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.

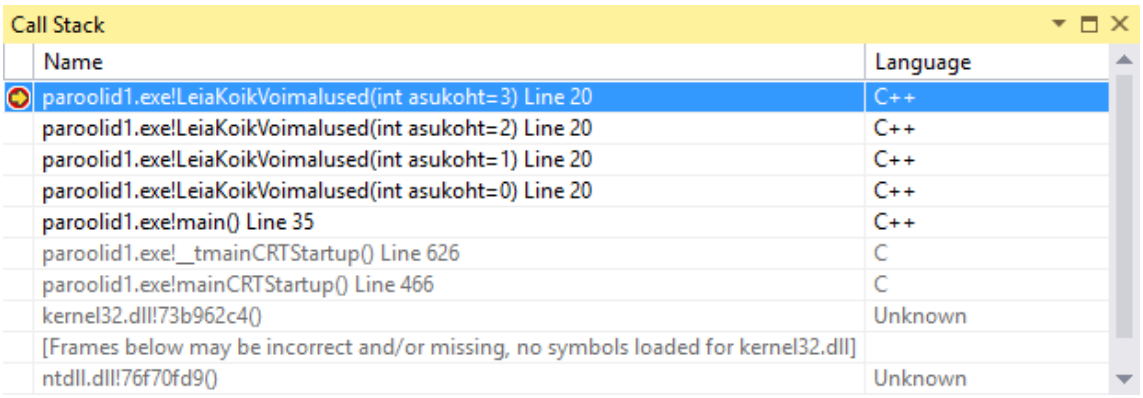
Sõltuvalt programmeerimiskeelest võidakse alamprogramme nimetada funktsioonideks, protseduurideks või meetoditeks. C ja sellest põlvnevad keeled kasutavad tavaliselt terminit „funktsioon“, mis viitab sarnasusele matemaatilise funktsiooniga. Nagu matemaatilisel funktsioonilgi, võivad programmi funktsioonil 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.

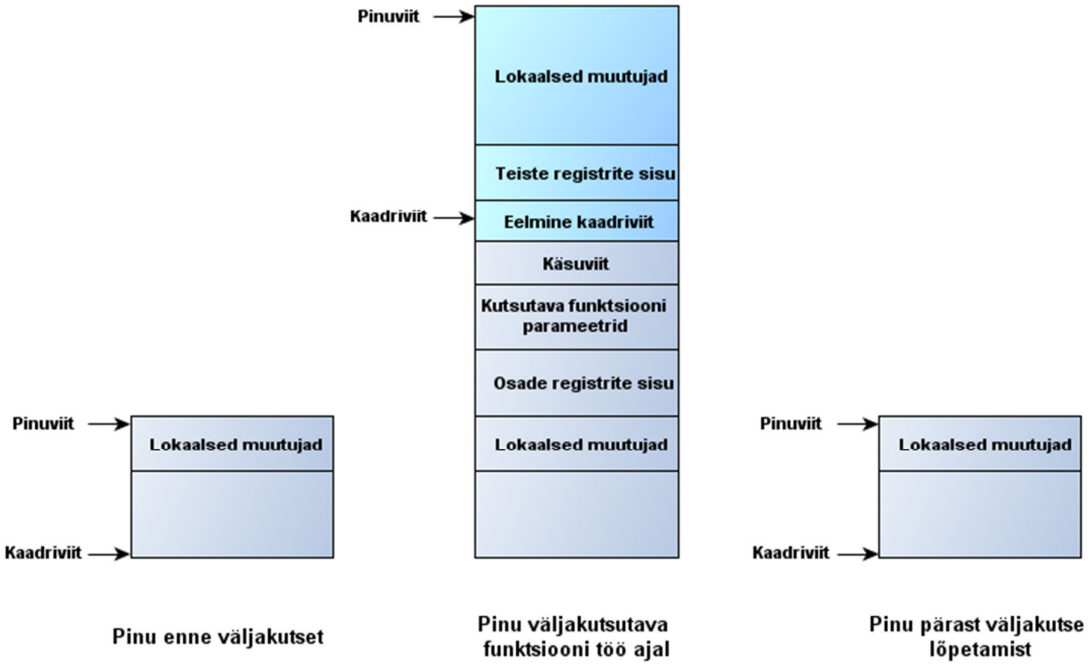
Ka väljakutustaval funktsioonil endal võivad olla muutujad. Needki hoitakse pinus. Siin tekib aga probleem, et kui kutsutud funktsioon asub välja kutsuma omakorda järgmist funktsiooni jne, siis läheb keeruliseks arvet pidada, kus üks pinukaader hakkab ja teine lõpeb. Selleks on kasutusel spetsiaalne 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

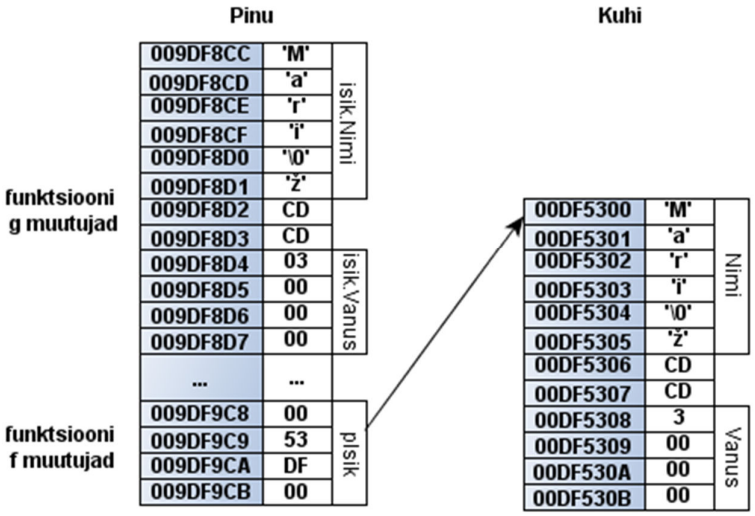
Kui kasutada funktsiooni parameetrina objekte, käituvad keeled mõnevõrra erinevalt. Järgnevalt mõned näited, milles kasutatakse klassi `Isik`, millel on väljad `Nimi` ja `Vanus` ning konstruktor, mis seab ka nime. C++ on klass defineeritud järgmiselt:

```
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;
}
```

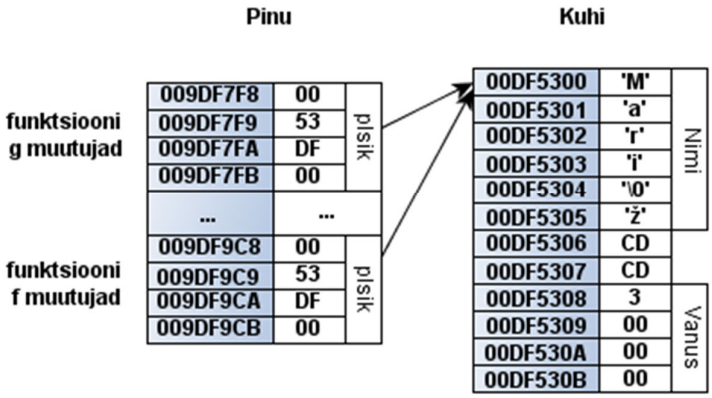


Funktsioonis f on kohaliku muutujana viit kuhimälus loodud Isik-tüüpi objektile. Funktsiooni g väljakutsel luuakse tervest objektist koopia pinus ning g muudab pärast seda ainult oma lokaalset koopiat. Funktsiooni f juurde tagasi pöördudes g koopia „kaob“. Seetõttu väljastab antud programmijupp vanuse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

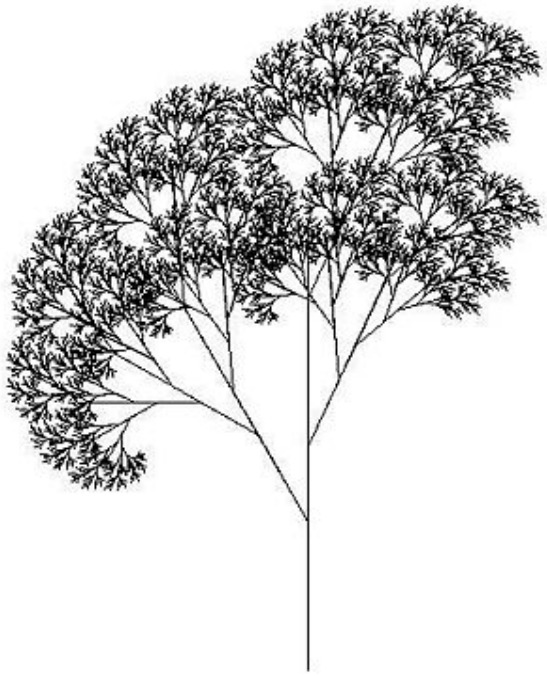
```
public void f() {
    Isik i = new Isik("Mari");
    g(i);
    System.out.println(i);
}
public void g(Isik i) {
    i.nimi = "Maria";
}
```

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.

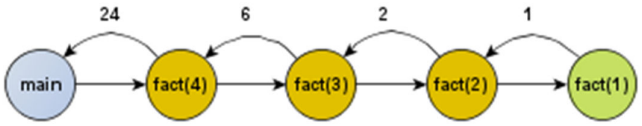
Hargnemistega rekursiooni ongi lihtne ette kujutada puuna ja seda nimetataksegi ka puurekursiooniks. Puu juur on töö alguspunkt, lehed on baasjuhtumid ning hargnemiskohad rekursiivsed väljakutsed. Oksi ehk servasid mööda liigub väljakutsete info ja funktsiooni tagastused.

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:

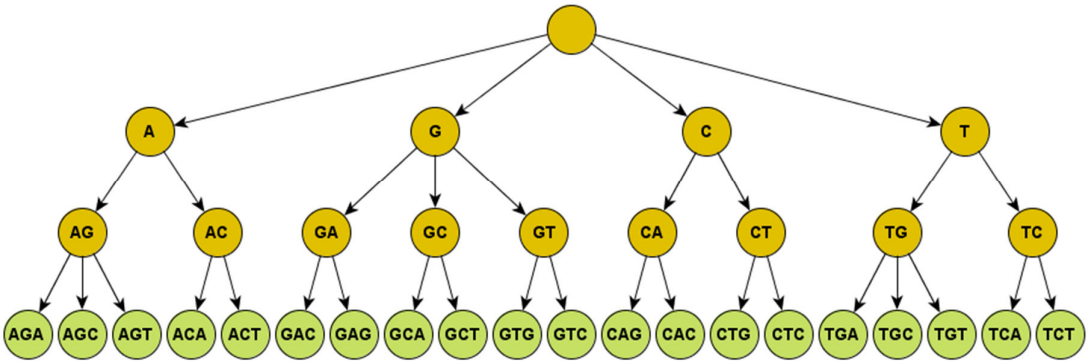
3

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:

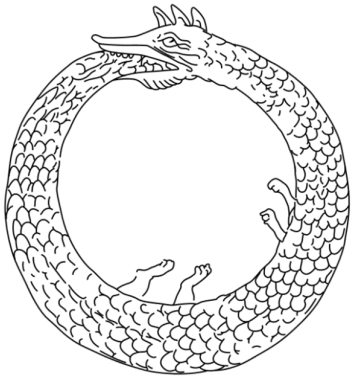


Piirangutega arvestav skeem ülesande lahendite kujunemisest

```
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:

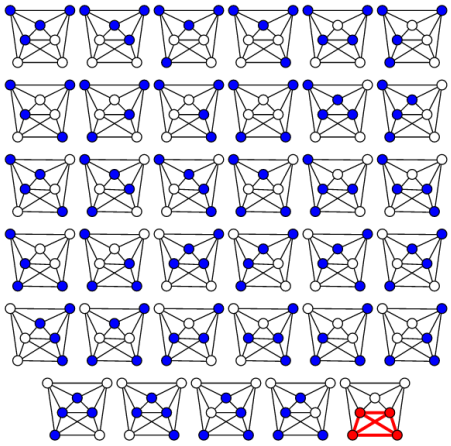
```
012812A0   mov     al,byte ptr [ecx]       ; stringi lõpu kontroll
012812A2   test    al,al
012812A4   je      strtoint+1Bh (012812BBh) ; je="jump if equal". Kui eelmise rea
                                           ; tulemus oli 0, hüppame funktsiooni
                                           ; 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:

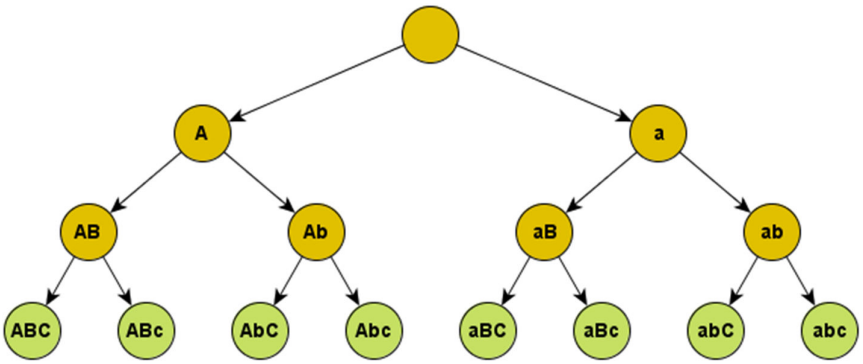
James Bond on tunginud vaenlase peakorterisse, kus superkurjategija parajasti maailma hävitamise masinasse parooli sisestab. Bond näeb sõrmede liigutamise järgi, millised märgid paroolis on, kuid ei näe, millisel hetkel pahalane shift-klahvi all hoiab. Seetõttu ei tea ta, millised on suur- ja millised 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.

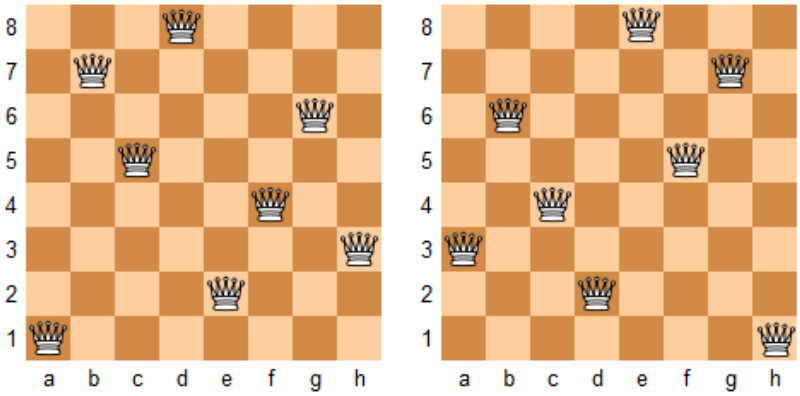
Vastav lahendus on järgmine:

```
// 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.

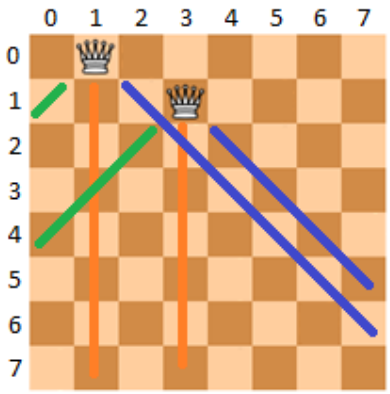
2.4.6 Andmestruktuuride optimeerimine

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.

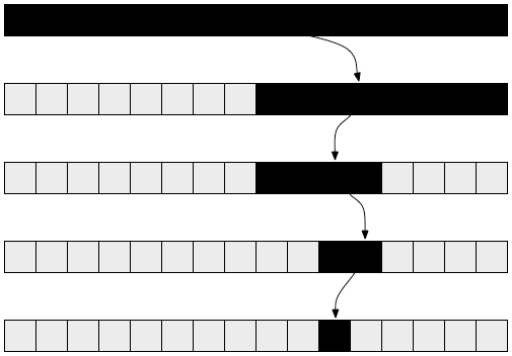
Tulemuseks on selline funktsioon:

```
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≤N≤100000`), teisel real asjade arv `M` (`1≤M≤100000`) ja viimasel real asjade suurused pakkimise järjekorras (`1≤M`$_i$`≤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 $x \leq y$ (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

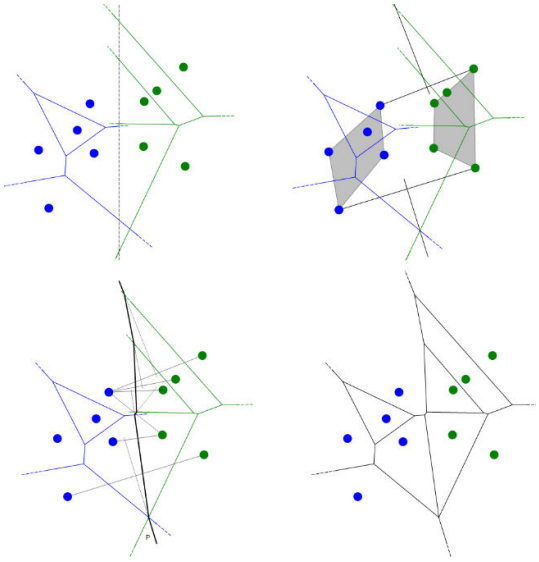
Vaatame veel korra lähemalt, kuidas arvud ridu ja veergu pidi sorteeritud tabelis paiknevad. Järgmisel joonisel on 20x20 tabel, milles on arvud vahemikus 0-1000. Sinised ruudud näitavad teed arvu 409 otsimisel lineaarselt liikudes:

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:

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

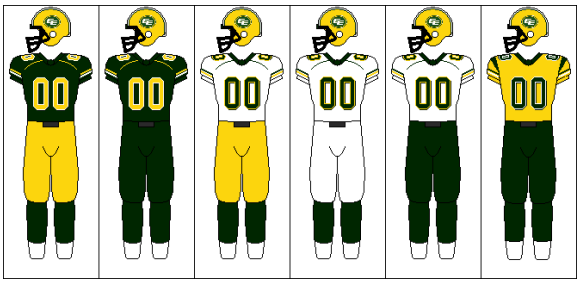
```
long double liidaKiirem(int start, int end)
{
      if (end - start < 1000) { //liita pole rohkem kui 1000 arvu, liidame tavaliselt
            long double vas = 0;
            for (int i = start; i <= end; i++){
                  vas += liidetavad[i];
            }
            return vas;
     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

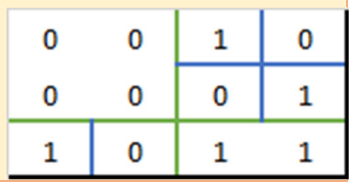
Vastus: 1 (piisab neljanda lipu liigutamisest kuuendale ruudule).

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`).

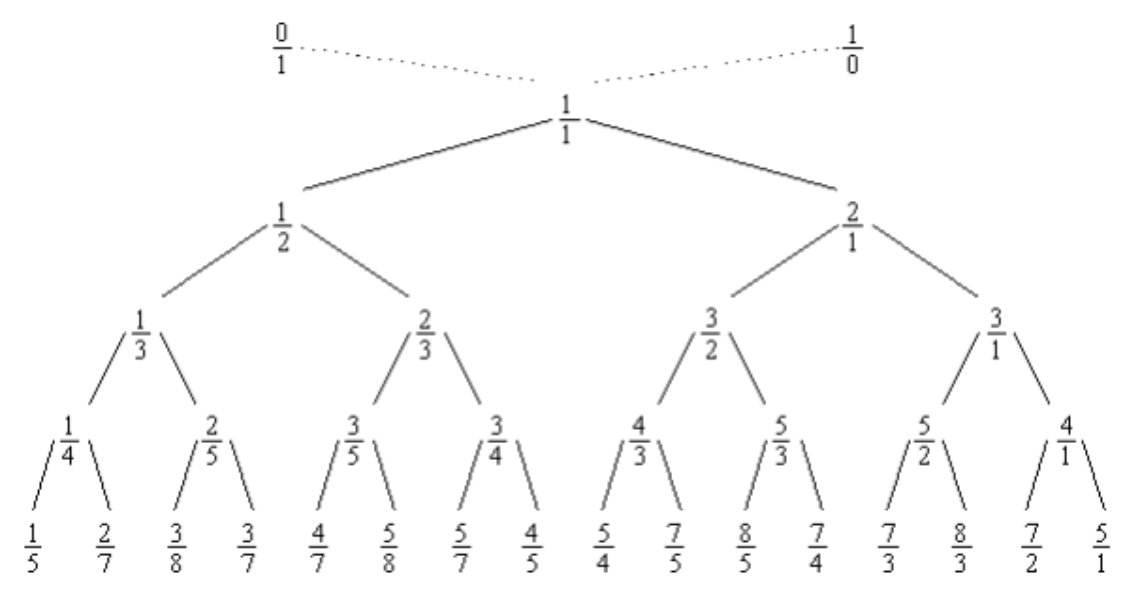


2.8.6 Stern-Brocot' puu

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:

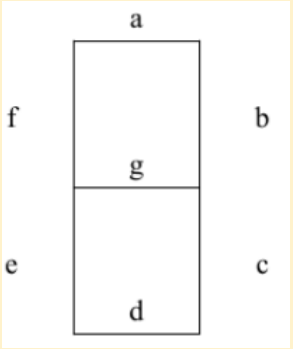
```
400 12
50 50 50 50 50 50 25 25 25 25 25 25
```

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:

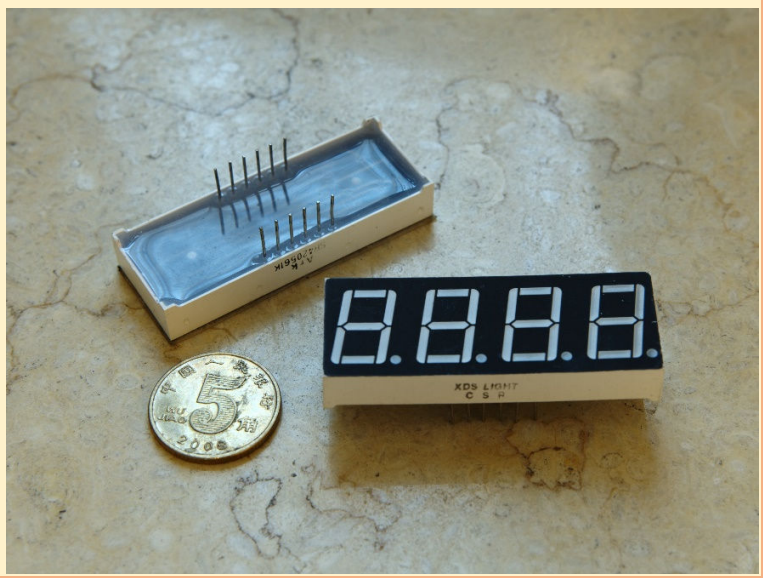
```
3
YNYYYYY
YNYYNYY
NYYNNYY
```

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.

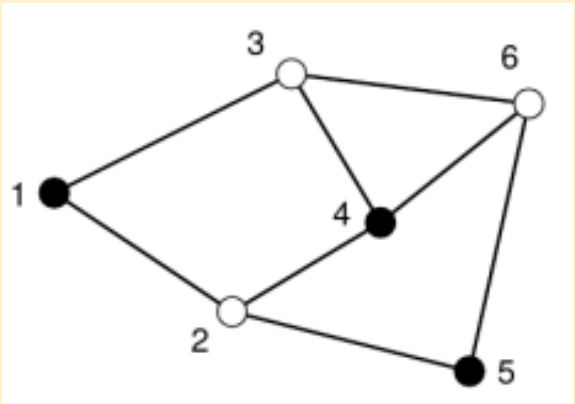
Väljundisse kirjutada kaks rida: esimesele mustaks värvitavate tippude arv, teisele see, millised tipud värvitakse, sordituna väiksemast suuremani. Kui võimalikke lahendusi on mitu, siis väljastada neist minimaalne, s.t selline, kus esimese tipu number on väikseim võimalik, seejärel teise tipu number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