ESIMENE OSA - ALGAJATELE

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

1 PROGRAMMIDE SISEMAAILM

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

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

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

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

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

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

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

1.1 PROGRAMMI ELUTSÜKKEL

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

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

1.1.1 Lähtekood

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

Ülesannet lahendav lähtekood keeles C++ on järgmine:

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

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

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

1.1.2 Lähtekoodi transleerimine

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

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

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

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

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

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

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

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

1.1.3 Teegid

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

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

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

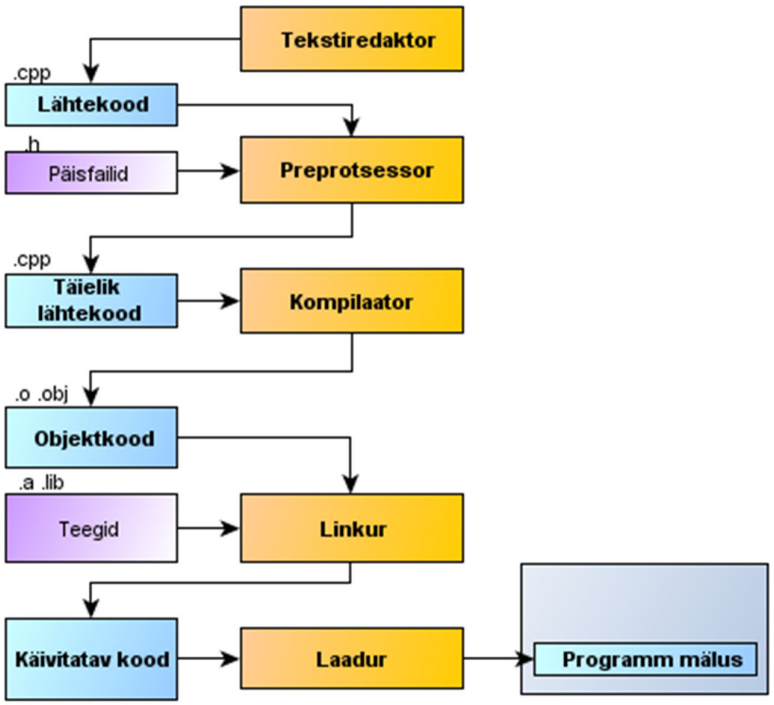
1.1.4 Linkimine

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

1.1.5 Laadimine ja täitmine

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

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



1.2 ANDMETE HOIDMINE JA TÖÖTLEMINE ARVUTIS

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

1.2.1 Protsessori tööpõhimõte

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

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

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

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

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

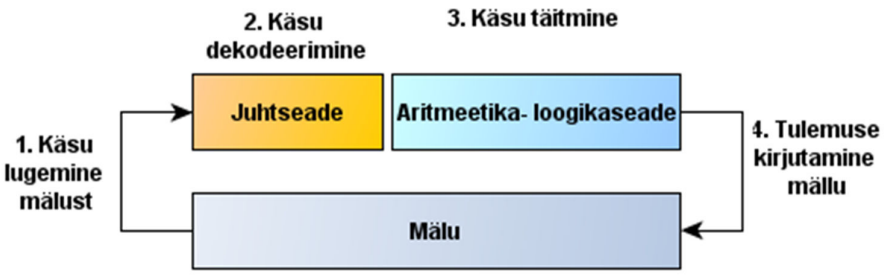
```
Käsukood  Aadress1 Aadress2 Väärtus
00100000001000100000000101010111
```

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

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

Lihtne protsessori töötsükkel on järgmine:

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



1.2.2 Protsessori jõudlus

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

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

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

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

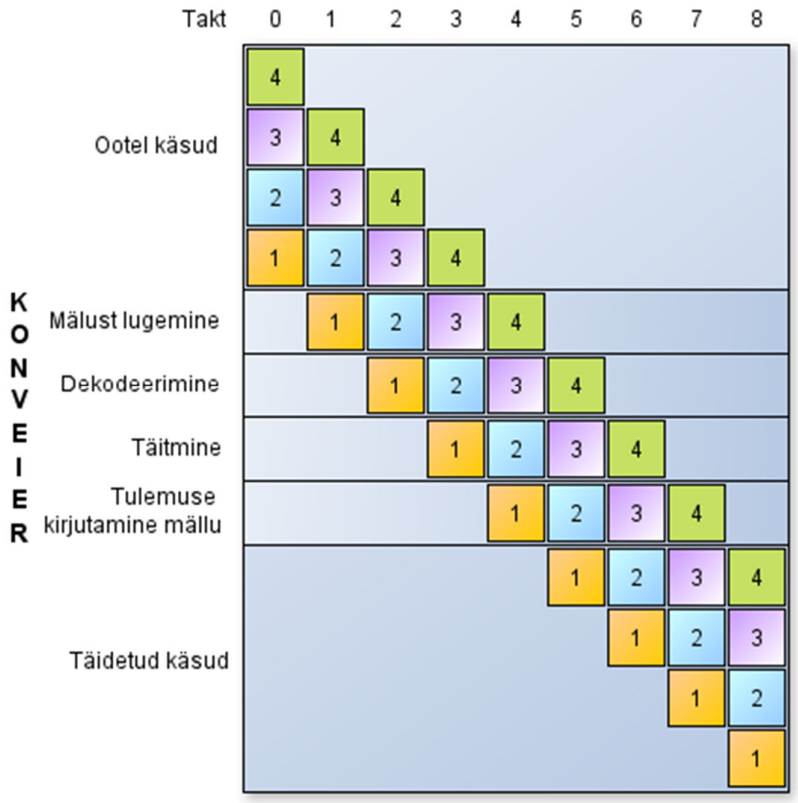
Rusikareeglina soovitatakse võistlusprogrammides arvestada umbes sada miljonit „tavapärast“ operatsiooni sekundi kohta, kus tavapärane on näiteks massiivis kahe väärtuse vahetamine, paarist tehtest koosnev aritmeetiline avaldis või muu suhteliselt lihtsa sisemise keerukusega operatsioon.

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

1.2.3 Käsukonveier

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

Protsessi illustreerib järgmine skeem:



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

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

1.2.4 Hargnemise ennustamine

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

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

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

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

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

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

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

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

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

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

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

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

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

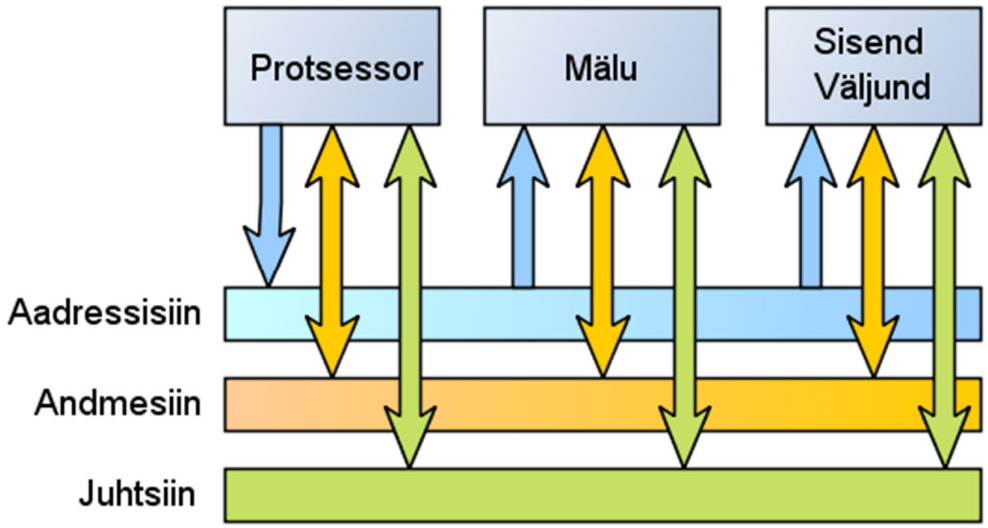
Sellepärast püüabki protsessor **hargnemist ennustada**, s.t mõistatada, kumba haru pidi edasi minnakse. Käsukonveierisse laaditakse vastava haru käsud ette ära. Kui ennustus oli õige, läheb protsess täie kiirusega edasi. Kui aga oli ekslik, siis visatakse konveierist järgmised käsud minema ja laaditakse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

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

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

1.2.5 Andmete liikumine

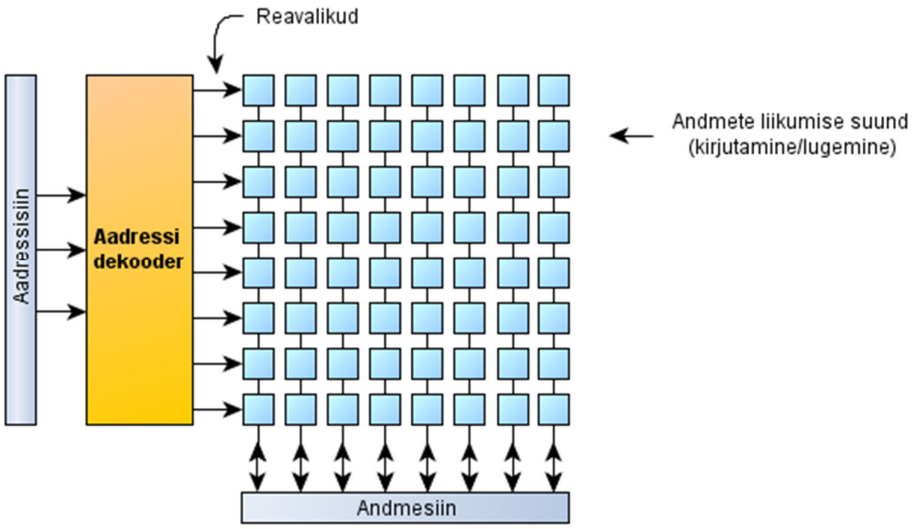
Andmete transportimiseks mälu, protsessori ja sisend-väljundseadmete vahel kasutatakse **siine** (*bus*): **andmesiinil** (*data* bus) liiguvad andmed, **aadressisiinil** (*address bus*) on info, kuhu andmed liiguvad, ja **juhtsiinil** (*control bus)* info, mis suunas ja mille vahel andmed parajasti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



1.2.6 Mälu

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

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



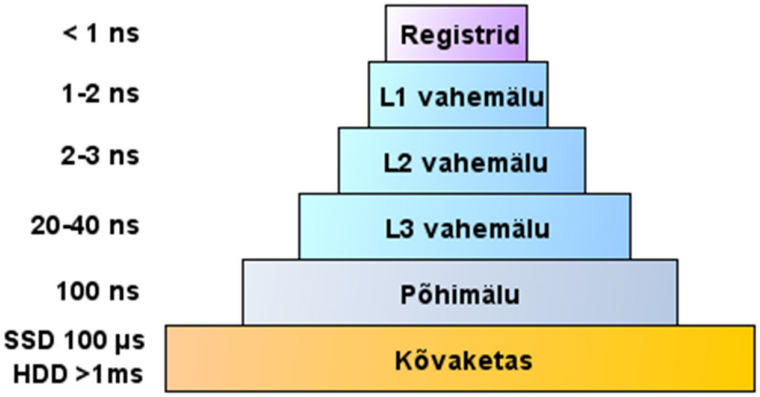
Mälu loeb aadressisiinilt aadressi, aadressi dekooder saadab signaali soovitud aadressliinile ning sellel liinil olevad bitid kirjutatakse andmesiinile või loetakse andmesiinilt vastavale aadressiliinile.

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

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

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

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



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

1.3 ANDMETÜÜBID

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

1.3.1 Täisarvud

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

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

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

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

Enamlevinud täisarvutüübid ja nende pikkused on toodud järgnevas tabelis:

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

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

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

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

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

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

Ületäitumine

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

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

1.3.2 Ujukomaarvud

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

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

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

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

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

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

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

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

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

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

Probleemid ujukomaarvudega

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

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

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

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

Teeme kontrolliks läbi järgmise näite Pythonis:

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

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

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

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

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

Pythonis on selleks ka funktsioon `math.isclose`:

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

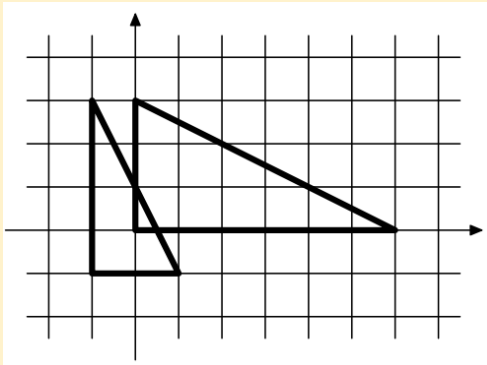
Ujukomaarvu konverteerimisel täisarvuks kasutatakse enamasti 0 suunas ümardamist, mis tähendab murdosa ärajätmist. Seetõttu võib täisarvuks konverteerimine anda ootamatuid tulemusi. Soovitatav on kasutada alati eelnevalt mõnd sobivat ümardamismeetodit ning vajadusel võtta taas abiks epsilon.

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

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

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

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



NÄIDE 1:

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

Vastus: `1.5` (vt joonist)

NÄIDE 2:

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

Vastus: `-1`

Esimene pähetulev lahendus on selline:

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

using namespace std;

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

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

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

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

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

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

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

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

Tulemusena saame järgmise koodi:

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

using namespace std;

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

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

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

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

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

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

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

1.3.3 Püsikomaarvud

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

1.3.4 Märgid

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

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

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

1.3.5 Tõeväärtused

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

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

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

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

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

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

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

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

Loogikatehted

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

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

Näiteks lauses

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

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

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

1.3.6 Viidad

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

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

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

1.3.7 Struktuursed tüübid ehk liittüübid

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

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

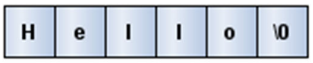
1.3.8 Kirjed

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

1.3.9 Stringid

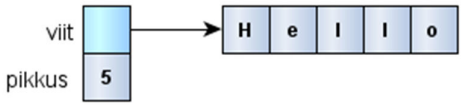
String ehk sõne on märkide jada ja kuna seda kasutatakse palju, on see sageli defineeritud eraldi andmetüübina. Stringide hoidmiseks mälus on kasutusel peamiselt kolm erinevat võimalust:

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



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

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



1.3.10 Massiivid

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

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

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

Mida väljastab selline kood?

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

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

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

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

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

1.4 OPERATSIOONID STRINGIDEGA

1.4.1 Märk kohal i

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

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

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

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

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

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

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

1.4.2 Stringide võrdlemine

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

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

Tähele tuleb aga panna, et tähestikuks on tegelikult kasutatav märgistik. Kuna enamasti on neis kõik suurtähed enne väiketähti, siis võrdlemisel on näiteks 'B' enne 'a'-d:

```
>>> "a" > "B"
True
>>> "a" > "b"
False
```

Kui on vaja sorteerida stringid tavapärasesse leksikograafilisse järjestusse, siis harilikult teisendatakse stringid kõik nii, et need koosnevad ainult väike- või suurtähtedest.

	C++	Java	Python
Süntaks	`ch2 = tolower(ch1);` `ch2 = toupper(ch1);`	`s2 = s1.toLowerCase();` `s2 = s1.toUpperCase();`	`s2 = s1.lower()` `s2 = s1.upper()`

C/C++ teisendatakse märke. Kogu stringi teisendamiseks on mugav kasutada algorithm teegis olevat `transform` funktsiooni:

```
transform(s1.begin(), s1.end(), s1.begin(), tolower);
```

1.4.3 Stringide ühendamine ehk liitmine

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ülesannet lahendav kood on nüüd selline:

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

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

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

Vastav C++ kood on siin:

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

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

1.5 SISEND-VÄLJUND

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

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

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

1.5.1 Standardvood

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

C++

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

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

Java

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

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

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

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

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

Python

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

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

Sisendvoo katkestamine

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

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

1.5.2 Failidest lugemine ja kirjutamine

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

	C/C++	C++	Python
päisfail	`csdtio`	`fstream`	-
Failimuutuja	`FILE`	`ifstream` `ofstream`	
Faili avamine lugemiseks fn - failinimi	`fopen(fn, "rd")`	`open(fn)`	`open(fn, "r")`
Faili avamine kirjutamiseks	`fopen(fn, "wt")`		`open(fn, "w")`
Failist lugemine (nt kaks täisarvu x1 ja x2 failist f)	`fscanf(f, "%d %d", &x1, &x2)`	`f >> x1 >> x2`	
Rea lugemine s – rida n – rea pikkus f - failimuutuja	`fgets(s, n, f)`	`getline(f, s)`	`f.readline()`
Faili kirjutamine (nt kaks täisarvu x1 ja x2 faili f)	`fprintf(f, "%d %d", x1, x2 )`	`f <<x1<<" "<<x2;`	`f.write()`
Faili sulgemine	`fclose(f)`	`f.close()`	`f.close()`

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

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

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

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

1.5.3 Sisendi töötlus

Sisendi lugemise ja käsitlemise erinevate võimaluste illustreerimiseks vaatame järgmist ülesannet:

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

Võrdleme Javas sisendi lugemise kaht levinumat viisi: `Scanneri` ja `BufferedReaderi` kasutamist.

Esimene lahendus:

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

Teine lahendus:

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

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

`Scanner` parsib sisendit ja sellel on mitmed programmeerija jaoks mugavad meetodid, nagu näiteks `nextInt()`. `BufferedReader` on lihtsalt üks suur puhver ja sinna loetud infot tuleb programmeerijal ise edasi parsida. Nagu näha, on suurte sisendandmete korral nn toores andmete lugemine ja nende hilisem töötlemine oluliselt kiirem.

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

C++ stiilis:

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

C stiilis:

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

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

1.5.4 Väljund

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

Korrektsuse seisukohalt on ülesannetel vahel eritingimused, mis nõuavad väljundi kirjutamist konkreetsete reeglite alusel formaadituna – näiteks peab reaalarvuline vastus olema antud täpselt kolme kohaga pärast koma. Selliste reeglite realiseerimiseks on andmete väljastamise funktsioonidel

parameetrid, mille kaudu saab vastavaid reegleid spetsifitseerida. Järgnevas tabelis on selleks mõned näited:

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

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

> Algajatel võistlejatel on sageli kombeks C++ keeles kasutada `endl` iga rea väljastamiseks, kuna see lisab ka reavahetuse. Kui vastuseks on vaja väljastada palju ridu, võib see aga tekitada kergesti olukorra, kus muidu korrektse programmi tööaeg ei mahu etteantud piiridesse. Sellisel juhul aitab, kui kasutada reavahetuse märki, näiteks
> `cout << vastus << '\n';`

Puhvri tühjendamiseks on eri keeltes järgmised meetodid:

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

1.5.5 Jooksvalt töötamine

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

Vaatame selle kontseptsiooni illustreerimiseks järgmist ülesannet:

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

Naiivne meetod selleks on teha nii:

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

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

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

Võistlejatel tasub kindlasti meeles pidada, et C++ kasutades on sisend/väljund kiirem kui kasutada `printf/scanf` funktsioone.
Ka `cin/cout` kiirendamiseks on mõned võimalused:
`ios_base::sync_with_stdio(false)` katkestab sünkroniseerimise C ja C++ standardvoogude vahel. Siin tuleb tähele panna, et sellisel juhul ei tohi kasutada C ja C++ stiilis sisendi/väljundi edastamise funktsioone läbisegi.
`cin.tie(NULL)` – `tie`-meetod tagab, et väljund kirjutatakse standardväljundisse iga kord enne sisendi lugemist. Seda on vaja interaktiivse sisendi/väljundi puhul, kus korduvalt on vaja kordamööda lugeda ja kirjutada, aga muul juhul see lihtsalt aeglustab sisendi/väljundi lugemist-kirjutamist.
Need meetodid tuleb lisada programmi `main` funktsiooni.

1.6 PROGRAMMEERIMISKEELTE VÕRDLUS

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

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

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

1.6.1 C++

Taani arvutiteadlane Bjarne Stroustrup uuris 1980. aastate alguses oma doktoritöö jaoks objekt-orienteeritud programmeerimist, kasutades peamiselt Simula 67 keelt. Simula oli praktiliseks kasutamiseks liiga aeglane, mistõttu Stroustrup hakkas lisama objekt-orienteerituse printsiipe C

keelele. Aastal 1983 valmiski C++ keele esimene variant. Paljud teised tänapäeval laialt levinud keeled nagu Java ja C# on sellest tugevalt mõjutatud.

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

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

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

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

ja objekt sellest klassist:

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

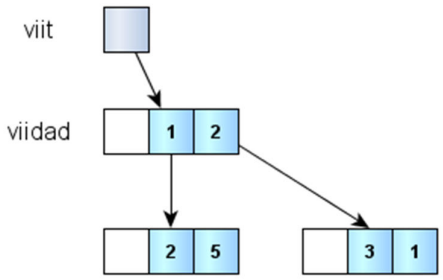
C++ kirjutab objekti väljad lihtsalt järjest mällu:

Kui meil on massiiv punktidest

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

siis kirjutatakse ka need lihtsalt järjest mällu.

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



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

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

1.6.2 Java

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

Sama koodi mitmel platvormil kasutatavuse ehk porditavuse eesmärk saavutatakse järgmiselt:

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

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

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

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

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

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

1.6.3 Python

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

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

Käesoleva peatüki esimese näiteülesande saab Pythonis kirjutada lihtsalt nii:

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

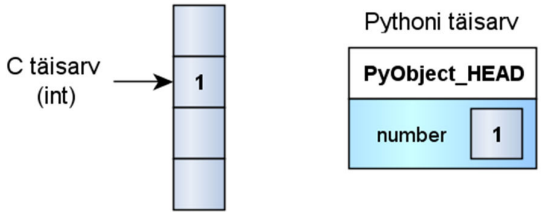
Pole vaja mingeid loogelisi sulge, spetsiaalseid funktsiooninimesid ega viiteid välistele teekidele.

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

Pythoni aeglusel on mitmed põhjused:

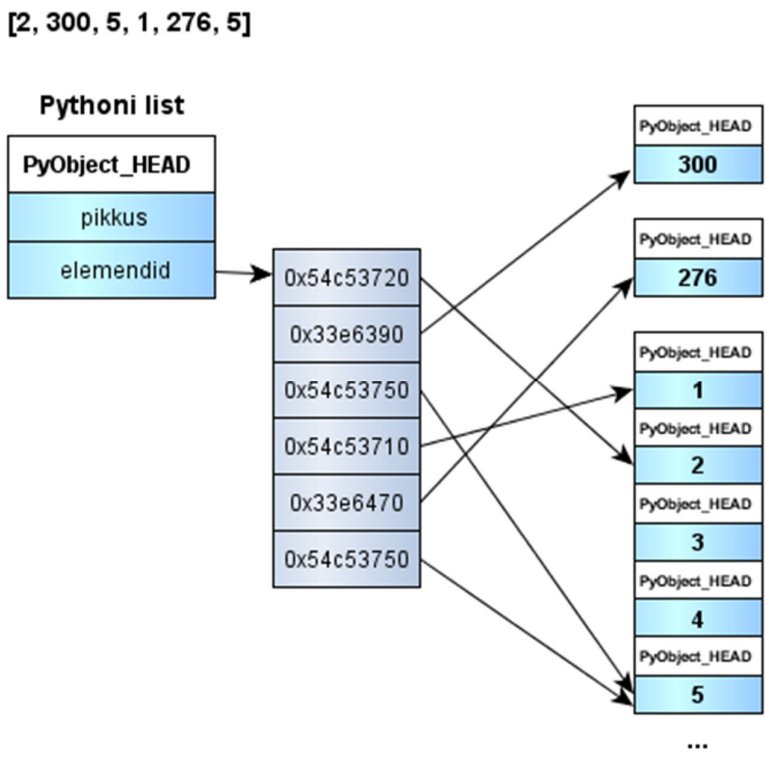
Esiteks on Python on **interpreteeritav** keel, mis tähendab, et koodi ei kompileerita, kompilaatorid aga optimeerivad koodi väga suure määral.

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



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

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



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

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

1.7 TESTIMINE

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

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

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

Järgnevalt vaatleme mõningaid olulisi testide kategooriad.

1.7.1 Piirjuhud

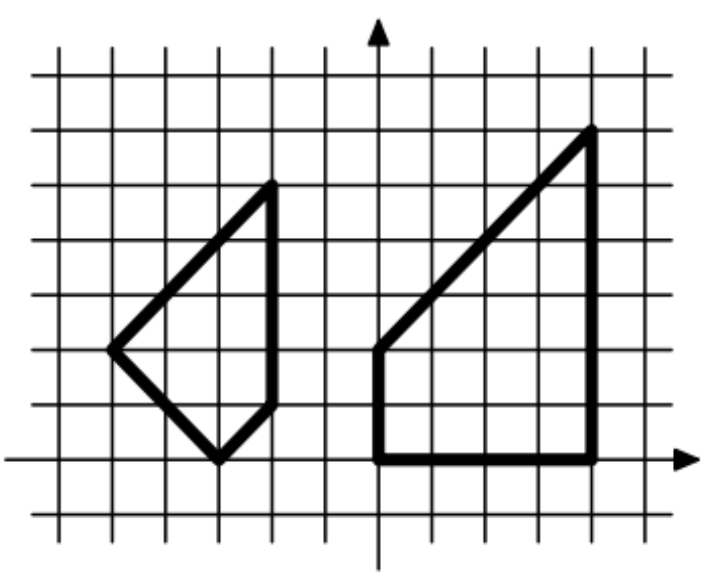
Enamikul ülesannetel esinevad **piirjuhud**, mida testidega üldjuhul kontrollitakse. Kui ülesandeks on leida tee linnast A linna B, tuleb kindlasti vaadelda juhtu, kus A ja B on sama linn. Kui vaja on leida arvu tegurid, tuleb käsitleda ka arvu 1. Kui juttu on ruudustikul toimuvast lauamängust, peab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

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

1.7.2 Õigsuse kontroll

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

Juku õpib koolis hulknurkade sarnasust ja saab teada, et hulknurgad on sarnased, kui nende vastavate nurkade suurused on võrdsed ja vastavate külgede pikkused võrdelised. Sarnased hulknurgad võivad olla omavahel pööratud, peegeldatud ja nihutatud. Sarnaste hulknurkade vastavate külgede pikkuste jagatist nimetatakse nende sarnasusteguriks.
Kodutööna saab ta hulga hulknurki, mille sarnasustegureid on vaja määrata. Jukul on fanaatiline matemaatikaõpetaja, kes andis tööna väga paljude nurkadega hulknurki. Aita Juku hädast välja.
Sisend. Tekstifaili esimesel real on hulknurga tippude arv N ($3 \le N \le 200\,000$).
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.
Hindamine. Pooltes testides on teada, et iga hulknurga küljed on kõik erineva pikkusega.



NÄIDE (vastavad hulknurgad on ka joonisel):

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

Vastus:

```
1.414213
3
```

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

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

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

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

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

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

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

1.7.3 Algoritmi jõudluse kontroll

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.8 SILUMINE

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

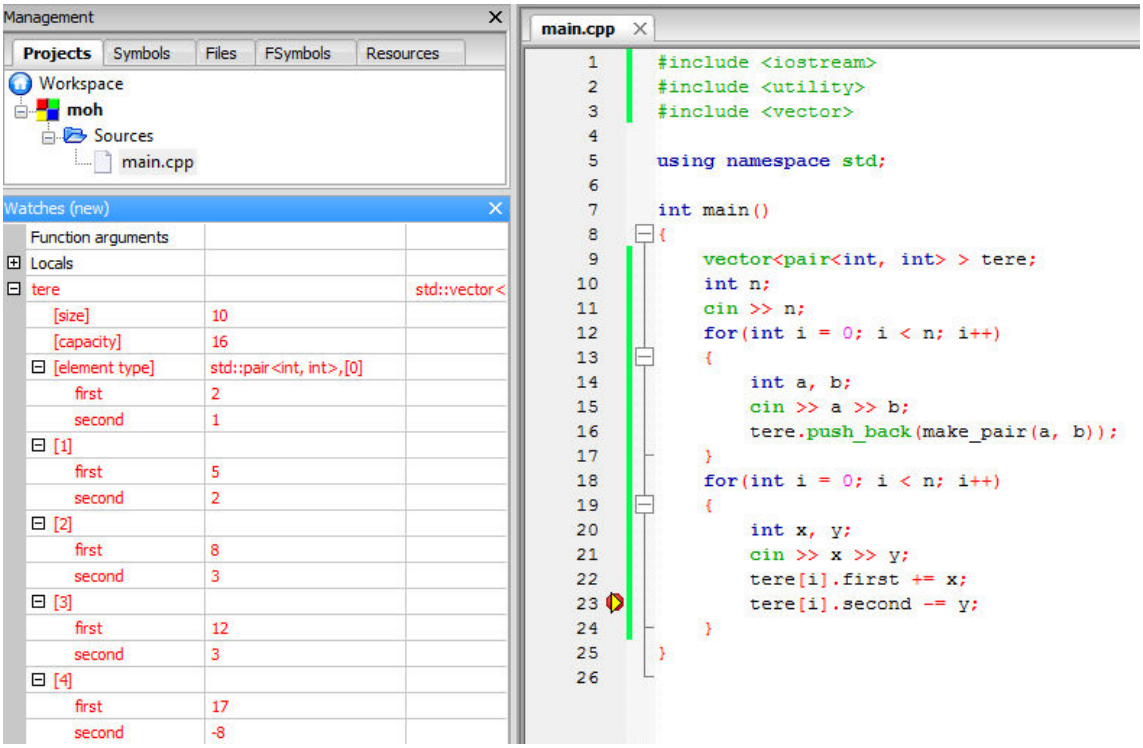
Tõhusam on kasutada spetsiaalset **silumisprogrammi** ehk **silurit**, mis võimaldab koodi samm-sammult läbi käia ja vaadata, mis seal sees täpselt toimub. Sellist tegevust nimetatakse **silumiseks** (*debugging* ehk meeleolukalt „putukate ärastamine“). Põhimõtteliselt võib siluris avada iga programmi,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

1.8.1 Arenduskeskkonnad

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

Enamik programmeerijaid kasutab tänapäeval aga pigem mõnd **integreeritud arenduskeskkonda** (*integrated development environment, IDE*), kus on koos lähtekoodi redigeerimise, kompileerimise ja silumise võimalused. Tavaliselt on kompilaator ja silur siiski eraldi programmid, kuid arenduskeskkond peidab nad oma kasutajaliidese taha.

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



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

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

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

1.8.2 Aja mõõtmine

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

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

Näide C++ ajamõõtmisest:

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

Näide Javas:

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

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

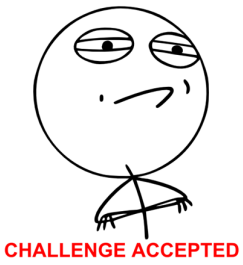
Näide Pythonis (alates 3.3):

```
import time

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

1.9 KONTROLLÜLESANDED

Esimese peatüki kontrollülesanded on mõeldud üldise programmeerimistehnika harjutamiseks ega vaja sügavamaid teadmisi algoritmidest, vastates raskuselt ligikaudu Eesti olümpiaadide lihtsaimatele ülesannetele. Kui need ülesanded



liiga kerged tunduvad, pole vaja muretseda, järgmistes peatükkides läheb asi raskemaks.

1.9.1 Järelmaks

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

NÄIDE 1:
`1000 20 5`
Vastus:
`2`
(Esimese kuu lõpus on väärtused võrdsed, seega peame ootama teise kuuni.)

	Alguses	1.kuu lõpuks	2. kuu lõpuks
Arvuti väärtus	1000	950	902,5
Järelmaksujääk	1000	950	900

Tabel väärtustega esimese näite juurde

NÄIDE 2:
`1000 10 5`
Vastus:
`1`
(Nõutav olukord tekib juba esimese kuu lõpuks.)

NÄIDE 3:
`2000 20 10`
Vastus:
`17`
Märkus: nagu paljusid võistlusülesandeid, saab ka seda ülesannet lahendada mitmel viisil. Võib kogu protsessi simuleerida või siis tuletada üldvalemi vastuse koheseks leidmiseks.

1.9.2 Kell

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

NÄIDE 1:
`9:00`
Vastus:
`90`

NÄIDE 2:
`8:10`
Vastus:
`175`
NÄIDE 3:
`23:59`
Vastus:
`5.5`

1.9.3 Tigu

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

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

1.9.4 3D printer

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


```
8
5
6
5
0
5
3
6
4
```
Vastus:
```
14
```

1.9.5 Mõttemeister

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

NÄIDE 1:

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

Vastus:

```
1 2
```

NÄIDE 2:

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

Vastus:

```
0 0
```

1.9.6 Male

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

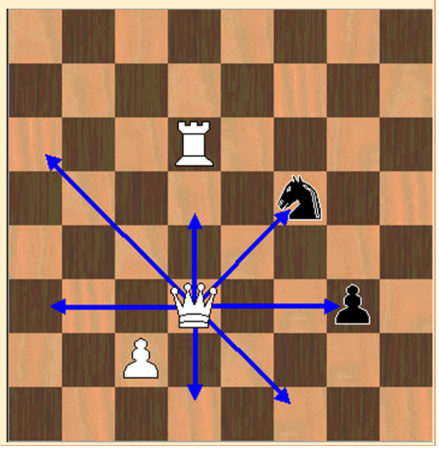
NÄIDE 1:

```
6
a3
a1 a6 a8 f4 c4 b4
```

Vastus:

```
3
```

(a1, a6 ja b4)



NÄIDE 2:

```
4
c3
a5 b6 e8 h1
```

Vastus:

```
1
```

1.9.7 Riigihanked

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

NÄIDE:

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

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

1.9.8 Bender

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

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

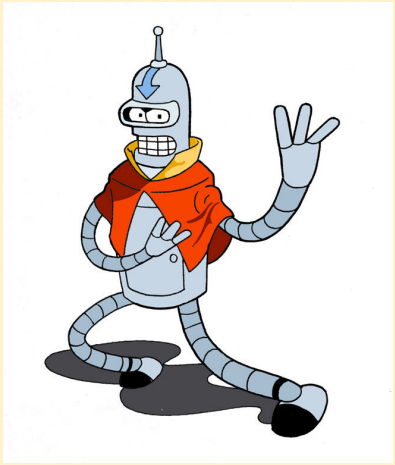
Väljastada täpset üks märk - üks kuuest suunast, milles võib pärast kõiki painutusi olla traadi lõpp: lisaks juba mainitud `U`, `D`, `R` ja `L` suundadele ka `F` (otse, samasuunaline traadi algusega) ja `B` (tagasi, vastassuunaline traadi algusega).

NÄIDE 1:

```
4
R
U
R
```

Vastus:

```
B
```



NÄIDE 2:

```
10
U
L
D
R
U
R
D
L
R
L
```

Vastus:

```
U
```

1.9.9 Pangatellerid

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

NÄIDE:

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

Vastus: 16,2

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

1.9.10 Interpretaator

Üks ebaviisakalt nimetatud programmeerimiskeel (https://en.wikipedia.org/wiki/Brainfuck) on disainitud võimalikult minimalistlikuks. Käesolevas ülesandes uurime selle veelgi vähendatud varianti. Programm kasutab viita, mis võib näidata erinevatele mälupesadele ja nendes olevaid väärtusi muuta. Töö alguses näitab viit esimesele mälupesale. Programmi tekstis saab kasutada nelja operaatorit:
> suurendab viida aadressi ühe võrra.
< vähendab viida aadressi ühe võrra.
\+ suurendab viidatavat väärtust ühe võrra.
\- vähendab viidatavat väärtust ühe võrra.
Masinal, mis nendest käskudest koosnevat programmi täidab, on 100 pesast koosnev mälu ning üks viit sellele mälule. Mälu on ringikujuline: kui me läheme ühelt poolt üle serva, satume tagasi teise serva. Mälupesades hoitav maksimaalne väärtus on 255 ja need väärtused on samuti „ringikujulised“, s.t 255 + 1 = 0 ja 0 – 1 = 255. Alguses on kõigis mälupesades nullid.
Sisendi ainsal real on antud Brainfuckis kirjutatud programm. Väljundisse kirjutada 100 tühikutega eraldatud täisarvu: mälupesade väärtused pärast programmi lõpuni jooksmist.
NÄIDE:

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

Vastus:

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

1.10 VIITED LISAMATERJALIDELE

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

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