Võistlusprogrammeerimine

ehk kuidas kirjutada kiiret koodi

Targo Tennisberg, Katrin Gabrel

0 SISSEJUHATUS

0.1 MIS ON VÕISTLUSPROGRAMMEERIMINE?

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

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

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

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

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

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

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

Siin on üks lihtne näide võistlusstiilis ülesandest:

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

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

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

Testimisel on üldjuhul neli võimalikku tulemust:

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

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

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

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

0.2 MIDA SIIT ÕPIKUST LEIDA VÕIB?

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

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

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

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

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

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

Raamatust võivad kasu saada mitmesugused huvilised:

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

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

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

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

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

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

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

Iga peatükk sisaldab järgmisi osi:

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

Kõik näited on saadaval kolmes programmeerimiskeeles:

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

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

0.3 KONTROLLÜLESANNETE LAHENDAMINE

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

Ülesandeid saab lahendada online võistluskeskkonnas CodeForces, kus:

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

Ülesannete lahendamiseks:

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

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

0.4 KASUTATUD ÜLESANDED JA PILDID

Kasutatud ülesanded on enamikus omalooming, koostatud kas spetsiaalselt käesoleva raamatu jaoks või taaskasutatud mõnelt Eesti olümpiaadilt. Teistest allikatest pärit ülesanded on vastavalt viidatud.

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

ESIMENE OSA - ALGAJATELE

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

1 PROGRAMMIDE SISEMAAILM

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

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

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

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

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

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

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

1.1 PROGRAMMI ELUTSÜKKEL

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

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

1.1.1 Lähtekood

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

Ülesannet lahendav lähtekood keeles C++ on järgmine:

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

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

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

1.1.2 Lähtekoodi transleerimine

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

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

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

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

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

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

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

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

1.1.3 Teegid

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

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

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

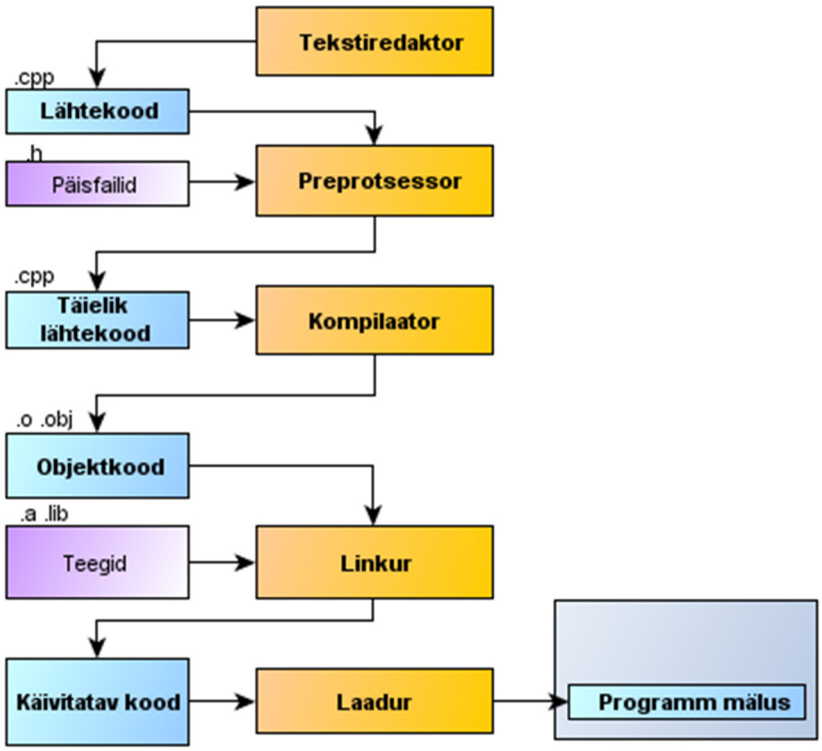
1.1.4 Linkimine

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

1.1.5 Laadimine ja täitmine

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

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



1.2 ANDMETE HOIDMINE JA TÖÖTLEMINE ARVUTIS

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

1.2.1 Protsessori tööpõhimõte

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

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

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

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

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

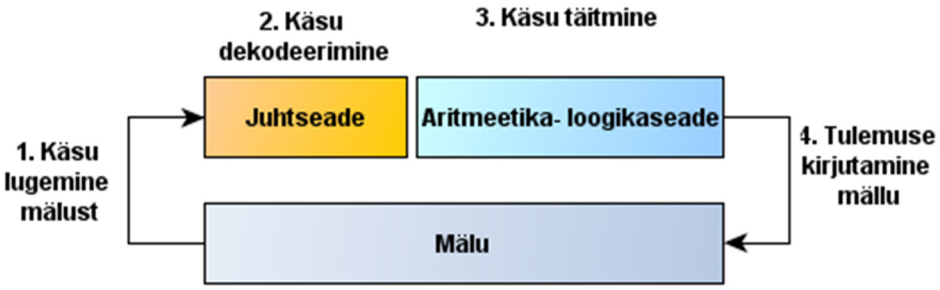
```
Käsukood  Aadress1 Aadress2 Väärtus
00100000001000100000000101010111
```

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

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

Lihtne protsessori töötsükkel on järgmine:

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



1.2.2 Protsessori jõudlus

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

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

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

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

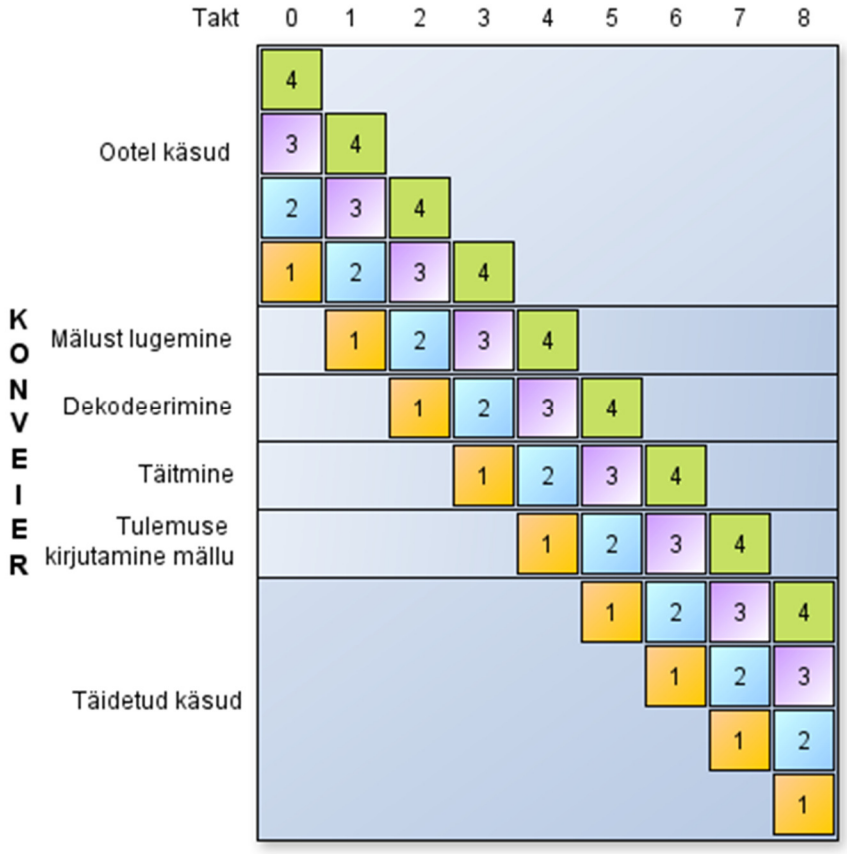
Rusikareeglina soovitatakse võistlusprogrammides arvestada umbes sada miljonit „tavapärast“ operatsiooni sekundi kohta, kus tavapärane on näiteks massiivis kahe väärtuse vahetamine, paarist tehtest koosnev aritmeetiline avaldis või muu suhteliselt lihtsa sisemise keerukusega operatsioon.

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

1.2.3 Käsukonveier

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

Protsessi illustreerib järgmine skeem:



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

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

1.2.4 Hargnemise ennustamine

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

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

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

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

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

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

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

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

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

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

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

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

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

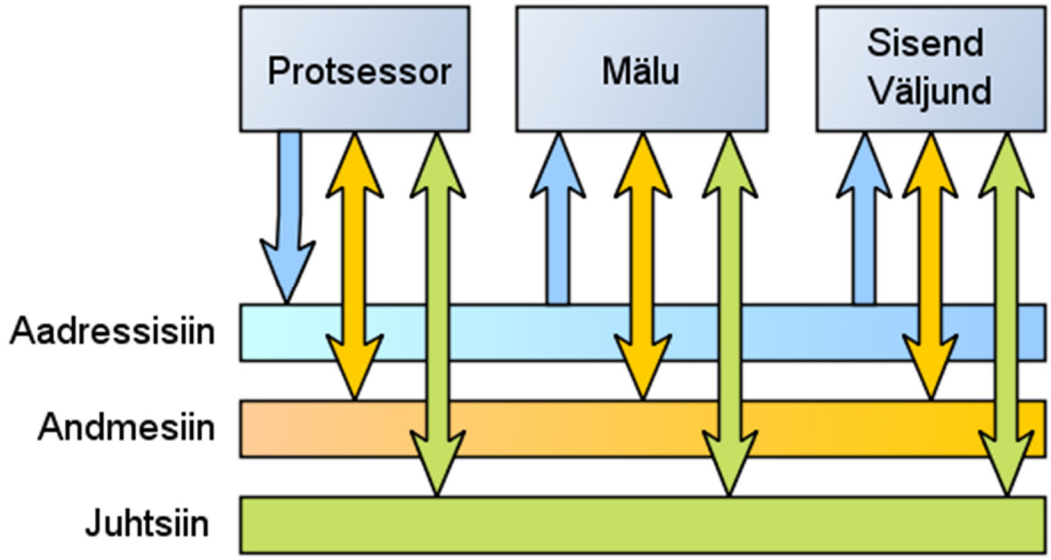
Sellepärast püüabki protsessor **hargnemist ennustada**, s.t mõistatada, kumba haru pidi edasi minnakse. Käsukonveierisse laaditakse vastava haru käsud ette ära. Kui ennustus oli õige, läheb protsess täie kiirusega edasi. Kui aga oli ekslik, siis visatakse konveierist järgmised käsud minema ja laaditakse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

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

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

1.2.5 Andmete liikumine

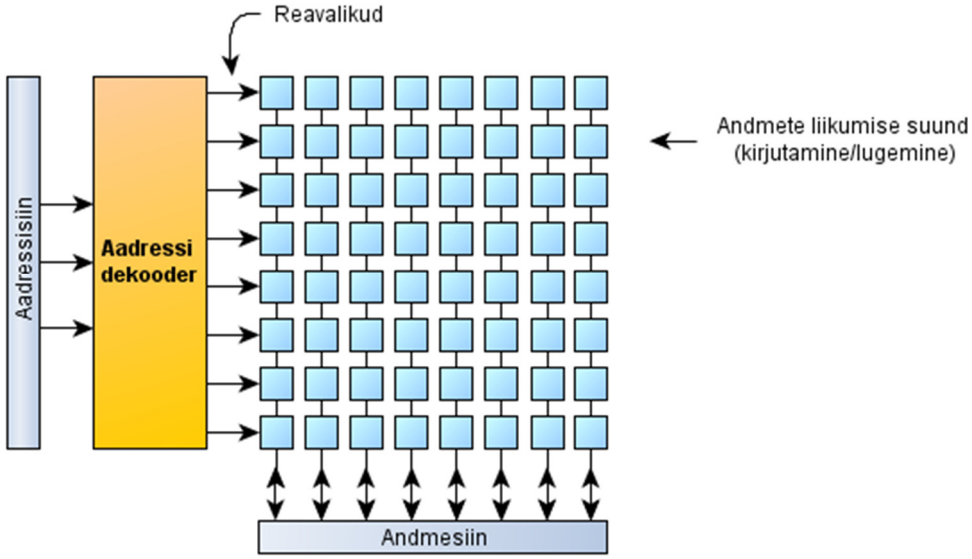
Andmete transportimiseks mälu, protsessori ja sisend-väljundseadmete vahel kasutatakse **siine** (*bus*): **andmesiinil** (*data* bus) liiguvad andmed, **aadressisiinil** (*address bus*) on info, kuhu andmed liiguvad, ja **juhtsiinil** (*control bus)* info, mis suunas ja mille vahel andmed parajasti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



1.2.6 Mälu

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

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



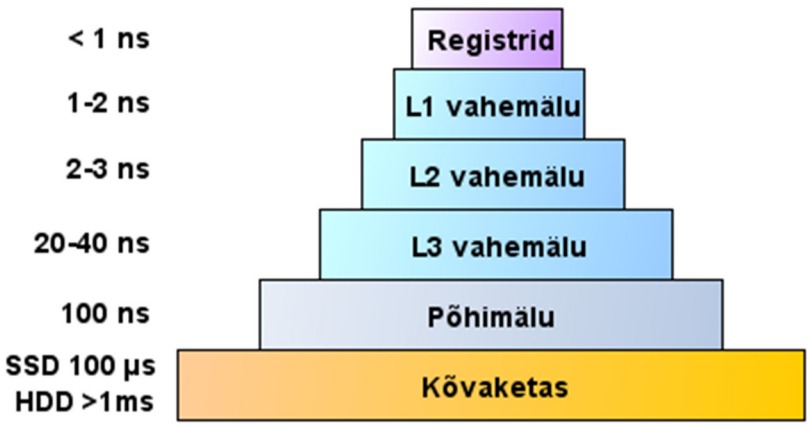
Mälu loeb aadressisiinilt aadressi, aadressi dekooder saadab signaali soovitud aadressliinile ning sellel liinil olevad bitid kirjutatakse andmesiinile või loetakse andmesiinilt vastavale aadressiliinile.

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

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

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

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



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

1.3 ANDMETÜÜBID

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

1.3.1 Täisarvud

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

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

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

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

Enamlevinud täisarvutüübid ja nende pikkused on toodud järgnevas tabelis:

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

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

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

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

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

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

Ületäitumine

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

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

1.3.2 Ujukomaarvud

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

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

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

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

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

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

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

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

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

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

Probleemid ujukomaarvudega

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

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

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

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

Teeme kontrolliks läbi järgmise näite Pythonis:

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

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

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

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

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

Pythonis on selleks ka funktsioon `math.isclose`:

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

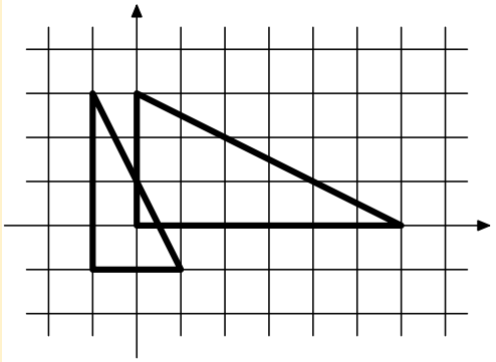
Ujukomaarvu konverteerimisel täisarvuks kasutatakse enamasti 0 suunas ümardamist, mis tähendab murdosa ärajätmist. Seetõttu võib täisarvuks konverteerimine anda ootamatuid tulemusi. Soovitatav on kasutada alati eelnevalt mõnd sobivat ümardamismeetodit ning vajadusel võtta taas abiks epsilon.

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

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

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

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



NÄIDE 1:

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

Vastus: `1.5` (vt joonist)

NÄIDE 2:

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

Vastus: `-1`

Esimene pähetulev lahendus on selline:

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

using namespace std;

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

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

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

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

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

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

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

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

Tulemusena saame järgmise koodi:

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

using namespace std;

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

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

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

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

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

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

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

1.3.3 Püsikomaarvud

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

1.3.4 Märgid

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

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

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

1.3.5 Tõeväärtused

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

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

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

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

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

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

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

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

Loogikatehted

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

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

Näiteks lauses

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

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

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

1.3.6 Viidad

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

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

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

1.3.7 Struktuursed tüübid ehk liittüübid

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

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

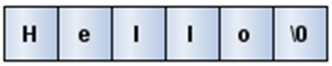
1.3.8 Kirjed

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

1.3.9 Stringid

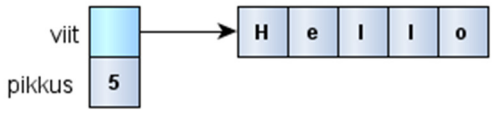
String ehk sõne on märkide jada ja kuna seda kasutatakse palju, on see sageli defineeritud eraldi andmetüübina. Stringide hoidmiseks mälus on kasutusel peamiselt kolm erinevat võimalust:

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



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

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



1.3.10 Massiivid

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

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

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

Mida väljastab selline kood?

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

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

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

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

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

1.4 OPERATSIOONID STRINGIDEGA

1.4.1 Märk kohal i

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

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

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

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

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

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

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

1.4.2 Stringide võrdlemine

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

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

1.4.3 Stringide ühendamine ehk liitmine

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ülesannet lahendav kood on nüüd selline:

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

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

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

Vastav C++ kood on siin:

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

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

1.5 SISEND-VÄLJUND

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

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

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

1.5.1 Standardvood

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

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

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

Java

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

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

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

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

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

Python

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

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

Sisendvoo katkestamine

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

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

1.5.2 Failidest lugemine ja kirjutamine

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

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

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

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

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

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

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

1.5.3 Sisendi töötlus

Sisendi lugemise ja käsitlemise erinevate võimaluste illustreerimiseks vaatame järgmist ülesannet:

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

Nagu eelnevalt juttu oli, on Javas sisendi lugemiseks kaks levinumat viisi: `Scanner`i ja `BufferedReader`i kasutamine.

Esimene lahendus:

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

Teine lahendus:

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

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

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

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

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

C++ stiilis:

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

C stiilis:

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

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

1.5.4 Väljund

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

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

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

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

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

Puhvri tühjendamiseks on eri keeltes järgmised meetodid:

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

1.5.5 Jooksvalt töötamine

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

Vaatame selle kontseptsiooni illustreerimiseks järgmist ülesannet:

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

Naiivne meetod selleks on teha nii:

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

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

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

1.6 PROGRAMMEERIMISKEELTE VÕRDLUS

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

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

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

1.6.1 C++

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

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

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

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

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

ja objekt sellest klassist:

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

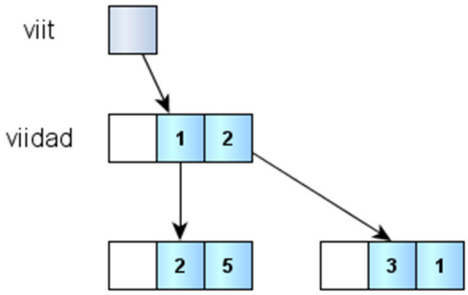
C++ kirjutab objekti väljad lihtsalt järjest mällu:

Kui meil on massiiv punktidest

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

siis kirjutatakse ka need lihtsalt järjest mällu.

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



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

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

1.6.2 Java

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

Sama koodi mitmel platvormil kasutatavuse ehk porditavuse eesmärk saavutatakse järgmiselt:

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

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

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

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

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

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

1.6.3 Python

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

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

Käesoleva peatüki esimese näiteülesande saab Pythonis kirjutada lihtsalt nii:

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

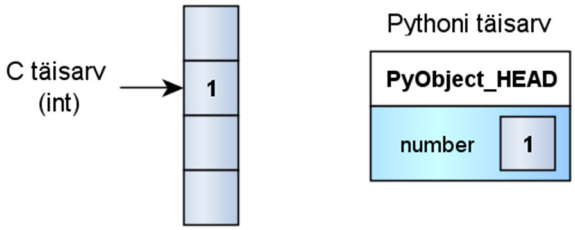
Pole vaja mingeid loogelisi sulge, spetsiaalseid funktsiooninimesid ega viiteid välistele teekidele.

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

Pythoni aeglusel on mitmed põhjused:

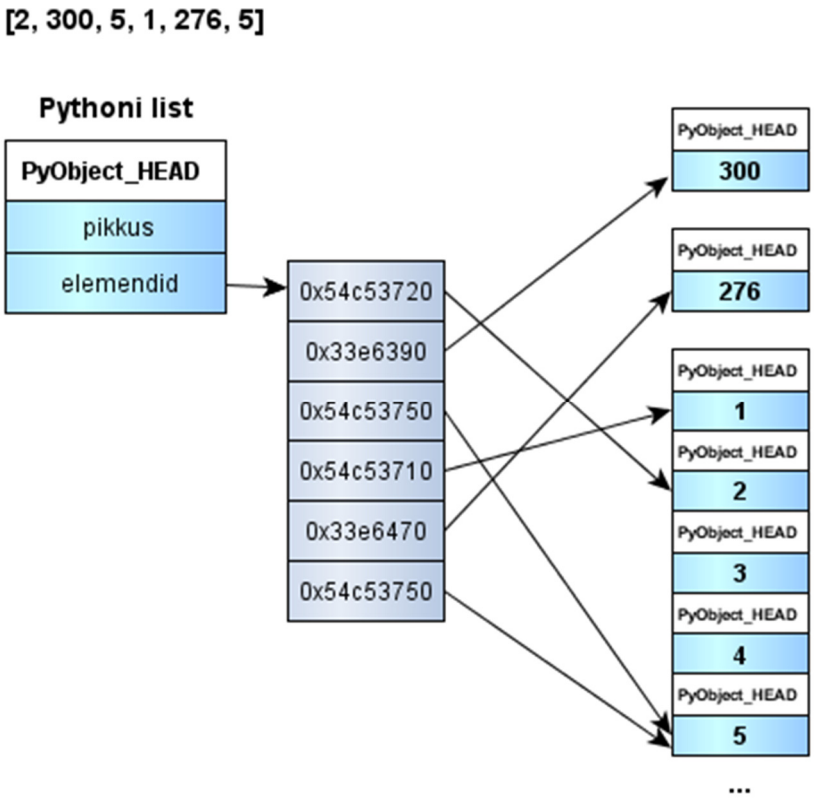
Esiteks on Python on **interpreteeritav** keel, mis tähendab, et koodi ei kompileerita, kompilaatorid aga optimeerivad koodi väga suure määral.

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



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

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



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

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

1.7 TESTIMINE

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

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

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

Järgnevalt mõned olulised testide kategooriad:

1.7.1 Piirjuhud

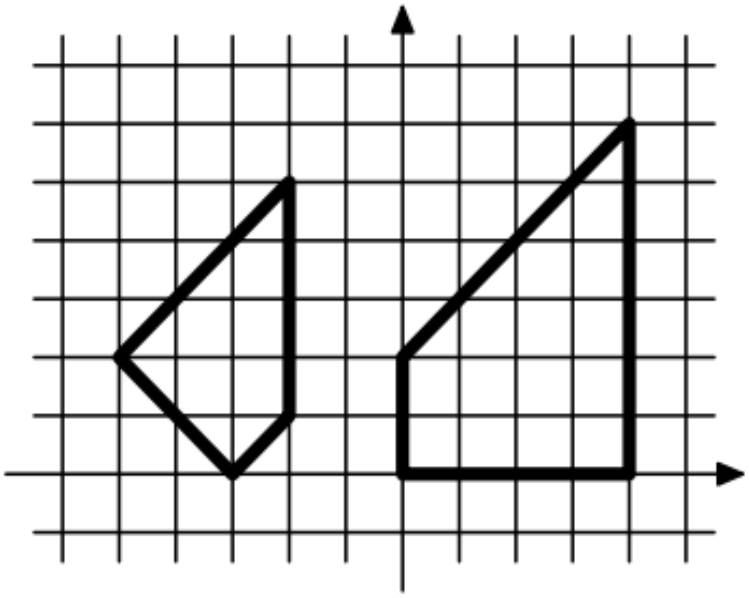
Enamikul ülesannetel esinevad **piirjuhud**, mida testidega üldjuhul kontrollitakse. Kui ülesandeks on leida tee linnast A linna B, tuleb kindlasti vaadelda juhtu, kus A ja B on sama linn. Kui vaja on leida arvu tegurid, tuleb käsitleda ka arvu 1. Kui juttu on ruudustikul toimuvast lauamängust, peab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

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

1.7.2 Õigsuse kontroll

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

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



NÄIDE (vastavad hulknurgad on ka joonisel):

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

Vastus:

```
1.414213
3
```

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

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

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

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

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

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

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

1.7.3 Algoritmi jõudluse kontroll

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.8 SILUMINE

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

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

1.8.1 Arenduskeskkonnad

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

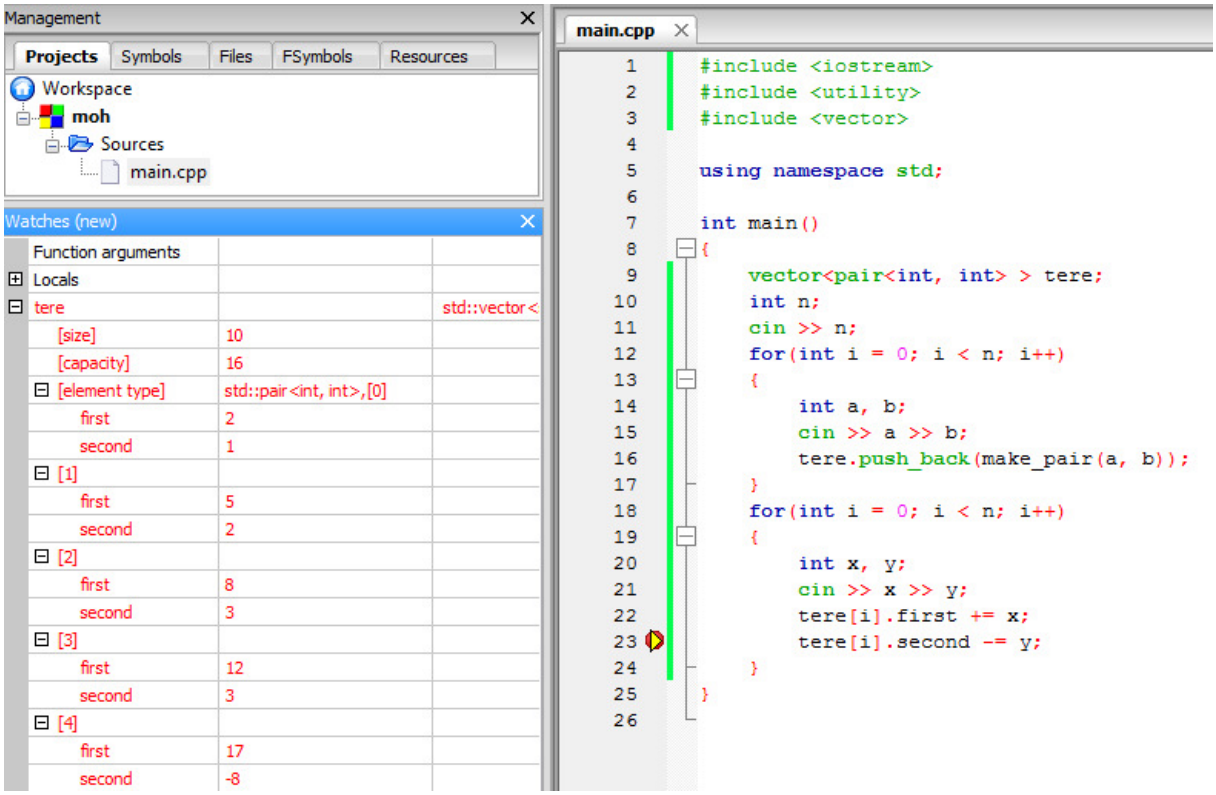
Enamik programmeerijaid kasutab tänapäeval aga pigem mõnd **integreeritud arenduskeskkonda** (*integrated development environment, IDE*), kus on koos lähtekoodi redigeerimise, kompileerimise ja

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

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

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

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



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

1.8.2 Aja mõõtmine

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

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

Näide C++ ajamõõtmisest:

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

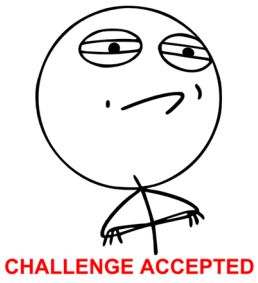
Näide Javas:

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

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

Näide Pythonis (alates 3.3):

```
import time

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

1.9 KONTROLLÜLESANDED



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

1.9.1 Järelmaks

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

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

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

NÄIDE 2:
2000 20 10
Vastus: 17

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

1.9.2 Kell

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

NÄIDE 1:
9:00
Vastus: 90

NÄIDE 2:
8:10
Vastus: 175

NÄIDE 3:
23:59
Vastus: 5.5

1.9.3 Tigu

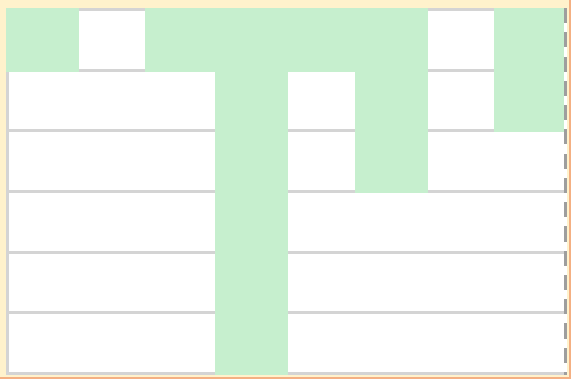
H meetri sügavuse kaevu põhjas on tigu, kes proovib sealt välja roomata. Iga päev suudab tigu roomata U meetrit ülespoole ja igal öösel libiseb ta D meetrit allapoole. Kuna kaevus pole toitu, väsib tigu igal järgmisel päeval järjest rohkem ära. Teisel päeval suudab ta ronida T protsenti vähem kui esimesel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

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

1.9.4 3D printer

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

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



1.9.5 Mõttemeister

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

Sisend koosneb kahest reast: esimesel real Kodeerija loodud kood ja teisel real Arvaja esitatud pakkumine. Väljundisse kirjutada kaks arvu: punaste ja valgete hindenuppude arv.

NÄIDE 1:

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

Vastus: `1 2`

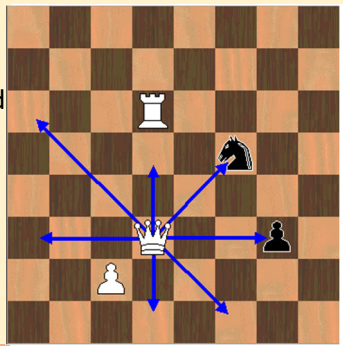
NÄIDE 2:

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

Vastus: `0 0`

1.9.6 Male

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



NÄIDE:

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

Vastus: 3 (a1, a6 ja b4)

1.9.7 Riigihanked

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

NÄIDE:

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

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

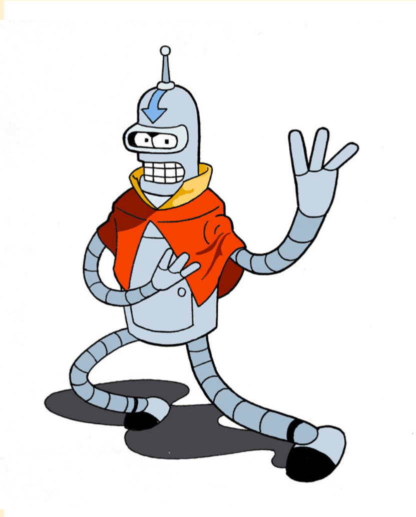
1.9.8 Bender

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

Leida, millises suunas on pärast kõiki painutusi traadi lõpp.

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

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



NÄIDE:

```
4
R
U
R
```

Vastus:

```
B
```

1.9.9 Interpretaator

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

> suurendab viida aadressi ühe võrra.

< vähendab viida aadressi ühe võrra.

+ suurendab viidatavat väärtust ühe võrra.

- vähendab viidatavat väärtust ühe võrra.

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

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

NÄIDE:

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

Vastus:

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

1.9.10 Pangatellerid

Pangas on N tellerit, kes kliente teenindavad. Kui klient tuleb panka, läheb ta esimese telleri juurde. Kui esimene teller on hõivatud, läheb ta teise juurde jne. Kui kõik tellerid on hõivatud, siis inimesed ootavad järjekorras ja lähevad selle telleri juurde, kes parajasti vabaneb.
Tellerid püüavad muuhulgas klientidele mitmesuguseid teenuseid müüa, aga neil on selleks erinev võimekus. Müüdud teenuste väärtus on telleri müügioskuse ja kliendi pangakonto seisu korrutis jagatud tuhandega.
Meil on pangapäeva kohta teada, millal kliendid saabusid ning lahkusid, samuti iga kliendi pangakonto seis ja iga telleri müügivõimekus. Leida, kui palju pangateenuseid sel päeval müüdi.
Sisendi esimesel real on kaks täisarvu: tellerite arv N ($1 \leq N \leq 100$) ja klientide arv M ($1 \leq M \leq 2000$). Järgmistel N real on igaühel üks täisarv, mis tähistab vastava telleri müügivõimekust (täisarv 1-100). Järgmistel M real on klientide pangakontode seisud (täisarvud 1 - 10000). Lõpuks tuleb 2M rida, igaühel üks täisarv, mis tähistavad klientide saabumise ja lahkumise järjekorda. Positiivsed arvud tähistavad vastava numbriga kliendi saabumist ja negatiivsed arvud tema lahkumist.

NÄIDE:

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

Vastus: 16,2

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

1.10 VIITED LISAMATERJALIDELE

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

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

2 LÄBIVAATUS- JA OTSINGUALGORITMID

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

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

Käesolevas peatükis vaatleme tehnikaid keeruliste ülesannete lihtsustamiseks.

2.1 ALAMPROGRAMMID

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

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

Programmi jagamine alamosadeks on kasulik mitmel põhjusel:

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

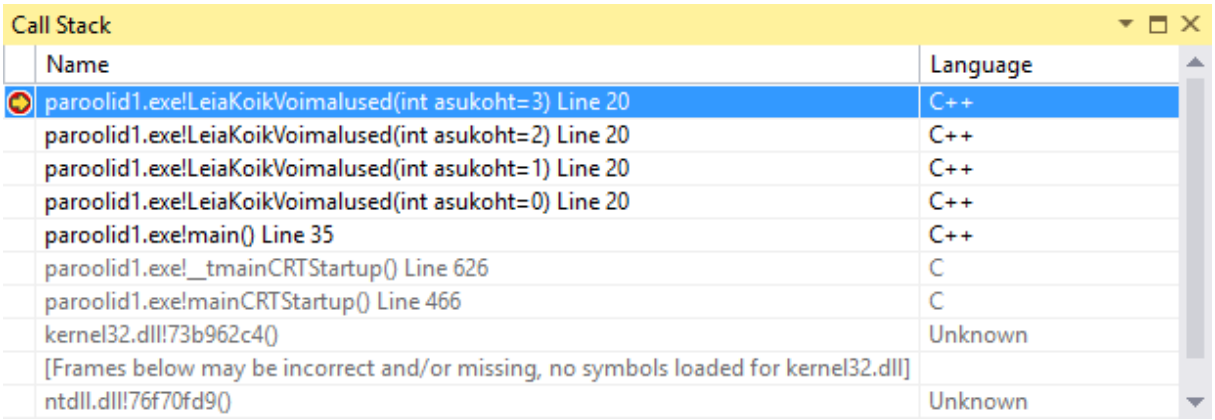
Sõltuvalt programmeerimiskeelest võidakse alamprogramme nimetada funktsioonideks, protseduurideks või meetoditeks. C ja sellest põlvnevad keeled kasutavad tavaliselt terminit „funktsioon“, mis viitab sarnasusele matemaatilise funktsiooniga. Nagu matemaatilisel funktsioonilgi, võivad programmi funktsioonil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

2.1.1 Pinumälu

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

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

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



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

2.1.2 Funktsiooni kohalikud andmed

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

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

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

2.1.3 Pinu ületäitumine

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

Ületäitumise illustreerimiseks kasutame järgmist iseennast väljakutsuvat funktsiooni:

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

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

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

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

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

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

2.1.4 Pinu kasutamine protsessoris

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

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

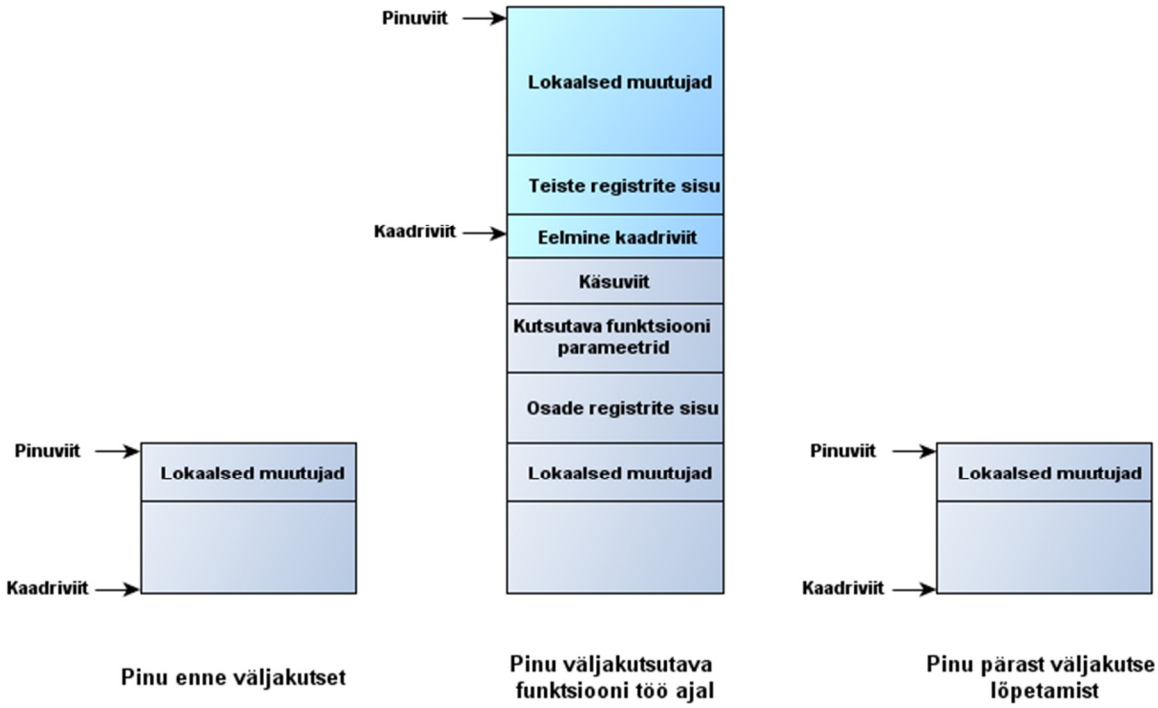
Ka väljakutustaval funktsioonil endal võivad olla muutujad. Needki hoitakse pinus. Siin tekib aga probleem, et kui kutsutud funktsioon asub välja kutsuma omakorda järgmist funktsiooni jne, siis läheb keeruliseks arvet pidada, kus üks pinukaader hakkab ja teine lõpeb. Selleks on kasutusel spetsiaalne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

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

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

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

Pinukaadrite struktuur on kokkuvõttes selline:



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

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

2.1.5 Kuhimälu

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

Pinu ja kuhja erinevusi illustreerib järgmine tabel:

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

2.1.6 Funktsiooni parameetrid

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

Vaatame järgmist koodi:

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

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

    vaheta(a, b);
}
```

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

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

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

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

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

2.1.7 Objektide edastamine parameetritena

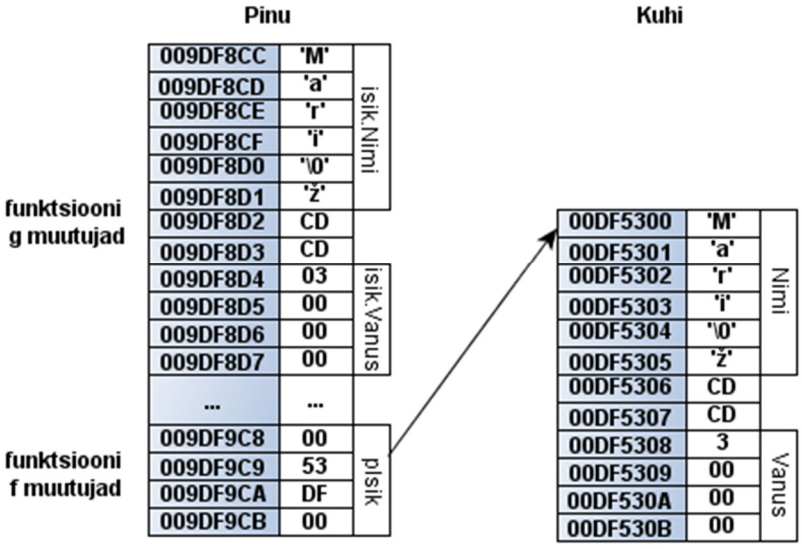
Kui kasutada funktsiooni parameetrina objekte, käituvad keeled mõnevõrra erinevalt. Järgnevalt mõned näited, milles kasutatakse klassi `Isik`, millel on väljad `Nimi` ja `Vanus` ning konstruktor, mis seab ka nime. C++ on klass defineeritud järgmiselt:

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

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

Objekt parameetrina

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

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

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

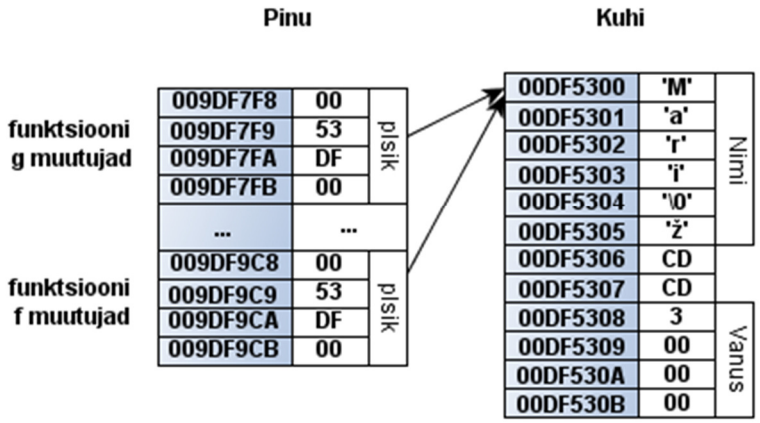


Funktsioonis f on kohaliku muutujana viit kuhimälus loodud Isik-tüüpi objektile. Funktsiooni g väljakutsel luuakse tervest objektist koopia pinus ning g muudab pärast seda ainult oma lokaalset koopiat. Funktsiooni f juurde tagasi pöördudes g koopia „kaob“. Seetõttu väljastab antud programmijupp vanuse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

Aadressiviit parameetrina

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

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

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



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

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

Python

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

Java

Javas lihttüüpide väärtused kopeeritakse, keerulisemad andmetüübid antakse edasi viitadena.

Järgnevalt üks näide Javas:

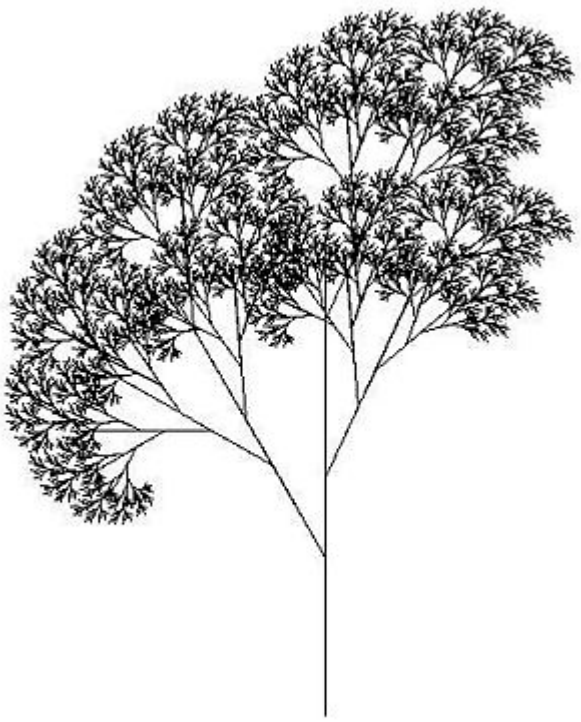
```
public void f() {
    Isik i = new Isik("Mari");
    g(i);
    System.out.println(i);
}
public void g(Isik i) {
    i.nimi = "Maria";
}
```

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

2.2 REKURSIOON

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

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



Programmeerimiskeeles Logo rekursiivselt joonistatud puu

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

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

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

2.2.1 Hargnemiseta rekursioon

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

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

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

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

2.2.2 Rekursioonivalem

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

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

Nii erinevaid samme kui ka baase võib olla mitu.

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

2.2.3 Hargnemistega rekursioon

Vaatame nüüd tõsisemat rekursiooniülesannet:

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

NÄIDE:

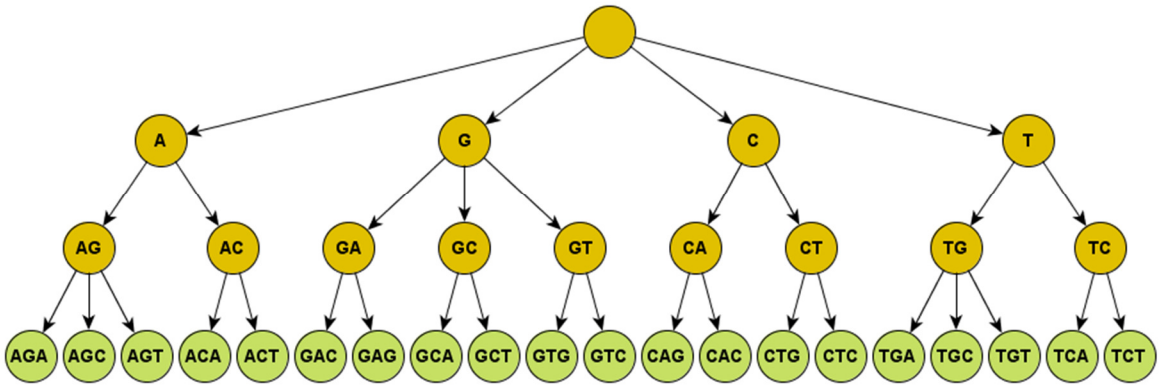
3

Vastus:

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

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

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

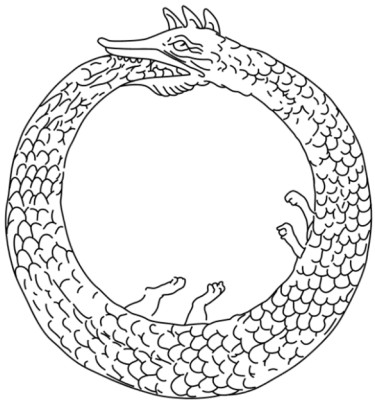


Piirangutega arvestav skeem ülesande lahendite kujunemisest

```
char* vastus;
int pikkus;

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

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

2.2.4 Sabarekursioon ja väljakutsete optimeerimine



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

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

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

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

Ilma optimeerimiseta genereerib minu kompilaator järgmise koodi:

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

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

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

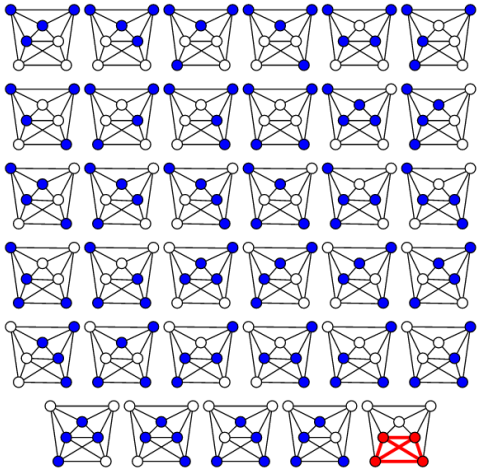
```
012812A0   mov     al,byte ptr [ecx]       ; stringi lõpu kontroll
012812A2   test    al,al
012812A4   je      strtoint+1Bh (012812BBh) ; je="jump if equal". Kui eelmise rea
                                           ; tulemus oli 0, hüppame funktsiooni
                                          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

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

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

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

2.3 VARIANTIDE LÄBIVAATAMINE



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

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

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

2.3.1 Tagurdusmeetod

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

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

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

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

2.3.2 Iteratsioonimeetod

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

Variantide läbivaatuse kontekstis tähendab iteratsioonimeetod, et

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

2.3.3 Tagurduse ja iteratsiooni võrdlus

Kirjeldatud meetodite illustreerimiseks vaatame järgmist ülesannet:

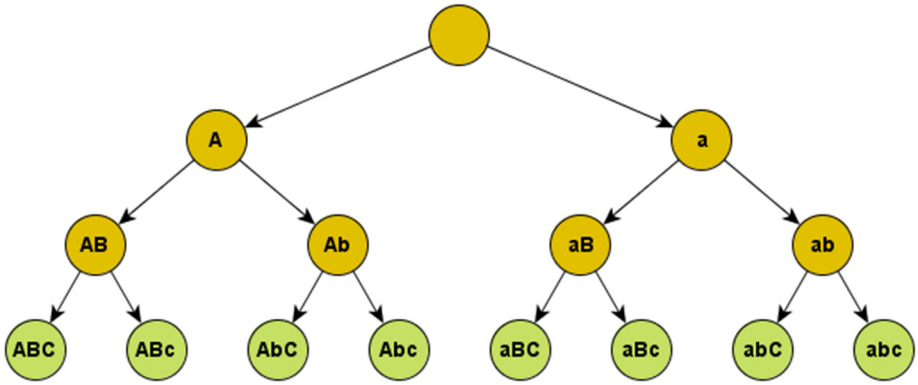
James Bond on tunginud vaenlase peakorterisse, kus superkurjategija parajasti maailma hävitamise masinasse parooli sisestab. Bond näeb sõrmede liigutamise järgi, millised märgid paroolis on, kuid ei näe, millisel hetkel pahalane shift-klahvi all hoiab. Seetõttu ei tea ta, millised on suur- ja millised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

2.3.4 Variantide läbivaatamine tagurdusmeetodiga

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

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

Saame järgmise skeemi:



Ülesande lahendamiseks tagurdusmeetodiga saame kirjutada järgmise funktsiooni:

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

2.3.5 Variantide läbivaatamine iteratsioonimeetodiga

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

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

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

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

2.3.6 Variantide läbivaatus programmeerimisvõistlustel

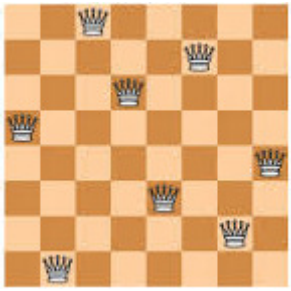
Variantide läbivaatus on enamasti küllaltki aeglane ja kohmakas võte ning mitmete ülesannete lahendamiseks on olemas kiiremad algoritmid. Siiski on variantide läbivaatusel programmeerimisvõistlustel oluline roll.

Variantide läbivaatust kasutatakse tavaliselt järgmistel juhtudel:

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

2.4 LÄBIVAATUSE OPTIMEERIMINE

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



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

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

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

2.4.1 Lihtne tagurdusmeetod

Üks üsna otsene lahendus ülesandele on järgmine:

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

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

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

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

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

2.4.2 Andmete optimeerimine

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

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

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

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

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

2.4.3 Ettearvutamine

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

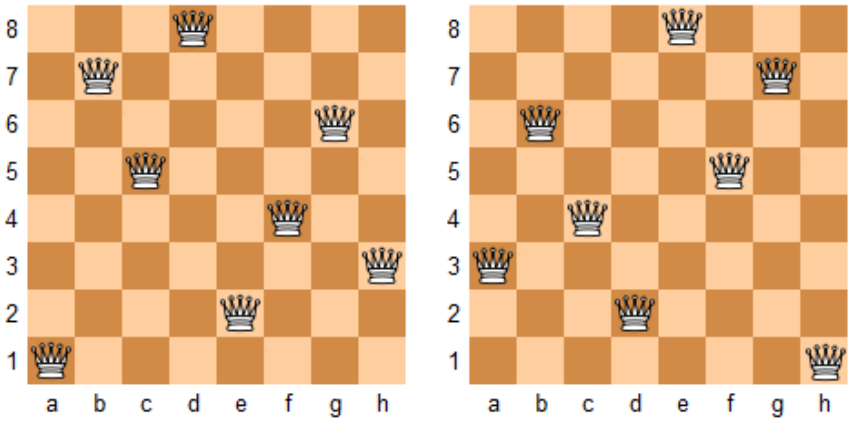
Vastav lahendus on järgmine:

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

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

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

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

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

    return res;
}
```

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

2.4.4 Otsingupuu ahendamine

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



Peegelpildis lahendused

Vastava täiendusega lahendus on selline:

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

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

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

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

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

    return res;
}
```

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

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

2.4.5 Sisemiste tsüklite optimeerimine

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return res;
}
```

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

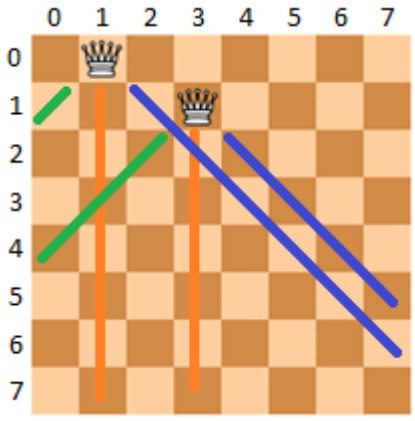
2.4.6 Andmestruktuuride optimeerimine

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

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

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

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

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

    return res;
}

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

Tööaeg oli minu arvutis 1,1 sekundit!



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

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

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

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

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

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

2.4.7 Rekursiooni eemaldamine

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

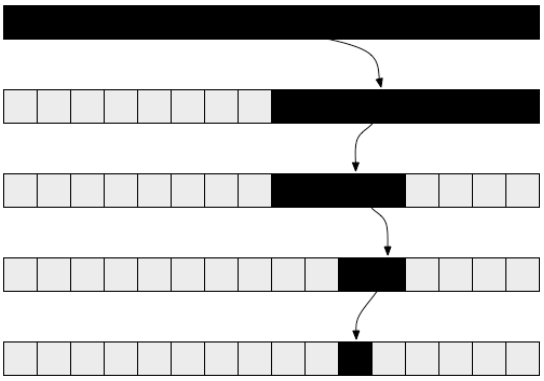
Tulemuseks on selline funktsioon:

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

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

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

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

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

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

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

2.5 KAHENDOTSING



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

2.5.1 Vastuse kahendotsing

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

Kostja kolib Itaaliasse ja pakib rutakalt asju kastidesse. Kuna tal on sellega kiire ja ta on ka üksjagu laisk, siis võtab ta asju sellises järjekorras, nagu kätte juhtub, ja asetab neid järjest kastidesse – kõigepealt ainult esimesse kasti, seejärel ainult teise jne. Kirjuta programm, mis aitab Kostjal valida väikseima võimaliku kasti suuruse, nii et kõik ta `M` asja mahuksid tema pakkumismeetodi juures ära maksimaalselt `N` kasti.
Sisendi esimesel real on antud kastide arv `N` $(1 \le N \le 100000)$, teisel real asjade arv `M` $(1 \le M \le 100000)$ ja viimasel real asjade suurused pakkimise järjekorras $(1 \le M_i \le 1000)$.
NÄIDE:
```
3
6
2 16 3 7 10 12
```
Vastus: 20 (esimesse kasti 1. ja 2. ese, teise 3., 4. ja 5. ese ning kolmandasse viimane ese).

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

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

Siin on kahendotsingu osa:

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

Tsükli lõpuks on muutujas `suurim` ülesande vastus.

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

2.5.2 Kahendotsing andmetest

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

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

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

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

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

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

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

Ja siin iteratiivne lahendus:

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

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

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

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

2.5.3 Topeltkahendotsing

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

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

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

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

2.5.4 Otsing kahemõõtmelisest tabelist sadulameetodil

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

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

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

Vastus:

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

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

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

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

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

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

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

Sadulameetod kahendotsinguga

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

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

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

Reast kahendotsimise funktsioon:

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

Veerust kahendotsimise funktsioon:

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

Sadulameetod topeltkahendotsinguga

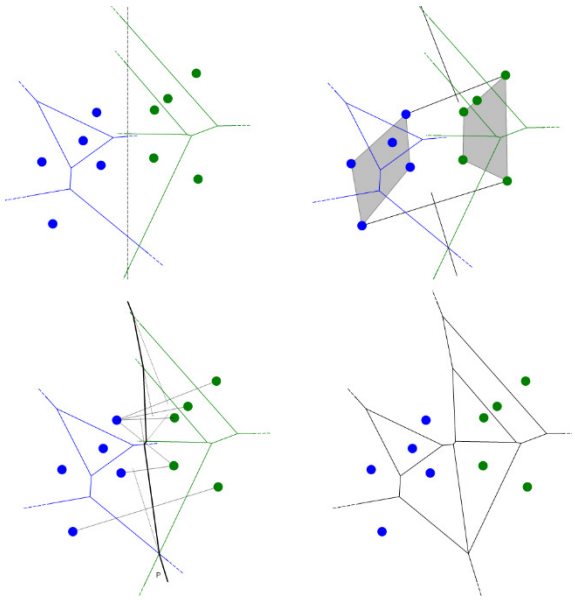
Vaatame veel korra lähemalt, kuidas arvud ridu ja veergu pidi sorteeritud tabelis paiknevad. Järgmisel joonisel on 20x20 tabel, milles on arvud vahemikus 0-1000. Sinised ruudud näitavad teed arvu 409 otsimisel lineaarselt liikudes:

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

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

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

2.6 JAGA JA VALITSE



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

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

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

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

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

```
long double* liidetavad;

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

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

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

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

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

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

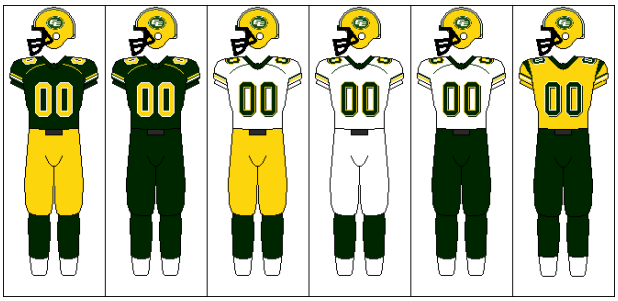
```
long double liidaKiirem(int start, int end)
{
	if (end - start < 1000) { //liita pole rohkem kui 1000 arvu, liidame tavaliselt
		long double vas = 0;
		for (int i = start; i <= end; i++){
			vas += liidetavad[i];
		}
		return vas;
	}
	//liita on rohkem arve, jagame ülesande pooleks
	 int keskkoht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

Tulemused erinevate meetoditega liitmisel:

Meetod	Tulemus
liida	5000428.6446836758
liidaKiirem	5000428.6446836758
Tavaline järjest liitmine	5000428.6446819436

2.7 SISSEJUHATUS KOMBINATOORIKASSE

2.7.1 Permutatsioonid



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

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

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

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

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

2.7.2 Permutatsioonide konstrueerimine

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

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

Üks võimalus on luua permutatsioonid rekursiivselt.

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

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

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

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

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

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

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

Algoritm ise on järgmine:

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

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

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

2.7.3 Kombinatsioonid

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

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

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

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

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

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

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

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

Järgmise ülesande lahendamisel saab kirjeldatud võtteid kasutada:

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

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

Vastus:

```
350
2 3 6 7
```

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

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

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

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

2.8 KONTROLLÜLESANDED

2.8.1 Autoturg

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

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

Väljundisse kirjutada ostjate järjekorras neile enim sobivate autode hinnad.

NÄIDE:

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

Vastus: `7 4 5 7`

2.8.2 Kaupmees ja maksukoguja

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

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

NÄIDE 1:
`59 237`
Vastus: 116 (osta mais ja oktoobris, kõigil teistel kuudel müüa)

NÄIDE 2:
`375 743`
Vastus: 28

NÄIDE 3:
`2500000 8000000`
Vastus: `-12000000`

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

2.8.3 Lippude vasturünnak

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

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

NÄIDE 1:
`1 2 3 4 5 6 7 8`
Vastus: 7

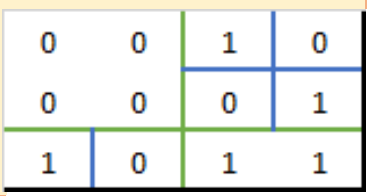
NÄIDE 2:
`1 5 2 1 3 7 4 8`
Vastus: 1 (piisab neljanda lipu liigutamisest kuuendale ruudule).

2.8.4 Akulaadija

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

2.8.5 Pildi pakkimine

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

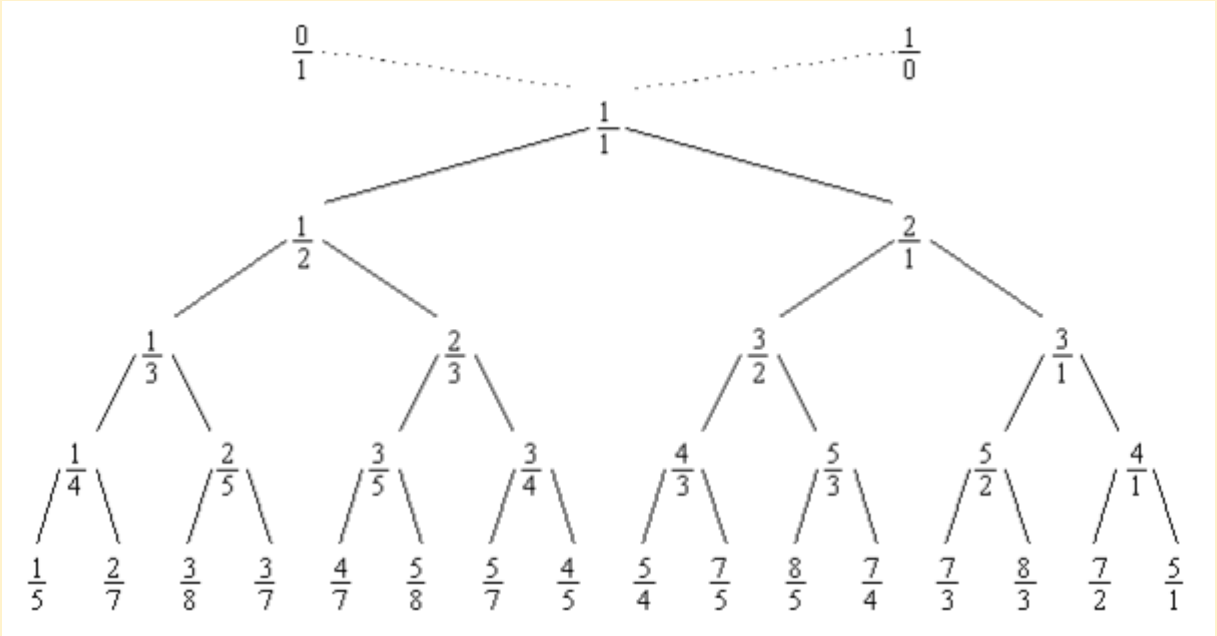


2.8.6 Stern-Brocot' puu

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

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

Tulemuseks on joonisel näidatud puu:



Näiteks $\frac{4}{3}$ on saadud murdudest $\frac{1}{1}$ ja $\frac{3}{2}$. Äärmiste väärtuste jaoks tuleb appi võtta esialgsed $\frac{0}{1}$ ja $\frac{1}{0}$, näiteks $\frac{4}{1}$ saadakse $\frac{3}{1}$ ja $\frac{1}{0}$ abil. Matemaatiliselt saab kergesti näidata, et kõik saadud murrud on taandumatud ja erinevad ning lõputu puu sisaldab parajasti kõiki positiivseid ratsionaalarve.
Viimase omaduse abil saame iga ratsionaalarvu esitada tema asukohana Stern-Brocot' puus, pannes kirja, kas mingis harus tuleb minna vasakule (L) või paremale (R).
Teatavasti saab iga ratsionaalarvu esitada taandatud lihtmurruna. Sisendis on antud kaks positiivset täisarvu, mis tähistavad otsitava arvu lihtmurruna esituse lugejat ja nimetajat. Väljundisse kirjutada string, mis vastab arvu asukohale Stern-Brocot' puus.

NÄIDE 1:
5 7
Vastus: LRRL

NÄIDE 2:
878 323
Vastus: RRLRRLRLLLLRLRRR

2.8.7 Viinamarjad

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

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

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

NÄIDE 1:

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

Vastus:

```
3
2
4
```

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

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

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

NÄIDE 2:

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

Vastus:

```
3
1
1
0
```

Sabile viinamarjaistandus Lätis

2.8.8 Erinevad summad

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

NÄIDE 1:

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

Vastus:

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

NÄIDE 2:

```
5 3
2 1 1
```

Vastus: (tühi string)

NÄIDE 3:

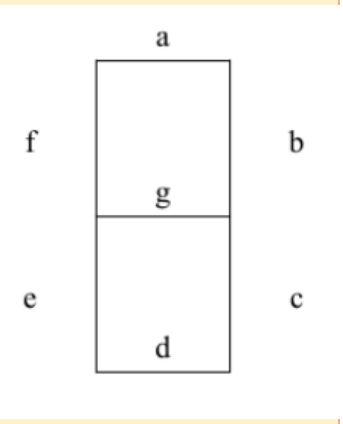
```
400 12
50 50 50 50 50 50 25 25 25 25 25 25
```

Vastus:

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

2.8.9 Kodumasinate näidikud

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




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

Displeid juhib mikrokontroller, millel on funktsioon numbrite allapoole loendamiseks. Sul tuleb kirjutada programm, mis kontrollib põlevate tulede alusel, kas loenduri funktsionaalsus töötab õigesti. Kahjuks on aga testis kasutatavad LEDid ise väga ebakvaliteetsed ja võivad tihti rikki minna ning põlemast lakata, seda nii enne testi kui ka testimise ajal. Kui mõni LED on rikki läinud, siis ta enam korda ei lähe. Seega tuleb pidada testi läbivateks ka juhtusid, kus mõned LEDid on katki läinud.

Sisendi esimesel real on näitude arv `N` (`1≤N≤10`). Järgmisel `N` real on igaühel 7-kohaline kahendarv, mis näitab, kas vastavad tuled põlevad või ei. Kui sisend vastab ükskõik millisele järjestikusele alamjadale jadast `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]` (arvestades ka võimalusega, et osad tuled ei tööta), siis kirjutada väljundisse `JAH`. Vastasel korral kirjutada väljundisse `EI`.

NÄIDE 1:

```
2
NNNNNNN
NNNNNNN
```

Vastus: `JAH`

NÄIDE 2:

```
2
YYYYYYY
YYYYYYY
```

Vastus: `EI`

NÄIDE 3:

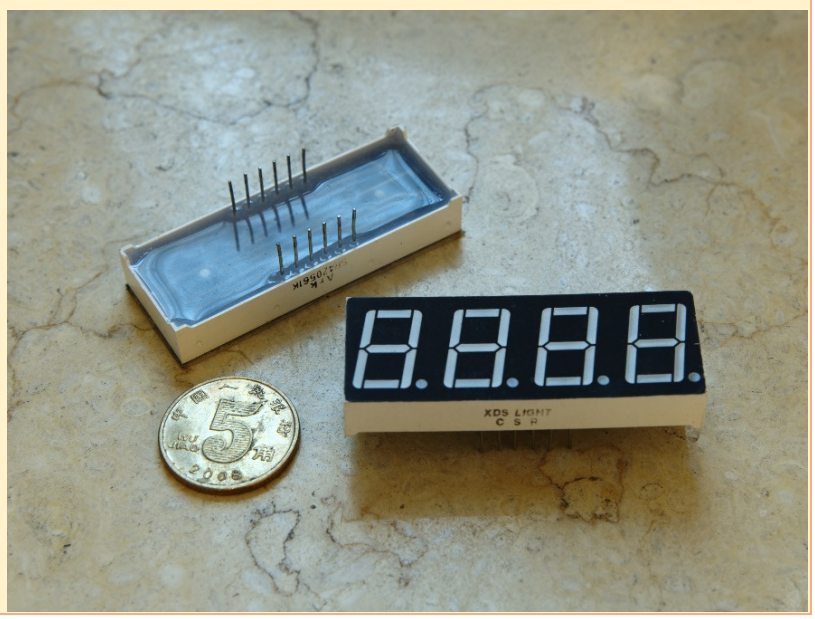
```
3
YNYYYYY
YNYYNYY
NYYNNYY
```

Vastus: `JAH`

NÄIDE 4:

```
3
YNYYYYN
YNYYNYN
NYYNYYN
```

Vastus: `EI`



2.8.10 Graafi värvimine

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

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

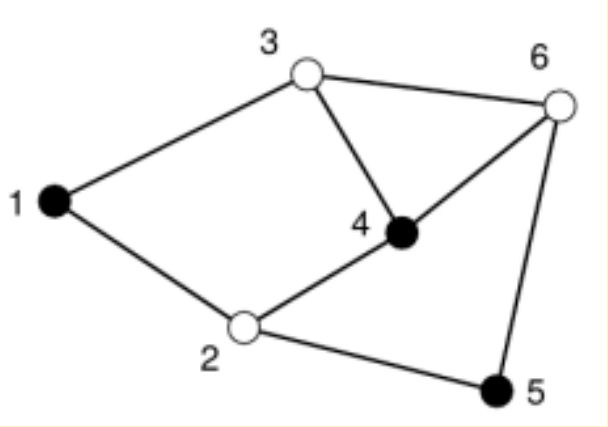
Väljundisse kirjutada kaks rida: esimesele mustaks värvitavate tippude arv, teisele see, millised tipud värvitakse, sordituna väiksemast suuremani. Kui võimalikke lahendusi on mitu, siis väljastada neist minimaalne, s.t selline, kus esimese tipu number on väikseim võimalik, seejärel teise tipu number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

NÄIDE (vastab joonisele):

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

Vastus:

```
3
1 4 5
```



2.9 VIITED LISAMATERJALIDELE

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

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

3 ALGORITMI KEERUKUS JA PÕHILISED ANDMESTRUKTUURID

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

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

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

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

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

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

3.1 ALGORITMI KEERUKUS

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

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

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

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

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

3.1.1 Keskmine ja halvim keerukus

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

3.1.2 Asümptootiline hinnang

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

Funktsioonide kasvukiiruse võrdlemisel kasutatakse tavaliselt järgmisi seoseid:

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

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

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

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

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

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

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

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

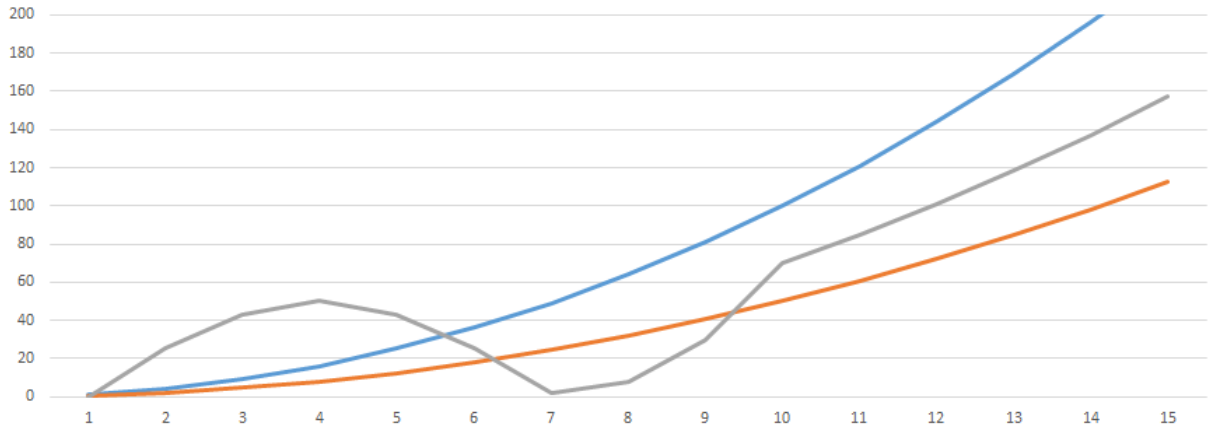
Seekord on funktsiooni f kasv alt tõkestatud funktsiooniga g. Väited $f(n) \in O\big(g(n)\big)$ ja $g(n) \in \Omega\big(f(n)\big)$ on samaväärsed.

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

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

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

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



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

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

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

3.1.3 Kasvuseoste omadused

Kasvuseostel kehtivad järgmised kasulikud omadused:

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

3.2 ALGORITMI KEERUKUSE HINDAMINE

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

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

kus $f_i(n)$ on i. käsu täitmisaeg.

3.2.1 Hargnemiste keerukuse hindamine

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

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

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

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

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

3.2.2 Tsüklite keerukuse hindamine

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

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

3.2.3 Mitmekordsete tsüklite keerukus

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

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

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

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

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

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

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

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

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

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

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

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

3.2.4 Rekursiooni keerukuse hindamine

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

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

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

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

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

Sagedamini esinevate rekursiivsete algoritmide võrrandid on toodud järgnevas tabelis:

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

3.3 TAVALISED KEERUKUSKLASSID

Vaatame siin algoritmide tavalisemaid keerukusklasse ja nende esinemise juhtusid.

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

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

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

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

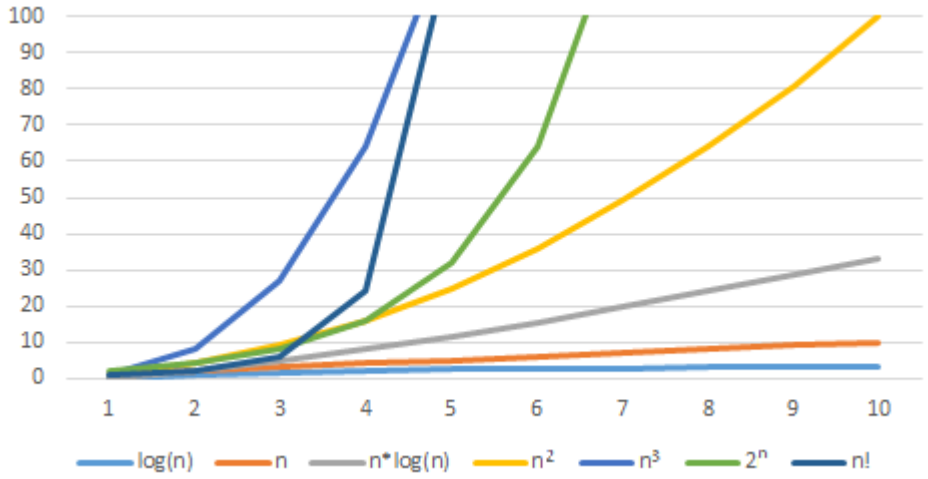
$O(n^2)$ – **ruutkeerukusega** algoritmid on tavaliselt kahekordse tsükliga algoritmid – niimoodi on näiteks võimalik läbi vaadata sisendi elementide kõikvõimalikud paarid.

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

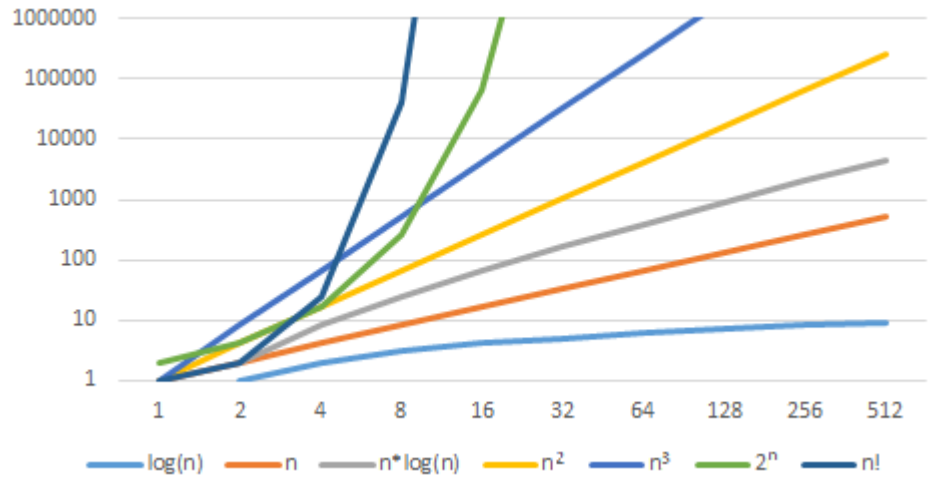
$O(2^n)$ – **eksponentsiaalse** keerukusega algoritmid käivad tavaliselt läbi sisendi kõikvõimalikud alamhulgad.

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

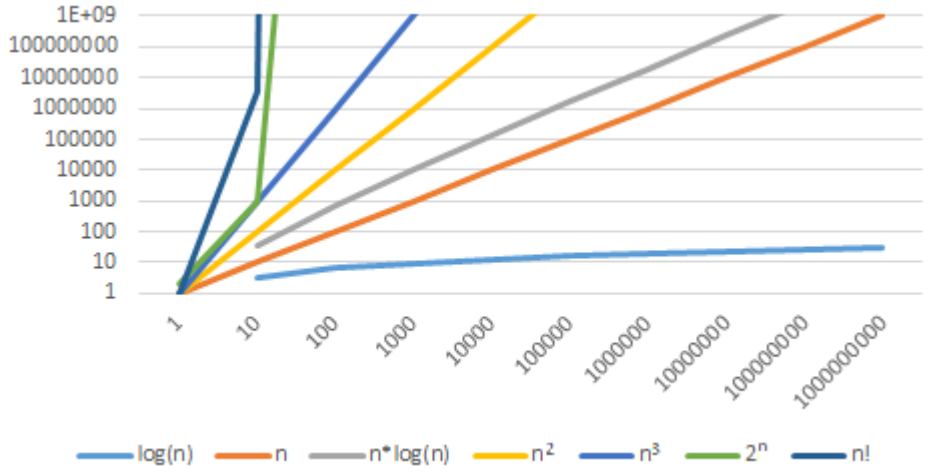
Nii näevad tavaliste keerukusklasside funktsioonid välja graafikul:



Järgmisena on esitatud samad funktsioonid logaritmilisel skaalal:



Veel üks graafik praktikas enam kasutatavate väärtustega:



3.3.1 Keerukusklass ja ajalimiit

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

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

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

3.4 ANDMESTRUKTUURID

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

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

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

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

3.4.1 Andmestruktuuride klassifitseerimine

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

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

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

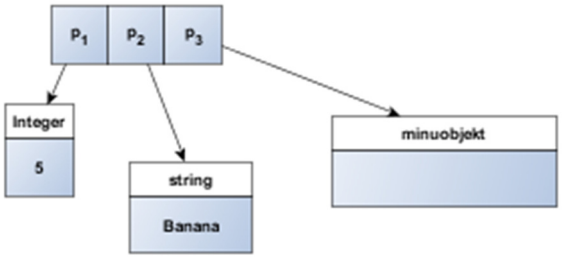
3.5 MASSIIV

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

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

3.5.1 Massiivid Pythonis

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



3.5.2 Massiivi võimalused ja piirangud

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

3.5.3 Mitmemõõtmelised massiivid

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

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

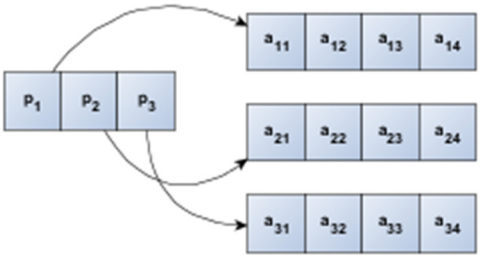
a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}

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

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

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

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



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

3.5.4 Dünaamilised massiivid

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

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

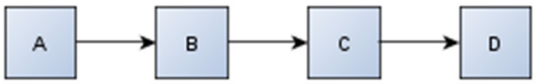
Muidugi võtab elementide kopeerimine uude massiivi aega, kuid suurendades iga kord vajadusel massiivi pikkust 2 korda, on see küllaltki harv olukord ning keskmine elemendi lisamise keerukus massiivi lõppu on enamikel juhtudel konstantne. Teisalt jälle võtab dünaamiline massiiv enamasti mälus rohkem ruumi,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

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

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

3.6 AHEL

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



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

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

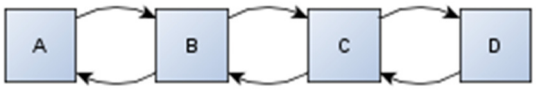
Ahelal on massiivi ees järgmised eelised:

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

Ahelal on ka negatiivsed küljed:

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

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



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

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

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

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

3.7 PINU

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

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

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

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

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

Pinu realiseeritakse enamasti (dünaamilise) massiivina. Massiivi korral tuleb valida piisavalt suur massiiv ning hoida meeles elementide arvu pinus. Tähistades massiivi A-ga ja pinus olevate elementide arvu n-ga, siis lisamisel suurendatakse n-i ühe võrra ja lisatakse uus väärtus kohale a_n. Eemaldamisel tagastatakse kohal a_n olev väärtus ja vähendatakse n-i ühe võrra.

Pinu on võimalik realiseerida ka ahelana. Sellisel juhul lisatakse uus element alati ahela algusesse ning eemaldamise korral tagastatakse ja eemaldatakse esimene element ahelast.

Mõlemal juhul on eemaldamise ja lisamise keerukus $O(1)$.

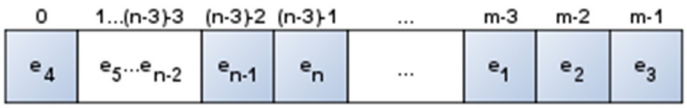
Ajalooliselt on pinu kohta eesti keeles kasutatud ka sõna „magasin“. Folkloori kohaselt kasutati Tartu Ülikoolis üht terminit ja Tallinnas teist, aga tänapäeval on „pinu“ rohkem levinud. Kõige paremini saab magasini mõiste selgeks aga siis, kui sul on kell 9 algoritmide ja andmestruktuuride eksam, aga ärkad kell 9:20. Siis lööd käega vastu otsmikku: „Oh, magasin!“

3.8 JÄRJEKORD

Ka järjekord on selline andmestruktuur, kus elementidele pääseb ligi kindlas järjekorras. Järjekorra põhimõte on nagu tavalises poesabas: uus element lisatakse alati lõppu ja järgmine element loetakse ja eemaldatakse alati algusest. Sellist põhimõtet tähistab lühend FIFO (*first in, first out* – esimesena sisse, esimesena välja).

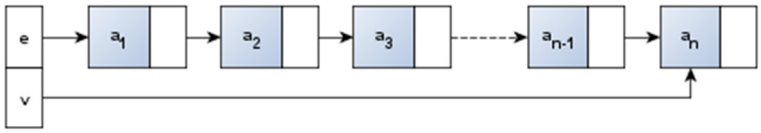
Sarnaselt pinuga on järjekorral operatsioonid elemendi lisamiseks ja eemaldamiseks (koos lugemisega).

Järjekord realiseeritakse tavaliselt samuti massiivina või ahelana, kuid mõlemad on veidi keerulisemad kui pinu korral. Massiivi kasutades on probleemiks see, et järjekorra korral ei püsi kumbki ots muutumatuna. Esimeseks võimaluseks on muidugi kas lisamisel või eemaldamisel nihutada kõiki järjekorras olevaid elemente massiivis, kuid sellisel juhul on lisamise või eemaldamise protseduuri keerukuseks juba $O(n)$. Teiseks võimaluseks on meeles pidada lisaks elementide arvule ka esimese elemendi indeks, mis siis elementi eemaldades nihkub ühe võrra edasi. Sellisel juhul aga on vaja päris korraliku pikkusvaruga massiivi ning ületäitumise oht on väga suur. Selle lahendusena käiakse massiivis ringiratast – massiivi viimasele elemendile järgneb järjekorras jälle esimene. Kuna on teada, et algus on alati lõpust eespool, siis selline lahendus toimib päris hästi.



Järjekord massiivina. m - massiivi pikkus, n - elementide arv järjekorras, $e_1 \ldots e_n$ – elemendid järjekorras.

Vajadus elementidele mõlemast otsast ligi pääseda, teeb ka lihtahela kasutamise järjekorra jaoks mõnevõrra keerulisemaks kui pinu realiseerimisel. Topeltseotud ahelat siiski vaja ei ole, piisab, kui peame eraldi meeles viita viimasele elemendile – sel moel saab lisada uut elementi kiiresti järjekorra lõppu. Tähelepanelik tuleb aga olla olukorras, kus järjekorras ongi vaid üks element – sellisel juhul on see korraga nii esimene kui ka viimane ning selle eemaldamisel tuleb nullida ahela viit nii esimesele kui ka viimasele elemendile.



Järjekord lihtahelana. e - viit esimesele elemendile, v – viit viimasele elemendile, $a_1 \ldots a_n$ – elemendid järjekorras.

Ka järjekorras on mõlemal juhul eemaldamise ja lisamise keerukus $O(1)$.

3.8.1 Kaheotsaline järjekord

Kaheotsaline järjekord (*double ended queue, deque*) on selline andmestruktuur, kus elemendi lisamine ja võtmine on lubatud mõlemast otsast. Praktikas on see andmestruktuur huvitav sellepärast, et seda saab omakorda kasutada nii järjekorra kui ka pinu realiseerimiseks.

3.8.2 Eelistusjärjekord

Eelistusjärjekord (*priority queue*) on selline järjekord, kus igal elemendil on prioriteet ning elemente võetakse järjekorrast vastavalt elemendi prioriteedile. Päris elus kasutatakse eelistusjärjekorda näiteks erakorralise meditsiini osakonnas (EMO) - kõigepealt hinnatakse patsiendi seisundi tõsidust ja kõrgema prioriteediga patsiente teenindatakse enne. Veelgi keerulisemaks teeb olukorra see, et mõnel juhul – nagu ka EMOs – võib elemendi prioriteet muutuda ning siis on vaja seda elementi järjekorras „nihutada“. Seetõttu peab eelistusjärjekorra realiseerimisel arvestama sellega, et elemendi võtmisel eelistusjärjekorrast tuleb otsida pidevalt, milline on suurima prioriteediga element, või tuleb hoida elemente pidevalt prioriteedi järgi sorteerituna, mis teeb jällegi lisamise ja prioriteedi muutmise keerulisemaks. Seetõttu on eelistusjärjekorra realiseerimiseks kõige parem andmestruktuur hoopiski kuhi, millest tuleb juttu alampeatükis 3.9.

3.9 PINU JA JÄRJEKORRA KASUTAMINE

Enamasti on keelte standardteekides pinu juba valmiskujul olemas. C++ on päisfailis `<stack>` klass `stack`, Javas on parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Deque`. Pythonis eraldi klassi pinu jaoks ei ole, kuid Pythoni `listil` on meetodid `append()` järjendi lõppu elemendi lisamiseks ja `pop()` elemendi eemaldamiseks lõpust.

Järgmine tabel illustreerib pinu kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Pinu p loomine (täisarvude jaoks)	stack<int> p	Deque<Integer> p = new ArrayDeque<Integer> ()	p = []
Elemendi 'a' lisamine	p.push(a)	p.push(a)	p.append(a)
Elemendi eemaldamine	p.pop()	a = p.pop()	a = p.pop()
Pealmise elemendi vaatamine	a = p.top()	a = p.peek()	a= p[-1]
Kas pinu on tühi?	p.empty()	p.isEmpty()	not p

Järjekorra leiab C++ keeles päisfailist `<queue>` (klass queue); Javas on taas üldjuhul parim kasutada `ArrayDeque` klassi `java.collections`'i liidesega `Queue`. Pythonis saab kasutada struktuuri `collections.deque`.

Järgmine tabel illustreerib järjekorra kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine (täisarvude jaoks)	queue<int> s	Queue<Integer> s = new ArrayDeque<Integer> (); Queue<Integer> s = new LinkedList<Integer> ();	s = deque([])
Elemendi 'a' lisamine	s.push(a)	s.add(a)	s.append(a)
Elemendi eemaldamine	s.pop()	a = s.remove(); a = s.poll()	a = s.popleft()
Esimese elemendi vaatamine	a = s.front()	a = s.element(); a = s.peek()	a= s[0]
Kas järjekord on tühi?	s.empty()	s.isEmpty()	not s

Vaatame vahepeal ühe ülesande näitel, kuidas neid andmestruktuure kasutada:

Ühes pisikeses linnakeses elavad lemmingud. Nende linnas käib ringiratast trammiliin, millel on n peatust.
Trammil on ainult üks uks ja tramm ise on kitsas. Seetõttu sisenevad lemmingud trammi ükshaaval, nii et esimesena sisenenud lemming liigub trammi tahaossa, tema järel sisenenud jääb tema ette jne. Väljumine trammist toimub sisenemisele vastupidises järjekorras: kõige viimasena sisenenud lemming väljub esimesena jne.
Igas peatuses on platvorm pikkusega p, millele mahub ootama täpselt p lemmingut. Platvormilt trammi sisenevad lemmingud täpselt ootejärjekorras: kes oli järjekorras enne, siseneb enne.
Liiklemine on korraldatud järgmiselt:
1. Tramm peatub peatuses platvormi lõpus ning lemmingud väljuvad järjest trammist.
2. Kui järgmisena väljuv lemming on jõudnud oma soovitud peatusesse, siis ta lahkub, kui mitte, siis läheb ta platvormil olevasse ootejärjekorda trammile sisenemist ootama.
3. Kui järgmine väljumist ootav lemming ei ole oma peatuses ning platvormil enam ruumi ei ole, siis rohkem lemminguid selles peatuses ei välju, isegi, kui see on nende sihtpeatus.
4. Keegi kellestki mööda ei trügi.
5. Lõpuks sõidab tramm platvormi etteotsa ning võtab järjekorrast peale nii palju lemminguid, kui trammi mahub.
Seejärel sõidab tramm järgmisesse peatusesse ning seal kordub sama protseduur.
Õhtul lahkuvad kõik lemmingid töölt samal ajal ja moodustavad peatustesse järjekorrad. Leia, kui palju aega kulub trammil kõigi lemmingute kodupeatustesse toimetamiseks aega, kui iga sisenemine ja väljumine võtab 1 sekundi, platvormi ühest otsast teise sõit võtab trammil 5 sekundit ja peatuste-vaheline sõit kestab täpselt 1 minuti. Tramm alustab oma teekonda esimesest peatusest.
Sisendi esimesel real on kolm arvu: peatuste arv n ($2 \le n \le 100$), kohtade arv trammis k ($1 \le k \le 100$), platvormi pikkus p ($1 \le p \le 100$). Peatuste platvormid on kõik sama pikkusega.
Järgneval n real on iga peatuse kohta `1...n` seal ootavate lemmingute arv l ($0 \le l \le p$) ja selle järel iga lemmingu l_i ($0 \le l_i \le l$) kohta peatuse number n_j ($1 \le n_j \le n$), kuhu ta sõita soovib. Järjekorras esimene lemmingu peatus on antud esimesena, teise peatus teisena jne.

NÄIDE:

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

Vastus:

```
1220
```

Kui nüüd pinu ja järjekorra mõisted värskelt meeles on, siis on kerge taibata, et trammi sisenemine ja väljumine käib pinu põhimõttel ning peatustes olevad trammisabad on järjekorrad. Muud polegi vaja teha, kui sisendandmed sisse lugeda ja seejärel kogu situatsioon „läbi mängida.“ Iga peatuse jaoks teeme eraldi järjekorra, mida hoiame ühes järjekordade massiivis.

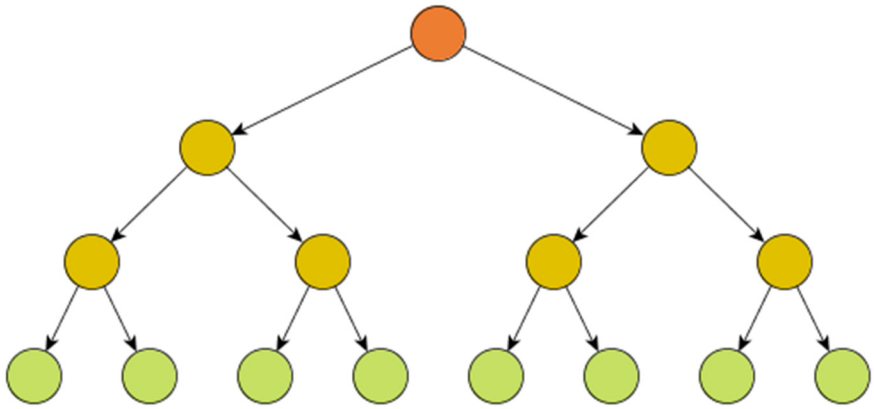
```cpp
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main() {
    int peatuste_arv, kohtade_arv, jk_pikkus;

    //pinu trammis olevate reisijate hoidmiseks, viimasena sisenenu väljub esimesena
    stack<int> tramm;
    //Massiiv trammisabadest. Iga saba on järjekord, kes enne sabas, läheb enne peale
    queue<int> trammisabad[100];

    cin >> peatuste_arv >> kohtade_arv >> jk_pikkus;

    for (int i = 0; i < peatuste_arv; i++) { // iga peatuse kohta
        int ootajate_arv;
        cin >> ootajate_arv; // loeme ootajate arvu peatuses
        for (int j = 0; j < ootajate_arv; j++) { // ja iga ootaja kohta
            int sihtkoht;
            cin >> sihtkoht; // loeme tema sihtkoha
            trammisabad[i].push(sihtkoht - 1); // ning paneme selles peatuses sabasse
        }
    }

    int peatus = 0, kulunud_aeg = 0; // alustame esimesest peatusest
    while (true) {
        // mahalaadimine
        while (!tramm.empty() && // kuni trammis on reisijaid ja
            ((trammisabad[peatus].size() < jk_pikkus) || // peatuses on ruumi või
                tramm.top() == peatus)) { // on järgmine mahamineja jõudnud kohale
            if (tramm.top() != peatus) { // kui järgmine väljuja ei ole oma peatuses
                trammisabad[peatus].push(tramm.top()); // läheb ta selle peatuse sappa
            }
            tramm.pop(); // eemaldame väljuja trammist
            kulunud_aeg++; // lisame väljumiseks kulunud aja
        }

        bool valmis = tramm.empty(); // kas tramm on tühi?
        for (int i = 0; i < peatuste_arv; i++) {
            valmis &= trammisabad[i].empty(); // kas kõigi peatuste sabad on tühjad?
        }
        if (valmis)
            break; // kõik reisijad on kohal
        kulunud_aeg += 5; // liidame peatuses sõitmiseks kulunud aja
        // pealelaadimine
        while ((tramm.size() < kohtade_arv) && // kuni trammis on ruumi
                !trammisabad[peatus].empty()) { // ja peatuses on ootajaid
            tramm.push(trammisabad[peatus].front()); // paneme trammi esimese ootaja
            trammisabad[peatus].pop(); // ja eemaldame ta trammisabast
            kulunud_aeg++; // lisame sisenemiseks kulunud aja
        }

        peatus = (peatus + 1) % peatuste_arv; // järgmine peatus
        kulunud_aeg += 60; // liidame peatuste vahel sõitmiseks kulunud aja
    }

    cout << kulunud_aeg << endl;

    return 0;
}
```

3.10 KAHENDPUU

Puu on dünaamiline andmestruktuur, mille elemente nimetatakse **tippudeks**. Tipud on korraldatud range hierarhiana, see tähendab, et igal tipul võivad olla **alluvad**, aga neil saab olla ainult üks vahetu **ülemus**. **Juurtipp** kõige kõrgema taseme ülemus, sellel ülemust enam pole. Tippu, millel ei ole alluvaid, nimetatakse **leheks,** alluvatega tippu nimetatakse **vahetipuks** ehk **sõlmeks**.

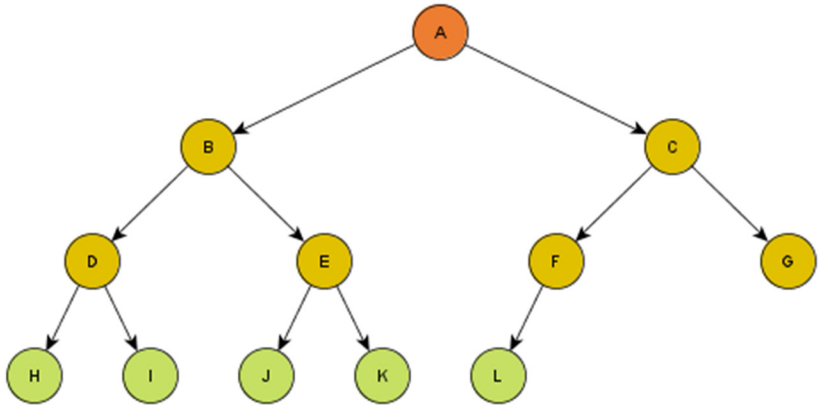
Juurtipp asub esimesel tasemel, kõik tema vahetud alluvad 2. tasemel, nende vahetud alluvad omakorda 3. tasemel jne. Tasemete arvu puus nimetatakse puu **kõrguseks.**

Puud, mille igal sõlmel on kuni n alluvat, nimetatakse n-puuks. Kõige levinum n-puu on **kahendpuu**. Puud nimetatakse **täielikuks**, kui tema igal vahetipul on sama palju alluvaid ning kõik lehed asuvad samal tasemel. n tipuga täieliku kahendpuu kõrgus on $\log_2 n$.



Täielik kahendpuu. Juurtipp on punane, lehed rohelised.

Kui täielikust kahendpuust on viimaselt tasemelt paremalt vasakule eemaldatud 0 või rohkem lehte, siis nimetatakse sellist kahendpuud **kompaktseks**.



Kompaktne kahendpuu.

Kahendpuus on sageli kasulik eristada, kas tipp on oma ülemuse vasak või parem alluv, seda ka juhul, kui ülemusel ongi vaid üks alluv.

Puudel tavaliselt realiseeritavatest operatsioonidest ja nende kasutamises tuleb põhjalikumalt juttu peatükis 6, Graafiteooria.

3.10.1 Kahendpuu esitus

Kahendpuud võib realiseerida nii, et iga tipp on objekt, millel on väljadena viidad tema vasakule ja paremale alluvale.

Sageli on efektiivsem ja lihtsam realiseerida kahendpuu massiivi peale. Puu juur on massiivi esimene element ehk a[0]. Selle vasakpoolne alluv on kohal a[1] ja parempoolne kohal a[2]. a[1] vasakpoolne alluv on kohal a[3] ja parempoolne alluv a[4] jne. Selline esitus sobib hästi kompaktsete kahendpuude hoidmiseks, kuna n-tipuline puu täidab täpselt massiivi pikkusega n.

Eelmisel joonisel kujutatud kompaktse kahendpuu esitus massiivina

Tipp	Asukoht massiivis
Puu juurelement	a[0]
Elemendi a[k] vasak alluv	a[2k+1]
parem alluv	a[2k+2]
ülemus	a[(k-1)/2]

3.10.2 Kahendotsingu puu

Olgu puu igas tipus võtmega kirje. Kahendotsingu puuks (*binary search tree*) nimetatakse sellist puud, mis on tühi või mille vasaku alampuu kõigi tippude võtmed on väiksemad kui juurtipu võti ning parema alampuu tippude võtmed on kõik suuremad kui juurtipu võti ning mõlemad alampuud on kahendotsingu puud.

Kahendotsingu puul kasutatavad operatsioonid on võtme otsimine, uue kirje lisamine ja kirje eemaldamine, samuti vähima ja suurima võtmega kirje leidmine.

Otsimine otsingupuust on lihtne: võrdleme kõigepealt otsitavat väärtust juurtipu võtmega – kui otsitav väärtus on suurem, peab see asuma parempoolses alampuus, kui aga väiksem, siis vasakpoolses. Sama reegel kehtib ka alampuudest otsimisel. Kuna igal sammul liigutakse puus ühe taseme võrra alla, siis sellise otsimise korral on halvima juhu keerukuseks puu kõrgus. Seetõttu on oluline hoida kahendotsingu puu kõrgus võimalikult väike ehk hoida puu tasakaalus.

Tasakaalustatud kahendotsingu puuks nimetatakse sellist kahendotsingu puud, mille iga tipu vasak ja parem alampuu on enam-vähem sama kõrgusega. Kui ühegi tipu alampuude kõrgused ei erine rohkem kui ühe võrra, siis sellist kahendotsingu puud nimetatakse **AVL-puuks.** Nimi tuleb selle puu leiutajate G. Adelson-Velski ja E. Landise nimetähtedest.

Ühtlase kõrguse hoidmine on oluline selleks, et sellises puus on tavapärased operatsioonid teostatavad keerukusega $O(log\ n)$. Loomulikult võib nii tipu lisamine kui ka eemaldamine viia sellise puu tasakaalust välja, seetõttu on oluline puu vajadusel pärast lisamist/eemaldamist tasakaalustada.

3.11 KUJUTIS

Kujutis (*map*) on andmestruktuur, mille elementideks on kaheosalised kirjed – võtme ja väärtuse paarid. Võtmed on kordumatud, st ühe ja sama võtmega elemente võib olla vaid üks. Peamisteks operatsioonideks on kirje lisamine ja eemaldamine ning võtme järgi kirje leidmine.

Kui võtmete võimalik väärtusvahemik on teada ning see ei ole kuigi ulatuslik, saab kasutada massiivi koos **vahetu indekseerimisega** – võtmele v vastava element kirjutatakse massiivi kohale $a[v]$. Puuduvate võtmete kohale kirjutatakse mingi spetsiaalne väärtus, et neid tegelikest võtmetest eristada. Sellisel moel on nii lisamine, eemaldamine kui ka võtme järgi elemendi leidmine keerukusega $O(1)$.

Kui aga võtmeid on rohkem, kasutatakse kujutise realiseerimiseks **paisktabelit** *(hash table)*. Paisktabel on samuti massiiv, kuid elementi võtmega v ei kirjutata otse kohale a[v], vaid kasutatakse **paiskfunktsiooni**, mis seab igale võtmele vastavusse mingi indeksi. See tähendab, et võtmele v vastav kirje kirjutatakse paisktabelisse kohale a[$f(v)$], kus f on paiskfunktsioon.

Paiskfunktsioon peab olema kiiresti arvutatav. Harilikult valitakse paiskfunktsiooniks $f(v) = v \bmod m$, kus m on paisktabeli ridade arv.

Paisktabeli realiseerimisel võetakse tavaliselt mugavalt palju ruumi, et andmed paisktabelisse vabalt ära mahuvad. Sellegipoolest võib paisktabelis esineda **põrkeid** ehk **kollisioone**. Põrge on selline olukord, kus paiskfunktsioon f annab kahe erineva võtme v_1 ja v_2 jaoks sama väärtuse, st $f(v_1) = f(v_2)$, kui $v_1 \neq v_1$.. Sisuliselt tähendab see, et kaks kirjet peaks asuma paisktabelis samal kohal.

Kokkupõrgete lahendamiseks kasutatakse peamiselt kahte erinevat võimalust. **Välisahelate** meetodi korral on igal real kaks lisavälja: üks selle jaoks, et märkida, kas rida on juba hõivatud ja teine on viit lihtahelale. Kui objekti võtmega v salvestatakse reale $f(v)$, kuid see rida on juba hõivatud, salvestatakse uus objekt realt $f(v)$ algavasse lihtahelasse.

Lahtise adresseerimise meetodi korral lahendatakse põrge järgmiselt: kui võtmele v vastav rida $f(v)$ on juba hõivatud, otsitakse eelmine vaba rida ning lisatav kirje salvestatakse sinna. Lugemisel tuleb siis samuti otsida võtit igalt eelnevalt realt, kuni võti leitakse või jõutakse tühja reani, mis tähendab, et antud võtit tabelis ei esine. Veelgi parem on kasutada topeltpaiskamisega lahtist adresseerimist, mille korral ei otsita vaba kohta järjest, vaid kasutatakse mingit teist funktsiooni $f_2(v)$, mis tagastab sammu pikkuse, mitu kohta tuleb esialgsest asukohast edasi minna, et leida võtmele vastav uus asukoht. Sellise meetodiga leitakse kirje otsimisel üldiselt kiiremini üles.

3.11.1 Kujutis programmeerimiskeeltes

C++ pakub standardteegis kahte klassi `map` ja `unordered_map`. Mõlemad on mõeldud võti-väärtus paaride hoidmiseks, kuid `map` on tavaliselt realiseeritud kahendotsingupuuna, mistõttu on sealt elemendi leidmine aeglasem operatsioon, see-eest võimaldab mugavamat elementide läbikäimist (elemendid on võtme järgi sorteeritud). `unordered_map` on enamasti realiseeritud paisktabelina ja võimaldab võtme järgi ligipääsu konstantse keerukusega.

Java valik pole halvem: seal on klassid `TreeMap` ja `HashMap` ning nendel liides `Map`.

Pythonis on olemas andmetüüp `dict` (*dictionary*) võti-väärtus paaride hoidmiseks. Realiseeritud on see topeltpaiskamisega lahtise adresseerimisega paisktabelina.

3.12 PRAKTILINE ÜLESANNE

Usina linna avatud ülikoolis on igaühel võimalik käia kuulamas erinevaid kursusi. Usinas linnas elavad muidugi usinad õppijad, kes enamasti osalevad aastas viiel kursusel. Seetõttu on ülikoolil plaanis pakkuda 5-kursuselisi valmiskomplekte. Esialgu otsustatakse katsetada kombinatsiooni kõige populaarsematest kursustest, selleks aga on vaja teada, millise viie kursuse kombinatsioon esineb kõige sagedamini. Erinevaid kursusi on palju ja need on nummerdatud arvudega 100…499.
Sisendi esimesel real on arv n ($1 \le n \le 10000$) mis näitab, kui palju on olnud eelneval aastal viie kursuse valijaid, ning järgneval n real igaühel ühe õppija valitud kursuste numbrid valimise järjekorras, tühikutega eraldatuna. Väljastada kõige sagedasem kursuse kombinatsioon. Kui sama sagedusega on mitu kombinatsiooni, väljastada kõik erinevad suurima sagedusega kombinatsioonid, iga kombinatsioon eraldi real.

NÄIDE:

```
5
101 102 103 104 105
217 301 101 105 400
101 301 400 217 105
101 301 103 104 105
105 104 103 102 101
```

Vastus

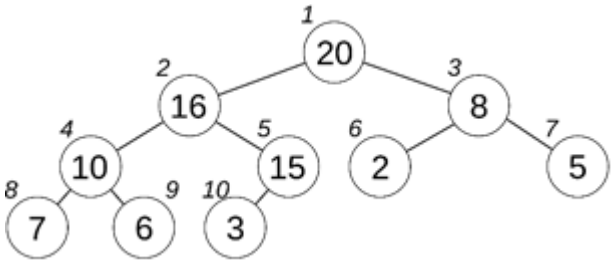
```
101 102 103 104 105
101 105 217 301 400
```

Siin on peamiseks küsimuseks, kuidas leida korduvad kombinatsioonid ja neid loendada. Erinevaid võimalikke kombinatsioone on $\binom{400}{5}$, mis on ligikaudu 83 miljardit. See ei aita meid kuidagi edasi. Üheks võimaluseks oleks välja mõelda funktsioon, mis igale kombinatsioonile seab vastavusse ühe arvu – nii saaks leida iga kombinatsiooni jaoks sisendis sellele vastava arvu ning loendada, kui mitu korda mingit arvu esineb. Antud juhul on loomulik meetod kasutada kursusenumbreid stringidena ja need kokku kleepida üheks pikaks stringiks. Kui enne kursused mingis järjekorras sorteerida, siis samale kombinatsioonile vastab sama string (ja vastupidi – ühele stringile vastab üks kindel kombinatsioon) ja seda stringi saame kasutada võtmena:

```
#include <map>
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    map<string, int> logi;

    // siin hoiame seni leitud suurimat esinemissagedust. Kuna ülesande tingimustes on
    //vähemalt 1 kombinatsioon olemas, võib selle panna üheks.
    int maxN = 1;
    string kursused[5]; //siin hoiame ühe inimese võetud kursusi
```

```
    for (int i = 0; i < n; i++) {
        cin>> kursused[0] >> kursused[1] >> kursused[2] >> kursused[3] >> kursused[4];
        sort(kursused, kursused + 5); // sorteerime kursused kasvavas järjekorras

        string voti;
        for (int j = 0; j < 5; j++) {
            voti += kursused[j]; // ühendame kursuste numbrid üheks stringiks
        }
        // kui sellist võtit pole (sellist kombinatsiooni ei esine)
        if (!logi.count(voti))
{           logi[voti] = 1; // lisame selle väärtusega 1.
        }
        else {  // kui võti on
            int m = logi[voti] + 1; // suurendame kombinatsiooni korduste arvu
            logi[voti] = m; // ja muudame vastavaks
            // kui on suurem korduste arv, kui senine maksimum, muudame maksimumi.
            maxN = max(maxN, m);
        }
    }

    map<string, int>::iterator it;
    for (it = logi.begin(); it != logi.end(); it++) { // käime mapi läbi
        if (it->second == maxN) // kui on suurim korduste arv
            for (int i = 0; i < 13; i += 3) {
                // eraldame võtmest kursuste numbrid ja väljastame need
                cout << it->first.substr(i, 3) << " ";
            }
        cout << endl;
    }

    return 0;
}
```

3.13 KUHI



Kuhi (*heap*) on selline puu, millel kehtib **kuhjaomadus**: ühegi tipu võti ei ole suurem kui tema ülemuse võti. Kui kehtib vastupidine omadus – ühegi tipu võti ei ole väiksem kui tema ülemuse võti – siis on tegemist **pöördkuhjaga**.

Kahendkuhi on kuhi, mis on ühtlasi ka kompaktne kahendpuu. Sageli mõeldaksegi kuhja all justnimelt kahendkuhja ning siingi peatükis vaatame edasi vaid kahendkuhja.

Kuna kuhi on kompaktne kahendpuu, siis sobib selle esituseks väga hästi massiiv.

Kuhjal on enamasti realiseeritud järgmised operatsioonid:

- Suurima (või vähima) elemendi (võtme)väärtuse leidmine, keerukus $\Theta(1)$;
- Lisamine, keerukus $O(log(n))$;
- Suurima (vähima) elemendi kustutamine, keerukus $\Theta(log(n))$
- Kuhjastamine ehk etteantud elementide kuhjaks paigutamine;

Operatsioone kuhjal võib realiseerida mitmel moel, lisamisel ja eemaldamisel tuleb aga hoolega silmas pidada, et kuhja omadus säiliks. Selleks on kuhjal enamasti meetodid elementide nihutamiseks üles- ja allapoole. Lisamine on harilikult realiseeritud nii, et uus element lisatakse kuhja lõppu ning

nihutatakse seda kuhjas ülespoole, kuni kuhja omadus on taastatud. Eemaldamisel asendatakse juurelement viimase elemendiga kuhjas, seejärel kukutatakse see element kuhjas õigesse kohta ja kuhjaomadus saab taastatud.

Kuhjasid kasutatakse:

- kuhjameetodil sorteerimiseks ($O(n \log n)$),
- valikualgorimide (*selection algorithms*) realiseerimiseks, kuna miinimum ja maksimumelemendi leidmine on kuhjas konstantse ajaga ning mediaani ja k. elemendi leidmine toimuvad vähem kui lineaarse ajaga,
- mitmete graafitöötlusalgoritmide realiseerimiseks,
- eelistusjärjekorra realiseerimiseks.

C++ keeles on päisfailis `<queue>` klass `priority_queue`. Selle esimene, eemaldatav element on alati suurim element. Kuna kuhja on mugav hoida massiivis, siis `priority_queue` kasutab vaikimisi `vector` klassi tegeliku konteinerina ning rakendab sellel operatsioone kuhjastruktuuri säilitamiseks.

Javas on klass `PriorityQueue`, mis on oma olemuselt pöördkuhi, nii et Javas võetakse eelistusjärjekorrast järgmisena kõige väiksem element.

Pythonis on hea kasutada `Lib/heapq.py`. Nagu Javagi, kasutab Python pöördkuhja ehk esimene element on kõige väiksem.

Järgmine tabel illustreerib kuhja kasutamist erinevates programmeerimiskeeltes:

Tegevus	C++	Java	Python
Järjekorra s loomine	priority_queue<int> e	PriorityQueue<Integer> e = new PriorityQueue<Integer> ();	e = []; heapify(e)
Elemendi 'a' lisamine	e.push(a)	e.add(a)	heappush(e, a)
Elemendi eemaldamine	e.pop()	a = e.remove(); a = e.poll()	a = heappop(e)
Esimese elemendi vaatamine	a = e.top()	a = e.element(); a = e.peek()	a= e[0]
Kas järjekord on tühi?	e.empty()	e.isEmpty()	not e

3.14 SORTIMINE

Sortimine on etteantud hulga elementide mingisse kindlasse järjestusse seadmine. Enamasti kasutatakse numbrilist või leksikograafilist järjestust. Elemendid antakse harilikult ette massiivina ning sortimise tulemuseks on massiiv, kus

- ükski eelnev element ei ole suurem järgnevast,
- väljund on sisendi permutatsioon.

Erinevaid sortimisalgoritme on päris palju. Sortimisalgoritmi nimetatakse **stabiilseks**, kui võrdsete elementide korral säilib nende omavaheline järjestus algses jadas. Näiteks, kui on antud mängukaardid järjestuses ♣7 ♥2 ♥7 ♣8, siis numbrite järgi sorteerides on korrektsed nii ♥2 ♣7 ♥7 ♣8 kui ka ♥2 ♥7 ♣7 ♣8, kuid ainult esimeses on säilinud seitsmete omavaheline algne järjestus: risti seitse on eespool ärtu seitsmest.

Öeldakse, et sortimisalgoritm töötab **kohapeal** *(in-place)*, kui algoritm ei vaja algandmete sorteerimiseks rohkem kui konstantset hulka lisamälu.

Sõnastame sorteerimisülesande ja vaatame selle erinevaid lahendusi:

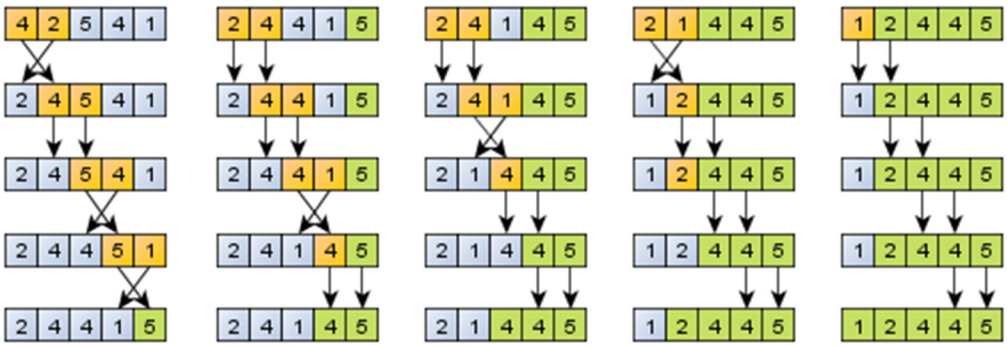
On antud n-elemendiline täisarvudest koosnev jada. Paiguta elemendid ümber nii, et $a_1 \leq a_2 \leq \cdots \leq a_n$.

3.14.1 Mullimeetod

Mullimeetod (*bubble sort*) on üks vanemaid ja tuntumaid sortimisalgoritme. Algoritm on lihtsasti mõistetav ning ka väga lihtne programmeerida, mistõttu tutvustatakse seda sageli programmeerimise algkursustel:

```
int mullimeetod(int* jada, int pikkus)
{
    bool vahetus = true;
    while (vahetus) { //kui on olnud vahetusi
        vahetus = false; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                vahetus = true; //ja peame meeles, et vahetus toimus
            }
        }
    }
}
```

Järgnev skeem kujutab selle algoritmi tööd:



Skeemilt on lihtne märgata, et esimesel läbikäimisel satub kõige suurem element jada lõppu, teisel läbikäimisel pannakse eelviimane element oma kohale jne, mis tähendab, et tegelikult ei ole vaja igal sammul kogu jada läbi käia. Veelgi enam – pole raske veenduda, et kõik elemendid, mis asuvad viimasest vahetusest tagapool, on juba sorteeritud. Siin on optimeeritud lahendus:

```
int optmullimeetod(int* jada, int pikkus)
{
    int viimane = pikkus;
    while (viimane > 0) { //kui on olnud vahetusi
        viimane = 0; //veel ei ole vahetusi olnud
        for (int i = 1; i < pikkus; i++) {
            if (jada[i - 1] > jada[i]) { //kui eelnev on suurem järgmisest
                int temp = jada[i - 1]; //vahetame elemendid
                jada[i - 1] = jada[i];
                jada[i] = temp;
                viimane = i; //ja peame meeles viimase vahetuse koha
            }
        }
        pikkus = viimane;
    }
}
```

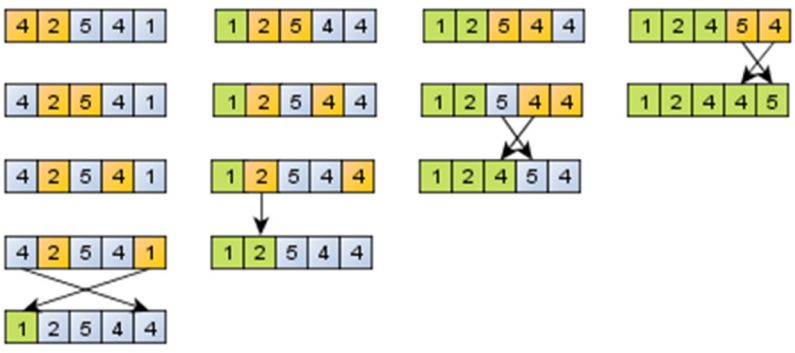
Mullimeetodi keerukus on $O(n^2)$ ning isegi teiste sama keerukusklassi sortimisalgoritmidega võrreldes (nt pistemeetod) on mullimeetod üldjuhul aeglasem, kuna keskmiselt tehakse rohkem elementide vahetusi.

3.14.2 Valikmeetod

Valikmeetod (*selection sort*) on samuti väga lihtne sortimisalgoritm: kõigepealt leitakse jadas kõige väiksem element ning vahetatakse see kõige esimese elemendiga. Seega pärast vahetust on kõige väiksem element oma kohal ning protseduuri korratakse ülejäänud jada elementidel, kuni kõik elemendid on õige koha leidnud:

```
void valikmeetod(int* jada, int pikkus)
{
    for (int j = 0; j < pikkus - 1; j++) { //üksik element on vähim, piisab
    // eelviimaseni vaatamisest
        int min = j; //paneme esimese õige asukohata elemendi minimaalseks
        for (int i = j + 1; i < pikkus; i++) { //käime jada lõpuni läbi
            if (jada[i] < jada[min]) { //leidime väiksema elemendi
                min = i; //jätame asukoha meelde
            }
        }
        if (min != j) { //kui vähim element ei ole juba õigel kohal
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[min];
            jada[min] = temp;
        }
    }
}
```

Siin on skeem, mis kujutab selle algoritmi tööd:



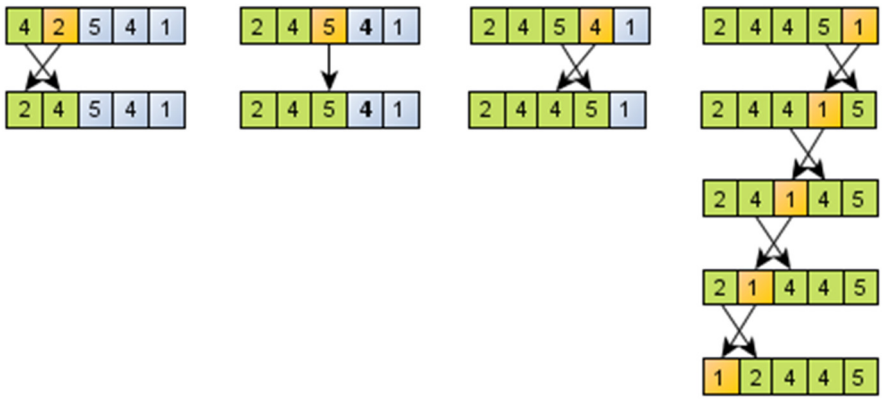
Valikmeetodi keerukus on samuti on $O(n^2)$. Samas on skeemilt kohe näha, et tegelikke vahetusi tehakse oluliselt vähem kui mullimeetodit kasutades – vahetuste arv ei ole kunagi suurem kui elementide arv.

3.14.3 Pistemeetod

Kolmas $O(n^2)$ keerukusega sortimisalgoritm on pistemeetod (*insertion sort*). Pistemeetod on sageli kasutuses väiksematel andmemahtudel ning on keskmiselt kiirem kui mulli- või valikmeetod ning sedagi on lihtne implementeerida:

```
void pistemeetod(int* jada, int pikkus)
{
    for (int i = 1; i < pikkus; i++) { //vaatame iga elementi alates teisest
    //kuni vaadeldava elemendini jada[i] on jada sorteeritud. Otsime õige koha jadas
        for (int j = i; j > 0 && jada[j - 1] > jada[j]; j--) {
            int temp = jada[j]; //vahetame elemendid
            jada[j] = jada[j - 1];
            jada[j - 1] = temp;
        }
    }
}
```

Pistemeetod on kujutatud järgmisel skeemil:



Pistemeetodi tegelik keerukus oleneb algandmetest. Nagu mainitud, siis halvimal juhul – kui jada on kahanevas järjestuses – tuleb teha n^2 võrdlust ja vahetust, samas, kui andmed on juba õiges järjekorras, tehakse vaid n võrdlust. Väikeste jadade korral ($n < 20$) on pistemeetod kiirem kui keerulisemad sortimisalgoritmid ja seda kombineeritakse sageli kiirmeetodiga.

3.14.4 Põimemeetod

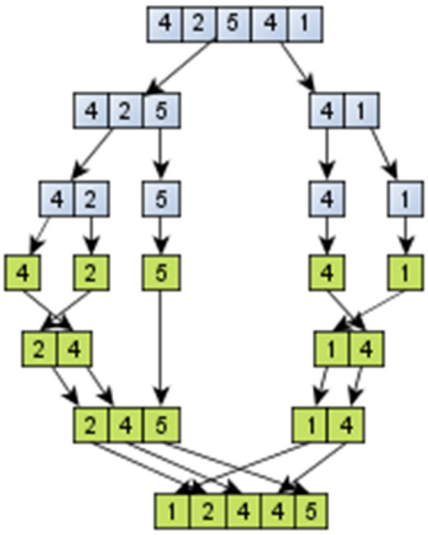
Põimemeetod (*merge sort*) on juba mõnevõrra keerulisem sortimisalgoritm. Selle mõtles 1945. aastal välja arvutiteaduse üks „isasid“ John von Neumann. Tegemist on „jaga ja valitse“ tüüpi algoritmiga, kus jada jagatakse pooleks ja sorteeritakse kumbki pool eraldi, hiljem sorteeritud pooled põimitakse. Loomulikult ei piirduta vaid ühekordse poolitamise ja põimimisega, vaid seda tehakse seni, kuni alamjadad on soovitud pikkusega.

Algoritmi põhiosa on lihtne:

```
void poimemeetod(int* jada, int v, int p)
{
    if (v < p) {
        int keskkoht = (v + p) / 2;
        poimemeetod(jada, v, keskkoht);
        poimemeetod(jada, keskkoht + 1, p);
        poimi(jada, v, keskkoht, p);
    }
}
```

Põimimise osa nõuab rohkem näputööd:

```
void poimi(int* jada, int v, int keskkoht, int p)
{
    int i, j, k;
    int vpikkus = keskkoht - v + 1; // esimese lõigu pikkus
    int ppikkus = p - keskkoht; // teise lõigu pikkus

    int *V = new int[vpikkus]; // põimimiseks teeme vasaku ja
    int *P = new int[ppikkus]; // parema lõigu jaoks eraldi massiivid

    for (i = 0; i < vpikkus; i++)
        V[i] = jada[v + i];
    for (j = 0; j < ppikkus; j++)
        P[j] = jada[keskkoht + 1 + j];

    i = 0; // vasakpoolse jada esimese elemendi indeks
    j = 0; // parempoolse jada esimese elemendi indeks
    k = v; // põimitava koha algusindeks jadas
    while (i < vpikkus && j < ppikkus) { // kuni mõlemas jadas on vaatamata elemente
        if (V[i] <= P[j]) { // vasakul on mittesuurem paigutamata element
            jada[k++] = V[i++]; // paneme selle jadasse
        }
        else { // paremal on väiksem element
            jada[k++] = P[j++]; // paigutame selle jadasse
        }
    }

    while (i < vpikkus) {//kui vasakus jadas on veel elemente
        jada[k++] = V[i++]; //kopeerime need järjest jadasse
    }

    while (j < ppikkus) { // kui paremas jadas on veel elemente
        jada[k++] = P[j++]; // kopeerime need järjest jadasse
    }
}
```



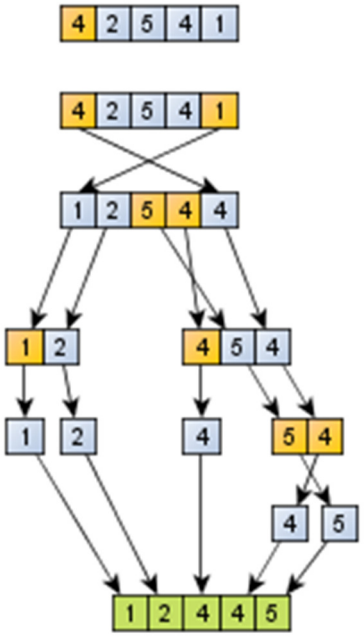
Põimemeetodi keerukusklass (ka halvimal juhul) on $O(n \log n)$ ja see sobib eelkõige suuremate jadade sorteerimiseks. Põimemeetodi miinuseks on, et see meetod nõuab lisamälu mahus $\Omega(n)$. Eksisteerib ka selline variant põimimisest, mis teeb seda kohapeal, ilma märkimisväärselt lisamälu kasutamata, kuid see algoritm on juba päris keeruline. Huvilistele: https://github.com/liuxinyu95/AlgoXY/blob/algoxy/sorting/merge-sort/src/mergesort.c
Põimemeetod sobib väga hästi ahelate sorteerimiseks. Sellisel juhul puudub ka vajadus lisamälu jaoks.

3.14.5 Kiirmeetod

Praegu tuntud algoritmidest on üldjuhul kõige kiirem meetod suurte jadade sortimiseks kiirmeetod (*quicksort*). Ka kiirmeetod on „jaga ja valitse“ tüüpi algoritm. Algoritmi idee on järgmine: sorteerimata jadast võetakse üks element, nn veelahe (*pivot*) ning võrreldakse teisi elemente sellega. Kõik elemendid, mis on veelahkmest väiksemad või võrdsed sellega, paigutatakse valitud elemendist vasakule ehk ettepoole, need aga, mis on suuremad, paigutatakse paremale ehk tahapoole. Selle tulemusena on veelahkmest vasakul ainult sellest mittesuuremad elemendid ja paremal ainult suuremad elemendid. Seejärel sooritatakse sama protseduur valitud elemendist vasakul ja paremal pool eraldi. Kui vasakul või paremal on vähem kui kaks elementi, on see osa juba sorteeritud ja rohkem ei olegi vaja midagi teha.

```
int jaota(int* jada, int v, int p)
{
    int e = jada[v];
    int i = v - 1;
    int j = p + 1;
    while (true) {
        while (jada[++i] < e) {}
        while (jada[--j] > e) {}
        if (i >= j) return j;
        int temp = jada[i];
        jada[i] = jada[j];
        jada[j] = temp;
    }
}
```

```
void kiirmeetod(int* jada, int v, int p)
{
    if (v < p) {
        int e = jaota(jada, v, p);
        kiirmeetod(jada, v, e);
        kiirmeetod(jada, e + 1, p);
    }
}
```



Kiirmeetodi keskmine keerukus on $O(nlog\ n)$. Halvimal juhul on kiirmeetodi keerukus küll $O(n^2)$, kuid praktikas on tegemist siiski kõige kiirema sorteerimisalgoritmiga. Sageli kombineeritakse kiirmeetodit pistemeetodiga – kui sorteeritava alamjada pikkus on väike, näiteks kuni 20 elementi, sorteeritakse see pistemeetodil, vastasel juhul vastavalt kiirmeetodi algoritmile.

Kiirmeetodi probleemseks kohaks võib osutuda selle meetodi ebastabiilsus.

3.14.6 Võrdlustel põhineva sortimise keerukuse alampiir

Eelnevalt vaadeldud sortimisalgoritmid põhinesid kõik elementide võrdlemisel. Selliste algoritmide keerukuse alampiir on $O(n \log n)$. Miks?

Iga võrdlustehe annab meile informatsiooni jada järjestuse kohta. Kokku on n-elemendilisel jadal $n!$ erinevat võimalikku järjestust. Saame konstrueerida otsustuste puu, kus lehtedeks on kõik võimalikud n permutatsioonid ja sõlmedeks võrdlustehted.

Võrdlustehtega sõlmel on

- vasakuks alluvaks on võrdlustehe, mille algoritm sooritab siis, kui võrdluse tulemus on negatiivne,
- paremaks alluvaks on võrdlustehe, mis sooritatakse siis, kui tulemus oli positiivne.

Iga permutatsioonini viib täpselt üks otsustuste tee. Kõige lühem selline tee ehk lehe vähim võimalik kaugus puu juurest vastab parimale juhule – kõige pikem tee – ehk puu kõrgus vastab halvimale võimalikule juhule. $n!$ lehega kahendpuu minimaalseks kõrguseks on $\log_2(n!) = \log_2 1 + \log_2 2$

+…+ $\log_2 n$. Kuna hetkel huvitab meid ainult alampiir, siis pole vaja leida täpset tulemust. Selle summa viimased $n/2$ liiget $\log_2 \frac{n}{2} \ldots \log_2 n$ ei ole ükski suurem kui $\log_2 \frac{n}{2}$. Seega

$$\log_2(n!) \geq \frac{n}{2}\log_2 \frac{n}{2} = \frac{n}{2}\log_2 n - \frac{n}{2}.$$

Järelikult on sellise puu minimaalne kõrgus $O(n \log n)$ ja seega ei saa võrdlemistel põhineva sorteerimisalgoritmi keerukus halvimal juhul olla parem kui $\Omega(n \log n)$.

3.15 SORTIMISE ERIMEETODID

Kui sisendandmed vastavad teatud täiendavatele piirangutele, siis mõningatel juhtudel on võimalik kasutada ka efektiivsemaid sortimisalgoritme, mille keerukusklass on $O(n)$. Tuntumad neist on loendamismeetod (*count sort*), positsioonimeetod (*radix sort*) ja kimbumeetod (*bucket sort*). Kõik need meetodid kasutavad lisamälu mahus $\Omega(n)$.

3.15.1 Loendamismeetod

Loendamismeetodit saab kasutada, kui andmed on teadaolevas ja küllaltki kitsas vahemikus. Loendamismeetodi idee seisneb selles, et luuakse eraldi loendusmassiiv, kus igale indeksile vastab üks elemendi võimalik väärtus. Kui suurim jadas esinev väärtus on max ja vähim min, siis on loodava jada pikkus $max - min + 1$. Seejärel käiakse esialgne jada elementhaaval läbi ning suurendatakse loendusmassiivis elementi kohal a_{i-min}. Sellisel moel loendatakse iga element.

0	1	2	3	4	indeks
1	1	0	2	1	
1	**2**	**3**	**4**	**5**	**väärtus, mida loendame**

Massiivile 4,5,2,4,1 vastav loendusmassiiv

Kui loendasime arve, pole edasi vaja teha muud, kui vastavalt loendusmassiivile kirjutada igale indeksile vastavat väärtust esialgsesse massiivi nii mitu korda, kui seda esines. Kui tegemist on aga objektidega, siis saab arvutada, mitmendal kohal peaks sorteeritud massiivis olema antud võtmeväärtusega element. Selleks arvutatakse loendusmassiivis iga elemendi m[i] jaoks alates teisest m[i] + m[i-1], s.t leitakse selle võtmega elemendi maksimaalne positsioon sorteeritud jadas. Loendusmassiiv on siis kujul [1, 2, 2, 4, 5].

Käime esialgse jada tagurpidi läbi ja iga elemendi jaoks leiame selle positsiooni sorteeritud jadas: j = m[a[i]- v] – 1. Kirjutamegi elemendi sellele positsioonile uude vastusjadasse ning vähendame loendusmassiivis selle elemendi võtmele vastavat väärtust. Näiteks, pannud viimase elemendi (1) vastusjadasse `[1,x,x,x,x]`, vähendame loendusmassiivis väärtust kohal 1-1 ja saame `[0,2,2,4,5]`. Järgmiseks paigutame massiivi eelviimase elemendi 4, ning saame vastusmassiivi `[1,x,x,4,x]` ning loendusmassiivi `[0,2,2,3,5]` – järgmine „4“, mis algses jadas ette tuleb, paigutatakse nüüd kohale 3 - 1 = 2. Sellisel moel elemente paigutades on tagatud ka stabiilsus.

See meetod vajab lisamälu vähemalt loendusmassiivi pikkuse k jagu. Objektide korral on lisaks sellele vaja veel ühe n-elemendilise jada jagu mälu sorteeritud elementide hoidmiseks, mis teeb kokku lisamälu vajaduseks $\Omega(n + k)$. Ka loendusmassiiv tuleb korra läbi käia, mistõttu omab selle algoritmi kasutamine mõtet siis, kui k on väike.

3.15.2 Positsioonimeetod

Positsioonimeetodiga sorteeritakse harilikult positiivseid, mitte väga pikki täisarve. Meetod seisneb selles, et arve ei vaadelda arvudena, vaid numbritest koosnevate liitvõtmetena. Seega kui näiteks kolmekohalisi arve sorteerides sorteerime need esmalt üheliste järgi, seejärel kümneliste järgi ja siis sajaliste järgi, saamegi tulemuseks sorteeritud jada.

Alamalgoritmina kasutatakse sorteerimiseks loendamismeetodit, seetõttu peavad võtmed olema kõik sama pikkusega. Kasutus ei piirdu siiski ainult arvudega, sel moel saab sorteerida ka sõnu.

3.15.3 Kimbumeetod

Kimbumeetod seisneb selles, et väärtuste järgi jaotatakse elemendid eraldi kimpudesse või ämbritesse, mida siis sorteeritakse edasi kas kimbumeetodil või mõnel muul viisil. Näiteks sorteerides jada, kus on arvud 0…99, eraldame ühte kimpu kõik arvud 0..9, teise 10…19 jne. Sorteerides need kimbud eraldi, saame tulemustest kokku panna sorteeritud jada.

3.16 MITME KRITEERIUMI JÄRGI SORTIMINE

Mõnikord võib olla vaja sorteerida mitme kriteeriumi järgi, näiteks kaarte masti ja väärtuse järgi. Üheks võimaluseks on kasutada mõnda stabiilset sortimismeetodit ning sortida iga kriteeriumi järgi eraldi, alustades kõige vähemtähtsamast. Kui kriteeriume on k, tuleb jada sorteerida k korda ehk sellise lähenemise keerukuseks on $O(kn \log n)$.

Tavalisemaks lahenduseks on koostada oma spetsiaalne võrdlemisfunktsioon, mis suudab objektid etteantud järjestuses reastada, võrreldes kõigepealt kõige olulisemat kriteeriumi ja kui selle järgi on objektid võrdsed, siis tähtsuselt järgmist jne. Halvimal juhul tuleb iga võrdlusoperatsiooni korrata k korda – iga kriteeriumi jaoks – ja seegi lähenemine annab halvimal juhul keerukuseks $O(kn \log n)$. Päriselt töötab see lahendus aga kiiremini kui eelmine, sest enamasti ei ole vaja siiski võrrelda kõiki kriteeriume ning teiseks päästab see lähenemine vajadusest kasutada stabiilset sortimisalgoritmi.

3.17 PROGRAMMEERIMISKEELTE STANDARDTEEGID

Enamasti pole siiski vaja jalgratast leiutada, kuna mõistlik üldine sorteerimine on keelte standardteekides olemas. Eriti oluline on see võistlustel, kus lahendamisaeg on piiratud ning oma sortimisalgoritmi kirjutamine ja testimine on liigne ajakulu. Seetõttu peaks iga võistleja end kurssi viima oma keele pakutavate võimalustega.

Keele spetsifikatsioon määrab enamasti ära, mis tingimused sortimismeetodile kehtima peavad. Praegu nõuavad nii C++, Java kui ka Python, et sortimisfunktsiooni halvima juhu keerukus on $O(n \log n)$. Java ja Pythoni sortimisfunktsioonid on stabiilsed, C++ seda nõuet ei esita, rõhutakse pigem nn kohapeal sortimisele. Millist algoritmi sortimiseks konkreetselt kasutatakse, sõltub juba konkreetselt kasutatavast teegist. Näiteks GNU C++ standardteek kasutab *introsort* algoritmi kombineerituna pistemeetodiga. *Introsort* on kiirmeetodi ja kuhjameetodi hübriid. Pythoniga tuli kasutusse *Timsort* algoritm, mis on tuletatud põime- ja pistemeetodist. Ka uuemad Java versioonid kasutavad *Timsort*i objektide sortimiseks, arvutüüpidel kasutatakse kahe lahkmega kiirmeetodi varianti.

3.17.1 C++

C++ on päisfailis <algorithm> defineeritud funktsioon sort, millele antakse ette mingi järjendi algus ja lõpp. Näiteks massiivi korral:

```
int n = 5
array size int t[] = {4,2,5,3,5}
sort(t, t+n);
```

vektori (`vector`) korral:

```
sort(v.begin(), v.end());
```

Vaikimisi sorteeritakse elemendid kasvavas järjekorras ja võrdlusoperaatori järgi. Paaride (`pair`) ja ennikute (`tuple`) korral sorteeritakse need vaikimisi esimese elemendi järgi, kui esimesed elemendid on võrdsed, siis teise järgi jne. Enda defineeritud struktuuridel ja objektidel tuleb ise defineerida ka võrdlusoperaator '<'. Samuti saab sort funktsioonile parameetrina ette anda võrdlusfunktsiooni, mis saab ette kaks argumenti ja tagastab tõeväärtuse: tõese, kui esimene argument on väiksem kui teine ja väära muul juhul:

```
bool cmp(string a, string b)
{
    if (a.size() != b.size())
        return a.size() < b.size();
    return a < b;
}
sort(v.begin(), v.end(), cmp);
```

On väga oluline meeles pidada, et sort ei ole stabiilne. Kui on vaja tagada stabiilsus, tuleb kasutada funktsiooni stable_sort, mis enamasti põhineb põimemeetodil.

3.17.2 Java

Javas on päises `Java.Util.Arrays` meetod `Arrays.sort`, millele antakse ette jada elementidest. Kui on vaja sorteerida lõiku jadast, saab ette anda ka alguse ja lõpu indeksid. Tasub tähele panna, et algus kuulub lõiku, lõpp aga mitte.

3.17.3 Python

Pythonis saab sortimiseks kasutada kaht peamist viisi:

- funktsioon `sorted` saab ette järjendi ja tagastab teise, sorteeritud järjendi,
- järjendil endal on meetod `sort`.

Vaikimisi sorteeritakse elemendid kasvavas järjekorras:

```
>>> sorted([4,5,2,4,1])
[1, 2, 4, 4, 5]
```

Mõlemale funktsioonile saab ette anda lipu reverse, mille tõese väärtuse korral jada sorteeritakse kahanevalt.

Lisaks on neil funktsioonidel ka parameeter `key`, mis määrab, mille järgi tegelikult sorteeritakse. `key` on üheargumendiline funktsioon, mille argumendiks on element ja tagastatavaks väärtuseks selle elemendi võti, mida kasutatakse sorteerimiseks.

Kindlasti tasub tutvuda ka mooduliga operator, kus on funktsioonid itemgetter, attrgetter ja methodcaller, mis teevad võtme etteandmise oluliselt mugavamaks ja võimaldavad ka mitme parameetri järgi sorteerimist:

```
>>> from operator import itemgetter
>>> opilased = [('Andres', 14 , 8),('Kati', 10, 3),('Mati', 9, 3)]

>>> sorted(opilased, key=itemgetter(2, 0, 1))
[('Kati', 10, 3), ('Mati', 9, 3), ('Andres', 14, 8)]
```

Muidugi võib võtme leidmise funktsiooni alati ise kirjutada. Võtme leidmise funktsioon ei pea olema rangelt objekti juures, vaid saab kasutada ka muid funktsioone:

```
>>> opilased = ['Andres', 'Kati', 'Mati']
>>> hinded = {'Mati': '3', 'Kati':'4', 'Andres': '5'}
>>> sorted(opilased, key=hinded.__getitem__, reverse=True)
['Andres', 'Kati', 'Mati']
```

Python3 versioonides kasutab sort objektide __lt__() funktsiooni. Kui oma objektil see meetod ise defineerida, saab määrata ka objektide sorteerimise järjestust. Python2 kasutades on soovitatav defineerida kõik võrdlusfunktsioonid.

Üks näiteülesanne:

Pärtlil on n mängukaarti. Ta soovib sorteerida kaardid nii, et kõige peal on potid, siis ärtud, seejärel ruutud ning kõige lõpuks ristid. Lisaks soovib ta, et iga mast on sorteeritud nii, et kõige peal on äss, seejärel kuningas, emand, soldat ja siis numbrid kahanevas järjekorras (10...2). Sisendi esimesel real on kaartide arv n(1≤n≤1000000). Järgmisel real on n kahemärgilist stringi, millest igaüks vastab mingile kaardile kaardipakist. Esimene tähemärk tähistab masti (C – risti, D – ruutu, H – ärtu ja S – poti), teine märk väärtust (2...9, T – 10, J – soldat, Q – emand, K – kuningas ja A – äss). Väljundiks on sorteeritud kaardipakk, nii et esimene on pealmine kaart, 2. pealmisest järgmine kaart jne.

NÄIDE:

```
5
C8 D8 SK DQ SK
```

Vastus:

```
SK SK DQ D8 C8
```

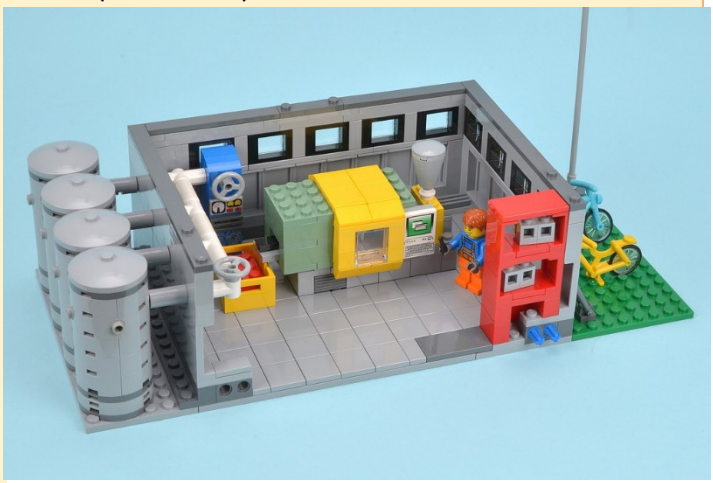
```
#include <iostream>
#include <string>
#include <algorithm> //siin on sort meetod
using namespace std;

string mudel = "AKQJT98765432";

bool on_pealpool(string a, string b) // võrdlusfunktsioon kaardistringide võrdlemiseks
{
    if (a[0] != b[0])
        return (a[0] > b[0]); // mastid on tähestiku järjekorras tagurpidi
    else {
       // võrdleme teise tähemärgi positsioone mudelstringis: see, mis on eespool, on
       // väiksem (ehk pakis pealpool)
        return mudel.find(a[1]) < mudel.find(b[1]);
   }
}
```

```
int main()
{
    int n;
    cin >> n;
    string* pakk = new string[n];

    for (int i = 0; i < n; i++)
        cin >> pakk[i];

    // sorteerime paki, kasutades oma võrdlusfunktsiooni
    sort(pakk, pakk + n, on_pealpool);
    for (int i = 0; i < n; i++)
        cout << pakk[i] + ' ';
    return 0;
}
```

3.18 KONTROLLÜLESANDED

3.18.1 Vabrikud

Päikeselinnas on N vabrikut (1≤N≤100), millest igaüks valmistab oma graafiku alusel kaupa. Täpsemalt kulub igal vabrikul uue kaubapartii valmistamiseks P_i päeva, kus 1≤i≤N. Kui kaup on valmis, tullakse sellele veoautoga järele. Igal päeval, mil mõnes vabrikus on kaupa, tuleb veoautojuht ja toob kõigist vabrikutest valminud kaubad ära. Juht töötab aga ainult tööpäevadel, laupäeviti ja pühapäeviti tuleb valminud kaubad lihtsalt kohapeal ära müüa.
Antud on periood, mida uurida ning vabrikute andmed. Leida, mitu reisi peab veoautojuht selle aja jooksul tegema. Aja lugemine algab alati esmaspäevast.
Sisendi esimesel real on uuritava perioodi pikkus T (1≤T≤10000). Teisel real on vabrikute arv N. Järgmistel N real on arvud P_i (1≤Pi≤1000). Väljastada veoauto reiside arv.



NÄIDE:
14
3
3
4
8
Vastus:
5 (esimene vabrik väljastab kaupa 3., 6., 9. ja 12. päeval, teine vabrik 4., 8. ja 12. päeval, kolmas ainult 8. päeval. Kuues päev jääb lugemata, sest see on laupäev).

3.18.2 Jalgpalliturniir

Jalgpalliturniiril saadakse 3 punkti võidu, 1 punkt viigi ning 0 punkti kaotuse eest. Meeskonnad järjestatakse järgmiste kriteeriumide alusel (kui kõigi eelmiste kriteeriumide tulemus on võrdne, võetakse järgmine)

1. Punktide arv (rohkem on parem)
2. Võitude arv (rohkem on parem)
3. Väravate vahe (suurem on parem)
4. Löödud väravate arv (rohkem on parem)
5. Mängitud mängude arv (vähem on parem)
6. Tähestikuline järjekord

Antud on jalgpalliturniiri mängude tulemused. Leida selle põhjal meeskondade järjestus.

Sisendi esimesel real on meeskondade arv `N(1≤N≤1000)`. Järgmisel `N` real on igaühel ühe meeskonna nimi. Nimed koosnevad kuni 32 märgist ja ainult ladina tähtedest. Meeskondadele järgneval real on mängude arv `M (1≤M≤1000)`. Järgmisel `M` real on igaühel ühe mängu tulemus järgmises formaadis: <Nimi1>-<Nimi2> <skoor1>:<skoor2>, näiteks `Eesti-Soome 1:2` Väljundisse kirjutada meeskondade järjestus ülaltoodud reeglite alusel.

NÄIDE (unistamine on ju lubatud, eks):

```
10
Eesti
Lati
Leedu
Poola
Taani
Norra
Rootsi
Soome
Prantsusmaa
Inglismaa
11
Eesti-Lati 2:1
Lati-Leedu 0:1
Poola-Eesti 1:1
Poola-Eesti 0:0
Poola-Eesti 2:2
Taani-Eesti 3:2
Norra-Lati 2:0
Rootsi-Eesti 3:1
Soome-Lati 2:0
Prantsusmaa-Eesti 0:1
Inglismaa-Eesti 0:1
Prantsusmaa-Lati 0:1
```

Vastus:

```
Eesti
Lati
Rootsi
Norra
Soome
Taani
Leedu
Poola
Inglismaa
Prantsusmaa
```

3.18.3 Erdöse arvud

Ungari matemaatik Paul Erdös (pildil) oli kuulus selle poolest, et ta kirjutas oma elu jooksul palju teadusartikleid (üle 1500), tehes samas koostööd paljude teiste teadlastega. Selle põhjal on defineeritud Erdöse arvu mõiste. Nende teadlaste Erdöse arv, kes Erdösega koos midagi kirjutasid, on 1. Nende Erdöse arv, kes kirjutasid midagi koos Erdöse kaasautoritega, on 2 jne.
On antud hulk teadustöid koos autorite ninedega. Leida iga autori Erdöse arv. Kui autor pole Erdösega seotud, väljastada tema kohta -1.
Sisendi esimesel real on teadustööde arv `N (1<=N<=1000)`. Järgmisel `N` real on igaühel vastava teadustöö info: kõigepealt semikoolonitega eraldatud autorite nimed, siis töö pealkiri.
Väljundisse kirjutada tähestiku järjekorras autorid ja nende Erdöse arvud. Vältimaks kodeeringust tulenevaid probleeme, on Paul Erdös on alati esitatud kui Erdos, Paul. Paul Erdöst ennast pole vaja vastusesse kirjutada.

NÄIDE:

```
10
Brown, Tom C.; Erdos, Paul; Freedman, A. R.; Quasi-progressions and descending
waves.
Li, Yong; Lipmaa, Helger; Pei, Dingyi; On delegatability of four designated
verifier signatures.
Mielikäinen, Taneli; Ukkonen, Esko; The complexity of maximum matroid-greedoid
intersection and weighted greedoid maximization.
Brown, Tom C.; Pei, Dingyi; Shiue, Peter; Irrational sums.
Brazma, Alvis; Jonassen, Inge; Vilo, Jaak; Ukkonen, Esko; Pattern discovery in
biosequences
Targo Tennisberg; Katrin Gabrel; Võistlusprogrammeerimine
Bollobas, Bela; Erdos, Paul; Graphs of extremal weights.
Lipmaa, Helger; Mielikäinen, Taneli; On private scalar product computation for
privacy-preserving data mining.
Bollobas, Bela; Das, Gautam; Gunopulos, Dimitrios; Mannila, Heikki; Time-series
similarity problems and well-separated geometric sets.
Mannila, Heikki; Ukkonen, Esko; A simple linear-time algorithm for in situ
merging.
```

Vastus:

```
Brazma, Alvis 4
Bollobas, Bela 1
Brown, Tom C. 1
Das, Gautam 2
Freedman, A. R. 1
Gunopulos, Dimitrios 2
Jonassen, Inge 4
Katrin Gabrel -1
Li, Yong 3
Mannila, Heikki 2
Mielikäinen, Taneli 3
Pei, Dingyi 2
Shiue, Peter 2
Targo Tennisberg -1
Ukkonen, Esko 3
Vilo, Jaak 4
```

3.18.4 Programmeerimisvõistlus

Programmeerimisvõistlusel on 1-100 osalejat ning 1-9 ülesannet. Osalejad saavad esitada lahendusi, mida loetakse kas õigeteks või valedeks. Tulemustes on osalejad järjestatud esmalt õigesti lahendatud ülesannete arvu järgi ja kui see on võrdne, siis kulunud aja järgi. Kui aeg on samuti võrdne, on osalejad oma numbrite järjekorras. Kulunud aega mõõdetakse vastavalt sellele, mitmendal võistluse minutil iga õige lahendus esitati. Kui võistleja esitas enne õiget lahendust samale ülesandele ka vale lahenduse, liidetakse ajale selle eest 20 minutit.
Sisendis on esimesel real esitatud lahenduste arv `N` ja järgmistel `N` real lahenduste logi ajalises järjestuses. Igal logi real on võistleja number, ülesande number, võistluse algusest kulunud aeg ning tulemus: õige (`C`) või vale (`I`).
Väljundisse kirjutada võistlejate paremusjärjestus vastavalt eespool kirjeldatud reeglitele. Iga võistleja kohta kirjutada tema number, lahendatud ülesannete arv ning kokku kulunud aeg.
NÄIDE:

```
1 2 9 I
3 1 11 C
1 3 19 I
1 2 21 C
1 1 25 C
```

Vastus:

```
1 2 66
3 1 11
```

Esimese võistleja lahendatud kolmas ülesanne ei mõjuta skoori, sest ta ei esitanud sellele ühtki õiget lahendust.

3.18.5 Pannkoogid

Vanaema teeb pannkooke ja asetab need taldrikule kuhja. Pannkoogid on natuke erineva suurusega, tähistame seda positiivse täisarvuga 1-100. Pannkoogikuhja on võimalik pannilabida abil osaliselt või täielikult ümber pöörata.
Olgu meil näiteks kuhi 8 4 6 7 5 2. Kui panna pannilabidas alt kolmanda koogi alla ja selle peale jäävad koogid ümber pöörata, saame kuhja 7 6 4 8 5 2. Kui järgmiseks teha pööre alates esimesest koogist, on kuhi pärast seda 2 5 8 4 6 7.
Sisendi esimesel real on pannkookide arv `N (1<=N<=30)`. Teisel real on pannkookide suurusi tähistavad N arvu. Eesmärgiks on sorteerida pannkoogid nii, et suuremad oleks allpool ja väiksemad peal. Väljundisse kirjutada selleks vajalikud pööramised.

NÄIDE:

```
5
5 4 3 2 1
```

Vastus:

```
1
```

NÄIDE:

```
5
5 1 2 3 4
```

Vastus:

```
1 2
```

3.18.6 Kaartide segamine

Tavalises kaardipakis on 52 mängukaarti. 19. sajandi teisel poolel elanud ja vesternitest tuntud kaardimängija Doc Holliday (pildil Val Kilmeri kehastatuna) oskas kaarte segada nii, et nad muutsid oma järjekorda spetsiifilisel viisil. Samas on tema käte liikumise järgi võimalik taibata, millist viisi ta parajasti kasutab. Leida, mis on kaartide lõplik järjekord eeldusel, et Doc Holliday alustab segamist uue pakiga, kus kaardid on sorteeritud.
Sisendi esimesel real on kaks täisarvu: võimalike segamisviiside arv `N` (`1<=N<=100`) ja segamiste arv `M` (`1<=M<=1000`). Järgmisel `N` real on igaühel mingis arvud 1-52, mis näitavad, millisesse järjestusse esialgne pakk seda segamisviisi kasutades segatakse. Lõpuks tuleb `M` rida, millest igaühel on 1 täisarv, mis näitab, mitmendat segamisviisi Doc Holliday järjest kasutab.
Väljundisse kirjutada arvud 1-52 sellises järjestuses nagu kaardid lõpuks on.

NÄIDE:

```
2 2
2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 51
52 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 1
1
2
```

Vastus:

```
51 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 2
```

Esimene segamine vahetab omavahel esimesed kaks ja viimased kaks kaarti. Teine segamine vahetab esimese ja viimase kaardi.

3.18.7 Kass klaviatuuril

Juhan kirjutab arvutis teksti, aga ei vaata samal ajal ekraanile. Samal ajal istub laual kass, kes läheb aeg-ajalt `Home` ja `End` klahvide pihta, viies Juhani kursori vastavalt teksti algusse või lõppu. Leida, milline tekst niiviisi lõpuks välja tuleb.

Sisendis on täpselt üks tekstirida, mis koosneb tavalistest tähtedest ja tühikutest, see vastab Juhani kirjutatavale tekstile. Kohad, kus kass vajutas `Home`, on tähistatud märgiga `[` ja kohad, kus kass vajutas `End`, on tähistatud märgiga `]`. Väljundisse kirjutada tegelik tekst.

NÄIDE:

```
tere [maa]ilm
```

Vastus:

```
maatere ilm
```

3.18.8 Pinugrammid

Pinu andmestruktuuri saab kasutada anagrammide koostamiseks. Tähistame pinu operatsiooni *push* tähega `i` ning *pop* tähega `o`. Esialgsest sõnast saab anagrammi järgmiselt:
1. Lükkame kõik selle sõna tähed pinusse.
2. Pinust välja tõmmatud tähtedest koostame uue sõna.
Vastavalt operatsioonide järjekorrale võib saada erinevaid anagramme.
Näiteks esialgsest sõnast TROT võib saada sõna TORT kahel viisil:

```
i i i i o o o o
i o i i o o i o.
```

Sisendi kahel real on antud sõnade paar, mis moodustavad anagrammi. Leida kõik võimalused, kuidas kirjeldatud pinuoperatsioonide abil on võimalik esimene sõna teiseks muuta. Võimalused esitada leksikograafilises järjekorras.

NÄIDE:

```
madam
adamm
```

Vastus:

```
i i i i o o o i o o
i i i i o o o o i o
i i o i o i o i o o
i i o i o i o o i o
```

NÄIDE:

```
bahama
Bahama
```

Vastus:

```
i o i i i o o i i o o o
i o i i i o o o i o i o
i o i o i o i i i o o o
i o i o i o i o i o i o
```

NÄIDE:

```
eric
rice
```

Vastus:

```
i i o i o i o o
```

3.18.9 Parv

Enamasti ületavad autod jõgesid sildade kaudu. Mõnes kohas pole siiski sildu ehitatud ja nende asemel kasutatakse parvesid. Kui ühtki autot ei ole, siis parv lihtsalt ootab. Kui autosid saabub, läheb parv vajadusel õigele kaldale, võtab auto(d) peale ning toimetab nad üle. Kui teisel pool on samal ajal autosid ootamas, võtab parv nad peale ja toob üle. Parv mahutab korraga N autot ja jõe ületamine võtab T minutit. Eeldame lihtsuse mõttes, et parve peale ja parvelt maha sõit aega ei võta. Alguses on parv jõe vasakul kaldal.

Sisendi esimesel real on kolm täisarvu: parvele korraga mahtuvate autode arv `N`, jõe ületamiseks kuluv aeg `T` ja autode koguarv `M`. Järgmistel `M` real on iga auto saabumise aeg ja kallas, kuhu ta saabub (`L` või `R`). Väljundisse kirjutada iga auto jaoks, millisel ajal ta vastaskaldale jõuab.

NÄIDE:

```
2 10 10
0 L
10 L
20 L
30 L
40 L
50 L
60 L
70 L
80 L
90 L
```

Vastus:

```
10
30
30
50
50
70
70
90
90
110
```

NÄIDE:

```
10 R
25 L
40 L
```

Vastus:

```
30
40
60
```

3.18.10 Ahelmurrud

Ahelmurruks nimetatakse avaldist kujul

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

Ahelmurdu saab lühendatult kirja panna kujul [a_0, a_1, a_2,…,a_n]. Näiteks järjend [2, 3, 1, 4] on võrdne murruga

$$2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{4}}} = \frac{43}{19}$$

Iga lihtmurdu on võimalik teisendada selliseks ahelmurruks. Antud on kaks positiivset täisarvu: lihtmurru lugeja ja nimetaja, leida sellele vastav ahelmurd ja väljastada see järjendiesituses.

NÄIDE:

```
43 19
```

Vastus:

```
2 3 1 4
```

NÄIDE 2:

```
1 2
```

Vastus:

```
0 2
```

3.19 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk3 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Failid	Kirjeldus
Lemming.cpp, Lemming.java, Lemming.py	Lemmingute ülesanne (pinu ja järjekord)
Kursused.cpp, Kursused.java, Kursused.py	Usina ülikooli ülesanne (kujutis)
Sortimine.cpp, Sortimine.java, Sortimine.py	Erinevad sorteerimismeetodid
Kaardid.cpp, Kaardid.java, Kaardid.py	Kaardipaki ülesanne (oma võrdlemine)

4 ARVUTEOORIA

4.1 JAGUVUS

4.2 ALGARVUD.

Eratosthenese sõel.

Euleri korrutis

4.3 SÜT JA VÜK

Eukleidese algoritm.

Kongruentsid.

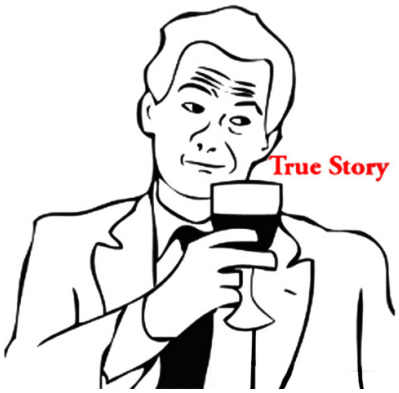
4.4 POSITSIOONILISED ARVUSÜSTEEMID

Negatiivse ja mittetäisarvulise alusega arvusüsteemid.

4.5 SUURTE ARVUDEGA ARVUTAMINE

Efektiivne astendamine

Alati pole suuri arve vaja, selle asemel võib kasutada logaritmi ja eksponenti!

Näide-pokker 

4.6 KONTROLLÜLESANDED

5 DÜNAAMILINE PLANEERIMINE ALGAJATELE

Dünaamiline planeerimine on üks algoritmitehnika leivanumbreid, seda on võimalik kasutada paljudes situatsioonides otse ja see moodustab ka tähtsa koostisosa mitmes spetsiifilises algoritmis.

Inglise keeles kasutatakse enamasti terminit *dynamic programming*, aga ajalooline taust on ka seal seotud planeerimisega. Dünaamilise planeerimise leiutas ameerika rakendusmatemaatik Richard Bellman 1950. aastate alguses, kui ta töötas RAND Corporationis, mis omakorda täitis USA Õhujõudude tellimusi. Bellman uuris, kuidas paremini planeerida mitme-etapilisi protsesse, kuid leidis, et sõna „planeerimine“ oli liiga üldine ja mitmetähenduslik. Seetõttu nimetas ta oma uurimise tulemust dünaamiliseks programmeerimiseks, pidades silmas protsessi kavandamist spetsiifilisel viisil. Hiljem on sõna „programmeerimine“ saanud hoopis teiselaadse sisu, aga termin on jäänud. Eesti keeles öeldakse nii dünaamiline planeerimine kui ka dünaamiline programmeerimine. Edaspidi ütlen lihtsalt DP, igaüks võib ise otsustada, kumb sõna talle suupärasem on. Kui sa oled internetist lugenud, et DP on hoopis midagi muud, siis oled valejälgedel.

DP on rohkem üldine lähenemisviis kui konkreetne retsept või algoritm. Seepärast on siin peatükis ka ohtralt näidisülesandeid, mis illustreerivad erinevaid DP kasutamise võimalusi.

Käesolevas peatükis räägitakse tihti algoritmi keerukusest. Nagu kolmandas peatükis öeldud, on keerukuse mõõtmisel alati vaja defineerida, millise elementaaroperatsiooni suhtes me seda mõõdame. Siin peatükis on lihtsuse ja ülevaatlikkuse huvides elementaaroperatsioonina kasutatud vastava algoritmi üksikuid samme, olgu siis tegu objektide võrdlemise, ümber tõstmise või muu sarnasega. Valdava enamiku reaalsete ülesannete puhul on elementaaroperatsioonid piiratud pikkusega (nt 64-bitiste) arvudega sooritatavad tehted. Nende puhul loeme lihtsuse mõttes, et nad võtavad alati konstantse aja, nii et kirjeldatud keerukus on ühtlasi ka ajaline keerukus.

5.1 SISSEJUHATAV ÜLESANNE – FIBONACCI JADA

Järgnevat ülesannet võib sageli kohata juba programmeerimise algõppe kursustel. Fibonacci jada on defineeritud lihtsalt: alustatakse nullist ja ühest ning iga järgmine element on eelmise kahe summa:

$$F_n = \begin{cases} 0, & \text{kui } n = 0 \\ 1, & \text{kui } n = 1 \\ F_{n-1} + F_{n-2} & \text{muul juhul} \end{cases}$$

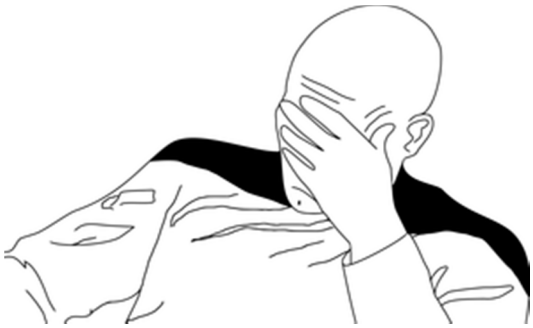
Definitsioon on väga lihtne, aga matemaatikud on Fibonacci jadal leidnud tohutult palju huvitavaid omadusi ja leiavad neid edasi. Samamoodi saab Fibonacci jada abil illustreerida huvitavaid kontseptsioone programmeerimisest.

Ülesande sõnastus on samuti lihtne:

Leida Fibonacci jada liige kohal n.
NÄIDE:
`n = 10`
Vastus: 55

5.1.1 Tavaline rekursioon ehk jõumeetod

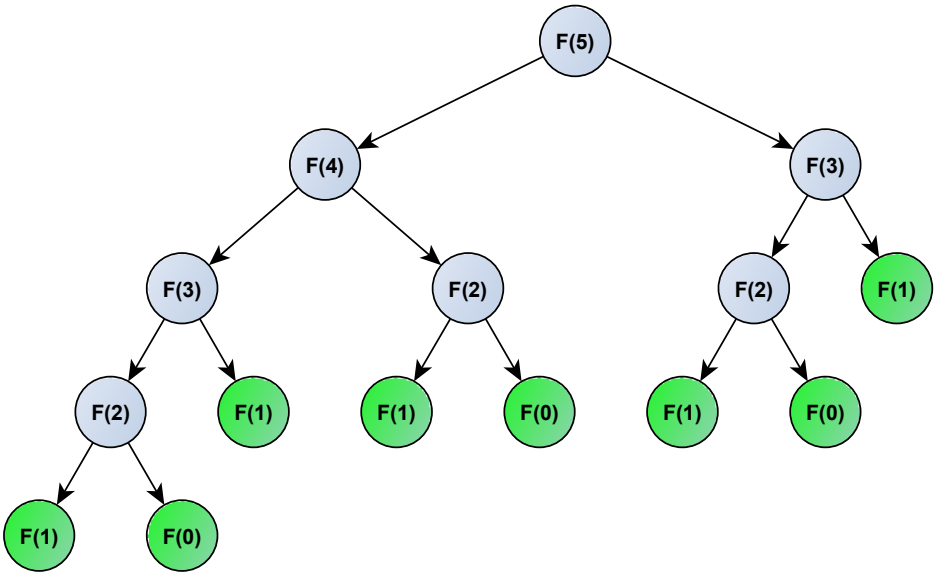
Eelneva ülesande lahendamiseks kirjutab algaja programmeerija tavaliselt definitsiooni põhjal järgmise funktsiooni:



```
int Fib1(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fib1(n - 1) + Fib1(n - 2);
}
```

Algoritmide programmeerimises kogenum programmeerija võtab selle koodi peale aga peast kinni. Miks? Sest näiteks `Fib1(5)` arvutamiseks leiab programm `Fib1(4)` ja `Fib1(3)`. `Fib1(4)` jaoks arvutab programm (uuesti!) `Fib1(3)` ja `Fib1(2)` väärtused.

Kokku moodustavad väljakutsed sellise puu:



Funktsiooni väljakutsete arv (ning sellega seoses programmi tööaeg) lähevad kiiresti käest ära:

N	Vastus	Väljakutsete arv
1	1	1
2	1	3
3	2	5
4	3	9
5	5	15
6	8	25
7	13	41
8	21	67
9	34	109
10	55	177
20	6765	21891
30	832040	2692537
40	102334155	331160281

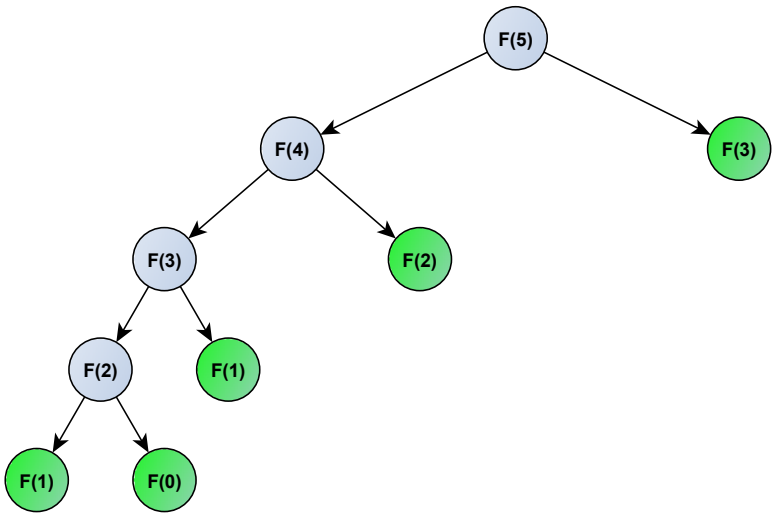
Väga kaugele ilmselt nii ei jõua.

5.1.2 Mäluga rekursioon

Väljakutsete arv kasvab väga kiiresti, kuid nagu eelpool juba mainitud, siis funktsiooni kutsutakse välja palju kordi täpselt sama parameetriga. Sellisel juhul on targem hoida juba arvutatud vahetulemused meeles:

```
const int Max = 1000;
long A[Max];
long Fib2(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;

    if (A[n] == 0)  // Kui A[n] väärtus pole veel teada, leiame selle
        A[n] = Fib2(n - 1) + Fib2(n - 2);

    return A[n];  // Aga kui on juba teada, siis lihtsalt tagastame
}
```

Seekord on puu palju väiksem:



Ja töö palju kiirem:

N	Vastus	Väljakutsete arv
1	1	1
2	1	3
3	2	5
4	3	7
5	5	9
6	8	11
7	13	13
8	21	15
9	34	17
10	55	19
20	6765	39
30	832040	59
40	102334155	79
50	12586269025	99
60	1548008755920	119
70	190392490709135	139
80	23416728348467685	159
90	2880067194370816120	179

Meetodi väljakutsete arv kahanes eksponentsiaalsest lineaarseks! Sellist vahepealsete tulemuste meelde jätmist nimetatakse **mäluga rekursiooniks** ja see on intuitiivselt loogiline samm DP mõtlemisviisi mõistmisel. Pane ka tähele, et mäluga rekursiooni puhul vahetatakse ajaline keerukus täiendava mäluvajaduse vastu – toodud lahendus hoiab kõiki vahetulemusi meeles.

5.1.3 Alt üles DP

Eelmine lähenemine töötas „ülalt alla“ – alguses leiti suurem väärtus, millest liiguti väiksematele. Tulemuse seisukohalt võib aga sama hästi liikuda „alt üles“ ehk väiksematelt arvudelt suurematele.

Sellele vastab järgmine kood:

```
long Fib3(int n)
{
    const int Max = 100000;
    long A[Max];
    A[0] = 0;
    A[1] = 1;

    for (int i = 2; i <= n; i++) {  // Arvutame algusest peale terve jada
        A[i] = A[i - 1] + A[i - 2];
    }

    return A[n];
}
```

Alt üles lähenemine on nn „stiilipuhas“ DP. Kui DP-st räägitakse, peetakse tavaliselt silmas just seda varianti. Selline lahendus töötab enamasti ka kiiremini, sest meil pole enam rekursiooni, vaid ainult lihtne tsükkel. Kaasaegsed kompilaatorid suudavad küll ka eelmisest lahendusest ise rekursiooni kõrvaldada, nii et `Fib2` ja `Fib3` kompileeritud koodi efektiivsus on sama, kuid keerulisema näite puhul ei tarvitse see enam nii olla.

5.1.4 Kokkuhoidlik DP

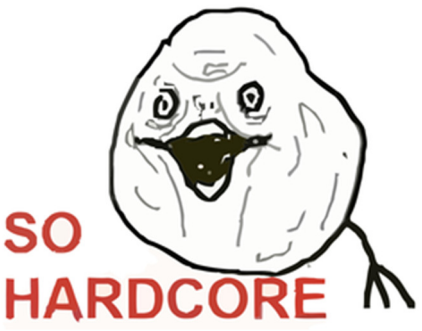
Mõttearenduse viimase sammuna saab teha veel ühe olulise optimiseerimise: pane tähele, et väärtuste massiivis kasutatakse korraga ainult kolme väärtust, seega pole ülejäänud massiivi vaja.

Tulemuseks on kood, mille mäluvajadus on konstantne ning tööaeg lineaarne:

```
long Fib4(int n)
{
    long A[3];
    A[0] = 0;
    A[1] = 1;

    for (int i = 2; i <= n; i++) {
        A[i % 3] =
            A[(i - 1) % 3] +
            A[(i - 2) % 3];
    }

    return A[n % 3];
}
```

Märkus: kui matemaatika appi võtta, siis saab Fibonacci arve leida veel efektiivsemalt, kasutades Binet’ valemit:

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}, \text{kus } \varphi = \frac{1+\sqrt{5}}{2} \text{ ja } \psi = \frac{1-\sqrt{5}}{2}$$



Kui vastust soovitakse piiratud täpsusega, saab Binet’ valemi abil leida vastuse konstantse ajaga, kuid kui soovime piiramatut tüvekohtade arvu, on ka see valem lineaarse keerukusega.

Kas kuitahes suuri Fibonacci arve saab üldse leida kiiremini kui lineaarse ajaga? Ei saa, sest F_n sõltub arvust n eksponentsiaalselt ning F_n tüvekohtade arv sõltub F_n-ist omakorda logaritmiliselt. Seega

sõltub F_n tüvekohtade arv n-ist lineaarselt, mis tähendab, et ainuüksi tulemuse välja kirjutamine (või kusagile salvestamine) võtab lineaarse aja.

Põhjalikumat Fibonacci arvude leidmise käsitlust saab lugeda ka aadressilt https://www.nayuki.io/page/fast-fibonacci-algorithms.

5.1.5 DP retsept

Uuritud näide oli üpris lihtne, aga paljude DP ülesannete lahenduste leidmine on taandatav samadele mõttelistele sammudele, mida ma nimetan **DP retseptiks**:

1. **Mõtle, milline võiks olla jõumeetodiga rekursiivne lahendus. Kui vaja, joonista programmi töö puuna välja.**
2. **Proovi rekursioonipuu harusid taaskasutada, jättes vahepealsed vastused meelde.**
3. **Alustagi kohe nende vahepealsete vastuste väljaarvutamisest.**
4. **Kui võimalik, taaskasuta uute vastuste jaoks eelmiste vastuste alt vabanenud mälu.**

DP tavapäraseks võtteks on, et neid vahepealseid vastuseid hoitakse **väärtuste tabelis**. Kui Fibonacci arvude puhul jäi tabel lõpuks ainult kolme lahtri suuruseks, on see tavaliselt keerulisema struktuuriga. Järgmised jaotused uurivadki mitmesuguseid kvalitatiivselt erinevaid võimalusi läbivaatuspuude ahendamiseks lihtsamatesse tabelitesse.

5.2 LINEAARNE VÄÄRTUSTE TABEL - OPTIMAALNE MAKSMINE

Järgmine optimeerimisega seotud situatsioon juhtub sageli toidupoes, aga olemuselt sarnast probleemi tuleb lahendada ka tõsisemates olukordades. Tegemist on klassikalise probleemiga, mis on inglise keeles tuntud kui *change-making problem.*

On antud N erineva väärtusega münte: V_1, V_2, …, V_N. Leida minimaalne müntide arv, millega saab tasuda summa S. Iga väärtusega münte on kasutada piiramatult.
NÄIDE:

```
N = 3
V = [1, 3, 4]
S = 6
```

Vastus: 2 ([3, 3])

5.2.1 Ahne algoritm

Teatud väärtusega müntide (nt tavalised euromündid) puhul saab kasutada lihtsat ahnet lähenemist: võta kõige suurema väärtusega münte nii palju, kuni nende koguväärtus ei ole suurem kui vajalik summa S, seejärel lisa järgmise suurima väärtusega münte nii palju, kui mahub jne kuni soovitud summa on käes. Näiteks 84 sendi maksmiseks on vaja 5 münti: 50-, 20-, 10-, 2- ja 2-sendine.

Ahne algoritm on väga efektiivne. Kui antud müntide väärtused on suuruse järgi sorteeritud, siis on see lineaarse keerukusega $O(n)$. Kui väärtused on antud juhuslikus järjekorras, tuleb need enne sorteerida (või otsida iga kord kõige suuremat kasutamata münti, mis veel maksmata summat ei ületa) ja siis on keerukusklass $O(n\ log\ n)$.

Kui mündid on aga suvaliste väärtustega, siis võib ahne algoritm anda vale tulemuse. Kui on antud näiteks mündid väärtustega 1, 3 ja 4 ning tarvis on saada summa S = 6, annab ahne lähenemine

tulemuseks 3 münti (4, 1 ja 1), mis ei ole õige, sest optimaalne lahendus on kaks kolmelist münti. Tundub, et garanteeritult õige vastuse saamiseks tuleb siiski kõik võimalused läbi vaadata.

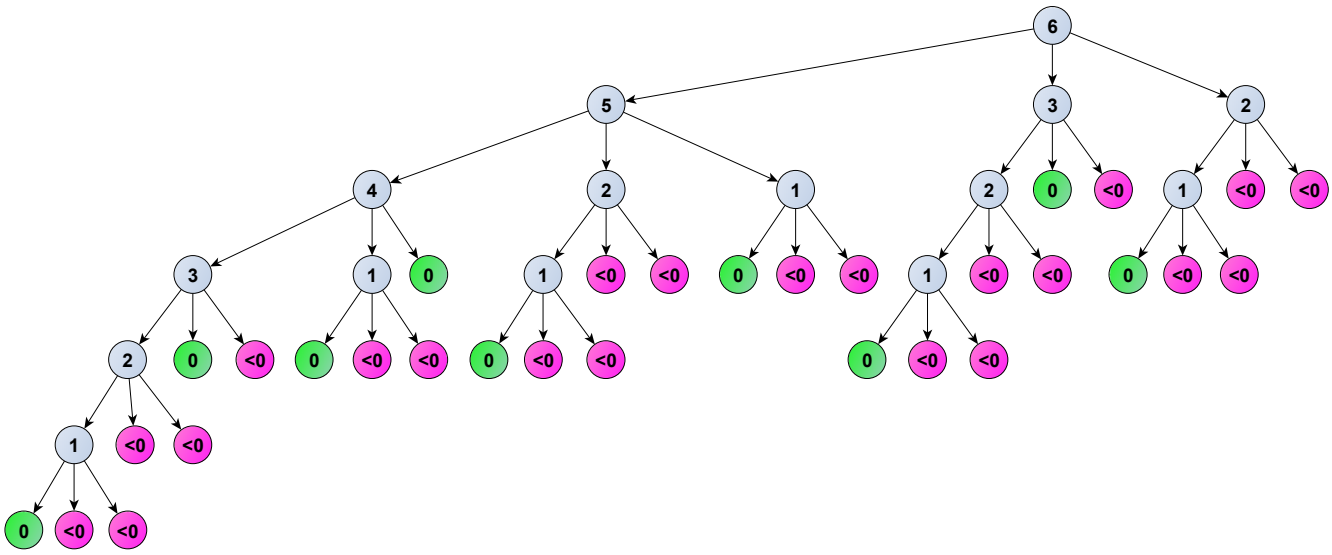
5.2.2 Kõigi variantide läbivaatamine (rekursioon)

Kui teises peatükis rekursioon selgeks sai, siis antud ülesande lahendamine kõikide variantide läbivaatamise teel on lihtne. Alustame olukorrast, kus meil oleks kogu summa makstud, aga müntide arv teadmata, ning uurime, kuidas me võisime selle olukorra saavutada. Selleks oletame iga väärtuse jaoks, et viimati lisati just selle väärtusega münt, ning kordame rekursiivselt sama protseduuri. Igal järgmisel tasemel on seega üks münt vähem ja väärtuste summa selle mündi väärtuse võrra väiksem. S = 0 jaoks on vaja võtta 0 münti. Kui aga summa läheb negatiivseks, siis sellist müntide kombinatsiooni esineda ei saa.

$$leiaKogus(S) = \begin{cases} \infty, kui\ S < 0 \\ 0, kui\ S = 0 \\ \min\limits_{V_i \le S}(1 + leiaKogus(S - V_i)) \end{cases}$$

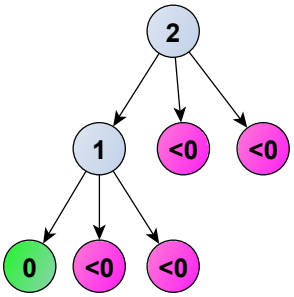
```
int leiaKogus(int S)
{
    if (S < 0) {
        return MaxInt;  //tagastame "lõpmatuse"
    }
    if (S == 0) {
        return 0;
    }
    else {
        int vastus = MaxInt;

        for (int i = 0; i < N; i++) {
            vastus = min(vastus, 1 + leiaKogus(S - V[i]));
       }
       return vastus;
    }
}
```

Selline lähenemine annab küll õige vastuse, kuid on eksponentsiaalse keerukusega $O(N^S)$. Näites toodud andmetega käib programm läbi kogu järgmise puu (igas sõlmes on vastav S väärtus):



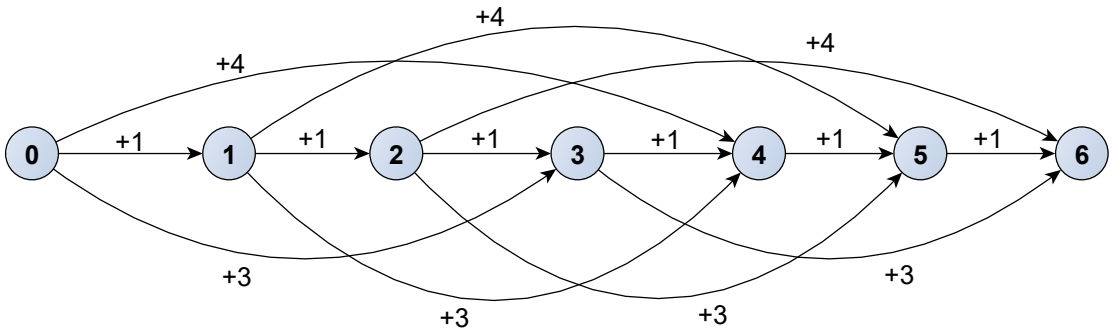
5.2.3 DP lahendus

Tähelepanelik lugeja märkab kindlasti, et selles puus mitmed harud korduvad, nt tipust 2 algavat alampuud



esineb ülaltoodud puus tervelt 4 korda! See tähendab, et tegelikult arvutab programm sama tulemust mitmeid kordi. See võiks viia mõtted kohe mäluga rekursiooni ehk ülevalt alla DP juurde. Kuna antud ülesandes järgmine samm ei sõltu sellest, kuidas eelmine tulemus saadi (iga nimiväärtusega münti on lõputul hulgal), siis piisab iga järgmise sammu leidmiseks eelmise sammu tulemuse teadmisest. Kokku on võimalikke tulemusi (müntide väärtuste summasid) 0 kuni S, kus S on otsitav summa. Niisiis piisab ülesande lahendamiseks teadmisest, kui mitu münti kulub mingi summa moodustamiseks.

Teame, et S = 0 korral kulub 0 münti, uurime, mis juhtub, kui siia münte lisada: saab lisada ühe mündi ja saame summad 1, 3 ja 4. Nüüd uurime järgmist vähimat leitud summat (S = 1) ja lisame siia ühe mündi (tulemuseks summad 2, 4 ja 5) jne. Seda kujutab järgmine graaf, kus tippudes on summad ning servades lisatava mündi nimiväärtused:



Kuigi esmapilgul tunduvad ülaltoodud puu ja graaf üsna erinevad, on neil siiski palju ühist. Sellel puul on kõik ühe ja sama numbriga alamtipust algavad alampuud täpselt samasugused. Kui kõik sama numbriga tipud omavahel kokku viia, säilitades samas kõik servad, saame väga sarnase tulemuse üleval konstrueeritud graafiga, kus peamiseks erinevuseks on noolte suund. Samuti pole nullist alustades vaja allapoole minna (seetõttu siseneb sõlmedesse 1 ja 2 ainult üks serv, sõlme 3 kaks serva). Nii on ka visuaalselt hea aru saada, mida mõeldakse „ülevalt alla“ ja „alt üles“ lähenemiste puhul. See on ka põhjus, miks me väärtuste jada üles ehitades võtame aluseks alati seni vähima uurimata väärtuse.

Funktsioon, mis leiab „alt-üles“ DP-d kasutades lahenduse:

```
int leiaKogus(int S)
{
    int kogused[S + 1];
    kogused[0] = 0;
    for (int i = 1; i <= S; i++) {
       //väärtusteks alguses „lõpmatused“, kui selle koguseni ei jõuta, siis jääb see
       //väärtus - sisuliselt tähendab see, et sellel kohal asuvat summat ei saa antud
       //müntidest moodustada
      kogused[i] = MaxInt;
    }

    for (int i = 1; i <= S; i++) {
      for (int j = 0; j < N; j++) {
          if (V[j] <= i) { //jätab vahele mündid, mille väärtus on suurem kui summa
              kogused[i] = min(kogused[i - V[j]] + 1, kogused[i]);
          }
      }
    }
    return kogused[S];
}
```

Kui sa pole alt-üles DP lähenemisega koodiga varem kokku puutunud, võib see kood paista arusaamatu. Praegune näide on üsna lihtne, kuid edaspidised lähevad keerulisemaks. Kui miski jääb segaseks, soovitan võtta siit koodi, siluris läbi käia ja kõigi vajalike muutujate väärtusi samm-sammu haaval jälgida. See on minu kogemuses parim viis DP õppimiseks.

Massiivis `kogused` hoitakse kõige väiksemat müntide arvu, millega senini indeksile vastav summa on saadud. Kohale 0 omistatakse väärtuseks 0, sest summa S = 0 saamiseks on vaja võtta 0 münti. Teistele kohtadele pannakse mingi suur väärtus, mida normaalselt maksmisel pole võimalik saada (näiteks suurem kui S – siis on kindel, et isegi ühese väärtusega müntidest ei saa vastavat väärtust kokku). Edasi vaadatakse iga summa korral läbi kõik müntide väärtused ja leitakse, kas valitud münti eelmisele seisule lisades saab vaadeldava seisu jaoks parema tulemuse, kui juba leitud on. Järgmises tabelis on toodud leitud kogused ja see, milliste müntide abil need on saadud:

Summa S	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Kogus	0	1	2	1	1	2	2
Lisatud mündi väärtus (eelmine summa)	-	1 (0)	1 (1)	3 (0)	4 (0)	4 (1)	3 (3)

Summa S = 6 on saadud summale S = 3 mündi väärtusega 3 lisamise teel. Summa S = 3 omakorda on saadud summale S = 0 mündi väärtusega 3 lisamise teel. Säilitades info iga summa saamiseks viimati lisatud mündi väärtuse kohta, saab lahendada ka ülesandeid, kus on vaja tuua välja, mis väärtusega münte kasutati. Järgenvalt näide koodist, mis just seda teeb:

```
int leiaKogus(int S)
{
	int kogused[S + 1];
	int viimased[S + 1]; //viimasena lisatud mündi väärtus iga summa jaoks
	kogused[0] = 0;
	viimased[0] = 0;
	for (int i = 1; i <= S; i++) {
		kogused[i] = MaxInt;
		viimased[i] = 0;
	}
	for (int i = 1; i <= S; i++) {
		for (int j = 0; j < N; j++) {
			if (V[j] <= i && kogused[i - V[j]] + 1 < kogused[i]){
				kogused[i] = kogused[i - V[j]] + 1;
				viimased[i] = V[j];
			}
		}
	}
	int k = S;
	while (k > 0) {
		cout << viimased[k] << endl;
		k -= viimased[k];
	}
	return S;
}
```

Sellisel viisil läbitud tee taastamist nimetatakse **tagurdusmeetodiks** (i.k *backtracking*) ning see on sage võte DP-d kasutavate lahenduste juures. Eelmises näites on meeles peetud viimati lisatud mündi väärtus ja selle järgi saab arvutada selle mündi lisamisele eelnenud seisu. Summa (=massiivi indeks) väheneb sel juhul viimasena lisatud mündi väärtuse võrra. Alternatiivina võib pidada meeles ka eelmise seisu indeksi ehk eelnenud summa. Sellisel juhul tuleb arvutada kasutatud mündi väärtus, milleks on vaadeldava summa ja eelmise summa vahe.

5.3 PIKIMA KASVAVA OSAJADA LEIDMINE

Jällegi on tegemist klassikalise probleemiga, inglise keeles *the **longest increasing subsequence (LIS)** problem*, mis esineb sageli keerulisemate probleemide alamosana:

On antud N arvust koosnev jada. Leida selles jadas pikima kasvava osajada pikkus. Jada on kasvav siis, kui iga i korral $J_i < J_{i+1}$. Osajada elemendid ei pea olema omavahel järjest, s.t seal võib olla ka auke.
NÄIDE:
`N = 6`
`Jada = [1, 3, 2, 4, 3, 7]`
Vastus: 4 (näiteks [1, 3, 4, 7])

5.3.1 Kõigi võimaluste läbivaatus

Üks lahendusvõimalus on täielik läbivaatus. Seda lahendust saab näha kaasasolevas failis VP_lisad.zip nimega LIS.cpp, LIS.java või LIS.py.

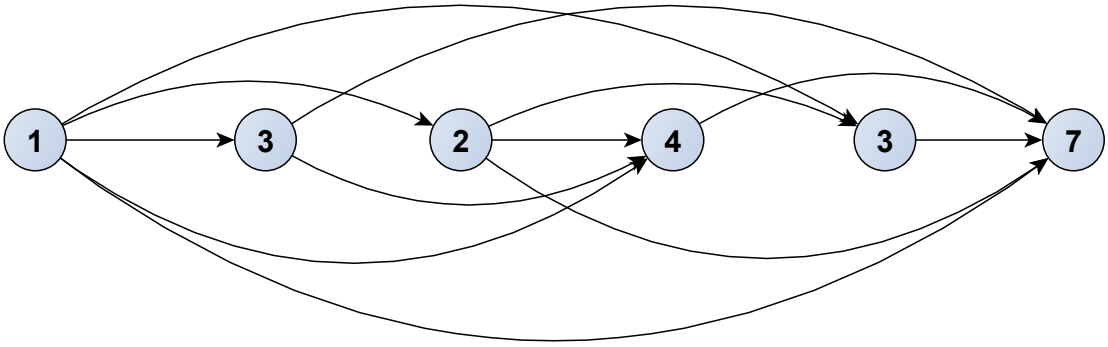
Selle lähenemise keerukus on $O(2^N)$, mis tähendab, et veidigi pikema jada korral muutub ülesanne praktiliselt lahendamatuks.

5.3.2 DP lahendus

Kui on teada pikim võimalik osajada kuni elemendini J_i, siis element J_{i+1} kas pikendab seda osajada ühe võrra või jääb pikima võimaliku osajada pikkus muutumatuks. J_{i+1} saab pikendada ainult sellist osajada, mille viimane element $J_k < J_{i+1}$.

Et seda võrratust efektiivselt kontrollida, on hea, kui teame J_k jaoks, kus $k < i + 1$, milline on pikim leitud osajada, mis lõpeb elemendiga J_k. Teiste sõnadega, hoiame meeles kõik võimalikud senised pikimad osajadad. Erinevaid võimalikke viimaseid elemente on sama palju, kui jadas elemente kokku ehk N. Üheelemendilise jada pikima kasvava osajada pikkus on muidugi 1.

Elementide lisamist illustreerib järgnev graaf:



Tippudeks on sisendjada elemendid ning kaared tähistavad, milliseid elemente saab antud elemendiga lõppevale jadale lisada. Iga serva väärtus on 1 ja vaja on leida pikim võimalik tee ühest servast teise.

Üheks võimaluseks on iga elemendi korral vaadata, millised järgmised elemendid antud elemendiga lõppevat pikimat osajada veelgi pikendavad. Kui mõni järgnev element on suurem kui vaadeldav element ja vaadeldava osajada pikendamine järgneva elemendiga annab pikema järgneva elemendiga lõppeva osajada, kui seni leitud, tähendab see, et leitud on parem vahetulemus. Siin on funktsioon, mis seda teeb:

```
int LeiaPikimPikkus(int N, int* jada) {
    int* pikkused = new int[N];
    for (int i = 0; i < N; i++) {
        pikkused[i] = 1;
    }

    int vastus = 0;
    for (int i = 0; i < N; i++) {
        vastus = max(vastus, pikkused[i]);
        for (int j = i + 1; j < N; j++) {
            if (jada[j] > jada[i]) {
                pikkused[j] = max(pikkused[j], pikkused[i] + 1);
            }
        }
    }
    return vastus;
}
```

Näitejada puhul pannakse kõigepealt igale elemendile vastavad pikima osajada pikkused võrdseks ühega, sest osajada pikkusega üks on alati võimalik. Seejärel asutakse võrdlema esimese elemendiga lõppevat osajada, milleks on [1] ja mille pikkus on 1. Nüüd käiakse tsükliga läbi kõik järgnevad

elemendid: antud juhul saab neid kõiki lisada vaadeldavale osajadale ja nii muudetakse pikkused 2-ks. Seejärel liigutakse järgmisele elemendile „3“ ja vaadatakse, kas sellele saab lisada järgnevaid elemente. Saame lisada elemendid 4 ja 7, nendega lõppevate osajadade pikimat pikkust muudetakse vastavalt. Järgnevas tabelis on toodud rea haaval, kuidas pikkuste jada muutub (nurksulgudes on toodud seni leitud pikim osajada; kui antud sammul on sama pikkusega alternatiivne jada, on see toodud sulgudes):

J	1	3	2	4	3	7
i = 0	1 [1]	1 [3]	1 [2]	1 [4]	1 [3]	1 [7]
i = 1	1 [1]	2 [1, 3]	2 [1, 2]	2 [1, 4]	2 [1 ,3]	2 [1, 7]
i = 2	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	2[1 ,3]	3 [1,3, 7]
i = 3	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4] ([1,2,4])	3 [1, 2, 3]	3 [1, 3, 7] ([1, 2, 7])
i = 4	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	3 [1, 2, 3]	4 [1,3,4,7] ([1,2,3,7])
i = 5	1 [1]	2 [1, 3]	2 [1, 2]	3 [1,3,4]	3 [1, 2, 3]	4 [1,3,4,7]

DP lahenduse keerukus on $O(N^2)$.

5.3.3 Tagurdusmeetodiga lahendus

Mõnikord soovitakse sarnast tüüpi ülesande vastuseks ka mõnd konkreetset osajada. Selleks on vaja pikkuse muutmisel meeles pidada viimase elemendi indeks. Osajada leidmiseks saab siis kasutada taas tagurdusmeetodit:

```
int* LeiaPikimKasvav(int N, int* S)
{
    int* pikkused = new int[N];
    int* indeksid = new int[N];

    for (int i = 0; i < N; i++) {
        pikkused[i] = 1;
        indeksid[i] = i;
    }
    int pikim = 0;
    int pikimaIndeks = 0;
    for (int i = 0; i < N; i++) {
        if (pikkused[i] > pikim) {
            pikim = pikkused[i];
            pikimaIndeks = i;
        }
        for (int j = i + 1; j < N; j++) {
            if (S[j] > S[i] && pikkused[i] + 1 > pikkused[j]) {
                pikkused[j] = pikkused[i] + 1;
                indeksid[j] = i;
            }
        }
    }

    int* osajada = new int[pikim];
        for (int i = pikim; i > 0; --i) {
            osajada[i-1] = S[pikimaIndeks];
            pikimaIndeks = indeksid[pikimaIndeks];
        }

    return osajada;
}
```

Märkus: pikima kasvava osajada ülesannet saab lahendada ka $O(n\ log\ n)$ keerukusega, vt näiteks https://en.wikipedia.org/wiki/Longest_increasing_subsequence

5.4 KAHEMÕÕTMELINE VÄÄRTUSTE TABEL – PIKIM ÜHINE OSAJADA

On antud kaks arvujada J1 pikkusega m ja J2 pikkusega n. Leida J1 ja J2 pikima ühise osajada pikkus.
NÄIDE:

```
m = 7
J1 = [1, 0, 3, 2, 5, 4, 6]
n = 6
J2 = [3, 1, 4, 0, 5, 4]
```

Vastus: 4 ([1, 0, 5, 4])

See on taas üks klassikaline progammeerimisülesanne, inglise keeles tuntud kui *the longest common subsequence (LCS) problem.*

5.4.1 DP lahendus

Selleks, et kasutada DP-d, on vaja kõigepealt leida, kas ülesandel on olemas väiksemad alamprobleemid, mille põhjal saaks suuremaid probleeme lahendada. Esimese ja viimase elemendi puhul on lihtne otsustada, kas see saab kuuluda pikimasse osajadasse või mitte. Seame nende jadade viimased elemendid kohakuti:

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

On kaks võimalust – viimased elemendid on kas samad või erinevad. Kui need elemendid on erinevad, nagu toodud näites, siis vähemalt ühe jada viimane element ei sobi pikimasse ühisesse osajadasse.

Pikim ühine osajada on siis sama kui jadade

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

või jadade

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

pikim ühine osajada.

Kindlasti tuleb proovida mõlemat võimalust, et teada saada, kumb annab parema tulemuse. Need probleemid on aga juba väiksemad kui esialgne ülesanne.

Vaatame alamülesandeid täpsemalt. Kui mõlema (alam)jada viimane element on sama, nagu näiteks:

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

siis see element sobib nende jadade ühisesse osajadasse, tuleb vaid leida eelnevate elementide pikim ühine osajada, eemaldades viimase elemendi:

1	0	3	2	5
3	1	4	0	5

Kuna elemente eemaldatakse ainult jada lõpust, siis jäävad jadad alati algusest kuni eemaldatava elemendini muutumatuks. Jada sellist algosa nimetatakse jada **prefiksiks**. Erinevaid prefikseid saab jadal olla sama palju, kui on jadas liikmeid (kui lugeda iga jada prefiksiks ka tühi jada, siis ühe võrra rohkem).

Nüüd tuleb veel mõelda, kust alata. Kui on antud kaks tühja jada, siis nende pikim ühine osajada on samuti tühi. Samuti, kui üks jadadest on tühi, on ka ühine osajada tühi ja pikkusega 0. Formaalselt:

$$Pikkus(X_i, Y_j) = \begin{cases} 0, kui\ i = 0 \vee j = 0 \\ Pikkus(X_{i-1}, Y_{j-1}) + 1, kui\ X_i = Y_j \\ max(Pikkus(X_{i-1}, Y_j), Pikkus(X_i, Y_{j-1})) \end{cases}$$

Siin on tabel, kus ridades on esimese osajada ning veergudes teise osajada kõik võimalikud prefiksid ning lahtrites on leitud nende prefiksite pikimad ühised osajadad:

	[]	3	3, 1	3, 1, 4	3, 1, 4, 0	3, 1, 4, 0, 5	3, 1, 4, 0, 5, 4
[]	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
1, 0	0	0	1	1	2	2	2
1, 0, 3	0	1	1	1	2	2	2
1, 0, 3, 2	0	1	1	1	2	2	2
1, 0, 3, 2, 5	0	1	1	1	2	3	3
1, 0, 3, 2, 5, 4	0	1	1	2	2	3	4
1, 0, 3, 2, 5, 4, 6	0	1	1	2	2	3	4

Täites alguses esimese rea ja veeru nullidega, saab kas mööda ridu või veerge (või ka diagonaale) liikudes täita lihtsa arvutusega kõik lahtrid: kui vaadeldavate prefiksite viimased elemendid langevad kokku, liidetakse eelmises reas ja veerus (diagonaalis üleval vasakul) leitud pikkusele üks. Tabelis on need lahtrid tähistatud sinise taustaga. Kui aga viimased elemendid on erinevad, täidetakse lahter kas samas veerus eelmise leitud pikkusega (üleval) või samas reas eelmise leitud pikkusega (vasakul), olenevalt, kumb on suurem. Viimases lahtris on ülesande vastus.

Siin on funktsioon, mis just sel moel ülesande lahendab:

```
int leiaYhised(int* jada1, int* jada2, int p1, int p2)
{
    int** yhised = new int*[p1 + 1];
    for (int i = 0; i <= p1; i++) {
        yhised[i] = new int[p2 + 1];
    }
    for (int i = 0; i <= p1; i++) {
        yhised[i][0] = 0;
    }
    for (int j = 0; j <= p2; j++) {
        yhised[0][j] = 0;
    }
    for (int i = 1; i <= p1; i++) {
        for (int j = 1; j <= p2; j++) {
            if (jada1[i - 1] == jada2[j - 1]) {
                yhised[i][j] = yhised[i - 1][j - 1] + 1;
            }
            else {
                yhised[i][j] = max((yhised[i - 1][j]), (yhised[i][j - 1]));
            }
        }
    }
    return yhised[p1][p2];
}
```

DP lahenduse keerukus on $O(nm)$, kus n ja m on jadade pikkused.

5.5 RISTSUMMADE LOENDAMINE

Nüüd veidi keerulisem ülesanne, kus läheb tarvis kahemõõtmelist väärtuste tabelit:

Naturaalarvu ristsummaks nimetatakse selle numbrite summat. Näiteks arvu 123 ristsumma on 1 + 2 + 3 = 6. Ülesandeks on leida, kui palju on arvust N väiksemaid arve, mille ristsumma võrdub arvu N ristsummaga (Eesti lahtiselt programmeerimisvõistluselt 2013).
NÄIDE:
N = 123
Vastus: 9 (Loendatavad arvud on 6, 15, 24, 33, 42, 51, 60, 105, 114).

5.5.1 Ristsummade arvutamine

Kui N ei ole kuigi suur (näiteks kuni miljon), on lihtne kõigi arvude ristsummad välja arvutada ning neid omavahel võrrelda. Edasi saab appi võtta matemaatilist trikitamist, arvestades näiteks, et arvud A ja B saavad anda sama ristsumma ainult siis, kui |A-B| jagub üheksaga. Seega piisab, kui vaadata ainult iga üheksandat arvu. Lisaks leidub veel teisi mustreid, mis võimaldavad teatud arvuvahemikke vahele jätta. Nii on võimalik ülempiiri edasi venitada, näiteks miljardini. Võistlusel oli aga ülempiiriks seatud N = 10^{18}, mida teha?

5.5.2 DP lähenemine

Appi tuleb DP retsept. Mõtleme ülesandest arvujärkude kaupa: kõik ühelised, kõik kümnelised, kõik sajalised jne. Arvude loendamine on siis nagu rekursiivne puu läbimine: alustatakse kõrgemast järgust ja kutsutakse kümme korda välja madalamat järku, sellest kutsutakse jälle kümme korda välja järgmist järku jne. Tekib palju kordusi. Näiteks arvu 123 ristsumma arvutamisel saab teada, et arvu 23

ristsumma on 5. Seega arvu 223 ristsumma arvutamisel pole alumise kahe koha ristsummat enam vaja leida, sest see on juba teada!

Siit ka oluline tähelepanek – iga järgmise järgu juures on oluline teada ainult seda, kui palju mingi ristsummaga arve eelmises järgus oli – kuidas need täpselt leiti, ei ole oluline.

Kuidas koostada väärtuste tabelit? Kuna antud kontekstis küsitakse võimaluste koguarvu, peaks need olema ka väärtused, mida meeles hoitakse. Kuna loendamisel toimuva “rekursiooni” madalamateks tasemeteks on teadmine, mitu mingi konkreetse ristsummaga arvu väikemate arvujärkude juures oli, sobib väärtuste tabeli üheks dimensiooniks senine maksimaalne arvujärk. Teiseks dimensiooniks sobib hästi ristsumma. Erinevaid ristsummasid saab olla ainult 9*K + 1, kus K on uuritavate arvude maksimaalne pikkus. Tabelisse paneme info, mitu mingi konkreetse ristsummaga arvu on leitud, st tabelis veerus i ja reas j on tulemus, mitu kuni i-kohalist arvu on olemas, mille ristsumma on j.

Kokkuvõttes tekib kahemõõtmeline tabel, mille üheks mõõtmeks on ristsumma väärtus ja teiseks mõõtmeks uuritava arvu maksimaalne kohtade arv.

Tabeli suurus on seega K x (9K + 1). K = 18 puhul teeb see 1467 lahtrit, mille täitmine on kvintiljoni arvu loendamise kõrval mõistagi tühiasi.

5.5.3 DP Exceliga

Kui tegu on ühe- või kahemõõtmelise väärtuste tabeliga, siis on vahel väga mugav kasutada tabelarvutusprogrammi, näiteks Excelit. Selles on väärtuste tabelit hea visualiseerida, samuti on olemas lihtsalt kasutatavad vahendid arvutuste realiseerimiseks, mis ühtede väärtuste alusel teisi leiavad.

Järgmiseks on toodud näide ristsumma ülesande põhjal genereeritud Exceli tabelist (originaalfail õpiku lisamaterjalides).

Veergude indeksid on arvude pikkused: esimene veerg tähistab kuni ühekohalisi arve, teine veerg kuni kahekohalisi jne. Read tähistavad ristsummasid: rida 0 ristsummat 0, rida 1 ristsummat 1 jne.

Üksikutes lahtrites on toodud, kui mitu sellise ristsummaga arvu olemas on: näiteks see, et reas 8 ja veerus 6 on arv 1287 tähendab, et on olemas 1287 kuni 6-kohalist arvu, mille ristsumma on 8 (antud kontekstis arvestame ka lühemaid arve, näiteks 125 on nagu 000125 ja loeb samuti arvuna, mille ristsumma on 8).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120	136	153	171
3	1	4	10	20	35	56	84	120	165	220	286	364	455	560	680	816	969	1140
4	1	5	15	35	70	126	210	330	495	715	1001	1365	1820	2380	3060	3876	4845	5985
5	1	6	21	56	126	252	462	792	1287	2002	3003	4368	6188	8568	11628	15504	20349	26334
6	1	7	28	84	210	462	924	1716	3003	5005	8008	12376	18564	27132	38760	54264	74613	100947
7	1	8	36	120	330	792	1716	3432	6435	11440	19448	31824	50388	77520	116280	170544	245157	346104
8	1	9	45	165	495	1287	3003	6435	12870	24310	43758	75582	125970	203490	319770	490314	735471	1081575
9	1	10	55	220	715	2002	5005	11440	24310	48620	92378	167960	293930	497420	817190	1307504	2042975	3124550
10	0	9	63	282	996	2997	8001	19440	43749	92368	184745	352704	646633	1144052	1961241	3268744	5311718	8436267
11	0	8	69	348	1340	4332	12327	31760	75501	167860	352595	705288	1351909	2495948	4457175	7725904	13037606	21473856
12	0	7	73	415	1745	6062	18368	50100	125565	293380	645920	1351142	2702973	5198830	9655900	17381684	30419154	51892857
13	0	6	75	480	2205	8232	26544	76560	202005	495220	1140920	2491776	5194385	10392760	20048100	37429104	67847442	119739330
14	0	5	75	540	2710	10872	37290	113640	315315	810040	1950245	4441020	9634040	20024980	40070700	77496744	145340310	265074795
15	0	4	73	592	3246	13992	51030	164208	478731	1287484	3235727	7673744	17303416	37322208	77384340	154869456	300194262	565248708
16	0	3	69	633	3795	17577	68145	231429	708444	1992925	5223647	12889383	30180423	67484067	144841275	299671971	599811969	1164986064
17	0	2	63	660	4335	21582	88935	318648	1023660	3010150	8222357	21092292	51240891	118674570	263438325	562994016	1162635441	2327376348
18	0	1	55	670	4840	25927	113575	429220	1446445	4443725	12641772	33690306	84855615	203404215	466639050	1029313296	2191458423	4518099300
19	0	0	45	660	5280	30492	142065	566280	2001285	6420700	19013852	52611780	137299435	340409720	806551350	1835047456	4025198375	8541254700
20	0	0	36	633	5631	35127	174195	732474	2714319	9091270	28012754	80439789	217386520	557149607	1362556905	3195643120	7217572751	15753515733
21	0	0	28	592	5875	39662	209525	929672	3612231	12628000	40472894	120560088	337241320	893039018	2253099975	5444285920	12654132767	28394610894
22	0	0	21	540	6000	43917	247380	1158684	4720815	17223250	57402764	177316932	513207110	1403543155	3651444300	9086074320	21722825403	
23	0	0	15	480	6000	47712	286860	1419000	6063255	23084500	79992044	256168056	766883390	2165232160	5806283700	14872309920	365577062	
24	0	0	10	415	5875	50877	326865	1708575	7658190	30427375	109609379	363827190	1126269560	3281867680	9068126400	23900365620	603805	
25	0	0	6	348	5631	53262	366135	2023680	9517662	39466306	147788201	508379664	1626975480	4891539744	13922343936	37745325216	9797	
26	0	0	3	282	5280	54747	403305	2358840	11645073	50402935	196198211	699354228	2313440325	7174799646	21029659515	58630143456	1,5	
27	0	0	1	220	4840	55252	436975	2706880	14033305	63412580	256600641	947732512	3240080545	10363639300	31274624245	89641329376		
28	0	0	0	165	4335	54747	465795	3059100	16663185	78629320	330786236	1265876976	4472267215	14751050900	45822270930	1,34997E+11	3	
29	0	0	0	120	3795	53262	488565	3405600	19502505	96130540	420496076	1667359200	6087014635	20700766100	66182627310	2,00373E+11	5,77885E+11	
30	0	0	0	84	3246	50877	504315	3735720	22505751	115921972	527326778	2166673224	8173248070	28656627650	94282105353	2,93293E+11	8,67982E+11	2,4
31	0	0	0	56	2710	47712	512365	4038560	25614639	137924380	652623158	2778823488	10831511470	39150897800	1,3254E+11	4,23579E+11	1,28612E+12	3,7
32	0	0	0	35	2205	43917	512365	4303545	28759500	161963065	797362973	3518783697	14172978235	52810668925	1,83947E+11	6,03875E+11	1,88091E+12	5,58
33	0	0	0	20	1745	39662	504315	4521000	31861500	187761310	962039783	4400831436	18317641615	70361427150	2,52143E+11	8,50212E+11	2,71625E+12	8,268
34	0	0	0	10	1340	35127	488565	4682700	34835625	214938745	1146551153	5437773210	23391587635	92626745225	3,41488E+11	1,18263E+12	3,87498E+12	1,208
35	0	0	0	4	996	30492	465795	4782360	37594305	243015388	1350100235	6640085244	29523293215	1,20523E+11	4,5712E+11	1,62583E+12	5,46306E+12	1,74482E+13
36	0	0	0	1	715	25927	436975	4816030	40051495	271421810	1571119110	8015006143	36838945130	1,55049E+11	6,04993E+11	2,20979E+12	7,61423E+12	2,49061E+13
37	0	0	0	0	495	21582	403305	4782360	42126975	299515480	1807222010	9565627512	45456840130	1,97265E+11	7,91895E+11	2,97041E+12	1,0495E+13	3,51557E+13
38	0	0	0	0	330	17577	366135	4682700	43750575	326602870	2055195560	11290036836	55480999990	2,48274E+11	1,02542E+12	3,95001E+12	1,431E+13	4,90864E+13
39	0	0	0	0	210	13992	326865	4521000	44865975	351966340	2311031360	13180572120	66994212910	3,09181E+11	1,3139E+12	5,19773E+12	1,93074E+13	6,78159E+13

Konkreetne valem igas lahtris on sellest ühe võrra vasakul asuva kuni kümne lahtri väärtuste summa. Näiteks real 14 ja veerus 5 on arv 2710. See on saadud summeerides veerus 4 lahtrid ridadel 5 kuni 14. Arvutuse loogika on järgmine: kuni 5-kohalise arvu saamiseks, mille ristsumma on 14, on 10 võimalust:

- Esimene number on 0 ja lisanduvad kõik neljakohalised arvud ristsummaga 14.
- Esimene number on 1 ja lisanduvad kõik neljakohalised arvud ristsummaga 13.
- …
- Esimene number on 9 ja lisanduvad kõik neljakohalised arvud ristsummaga 5.

5.5.4 Tabeli koostamine programselt

Siin on kood samasuguse tabeli moodustamiseks:

```
int tabel[100][100];

tabel[0][0] = 1;
for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 90; j++) {
        for (int k = 0; k < 10; k++) {
            tabel[i + 1][j + k] += tabel[i][j];
        }
    }
}
```

5.5.5 Ülesande lahendus (tabeli kasutamine)

Tabelist saab vaadata, mitu antud ristsummaga arvu on n-kohaliste arvude seas. Ülesandes aga tahetakse teada, kui palju on antud arvust väiksemaid, sama ristsummaga arve. Selle leidmiseks sobib järgmine rekursiivne funktsioon:

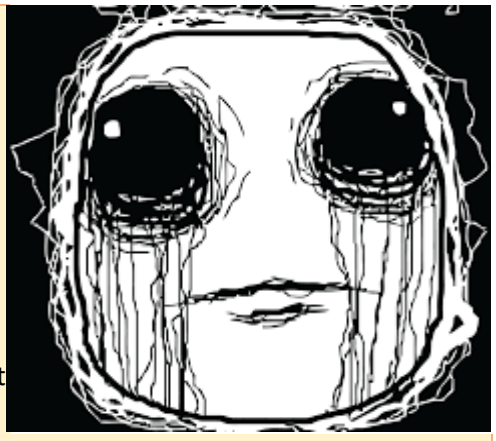
```
int loenda(int n, int ristsumma, int jark)
{
    if (n == 0)
        return 0;
    int esimene = n / pow(10, jark); //leiame esimese numbri arvus
    if (ristsumma < esimene)
        return 0;
    int aste = rint(pow(10, jark)); //n % aste annab ülejäänud kohad arvus
    int vastus = loenda(n % aste, ristsumma - esimene, jark - 1);
    for (int i = 0; i < esimene; i++) {
        vastus += tabel[jark][ristsumma - i];
    }
    return vastus;
}
```

Tabelist õige info leidmine sarnaneb tabeli koostamise põhimõttele. Näiteks kui N = 1234, on vaja leida kõik 1234-st väiksemad arvud, mille ristsumma on 1 + 2 + 3 + 4 = 10. Loomulikult on kõik sellised arvud need, kus kohtade arv väiksem kui 4 ning mille ristsumma on 10. Nende arv on tabelis 3. veerus 11. reas. Kuid lisaks sobivad ka kõik neljakohalised arvud, mis on väiksemad kui 1234. Nende leidmiseks saab kasutada tabeli koostamiseks kasutatud loogikat: 1234-st väiksemad neljakohalised arvud, mille ristsumma on 10, peavad algama 1-ga ning ülejäänud kohtade ristsumma peab olema 9 ning neist moodustatud arv peab olema väiksem kui 234. Sellisteks arvudeks on kõik kahekohalised arvud, mille ristsumma on 9 (tabelis 2. veerus 10. reas), ning need ühega algavad arvud, mille ülejäänud numbrite summa on 9 - 1 = 8 (tabelis 2. veerus 9. reas) ning need 2-ga algavad arvud, mille ülejäänud numbrite summa on 7 ning mis ei ole suuremad kui 34. See moodustub siis kõikidest ühekohalistest arvudest, mille ristsumma on 7 (ehk siis ainult 7-st); kõigist kahekohalistest arvudest, mille esimene number on 1 ja ristsumma 6 [16]; kõigist kolmekohalistest arvudest, mille esimene number on 2 ja ristsumma 5 [25]. Siin on tabel funktsiooni „Loenda“ väljakutsetega ning tagastatava vastuse kujunemisega:

Väljakutse	Lisatavad tabeli kohad	Vastus	Näide
Loenda(1234, 10, 4)	-	85	
Loenda(1234, 10, 3)	tabel[3][10] = 63	85	Kõik kuni neljakohalised ja 1009, 1018, 1027, 1036, 1045, 1054, 1063, 1072, 1081,1090 ja kolmekohalised: 1108, 1117, 1126, 1135, 1144, 1153, 1162, 1171, 1180 1207,1216, 1225
Loenda(234, 9, 2)	tabel[2][9] = 10 tabel[2][8] = 9	22	Kõik vähem kui kolmekohalised: 009, 018, 027, 036, 045, 054, 063, 072, 081, 090 ja kolmekohalised: 108, 117, 126, 135, 144, 153, 162, 171, 180 Lisaks: 207, 216, 225
Loenda(34, 7, 1)	tabel[1][7] = 1 tabel[1][8] = 1 tabel[1][9] = 1	3	07, 16 ja 25
Loenda(4, 4, 0)	tabel[0][4] = 0 tabel[0][3] = 0 tabel[0][2] = 0 tabel[0][1] = 0	0	-

5.6 MITMEMÕÕTMELINE DP TABEL – MILJONÄR JA VAESLAPSED

Järgmine ülesanne on Eesti lahtiselt programmeerimisvõistluselt aastast 2015. See on kõigi aegade kõige kurvema tekstiga programmeerimisülesanne, aga selle lahendus on üle kantav paljudele teistele statistilise modelleerimise ülesannetele.



Dickensi-aegsel Inglismaal elas miljonär Mortimer. Temaga samas linnas asusid kolm lastekodu, kus elasid vaeslapsed, kellele Mortimer tavatses jõulukinke teha. Kinkide jagamise protseduur oli järgmine:
1. Iga vaeslaps saab oma lastekodust korvi, millega ta kingi järele läheb.
2. Mortimer viskab kingitusi järjest laste sekka, mida nood oma korvidega püüavad.
3. Iga kingitus püütakse alati kinni.
4. Lapsed saavad kingitusi kätte juhuslikult, kuid tõenäosus, et konkreetne laps kingituse kätte saab, on võrdeline tema korvisuu pindalaga.
5. Sama lastekodu lastel on sama suurusega korvid.
6. Kui mõni laps saab kingituse kätte, läheb ta sellega kohe lastekodusse tagasi ja rohkem püüdmises ei osale.
7. Lapsi võib olla rohkem kui kingitusi :(
Igal kingitusel on väärtus. Leida iga lastekodu kohta, milline on selle kodu laste poolt saadud kingituste väärtuste keskmine eeldatav summa.

NÄIDE:
Esimesest lastekodust on kohal $L_1 = 1$ laps korvi pindalaga $K_1 = 1$.
Teisest lastekodust on kohal $L_2 = 1$ laps korvi pindalaga $K_2 = 2$.
Kolmandast lastekodust on kohal $L_3 = 1$ laps korvi pindalaga $K_3 = 3$.
Mortimeril on `N = 2` kingitust väärtustega `V = [10, 20]`.
Vastus:
L_1: 6,666667
L_2: 11,333333
L_3: 12

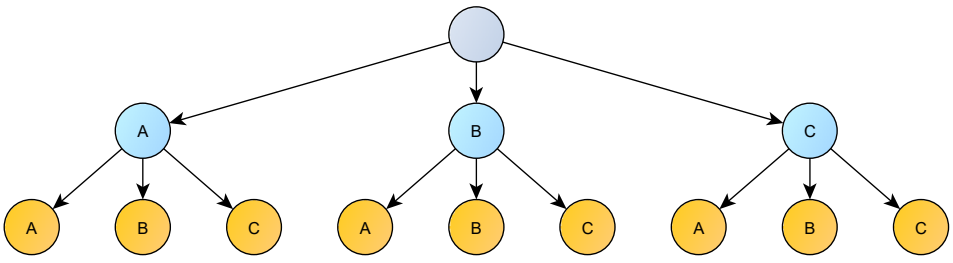
Näiteandmeid kasutades kujuneb vastus esimese lastekodu kohta järgnevalt:

- Esimese lastekodu ainus laps saab esimese kingituse kätte tõenäosusega $\frac{1}{6}$, see annab oodatavaks väärtuseks $10 * \frac{1}{6} = \frac{5}{3} \approx 1{,}666667$ ning teiste kingituste püüdmises ta ei osaleks.
- Tõenäosusega $\frac{1}{3}$ saab esimese kingi teise lastekodu laps ja läheb ära. Sel juhul on esimesel lapsel teise kingi saamise tõenäosus $\frac{1}{4}$. Oodatav väärtus on siis $20 * \frac{1}{4} * \frac{1}{3} = \frac{5}{3} \approx 1{,}666667$.
- Tõenäosusega $\frac{1}{2}$ saab esimese kingi kolmanda lastekodu laps ja lahkub. Sel juhul on esimesel lapsel teise kingi saamise tõenäosus $\frac{1}{3}$, oodatav väärtus $20 * \frac{1}{2} * \frac{1}{3} = \frac{10}{3} \approx 3{,}333333$.

Kokku tulebki vastuseks $\frac{5}{3} + \frac{5}{3} + \frac{10}{3} = \frac{20}{3} \approx 6{,}666667$. Samasugust arutlust saab kasutada ka teiste lastekodude jaoks.

5.6.1 Kõikide võimaluste läbivaatus

Tõenäosus, et kingi saab kätte mingi konkreetse lastekodu laps, sõltub kogu selle lastekodu püüdmisel osalevate korvide pindala suhtest kõigi veel püüdmisel osalevate korvide pindalasse. Iga kord, kui kink kätte saadakse, väheneb ühe lastekodu korvide kogupindala, kuna kingi saanud laps lahkub koos oma korviga. Seetõttu väheneb ka kõigi korvide pindala. Kinkide püüdmisele vastab järgmine puu:

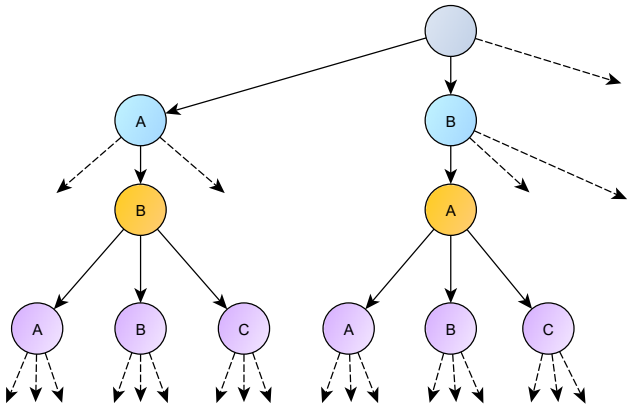


A, B ja C tähistavad erinevaid lastekodusid, esimene tase (sinised tipud) esimest kingitust, teine tase (oranžid tipud) teist kingitust. Igas tipus on sellele lastekodule vastav täht, kes vastava taseme kingituse sai. Kõikide variantide läbivaatamise keerukus sõltub kinkide arvust N ja on $O(3^N)$.

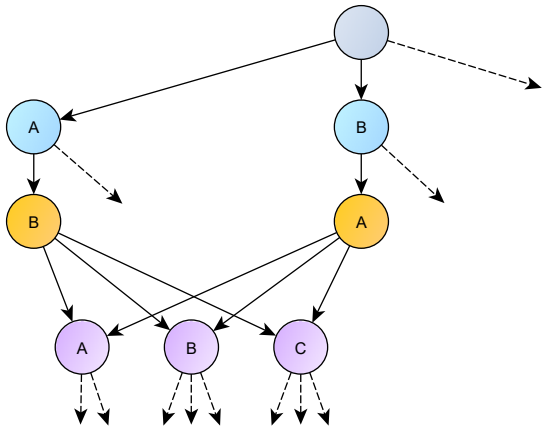
5.6.2 Korduvad harud

Kas läbivaatuste puus hakkavad mingid osad korduma? Lapsed saavad ükshaaval kinke ning lahkuvad seejärel. i-nda kingi jagamise ajaks on lahkunud i last ja lastekodude ning ka kogu korvide pindala on

vähenenud vastavalt sellele, milliste lastekodude lapsed olid juba lahkunud. Samas ei sõltu allesjäänud laste ja korvide arv sellest, millises järjekorras eelmised lapsed oma kinke said. Siin on alamosa sügavamast rekursioonipuust:



Ülesande seisukohast on pärast oranži taset toimuv kinkide jagamine täpselt sama:



5.6.3 DP tabeli koostamine

Kui kõik kingid on jagatud, siis kingi saamise tõenäosus on 0. Kui on alles viimane kink, siis tõenäosus, et selle saab mõni esimese lastekodu laps, on esimese lastekodu veel kohale jäänud laste arv LJ_1 korrutatud esimese lastekodu korvi pindalaga K_1 ning jagatud kõikide allesjäänud korvide pindalaga. Et seda leida, on vaja teada, mitu last igast lastekodust alles on. Hea on koostada kolmemõõtmeline massiiv, kus iga mõõde vastab ühe lastekodu kohal olevate laste arvule. Näiteks, kui on vaja jagada 5 kinki ja lastekodudest on kohal vastavalt 2, 5 ja 1 last, siis viimase kingi jagamise ajaks võib olla esimesest lastekodust alles 0 kuni 2 last, teisest 1 kuni 4 last ja kolmandast 0 või 1 last. Kokku on järel 2 + 5 + 1 - 4 = 4 last. Järgnevas tabelis on toodud viimase kingi jagamisest saadavad keskmised eeldatavad summad esimese lastekodu jaoks:

A	B C	1	2	3	4	5
0	0	-	-	-	0	-
	1	-	-	0	-	-
1	0	-	-	$\frac{K_1}{K_1 + 3 * K_2} * V$	-	-
	1	-	$\frac{K_1}{K_1 + 2 * K_2 + \mathrm{K}_3} * V$	-	-	-
2	0	-	$\frac{2 * \mathrm{K}_1}{2 * K_1 + 2 * K_2} * V$	-	-	-
	1	$\frac{2 * K_1}{2 * K_1 + K_2 + K_3} * V$	-	-	-	-

A, B ja C tähistavad vastavalt esimesest, teisest ja kolmandast lastekodust alles jäänud laste arvu. K_1, K_2 ja K_3 tähistavad vastavalt lastekodude korvi pindalasid ning V viimase kingituse väärtust.

Kuidas leida, milline on oodatav kinkide väärtus üks käik enne seda, näiteks juhul, kui alles on kaks last esimesest, kaks teisest ja üks kolmandast lastekodust? Siis on 3 võimalust, millise lastekodu laps kingi saab:

1. Kingi saab esimese lastekodu laps ning pärast seda on situatsioon, kus lapsi on järel A = 1, B = 2 ja C = 1 (tabelis oranži taustaga lahter).
2. Kingi saab teise lastekodu laps ning pärast seda on lapsi järel A = 2, B = 1 ja C = 1 (tabelis rohelise taustaga lahter).
3. Kingi saab kolmanda lastekodu laps ning pärast seda on lapsi A = 2, B = 2 ja C = 0 (tabelis sinise taustaga lahter).

Tähistagu kingi saamise tõenäosusi antud sammul vastavalt T_A, T_B ja T_C ja parasjagu jagatava kingituse väärtust VP, siis

$$E(A_i, B_{j,} C_k) = \begin{cases} 0, kui\ i = 0, j = 0, k = 0 \\ T_A * VP\ +\ T_A * E(A_{i-1}, B_j, C_k)\ +\ T_B * E(A_i, B_{j-1}, C_k)\ +\ T_C * E(A_i, B_j, C_{k-1}) \end{cases}$$

Ülesandes tahetakse vastust kõigi kolme lastekodu jaoks, seega tuleb teha 3 tabelit: üks iga lastekodu jaoks.

5.6.4 Lahendus

Siin on funktsioon, mis antud ülesande lahendab:

```
int kingid(int La, int Lb, int Lc, int Ka, int Kb, int Kc, int N, int* kingid)
{
    double**** kv = new double***[La + 1]; //1. lastekodust kohalolevate laste arv
    for (int i = 0; i <= La; i++) {
        kv[i] = new double**[Lb + 1]; //2. lastekodust kohalolevate laste arv
        for (int j = 0; j <= Lb; j++) {
            kv[i][j] = new double*[Lc + 1]; //3. lastekodust kohalolevate laste arv
            for (int k = 0; k <= Lc; k++) {
                kv[i][j][k] = new double[3]; // lastekodu
            }
        }
    }

    int lapsiKokku = La + Lb + Lc;
    //Alustame sellest, kui kõik lapsed on lahkunud kuni selleni, kui kõik veel alles
    for (int al = lapsiKokku - N; al <= lapsiKokku; al++) {
        int m1 = min(La, al);//nii palju saab maksimaalselt olla alles 1. lastekodust
        for (int i = 0; i <= m1; i++) {
            //nii palju saab olla 2. lastekodust, kui esimesest on alles i last:
            int m2 = min(Lb, al - i);
            for (int j = 0; j <= m2; j++) {
                if (al - i - j <= Lc) {
                    int k = al - i - j; //nii palju on 3. lastekodust kohal
                    kv[i][j][k][0] = 0;
                    kv[i][j][k][1] = 0;
                    kv[i][j][k][2] = 0;
                    if (al == lapsiKokku - N) {
                        continue; //kedagi ei ole, jätkame
                    }
                    double p1 = i * Ka;
                    double p2 = j * Kb;
                    double p3 = k * Kc;
                    double kp = p1 + p2 + p3; //alles korvide kogupindala
                    double t1 = p1 / kp; //Tõenäosus, et kingi saab 1. lastekodu
                    double t2 = p2 / kp; //2. lastekodu
                    double t3 = p3 / kp; //3. lastekodu
                    int jk = kingid[lapsiKokku - al]; //jagatava kingi väärtus

                    if (i != 0) {
                        kv[i][j][k][0] += t1 * (jk + kv[i - 1][j][k][0]);
                        kv[i][j][k][1] += t1 * kv[i - 1][j][k][1];
                        kv[i][j][k][2] += t1 * kv[i - 1][j][k][2];
                    }
                    if (j != 0) {
                        kv[i][j][k][0] += t2 * kv[i][j - 1][k][0];
                        kv[i][j][k][1] += t2 * (jk + kv[i][j - 1][k][1]);
                        kv[i][j][k][2] += t2 * kv[i][j - 1][k][2];
                    }
                    if (k != 0) {
                        kv[i][j][k][0] += t3 * kv[i][j][k - 1][0];
                        kv[i][j][k][1] += t3 * kv[i][j][k - 1][1];
                        kv[i][j][k][2] += t3 * (jk + kv[i][j][k - 1][2]);
                    }
                }
            }
        }
    }

    cout << fixed << setprecision(10) << kv[La][Lb][Lc][0] << endl <<
        kv[La][Lb][Lc][1] << endl << kv[La][Lb][Lc][2] << endl;
    return 0;
}
```

Selles lahenduses on DP tabeli dimensioonideks kasutatud küll lastekodulaste arvu, kuid tegelikud arvutused toimuvad ainult laste arvu ja kinkide arvu vahe osas (täpsemalt võiks mõõtmeteks olla nt lahkunud laste arv, mis saab maksimaalselt võrduda kinkide arvuga). Seega on kogu selle lahenduse keerukus $O(N^3)$.

5.7 TÕEVÄÄRTUSTE TABELIGA DP - ÕIGLANE JAGAMINE

See ülesanne on teisend klassikalisest seljakoti pakkimise ülesandest. Seljakoti pakkimisest tuleb hiljem rohkem juttu kaheksandas peatükis – DP edasijõudnutele.

Kaks venda, Albert ja Benno, tahavad jagada omavahel hulka kingitusi. Iga kingituse peab andma kas Albertile või Bennole; kingitusi poolitada ei saa. Igal kingitusel on väärtus. A ja B tähistavad vastavalt Albertile ja Bennole antud kingituste kogusummat. Minimeerida vahe A - B absoluutväärtus. Kingituste arv N ei ületa 100. Kingituste väärtused on positiivsed täisarvud, mis ei ületa 200.
NÄIDE:
```
N = 4
V = [2, 3, 4, 7]
```
Vastus: 2 (näiteks jaotused [2, 7] ja [3, 4] või [7] ja [2, 3, 4])

5.7.1 Kõik kombinatsioonid

Esimese hooga tuleb mõte vaadata läbi kõik kombinatsioonid. Kuna kinkide jagamisel tekkiv vahe on sümmeetriline (st ei ole vahet, kas konkreetse komplekti kingitusi saab Albert ja ülejäänud jääb Bennole või vastupidi), siis piisab pooltest võimalustest. Näiteandmete jaoks on need toodud järgmises tabelis:

Alberti kinkide väärtused	2, 3, 4, 7	3, 4, 7	2, 4, 7	2, 3, 7	2, 3, 4	4, 7	3, 7	3, 4
Benno kinkide väärtused	0	2	3	4	7	2, 3	2, 4	2, 7
Vahe	16	12	10	8	2	6	4	2

Tundub hea plaan. Erinevaid kombinatsioone kinkide jagamiseks on 2^{N-1}. Kui N on väike, nagu antud näites, siis töötab kõik suurepäraselt, kuid kui N = 100, teeb see juba 2^{99} erinevat kombinatsiooni ja see on isegi masinale liiga palju tööd.

5.7.2 DP lahendus

Eelmises tabelis on näha, et üks kinkide jaotamisel tekkinud väärtuste vahe kordub. Kuna kinkide lubatud väärtus on kuni 200, siis nii väikese kinkide arvu juures nagu näites (N = 4) see ei pruugi ilmtingimata nii olla, kuid suurema arvu kinkide jagamisel ilmselt peavad hakkama vahed korduma, sest kõigi võimalike vahede arv on fikseeritud. Näiteks maksimumsisendi juures, kus kinke on 100 ja igaüks väärtusega 200, siis juhul kui Albert ahnitseb kõik kingitused endale, on Alberti kingituste koguväärtus kõigi kingituste väärtuste summa, mis on maksimaalselt 100 * 200 = 200 000. Benno kingituste koguväärtus on loomulikult 0, seega maksimaalne vahe saab olla 200 000. See tähendab aga, et nende kingituste ükskõik millisel jagamisel ei saa tekkida rohkem kui 200 001 erinevat vahet (0 sobib ka). Pane tähele: see kehtib seetõttu, et kingituste väärtused ja seega ka vahed saavad olla ainult täisarvud, reaalarvuliste väärtuste korral saab sellist lähenemist kasutada ainult teatud tingimustel.

Näites on kingituste kogusumma 2 + 3 + 4 + 7 = 16, järelikult võimalikud vahed on 0…16. Kuidas muutuvad vahed kingituste jagamise käigus?

Kui jagatakse ainult üks kingitus, saab vaheks olla ainult selle kingituse väärtus (tabelis '+' tähistab võimalikku vahet, '-' võimatut. Reas on kingituse nr(= väärtus)):

Vahe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Kui jagatakse järgmine kingitus, siis kõik võimalikud vahed saavad kas kasvada või kahaneda täpselt selle kingituse väärtuse võrra, nt kui esimese kingituse väärtusega 2 sai Albert ja teise, väärtusega 3, saab samuti Albert, on vahe 2 + 3 = 5. Kui teine kink läheb aga Bennole, on vahe |2 – 3| = 1.

Vahe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-

Kolmanda kingituse jagamisel saab vahe muutuda 4 võrra, sest see on kolmanda kingituse väärtuseks. Kui enne sai olla väärtuste vahe kas 1 või 5, siis nüüd on võimalikud vahed:

- 1 + 4 = 5,
- |1 – 4| = 3,
- 5 + 4 = 9 ja
- 5 – 4 = 1.

Samamoodi saab arvutada kõik võimalikud vahed ka viimase kingituse jaoks. Siin on kogu täidetud tabel näites toodud 4 kingi jaoks:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-
3	-	+	-	+	-	+	-	-	-	+	-	-	-	-	-	-	-
4	-	-	+	-	+	-	+	-	+	-	+	-	+	-	-	-	+

Siin on ka võimalus kokkuhoidliku DP kasutamiseks: iga kord kasutatakse järgmise rea arvutamiseks ainult eelmisel real olevat infot – see tähendab, et kui ülesande vastuseks soovitakse ainult minimaalset vahet, piisab tegelikult kahe reaga tabelist: vaja on meeles pidada eelmine rida ning käesolev rida.

Järgmine funktsioon illustreerib kokkuhoidlikku DP lahendust:

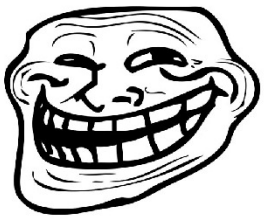
```
int LeiaVahe(int mituPakki, int* pakid)
{
    int suurimVahe = 0;
    for (int i = 0; i < mituPakki; i++) {
        suurimVahe += pakid[i]; //Leiame suurima võimaliku vahe
    }
    bool** tabel = new bool*[suurimVahe + 1];
    for (int i = 0; i <= suurimVahe; i++) {
        tabel[i] = new bool[2];//kokkuhoidliku DP jaoks on vaja vaid 2 rida
    }
    tabel[0][1] = 1; //kui midagi jagada pole, on vahe 0
    for (int i = 1; i <= suurimVahe; i++) {
        tabel[i][1] = 0; //ülejäänud vahed on võimatud
    }
    for (int i = 0; i < mituPakki; i++) {
        int praeguneRida = i % 2;
        int eelmineRida = (i + 1) % 2;
        for (int j = 0; j < suurimVahe; j++) {
            int eelmineVahe = j - pakid[i];
            if (eelmineVahe < 0){
                eelmineVahe = -eelmineVahe;
            }
            if (tabel[eelmineVahe][eelmineRida]) {
                tabel[j][praeguneRida] = 1;
                continue;
            }
            eelmineVahe = j + pakid[i];
            if (eelmineVahe > suurimVahe || !tabel[eelmineVahe][eelmineRida]) {
                tabel[j][praeguneRida] = 0;
                continue;
            }
            tabel[j][praeguneRida] = 1;
        }
    }
    int vastus;
    for (int i = 0; i < suurimVahe; i++) {
        int praeguneRida = (mituPakki - 1) % 2;
        if (tabel[i][praeguneRida]) {
            vastus = i; //leiame esimese võimaliku vahe, see ongi vastuseks
            break;
        }
    }
    return vastus;
}
```

5.8 BITIMASKIDE PÕHINE VÄÄRTUSTE TABEL - MOEKUNSTNIK JA KOILIBLIKAS

Järgmine ülesanne on samuti Eesti lahtiselt programmeerimisvõistluselt aastast 2013. Ülesande originaaltekst kuulub selgelt fantastika valdkonda, kuid samal põhimõttel saab lahendada mitmesuguseid mänguteooria või riskianalüüsi situatsioone.

Moekunstnik Märdil on kapis N ilusat salli, millega ta käib laupäeviti moekunstnike koosolekul. Igal sallil on oma moeväärtus ja koosolekul saab Märt vastava hulga feimi. Kaks korda sama salliga kohale tulla oleks suur *faux pas* ja Märt ei tee seda kunagi. Kantud sallid paneb ta kappi tagasi, aga rohkem neid ei kanna. Märdi kapis elab ka koiliblikas Kärt, kes sööb igal pühapäeval ühele sallile augu sisse. Auguga salli enam kanda ei saa. Kärt moeväärtusest ei hooli, vaid sööb salle juhuslikult. Mingile sallile augu söömise tõenäosus on võrdeline salli pikkusega. Kärt võib sama salli süüa ka mitu korda. On selge, et Märt saaks koosolekul käia ülimalt N korda, kui Kärt sööks ainult juba kantud salle. Kui Kärt sööb mõnikord ka kandmata salle, jääb koosolekute arv sellevõrra väiksemaks. Kirjutada programm, mis leiab, kui palju Märt keskmiselt feimi saab, kui ta kasutab parimat võimalikku strateegiat, aga Kärt sööb salle juhuslikult. Tegevus algab nädala alguses, seega esimese koosoleku ajaks on kõik sallid veel terved. Sisendina on antud kõigi sallide pikkused ja moeväärtused.
Sisendi esimesel real on sallide moeväärtused ja teisel real nende pikkused.
NÄIDE:
Pikkused: `3 2 1`
Moeväärtused: `10 20 30`
Vastus:
48,333333

(Pildil Märt, Kärt ja ülesande autor Targo).



Näite selgitus: Antud näites on Märdi optimaalne strateegia võtta kõigepealt teine sall. Pärast seda on:

- tõenäosusega $\frac{1}{2}$ auk esimeses sallis, teiseks koosolekuks alles ainult kolmas sall ja saame kokku 50 feimi;
- tõenäosusega $\frac{1}{3}$ auk teises sallis ja kaks salli alles; võtame neist kõigepealt kolmanda ning $\frac{1}{2}$ tõenäosusega saame ka esimese ära kanda; keskmiselt kokku 55 feimi; tõenäosusega $\frac{1}{6}$ auk kolmandas sallis, alles ainult esimene ja saame kokku 30 feimi.

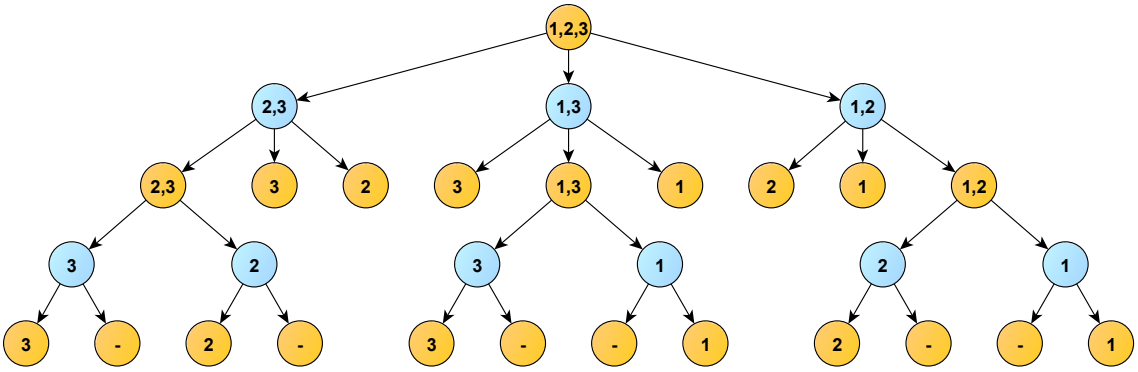
Nende variantide kaalutud keskmine on $48\frac{1}{3}$ feimi.

5.8.1 Kõigi läbivaatuste puu

Nagu ikka, alustame jõumeetodi analüüsiga.

Alguses on Märdil valida kõikide sallide vahel. Ta valib neist ühe ning järgmiseks korraks jääb N - 1 valikut. Nüüd sööb Kärt ühele sallile augu sisse. Kui Kärt valis einestamiseks juba kantud salli, siis

Märdi valikud ei muutu. Kui aga Kärt sõi mõne kandmata salli, jääb Märdile N - 2 valikut jne. Siin on puu kolme salli jaoks:



Numbrid tippudes näitavad, millised sallid on veel Märdile kasutatavad. Oranžides ringides on Märdi käikudele, sinistes Kärdi käikudele eelnev seis.

Märt saab igal sammul valida järelejäänud sallide hulgast – Kärt valib küll kõigi sallide hulgast, kuid tegelikult pole ülesande seisukohalt oluline, millise katkistest või kantud sallidest Kärt valib, neid võib vaadelda ühe juhuna. Sellisel juhul kõikide variantide läbivaatamisega lahenduse keerukus on $O((N!)^2)$. Seega juba N = 20 juures tuleb teha $(20!)^2 = 5{,}919*10^{36}$ läbivaatust. Arvestades 10^8 läbivaatust sekundis, kulub $2*10^{21}$ aastat!

5.8.2 Tee DP poole

Kui vaadata hoolega eespool joonistatud puud, pole raske märgata, et puus hakkavad harud korduma. Sisuliselt tähendab see seda, et kui mingil hetkel on Märdil valida täpselt samade sallide vahel, siis on tal tark täpselt samamoodi käituda ning ta saab keskmiselt samapalju feimi juurde eelnevalt kogutule: nii näiteks kolmanda päeva valik ei sõltu enam sellest, mida Märt ja Kärt esimesel kahel päeval ette võtsid, vaid ainult sellest, millised sallid veel selleks ajaks järel on. Piisab, kui samasuguseid alampuid arvutame vaid ühel korral, edaspidi võib kasutada meelde jäetud väärtust. Seega võiks mõte minna DP lahenduse peale. Erinevaid seise on sama palju kui parajasti alles olevate sallide võimalikke kombinatsioone ehk 2^N. Iga kombinatsiooni kohta on vaja meeles pidada parim tulemus. Kuidas aga järjestada kombinatsioone?

5.8.3 Bitimaskide kasutamine tabeli indeksitena

Siin on olukord, kus süsteemi võimalikud olekud on kirjeldatavad bitimaskidena: mingi hulk objekte on parajasti kas aktiivsed või mitteaktiivsed. N objekti puhul on selliseid võimalusi 2^N. Sellisel juhul on kaval luua 2^N elemendiga massiiv, kus iga element tähistab üht võimalikku kombinatsiooni. Kui massiivi indeksid on 0 kuni $2^N - 1$, annab see meile ka loomuliku viisi kombinatsioonide järjestamiseks: iga massiivi element tähistab parajasti nende objektide olemasolu, millele vastavad bitid on indeksi kahendesituses ühed.

Näiteks nelja objekti puhul tähendaks see järgmist tabelit:

Massiivi indeks	Kahendesitus	Allesolevad objektid
0	0000	Mitte ühtegi
1	0001	1.
2	0010	2.
3	0011	1. ja 2.
4	0100	3.
5	0101	1. ja 3.
6	0110	2. ja 3.
7	0111	1., 2. ja 3.
8	1000	4.
9	1001	1. ja 4.
10	1010	2. ja 4.
11	1011	1., 2. ja 4.
12	1100	3. ja 4.
13	1101	1., 3. ja 4.
14	1110	2., 3. ja 4.
15	1111	Kõik objektid

Madalamad bitid tähistavad esimesi objekte, sest siis me ei piira objektide koguarvu – kui neid on rohkem, saab võtta lihtsalt pikema arvu.

Tavapäraste operatsioonide sooritamiseks on kasulikud bitikaupa tehted:

Operatsioon	Operaator	Näide kahendesituses	Näide kümnendesituses
Bittide nihutamine vasakule. M << N on ekvivalentne tehtega $M*2^N$.	<<	0101 << 2 == 010100	5 << 2 == 20
Bitikaupa korrutamine	&	0101 & 0111 == 0001	5 & 7 == 1
Bitikaupa liitmine	|	0101 | 0011 == 0111	5 | 3 == 7
Bitikaupa eitus	~	~0101 == 1010	Vastus oleneb arvu pikkusest
Bitikaupa välistav või	^	0101 ^ 0011 == 0110	5 ^ 3 == 6

Nende abil saab teha järgmisi operatsioone väga efektiivselt:

- Kõigi kombinatsioonide arvu (2^N) leidmine.

```
1 << N
```

- Konkreetsele objektile vastava arvu leidmine. Tehe tagastab kahendarvu, kus altpoolt m. bitt on 1 ja teised bitid nullid.

```
1 << (m - 1)
```

- Kombinatsiooni arvu leidmine, kus on olemas kõik objektid peale ühe. Tegemist on eelmise tehte tulemuse bittide ümberpööramisega.

```
~(1 << (m - 1))
```

- Tsükkel, mis genereerib üksikutele objektidele vastavaid kahendarve (sisuliselt kahe astmeid):

```
for (int mbit = 1; mbit <= max; mbit *= 2)
```

- Kontroll, kas kombinatsioon i sisaldab objekti number m. Kui kombinatsioonile vastavat arvu bitikaupa korrutada ühele objektile vastava arvuga, on vastus positiivne parajasti siis, kui kombinatsiooni arvus on ka m. bitt seatud.

```
i & 1 << (m - 1) > 0
```

- Objekti eemaldamine kombinatsioonist (täpsemalt, kombinatsiooni i alusel sellise kombinatsiooni leidmine, kus puudub objekt m).

```
i & ~(1 << (m - 1))
```

- Objekti lisamine kombinatsioonile (täpsemalt, indeksi i alusel sellise indeksi leidmine, kuhu on lisatud objekt m):

```
i | 1 << (m - 1)
```

- Objekti sisse-väljalülitamine (lihtsustab koodi juhul, kui me juba ette teadsime, kas see objekt oli enne aktiivne või ei):

```
i ^ 1 << (m - 1)
```

Selline lähenemine on arvutuslikult üliefektiivne. Konkreetsetele objektikombinatsioonidele vastavate massiivi indeksite leidmine ja nendega opereerimine taanduvad täisarvutehetele, mille jaoks kaasaegsetes protsessorites on hästi optimiseeritud meetodid ja mille täitmist mõõdetakse üksikutes nanosekundites.

5.8.4 DP lahendus

Siin on põhjalike kommentaaridega funktsioon, mis antud ülesandele vastuse leiab:

```
double leiaFeim(int N, int* moevaartused, int* pikkused)
{
    // võimalike kombinatsioonide arv, millised sallid võivad söödud olla = 2^n
    int kombinatsioonid = 1 << N;
    int kogupikkus = 0;
    for (int i = 0; i < N; i++) {
        kogupikkus += pikkused[i];
    }
    // Siin massiivis hoiame parimaid võimalikke tulemusi, mida on võimalik saavutada,
    // kui meil on alles mingi konkreetne kombinatsioon salle.
    // Massiivi indeksid vastavad sallikombinatsioonide kahendarvulistele väärtustele.
    // Näiteks feimid[37] sisaldab maksimaalset võimalikku väärtust, mida Märdil on
    // võimalik saavutada juhul, kui alles on 1., 3. ja 6. sall (sest 10-süsteemis 37
    // on kahendsüsteemis 00100101, seatud on altpoolt lugedes 1., 3. ja 6. bitt)
    double* feimid = new double[kombinatsioonid];
    feimid[0] = 0;
```

```
    // konstrueerime kõik kombinatsioonid alates lihtsamatest
    for (int i = 1; i < kombinatsioonid; i++) {
        feimid[i] = 0;

        // Proovime uue kombinatsiooni üles ehitada väiksematest.
        // mvalik on Märdi tehtav valik, st mitmenda salli ta võtab.
        // Proovime kõiki Märdi valikuid
        for (int mvalik = 0, mbit = 1; mbit <= i; mbit *= 2, mvalik++) {
            // bitt pole seatud, st vaadeldav kombinatsioon ei sisalda seda salli
            if ((mbit & i) == 0){
                continue;
            }
            double alles = 1;  // tõenäosus, et ühtki head salli ei sööda ära
            double feim = moevaartused[mvalik];  // Märdi käigu väärtus
            // kvalik on Kärdi tehtav valik, st mitmenda salli sisse ta augu sööb
            for (int kvalik = 0, kbit = 1; kbit <= i; kbit *= 2, kvalik++) {
                if ((i & kbit) == 0 || mvalik == kvalik) continue;
                double prob = (double)pikkused[kvalik] / kogupikkus;
                // väärtusele lisandub selle allesjääva kombinatsiooni väärtus, kust
                //on eemaldatud Märdi ja Kärdi sall
                feim += prob * feimid[i ^ mbit ^ kbit];
                alles -= prob;
            }
            // Kui Kärt ei söönud ühtki head salli ära, lisandub selle kombinatsiooni
            // väärtus, kus on kõik sallid peale Märdi praeguse valiku
            feim += alles * (feimid[i ^ mbit]);
            if (feim > feimid[i]) {
                feimid[i] = feim;
            }
        }
    }
    return feimid[kombinatsioonid - 1];
}
```

5.9 KUIDAS JA MILLAL DP-D KASUTADA

Eelnevalt toodud näidetest selgus, et DP aitab algoritmi keerukust sageli dramaatiliselt parandada:

Ülesanne	Jõumeetodiga keerukus	DP keerukus
Fibonacci	$O(\varphi^N)$	O(N)
Optimaalne maksmine	$O(2^N)$	O(N)
Pikim kasvav osajada	$O(2^N)$	$O(N^2)$
Pikim ühine osajada	$O(2^{N+M})$	O(NM)
Ristsumma	O(N)	O(log N)
Õiglane jagamine	$O(2^N)$	O(N*log(W)), kus W on kingi maksimaalne väärtus
Miljonär ja vaeslapsed	$O(3^N)$	$O(N^3)$
Moekunstnik ja koiliblikas	$O(N!^2)$	$O(2^N)$

Kuidas aga ära tunda ülesandeid, kus on võimalik DP-d kasutada?

- Probleemi saab jagada väiksemateks alamprobleemideks.
- Suurema probleemi lahenduse saab leida ühe või enama alamprobleemi lahenduse põhjal.
- Alamprobleemid on järjestatavad.

- Suurema probleemi lahendus sõltub ühest või enamast alamprobleemi lahendusest, kuid mitte sellest, kuidas need lahendused täpselt saadi.

Tavalisi DP ülesannete struktuure:

Sisend	DP tabelis kasutatav(ad) väärtus(ed)	Üleminek	Näiteid
Jada $x_1, x_2, \ldots, x_n$	Jada element i	Vali jada esimesed i-1 elementi, töötle i, lisa i eelmisele seisule või mitte.	Pikim kasvav osajada
Kaks jada $x_1, x_2, \ldots, x_n$ ja $y_1, \ldots, y_m$	Jadade elementide paar i (esimesest jadast) ja j (teisest jadast)	Nagu eelmine, aga muuda üht indeksit korraga.	Pikim ühine osajada
Jada $x_1,\ldots x_i,\ldots x_j,\ldots,x_n$	Alamjada $x_i,\ldots,x_j$	Jaga lõik i,…,j lõikudeks i,…,k k+1,…,j	Maatriksite jada korrutamine (vt peatükk 8)
Suunatud tsükliteta graaf	Tipp graafis	Töötle vaadeldava tipu naabrid	Pikimate või lühimate teede leidmine graafis, teede loendamine
Piirav arvuline väärtus	Piiriväärtusest väiksem (või suurem) arv	Suurenda (või vähenda) vaadeldavat väärtust kuni jõuad piiriväärtuseni.	Seljakotiülesanne, mitmesugused maksmisülesanded
Objektide hulk	Objektide alamhulk (tavaliselt bitimaskina)	Lülita alamhulgas elemente sisse-välja	

5.9.1 Ülalt alla vs alt üles DP

	Ülalt alla	Alt üles
Plussid	- Loomulik järgmine samm täisläbivaatusega lahendusest. - Leiab alamvastused ainult siis, kui see on vajalik (mõnedes olukordades kiirem).	- Kiirem juhul kui paljusid alamprobleeme „külastatakse“ korduvalt - Võimaldab mälu kokkuhoidu, kui kasutame „kokkuhoidliku DP“ trikki.
Miinused	- Rekursiivsel funktsioonide väljakutsumisel on lisakulu. - Kõigi olekute meelespidamine võib põhjustada mäluprobleeme.	- Lugejale, kes pole DP-ga tuttav, on koodi intuitiivne mõistmine raskem. - Leiab kõik alamvastused ka juhul, kui see pole otseselt vajalik.

5.9.2 Kidunud DP

Vahel on võimalik leida DP ka olukordades, kus seda esmapilgul ei ole. Olgu ülesandeks näiteks:

Leida arvude jadas maksimaalne element.

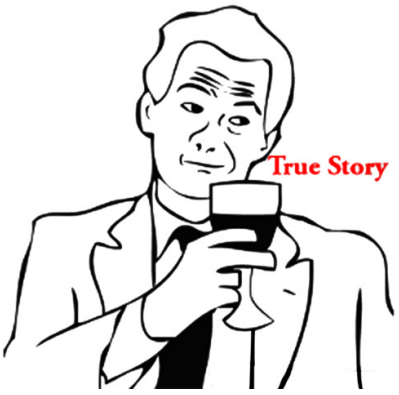
Loogiline ja intuitiivne lahendus on meeles pidada, mis on seni leitud maksimaalne element, võrrelda seda kõigi järgmiste elementidega ning kui leiame suurema, siis meeles peetud element selle suurema väärtuse vastu vahetada. Samas võib ka sellest algoritmist mõelda kui DP algoritmist, kus „tabel“ koosnebki sellestsamast seni leitud maksimaalsest elemendist. Alternatiivne „jõumeetodiga“ lahendus samal ülesandele oleks iga elemendi jaoks proovida, kas ta on kõigist teistest suurem või ei: $O(N^2)$ ajalise keerukusega algoritm.

5.10 DP PRAKTILISED KASUTUSALAD

Me kasutame sageli tööriistu, mis sisaldavad endas ühte või teist DP-l põhinevat algoritmi.

Mõned huvitavad DP kasutusalad:

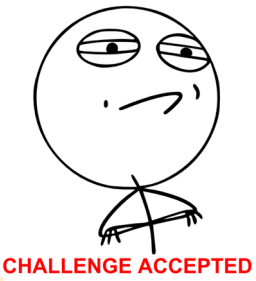
- diff (https://en.wikipedia.org/wiki/Diff_utility) ja tema sugulased, mis kasutavad pikima ühise osajada leidmist.
- Polünoomide interpoleerimine (https://en.wikipedia.org/wiki/Polynomial_interpolation) on kergesti realiseeritav DP abil. Polünoomide interpoleerimine ise leiab kasutust mitmel pool arvutigraafikas, kus meil on vaja keerulisi jooni lihtsa vaevaga kuvada. Näiteks fontide skaleerimist saab realiseerida, kasutades polünoomide interpolatsiooni.
- Spordiennustus ja modelleerimine, eriti mängudes, mis koosnevad diskreetsetest etappidest, näiteks kriket ja pesapall. Vt nt https://en.wikipedia.org/wiki/WASP_(cricket_calculation_tool)
- Teksti joondamise ja poolitamise algoritmid (kuidas lõpetada ridu nii, et raisataks võimalikult vähe ruumi), mida kasutavad TeX ja teised küljendusprogrammid.
- Plagiaadi leidmise tööriistad kasutavad teisenduskauguse (i.k *edit distance*) leidmise algoritmi, mis põhineb DP-l. Teisenduskaugusest tuleb rohkem juttu peatükis „DP edasijõudnutele“.
- Lühima tee leidmise algoritmid kaardirakendustes.
- Logistika planeerimine.
- Markovi ahelad DNA-s mustrite leidmiseks.
- Soovituste andmine otsingumootorites.
- Võimsuse optimaalne planeerimine elektrijaamades.



Üks päriselu olukord, kus mul isiklikult DP-laadsest lähenemisest kõige rohkem kasu on olnud, oli seotud andmebaasidega. Tegu oli suure, paljude klientidega olmeettevõttega. Kõigil klientidel oli palju mitmesuguseid tarbimisi. Iga kuu lõpus oli ärianalüütikutel vaja teha musttuhat raportit ja päringut erinevate tarbimisliikide, kliendisegmentide jm kohta. Täiendavalt tuli teha simulatsioonid, et mis juhtuks, kui hinnad muutuksid. Algne lahendus sisaldas hulka andmebaasipäringuid, mis töötasid aga terve nädalavahetuse, enne kui raportid valmis said. Kui asja uurisin, siis selgus, et üsna paljusid andmeid päriti kogu selles protsessis palju kordi. Et seda vältida, ehitasin eelpool kirjeldatud DP-tabelite analoogina hulga ajutisi tabeleid, milles jupphaaval lõplikku statistikat koostati. Kokkuvõttes valmisid raportid nüüd päevade asemel minutitega.

Selline lähenemine, kus andmebaasis andmeid teatud päringute huvides denormaliseeritakse, on küllaltki levinud (vt nt https://en.wikipedia.org/wiki/Denormalization), aga selle kombineerimine DP mõtteviisiga lisab su tööriistakasti veel mõned huvitavad võimalused.

5.11 KONTROLLÜLESANDED



5.11.1 Aktsiaturg

Kui siin õpikus oleks kirjas kindel retsept aktsiaturul raha teenimiseks, võiks selle raamatu iga eksemplari eest küsida tuhandeid eurosid. Kuna aga raamatutest pole võimalik kindlaid meetodeid leida, luuravad paljud kauplejad üksteise järel, püüdes jälile saada võitvatele strateegiatele.

Sul on antud võistleva kaupleja tehingute nimekiri, mille igal real on üks number tehingu kohta – täisarvuline kasum või kahjum.

Eesmärgiks on leida sellest jadast maksimaalse summaga alamjada. Kui kasumlikke tehinguid ei ole, väljastada 0.

Sisendi esimesel real on tehingute arv N ($1 \le N \le 10000$). Teisel real on igast tehingust saadud kasum või kahjum.

NÄIDE 1:

```
5
12 -4 -10 4 9
```

Vastus: 13

NÄIDE 2:

```
20
-896 995 -588 -23 648 -980 51 -705 -620 -640 915 663 444 307 -207 81 -763 -993
24 665
```

Vastus: 2329

5.11.2 Protsessori planeerimine

Operatsioonisüsteem peab järjekorda seadma hulga tegevusi, mille sooritamiseks on vaja protsessoriaega. Tegevusi on kaht sorti, esimeste täitmine võtab `n` millisekundit ja teiste täitmine `m` millisekundit.

Eesmärgid (prioriteedijärjekorras) on esiteks minimeerida aega, mil protsessor on jõude (ideaalis null, kui null pole võimalik, siis nii väike kui saab) ning teiseks maksimeerida täidetud ülesannete arvu. Töö tegemiseks on aega `t` millisekundit. Tööd ei tohi pooleli jääda, see oleks sama hea kui jõudeolek.

Kui antud on arvud `m`, `n` ja `t` (täisarvud lõigust 0 - 10000), leida mitu ülesannet on nende reeglite alusel võimalik maksimaalselt lõpetada.

NÄIDE 1:

```
3 5 54
```

Vastus: 18 (kõik kolmesed ülesanded)

NÄIDE 2:

```
3 5 55
```

Vastus: 17 (kaks viiest ja 15 kolmest ülesannet).

NÄIDE 3:

```
886 340 6177
```

Vastus: 10

5.11.3 Maksmise võimalused

Soomes ei ole ühe- ja kahesendised euromündid üldiselt kasutusel. Ignoreerides ühe- ja kahesendiseid (kõiki teisi europangatähti ja münte võib kasutada, hulk ei ole piiratud), leida, mitu võimalust on konkreetse summa maksmiseks. Maksimaalne uuritav summa on 500 eurot. Sisendiks on summa sentides.

NÄIDE 1:

`20`

Vastus: 4 (1x20, 2x10, 1x10 + 2x5 ning 4x5 senti)

NÄIDE 2:

`79`

Vastus: 24

5.11.4 Statistika manipuleerimine

Hoolimata sellest, et statistika on täppisteadus ega saa kuidagi „valetada“, nimetatakse statistikat siiski sageli valelikuks. Tegelikult on valelikud muidugi hoopis need, kes statistikat väärkasutavad, valides välja ainult osad andmed, võrreldes omavahel asjaolusid, mis on tegelikult erinevad, või visualiseerides andmeid eksitavalt.

Vaatame käesolevas ülesandes üht sellist statistika manipuleerimise viisi. Oletame, et keegi tahab näidata, et kallite autodega juhtub vähem õnnetusi. Selleks oleks efektne meetod luua nimekiri automarkidest, mis on üheaegselt hinna poolest kasvavas, aga õnnetuste arvu poolest kahanevas järjekorras.

Antud on hulk automarke koos nende hindade (tuhandetes eurodes) ja õnnetuste arvuga (miljoni auto kohta).

Ülesandeks on leida nende markide seast maksimaalse pikkusega jada, kus automargid on hinna poolest kasvavas, aga õnnetuste poolest kahanevas järjekorras. Jada väljastada esialgse jada järjekorranumbritena. Kõik võrratused on ranged, s.t hinnad peavad olema rangelt kasvavas ja õnnetused rangelt kahanevas järjekorras.

Sisendi esimesel real on automarkide arv `N` (`1 ≤ N ≤ 1000`). Järgmisel `N` real on täisarvupaarid, kus esimene arv tähistab hinda ja teine õnnetuste arvu `A` (`1 ≤ A ≤ 10000`).

NÄIDE:

```
9
13 6008
21 6000
20 500
40 1000
30 1100
20 6000
14 8000
12 6000
19 2000
```

Vastus:

```
7 2 5 4
```

PS! Kui sul tegelikult kästakse kunagi statistikat võltsida, siis intellektuaalselt aus oleks töökohta vahetada ☺

5.11.5 Lennukikütus

Reisilennukid püüavad teatavasti oma kütusekulu võimalikult hästi optimeerida. Oluliseks faktoriks on siin tuule kiirus. Ilmateatest on teada, milline tuul eri paikades ja kõrgustel puhub, ülesandeks on koostada optimaalne lennutrajektoor.
Iga 10 lennukilomeetri jaoks on kolm võimalust: hoida olemasolevat kõrgust (kulub 30 liitrit kütust), tõusta kilomeetri võrra (kulub 60 liitrit kütust) ja laskuda kilomeetri võrra (kulub 20 liitrit kütust).
Tuule kiirus erinevatel kõrgustel on antud 10 km/h täpsusega ja tuule kiirus võib olla -100 km/h (vastutuul) kuni 100 km/h (pärituul). Iga 10 km/h tuult suurendab või vähendab kütusekulu ühe liitri võrra. Tuult arvestatakse algkõrgusest lähtuvalt.
Lubatud on lennukõrgused 1 km kuni 9 km.
Sisendi esimesel real on antud soovitud lennukaugus L (kümnega jaguv positiivne täisarv kuni 10000). Järgmisel 10 real on igaühel $\frac{L}{10}$ täisarvu, mis tähistavad tuulekiirusi vastaval kõrgusel ja teelõigul (kõrgemad enne). Esimene rida tähistab tuult kõrgusel 9 ja alumine rida tuult kõrgusel 0. Reisi algus ja lõpp on mõlemad kõrgusel 0, mis tähendab, et alguses ja lõpus tuleb kindlasti tõusta ja laskuda.

NÄIDE 1:

```
40
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
10 90 90 10
10 -90 -90 10
```

Vastus: 120 (Alustame +10 pärituulega, tõuseme +90 tuulega kõrgusele, hoiame kõrgust ja siis laskume).

NÄIDE 2:

```
100
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
70 70 70 70 70 70 70 70 70 70
-50 -50 -50 -50 -50 -50 -50 -50 -50 -50
-70 -30 -70 -70 -70 -70 -70 -70 -70 -70
-90 -90 -90 -90 -90 -90 -90 -90 -90 -90
```

Vastus: 354

5.11.6 Kahjuritõrje

Farmer Fredil on ristkülikukujuline põld, mis on jagatud 10x10m ruutudeks. Ruute on kokku `MxN`. Nüüd on tal vaja põldu taimekaitsevahenditega pritsida. Pritsimine toob nii kasu (hävitades kahjureid) kui kahju (suurendades saagi kemikaalisisaldust). Iga ruudu kohta on teada, kui palju kasu või kahju selle ruudu pritsimine kokku annab.
Fred saab tellida lennuki, mis pritsib üle parajasti mingi ühe konkreetse ristkülikukujulise osa põllust (võib ka terve). Leida, kui palju kasu on võimalik lennuki tellimisest saada. Formaalsemalt väljendudes: leida maksimaalse summaga alamristkülik.
Sisendi esimesel real on täisarvud `M` ja `N` (`1 ≤ M, N ≤ 100`). Järgmistel `M` real on igaühel `N` arvu, mis tähistavad pritsimisväärtusi (täisarvud -127 kuni 127).

NÄIDE 1:

```
4 4
0 -2 -7 0
9 2 -6 2
-4 1 -4 1
-1 8 0 -2
```

Vastus: 15 (maksimaalse summaarse väärtusega alamristkülik asub vasakus alumises nurgas:

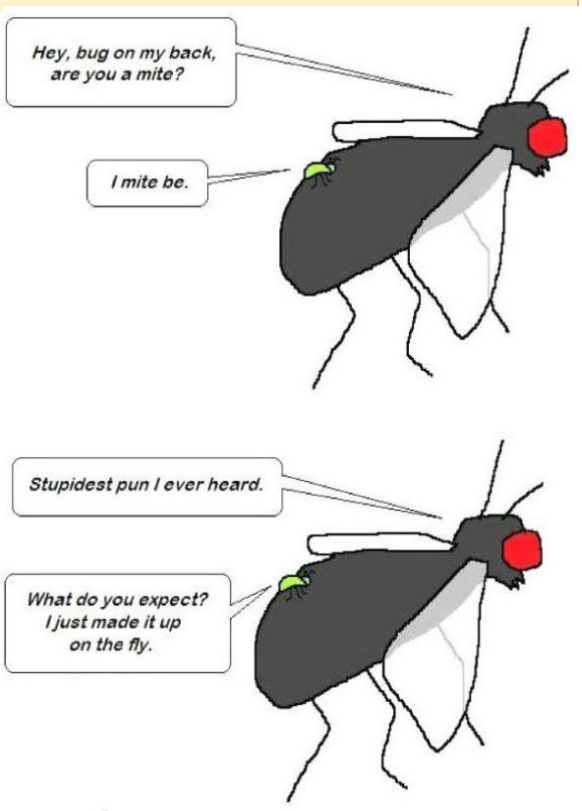
```
9 2
-4 1
-1 8
```

ja selle summa on 15)

NÄIDE 2:

```
5 5
125 85 -120 -114 63
-110 -77 49 27 47
13 -10 101 -116 -34
60 111 -1 61 -24
-120 110 80 69 107
```

Vastus: 513



5.11.7 Torni ehitamine

Väike Jaak ehitab klotsidest torni. Kõik klotsid on risttahukakujulised. Kuna Jaak on alles noor, suudab ta laduda klotse ainult nii, et asetab väiksema klotsi suurema peale. Täpsemalt tähendab see, et pealmise klotsi horisontaalsed mõõtmed (pikkus ja laius) peavad olema rangelt väiksemad kui alumise klotsi pikkus ja laius.
Samas võib klotse igat pidi pöörata, nii et nende pikkus, laius ja kõrgus on omavahel vahetatavad.
Jaagu vanem vend Ants soovib nooremat venda aidata ja annab talle kätte klotse sellises järjekorras ja niipidi pööratult, et Jaak saaks kokku võimalikult kõrge torni.
Kui vendadel on kokku `N` sorti klotse ja igat klotsi on võimalik kasutada piiramatul hulgal, leida, kui kõrge torni nad kokku saaksid.
Sisendis on antud arv `N` ning seejärel iga `N` klotsi kohta tema dimensioonid.

NÄIDE 1:

```
1
30 20 10
```

Vastus: 40 (kõigepealt klots lapiti, kõrgus 10, siis teine klots püsti esimese otsa, kõrgus 30, kokku 40)

NÄIDE 2:

```
2
6 8 10
5 5 5
```

Vastus: 21 (kõrgused 6, 10 ja 5)

NÄIDE 3:

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

Vastus: 28 (suuremast väiksemani üksteise otsas)

NÄIDE 4:

```
5
31 41 59
26 53 58
97 93 23
84 62 64
33 83 27
```

Vastus: 342

5.11.8 Putin sukeldumas

Vene Föderatsiooni president Vladimir Vladimirovitš Putin on teatavasti tuntud oma füüsiliste vägitükkide poolest, olgu siis tegu ratsutamise, lendamise või võitlemisega. Muuhulgas on ta paistnud silma oma sukeldumisoskusega, tuues Musta mere põhjast ära antiikseid vaase.
Vähem on aga teada, et sukeldumisele eelnes põhjalik planeerimine, kui palju ja millist varandust Putin ära saaks tuua.
Sul on unikaalne võimalus olla V.V. Putini sukeldumisülem ja planeerida võimalikult tulus sukeldumine. On teada, et merre on eelnevalt paigutatud `N` vaasi, sügavustele vastavalt $s_1,\ldots,s_N$ meetrit ning väärtustega $v_1,\ldots,v_N$. Putinil on hapnikuballoon, millest piisab, et olla vee all `t` sekundit. Vaasid tuleb ära tuua ükshaaval. Meres laskumine võtab aega `w` sekundit meetri kohta ning tõusmine `2*w` sekundit meetri kohta.
Sul tuleb leida parim strateegia, millises järjekorras peaks Putin vaase ära tooma, et nende koguväärtus oleks maksimaalne.
Sisendi esimesel real on arvud `t`, `w` ja `N`. Järgmisel `N` real on vaaside sügavused ning väärtused.
Väljastada iga vaasi sügavus ja väärtus, mille Putin saab ära tuua, kasutades parimat võimalikku strateegiat. `1 ≤ t ≤ 1000`, `N ≤ 30`.

NÄIDE 1:
```
210 4 3
10 5
10 1
7 2
```
Vastus:
```
10 5
7 2
```
NÄIDE 2:
```
656 2 5
79 74
97 75
37 81
21 99
38 25
```
Vastus:
```
37 81
21 99
38 25
```

5.11.9 Arvude liitmine

Antud on positiivsed täisarvud `K` ja `N` (`1 ≤ K, N ≤ 100`).
Mitu võimalust on arvu `N` esitamiseks K mittenegatiivse täisarvu summana? Sisendis on antud arvud `K` ja `N`, väljastada üks arv vastusena.

NÄIDE 1:
```
2 5
```
Vastus: 6 (0 + 5, 1 + 4, 2 + 3, 3 + 2, 4 + 1, 5 + 0)

NÄIDE 2:
```
3 4
```
Vastus: 15

NÄIDE 3:
```
100 3
```
Vastus: 5151

5.11.10 Templisambad

Antiikse templi restaureerimiseks on antud N marmorist silindrikujulist kivi, mis võivad olla erineva kõrgusega. Silindritest tuleb laduda K millimeetri kõrgune sammas. Kui täpselt K kõrgust sammast pole võimalik ehitada, tuleb mõni kivi madalamaks lihvida, mis tähendab, et kallist materjali läheb raisku. Eesmärgiks on minimeerida esiteks kaotsimineva materjali hulka ning teiseks vajaminevate kivide arvu. Sisendi esimesel real on arv K, teisel real arv N ($N \leq 100$). Järgmisel N real on kivide kõrgused (täisarvud kuni 2000). Väljastada vajaminevate kivide arv ja nende kõrgus enne madalamaks lihvimist.

NÄIDE 1:

```
1400
3
500 1000 2000
```

Vastus:

```
2 1500
```

NÄIDE 2:

```
2508
5
331 1585 1395 278 537
```

Vastus:

```
4 2541
```

(Pildil on Apolloni templi varemed Delfis).

5.12 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk5 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
Fib.cpp, Fib.java, Fib.py	Fibonacci arvu leidmine rekursiooniga
Fib2.cpp, Fib2.java, Fib2.py	Fibonacci arvu leidmine mäluga rekursiooniga
Fib3.cpp, Fib3.java, Fib3.py	Fibonacci arvu leidmine alt-üles DP-ga
Fib4.cpp, Fib4.java, Fib4.py	Fibonacci arvu leidmine kokkuhoidliku DP-ga
Myndid1.cpp, Myndid1.java, Myndid1.py	Mündiülesande lahendus jõumeetodil
Myndid2.cpp, Myndid2.java, Myndid2.py	Mündiülesande lahendus DP-ga
Myndid3.cpp, Myndid3.java, Myndid3.py	Mündiülesande lahendus koos tagurdusmeetodiga (tagastab müntide väärtused)
LIS.cpp, LIS.java, LIS.py	Pikima kasvava osajada pikkuse leidmine jõumeetodil
LIS2.cpp, LIS2.java, LIS2.py	Pikima kasvava osajada pikkuse leidmine DP-ga
LIS3.cpp, LIS3.java, LIS3.py	Pikima kasvava osajada leidmine DP-ga
LCS.cpp, LCS.java, LCS.py	Pikima ühise osajada pikkuse leidmine DP-ga
Ristsumma.cpp, Ristsumma.java, Ristsumma.py	Ristsumma ülesande DP lahendus
ristsumma.xlsx	Ristsumma Excelis
Mortimer.cpp, Mortimer.java, Mortimer.py	Miljonäri ja vaeslaste ülesande lahendus
Vennad.cpp, Vennad.java, Vennad.py	Õiglase jagamise ülesande lahendus
Sallid.cpp, Sallid.java, Sallid.py	Moekunstniku ja koiliblika ülesande lahendus

6 SISSEJUHATUS GRAAFITEOORIASSE

6.1 GRAAFITEOORIA TERMINID.

6.2 GRAAFIDE ESITUSVIISID

naabrusmaatriks, tippude list, servade list.

6.3 GRAAFI LÄBIMINE.

Laiuti läbimine. Sügavuti läbimine.

6.4 FLOOD FILL

6.5 TOPOLOOGILINE SORTEERIMINE.

6.6 KAALUDE JA SUUNDADEGA GRAAFID.

6.7 DIJKSTRA ALGORITM.

6.8 KONTROLLÜLESANDED

TEINE OSA - EDASIJÕUDNUTELE

Eeldus: esimese kursuse teadmised on omandatud.

7 AHNED ALGORITMID.

Coin change, load balancing, internal covering.

7.1 KONTROLLÜLESANDED

8 DÜNAAMILINE PLANEERIMINE EDASIJÕUDNUTELE

8.1 DP KUI GRAAFI LINEARISEERIMINE

8.2 SELJAKOTI PAKKIMINE

Paljud lugejad on tõenäoliselt kuulnud seljakotiülesandest: meil on seljakott, millesse on lubatud panna fikseeritud maksimaalse kaalu jagu asju ja hulk esemeid, millel on väärtused. Ülesandeks on täita seljakott võimalikult suure väärtusega sisuga. Formaalselt väljendudes: (kopi wikipediast: https://en.wikipedia.org/wiki/Knapsack_problem)

Seljakotiülesandele taanduvaid ülesandeid võib kohata paljudes optimiseerimisvaldkondades.

Paljud on ka kuulnud, et seljakotiülesanne on NP-keeruline ja seda ei saa polünomiaalse ajaga lahendada. Ülesandeklassi sagedast esinemist arvestades on aga kasulik teada, et paljude tavapäraste sisendite jaoks eksisteerib lihtne DP algoritm.

Vaatame siinkohal näidet, kus seljakotti pakitavate esemete kaalud on kõik positiivsed täisarvud, maksimaalse suurusega W.

```
int pakiKott(int maksKaal, int kogus, int* kaalud, int* vaartused)
{
    int** parimad = new int*[maksKaal + 1];
    for (int i = 0; i <= maksKaal; i++){
        parimad[i] = new int[kogus];
    }

    for (int i = 0; i < kogus; i++){
        parimad[0][i] = 0;
    }
    for (int i = 0; i <= maksKaal; i++){
        parimad[i][0] = 0;
        if (kaalud[0] <= i){
            parimad[i][0] = std::max(parimad[i][0], vaartused[0]);
        }
    }
    int vastus = 0;
    for (int i = 1; i <= maksKaal; i++){
        for (int j = 1; j < kogus; j++){
            parimad[i][j] = parimad[i][j - 1];
            if (kaalud[j] <= i){
                parimad[i][j] =
                    max(parimad[i][j], parimad[i - kaalud[j]][j - 1] + vaartused[j]);
            }
            vastus = max(vastus, parimad[i][j]);
        }
    }
    return vastus;
}
```

8.3 Edit distance.

8.4 Geenijärjendite leidmine

8.5 Rändkaupmehe ülesanne

8.6 Dynamic Time Warping

8.7 Maatriksite korrutamine

8.8 Floyd-Warshalli algoritm

8.9 Kontrollülesanded

9 GRAAFITEOORIA

9.1 TOESEPUU

Kruskali algoritm. Union-find

9.2 EULERI TSÜKLI LEIDMINE

Floyd-Warshalli algoritm.

9.3 KAHEALUSELISED GRAAFID.

9.4 MAKSIMAALNE VOOG JA MINIMAALNE LÕIGE

Ford-Fulkersoni algoritm.

9.5 KONTROLLÜLESANDED

10 ANDMESTRUKTUURID EDASIJÕUDNUTELE

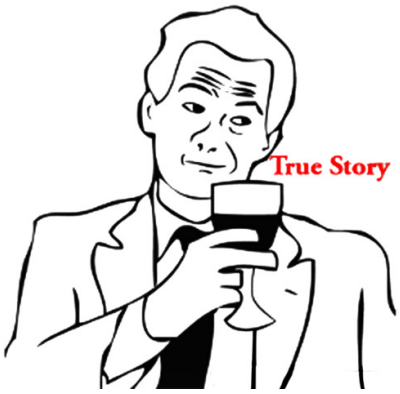
10.1 FENWICKI PUU.

10.2 2D FENWICKI PUU.

10.3 LÕIKUDE PUU.

10.4 LAISK VÄÄRTUSTAMINE LÕIKUDE PUUDES.

10.5 O(LOG N) OPERATSIOONID PUUDEL.

Päriselu näide: ajamärkija 

10.6 KONTROLLÜLESANDED

11 TEKSTIALGORITMID

11.1 TEKSTI PARSIMINE

Aritmeetilise avaldise parsimine

11.2 RÄSID

11.3 KNUTH-MORRIS-PRATTI ALGORITM

11.4 AHO-CORASICKI ALGORITM

11.5 SUFIKSIPUUD

11.6 KONTROLLÜLESANDED

12 ARVUTUSGEOMEETRIA

12.1 SAMAL JOONEL ASUMISE KONTROLL

12.2 PARALLEELSUSE JA SAMASIHILISUSE KONTROLL

12.3 LÕIKUDE LÕIKUMINE

12.4 KUJUNDI PINDALA

12.5 PUNKTI SISALDUMINE KUJUNDIS

12.6 KOORDINAATIDE PAKKIMINE

12.7 KUMER KATE

12.8 SWEEPING LINE

BOI ülesanne – ring ja ruut

12.9 BRESENHAMI ALGORITM

12.10 MAGIC MISSILE

12.11 KONTROLLÜLESANDED