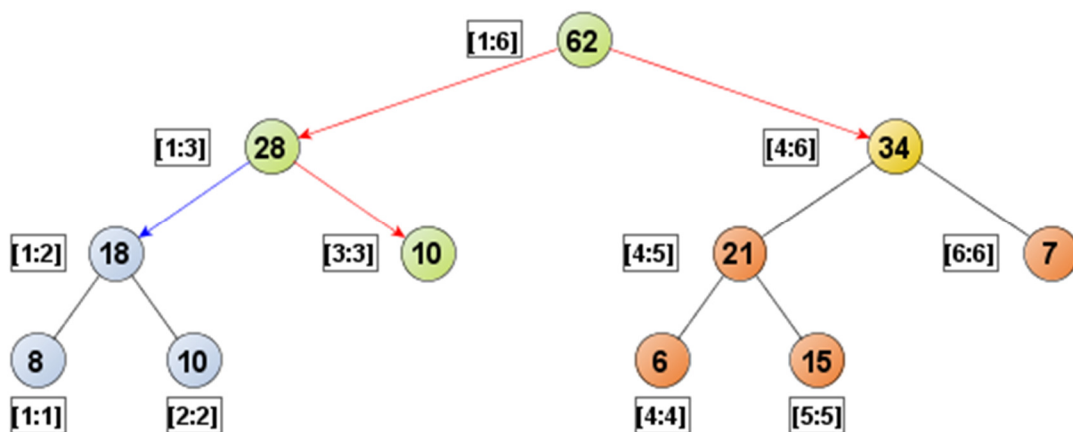


Lõikude puu, mille tippudes on vastava lõigu elementide summa. Juurtipus on kogu jada summa, lehtedes üksikud elemendid.

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.

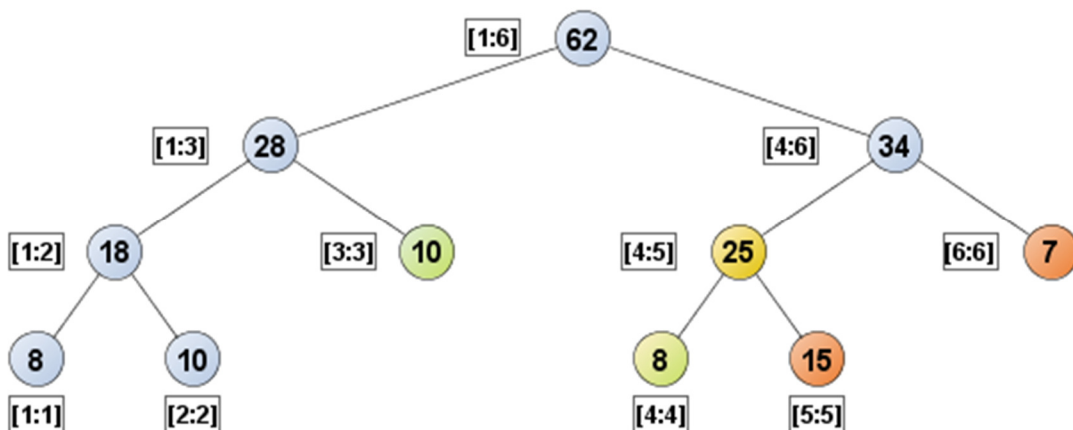
Alguses on meil uuenduste jada U elemendid määramata. Juurtipp ei asu tervenisti selles vahemikus, samuti kuna $U[1]$ on määramata, ei ole sellel vaja teha uuendusi. Liigume juure alampuudele ning kontrollime, ega need uuendamist ei vaja. Praegu muidugi veel ei vaja. Vasak alampuu [1, 3] ei asu samuti tervenisti muudetavas vahemikus, liigume omakorda selle alampuudele. Vasak alampuu on tervenisti uuendatavast vahemikust väljas ega vaja uuendamist, sellega ei tee midagi. Parema alampuu aga on tervenisti muudetavas vahemikus, uuendame selle väärtuse ja samuti siis vahemiku [1, 3] väärtuse. Siis 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.

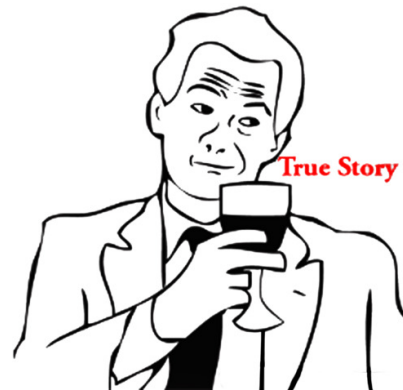
Ühe mugavusena koodi kirjutamisel saab antud näite juures ära kasutada ka seda, et nulli võib vabalt liita ja lahutada, ilma et summa sellest muutuks, see tähendab, et võime alguses väärtustada muutuste massiivi nullidega ning igale tipule liita massiivis oleva väärtuse ilma eraldi kontrollimata, kas 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.

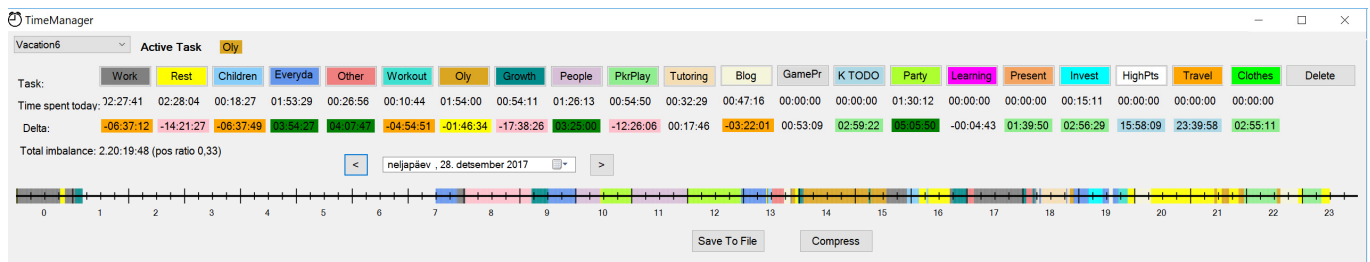
9.2.1 Paberiribade kleepimine

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, hobbisid ja sebmisi 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 ($1 \leq L \leq 10^9$) ning kleebitavate ribade arv N ($0 \leq N \leq 500\,000$). Järgmisel N real on igaühel kolm täisarvu: ühe riba värvikood K ($1 \leq K \leq 100$) ning selle alguse A ja lõpu B kaugus musta riba algusest ($0 \leq A < B \leq L$). 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.

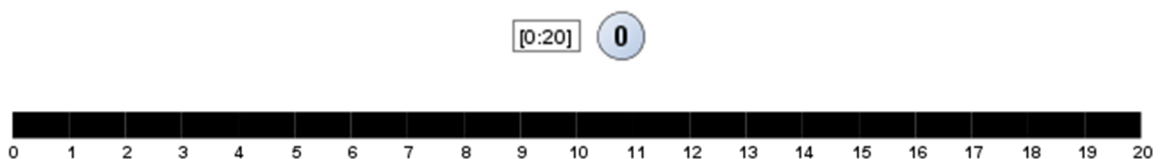
NÄIDE:

```
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):

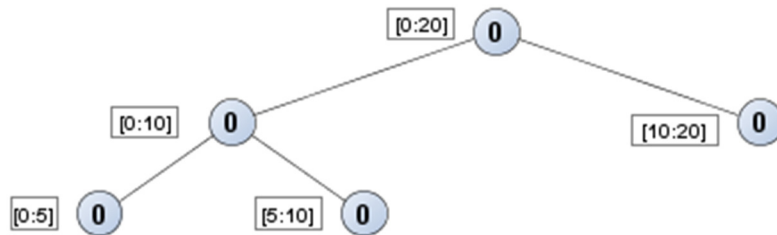


Nüüd on vaja lisada esimene riba värviga 1 (oletame, et see on punane). Kuna riba $[2, 10]$ ei kata kogu musta riba, on vaja liikuda selle alampuudele. Alampuud pärivad esialgu sama värvi, mis vanemal:

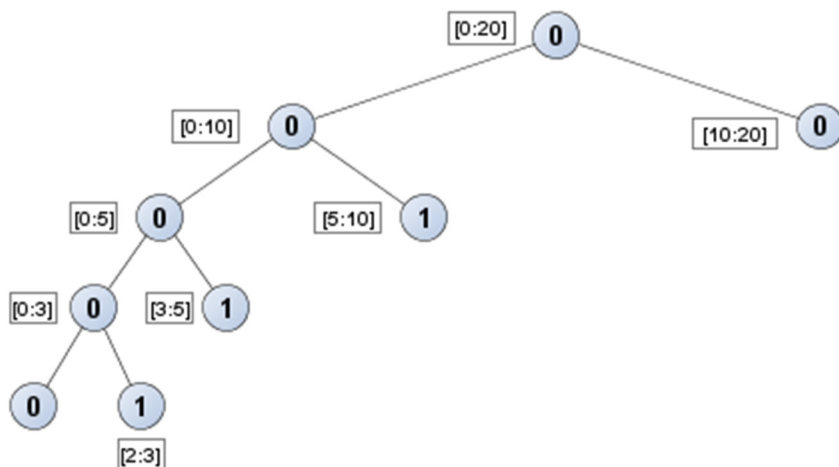


Vaatame kõigepealt, mis juhtub parempoolse alampuuga, mis vastab lõigule $[10, 20]$. Kuna see lõik jääb tervenisti kleebitavast ribast mõjutamata, siis jääb selle värviks must (0) ning sealt edasi liikuda

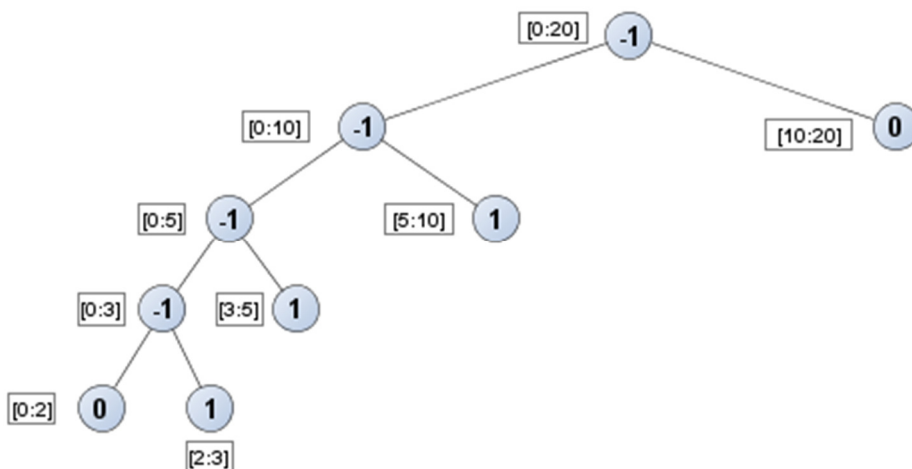
ei ole vaja. Vasakpoolsele alampuule vastav lõik $[0, 10]$ on osaliselt kleebitava ribaga kaetud, seega tuleb liikuda selle alampuudele:



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:

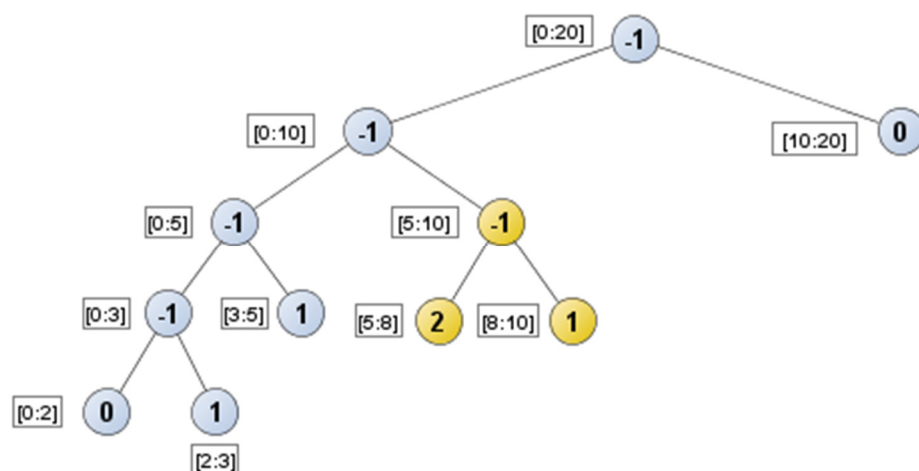


Nüüd on probleem selles, et mitmetel vanematel on värv erinev alluvate omast. Seega paneme selliste vanemate, millel on eri värvi alluvaid, värvi väärtuseks -1. Nii on puud läbi käies lihtne aru saada, mis toimub – kui lõigu värviks on -1, siis selle lõigu tegelikku värvi me ei tea ja peame uurima tema 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:

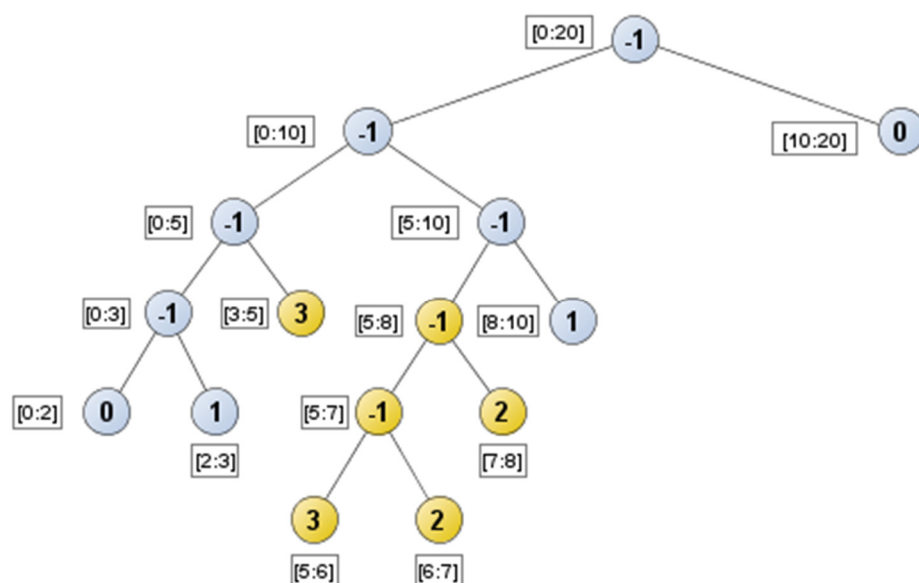




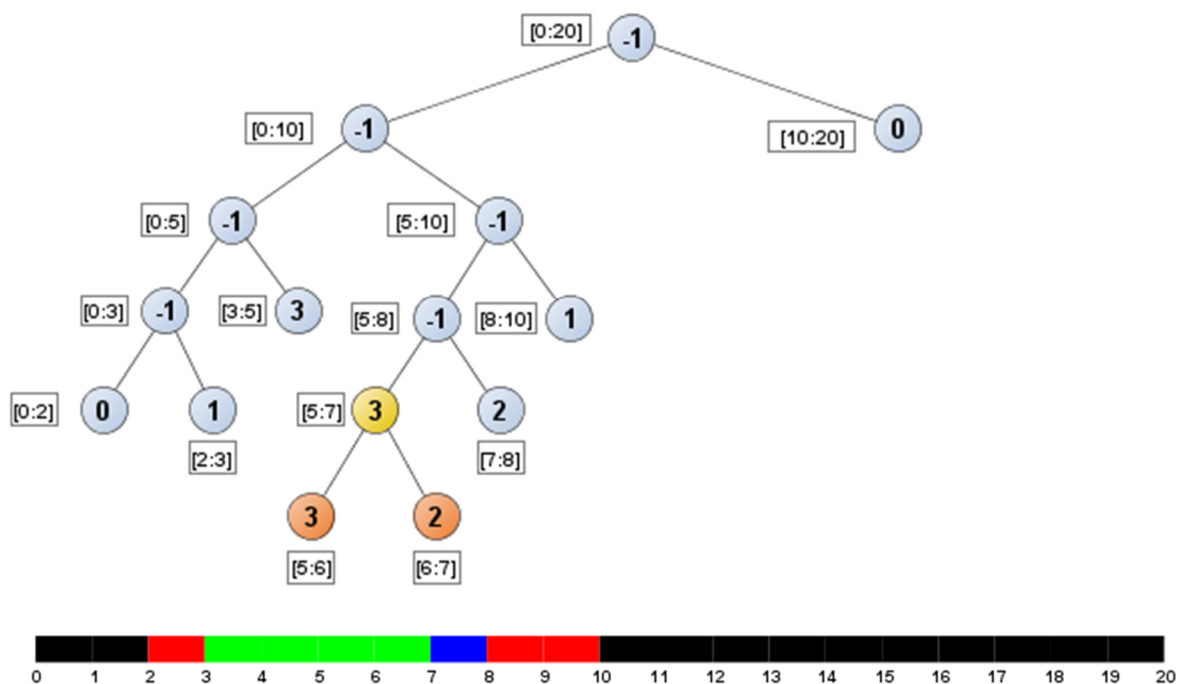
Teise riba (sinine) lisamise järgselt on pilt selline (kollased on muutunud väärtustega tipud):



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 Taipeiis 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 ($0 \leq n \leq 2\,000\,000$) – tulpade arv seinas ja k ($0 \leq k \leq 500\,000$) – 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 $v \leq p$, ja h ($0 \leq h \leq 100\,000$) – 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 terveniisti 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;
}

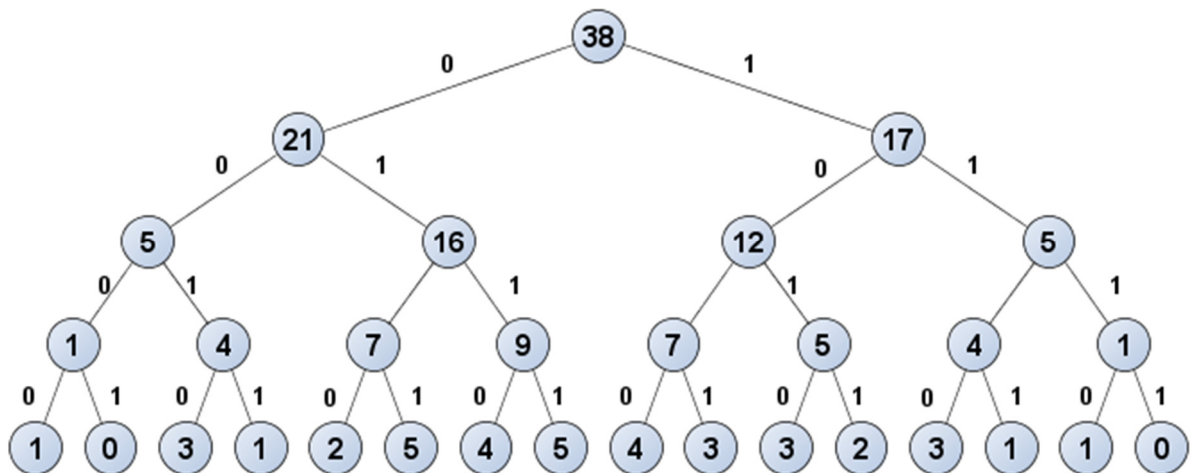
```

9.3 FENWICKI PUU

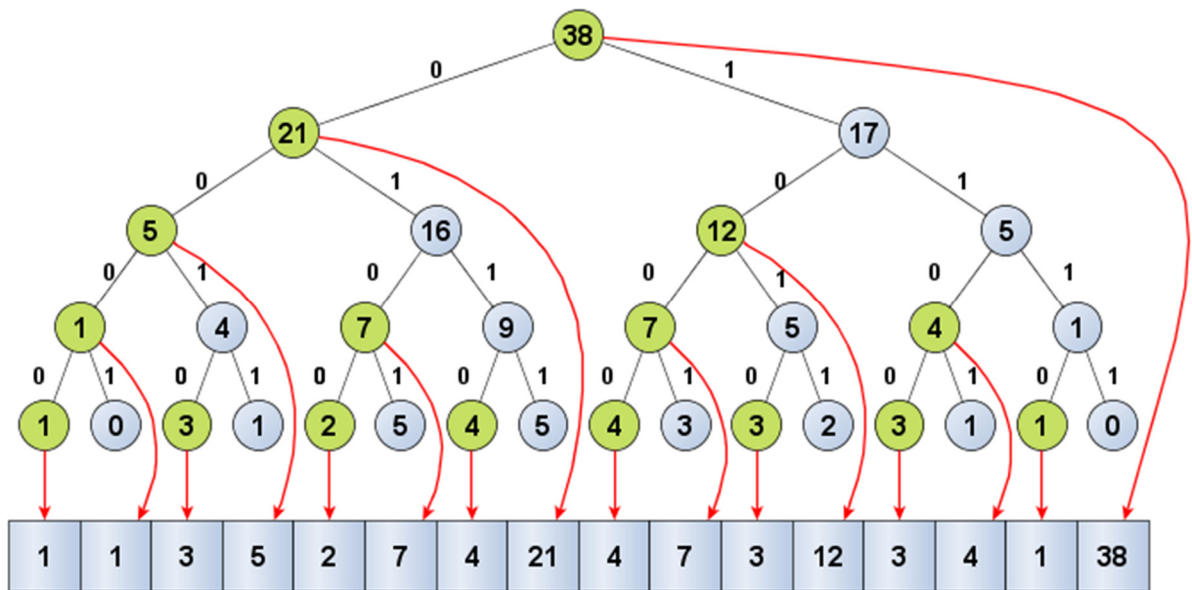
Fenwicki puu ehk **kahendindekseeritud puu** leiutas 1994. aastal Uus-Meremaa arvutiteadlane Peter Fenwick, kes uuris võimalusi andmepakkimisalgoritmide efektiivsuse suurendamiseks.

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 alamloikude 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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	3	5	4	9	4	23	4	7	3	12	3	4	1	40

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);
}
```

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

9.4 2-MÕÕTMELINE FENWICKI PUU

Vaatame järgmiseks ülesannet, kus on vaja punkte tasandile lisada ja eemaldada ning leida parasjagu mingisse ristikulikukujulisse 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.

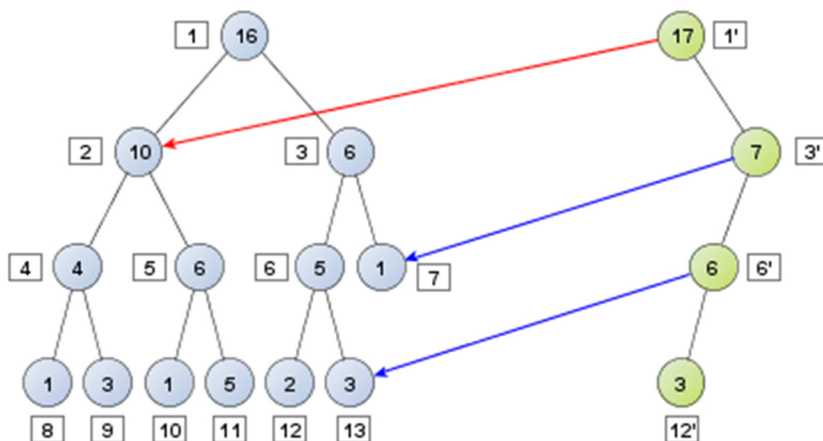
Üks võimalus selleks on nn **paksu elemendi** meetod – iga üksikelemendi asemel hoiame andmestruktuuris isekasvatavat massiivi (nt vectorit või listi). Kui vastavat väärtust muudetakse, lisatakse massiivi järgmine element. Paksu elemendi meetod sobib näiteks ajalooga massiivi loomiseks.

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 mees vaid see muudatuste puu – ülejäänud tipud on samad, mis eelmises versioonis.



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 mees 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 $i = 1, 2, \dots$) võtab tagasi viimase ülimalt $i-1$ taseme operatsiooni, mis pole veel tagasi võetud. Näiteks 1. taseme tagasivõtt saab tagasi võtta ainult redigeerimisoperatsioone, 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 X_1) operatsiooni seisus aktiivne ning muudame operatsiooni X_1 seisu tagasivõetud. Kui X_1 on samuti tagasivõtuoperatsioon, siis tuleb meil ka see operatsioon, mille X_1 tagasi võttis (tähistame selle X_2), 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 ($-n \leq a_i \leq n$, $a_i \neq 0$). 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 redigeerimisoperatsiooni, 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 <stdio>

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:

$$T[i, j] = F_{k=i \dots i+2^j-1} \{Jada[k]\}, \text{ kus } 1 \leq i \leq n, 1 \leq j \leq \lfloor \log n \rfloor$$

Kiiremaks koostamiseks saab kasutada dünaamilist programmeerimist. Näiteks minimaalse elemendi leidmise korral on rekursioonivalem järgmine:

$$T[i, j] = F(T[i, j - 1], T[i + 2^{j-1}, j - 1])$$

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 terveni katavad otsitava lõigu $[i, j]$. Pikim tabeli lõik, mis mahub lõiku $[i, j]$ on pikkusega $2^{\lceil \log(j-i+1) \rceil}$.

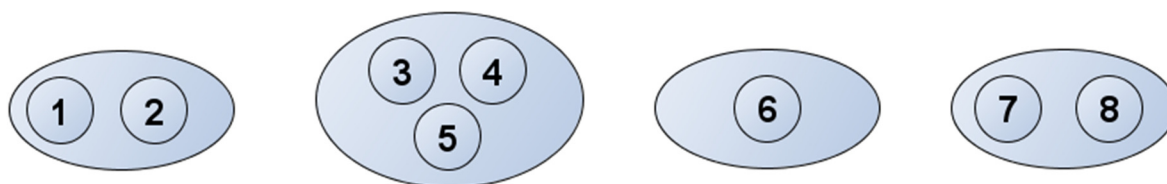
$$RMQ[i, j] = \begin{cases} T[i, k], & \text{kui } Jada[T[i, k]] \leq Jada[T[j - 2^k + 1, k]] \\ T[j - 2^k + 1, k] \end{cases}$$

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.

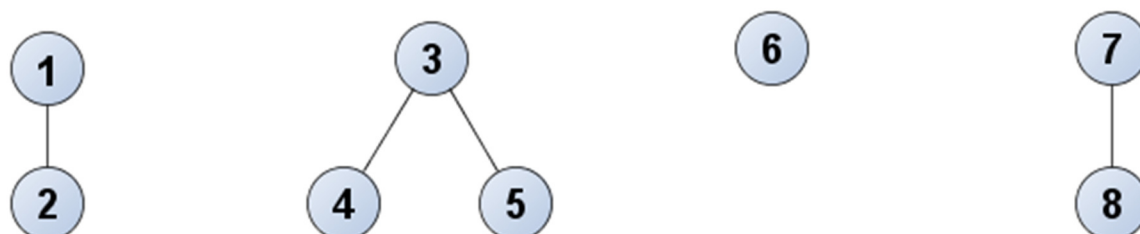
Mõnikord on aga vaja tervet hulkade komplekti nii, et iga element esineb mitte rohkem kui ühes hulgas. Sellist struktuuri võib nimetada ühisosata hulkadeks (i. k. *disjoint sets*).



Peamised operatsioonid, mida on vaja sooritada, on hulga loomine, kahe hulga ühendamise ning leidmine, millisesse hulka element kuulub.

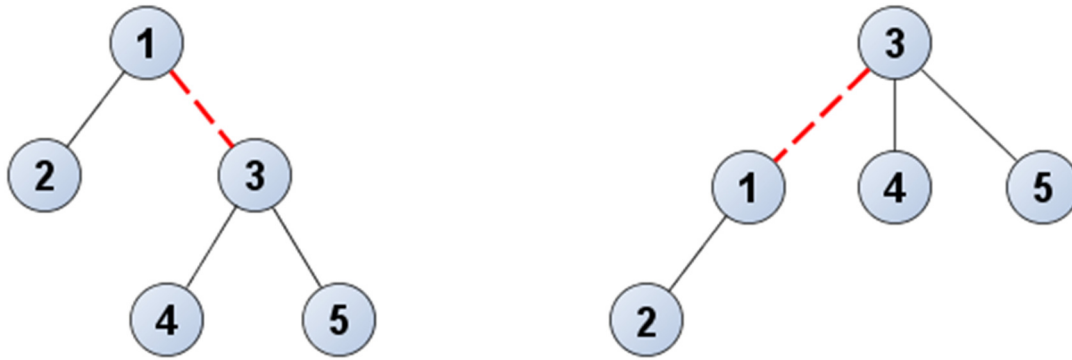
Abstraktselt on tegemist küllaltki lihtsa struktuuriga, küsimus on, et kuidas hoida hulki ja nende elemente nii, et mainitud operatsioonid oleksid võimalikult efektiivsed.

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 induktiooniga. 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 $2^{2^{\dots^2}}$, kus siis astendatakse n korda. Selle funktsiooni väärtus kasvab väga kiiresti: 2, 4, 16, 65 536, $2 \cdot 10^{19728}$. 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 ($1 \leq n \leq 200000$) lahoonet. i -ndas lahoones on r ($0 \leq r \leq 1000$) võistlusprogrammeerimise õpikut. Vahetevahel lisatakse õpikuid lahoonesse juurde või võetakse neid sealt ära. Samuti tahetakse aeg-ajalt teada, mitu õpikut on mingites järjestikku asuvates lahoonetes kokku.

Sisendi esimesel real on täisarv n . Järgmisel n real on lahoonetes olevate raamatute arv. Sellele järgneb ühel real arv q ($1 \leq q \leq 200000$) – reas tehtavate operatsioonide arv. Järgmised q rida on kas kujul „L x r“ ($1 \leq x \leq n$; $1 \leq r \leq 1000$) – tehakse operatsioon, mille tagajärjel on lahoones x r raamatut, või „U x y“ ($1 \leq x \leq y \leq n$) – uuritakse, mitu õpikut on lahoonetes 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 ($1 \leq N, K \leq 10^5$). Teisel real on N täisarvu – esialgse jada väärtused x ($-100 \leq x \leq 100$). Järgmisel K real on käsud, mis on kas muutmise või korrutamise käsud. Muutmise käsk antakse ette kujul $M \ x \ y$ ($1 \leq x \leq N$; $-100 \leq y \leq 100$), mis tähendab, et arv kohal x on nüüd väärtusega y . Korrutamise käsk antakse ette kujul $K \ x \ y$ ($1 \leq x \leq y \leq N$), 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 ($1 \leq n, q \leq 100000$). Järgmisel real on n täisarvu vahemikus $-100000 \dots 100000$ – majade kõrgused merepinnast. Järgmisel q real on kaks täisarvu: x ja y ($1 \leq x \leq y \leq n$) – 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, kõrgendamiseks. 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 ($1 \leq n \leq 100000$) – vajalike müürilõikude arv.

Järgmisel n real on kolm täisarvu: l , r ja h ($1 \leq l < r \leq 100000$; $1 \leq h \leq 10^9$) – müürijupi otspunktid ning kõrgus.

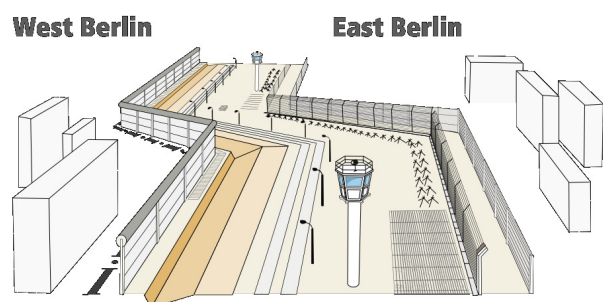
Väljundisse kirjutada üks täisarv: töö alustamiste arv.

NÄIDE:

```
3
5 11 3
1 10 1
3 13 2
```

Vastus:

```
14
```



9.8.5 Lambipirnid

Reas on N ($1 \leq N \leq 1000000$) 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 ($1 \leq q \leq 1000$) – päringute arv, mis tehakse. Järgmisel q real on tähemärk ning kaks arvu kujul $c \ a \ b$ ($0 \leq a \leq b < n$). 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 Sorø. See nn Sorø müür on $1 = 10^7$ millimeetrit pikk, kus iga millimeetri järel on tehtud märged. Plakat, mis seinale riputatakse, peab olema riputatud täpselt märkmest märkmeni ning selle pikkus peab olema vähemalt kaks millimeetrit. Valimiste eel riputatakse Sorø müürile n ($1 \leq n \leq 40000$) plakati. Kuna plakat või osaliselt või täielikult katta ka teist plakati, leida, mitu plakati on valimispäeval osaliselt või täielikult nähtaval.

Sisendi esimesel real on täisarv n . Järgneval n real on kaks täisarvu:

v ja p ($1 \leq v < p \leq 1$) – 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 $i \dots j$.

Sisendi esimesel real on kaks arvu: n ja m ($1 \leq n \leq 100000$, $1 \leq m \leq 5000$). Sisendi teisel real on jada pikkusega n – reas seisvate laste pikkused. Järgneval m real on kolm arvu:

i, j ja k ($1 \leq i \leq j \leq n$, $1 \leq k \leq j - i + 1$).

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 ($N, M \leq 100000$). 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 Platvormid

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 ($0 \leq N \leq 500$) – toa küljepikkus. Järgneval N real on igal real N tühikutega eraldatud täisarvu – platvormide esialgsed kõrgused. Järgmisel real on täisarv Q ($0 \leq Q \leq 40000$). 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

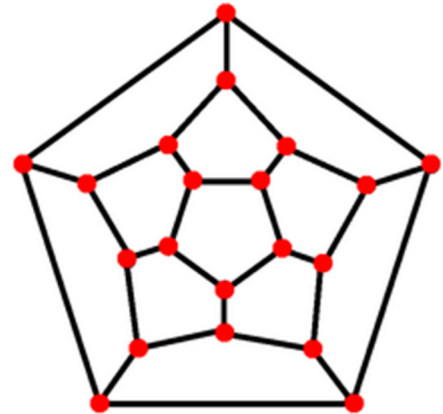
Kaasasolevas failis VP_lisad.zip, peatükk10 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

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 $O(T + S * \log S)$, kus T on tippude ja S servade arv. Läbides seda algoritmi T korda, on keerukuseks $O(T^2 + T * S * \log S)$. 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: Kauguste maatriks

Ühes riigis on N linna, millest mõned on omavahel ühendatud otseteedega. Sulle on antud $N \times N$ tabel, kus kohal $[i, j]$ on kirjas otsetee pikkus linnast i linna j , kui selline tee eksisteerib. Kui teed ei eksisteeri, on tabelis kirjas -1 .

Leida ja väljastada kauguste tabel, kus kohal $[i, j]$ on kirjas lühima tee pikkus linnast i linna j .

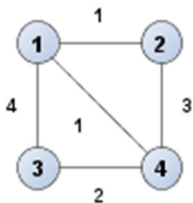
NÄIDE:

```
0 1 4 1
1 0 -1 3
4 -1 0 2
1 3 2 0
```

Vastus:

```
0 1 3 1
1 0 4 2
3 4 0 2
1 2 2 0
```

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:

$$T(i, j, k) = \begin{cases} P(i, j), & \text{kui } k = 0 \\ \min(T(i, j, k-1), T(i, k, k-1) + T(k, j, k-1)) \end{cases}$$

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:

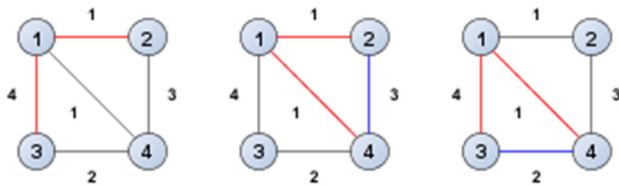
K=0	1	2	3	4
1	0	1	4	1
2		0	∞	3
3			0	2
4				0

Järgmisel sammul $k = 1$ proovime minna igast tipust igasse tippu läbi tipu 1.

$$T(2, 3, 1) = \min(T(2, 3, 0), T(2, 1, 0) + T(1, 3, 0)) = \min(\infty, 1 + 4) = 5$$

$$T(2, 4, 1) = \min(T(2, 4, 0), T(2, 1, 0) + T(1, 4, 0)) = \min(3, 1 + 1) = 2$$

$$T(3, 4, 1) = \min(T(3, 4, 0), T(3, 1, 0) + T(1, 4, 0)) = \min(2, 4 + 1) = 2$$



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:

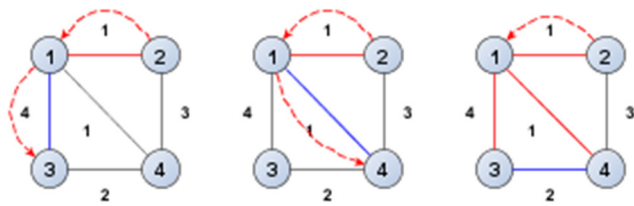
$$T(1, 3, 2) = \min(T(1, 3, 1), T(1, 2, 1) + T(2, 3, 1)) = \min(4, 1 + 5) = 4$$

$$T(1, 4, 2) = \min(T(1, 4, 1), T(1, 2, 1) + T(2, 4, 1)) = \min(1, 1 + 2) = 1$$

$$T(3, 4, 2) = \min(T(3, 4, 1), T(3, 2, 1) + T(2, 4, 1)) = \min(2, 5 + 2) = 2$$

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:

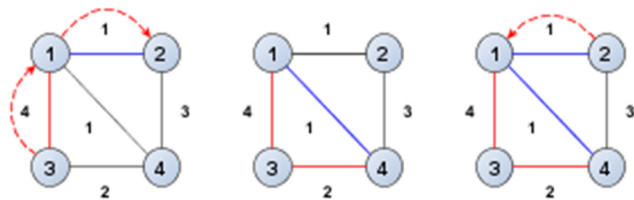
$$T(1,2,3) = \min(T(1,2,2), T(1,3,2) + T(3,2,2)) = \min(1, 4 + 5) = 1$$

$$T(1,4,3) = \min(T(1,4,2), T(1,3,2) + T(3,4,2)) = \min(1, 4 + 2) = 1$$

$$T(2,4,3) = \min(T(2,4,2), T(2,3,2) + T(3,4,2)) = \min(2, 5 + 2) = 2$$

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:

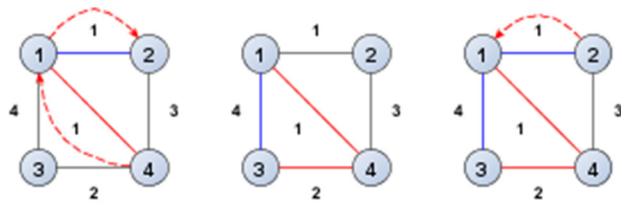
$$T(1,2,4) = \min(T(1,2,3), T(1,4,3) + T(4,2,3)) = \min(1, 1 + 2) = 1$$

$$T(1,3,4) = \min(T(1,3,3), T(1,4,3) + T(4,3,3)) = \min(4, 1 + 2) = 3$$

$$T(2,3,4) = \min(T(2,3,3), T(2,4,3) + T(4,3,3)) = \min(5, 2 + 2) = 4$$

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 jooksumist ei ole mõne i väärtuse puhul diagonaalil asuv $[i, i]$ väärtus enam lõpmatu, siis leidub graafis tsükel, mis läbib tippu i . Kõige väiksema väärtusega selline tsükel ongi minimaalne tsükel graafis. Kui mõni väärtus $[i, i]$ on negatiivne, siis on graafis negatiivne tsükel.

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.

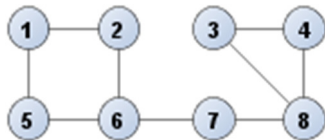
Lahendiks olevast kauguste tabelist saab lihtsalt näha, kas kahe punkti vahel üldse leidub tee või mitte. Kui ainult seda infot vaja on, võimegi tabelis meeles pidada vaid tee leidumist (või mitteleidumist). Taas on see praktiline ainult siis, kui tippude hulk on väike.

Sel moel aga saab tuvastada ka tugevalt sidusad komponendid suunatud graafis. Pärast algoritmi jooksumist kontrollime tulemuseks saadud tabelit. Näiteks mingi tipuga i kuuluvad samasse sidususkomponenti need tipud j , kus $[i, j] = [j, i]$.

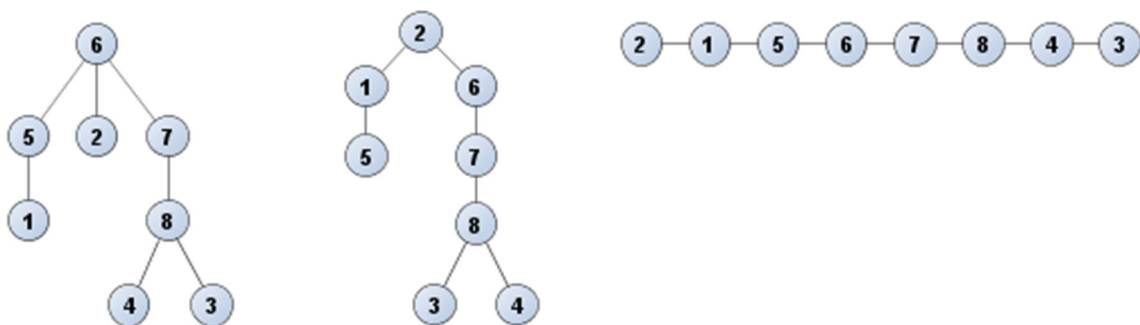
10.2 TOESE MÕISTE

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:

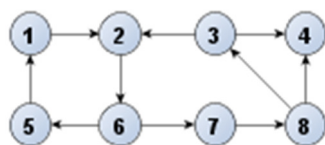


10.2.1 Servade liigitamine

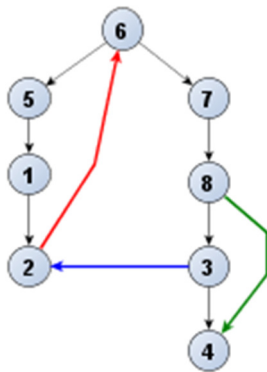
Graafi sügavuti läbides tekib nn sügavuti otsimise puu, mis on graafi aluspuu. Milline see puu täpselt on, sõltub sellest, millisest tipust sügavuti läbimist alustatakse ning mis järjekorras naabreid valitakse. Sügavuti otsimisel saab suunatud graafis eristada nelja tüüpi servi:

- 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.

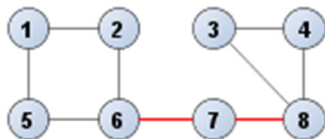
Funktsioon, mis liigita servad:

```
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 ARTIKULATSIIOONIPUNKTID

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.

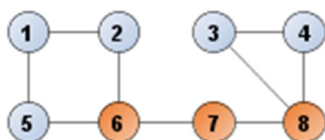
Näiteks graafis:



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 $0 - n-1$ arv n – dast lõkkest paistvate lõkete järjekorranumbrid. Väljastada nende lõkkekohtade järjekorranumbrid, mida ei tohi kindlasti jätta süütamata. Järjekorranumbrid väljastada kasvavas järjestuses.

NÄIDE:

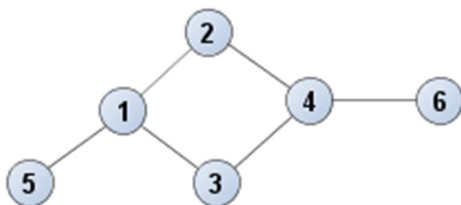
```
6
2 3 5
4
4
6
```

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.



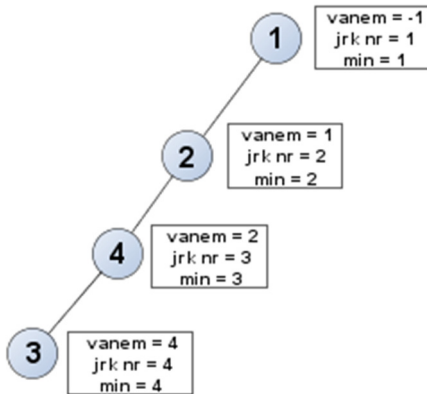
Ülesande näiteandmetele vastav graaf

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 $O(V * (V + E))$.

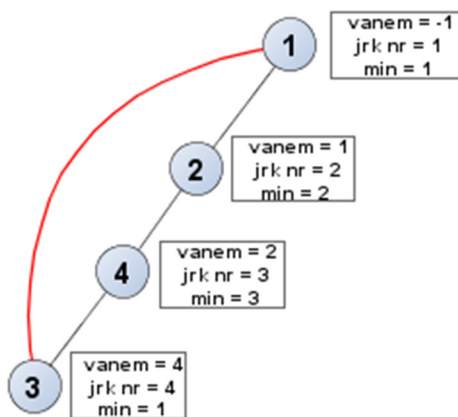
Saab ka paremini. Kasutades sügavuti läbimist, nimetame tipu u tipu v eellaseks, kui tippu v jõudmiseks läbisime tipu u . Tipp u on artikulatsioonipunkt kui

- 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.

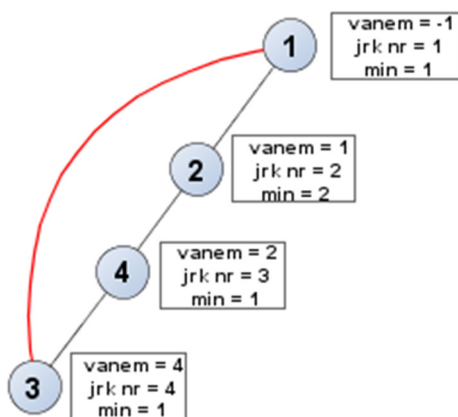
Nii on artikulatsioonipunkti leidmise algoritmi aluseks toeseppuu. Hakkame seda looma, aga iga tipu juures peame meeles lisaks kahte numbrit – tipu avastamise järjekorranumbrit ehk leidmise aega ja üht teist numbrit, mida nimetame minimaalseks leidmise ajaks – see on vähima järjekorranumbriga tipu, millesse tema alampuust jõuda võib, number. Lisaks peame meeles, kes on tipu vanem. Alustades tipust 1 ja võttes tippe järjekorras, saame esialgu järgmise toeseppuu alguse:



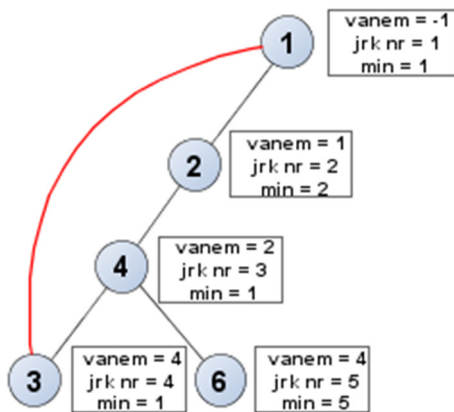
Nüüd tipust 3 läheb tagasiiviiv serv tippu 1. Uuendame tipu 3 minimaalset leidmise aega milleks on kas tipu 3 senine minimaalne leidmise aeg või tipu 3 naabri 1 järjekorranumber – see, mis on neist kahest väiksem. Saame:



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:

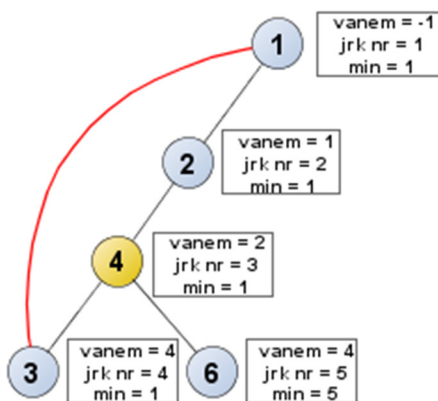


Hetkel tipp 4 ei kvalifitseeru artikulatsioonipunktiks, kuid sel tipul on veel naabreid, nimelt tipp 6:

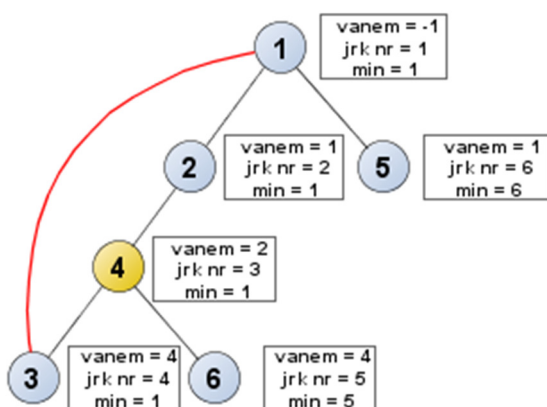


Kuna tipul 6 naabreid ei ole, siis sellega midagi teha ei ole vaja. Selline tipp ei saa olla ka artikulatsioonipunktiks. Küll aga on artikulatsioonipunktiks tipp 4, sest see on ainus tipp, mille kaudu saab tippu 6. Märgime tipu 4 artikulatsioonipunktiks (joonisel edaspidi kollane).

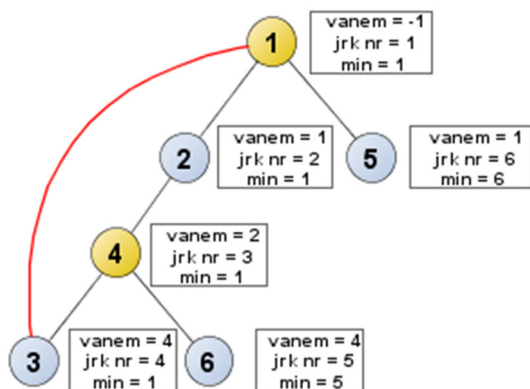
Tipu 4 naabrid on nüüd kõik üle vaadatud ja läheme tagasi tipu 2 juurde. Siin saame taas parandada madalaimat aega tipu 4 andmete põhjal ning otsustada, et 2 hetkel ei ole artikulatsioonipunktiks:



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 artikulatsioonipunktiks.



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 artikulasioonipunktid
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]; // hoia me iga tipu jaoks meeles tema vanema

void leiaAPd(int u)
{
    int alluvaid = 0; // loendame alluvaid alampuusi
    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 artikulasioonipunkt, 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;
        }

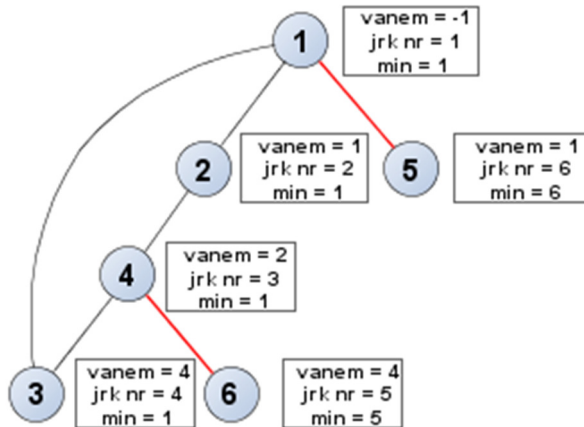
        else if (v != vanemad[u]) // kui v ei ole u vanem
            madalaim[u] = min(madalaim[u], aeg[v]);
    }
}

```

10.3.2 Sildade leidmine

Sildade leidmise algoritm on sarnane artikulatsioonipunktide leidmisele. Tipp u on artikulatsioonipunkt siis, kui $\text{madalaim}[v] \geq \text{aeg}[u]$, serv (u, v) on sild siis, kui $\text{madalaim}[v] > \text{aeg}[u]$. 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]);

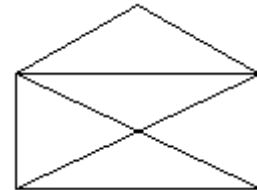
            //Kas v-l on ühendus mõne u eellasega? Kui mitte, siis on u-v sild
            if (madalaim[v] > aeg[u])
                cout << u << " " << v << endl;
        }

        // 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 pliatsit paberilt tõstmata ja ühtki joont topelt tõmbamata. Selleks on rohkem kui üks võimalus, aga neid kõiki ühendab see, et alustada tuleb ühest alumisest punktist ning lõpetada teises alumises punktis.

Graafiteoorias nimetatakse graafis sellist teed ehk ahelat, mis läbib iga selle graafi serva täpselt üks kord, **Euleri ahelaks**.



Euleri tsüklik nimetatakse sellist Euleri ahelat, mis algab ja lõpeb 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 lõpeb 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 tsükklis 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 ülejäänud 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

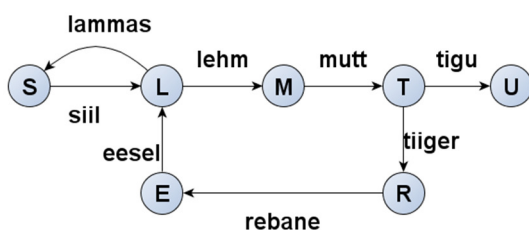
Vastus:

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:

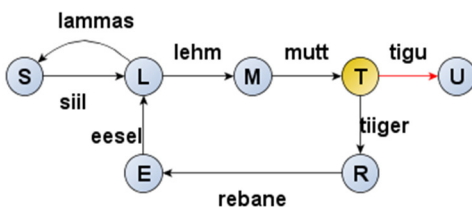


Lahendiks on Euleri tee selles graafis, kui selline leidub. Seega teiseks ülesandeks on teha kindlaks, kas see tee leidub ning ühes sellega, milline on tee alguspunkt. Seda teeb järgnev funktsioon:

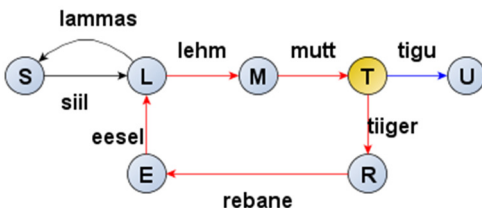
```
bool onTee()
{
    int alguseid = 0;
    int loppe = 0;
    for (int i = 0; i < 26; i++) {
        int valja = graaf[i].size();
        if (sisse[i] > valja + 1 || sisse[i] + 1 < valja)
            return false; //sisse ja väljatulevate vahe on rohkem kui 1
        if (sisse[i] == valja + 1) {
            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.

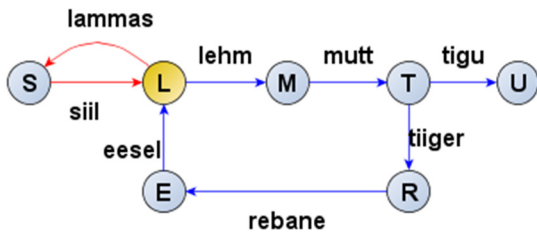
Alustame tipust T ja võtame suvalise serva, näiteks „tigu“:



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 { // jõudsim 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();
        }
    }

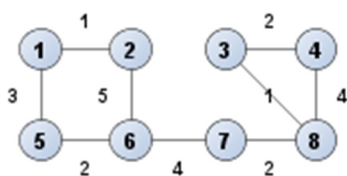
    // tee väljastamine
    for (int i = tee.size() - 1; i > 0; i--) {
        cout << servad[tee[i]][tee[i - 1]].top() << endl;
        servad[tee[i]][tee[i - 1]].pop();
    }
}

```

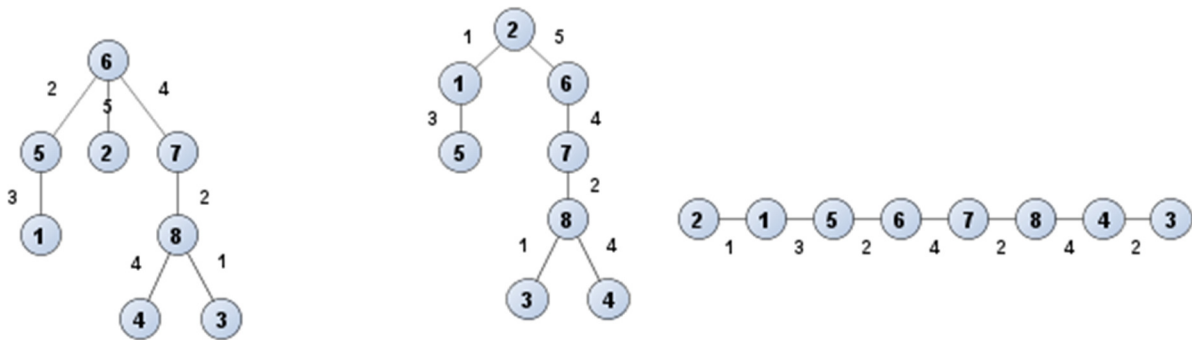
10.5 MINIMAALNE TOES

Minimaalsest toesest saab rääkida siis, kui tegemist on kaalutud graafiga. Nimetame toese puu kaaluks kõigi tema servade kaalude summat. Erinevate toese puude kaalud võivad olla erinevad.

Näiteks graafis

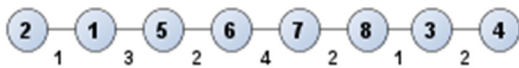


on eelpool toodud toese puude kaalud



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 tosepuu 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.

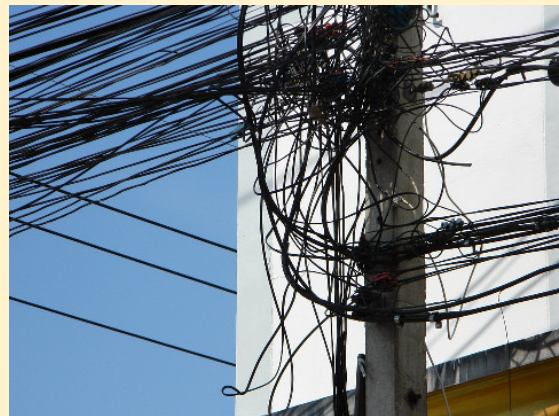
Väljastada üks täisarv minimaalne liinide kogupikkus, millega saab kõik tarbijad omavahel ühendada.

NÄIDE:

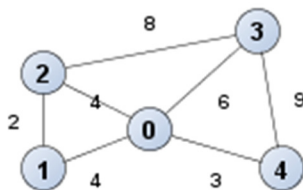
```
5 7
1 0 4
1 2 2
0 2 4
0 4 3
0 3 6
2 3 8
3 4 9
```

Vastus:

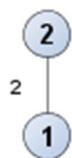
15



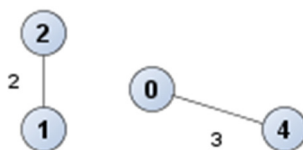
Esiteks, kuna tegu on graafiülesandega, tuleb luua graaf. Visuaalselt näeb näiteandmetel põhinev graaf välja selline:



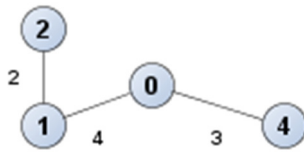
See on küllaltki lihtne ja läbipaistev minimaalse toese ülesanne. Kasutame ülesande lahendamiseks Kruskali algoritmi. Hakkame looma graafi aluspuud ja kõigepealt valime sinna kõige kasulikuma ehk kõige väiksema kaaluga serva, milleks on (1,2):



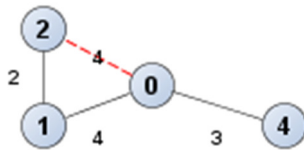
Järgmiseks vaatame kaalult järgmist serva (0,4). See ei tekita eelmise servaga tsüklit, lisame selle:



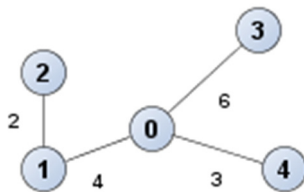
Nüüd on meil valida kahe serva vahel, mis on võrdse kaaluga: (0,1) ja (0,2). Proovime lisada neist esimese, mis ka õnnestub, kuna tsüklit ei teki:



Siin tasub tähele panna, et uus serv ühendas kaks enne iseseisvat komponenti. Seejärel proovime lisada serva (0,2). Seda aga toessesse lisada ei saa, kuna tipud 0 ja 2 on juba samas komponendis ja nii tekiks tsüklil:

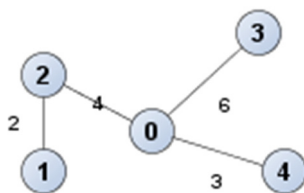


Seega serv (0,2) ei kuulu minimaalsesse toessesse, jätame selle kõrvale ja jätkame servaga (0,3). Selle saab taas lisada olemasolevale komponendile:



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 toeseppuu on leitud.

Kui oleksime valinud serva (0,2) enne serva (0,1), saanuksime järgmise minimaalse toeseppuu:



Seega ka minimaalseid toeseppuid võib olla mitu.

Konkreetses ülesandes teeb lihtsamaks asjaolu, et tegelikku toest leida pole vajagi – piisab selle kaalust. Seega toessesse kuuluvate servade meelespidamise asemel piisab, kui toessesse kuuluva serva leidmisel lisame selle serva kaalu summale. Kuna tegelikke minimaalseid toeseppuid 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 jrjm_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[jrjm_serv].first;
        pair<int, int> otsad = servad[jrjm_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);
        }
        jrjm_serv++;
    }

    return vastus;
}

```

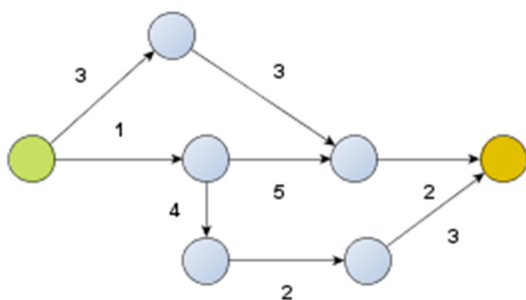
10.6 MAKSIMAALNE VOOG JA MINIMAALNE LÕIGE

Probleem sellest, kui suur on mingi võrgustiku (milleks võib olla veevõrk, 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.



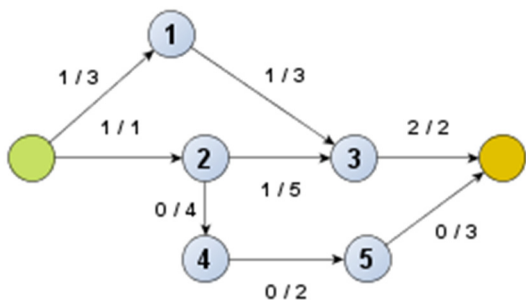
Transpordivõrk. Roheline tipp on lähe ja kollane tipp suue. Kaartel on läbilaskevõime.

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.

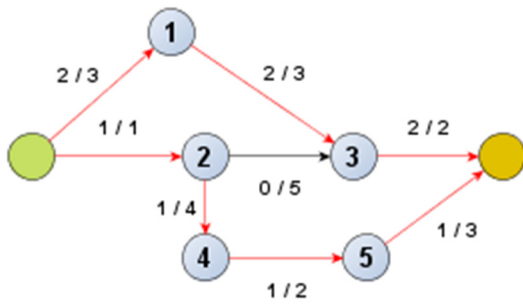


Voogudega võrk. Esimene arv kaare juures on voog ja teine läbilaskevõime.

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

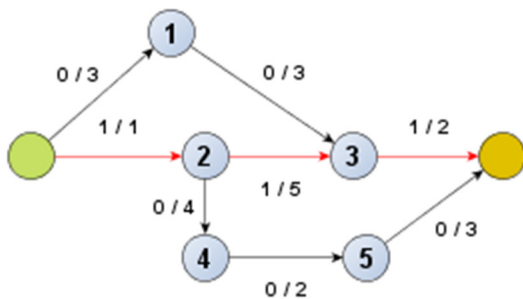
Võrkude puhul on enamasti kõige huvitavamaks küsimuseks, kui palju maksimaalselt saab midagi võrku läbides jõuda lähtest suudmesse ehk leida võrgu **maksimaalne voog**.



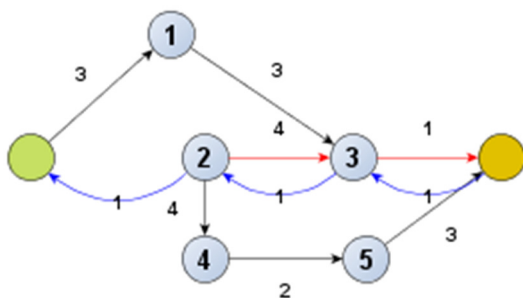
Võrgu maksimaalne voog. Maksimaalse voo väärtus on 3 – nii palju jõuab lähtest suudmesse.

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 esialgses 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:

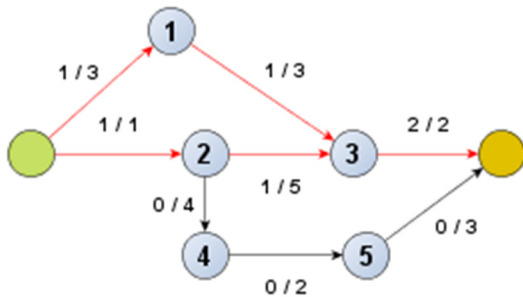


Eelmisele joonisele vastav jääkvõrk. Sinisega on tähistatud tagasikaared.

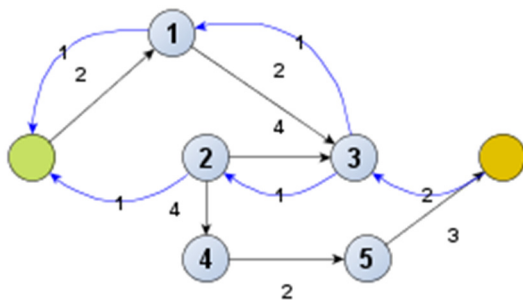
Kui vooga võrgu jääkvõrgus leidub selline tee, mis algab lähtest ja lõpeb suudmes ning mida mööda saab veel ressursse liigutada ehk iga kaare läbilaskevõime on positiivne, saame algse graafi voogu suurendada selle tee läbilaskevõime võrra. Sellist teed jääkvõrgus nimetatakse **täiendavaks teeks** (i.k. *augmenting path*).

Võrgul on leitud maksimaalne voog siis ja ainult siis, kui selle vooga võrgu jääkvõrgus ei leidu täiendavat teed.

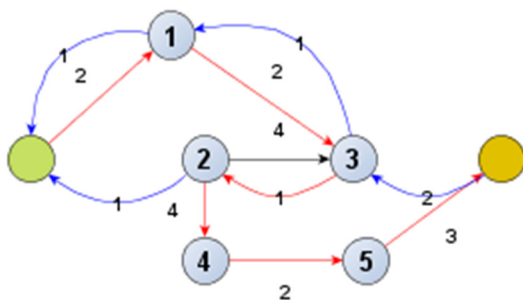
Ülaltoodud jääkvõrgus selline tee leidub, näiteks L-1-3-S. Kuna serva (3,S) läbilaskevõime on vaid 1, saame nii suurendada algset voogu ühe võrra. Selleks liidame algse graafi vastavate servade voogudele ühe juurde:



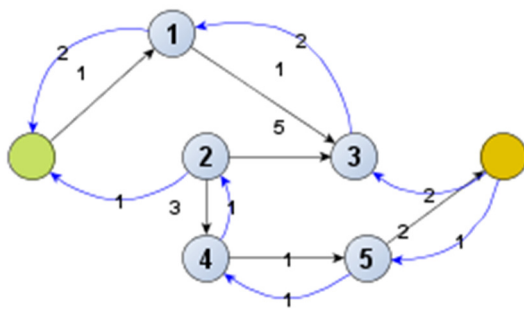
ja leiame tekkinud võrgu jääkvõrgu:



Nüüd tuleb jääkvõrgu tegelik olemus ja vajadus paremini ilmsiks, sest kui eelmisel sammul oli tee lähtest suudmesse olemas ka algset 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.

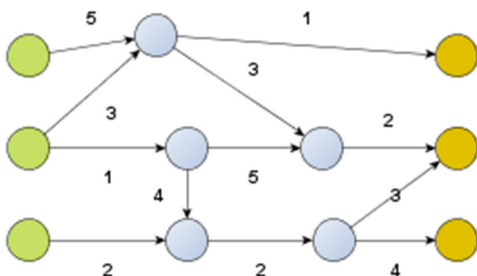
Kirjeldatud juhul ongi Ford-Fulkersoni algoritm. Mõnikord nimetatakse seda ka Ford-Fulkersoni meetodiks, kuna puudub täpne eeskiri, kuidas leida täiendavat teed jääkgraafis, viimasest aga võib sõltuda algoritmi keerukus.

Täisarvuliste läbilaskevõimete korral on selle keerukuseks $O(Ef)$, kus f on maksimaalse voo väärtus. Reaalarvuliste läbilaskevõimete korral ei pruugi antud algoritm tööd lõpetada.

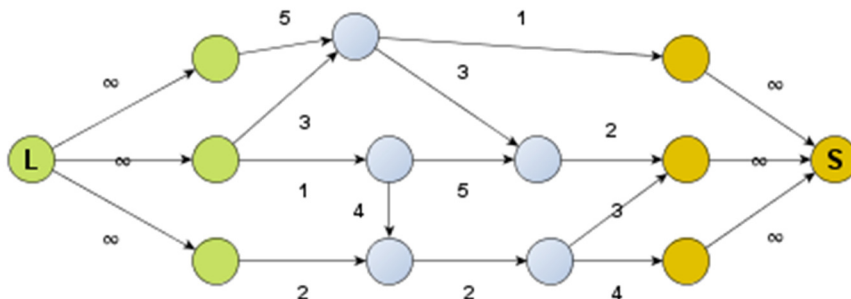
Kõige tuntum Ford-Fulkersoni meetodi implementatsioon on **Edmonds-Karpi algoritm**. See kasutab täiendava tee leidmiseks laiuti läbimist ning on keerukusega $O(VE^2)$.

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.



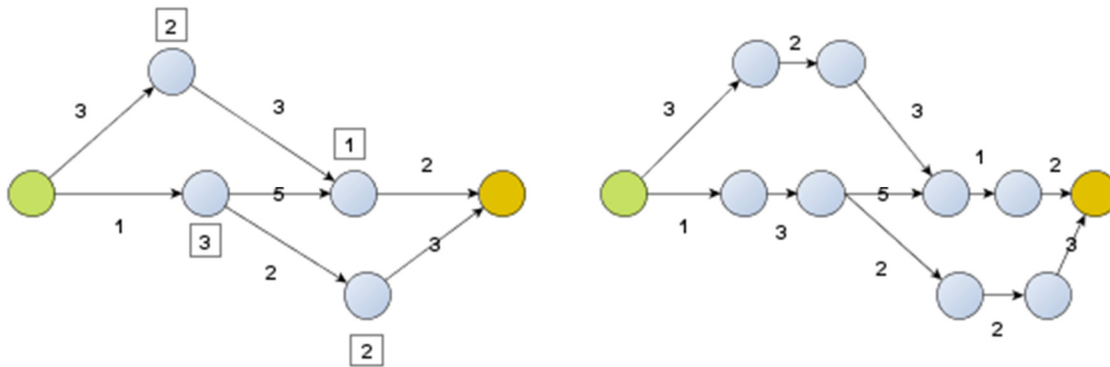
Sellisel juhul täiendatakse võrku kahe tipuga, nn superlähte ja –suudmega ∞ :



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

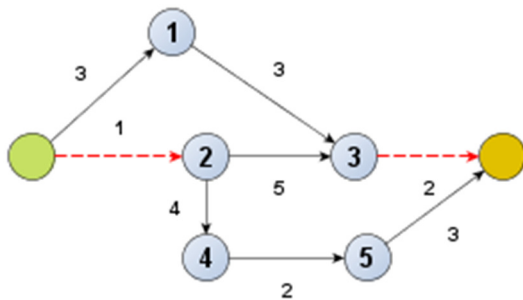
edasi. See lahendatakse nii, et tipp dubleeritakse ja sinna vahele lisatakse ekstra kaal tipu läbilaskevõimega:



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



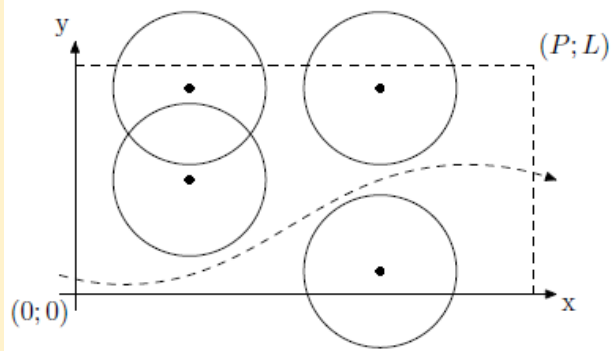
Minimaalseks lõikeks on kaared $(L, 2)$ ja $(3, S)$ ning lõike väärtuseks $1+2=3$.

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 ($1 \leq P \leq 50\,000$), L ($1 \leq L \leq 50\,000$) ja N ($1 \leq N \leq 250$). Faili järgmisel N real on igaühel ühe valvuri täisarvulised koordinaadid X_i ja Y_i ($0 \leq X_i \leq P$, $0 \leq Y_i \leq L$). Kanjoni edelanurga koordinaadid on $(0; 0)$ ja kirdenurga koordinaadid $(P; L)$. Vangid võivad kanjonisse siseneda mistahes punktis $(0; Y_s)$, kus $0 \leq Y_s \leq L$, ja väljuda mistahes punktis $(P; Y_v)$, kus $0 \leq Y_v \leq L$.

Seejuures ei pea Y_s ja Y_v olema täisarvud.

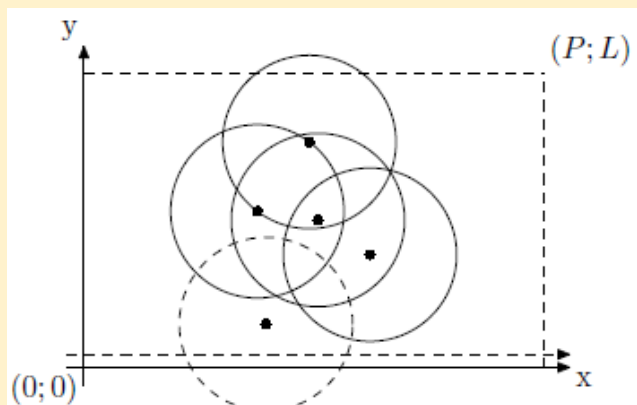
Väljundi ainsale reale väljastada mittenegatiivne täisarv, mis näitab vähimat võimalikku kõrvaldatavate valvurite arvu.

NÄIDE:

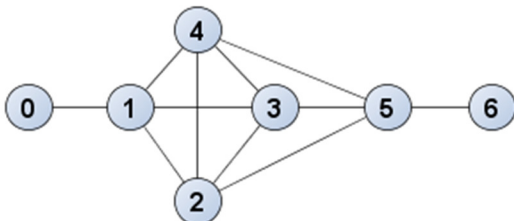
```
530 340 5
210 50
330 130
270 170
200 180
260 260
```

Vastus:

1

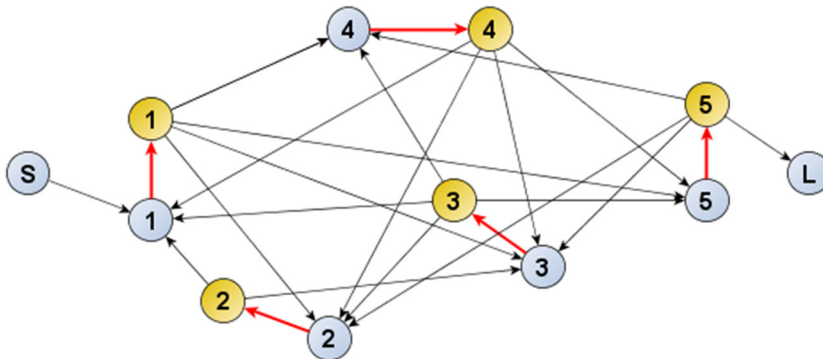


Esimeseks ülesandeks on jällegi koostada graaf. Võtame kõigepealt graafi tippudeks valvurid ja lisaks kaks tippu - kanjoni servad. Serv on kahe tipu vahel siis, kui nende valvealad puutuvad või kattuvad.



On küllaltki ilmne, et kui leidub tee tipust 0 tippu 6 ehk kanjoni seinte vahel, on põgenemine võimalik. 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 näiteandmete 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 leiame 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) { // jõudsimse 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};
    }

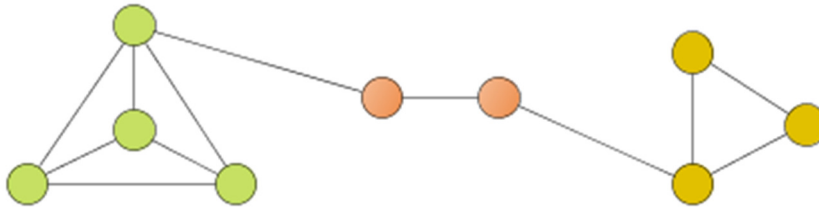
    for (int i = 0; i < valvureid; i++) {
        if (y[i] <= 100) { // kui on alumise serva lähedal
            kaarte_maht[0][i * 2 + 1]++; // lisame serva
            naabrid[0].push_back(i * 2 + 1);
            naabrid[i * 2 + 1].push_back(0); // tagasiserv
            naabrite_arv[0]++;
            naabrite_arv[i * 2 + 1]++;
        }
        if (y[i] + 100 >= 1) { // 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: roheline tipuhulgaga klikk, kollase tipuhulgaga klikk ja kolm kahetipulist klikki, servaga rohelse 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 – roheline 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. Piduga aga pole päris see, kui kellelgi on mõned sõbrad peolt puudu.

Leia, mitu erineva kosseisu 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 $m_i - i$. n -nda õpilase väidetav sõprade arv. Selle järel on m_i täisarvu $0 \dots n-1$ – tema sõprade numbrid. On teada, et õpilane ei nimeta kunagi ennast oma sõbraks.

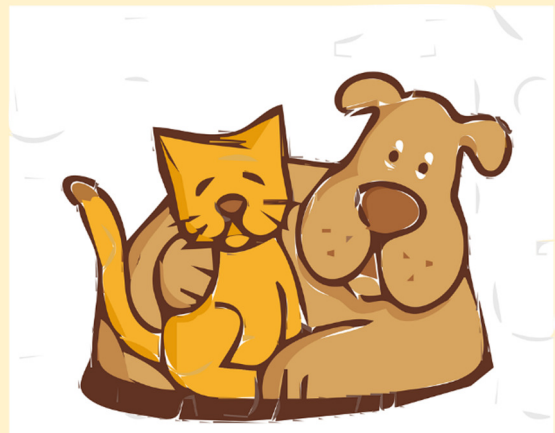
Väljundi ainsale reale väljasta üks täisarv – ülesande tingimustele vastavate peokosseisude arv.

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 klikid_suurus = klikid[i].size();
        for (int j = 0; j < klikid_suurus; j++) {
            int v = klikid[i][j];
            if (graaf[v].size() - klikid_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 ($2 \leq n \leq 100$): 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 ($1 \leq n \leq 10000$, $1 \leq m \leq 100000$, $0 < a \leq 10000$) – 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 ($1 \leq x, y \leq n$, $0 < c \leq 10000$) – 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 ($0 < p \leq 500$) – linnade koguarv. Järgmisel p real on kaks tühikuga eraldatud täisarvu: x ja y ($0 \leq x, y \leq 10000$) 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 ($1 \leq n \leq 50$) linna, mis on ühendatud omavahel m ($1 \leq m \leq 1225$) teega. Riigil läheb hästi ning seega otsustatakse kahe linna, millel puudub otseühendus, vahele luua otseühendus. Selle jaoks võetakse arvesse parima kasumi valemit: $\text{kasum}(u, v) = \text{Sum}(\text{pc}(i, j) - \text{cc}(i, j))$, kus iga linnapaari i, j jaoks on $\text{pc}(i, j)$ lühima tee pikkus linnast i linna j , kui tee u ja v vahel puudub, ning $\text{cc}(i, j)$ 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 ($-1000 \leq x, y \leq 1000$), linna koordinaadid. Järgmisel m real on kaks tühikuga eraldatud täisarvu: a ja b ($1 \leq a, b \leq n$), 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 ($1 \leq n \leq 120$), organisatsioonide arv. Järgmisel n real on üks sõne – organisatsiooni nimetus. Seejärel on eraldi real täisarv m ($1 \leq m \leq n$), 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 ($2 \leq n \leq 10$) riiki ning m ($5 \leq m \leq 25$) 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 ($0 \leq k_i \leq 50$), 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 mõõtmatega $N * M$ ($1 \leq N, M \leq 30$) 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 ($1 \leq P \leq 10$) 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 üherealine 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 ($1 \leq n \leq 100$). Järgmisel n real on neli tühikutega eraldatud täisarvu: x_1, y_1, x_2, y_2 - tänavate 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 Domino

Sul on n ($1 \leq n \leq 1000$) doomino kloti, mida sa tahad järjest ringi panna, nii et kahel järjestikusel klotil on ühendumiskohal sama number ning viimane klot 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 ($1 \leq a, b \leq 50$), doomino klotile 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 ($1 \leq d, v \leq 100$). Pärast saadet hääletavad n ($0 \leq n \leq 500$) 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 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

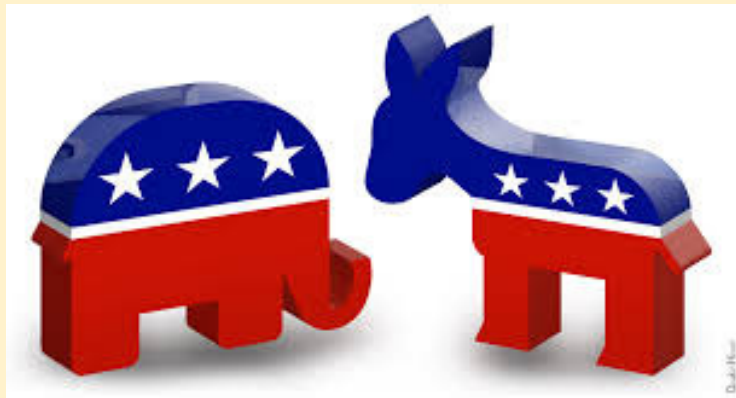
D1 V1

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	Artikulationipunktide 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 mälupeale 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 abstraksioonitaseme 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

```
<?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>
```

kiirem. See tähendab, et kaasaegne arvuti kulutab enamiku oma protsessoriajast just nimelt tekstitüüpi andmetega tegeledes.

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 „proosatekstit“, 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.

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**.

$$\begin{aligned} \frac{\frac{1}{3a} - \frac{1}{3b}}{\frac{a}{b} - \frac{b}{a}} &= \frac{\left(\frac{b-a}{3ab}\right)}{\left(\frac{a^2-b^2}{ab}\right)} \\ &= \frac{b-a}{3ab} \cdot \frac{ab}{a^2-b^2} = \frac{(b-a)}{3ab} \cdot \frac{ab}{(a+b)(a-b)} \\ &= \frac{(b-a)}{3ab} \cdot \frac{ab}{(a+b)(a-b)} = -\frac{1}{3(a+b)} \end{aligned}$$

Vaatame järgmist klassikalist ülesannet:

11.1.1 Ülesanne: arvutustehe

On antud aritmeetiline avaldis, mis sisaldab positiivseid täisarve, liitmis-, lahutamise, korrutamise- ja jagamismärke (+, -, *, :) 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 nõ 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 tekstitötlusalgoritmide 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 sufiks 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 prefiksiks on p, pe, pes, pesu, pesuk, pesuka, pesukar ja pesukaru. Nii nagu sufiksiks, 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 $\frac{1}{200 \cdot 200}$ 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

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
0	1	2	3	4
0	1	2	3	4

012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
kook kodu

Kuna siingi kokkulangevust pole, siis edasi on veel üksaaval minekut kuni $i=21$, millele järgnevad taas kokkulangevused, kuni $i=28$ ja $j=7$:

1	2	3	4	5
0	1	2	3	4
0	1	2	3	4

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 $s[29]\neq a[4]$:

1	2	3	4	5
0	1	2	3	4
0	1	2	3	4

012345678901234567890123456789012345678901234567890
pannkook kohupiimaga kook koorega kook kodujuustuga
kook kodu

Kuna ka siin kokkulangevust pole, liigume alates $i=29$ ja $j=0$ üksaaval kuni $i=34$:

1	2	3	4	5
0	1	2	3	4
0	1	2	3	4

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 ($3 \leq N \leq 200\,000$).

Faili teisel real on $2 \cdot N$ täisarvu lõigust -10^9 kuni 10^9 : esimese hulknurga tippude x - ja y -koordinaadid. Kolmandal real on samuti $2 \cdot N$ 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

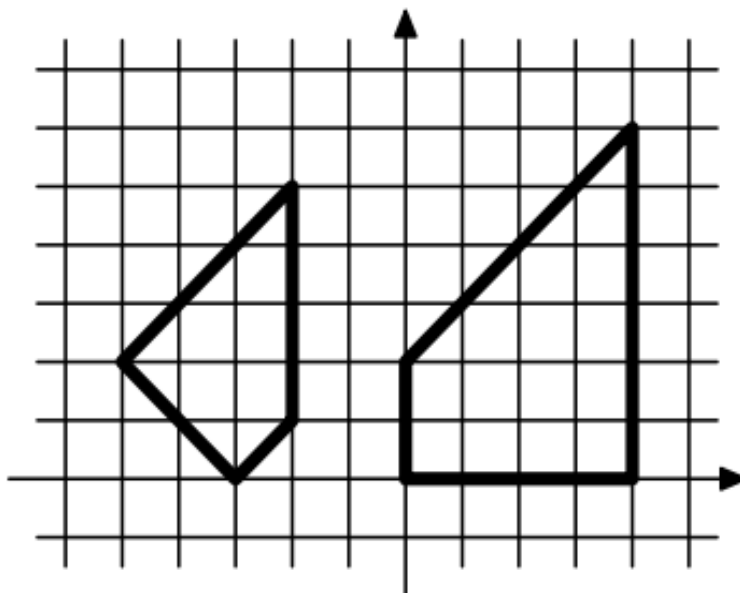
-2 5 -2 1 -3 0 -5 2

Vastus:

1.414213

3

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



Näitele vastav joonis

Sellel ülesandel oli kaks poolt: geomeetriline ja tekstitöötamise 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 koopias alamstring. Seega eeltoodud näites CABAABBA ja BBACABAA ekvivalentsus järeldub sellest, et CABAABBA on BBACABAABACABAA alamstring (ja ka BBACABAA on CABAABBACABAABBA 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;
}
```

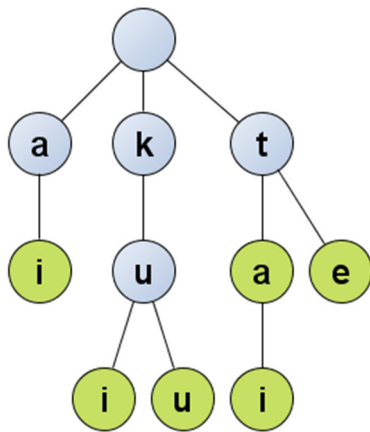
11.4 SUFIKSIPUUD

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

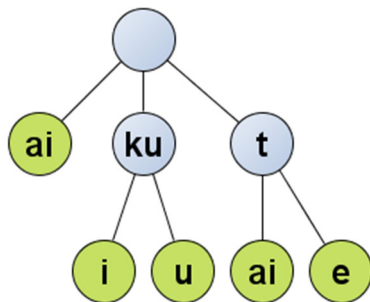
Prefiksipuu (i.k *trie*, *digital tree*, *prefix tree*) ingliskeelne nimetus *trie* tuleb sõnast *retrieve*, mis viitabki otsimisele. Tuletame meelde, et otsingupuu on puu, kus hoitakse võtme järgi väärtusi ning oluline ongi, et väärtus on võtme järgi kiiresti ja efektiivselt leitav, st võtmed on mingil moel sorteeritud. Kahendotsingupuus näiteks kehtis reegel, et tipu vasakus alampuu 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 paistabeli ees:

- Halvim juht info leidmiseks on $O(m)$, kus m on võtme pikkus
- Kokkupõrgete ehk kollisioonide puudumine
- Saab esitada võtmed kiiresti tähestikulises järjekorras

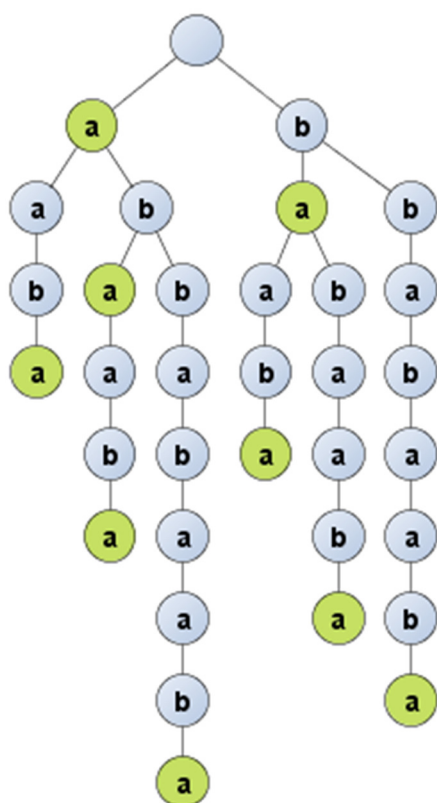
Prefiksipuid kasutatakse näiteks sõnade äraarvamisel (*autocomplete*) ja spellerites.

11.4.2 Sufiksipuu

Nimetame stringi S i-ndaks sufiksiks sellist stringi s sufiksit, mis algab positsioonilt i . Näiteks stringi $S = \text{'abbabaaba'}$ sufiksids on:

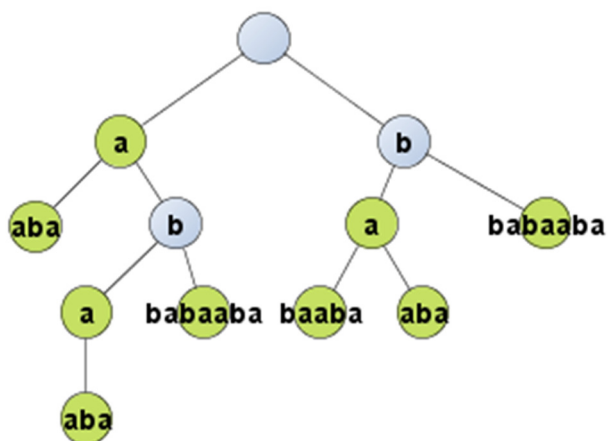
i	1	2	3	4	5	6	7	8	9
i-s sufiks	abbabaaba	bbabaaba	babaaba	abaaba	baaba	aaba	aba	ba	a

Sufiksipuuk (i.k *suffix trie*) nimetatakse stringi sufiksitest moodustatud prefiksipuud.



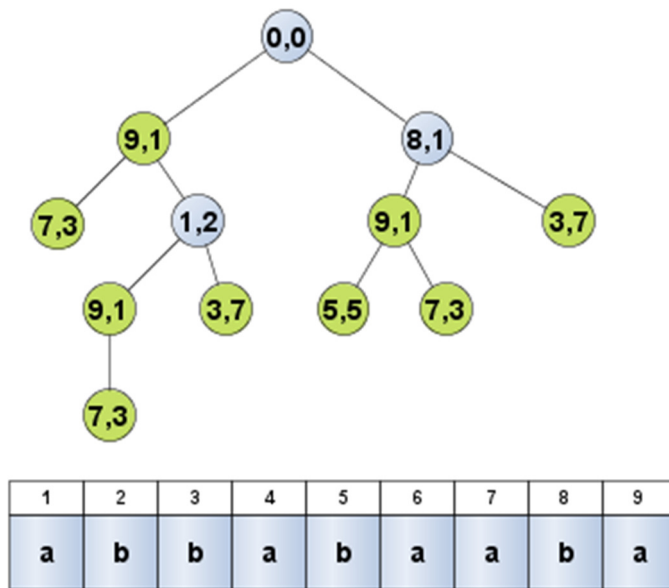
Stringi 'abbabaaba' sufiksipuu

Samuti nagu prefiksipuust sai teha kompaktsse prefiksipuu, võime esitada ka sufiksipuu kompaktsel kujul:



Stringi 'abbabaaba' kompaktna sufiksipuu

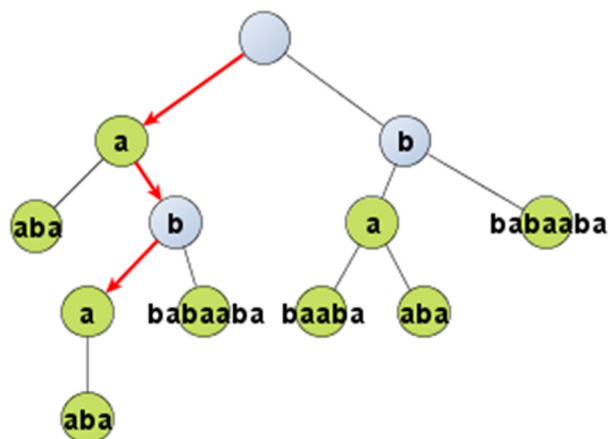
Veelgi enam: me võime puus hoida ainult algusindeksi ja pikkuse:



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 sufiksimuust.

Seda teeb ka järgnev kood:

```
using namespace std;
struct Tipp {
    char* voti;
    map<char, Tipp> alamad;
};

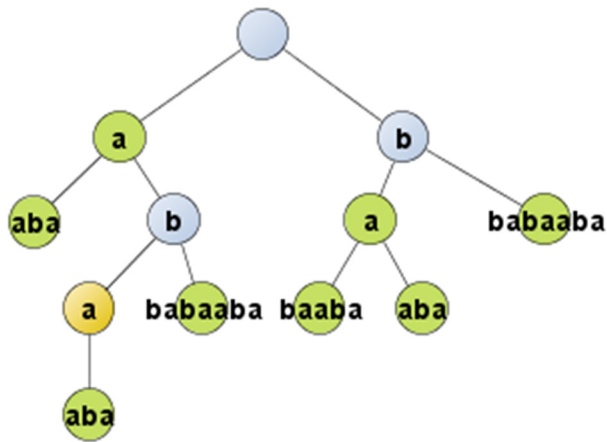
int leia_alamstring(Tipp *tipp, char* str, int indeks) {
    if (tipp == NULL) {
        return -1; // ei leidnud
    }
    int vas = -1;
    //kui tipp t ei ole juur
    if (tipp->voti != 0)
    {
        int vastus = kontrolli(str, indeks, tipp->voti);
        if (vastus != 0)
            return vastus;
    }
    //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!

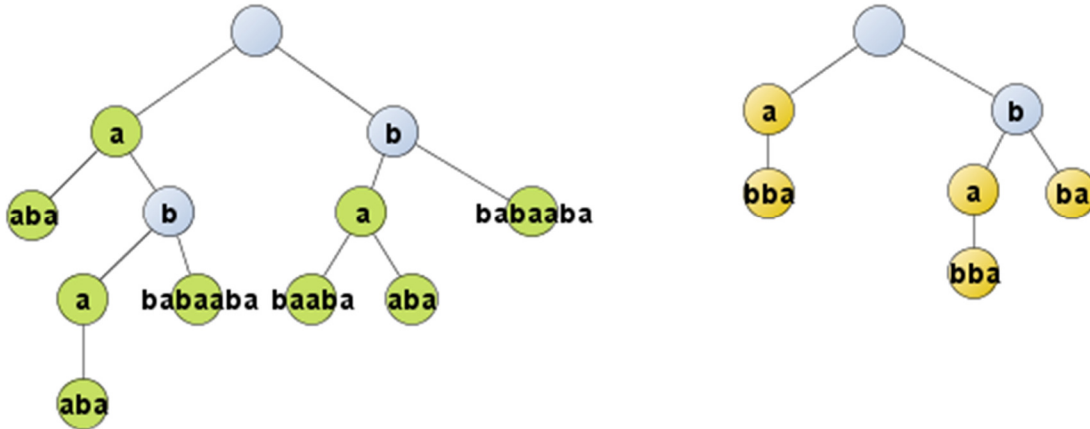
Pikim korduv alamstring on selline alamstring, mis esineb tekstis vähemalt kaks korda. Kui meil on olemas sufiksipuu, siis tuleb leida kõige sügavamal asuv hargnemine:



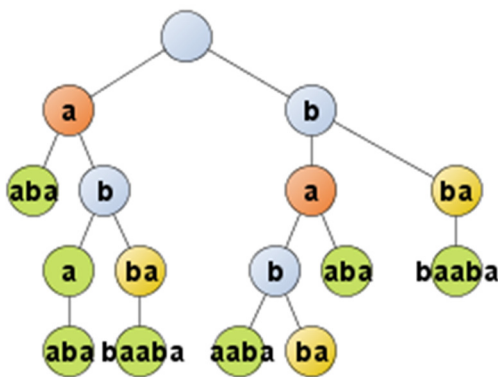
Selle lähenemise keerukus on $O(n)$.

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 **sufiksiloeendit** (i.k. *suffix array*).

Sufiksiloeendi 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 olema olemas 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 sufiksids (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 sufiksids ainult esimese tähe järgi, seejärel esimese kahe tähe järgi, siis esimese nelja tähe järgi jne. Kui meil on sorteeritud sufiksids 2^i esimese 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]; //algused 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 sufiksiste 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 räsidsid (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 räsids on üldjuhul kompaksemad 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 $S = s_1s_2s_3 \dots s_n$. 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 $S = abcab$.

$$\begin{aligned} f('abcab') &= a * 5^0 + b * 5^1 + c * 5^2 + a * 5^3 + b * 5^4 \\ &= 1 * 1 + 2 * 5 + 3 * 25 + 1 * 125 + 2 * 625 = 1461 \end{aligned}$$

Arvutuste käigus saame ka kõik prefiksids:

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

$$f(S) = \left(\sum_{i=0}^{n-1} s_i * p^i \right) \bmod m,$$

Arvutamine moodulitega:

1. $a \bmod m = a$, kui $0 \leq a < m$
2. $(a \bmod m) \bmod m = a \bmod m$
3. $(a + b) \bmod m = [(a \bmod m) + (b \bmod m)] \bmod m$
4. $(a - b) \bmod m = [(a \bmod m) - (b \bmod m) + m] \bmod m$
5. $(ab) \bmod m = [(a \bmod m)(b \bmod m)] \bmod m$
6. $a^b \bmod m = (a \bmod m)^b \bmod m$
7. $(a \div b) \bmod m = [(a \bmod m) \cdot (b^{-1} \bmod m)] \bmod m$, kus b^{-1} on arvu b modulaarne

mis omakorda on samaväärne

$$f(S) = \left(\sum_{i=0}^{n-1} (s_i * p^i) \bmod m \right) \bmod m.$$

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 $p * p' \equiv 1 \pmod{m}$. Kui $p = 5$, nagu meie näites, siis sobib $p' = 81$, sest

$$5 * 81 \pmod{101} = 405 \pmod{101} = 1.$$

Sellisel juhul saab prefiksitate tabeli 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:

$$\begin{aligned} f('ca') &= (f('abca') - f('ab')) * (p')^2 \pmod{101} = ((9 - 11) * 81^2) \pmod{101} \\ &= (9 - 11) \bmod 101 * 81^2 \bmod 101 = 99 * 97 \bmod 101 = 8. \end{aligned}$$

11.6.3 Jagamise vältimine

Kuna aga modulaarse vastandaru 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

$$f(S) = \left(\sum_{i=0}^{n-1} s_i * p^{n-1-i} \right) \bmod m.$$

Prefiksrite arvutamine läheb nüüd veidi põnevamaks, kuna iga prefiksi pikkus on ju erinev. Esimene prefiks a on ikka $a * 5^0 \bmod 101 = 1$. Järgmine, $f('ab') = a * 5^1 + b * 5^0 \bmod 101 = 7$ ja kolmas, $f('abc') = a * 5^2 + b * 5^1 + c * 5^0 \bmod 101 = 38$. Selline ükshaaval algusest peale arvutamine on muiudgi liiga ajamahukas, aga taas saab arvutada järgmise prefiksi räsi eelmist väljaarvutatud räsi kasutades. Pane tähele, et

$$\begin{aligned} f('abc') &= a * 5^2 + b * 5^1 + c * 5^0 \bmod 101 = 5(a * 5^1 + b * 5^0) + c * 5^0 \bmod 101 \\ &= 5(a * 5^1 + b * 5^0) \bmod 101 + c * 5^0 \bmod 101 \\ &= (5 * f('ab') + c) \bmod 101 \end{aligned}$$

Siin on taas uus prefiksrite räsiväärtuste tabel:

A	Ab	Abc	Abca	abcab
1	7	38	90	48

Arvutus tabelist:

$$f('ca') = f('abca') - f('ab') * p^k \bmod 101 = 90 - 7 * 5^2 = 16.$$

Veendume, et sama tulemuse saame ka otse arvutades:

$$f('ca') = c * 5^1 + a * 5^0 \bmod 101 = 16.$$

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 $s_i \neq a_i$, 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 räsidsid 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 prefiksrite 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-Karpi** 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 'abca'. $f('ca') = 16$. Arvutame kõigepealt tavapärasel moel stringi 'ab' räsi: $f(ab) = 7$. Kuna $16 \neq 7$, siis kokkulangevust kindlasti pole ja leiab järgmise räsi:

$$f('bc') = ((7 - 1 * 5 * \text{mod } 101) * 5 + 3) \text{mod } 101 = 13$$

Kuna ka $16 \neq 13$, saame asuda järgmist räsi arvutama:

$$f('ca') = ((13 - 2 * 5 * \text{mod } 101) * 5 + 1) \text{mod } 101 = 16$$

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*ras + 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 tähekombi-natsioon EIOTEE. 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:

EIOTEE

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 J 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 - fibonacciide arvude hulk. Järgmisel real on n fibonacci arvu. Kolmandal real on saadetud tekst.

Väljundisse kirjutada dešifreeritud tekst.

NÄIDE 1:

13
377 987 1597 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 TagaplaanII
LAEV LIIGUB ITTA



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;
}
```

A screenshot of C code implementing a tabbed interface. The code defines a structure for tabs, initializes an array of tabs, and sets up event listeners for clicking on tabs. It uses jQuery for DOM manipulation and jQuery UI for the transition effect. The code is partially visible, showing the initialization of the tabs array and the click event handler.

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:

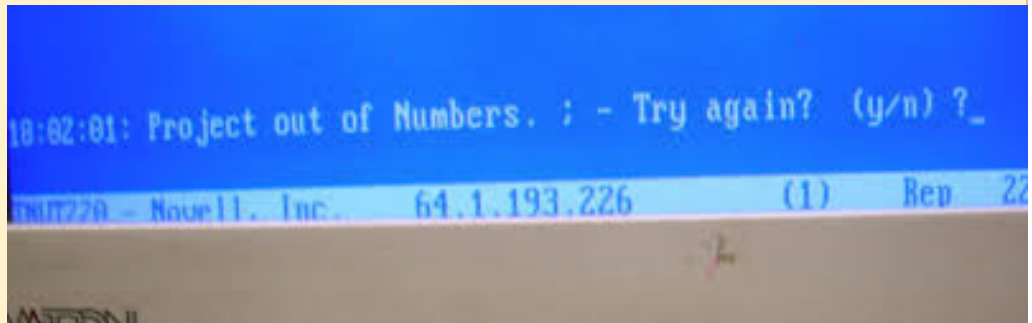
EI

NÄIDE 1:

5
113
12340
123440
12345
98346

Vastus:

JAH



11.7.7 Risti-rästi

Juhal on $n \times m$ 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
```

D	H	O	B	S	H	N	E	P	T	U	N	E	Y	VENUS
U	E	J	I	H	U	N	Y	S	T	H	A	O	R	EARTH
D	N	A	U	U	E	E	E	M	A	E	N	W	A	MARS
W	N	A	I	P	L	U	T	O	N	A	O	D	H	CERES
A	G	H	P	L	I	Z	O	O	E	R	U	S	U	ASTERIODS
R	D	E	I	H	C	T	M	N	W	T	N	S	H	JUPITER
F	H	Y	H	O	P	B	E	O	Q	H	I	U	E	SATURN
R	A	C	O	E	A	A	R	R	T	E	O	A	E	NEPTUNE
U	S	A	T	U	R	N	C	P	L	A	N	E	T	URANUS
R	T	A	E	H	F	T	U	E	U	L	E	E	E	PLUTO
I	E	U	C	U	F	A	R	O	V	C	E	I	O	DWARF
A	R	F	A	I	R	A	Y	A	O	E	I	R	H	PLANET
T	O	A	I	N	I	A	B	E	A	R	N	A	E	MOON
O	I	A	T	E	O	E	N	A	A	E	H	U	A	
E	D	I	D	D	O	E	D	U	T	S	E	T	S	
E	S	Z	E	E	H	O	P	H	S	L	U	M	S	

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:

sigä

Vastus:

3

NÄIDE 2:

ees

Vastus:

1

NÄIDE 3:

kuulilennuteetunneliluuk

Vastus:

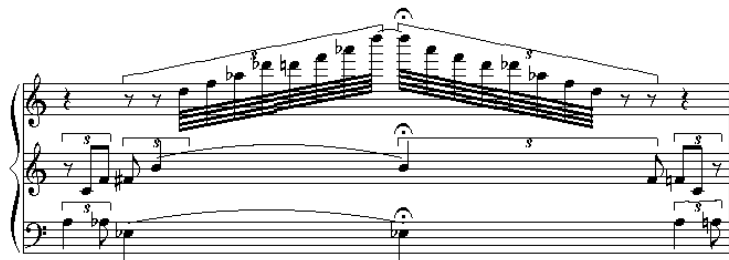
0

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

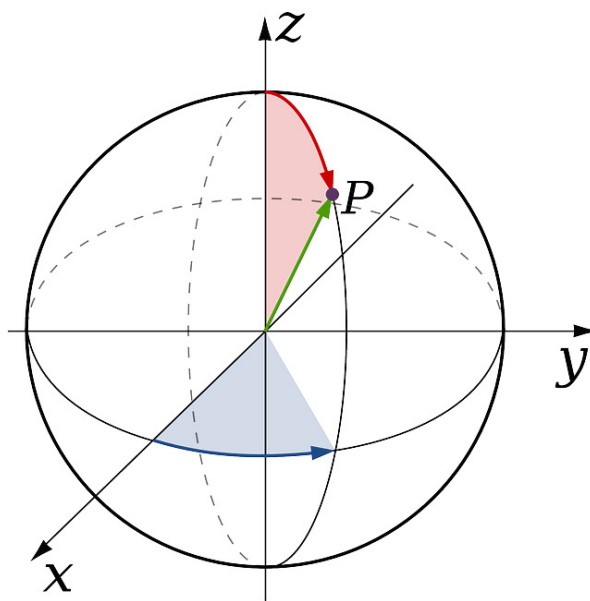
Kaasasolevas failis VP_lisad.zip, peatükk11 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

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 mõõtmised 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 $P(x, y)$.

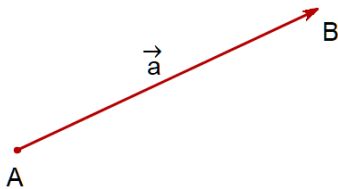
12.1.2 Lõik

Lõik (i.k. *line 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

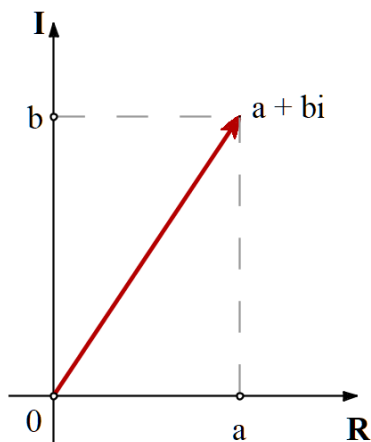
Vektor (i.k *vector*) on suunatud lõik, sellel on pikkus, siht ja suund. Kui kõik kolm parameetrit ühtivad, on vektorid võrdsed. Vektoreid, mis on samal sirgel, nimetatakse **kollineaarseteks**. Vektorit tähistame $\overrightarrow{AB}$. $\overrightarrow{AB} = -\overrightarrow{BA}$.

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 .

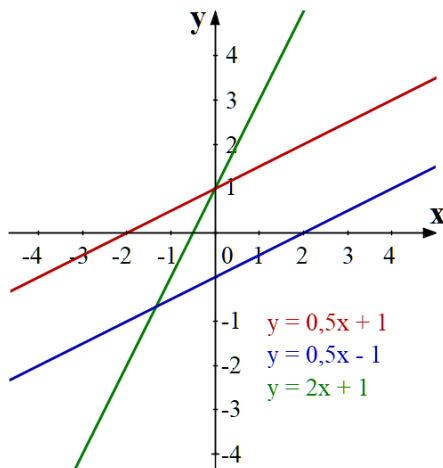
$\text{abs}(u)$ on vektori u pikkus. Seda saab kasutada ka kahe punkti vahelise kauguse leidmiseks: punktide u ja v omavaheline kaugus on $\text{abs}(v-u)$.

$\text{arg}(u)$ arvutab vektori u nurga x -telje suhtes radiaanides.

$\text{polar}(s, a)$ loob vektori pikkusega s ja nurgaga a . Vektorit saab keerata a kraadi, korrutades selle vektoriga, mille pikkus on 1 ja nurk a : $u * \text{polar}(1, a)$, keerab vektori 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

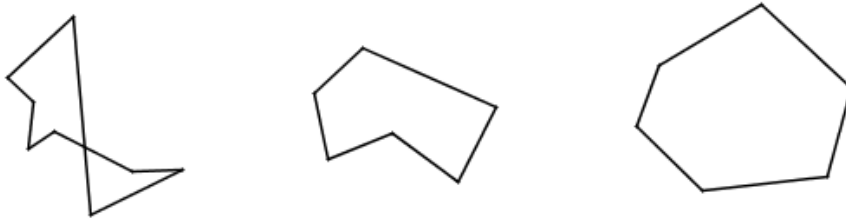
Murdjoon (i.k *polygonal chain*) on selline joon, mis koosneb järjestikustest, paarikaupa ühise otspunktiga lõikudest ehk lülidest $A_1A_2, A_2A_3, \dots, A_{n-1}A_n$. Murdjoone lülide otspunktid $A_1 \dots A_n$ on kõik erinevad. Kui $A_1 = A_n$, on tegemist kinnise murdjoonega. Murdjoont, mille ükski 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

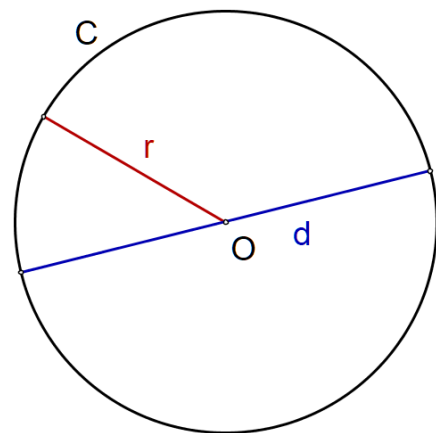
$$(x - a)^2 + (y - b)^2 = r^2,$$

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: $\pi = \arccos(-1)$.

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.

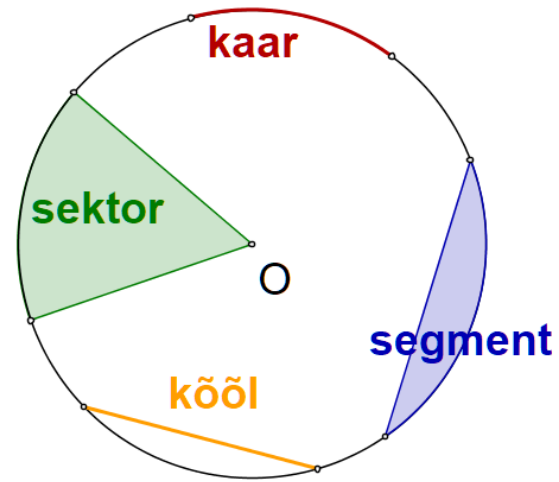
Sektoriks nimetatakse ala ringist, mis on piiratud kaarega ja kaare otspunktidest tõmmatud raadiustega. Sektori pindala saab leida sarnaselt kaare pikkusele:

$$S = \frac{\alpha}{360} \cdot A$$

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

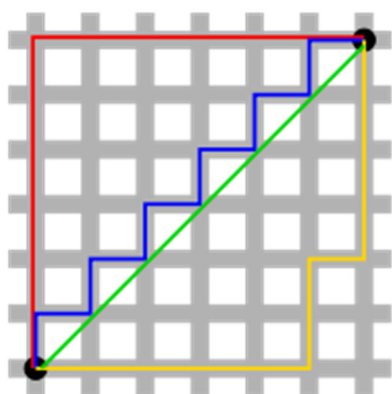
Kahe punkti vahelise kauguse tasandil saab leida Pythagorase teoreemi abil:

$$s = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

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 sirget asuvat punkti $P = (x_1, y_1)$ ja $Q = (x_2, y_2)$, siis neid läbiva sirge tõus on $(x_2 - x_1)/(y_2 - y_1)$. 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

$$y = \frac{a_2c_1 - a_1c_2}{(a_1b_2 - a_2b_1)}$$

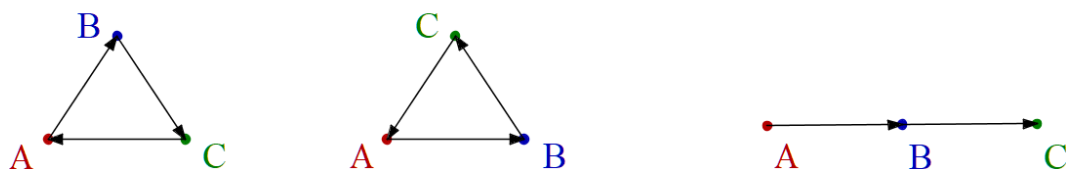
Funktsioon, mis leiab lõikepunktid:

```
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);
    }
}
```

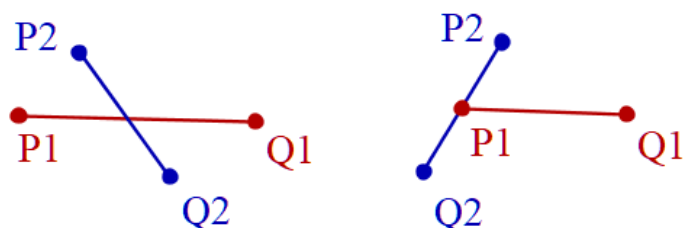
12.2.5 Lõikude lõikumine

Huvitavam ülesanne on aga määrata, kas kaks lõiku lõikuvad või kattuvad. Selleks vaatame kolme punkti tasandil ja nende vahelisi vektoreid. Eristame kolme võimalust:

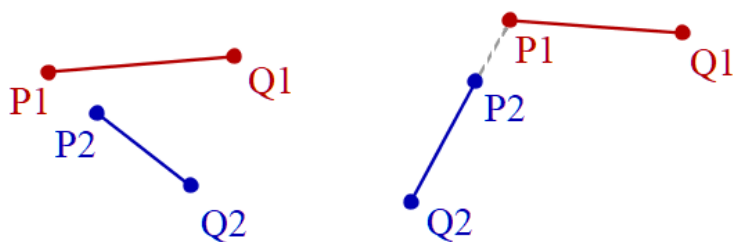


Esiteks, et punktid asuvad päripäeva, teiseks, et need asuvad vastupäeva ja kolmandaks, et need asuvad samal sirgel.

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 p_1, q_1, p_2 ja p_1, q_1, q_2 on erineva suunaga, aga p_2, q_2, p_1 ja p_2, q_2, q_1 on sama suunaga.

Erijuhaks 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
}

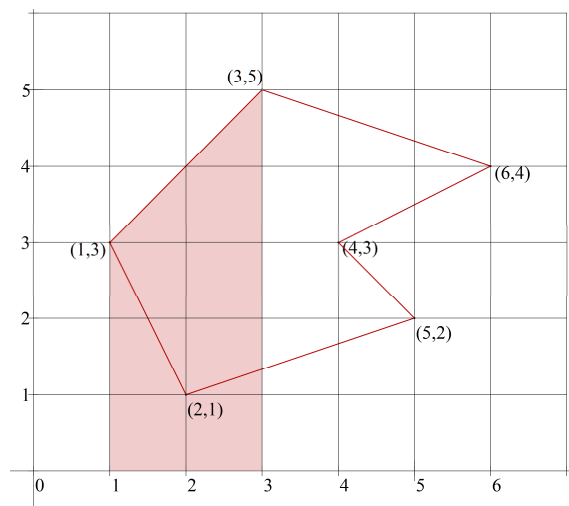
```

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.

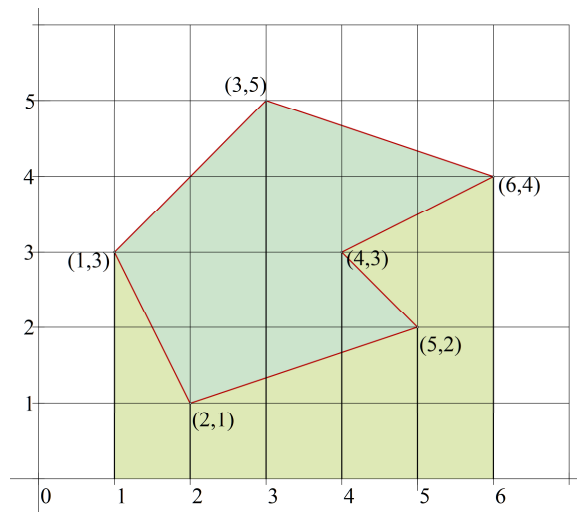
Pindalaga on veidi keerulisem, selle leidmiseks jagatakse hulknurk tavaliselt osadeks. Vaatame ühe külje ja x-telje vahele jäävat trapetsit:



Selle trapetsi pindala on ristküliku ja täisnurkse kolmnurga pindalade summa. Üheks kolmnurga kaatetik on külje projektsioon x teljele ehk serva otspunktide x koordinaatide vahe. Teiseks kaatetik 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:

$$\begin{aligned} & \frac{(3-1)(5+3)}{2} + \frac{(6-3)(4+5)}{2} + \frac{(4-6)(3+4)}{2} + \frac{(5-4)(2+3)}{2} + \frac{(2-5)(1+2)}{2} + \frac{(1-2)(3+1)}{2} = \\ & = \frac{2*8 + 3*9 - 2*7 + 5 - 3*3 - 4}{2} = \frac{21}{2} = 10,5 \text{ (ruutühikut)} \end{aligned}$$

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!

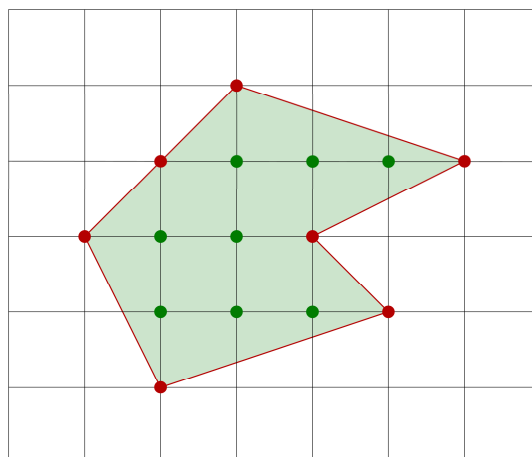
12.3.2 Picki teoreem

Kui hulknurga tippude koordinaadid on täisarvud, saab pindala leidmiseks kasutada ka **Picki teoreemi**. Nimelt avaldub hulknurga pindala valemiga

$$S = a + \frac{b}{2} - 1,$$

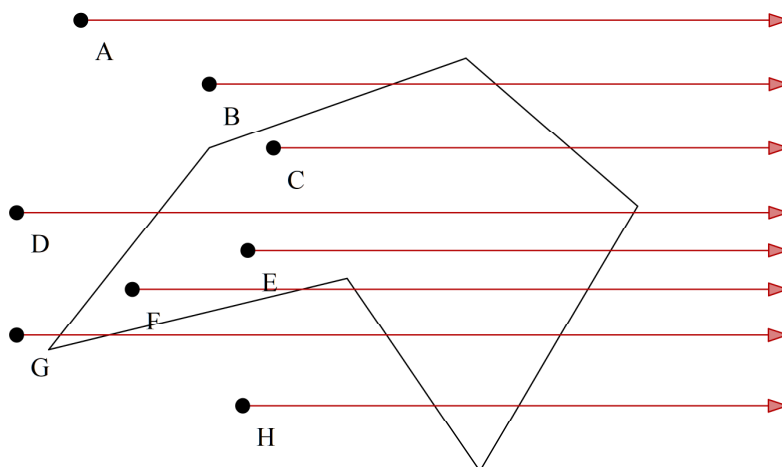
kus a on hulknurga sees asuvate täisarvuliste koordinaatidega punktide arv ja b on hulknurga servadel asuvate täisarvuliste koordinaatidega punktide arv.

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 punktist 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 $N \times N$ ruudustik ($1 \leq N \leq 10^9$). Ruudustikus on M ($1 \leq M \leq 100$) 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?

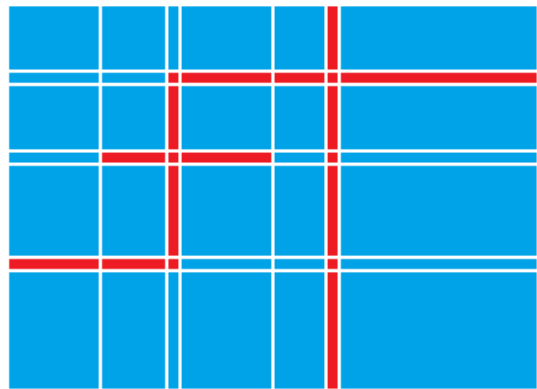
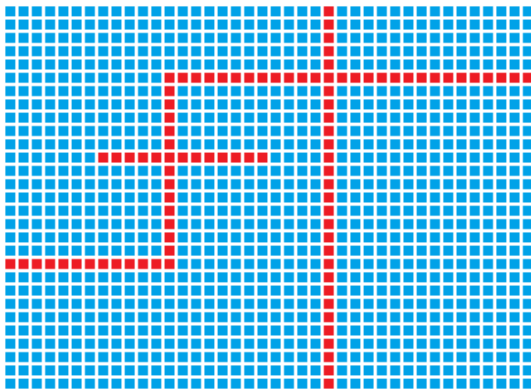
Paneme tähele, et kuigi ruudustik on suur, on takistusi seal hõredalt. Enamik read on täpselt samasugused, seega ei pea iga rida üksikult töötleva. Selle asemel saab kasutada meetodit, mida nimetatakse **koordinaatide pakkimiseks** (i.k. *coordinate compression*)

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 mees 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 ($1 \leq N \leq 1000$). Järgmisel N real on igaühel ühe tiku asend kujul $X \ Y \ S$, kus X ja Y on tiku väävlita otsa täisarvulised koordinaadid ($0 \leq X \leq 1000$, $0 \leq Y \leq 1000$) 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.

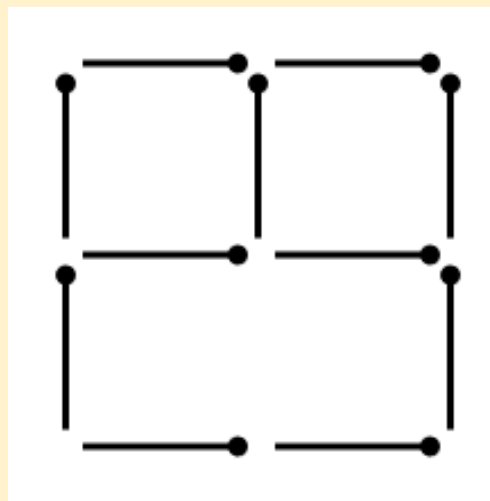
Väljundi ainsale reale väljastada moodustuvate ruutude arv.

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 vaatame horisontaalset tikk 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 tikk 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;

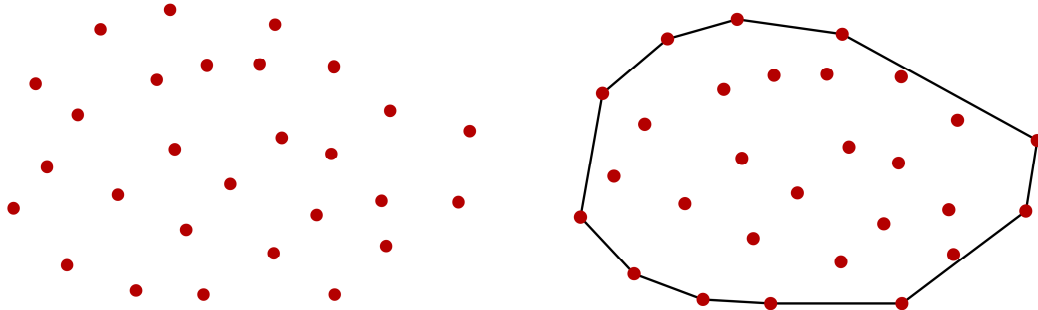
        if (on_hor_yhendus(tikk.second, tikk.second + kylje_pikkus - 1, tikk.first
+ kylje_pikkus) && // ülemine külg
            on_ver_yhendus(tikk.second + kylje_pikkus, tikk.first, tikk.first +
kylje_pikkus - 1)) // parem külg
            vastus++; //ruut leitud!
        kylje_pikkus++; //proovime suuremat
        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 $p - 1$. Indeksidsaab 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

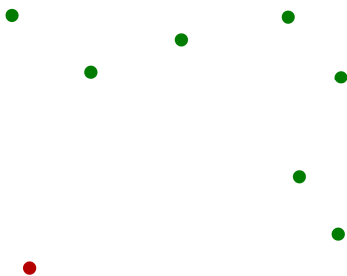
Kumer kate (i.k *convex hull*) on väikseim selline kumer hulknurk, mis sisaldab kõik etteantud punktid. Sisaldumine tähendab, et iga punkt asub kas hulknurga sees või mõnel selle küljel.



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 $O(n^3)$.

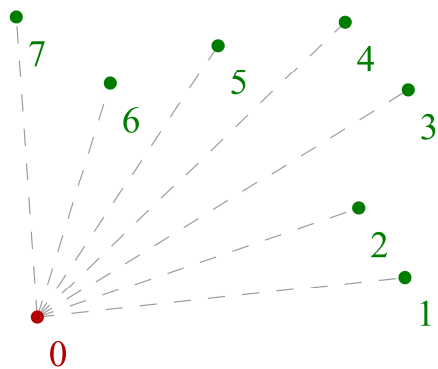
Kumera katte leidmiseks on olemas mitmeid paremaid algoritme. Üks neist on **Grahami seire**. Selle idee on järgmine:

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):

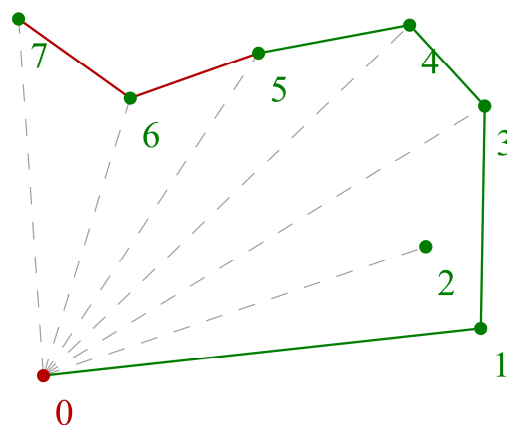
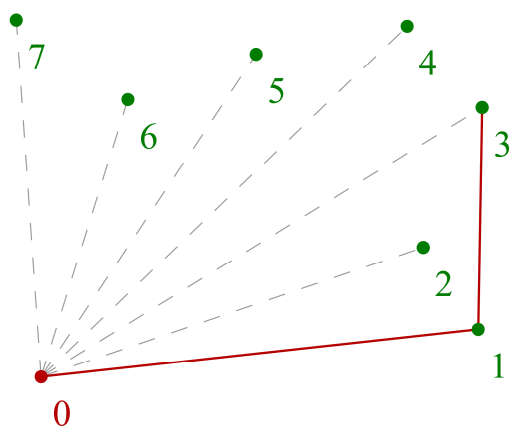
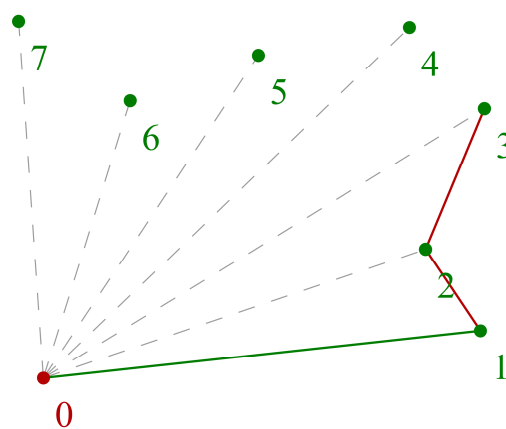
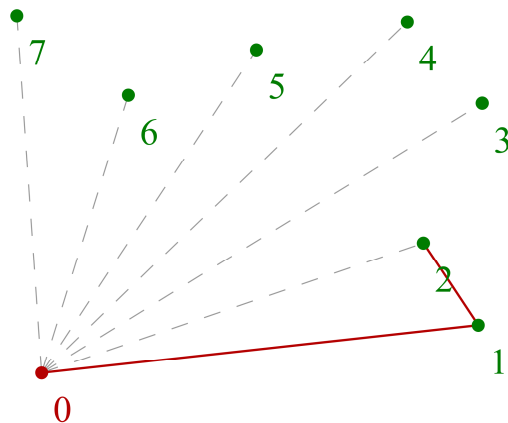


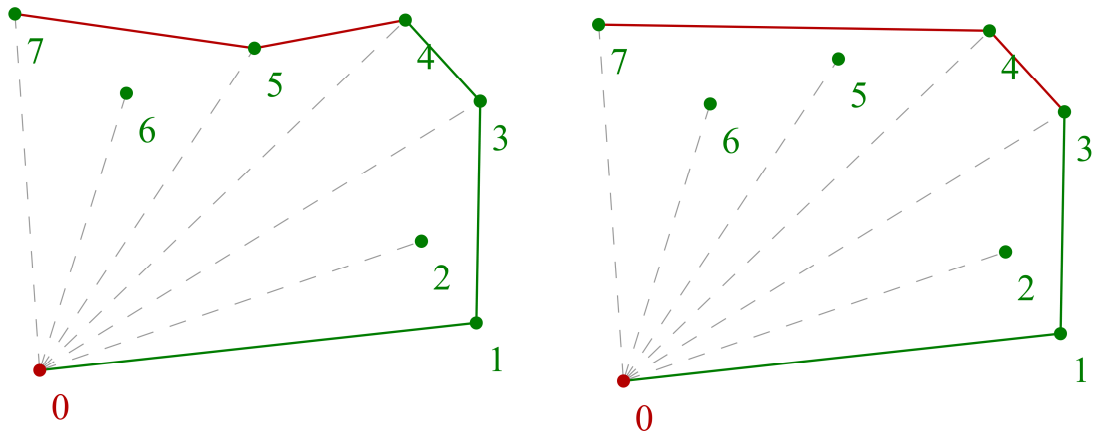
Sea see punkt punktiloendis esimeseks ja kuulub ise kindlasti kumerasse kattesesse.

2. Ülejäänud $n - 1$ punkti jaoks leia seirepunkti ja vaadeldavat punkti läbiva sirge tõusunurk. Sorteeri punktid leitud tõusunurga järgi vastupäeva.

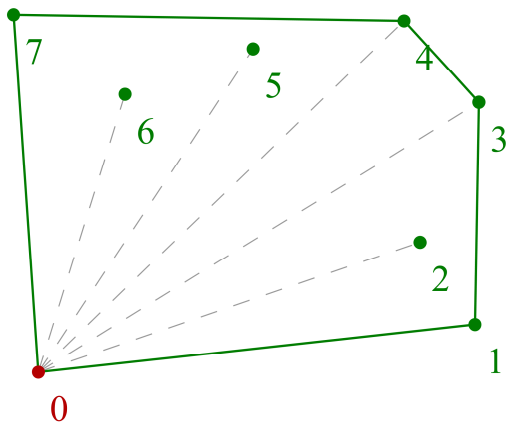


3. Kõigist sama tõusunurgaga punktidest jätta alles ainult see, mis on valitud punktist kõige kaugemal, sest vaid see saab kuuluda kumerasse kattesse.
4. Kui järele jääb vähem kui kolm punkti, siis kumerat katet ei eksisteeri (punktid asuvad samal sirgel).
5. Edasi tuleb iga punkti p_i jaoks otsustada, 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, jätta 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 kattedest. 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 kattedest. Aga ka pööre (4, 5, 7) on vales suunas, seega tuleb kattedest 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);

    // hoiaime 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:

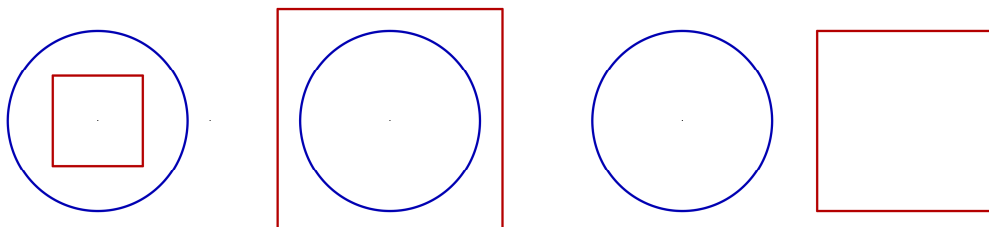
0 0 1

0 -2 0 2 4 -2 4 2

Vastus:

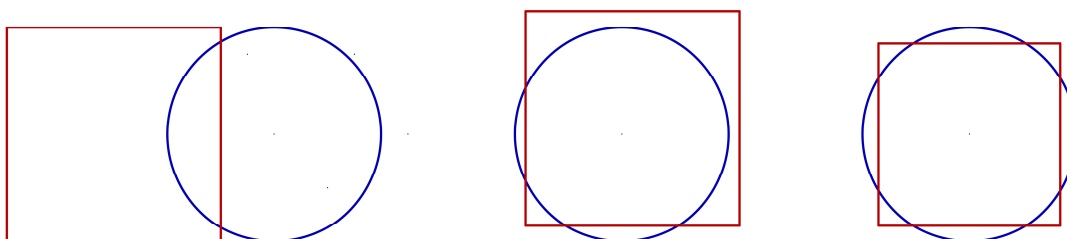
Algselt arvas ka ülesande väljamõtleja Indrek Jentson, et kui raske see ülesanne ikka olla saab, kuid erinevad testjuhud lähevad väga kiiresti käest ära. Uurime neid korraaks, tegu on ka kasuliku testimisalasena harjutusega.

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.

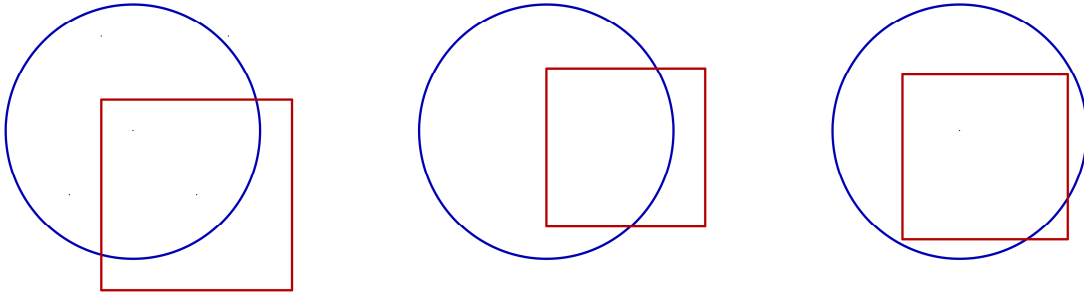
Lisaks on muidugi ka keerulisemaid võimalusi, näiteks asetused, kus ükski nurk ei ole ringi sees ja:



- 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 **geomeetiline lähenemine** – proovida iga erijuhtu 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;
}

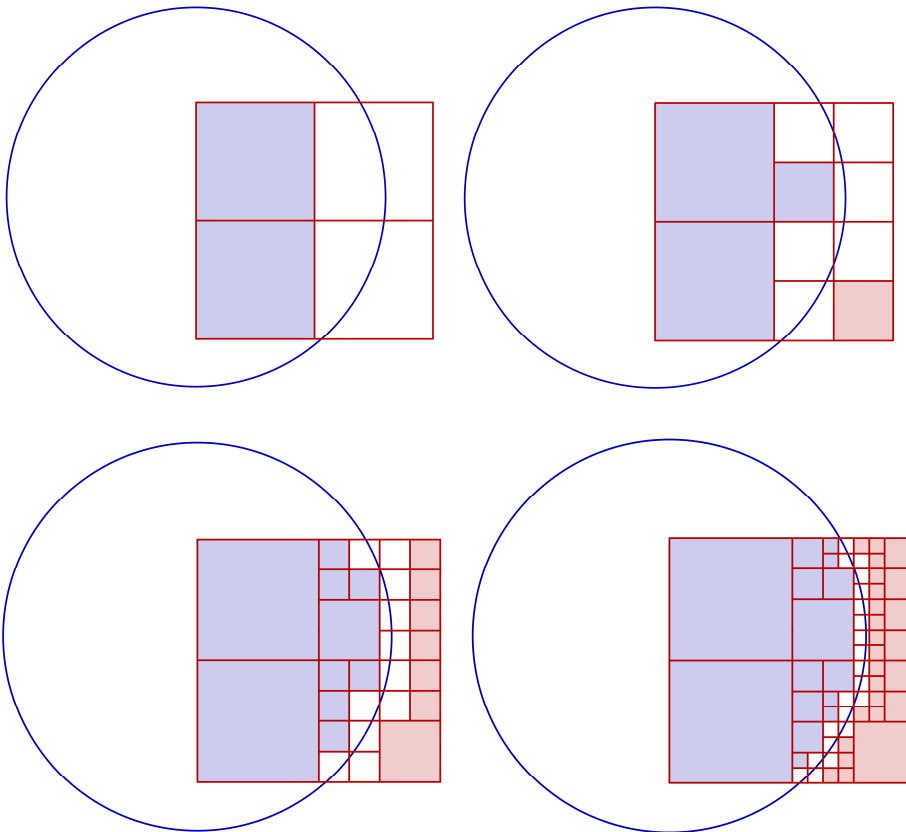
int tippe_ringis(pair<double, double> ruut[4]) {
    int summa = 0;
    for (int i = 0; i < 4; i++) {
        summa += tipp_ringis(ruut[i]);
    }
    return summa;
}
```

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);
}
```

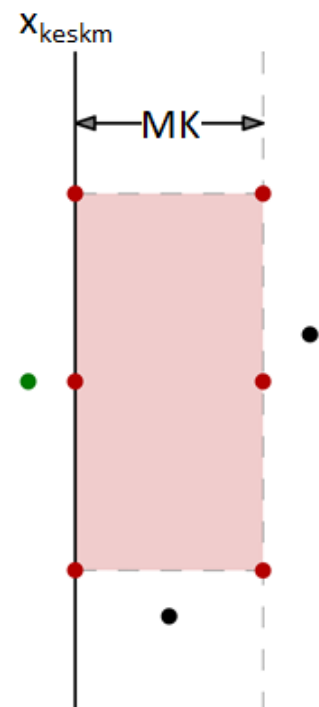
12.7 LÄHIM PUNKTIPAAR

Üks klassikaline arvutusgeomeetria ülesanne on veel punktihulgast omavahel lähima punktipaari leidmine. Naivne 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 riskülikus mõõtmetega $MK \times 2MK$. Lisaks ei saa see riskülik 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.

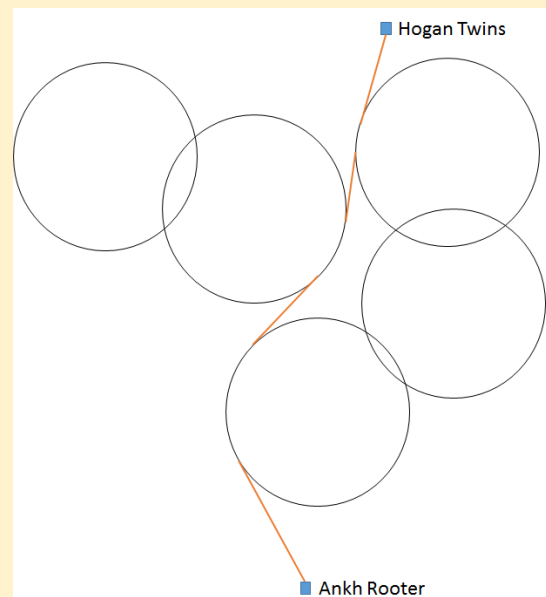
Kabob Maracal on mitmeid liitlasi Kettamaailmas ja ta on neile saatnud maagilise raketi vastase kilbi generaatorid. Kui maagiline rakett jõuab generaatorile lähemale kui R miili, kukub rakett ohutult alla. Gin Junk Om aga teab kilbigeneraatorite asukohti ning tahab, et sa arvutaks, kui mitu thaumi (võlujõu 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 ($1 \leq N \leq 1000$).

Järgneval N real on igaühel kaks täisarvu: ühe generaatori koordinaadid.

Väljundisse kirjuta üks arv: vajalik võlujõud. Kui raketti lennutada ei saa, väljastada -1.

NÄIDE:

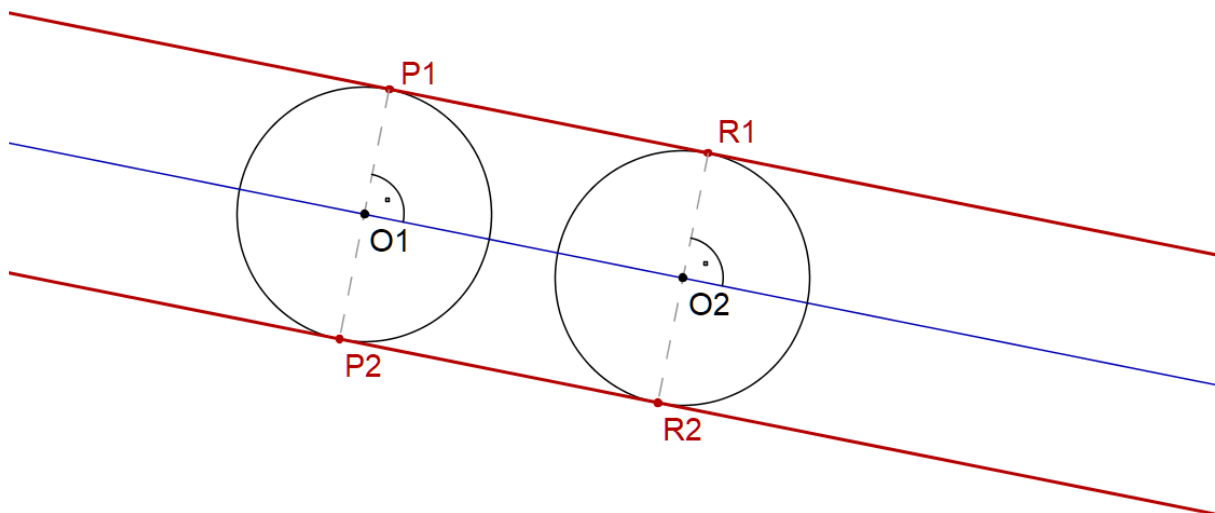


Selleks, et võimalikult otse lennata, peab rakett liikuma kas mööda kahe ringi ühist puutujat või mööda ringjoont. Algas ja lõppmarsruut on mööda puutujat, mis algab/lõpeb tornist.

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 löika 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 kaartide 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

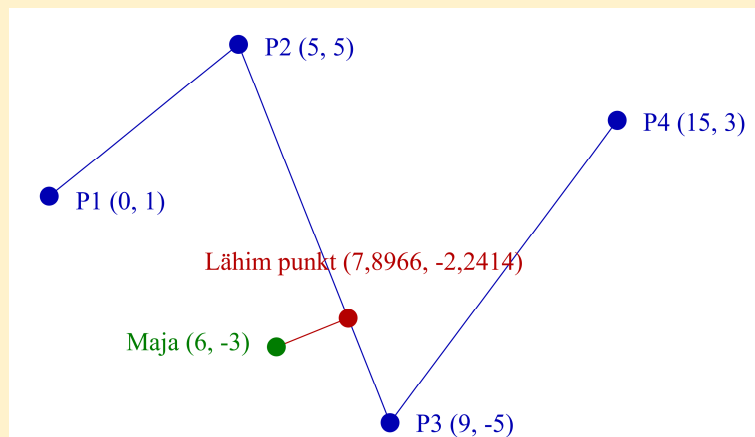
1

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 ($1 \leq N \leq 26$) kolmnurka ning M ($1 \leq M \leq 100$) 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 ($3 \leq N \leq 50$) 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 ($0 \leq S \leq 30000$). 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:

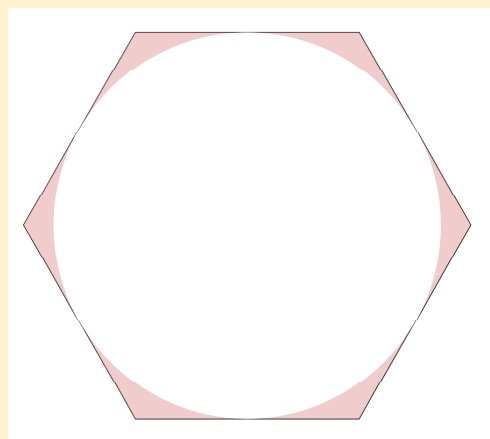
0.54352 0.24188

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 ($1 \leq N \leq 50000$), punktinelikute arv. Järgmisel $4N$ real on kaks tühikutega eraldatud täisarvu x ja y ($-10000 \leq x, y \leq 10000$), 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 ($1 \leq N, M \leq 30$), 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 ($1 \leq N \leq 100$), pargis olevate puude arv. Järgmisel N real on kaks arvu, puu koordinaadid.

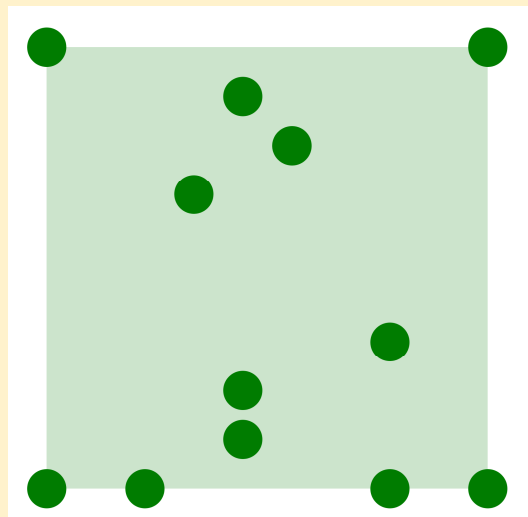
Väljundisse kirjutada üks arv, tekkiva pargi pindala.

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 Sүүrias 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 müüri piiratud. Leida see minimaalne ümbermõõt.

Sisendi esimesel real on täisarv N ($1 \leq N \leq 100$), 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 ($0 \leq J, T, P \leq 200$), *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 ($1 \leq N \leq 10$) alast, mis on nummerdatud arvudega 0 kuni $N-1$. Iga ala märklaua alal on kas ristkülik, ring või kolmnurk ning annab 2^i punkti, kus i on ala number. Mängija viskab M ($1 \leq M \leq 10$) 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 $W \times H$ ($1 \leq W, H \leq 1000$), mida läbib N ($0 \leq N \leq 100$) sirgjoonelist tulemüüri. Iga tulemüür ulatub täpselt servast serva. Viirus sattus internetis ühte punkti koordinaatidega x, y ($0 \leq x \leq W$; $0 \leq y \leq H$) 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
```