7 EFEKTIIVNE PROGRAMMEERIMISTEHNIKA

Algaja programmeerija jaoks on suur asi, kui tema programm tööle hakkab ja enam-vähem õiget asja teeb. Tõepoolest, see on päris uhke tunne ja selle tunde pärast armastavadki paljud inimesed programmeerimist: „Ma olen jumal, kes tegi mitte millestki midagi ja andis masinale elu!“

Need lugejad, kes on jõudnud siia, õpiku teise osani, pole aga enam algajad, vaid arvestatava kogemusega programmeerijad. Kui alguses oli ainsaks edukriteeriumiks see, kas programm töötab, siis nüüd tuleb esile veel palju teisi kriteeriume, millele üks hea programm vastama peaks:

- Jõudlus – käesoleva raamatu põhiteema: kas minu programm töötab piisavalt kiiresti?
- Skaleeruvus – leiab samuti siin raamatus kajastamist: kuidas teha nii, et programm ka suuremate andmehulkade puhul hästi töötaks.
- Kasutusmugavus – kui kergesti ja efektiivselt saab kasutaja antud programmiga hakkama? Kuna võistlusprogrammeerimise formaat on selline, et programme „kasutavad“ masinad ning sisend ja väljund on konkreetselt defineeritud, siis sellel kriteeriumil siin raamatus pikemalt ei peatuta.
- Hallatavus – kui kerge on programmis muudatusi teha? Kas muudatus ühes kohas võib midagi muud katki teha?
- Töökindlus – kui hästi saab programm hakkama erijuhtumite ja ootamatute olukordadega?
- Laiendatavus – kui lihtne on programmile lisada täiendavat funktsionaalsust?
- Turvalisus – kas programm hoiab ära „mitte ettenähtud tagajärgede“ tekitamise? Ka selle kriteeriumi pärast võistlusprogrammeerija üldjuhul eriti muretsema ei pea.

Kui me soovime kirjutada head „päris elu“ programmi, siis on kõik need punktid sama tähtsad kui see, kas programm teeb õiget asja, vahel isegi tähtsamad!

Võrdleme näiteks kahte olukorda:

1. Programm töötab valesti, aga on kergesti modifitseeritav.
2. Programm töötab õigesti, aga on kirjutatud segaselt ja keeruliselt, nii et sellesse pole võimalik mingeid muudatusi teha.

Esimesel juhul saab vead ära parandada ja kõik on korras. Kui aga teisel juhul on tarvis programmi mingeid muudatusi teha, on vahel on ainsaks väljapääsuks programm algusest peale uuesti kirjutada. Seega kasvab koos programmi elueaga ka hallatavuse ja laiendatavuse väärtus.

Võistlusprogrammeerimise seisukohalt on ülaltoodud loetelust tähtsaimad mõistagi **jõudlus** ja **skaleeruvus** ning enamik käesolevast raamatust tegelebki nende teemadega. Võistlusel kasutatavate programmide eluiga on tavaliselt ülilühike ja teised aspektid jäävad tavaeluga võrreldes tagaplaanile, kuid ka siin ei saa ignoreerida **hallatavust** (kui programmis on viga või vaja midagi ümber kirjutada, siis

kas seda on lihtne teha), **töökindlust** (igasuguste vigade vältimiseks) ja **laiendatavust** (ülesanded koosnevad sageli mitmest osast – kui üks osa on lahendatud, siis kui mugavalt on võimalik programmi täiendada).

Tänapäeva tarkvara on oma olemuselt ülikeeruline, koosnedes paljudest komponentidest, mis kõik peavad harmooniliselt koos töötama, kuid samas ka ülihabras, sest vahel võib üheainsa biti muutmine tulemuse valeks muuta või siis programmi üldse ära lõhkuda. Võime kujutleda, et tegu on tuhandete hammasratastega, mis peavad kõik sünkroonis pöörlema, ja niipea kui üks neist kinni kiilub, siis programm enam ei tööta.

Siin peatükis uurime, kuidas seda kinnikiilumist võimalikult hästi vältida ja kui see siiski juhtub, siis kuidas kinni jäänud ratast leida ja taas valla päästa.

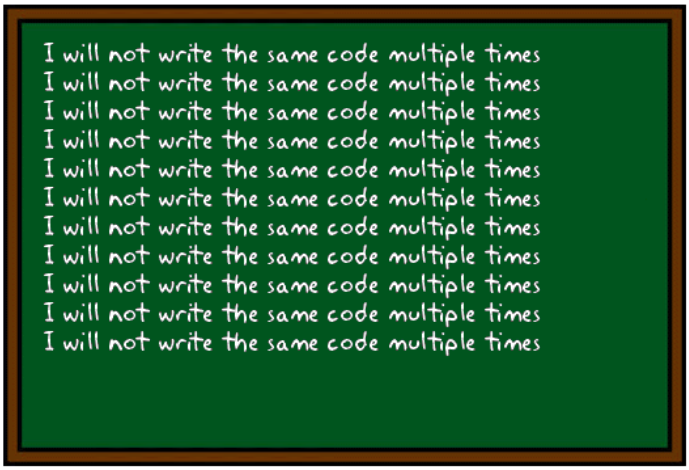
7.1 KOODISTIIL

7.1.1 Pikaajaline ja lühiajaline lähenemine

Mida saab programmiga teha? Intuitiivne vastus on „valmis kirjutada ja käivitada“. Hea küll, kirjutasime ja käivitasime, aga nüüd on selles vaja midagi muuta või parandada. Kui algne kood polnud kirjutatud just hetk tagasi, tuleb enne täiendamist teha veel midagi olulist: senist koodi **lugeda**, et aru saada, mida see teeb, muidu pole efektiivne täiendamine võimalik. On programme, mida ei loeta pärast kirjutamist mitte kunagi, kuid on ka programme, mida loetakse tuhandeid kordi – enamasti juhul, kui tegu on tiimitöö ja pikaajalise projektiga. Vastavalt sellele, kui palju me eeldame, et programmi on vaja lugeda, muutub ka **loetavusele optimiseerimine** järjest tähtsamaks: lugemisele kuluva aja säästmiseks on sageli vaja kirjutamisele rohkem aega kulutada.

Programmeerimisvõistlustel kiiresti ja hooletult koodi kirjutamine tekitab kergesti halbu harjumusi, sest võistlusel kasutatava programmi koodi on harva pärast võistlust tarvis. Siiski kulub hea stiil ka juba võistluse jooksul ära, hoides kokku vigade leidmisele ja parandamisele kuluvat aega.

7.1.2 Alamprogrammid



Programmeerimisvõistlustel on inimestel sageli kogu programm ühes põhifunktsioonis. Kui lahendus on lihtne ja vähem kui tunniga realiseeritav ning sa oled ka kindel, et tegemist on õige lahendusega, pole sellest iseenesest midagi katki. On aga palju võistlusi (päriselu ülesannetest rääkimata), kus ülesande lahendamiseks on aega mitu päeva ning töö ka selle võrra mahukam. Sel juhul tuleks programm kindlasti jagada väiksemateks funktsioonideks, kuna see teeb koodi lihtsamini loetavamaks ja paremini hallatavaks.

Täpsemalt on alamprogrammide kasutamisel järgmised olulised eelised:

- **Ülevaatlikkus**. Kui vaadeldav plokk on pikem kui inimene korraga haarata suudab, läheb lõppu jõudmise ajaks kergesti meelest, mis alguses toimus. Jagades koodi väiksemateks osadeks, lööme ühe hoobiga kaks kärbest: esiteks on iga osa puhul koha arusaadav, mis toimub, teiseks saab igale osale anda ülevaatliku, kirjeldava **nime**, mis hõlbustab mõistmist veelgi.
- **Probleemi jagamine alamosadeks**. Paljud ülesanded on võimalik jagada loogiliselt eristuvateks alamosadeks, sel juhul on hea, kui lahendus järgib sama struktuuri. Esiteks võimaldab see probleemi osade kaupa lahendamist, kuid teise, veel tähtsama võidu saavutame läbi parema **testitavuse**. Igale funktsioonile, mis probleemist teatud osa lahendab, on võimalik koostada eraldi testid, mis selle osa korrektsust kontrollivad – selline lähenemine hoiab sageli vigade otsimise aega kokku.
- **Koodi taaskasutuse võimaldamine.** Sageli tuleb ülesande raames sama või sarnast protseduuri mitu korda korrata. „Laisk“ meetod oleks seda koodi lihtsalt kopeerida, aga kui selgub, et selles koodis on viga või on tarvis midagi täiendada, maksab kopeeritud koodi mitmekordne parandamine peagi kätte – ühelt poolt läheb rohkem aega ja teiselt poolt on kerge uusi vigu teha.

Kui pikk tohiks üks alamprogramm olla? Tuntud tarkvarakonsultant ja agiilse arenduse kontseptsiooni üks loojaid Robert „Uncle Bob“ Martin (https://cleancoders.com/) soovitab, et alamprogrammi pikkus võiks olla umbes neli rida ja kümme on juba liiga palju.

Uncle Bob

Enamik programmeerijaid peab nii madalat piiri siiski liiga radikaalseks, kuid nõustuvad, et liiga pikad moodulid on halvad. Mina isiklikult kirjutasin oma esimesed programmid tekstimoodis kuvaril, mis näitas 25 rida ja 80 märki real, see oli üldine standard. Ka praegu avanevad paljud virtuaalterminalid vaikimisi just sellistes mõõtudes. Kui vaadeldav alamprogramm on suurem kui sellisesse aknasse mahub, tuleb seda kerida ning lugemine on juba raskendatud. Samuti on **25 rida** ka enamvähem piir, kui palju infot inimene korraga haarata suudab, mistõttu kasutatakse koodistandardites sageli just seda suurust.

Targo esimene programmeerimisarvuti oli samasugune

Taaskasutusest tuleb lähemalt juttu järgmises näites.

7.1.3 Näide: Tankid

2016/17 olümpiaadihooaja raames korraldati ka Tankide Turniir, kus iga osaleja sai programmeerida tanki, mis vastaval võistlusväljakul teistega võitlema peab. Sellise programmi kirjutamisel tekib üsna palju korduvaid osi, mida omaette funktsioonidesse koondada.

M1A1 Abrams tank

Näiteks vaatame siin üht võistlusel esinenud väga pikka funktsiooni (umbes 400 rida) `MineNurka`, mis proovib mängija tanki liigutada laual lähimasse nurka. Liikumise käigus tulistatakse ettejäävaid tanke ning vajadusel muudetakse liikumissuunda ja proovitakse minna ümber takistuse. Kogu funktsioon on liiga pikk, et siin ära tuua, aga siin on sellest esimene veerand:

```
int mineNurka()
{
    // mängulaud on m x n; x, y on tanki koordinaadid
    if (m - x > x) {  // mis pooles tank on, mis nurk on lähemal
        if (n - y > y) { // mis veerandis
            bool ol = 1; // on liikunud
            while (!ok[0][0] && (x != 0 || y != 0)) { // kuni nurk on vaba
                ol = 0;
                while (x != 0 && y != 0 && !ok[y - 1][x - 1]) { //liigu nurga suunas
                    ol = 1;
                    cout << "M5" << endl; // M5 liigu suunas alla-vasakule
                    lts(); // loeb info, sh tanki uued koordinaadid nx ja ny
                    if (nx == x && ny == y) { // ei liikunud (nx, ny - uued koordinaadid)
                        ok[y - 1][x - 1] = 1; // seal on keegi ees
                        fire(5, 6); // tulista sinna
                    }
                    xnx(); // uuendab koordinaadid
                }
                bool fm = 1; // diagonaalis liikuda ei saa
                while (x > 0 && !ok[y][x - 1] && (x > y || fm)) { // x telge mööda
                    ol = 1;
                    fm = 0;
                    cout << "M6" << endl; // liigu vasakule
                    lts();
                    if (nx == x && ny == y) {
                        ok[y][x - 1] = 1;
                        fire(6, 6);
                    }
                    xnx();
                }
                fm = 1;
                while (y > 0 && !ok[y - 1][x] && (x < y || fm)) { // alla
                    ol = 1;
                    fm = 0;
                    cout << "M4" << endl; // alla
                    lts();
                    if (nx == x && ny == y) {
                        ok[y - 1][x] = 1;
                        fire(4, 6);
                    }
                    xnx();
                }
                if (!ol) { // kui ei ole üldse liikunud
                    if (x < y) { // üleval vasakul võrreldes õige diagonaaliga
                        cout << "M3" << endl; // liigub alla paremale
                        lts();
                        if (nx == x && ny == y) { // seal ka keegi ees
                            ok[y - 1][x + 1] = 1;
                            fire(3, 6); // tulista
                            cout << "M2" << endl; // liigu paremale
                            lts();
                            if (nx == x && ny == y) {
                                ok[y][x + 1] = 1;
                                fire(2, 6); // tulista
                                cout << "M1" << endl; // liigu üles paremale
                                lwf(); // loe ilma tulistamata (lts, xnx järjest)
                                cout << "M3" << endl; // liigu alla paremale
                                lwf();
                                continue; // jätkab nurka minekut algusest (oma veerandis)
                            }
                            xnx(); // kui sai paremale, uued koordinaadid
```

```
                        cout << "M3" << endl; // alla paremale
                        lwf(); // lts; xnx
                        continue;
                    }
                    xnx();
                    continue; // jätka alla vasakule
                }
                cout << "M7" << endl; // liigu üles vasakule
                lts();
                if (nx == x && ny == y) { // kui ei õnnestunud
                    ok[y + 1][x - 1] = 1; // seal on keegi ees
                    fire(7, 6); // tulista
                    cout << "M0" << endl; // liigu üles
                    lts();
                    if (nx == x && ny == y) { // kui ei õnnestunud
                        ok[y + 1][x] = 1; // seal on keegi ees
                        fire(0, 6); // tulista
                        cout << "M1" << endl; // liigu üles paremale
                        lwf();
                        cout << "M7" << endl; // liigu üles vasakule
                        lwf();
                        continue; // jätka
                    }
                    xnx();
                    cout << "M7" << endl; // liigu üles vasakule
                    lwf();
                    continue;
                }
                xnx();
                continue;
            }
        }
        return 1;
    }
```

Edasi järgneb praktiliselt sama kood ülejäänud kolme nurga jaoks, erinevus on vaid liikumissuundades. Ilmselgelt on sellist koodi raske lugeda ja veelgi raskem on selles parandusi teha. Vaatame, kuidas saaks seda koodi lühematesse funktsioonidesse jagada. Esiteks soovitakse valida lähim nurk, kuhu minna. Põhifunktsioon `MineNurka` võiks olla siis järgmine:

```
int mineNurka() {
    int nurkX = (m - x > x) ? 0 : m - 1;
    int nurkY = (n - y > y) ? 0 : n - 1;
    return mineNurka(nurkX, nurkY);
}
```

Siit kutsutakse välja funktsiooni `MineNurka` juba konkreetse nurga koordinaatidega:

```
int mineNurka(int nurkX, int nurkY) {
    int suund = leiaSuund(nurkX, nurkY);
    while (!ok[nurkY][nurkX] && (x != nurkX || y != nurkY)) {
        if (!liiguNurgaSuunas(nurkX, nurkY, suund)) { // kui ei ole üldse liikunud
            poikle(nurkX, nurkY, suund);
        }
    }
    return leiaVastassuund(suund);
}
```

Funktsioonid `leiaSuund` ja `leiaVastassuund` on lihtsad funktsioonid, mis parandavad eelkõige loetavust:

```
int leiaSuund(int x, int y) {
    if (x == 0 && y == 0) return ALLA_VASAKULE;
    if (x == 0) return YLES_VASAKULE;
    if (y == 0) return ALLA_PAREMALE;
    return YLES_PAREMALE;
}
```

```
int leiaVastassuund(int suund) {
    return (suund + 4) % 8;
}
```

Olulisemad on siin funktsioonid `liiguNurgaSuunas` ja `põikle`. Kõigepealt esimene neist:

```
bool liiguNurgaSuunas(int nurkX, int nurkY, int suund){
    bool ol = 0;
    while (x != nurkX && y != nurkY && onVaba(suund)) {
        ol = liiguTulega(suund); //liigu diagonaalis
    }
    int xsuund = nurkX ? PAREMALE : VASAKULE;
    int ysuund = nurkY ? YLES : ALLA;
    bool fm = 1; // forsseeritud liikumine
    while (onLaual() && onVaba(xsuund) && (nurkX - x > nurkY - y || fm)) {
        ol = liiguTulega(xsuund); //liigu horisontaalselt
        fm = 0;
    }

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

Siingi on abifunktsioonid loetavuse parandamiseks `onLaual ja onVaba`:

```
bool onLaual() {
    return (x > -1) && (x < m) && (y > -1) && (y < n);
}
```

```
bool onVaba(int suund){
    return !ok[y + dy[suund]][x + dx[suund]];
}
```

Liikumise funktsioon on järgmine:

```
bool liigu(int suund, bool tulista) {
    cout << "M" << suund << endl;
    lts(); // info sisse, siin loetakse
    if (tulista && nx == x && ny == y) { // kui tulistamisega
        ok[y + dy[suund]][x + dx[suund]] = 1; // seal on keegi ees
        fire(suund, 6); // tulista sinna
    }
    xnx(); // muudab x ja y väärtused
    return true;
}
```

Funktsioonid `liiguTulega` ja `liiguTuleta` on taas loetavuse parandamiseks:

```
void liiguTuleta(int suund) {
    liigu(suund, false);
}
```

```
bool liiguTulega(int suund) {
    return liigu(suund, true);
}
```

Vaatamata on veel funktsioon `poikle`, milles pannakse peamiselt paika põiklemise suunad:

```
void poikle(int nurkX, int nurkY, int suund){
    int pSuund, d;
    if (!nurkX != !nurkY){
        pSuund = (suund + 2) % 8;
        d == 1;
    }
    else {
        pSuund = (suund + 6) % 8;
        d = -1;
    }

    if (nurkX - x < nurkY - y) {
        liiguPoikle(pSuund, d);
    }
    else {
        liiguPoikle(leiaVastassuund(pSuund), -1*d);
    }
}
```

Kõige lõpuks funktsioon `liiguPoikle`:

```
void liiguPoikle(int pSuund, int d) {
    cout << "M" << pSuund << endl; // liigub uues suunas
    lts();
    if (nx == x && ny == y) { //seal ka keegi ees
        ok[y + dy[pSuund]][x + dx[pSuund]] = 1;
        fire(pSuund, 6); //tulista
        int vSuund = (pSuund + d + 8) % 8;
        cout << "M" << vSuund << endl; //liigu jälle uues suunas
        lts();
        if (nx == x && ny == y) { // ikka keegi ees
            ok[y + dy[vSuund]][x + dx[vSuund]] = 1;
            fire(vSuund, 6); // tulista
            liiguTuleta((vSuund + d + 8) % 8); // liigu korraks vastassuunas
            liiguTuleta(pSuund); // ja tagasi põiklemise suunas
            return; // jätka nurka minekut algusest peale (oma veerandis)
        }
        xnx();
        liiguTuleta(pSuund);
        return;
    }
    xnx();
    return;
}
```

Kokku õnnestus kahandada üks umbes 400-realine funktsioon vähem kui 150le reale, mis on omakorda jagatud üsna mõistlikult arusaadavateks kirjeldava nimega osadeks.

7.1.4 Kommentaarid

Sageli võib programmeerimisõpikutest lugeda, et kood tuleb kommenteerida. Üheks soovituseks on kirjutada enne kommentaarid, mida kood võiks teha ning seejärel alles hakata koodi kirjutama. See on hea praktika, mis aitab oma mõtteid koondada ning ülesandest tervikpilti saada. Võib juhtuda, et esialgne idee oli üldse vildakas, ja palju parem on sellest aru saada pärast paari rea kommentaaride kirjutamist, mitte alles siis, kui kirjutatud juba hulk koodi.

Pahatihti minnakse aga kommenteerimise soovitustega liiale, näiteks paljudes ettevõtetes on kohustuslik kommenteerida iga meetod, parameeter, muutuja jne. Tulemuseks on nonsenss:

```
/*
** Tagastab kõrguse.
*/
double getHeight()
```

Sellistel kommentaaridel on ka suur oht vananeda. Kui nõuded muutuvad, võib ka kirjeldatava objekti olemus muutuda, kuid kommentaar jääb ikka samaks ning hakkab kasu asemel tooma kahju.

Näiteks leidsin ma koodist kord sellised read:

```
// ISO 3166 two letter country code
string threeLetterCode;
```

Uh-oh. Ilmselgelt oli millalgi kahetähelistelt koodidelt mindud üle kolmetähelistele ning kommentaar, mis enne oli olnud lihtsalt kasutu, muutus nüüd aktiivselt kahjulikuks.

> Kommentaar, mis ütleb midagi, mis koodis niigi kirjas on, ei lisa mingit väärtust.

Kommentaarid võivad olla vajalikud, et kirjeldada **miks** kood on mingil viisil kirjutatud. Kui aga juhtub, et me peame kommenteerima, **mida** kood teeb, saab selle asemel tavaliselt parandada koodi loetavust.

> Iga kommentaar on märgiks sellest, et kood ei kirjelda iseennast perfektselt ja seda peaks käsitlema kui võimalust koodi parandada.

Näiteks sellise koodi

```
// konverteerime eurod sentideks
a = x * 100;
```

asemel võib anda muutujatele paremad nimed:

```
sendid = eurod * 100;
```

Järgmine halb näide (kood on lihtne, aga mis maagiat siin tehakse?)

```
double r = n / 2;
while (abs(r - (n / r)) > t) {
    r = 0.5 * (r + (n / r));
}
```

Parem variant:

```
// leiame ruutjuure Newton-Raphsoni meetodiga
double r = n / 2;
while (abs(r - (n / r)) > t) {
    r = 0.5 * (r + (n / r));
}
```

Veel parem:

```
double LigikaudneRuutjuur(double n, double tapsus) {
    double r = n / 2;
    while (abs(r - (n / r)) > tapsus) {
        r = 0.5 * (r + (n / r));
    }
    return r;
}
```

Kommentaari asemel eraldi funktsiooni loomisega saavutasime kaks eesmärki:

- Eraldasime kasuliku funktsionaalsuse taaskasutatavaks alamprogrammiks.
- Andsime sellele kasuliku, iseennast kirjeldava nime, mis võimaldab koodist arusaamist ka seal, kus vastavat funktsiooni välja kutsutakse.

Samuti on tavapärane (aga kahjulik) nõuanne kirjutada iga muutuja juurde, mis on selle sisu. Selmet kirjutada iga muutuja juurde, milleks seda kasutatakse, tuleb nimetada muutuja kohe nii, et selle otstarve on selge.

7.1.5 Nimed

Kunagisele Netscape'i tarkvaraarhitektile Phil Karltonile omistatakse järgmist tsitaati:

„Tarkvaraarenduses on ainult kaks suurt probleemi:

1. puhvrite invalideerimine,
2. asjade nimetamine,
3. ühe võrra eksimused."

Originaalis: „There are only two hard things in software engineering: cache invalidation, naming things, and off-by-one errors."

Siin siis veidi näpunäiteid, kuidas teist probleemi paremini lahendada.

Phil Karlton

- Mida pikem on kood, kus muutujat kasutatakse, seda raskem on järge pidada, mida muutuja teeb. Sellest järeldub **skoobi reegel**: mida pikem on skoop, kus muutujat kasutatakse, seda pikem peaks olema muutuja nimi. Tsüklimuutujatena on sobilik kasutada `i` ja `j`, samuti vaid ühes lühikeses tsüklis kasutatav abimuutuja võib olla ühetähelise nimega. Globaalsete muutujate puhul seevastu on hea, kui neil on ilusad kirjeldavad nimed. Koodi kerimine selleks, et uurida, mis see t või p täpselt oli, on selgelt ebaotstarbekas.
- Muutujad vastavad tavaliselt mingitele asjadele või objektidele ning neid võiks nimetada nimisõnadega, nt `int[][] laud`.
- Veidi erandlikuks on `boolean` tüüpi muutujad, mida sobib nimetada pigem omadus- või määrsõnadega, soovi korral võib lisada ette ka tegusõna 'on' nt `valmis` või `onValmis`; `kiire` või `onKiire`.

Järgneb koodinäide Lemmingute ülesandest, kus esimeses tulbas on kirjeldavate nimedega kood, teises lühikestega:

```
while (!tramm.empty() &&
  ((trammisabad[peatus].size() < jk_pikkus) ||
  tramm.top() == peatus)) {
    if (tramm.top() != peatus) {
        trammisabad[peatus].push(tramm.top());
    }
    tramm.pop();
    aeg++;
}
```

```
while (!s.empty() &&
  ((q[p].size() < 1) ||
  s.top() == p)) {
    if (s.top() != p) {
        q[p].push(s.top());
    }
    s.pop();
    t++;
}
```

Vastuväide oleks, et pikemate nimede kirjutamine võtab rohkem aega, kuid esiteks on tänapäeva programmeerimiskeskkondades abistavad vahendid nimede automaatseks lõpetamiseks (*auto-complete*) ning teiseks on natuke pikema nime kirjutamisele kuluv aeg üsna väike võrreldes probleemi sisulisele lahendamisele ja programmi struktuuri loomisele kuluva ajaga. Võrreldes aga ajaga, mis kuluks vigade otsimisele, sest programm on muutunud ka autorile endale arusaamatuks, on need ekstra sekundid aga täiesti tühised.

Võistlusprogrammeerimise kontekstis võib ja on isegi soovitatav kasutada ülesande tekstis antud muutujate nimetusi (tavaliselt mingid N, M, K jne), see tähendab ka ühetähelisi tähistusi.

7.1.6 Funktsioonide nimetamine

Funktsiooni nimetused peaks ütlema, mida funktsioon teeb. Kui muutujatele sobivad nimed olid nimisõnad, siis funktsioonid on tegevused ja nende nimed võiks olla tegusõnad. Väga hästi sobivad näiteks `vaheta`, `otsi`, `looGraaf`, `leiaLyhimTee` ja muud sarnased nimed. Erandiks on funktsioonid, mille peamiseks otstarbeks on tõeväärtuse tagastamine, neid võiks nimetada sarnaselt `boolean`-tüüpi muutujatega, näiteks

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

Seda laadi nimetused teevad koodi hästi loetavaks:

```
if (onLaual(naaber_x, naaber_y, dim) {
    teeMidagi();
}
```

Ka funktsiooni parameetrid võiks olla funktsiooni päises mõistlikult nimetatud. Nt funktsioon päisega

```
long long liida(int a, int b);
```

võiks eeldatavasti liita kokku kaks argumendiks saadavat arvu a ja b, aga funktsioon päisega

```
long long liida(int start, int end);
```

liidab pigem mingit arvude vahemikku või mingit vahemikku jadas. Ainus erinevus on argumentide nimetustes! Või kui ülaltoodud funktsioon oleks päisega

```
bool onLaual(int a, int b, int c);
```

võib endalgi sassi minna, kas enne peaks ette andma koordinaadid ja siis laua mõõdu või vastupidi.

7.2 TESTIMINE

Võistlustel kasutatakse programmide hindamisel tavaliselt järgmisi meetodeid:

- **Testide kaupa** hindamine, kus punkte saab iga läbitud testi eest, mida rohkem teste ära teed, seda rohkem punkte. Enamik Eesti madalama taseme võistlusi kasutab seda lähenemist.

- „**Pakikaupa**“ testimine, kus lahendus peab punktide saamiseks läbima kõik testid. Õige lahendus annab 100% ja kui kasvõi üks test ei lähe läbi, on tulemuseks 0. Seda skeemi kasutatakse enamikel *online* võistlustel (CodeForces, TopCoder jne).
- Testid on jagatud **alamülesanneteks**, iga konkreetset alamülesannet testitakse pakikaupa. Seda kasutatakse enamikul rahvusvahelistel võistlustel nagu IOI (the International Olympiad in Informatics)või BOI (Baltic Olympiad in Informatics).
- **Avatud testid**, kus esitada tuleb testide vastused, mitte programm, mis ülesannet lahendab. Inglise keeles nimetatakse neid „*output-only*“ ülesanneteks.

Loomulikult ei ole testimine ainult võistluste teema – seda on vaja ka päris elus, sest kes soovib kasutada programmi, mis töötab ainult teatud juhtudel.

Kuidas siis ikkagi testida? Selleks on muidugi vaja programmi, mida testida, teste ehk andmestikku, millega testida, ning teada õigeid vastuseid, millega testitava programmi tulemusi võrrelda. Testide koostamise jaoks on vaja kõigepealt mõelda välja, millised testid võiks olla, ning juhuslike testide tegemiseks kasutada testigeneraatorit. Õige vastuse saamiseks sobib võimalusel nii käsitsi kontrollimine, aga ka mõni lihtsam programm, nt selline, mis ülesande jõumeetodil lahendab, kuid pole võistlusel esitamiseks piisavalt kiire. Veel üheks võimaluseks on kasutada lahenduse leidmiseks testigeneraatorit – st koostamise käigus saab leida vastuse või mõnikord on kavalam genereerida testid oodatavast vastusest lähtuvalt.

Selleks, et testimist veelgi efektiivsemalt läbi viia, tulevad kasuks skriptid, mis teste käivitavad ja tulemusi kontrollivad.

7.2.1 Testide koostamine

Milliseid teste üldse koostada? Üldiselt võib testid jagada kolme erinevasse liiki:

- Ülesande teksti analüüs. Siia kuuluvad maksimaalsete ja minimaalsete väärtustega testimine, testid tühjade väärtuste ja nullidega, samuti testid, mis kontrollivad väljundi korrektset formaatimist. Oluline on ka kontrollida, kas piirangutest on õigesti aru saadud, näiteks kas võrratus on range või mitterange.
- Ülesande sisu analüüs, millest võiks tulla testid erijuhtude ja piirjuhtude jaoks.
- Oma programmi teksti analüüs, nulliga jagamised, ületäitumised, ujukomaarvude täpsus.

Järgmine ülesanne on avatud testidega ülesanne Eesti Lahtiselt Võistluselt aastast 2015/16:

7.2.2 Ümbermõõt

Informaatikaolümpiaadi ettevalmistuslaagris anti osalejatele lahendada järgmine ülesanne:

Ristkülikute ühendi ümbermõõt

Tasandil on antud `N` ristkülikut, mille servad on koordinaattelgedega paralleelsed. Leida nende ristkülikute ühendi perimeeter ehk välise kontuuri pikkus. Tekkinud kujundi sees olevaid auke ei ole vaja arvesse võtta, kuid ühe ristkülikutest koosneva kujundi augus asuvat teist riskülikutest koosnevat kujundit, mis väliskujundit ei puuduta üheski punktis, tuleb lugeda ikkagi omaette kujundiks.

Sisend: Faili esimesel real on antud ristkülikute arv `N (1≤N≤1000)`. Järgmisel `N` real on igaühel neli reaalarvu `LX`, `LY`, `UX` ja `UY`, kus `LX` ja `LY` on ristküliku alumise vasaku ning `UX` ja `UY` ülemise parema nurga koordinaadid. Koordinaatide väärtused on `0` kuni `1 000 000`, mille esituseks piisab 32-bitisest `float` arvutüübist.

Väljund: Faili ainsale reale väljastada leitud ristkülikute ühendi perimeeter täpsusega kuni kolm kohta pärast koma.

Näide: Sisendfail

```
3
7.0 170.0 99.5 190.0
0.5 100.0 12.0 225.0
10.0 80.0 50.0 110.0
```

Väljundfail:

```
564.0
```

Teie ülesandeks on koostada testid, mis kontrolliksid esitatud lahenduste korrektsust.
Kõik testandmed koondada ühte tekstifaili järgnevalt kirjeldatud formaadis. Faili esimesel real on testide arv `K (1≤K≤20)`. Järgnevas `K` blokis on bloki esimesel real oodatav tulemus ning selle järel sisendandmed samal kujul kui ülesande kirjelduses. Lahendusena tuleb esitada see tekstifail.

NÄIDE:

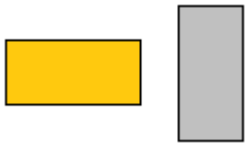
```
2
4.0
1
1.0 1.0 2.0 2.0
564.0
3
7.0 170.0 99.5 190.0
0.5 100.0 12.0 225.0
10.0 80.0 50.0 110.0
```

Ülesande sisu analüüsides tuleks kindlasti teha testid, mis kasutavad minimaalset ja maksimaalset lubatavat ristkülikute arvu, mis tähendab, et võiks olla test, milles on vaid üks ristkülik, ja test, milles ristkülikuid on täpselt tuhat.

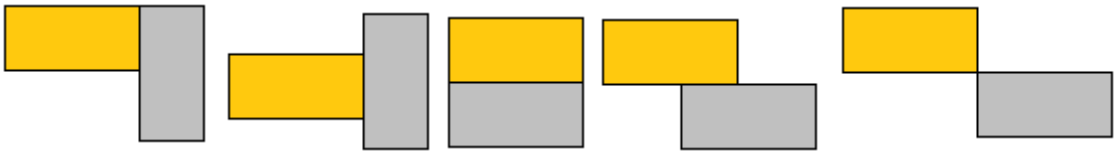
Kuna koordinaatideks võivad olla positiivsed reaalarvud, siis ei piisa testimisel kindlasti ainult täisarvulistest koordinaatidest, vaid tuleks kasutada nii väga väikeseid arve, paljude kohtadega arve, aga ka loomulikult maksimaalset lubatud koordinaadi väärtust.

Väljundi formaadi juures saaks siin kontrollida, kas vastus antakse ikka ettenähtud täpsusega kuni kolm kohta pärast koma. Seda küll raamülesandes ettekirjutatud formaat testida ei võimalda, kuid ise sellist ülesannet lahendades tuleks küll sellele tähelepanu pöörata.

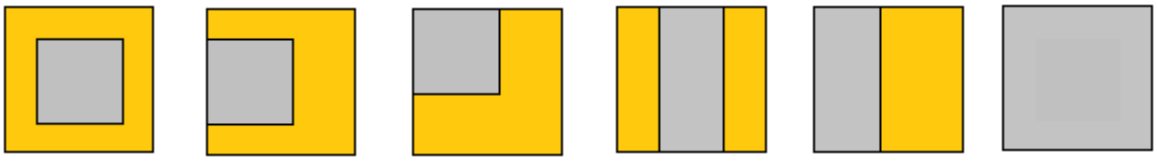
Asume nüüd ülesande sisulise analüüsi juurde. Vaatame kõigepealt, kuidas saavad 2 ristkülikut teineteise suhtes paikneda. Esimene võimalus on, et ristkülikud üldse kokku ei puutu:



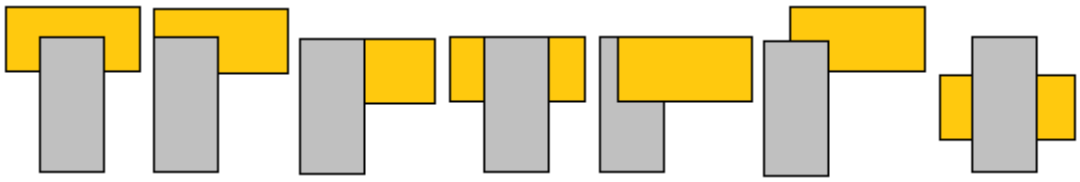
Puutuvatest, kuid mitte kattuvatest ristkülikutest koosnevad erinevad variandid:



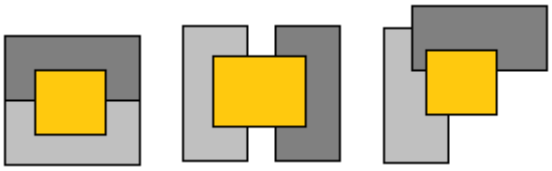
Üks ristkülik võib asuda ka täielikult teise sees. Kui arvestada ka võimalike puutumistega, saame järgmised võimalused (hall ristkülik on tervenisti oranži ristküliku sees):



Lõikuvad ristkülikud:

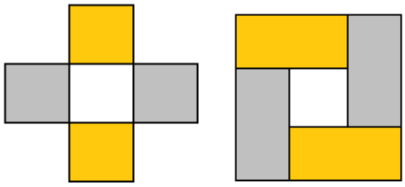


Nii on juba kahe ristküliku omavahelise paiknemise jaoks paarkümmend erinevat võimalust, kolmanda lisamine lisab veel hulga võimalikke kombinatsioone. Kõiki neid ei jõua siin ära tuua. Kolme ristkülikuga võimalustest huvitavamad on järgmised paiknemised:



Vasakpoolsel paikneb kolmas ristkülik tervenisti kahe teise ristküliku sees (aga mitte kummagi sees), parempoolsel on kattumine mõlema teise ristkülikuga, kõik kolm osalevad ümbermõõdu arvestamises, kolmandal on kõigil paarikaupa kattuvaid osi kui ka osa, mille katavad kõik kolm ristkülikut.

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



Kui lisada veel ristkülikuid, saab tekitada ühe kujundi sees ka mitu tühja ala. Kindlasti tasuks testida ka rohkemate kujunditega tekkivaid formatsioone, näiteks spiraali.

Nüüd on siin terve hulk ideid välja käidud, ülesandes lubatud testide arv on aga vaid 20. Mida sellisel juhul teha? Kuna eesmärgiks on ainult viga välja meelitada, siis võib mitu erinevat kombinatsiooni ühte testi panna. Loomulikult tuleb hoolega läbi mõelda, millised on mõttekas ühte testi panna, millised aga võivad nii mõne vea hoopis ära varjata. Ilmselge on, et ei saa ühes testis testida, kas programm töötab nii minimaalse kui ka maksimaalse ristkülikute arvu korral. Samuti enda programmi jaoks teste tehes on parem erinevad juhud eraldi testides hoida, kuna nii saab vea korral kiiremini jälile, millise juhtumiga programm hakkama ei saa.

See näide oli selleks, et tuua välja erinevaid aspekte testide koostamisel ja näidata, kuidas pealtnäha küllaltki lihtsal ülesandel võib olla üsna palju erinevaid juhte, millega on võimalik eksida. Järgevalt on toodud, milles siis ülesande programm tegelikult eksis, mis oleks tulnud lahendajal tuvastada:

- Lahendus ei erista välimist ja sisemist kontuuri.
- Lahendus töötab valesti, kui arvutuste täpsus on suurem kui 1 koht pärast koma.
- Lahendus ei leia kõigi väliste kontuuride perimeetrit, kui väliseid kontuure on rohkem kui üks.
- Lahendus töötab valesti, kui ristkülikute arv on 1000 ja viimane ristkülik osaleb perimeetri moodustamises.
- Lahendus töötab valesti, kui perimeetri pikkus ületab 16-bitilise arvu maksimumväärtust.
- Lahendus töötab valesti, kui leidub vähemalt 2 täpselt samasugust ristkülikut.
- Lahendus töötab valesti, kui leidub vähemalt 3 ristkülikut, mille vertikaalsed küljed on samal x-koordinaadil.
- Lahendus töötab valesti, kui leidub vähemalt 3 ristkülikut, mille horisontaalsed küljed on samal y-koordinaadil.
- Lahendus töötab valesti, kui kontuuris leidub külgede jada, kus on järjest 10 või enam ühesuunalist pööret (spiraal).
- Lahendus töötab valesti, kui üks välimine kontuur asetseb täielikult teise välimise kontuuri sees.

7.2.3 Testide genereerimine

Enamik ülesandeid õnneks nii põhjalikku läbimõtlemist ei vaja, kuid erineva andmestikuga testimine on siiski vajalik. Kui mõned spetsiifilisemad juhud on vaadatud, on mõttekas genereerida ka juhuslikke teste.

Viiendas peatükis oli üheks näiteks ülesanne „Miljonär ja vaeslapsed“:

Dickensi-aegsel Inglismaal elas miljonär Mortimer. Temaga samas linnas asusid kolm lastekodu, kus elasid vaeslapsed, kellele Mortimer tavatses jõulukinke teha. Kinkide jagamise protseduur oli järgmine:
1. Iga vaeslaps saab oma lastekodust korvi, millega ta kingi järele läheb.
2. Mortimer viskab kingitusi järjest laste sekka, mida nood oma korvidega püüavad.
3. Iga kingitus püütakse alati kinni.
4. Lapsed saavad kingitusi kätte juhuslikult, kuid tõenäosus, et konkreetne laps kingituse kätte saab, on võrdeline tema korvisuu pindalaga.
5. Sama lastekodu lastel on sama suurusega korvid.
6. Kui mõni laps saab kingituse kätte, läheb ta sellega kohe lastekodusse tagasi ja rohkem püüdmises ei osale.
7. Lapsi võib olla rohkem kui kingitusi :(
Igal kingitusel on väärtus. Leida iga lastekodu kohta, milline on selle kodu laste poolt saadud kingituste väärtuste keskmine eeldatav summa.
Sisendi esimesel kolmel real on igaühel kaks täisarvu L_i ($0 \le L_i \le 100$) ja K_i ($1 \le K_i \le 100$), mis tähistavad reanumbrile vastava lastekodu laste arvu ning korvi pindala. Sisendi neljandal real on kingituste arv N ($1 \le N \le L_1+L_2+L_3$). Viimasel real on N ($1 \le N \le 1000$) täisarvu: esimesel neist esimese kingituse väärtus, teisel teise jne.
Väljastada kolm reaalarvu (täpsusega vähemalt $0{,}0001$) kolmel real: iga lastekodu kohta, milline on selle kodu laste poolt saadud kinkide väärtuste keskmine eeldatav summa.

NÄIDE:

```
1 1
1 2
1 2
2
10 20
```

Vastus:

```
6,666667
11,333333
12
```

Selle ülesande juhuslike testide generaator võib olla näiteks selline:

```
int main()
{
    const string NIMI = "test";
    const int TESTE = 20;

    for (int i = 0; i < TESTE; i++) {
        ofstream fail;
        string fnimi = NIMI + to_string(i) + ".txt";
        fail.open(fnimi);
        int lapsi = 0;
        for (int i = 0; i < 3; i++) {
            int la = rand() % 101;
            lapsi += la;
            fail << la << " ";
            fail << 1 + rand() % 100 << endl;
        }

        int N = 1 + rand() % lapsi; // kinkide arv
        fail << N << endl;
        for (int i = 0; i < N - 1; i++) {
            fail << 1 + rand() % 1000 <<" "; //kinkide väärtused
        }
        fail << 1 + rand() % 1000 << endl;
        fail.close();
    }
    return 0;
}
```

Selle programmi väljundiks on 20 faili nimedega test0...test19 juhuslike andmetega, mis vastavad sisendi formaadile. Kui proovida need testid läbi lahendusega, siis saame teada ainult seda, kas me programm töötab etteantud ajalimiidi piires. Selleks, et kontrollida, kas saadud lahendus on ka korrektne, oleks vaja teada õiget lahendust. Siin tuleb appi jõumeetod ehk kõikide variantide läbivaatus:

```
int N;
int* kingid;
int K[3];
int L[3];
double vastus[3] = { 0 };

void jagaKingid(int kingiNr, int korvidePindala, double Tn)
{
    if(kingiNr == N) return; //kõik on jagatud

    for (int i = 0; i < 3; i++) {
        if (L[i] > 0) {
            double tn = Tn * L[i] * K[i] / korvidePindala;
            vastus[i] += kingid[kingiNr] * tn;
            L[i]--;
            jagaKingid(kingiNr + 1, korvidePindala - K[i], tn);
            L[i]++;
        }
    }
}
```

Kui meil just aastaid aega ei ole, siis tuleb aga sisendi limiite oluliselt kärpida, sest selle jõumeetodi keerukus on 3^N ja seega ei ole mõtet ette anda kuigi suurt N-i. Samas, kui me saame identsed vastused nii DP lahendusega kui ka jõumeetodiga kümnekonnale juhuslikule testile, siis on üldjuhul normaalne eeldada, et testitav DP lahendus annab korrektse tulemuse ka suuremate testide puhul. Eelpool toodud testide genereerimise algoritmis tuleks muuta kinkide arvu leidmise rida vastavalt:

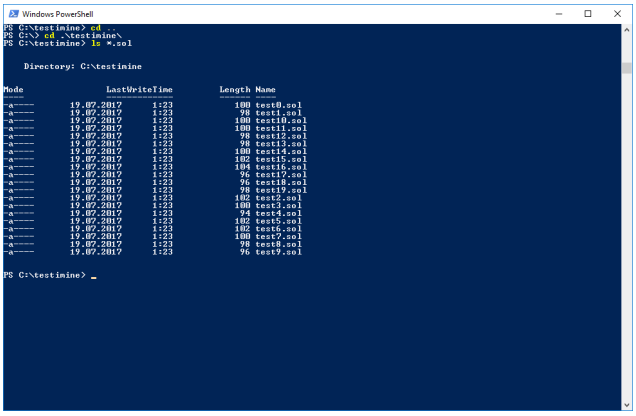
```
int N = 1 + rand() % min(lapsi, 15); // kinkide arv
```

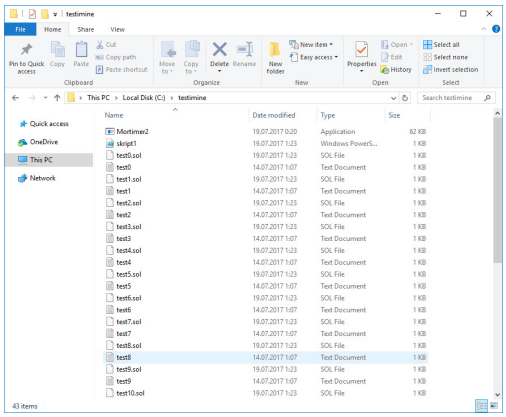
Pane tähele, et laste arvu (ja ammugi mitte kinkide väärtuste) piiramine ei ole vajalik.

7.2.4 Käsuinterpretaatorid ja skriptimine

Kui testid valmis, tuleb need ka läbi jooksutada. Üldine stsenaarium on selline, et iga testi jaoks tuleb käivitada programmid vastava sisendiga ja võrrelda väljundit etaloniga. Kui teste on palju, on kõigi nende käivitamine käsitsi päris aeganõudev, ning kui tulemusena tuleb programmis välja mõni viga, mis parandamist tahab, tuleb programm pärast parandust taas üle testida.

Õnneks saab neidki tegevusi automatiseerida, kasutades oskuslikult mõnd **käsuinterpretaatorit**. Käsuinterpretaator on tekstipõhine käsutöötlusprogramm, mille abil saab suhelda operatsioonisüsteemiga. Levinud käsuinterpretaatorid on näiteks Bash (Linux) ja PowerShell (Windows).





Vasakpoolsel pildidl on PowerShell (käsurealiides) ja parempoolsel File Explorer (graafiline liides).

*Shell*iks ehk **kestprogrammiks** nimetatakse programmi, mis võimaldab inimesel operatsioonisüsteemi funktsionaalsust kasutada (nt programme käivitada). Enamasti peetakse *shell*idest rääkides silmas tekstimoodis käsuinterpretaatoreid, kuid tegelikult on selleks ehitatud graafilised programmid (nt Windows Explorer) samuti kestprogrammid.

Operatsioonisüsteemiga suhtlemiseks kasutatakse **käske** (*commands*, PowerShellis *cmdlets*). Siin on tabel mõningate testide jooksutamiseks põhiliselt vajaminevate käskudega:

Käsk	Bash	PowerShell cmdlet	PowerShelli aliased
Kataloogi sisu	ls	Get-ChildItem	ls, dir, gci
Kataloogi muutmine	cd	Set-Location	cd, chdir, sl
Faili liigutamine	mv	Move-Item	mv, move, mi
Faili kopeerimine	cp	Copy-Item	cp, copy, cpi
Faili väljastamine	cat	Get-Content	cat, type, gc
Standardvoosse kirjutamine	echo	Write-Output	echo, write
Failide võrdlemine	diff	Compare-Object	diff
Kellaaeg	time	Get-Date	

Näiteks

```
PS C:\testimine> cp test2.txt sisend.in
```

Kopeerib faili `test2.txt` sisu uude faili nimega `sisend.in`

Käske täidetakse järjest ja igal käsul võib olla sisend ja väljund, mis vaikimisi loetakse ja kirjutatakse standardvoogudesse. Eelmise käsu väljundi saab suunata järgmise käsu sisendiks kasutades käskude vahel märki „|“. Sel moel suunamist nimetatakse inglise keeles ***piping***uks ja nii suunatakse edasi ainult standardväljund.

Vooge on võimalik suunata ka üksteise vahel. Standardväljundi saab suunata faili: ’>’ kirjutab standardväljundi voo faili juurde ja ’>>’ kirjutab faili üle.

Näiteks

```
PS C:\testimine> cat sisend.in | programm.exe >> vastus.out
```

väljastab faili `sisend.in` sisu ja suunab selle programmi `programm` sisendiks ning lõpuks suunab programmi standardväljundi faili `vastus.out`. Kui selle nimega fail on olemas, kirjutatakse selle sisu üle.

Nii Bash kui ka PowerShell toetavad skriptimist. Skript on faili valmis kirjutatud käsk või käskude jada, mida käsuinterpretaator teostab. Lisaks käskudele saab kasutada ka muutujaid, tingimuslauseid ja tsükleid. PowerShelli skriptide laiendiks on .ps1 ja Bashi skriptidel kasutatakse laiendit .sh

PowerShellis muutujate nimed algavad dollari märgiga $, Bashis deklareerimisel $ märki ei kasutata, küll aga muutuja väärtuse kasutamisel, näiteks:

PowerShell

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

Bash

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

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

PowerShell	Bash

`if tingimus {teeMidagi}`	`if tingimus`
	`then teeMidagi`
	`fi`

`foreach ($i in mingiHulk) {`
`  teeMidagi`
`}`
	`for i in mingiHulk`
	`do`
	`teeMidagi`
	`done`

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

Rohkem infot Bashi ja PowerShelli käskude ja süntaksi kohta leiab näiteks lehelt https://ss64.com/.

Vaatame siin veel paari kõige tõenäolisemat testimise stsenaariumit.

Oletame, et testid on `.in` failides ja nende vastused `.out` failides, näiteks on testfailid `test1.in`, `test2.in` jne, ja vastavate testide vastused on failides `test1.out`, `test2.out` jne. Siis on Bashi testijooksutamisskript järgmine:

```
for f in *.in # iga .in laiendiga failiga
  do bn=`basename $f .in` # bn on ilma laiendita failinimi
  # käivita programmi ’myprog’ sisendiga failist f ja suunab väljundi faili,
  # millel on sama nimi, mis failil f ja laiendiks on ’.sol’
  cat $f | myprog > $bn.sol
  diff $bn.out $bn.sol # võrdleb näidisvastusega faili programmi lahendusega
done
```

Ja sama PowerShellis:

```
$failid=@(Get-ChildItem .\*.in)
foreach ($f in $failid) { # iga in laiendiga failiga
  $bn= $f.basename # bn on ilma laiendita failinimi
  # käivita programmi ’myprog’ sisendiga failist f ja suunab väljundi faili,
  # millel on sama nimi, mis failil f ja laiendiks on ’.sol’
  $uusnimi = $bn + ".sol"
  cat $f | .\myprog.exe > $uusnimi
  # võrdleb näidisvastusega faili programmi lahendusega
  diff (cat $bn.out) (cat $bn.sol) #kuna PowerShell võrdleb vaikimisi objekte,
  # siis faili sisu võrdlemiseks tuleb anda argumentideks failide sisud
}
```

Kui sisend ja väljund tuleb lugeda/kirjutata etteantud nimega faili, olgu need näiteks `ylesanne.sis` ja `ylesanne.val` on skript järgmine:

```
for f in *.in
do
  cp $f ylesanne.sis # kopeerib faili f faili ’ylesanne.sis’
  myprog # käivitab programmi. Õige sisend on olemas eelmisest käsust,
  # tekib väljund ’ylesanne.val’
  bn=`basename $f .in`
  diff $bn.out ylesanne.val
done
```

Ja sama PowerShellis:

```
$failid=@(Get-ChildItem .\*.in)
foreach ($f in $failid) {
  cp $f ylesanne.sis # kopeerib faili f faili ’ylesanne.sis’
  ./myprog.exe # käivitab programmi. Õige sisend on olemas eelmisest käsust,
  # tekib väljund ’ylesanne.val’
  $un= $f.basename + ".out"
  diff (cat $un) (cat $ ylesanne.val)
```

7.2.5 Valideerimine

Kuna `diff` võrdleb faile märk-märgilt, siis mõnel juhul on kasulikum kirjutada oma validaator, mis vastuseid võrdleb. Üheks selliseks juhuks on muidugi see, kui vastuseks tuleb anda reaalarv, nagu „Miljonäri ja vaeslaste“ ülesandes.

```
int main(int argc, char* argv[])
{
    ifstream fail1, fail2;
    fail1.open(argv[1], ifstream::in);
    fail2.open(argv[2], ifstream::in);
    double delta = 0.0001;
    double vastus1, vastus2=0;
    for (int i = 0; i < 3; i++){
        fail1 >> vastus1;
        fail2 >> vastus2;
        if (abs(vastus1 - vastus2) > delta) {
            cout << "vale vastus" << endl;
            break;
        }
    }

    fail1.close();
    fail2.close();
    return 0;
}
```

Reaalarvud ja nende täpsus ei ole aga sugugi ainus koht, kus oma validaatorit vaja võib minna. Siin on veel mõned juhtumid:

- Vastus koosneb mitmest täisarvust või stringist, aga nende järjestus ei ole üheselt määratud. Sellisel juhul peaks validaator enne võrdlemist mõlemad vastused sorteerima sama kriteeriumi järgi.
- Ülesandel võib olla mitu õiget vastust. Üheks võimaluseks on etalonvastusena anda kõik võimalikud vastused ja validaator kontrollib, kas programmi vastus sobib ühega neist. Teise võimalusena võib validaator kontrollida, kas antud vastus sobib ülesande lahendiks.

7.3 TUNNE OMA KEELT

Hoolimata sellest, et programmeerimiskeeltel on palju sarnaseid omadusi ja võimalusi, on konkreetsetel keeltel ka spetsiifilisi võimalusi, mille tundmine suureks kasuks võib tulla – just nagu osav käsitöömeister suudab oma tööriistadega rohkem korda saata kui algaja.

7.3.1 Teegid

Suur osa programmeerijale suurepärastest abivahenditest asub keele standardteekides, nende võimaluste tundmine teeb programmi kirjutamise kiiremaks ja efektiivsemaks, kuna sageli päästab jalgratta leiutamisest või hoiab juhendi lugemisele kuluvat aega kokku.

Keelte võimalused on siinkohal erinevad ning igal keelel on omad tugevused ja nõrkused, mistõttu on hea tunda rohkem kui üht keelt. Kuigi üldjuhul on C++ võistlustel eelistatud, on ülesandeid, kus Java või Pythoni kasutamine on optimaalsem. Arvuteooria peatükis tuli suurte arvudega arvutamise juures C++ keeles kirjutada hulk funktsioone, mis Java `BigInteger` klassis olemas on, ning isegi Python suudab kiiresti arvutada suurte arvudega ilma, et programmeerija omalt poolt midagi erilist tegema peaks. Samuti võib C++ tekstitöötlus olla ebamõistlikult keeruline.

7.3.2 Andmestruktuurid

Andmestruktuurid on väga võimsad asjad, aga seda siis, kui neid õigel ajal õiges kohas kasutada. Klasside ja kirjete loomine on tore ning sageli aitab kaasa koodi loetavusele, samas ei maksa unustada ka võimalikke lihtsamaid alternatiive. Enne kirje loomist aga tasub läbi mõelda, kas see tegelikult ka midagi juurde annab võrreldes sama info hoidmisega mitmemõõtmelises või mitmes massiivis, seda muidugi eriti C/C++ kasutades. Näiteks kui ülesandes on vaja töödelda punktihulka, siis esimese mõttena võib tulla pähe luua kirje `Punkt` muutujatega `X` ja `Y`, kuid see ei pruugi anda eelist kahe massiivi `X` ja `Y` ees.

7.3.3 Makrod

Võrreldes enamiku tänapäeval kasutatavate keeltega on C/C++ keelel järgmine huvitav omadus: teksti **eeltöötlus** ehk preprotsessimine. Java, Python ja enamik teisi tänapäeva keeli võimaldavad koodi pakendamist moodulitesse, mille kohta on teada, milline funktsionaalsus selles realiseeritud on. Kui nüüd teises programmis on vaja seda moodulit kasutada, viidatakse sellele nime alusel, `import` käsu abil. C/C++ puhul on selle aseme **päisfailid**, mis sisaldavad kasutatavate funktsioonide deklaratsioone ja mille tekst lisatakse `#include` käsu abil otse kirjutatava programmi sisse - preprotsessor lihtsalt **asendab**

`#include` käsu vastava faili sisuga. Selliseid tekstioperatsioone nimetataksegi eeltöötluseks – selle tulemusena valmib lõplik programmi tekst, mida kompilaator tegelikult kompileerima hakkab.

Eeltöötluse käigus saab teha ka mitmeid muid tekstioperatsioone, näiteks defineerida, et mõni lekseem peab tähendama hoopis midagi muud. Selliseid asendusi nimetatakse **makrodeks.**

Mõned lihtsad makrod:

```
#define PI 3.14159
```

asendab kõik kohad koodis, kus kasutatakse konstanti `PI` vastava arvuga.

```
#define ll long long
```

võimaldab selle asemel, et igal pool kirjutada välja tüübinimi `long long` kasutada lühemat vormi „`ll`“.

Makrod võimaldavad ka avaldiste kasutamist, näiteks nii:

```
#define ruut(x) x*x
```

Selles näites on aga ohtlik viga, mis illustreerib makro ja tavalise funktsiooni erinevust! Mis juhtub, kui kirjutada näiteks:

```
int a=2, b=3;
cout << ruut(a+b);
```

Võiks arvata, et vastus on 25, kuid tegelikult asendatakse `ruut(a+b)` avaldisega `a+b*a+b`, mis annab tulemuseks hoopis 2+3*2+3 = 11. Sellise probleemi vältimiseks tuleb argumendid panna sulgudesse:

```
#define ruut(x) (x)*(x)
```

Makrod on võimas vahend, mida on kerge kuritarvitada, sest põhimõtteliselt võib kirjutada ka

```
#define 1 0
```

ja teisi hullumeelsusi. Suurem oht seisneb siiski selles, et makrode abil võib kirjutada väga keerulist ja arusaamatut koodi, mida on raske hallata ning debugida. Pole juhus, et „segase koodi võistlusi“ (*obfuscated code contest,* http://www.ioccc.org/) korraldatakse peamiselt C keeles, kus võistlustööd kasutavad ohtralt makrosid.

Võistlustel seisneb makrode kasutamise peamine väärtus võimaluses korduvaid asju lühemalt kirja panna, siin on mõned praktilised näited:

```
#define PB push_back
#define MP make_pair
#define sz(v) (in((v).size()))
#define forn(i,n) for(in i=0;i<(n);++i)
#define forv(i,v) forn(i,sz(v))
#define fors(i,s) for(auto i=(s).begin();i!=(s).end();++i)
#define all(v) (v).begin(),(v).end()
```

7.4 VÕISTLUSTE STRATEEGIA

Tänapäeval toimub palju erinevas formaadis võistlusi mitmesuguste raskusastmete, ajapiirangute ja testimissüsteemidega, mistõttu on ka võistlemise strateegiad erinevad.

Siin on siiski mõned üldised näpunäited. Peamine reegel on „parem pronks peos kui kuld katusel“ – liiga ambitsioonikas kohe raskete ülesannete kallale asumine on nii mõnelegi võistlejale tulemuse maksnud.

- Kui ülesannete eeldatav raskusjärjestus pole teada (nt IOI, kus kõik ülesanded annavad sama palju punkte), **loe enne lahendama asumist kõik ülesanded korralikult läbi**. Võib juhtuda, et kuigi esimene ülesanne tundub huvitav või lahendatav, on järgmine ülesanne veel kasulikum.
- **Maanda riske**. Ära võta esimeseks ülesannet, mille lahendust on keeruline kirjutada ja mis samal ajal kätkeb endas riski, et lahendus pole üldse õigegi. Halvimal juhul kulutad kogu võistluse aja sellele ülesandele, saad selle eest lõpuks nulli ning ei saa ka teiste ülesannete eest midagi.
- Kui sa arvad, et oskad mingit ülesannet lahendada, siis **ära jäta seda viimasele tunnile**, võttes enne ette mõne raskema ülesande, vaid lahenda pigem kohe ära. Sageli juhtub, et su esialgne idee polnud päris korrektne ja tegelikult kulub lahendusele oluliselt rohkem aega.
- Kui tegu on alamülesannetega võistlusega (nt IOI) ja tundub, et väiksema alamülesande lahendus võiks viia suurema lahenduseni, tee väiksem kiiresti ära ja esita see. Sellel on mitu eelist:
    - annab kinnituse, et oled algoritmiliselt õigel teel,
    - kui lahenduses on vigu, mille tõttu mõni test läbi ei lähe, saad sellest kohe teada,
    - nüüd on kindlalt mingid punktid käes ja pole riski üldse ilma jääda.
- Kasuta aega efektiivselt. Kuigi võistlustel on enamasti 4-5 tundi aega, kulub see üllatavalt kiirelt. Tee ka lihtsamad ülesanded ära võimalikult kiiresti, säästetud kümme minutit on tihti just see, mis jääb puudu viimase vea leidmisest võistluse lõpus.

7.5 AHNED ALGORITMID

Et peatükk liiga teoreetiline poleks, uurime mõnd praktilist ülesannet. Enamik selle raamatu peatükke käsitlevad spetsiifilisi algoritme ja ülesandeid, mille lahendamiseks vastavad algoritmid hädavajalikud on. Samas on palju ülesandeid, mida on võimalik lahendada „ahnelt“, lihtsalt ühest spetsiifilisest kohast pihta hakates.

Ahne algoritm on selline, kus igal sammul tehakse parasjagu kõige optimaalsem valik, lootuses, et see viib optimaalse lõpptulemuseni. Kui selline lähenemine töötab, siis lahendus on kiire ja lühike, seega väga efektiivne. Keerulisem osa seisneb pigem selliste ülesannete ära tundmises ja tõestamises, et ahne algoritm annab tõepoolest korrektse tulemuse. Ahne algoritmiga lahenduva probleemi ära tundmisel on abiks veel järgmine omadus: sellisel probleemil on optimaalsed alamstruktuurid. See tähendab, et optimaalne lahendus kogu ülesandele sisaldab optimaalseid lahendusi alamülesannetele. Eelmises peatükis käsitletud Dijkstra algoritm on tegelikult ahne algoritm – iga tippu vaadeldakse täpselt üks kord ja valitakse sellesse minemiseks hetkel parim tee.

Järgmiseks on toodud kolm klassikalist ahne algoritmiga lahenduvat ülesannet.

7.5.1 Mündid

Sul on vaja programmeerida müügiautomaadi jaoks raha tagastamise funktsioon. Automaat võtab vastu sularaha maksimaalselt kuni 10 eurot ja tagastab ainult münte (1-, 2-, 5-, 10-, 20- ja 50-sendiseid ja 1- ja 2- euroseid). Klientidele meeldib, kui nad saavad tagasi võimalikult vähe münte. Sinu funktsioon peab tagastama automaadi väljastatavad mündid. Võib eeldada, et münte on automaadis alati piisavalt. Sisendi ühel ja ainsal real on üks täisarv: tagastatav summa. Väljastada 8 tühikuga eraldatud täisarvu, milles esimene näitab, mitu 1-sendist automaat peab tagastama, teine mitu 2-sendist jne. Viimane arv näitab, mitu kahe eurost tuleb tagastada.

NÄIDE 1:

33

Vastus:

`0 0 0 1 1 0 1 1` (20-, 10-, 2- ja 1-sendine)

NÄIDE 2:

999

Vastus:

`4 1 1 2 0 1 2 0`

Sellest ülesandest oli juba põgusalt juttu 5. peatükis punktis 5.1.2 Ahne algoritm. Siin siis ahne lahendus, mis on nii lihtne, et vaevalt sobib siia, edasijõudnute osasse ning ei vaja seetõttu ka pikemat selgitust:

```
int main()
{
    int myndid[8] = { 200, 100, 50, 20, 10, 5, 2, 1 };
    int vastus[8] = { 0 };
    int summa;
    cin >> summa;

    for (int i = 0; i < 8; i++) {
        while (summa >= myndid[i]){
            summa -= myndid[i];
            vastus[i]++;
        }
        if (summa == 0) break;
    }

    for (int i = 0; i < 8; i++) {
        cout << vastus[i] << ' ';
    }
    cout << endl;
    return 0;
}
```

Oluline on siiski tähele panna, et müntide väärtused on kahanevas järjekorras, kuna kõigepealt tuleb väljastada kõige suuremat münti nii palju, kui antud summasse mahub, seejärel suuruselt järgmist jne. Seega, kui müntide väärtused oleks juhuslikult ette antud, tuleb need kindlasti sorteerida (või otsida iga kord suuruselt järgmist väärtust).

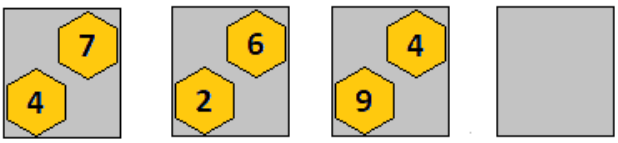
7.5.2 Kingid

Lastekodule on annetatud jõuludeks hulk mänguasju, millest tuleb lastele jõulupakid kokku panna. Igas pakis võib olla kuni kaks mänguasja ning need peavad olema komplekteeritud nii, et pakkide väärtused oleks võimalikult sarnased, st et summa iga paki väärtuse ja pakkide keskmise väärtuse vahel oleks võimalikult väike. Pakk võib olla ka tühi.
Sisendi esimesel real on kaks arvu: laste arv lastekodus `L` ja mänguasjade arv `M` (`L≤M≤2L`). Teisel real on `M` positiivset täisarvu: esimene on esimese mänguasja väärtus, teine teise oma jne. Väljastada üks arv: pakkide väärtuste keskmisest erinevuse minimaalne summa.
NÄIDE:

```
4 6
4 7 2 6 9 4
```

Vastus:
2 (pakid on asjadega väärtustega 4 ja 4 (vahe 0); 2 ja 6 (vahe 0); 9 (vahe 1)ja 7 (vahe 1))

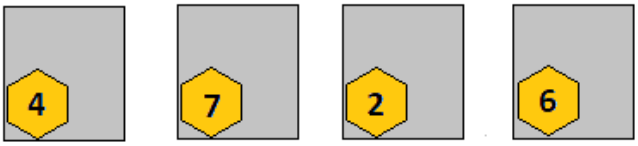
Selle ülesande puhul on veidi keerulisem läbi närida, kuidas ahne lähenemine töötada võiks. Vaatame esimest näidet nelja paki ja 6 mänguasjaga. Proovime kõigepealt lihtsalt jagada esimesse pakki 2 asja, teise 2 jne. Saame jaotuse:



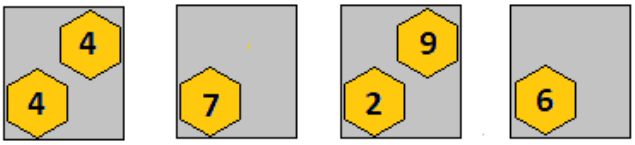
Esimese paki väärtus on 11, teise oma 8, kolmandal 13 ja viimasel 0. Kõigi pakkide keskmine väärtus on $(11 + 8 + 13 + 0) : 4 = 32 : 4 = 8$. Seega on praegu väärtuste vahede summa $|11-8| + |8 - 8| + |13 - 8| + |0 - 8| = 3 + 0 + 5 + 8 = 16$.

Kui tõsta mõnest pakist, kus on 2 asja, üks ese tühja pakki, siis on kaks võimalust: kui tõsta ese sellisest pakist, mille väärtus on kõrgem kui keskmine, siis väärtuste vahede summa paraneb (näiteks tõstes esimesest pakist eseme väärtusega 7 viimasesse pakki); kui tõsta ese ümber sellisest pakist, mille väärtus on võrdne või madalam keskmisest, siis väärtuste vahede summa jääb samaks (näiteks tõstes teisest kastist eseme väärtusega 6 viimasesse kasti). Veendu selles ise!

Kokkuvõtvalt: mõne kasti tühjaks jätmisega pakkide väärtuste vahede summat ei paranda, pigem vastupidi. Proovime siis jaotada kõigepealt igasse kasti täpselt ühe kingituse:

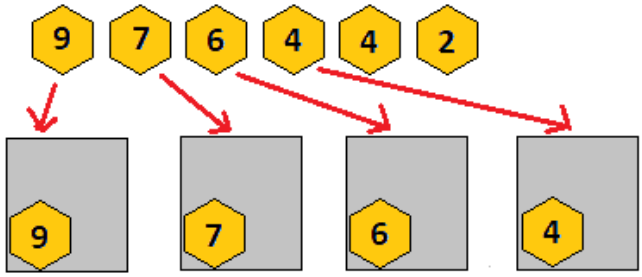


Vaja on veel lisada kingitused väärtustega 9 ja 4. Parima jaotuse saamiseks tuleks lisada kõige suurema väärtusega kingitus kõige väiksema väärtusega pakki, saame jaotuse:

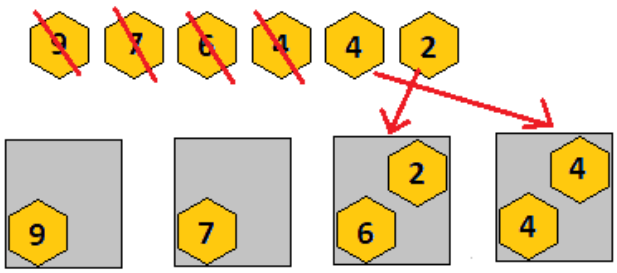


Siin on pakkide väärtuste keskmisest erinevuste summa $|8-8| + |7 - 8| + |11 - 8| + |6 - 8| = 6$. Aga seegi ei ole veel vähim võimalik.

Ma arvan, et nii mõnigi lugeja on juba taibanud, et esimeste asjade pakkidesse jaotus võiks juba olla ühtlasem, st esimesena tuleks panna kõige suurema väärtustega esemed ning seejärel lisada ülejäänud esemed, nagu kirjeldatud. Tõesti, kui sorteerime mänguasjad väärtuse järgi, läheb pilt kohe selgemaks:



Ning viimased tuleb jaotada täpselt vastupidises järjekorras, st alustades viimasest pakist:



Nii on vastuseks $|9-8|+|7-8|+|8-8|+|8-8|=2$. See ongi parim jaotus.

Programmis on sageli lihtsam oletada, et igasse pakki läheb täpselt 2 kingitust. Selleks lisame nii palju 0-väärtusega asju, et mänguasjade arv oleks täpselt kahekordne pakkide arv. Nii saame lihtsa vaevaga pakid kohe komplekteerida, pannes esimesse pakki kõige suurema ja väiksema väärtusega eseme, teise teise kõige suurema ja teise kõige väiksema eseme jne, kuni viimasesse pakki jääb kaks keskmise väärtusega eset. Siin on ülesande lahendus:

```
int main()
{
    int L, M, summa = 0;
    cin >> L >> M;

    int* vaartused = new int[2*L];
    int i = 0;
    for (i; i < M; i++) {
        summa += vaartused[i];
    }

    for (i; i < 2*L; i++) {
        vaartused[i] = 0;
    }

    sort(vaartused, vaartused + 2*L);
    double keskmine = double(summa / L);
    double vahe = 0;
    for (i = 0; i < L; i++){
        vahe += abs(keskmine - vaartused[i] - vaartused[2 * L - i - 1]);
    }

    cout << vahe;
    return 0;
}
```

Seda tüüpi ülesandeid nimetatakse **tasakaalustamise** (*load balancing*) ülesanneteks. Järgnevaks veel üks ahne algoritmi tüüpülesanne: **lõikude katmine** (*interval covering*).

7.5.3 Majakavahid

Kaugel-kaugel merel asub üksik laid. Laiul on majakas ja seal peab olema kohal majakavaht. Iga aasta detsembris annavad potentsiaalsed majakavahid teada perioodid, millal nemad saaksid majakas olla. Kuna inimeste viimine laiule ja sealt tagasi on kallis, siis soovitakse, et vahetusi oleks võimalikult vähe. Kirjuta programm, mis leiab minimaalse vahetuste arvu. Esimene vahetus alustab alati 1. jaanuaril ja viimane lõpetab 31. detsembril. Aastas on alati 365 päeva. Kui üks vaht saab kohal olla kuni 3. aprill ja teine saab alustada 4. aprillil, siis saab need vahid järjest vahetada, ilma, et majakas valveta jääks.
Sisendi esimesel real on üks arv: majakavahtide arv kokku. Järgnevalt on rida majakavahi numbriga ja tema vahetuste arvuga V. Järgneval V real on vahetuse algusaeg ja lõppaeg kujul pp.kk, eraldatud tühikuga. Seejärel on järgneva majakavahi number ja tema vahetuste kuupäevad, jne.
Väljastada üks number: vahetuste arv. Kui jääb katmata päevi, kus ükski vaht majakas olla ei saa, väljastada „EI SAA“.

NÄIDE 1:

```
4
1 3
04.04 08.07
09.09 10.10
11.11 31.12
2 3
01.01 20.02
03.03 05.05
11.10 10.11
3 2
01.01 03.03
01.07 01.10
4 2
01.08 10.10
12.11 31.12
```

Vastus:

```
7
```

(1.1-3.3, 3.3-5.5, 4.4-8.7, 1.7-1.10, 1.8-10.10, 11.10-10.11, 11.11-31.12)

NÄIDE 2:

```
2
1 1
01.01 01.06
2 1
01.06 30.12
```

Vastus:

```
EI SAA
```

Paldiski tuletorn

Kindel on see, et esimese vahetuse majakavaht peab alustama 1. jaanuaril. Kui sel päeval saab alustada vaid üks majakavaht, tuleb tema esimesena laiule toimetada. Kui aga see päev sobib mitmele vahile, siis milline neist valida? Kuna meid huvitab ainult minimaalne vahetuste arv, siis valime ahnelt selle, kelle vahetus lõpeb kõige hiljem. Ülejäänud vahetused, mis saaksid alustada 1. jaanuaril, võime täiesti kõrvale heita, kuna meie valik katab need täielikult. Mis aga valida järgmiseks? Teame, et algusperiood peab kattuma või vahetult järgnema esimese vahetuse lõpule. Taas on hea valida sobiva algusega vahetuste

seast see, mis lõppeb kõige hiljem. Nüüd taas võib kõik sobiva algusperioodiga vahetused, mille seast enne valisime, kõrvale jätta. Ja nii edasi, kuni aasta lõpp on käes. Kui mingil etapil sobiva algusajaga vahetust polegi, siis kattumata vahetusi tekitada ei saa.

```
// antud kuule eelnev päevade arv. Jaanuarile eelneb 0 päeva, veebruarile 31, märtsile //
31+28 jne
const int kuud[12] = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 };

int leiaPaevaNr(string kuupaev) {
    int k = kuupaev.find('.');
    int kuu, paev;
    paev = atoi(kuupaev.substr(0, k).c_str());
    kuu = atoi(kuupaev.substr(k+1, kuupaev.length()).c_str());
    return kuud[kuu-1] + paev;
}

int main()
{
    int M, vahetusi = 0;
    vector<pair<int, int>> vahetused;
    cin >> M;
    while (M--){
        int N, V;
        cin >> N >> V;
        while (V--){
            string algus, lopp;
            cin >> algus >> lopp;
            vahetused.push_back(make_pair(leiaPaevaNr(algus), leiaPaevaNr(lopp)));
        }
    }

    sort(vahetused.begin(), vahetused.end()); // sorteerime alguse järgi kasvavalt
    int esimenePaev = 1, viimanePaev = 0, vahetusiKokku = 0;
    int i = 0;

    while (viimanePaev < 365) { // Vahetused on leitud
        // ei ole jõudnud aasta lõppu, aga rohkem vahetusi pole või
        // järgmise vaatamata vahetuse esimene päev on liiga kaugel
        if (i == vahetused.size() || vahetused[i].first > esimenePaev) {
            cout << "EI SAA" << endl;
            return 0;
        }
        for (i; i < vahetused.size() && vahetused[i].first <= esimenePaev; i++) {
            if (vahetused[i].second > viimanePaev) { // parem viimane päev
                viimanePaev = vahetused[i].second;
            }
        }
        vahetusiKokku++; // oleme leidnud parima lõpp-päeva
        esimenePaev = viimanePaev + 1;
    }

    cout << vahetusiKokku << endl;
    return 0;
}
```

7.5.4 Veel ahnetest algoritmidest

Nagu juba mainitud, siis ahne lähenemise juures kõige raskem on tõestada, et see viib tõesti korrektse tulemuseni. Need kolm klassikalist näidet tasub siiski meelde jätta ja harjutada nende ära tundmist.

Ahned algoritmid üldiselt eeldavad, et andmed on sorteeritud. Mündiülesandes olid mündid antud juba sorteeritud järjekorras, kinkide ja majakavahtide ülesannetes tuli andmed ise sorteerida. Sageli aitab andmete eelnev sorteerimine kaasa sobiva ahne algoritmi leidmisele. Seega võistlusel, kui hea lahendus kohe pähe ei tule, on kasulik andmed sorteerida ning vaadata, kas sellest võiks mingit abi olla.

Kuna need lahendused on ülilihtsad programmeerida ja töötavad imekiiresti, siis võistlustel selliseid klassikalisi ahneid ülesandeid sageli ei kohta. Tundmatu ülesande lahendamine ahne meetodiga on küllaltki riskantne, tõestamine, et ahne algoritm korrektse tulemuse annab, on ajamahukas. Seega, kui ülesande andmete maht on selline, et variantide läbivaatus või dünaamiline planeerimine leiab vastuse etteantud ajalimiiti jäädes, on need lähenemised kindlamad, kuna leiavad kindlasti korrektse vastuse. Teisalt aga võivad kavalamad ülesande koostajad panna meelega madalamad piirid andmetele, et lahendajad kohe ahne algoritmi kasutamise võimalusele ei mõtleks.

Ahned on ka mitmed kavalad nimelised algoritmid. Eespool oli juba juttu, et Dijkstra algoritm lühima tee leidmiseks graafis on oma olemuselt ahne. Üheksandas peatükis tutvustatakse ka Kruskali ja Primi algoritme minimaalse toesepuu leidmiseks, mis on samuti ahned. Tuntud Huffmani pakkimisalgoritm kasutab samuti ahnet lähenemist.

Järgmiseks aga üks keerukam ülesanne Rahvusvaheliselt informaatikaolümpiaadilt aastast 2015 (http://ioi2015.kz/).

7.5.5 Suveniirid

Käimas on IOI 2015 avatseremoonia lõpuakt. Tseremoonia käigus pidi iga võistkond saama endale suveniiri. Kahjuks olid aga kõik korraldajad tseremoonia poolt nii võlutud, et unustasid need välja jagada. Ainus, kes suveniiridest midagi mäletab, on Aman. Tema on entusiastlik korraldaja ning tahab, et IOI-l oleks kõik ideaalne, seega üritab ta jagada kõik suveniirid minimaalse ajaga.
Avatseremoonia toimumiskohaks on ringikujuline hoone, mis on jagatud `L` sektsiooniks. Sektsioonid on nummerdatud järjest `0` kuni `L-1`-ni. Seega, iga `0≤i≤L-2` puhul on sektsioonid `i` ja `i+1` naabrid ning sektsioonid `L-1` ja `0` omavahel naabrid. Kokku on `N` võistkonda. Iga võistkond istub ühes sektsioonidest. Igas sektsioonis võib asuda suvaline arv võistkondi. Mõned sektsioonid võivad olla ka tühjad.
Jaotamiseks on `N` identset suveniiri. Algselt asub nii Aman kui ka kõik suveniirid sektsioonis `0`. Amanil tuleb igale võistkonnale kätte toimetada üks suveniir ning pärast viimase suveniiri äraandmist peab ta tagasi jõudma sektsiooni `0`. Pange tähele, et mõned võistkonnad võivad istuda ka sektsioonis `0`.
Igal hetkel saab Aman kaasas kanda maksimaalselt `K` suveniiri. Aman võtab suveniirid kaasa sektsioonis `0`, ning selleks ei kulu tal aega. Igat suveniiri tuleb kaasas kanda, kuni ta on mõnele võistkonnale kättetoimetatud. Kui Amanil on käes vähemalt üks suveniir ning ta jõuab mõne sektsioonini, kus istub võistkond, kes pole veel suveniiri saanud, võib ta sellele võistkonnale ühe kaasaskantud suveniiridest üle anda. Üleandmine juhtub ka momentaalselt ning pole keelatud ühes sektsioonis mitmele võistkonnale korraga suveniire jagada. Ainus asi, mille peale aeg kulub, on liikumine. Aman võib liikuda mööda ringikujulist hoonet mõlemas suunas. Liikumine kõrvalasuvasse sektsiooni (nii päri- kui ka vastupäeva suunas) võtab täpselt ühe sekundi, sõltumata sellest, kui palju suveniire on tal parajasti kaasas.

Teie ülesanne on leida minimaalne sekundite arv, mille jooksul Aman jõuab ära jagada kõik suveniirid ning naasta tagasi algsesse positsiooni.
Sisendi esimesel real on kolm arvu `N` ($1 \le N \le 10^7$), `K` ($1 \le K \le N$) ja `L` ($1 \le L \le 10^9$). Teisel real on `N` arvu: esimesena esimese meeskonna sektor, teisena teise meeskonna oma jne.

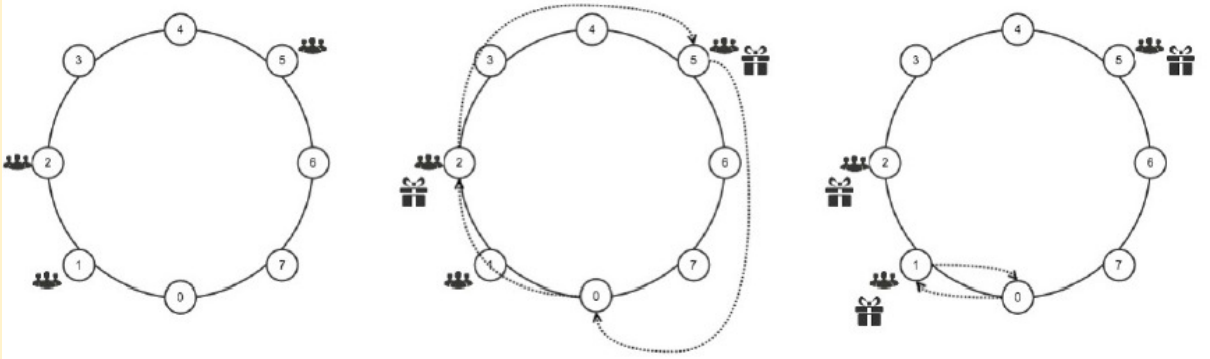
NÄIDE:

```
3 2 8
1 2 5
```

Vastus:

```
10
```

Üks võimalik optimaalne lahendus (näidatud pildil) on järgmine. Alguses võtab Aman kaasa kaks suveniiri, toimetab ühe sektsioonis 2 asuvale võistkonnale, teise sektsioonis 5 asuvale võistkonnale ning tuleb tagasi sektsiooni 0. See käik võtab tal 8 sekundit. Teisel käigul toob Aman alles jäänud suveniiri sektsioonis 1 istuvale võistkonnale ning tuleb tagasi sektsiooni 0. Tal kulub selle peale veel 2 sekundit. Koguaeg on seega 10 sekundit

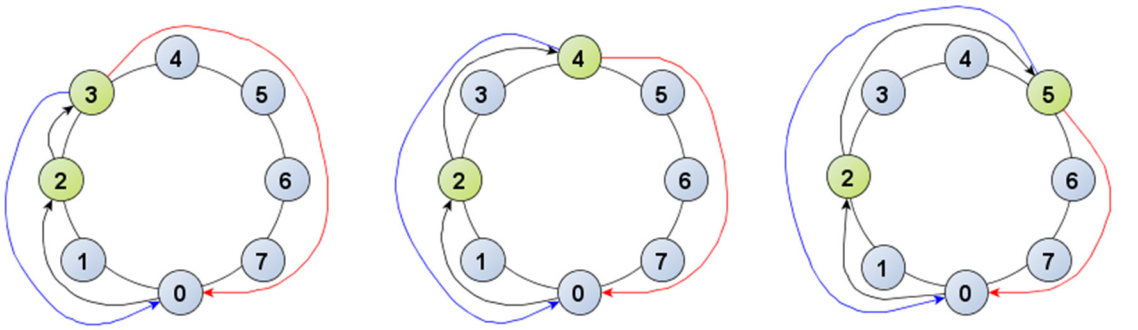


0-sektorist on võimalik suveniire viia teistesse sektoritesse liikudes kas päripäeva või vastupäeva. Ühe sületäie suveniiride jaotamiseks on kolm võimalust:

1. Viime suveniirid päripäeva liikudes ja tuleme tagasi vastupäeva
2. Viime suveniirid vastupäeva liikudes ja tuleme tagasi päripäeva
3. Viime suveniirid, tehes terve ringi. Tee pikkuse mõttes ei ole vahet, kas ring teha päri- või vastupäeva.

Ilmne on, et ühel käigul ei ole mõttekas teha rohkem kui üks täisring. Edasi-tagasi liikudes on mõtet minna maksimaalselt sektsioonini, kus suveniiri jagamine toimub. Edasi-tagasi liikudes ei ole tegelikult oluline, kas suveniiride jagamine soovitud sektsioonides toimub minnes või tulles, seega võib eeldada, et jagamine toimub alati minnes, samuti nagu ringiratast jagades.

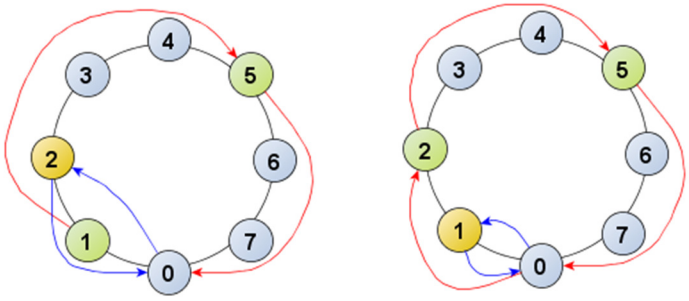
Oluline on tähele panna ka, et optimaalse jagamise korral ühe sületäie jagamisele ei saa kuluda rohkem kui L sekundit. Kui liigume ühe ringi, siis kulub selleks täpselt L sekundit. St kui me liigume päri- või vastupäeva jagades kaugemale kui poole ringi peale, on optimaalsem tagasi minemise asemel teha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



Vasakpoolsel joonisel viiakse kaks pakki sektsioonidesse 2 ja 3. Seejärel on tagasi 0-sektsiooni liikumiseks 2 võimalust: päripäeva (punane nool) ja vastupäeva (sinine nool). Kasulikum on tagasi liikuda vastupäeva, nii kulub 2 paki jagamiseks 6 sekundit, päripäeva kuluks 8 sekundit. Keskmisel joonisel viiakse kaks pakki sektsioonidesse 2 ja 4 ning seejärel liigutakse tagasi 0-sektsiooni kas päri- või vastupäeva. Mõlemad teekonnad on sama pikad ja võtavad 8 sekundit. Parempoolsel joonisel viiakse pakid sektoritesse 2 ja 5 ning liigutakse samuti edasi kas päri – või vastupäeva. Sellel juhul kulub päripäeva jätkates vähem aega: 8 sekundit (vs 10 sekundit).

Järgmiseks tähelepanekuks on, et optimaalsel lahendusel saame jagada suveniirid järjestikustes sektsioonides olevatele meeskondadele. Näites on mõttekas jagada ühel käigul suveniirid meeskondadele sektsioonides 1 ja 2 või 2 ja 5, aga mitte 1 ja 8 (liikudes teisest sektsioonist jagamata üle). Tõesti, ringiratast liikudes, kui jätame mõne sektsiooni vahele, siis saame selle vahetada sektsiooni vastu, mis on sektsioonile 0 lähemal. Ringiratast jagamise ajakulu see ei muuda, kuid 0-sektsioonile lähemal olevasse sektorisse on kiirem kui edasi-tagasi viimine.

Samuti edasi-tagasi käikudel tasub kogu sületäis viia korraga järjestikustesse sektsioonidesse, sest muidu saame samamoodi vahetada mõne kaugemal asuva meeskonna ja lähemal asuva meeskonna suveniiride jaotuse ja seega potentsiaalselt ainult vähendada jagamisele kuluvat aega.



Vasakpoolsel joonisel viiakse esmalt pakid sektsioonidesse 1 ja 5 (punase noolega näidatud teed) ja seejärel viimane pakk sektorisse 2 (sinised nooled). Kokku kulub 8 + 4 = 12 sekundit. Parempoolsel joonisel viiakse pakid esmalt sektsioonidesse 2 ja 5 ning seejärel viimane pakk sektsiooni 1. Kokku kulub 8 + 2 = 10 sekundit.

Veelgi huvitavam: meil ei ole optimaalse lahenduse korral vaja teha rohkem kui üht täisringi ning täisringil tuleks jagada võimalikult palju suveniire, st K suveniiri või kõik suveniirid, kui K ≥ N.

Teine pool sellest väitest on lihtsasti mõistetav. Kuna täisringi tegemine võtab L minutit sõltumata sellest, kui palju suveniire sellel jagatakse, siis on alati võimalik mõni edasi-tagasi käigul jagatav suveniir jagada hoopis ringiratast liikudes, ilma, et kogu ajakulu sellest suureneks.

Esimene osa on ehk veidi keerulisem taibata, aga kuna kindlasti ei ole aeglasem üht sületäit jagada järjestikuste mehitatud sektsioonide vahel, järjestikused sektsioonid kahel ringi poolel saavad aga olla ainult ühes lõigus. Muudel juhtudel me aga täisringi tehes ei võida ajas võrreldes edasi-tagasi käimisega.

```
long long delivery(int n, int k, int l, int p[])
{
    ll* ppAeg = new ll[n]; // meeskonnani paki viimise aeg ainult päripäeva liikudes
    ll* vpAeg = new ll[n]; // meeskonnani paki viimise aeg ainult vastupäeva liikudes
    // leiame ajad iga meeskonnani ainult päripäeva ja ainult vastupäeva liikudes,
    //arvestades, et eelnevad meeskonnad on suveniirid kätte saanud.
        for (int i = 0; i < n; i++) {
        if (i < k) { // esimesed k meeskonda mõlemast otsast
            ppAeg[i] = p[i] * 2; // käime päripäeva i-nda meeskonnani ja tagasi
            vpAeg[n-1-i] = (l - p[n-1-i]) * 2; // sama vastupäeva
        }
        else {
            int j = i-k; // meeskonna indeks, mis jääb kindlasti eelmisesse jaotusesse
            ppAeg[i] = p[i] * 2 + ppAeg[j];// eelmised jaotused + praeguse meeskonnani
            vpAeg[n-1-i] = (l - p[n-1-i]) * 2 + vpAeg[n-1-j]; //sama vastupäeva
        }
    }
    ll vastus = 1e18; // vastuseks suur arv, otsime miinimumi

    if (n == k) { // suveniirid saab korraga kaasa võtta ja ühe ringiga jaotada.
        vastus = l; // sel moel on üheks võimalikuks vastuseks sektorite arv.
    }

    // parim saab olla variant, kus jaotame kõik vastupäeva
    vastus = min(vastus, vpAeg[0]);
    // või üks täisring ja kõik ülejäänud vastupäeva
    vastus = min(vastus, vpAeg[k] + l);
    for (int i = 0; i < n; i++) {
        if (i == n - 1) {
            vastus = min(vastus, ppAeg[i]); // Kõik meeskonnad päripäeva jaotades
            continue;
        }
        // ainult edasi-tagasi: i. meeskonnani päripäeva ja i+1. alates vastupäeva.
        vastus = min(vastus, ppAeg[i] + vpAeg[i+1]);
        if (i + k + 1 == n) { // kui on ainult k meeskonda lõpuni
            vastus = min(vastus, ppAeg[i] + l); // üks täisring + ülejäänud päripäeva
        }
        else if (i + k + 1 < n) {
            // päripäeva i-nda meeskonnani, siis täisring (k pakki/meeskonda) ja siis
            // edasi alates meeskonnast i+k+1 vastupäeva.
            vastus = min(vastus, ppAeg[i] + vpAeg[i+k+1] + l);
        }
    }
    return vastus;
}
```

7.6 KONTROLLÜLESANDED

Nagu ka esimese osa esimeses peatükis, on käesoleva peatüki ülesanded suhteliselt lihtsad. Muretsemiseks pole põhjust, edasi tuleb ka raskemaid ülesandeid.

7.6.1 Onu Robert

Piilupart Donaldi lugudest tuntud Onu Robert külastab riiki, kus on kasutusel `N` erineva väärtusega rahatähte (`1≤N≤1000`). Kohale jõudes läheb ta kõigepealt panka, et kohalike kulutuste jaoks raha välja võtta. Pank väljastab raha „ahnelt“:
1. anda välja suurim kupüür, mis on väljastada jäänud summast väiksem,
2. korrata, kuni maksta jääv summa on null.

Väikseima kupüüri väärtus on alati 1, nii et väljastamine on võimalik. Kuna Onu Robert armastab erinevaid kupüüre, soovib ta saada võimalikult mitu erineva väärtusega rahatähte. On teada, et ta on põhjatult rikas ja saab küsida mistahes summa. Leida, mitme erineva väärtusega kupüüre on tal võimalik ühe väljavõtmisega saada.

Sisendi esimesel real on arv `N`. Teisel real on `N` täisarvu rangelt kasvavas järjestuses, mis tähistavad kupüüride väärtusi. Väljastada üks täisarv: erineva väärtusega rahatähtede arv, mida on võimalik pangast korraga välja võtta.

NÄIDE 1:

```
6
1 2 4 8 16 32
```

Vastus:

```
6
```

NÄIDE 2:

```
6
1 3 6 8 15 20
```

Vastus:

```
4
```

7.6.2 Bussijuhid

Ühes linnas töötab N bussijuhti (1≤N≤100). Samuti on seal linnas täpselt N hommikust ja N õhtust bussimarsruuti. Iga juht peab läbi sõitma ühe hommikuse ja ühe õhtuse marsruudi. Kui juhi poolt sõidetud tee pikkus on suurem kui etteantud arv D (1≤D≤10000), peab talle maksma lisatasu R tugrikut iga kilomeetri eest (1≤R≤5).
Ülesandeks on paigutada bussijuhid marsruutidele nii, et makstav lisatasu oleks minimaalne.
Sisendi esimesel real on arvud N, D ja R. Teisel real on N tühikutega eraldatud positiivset täisarvu, mis tähistavad hommikuste marsruutide pikkusi ning kolmandal real samuti N täisarvu, mis vastavad õhtustele marsruutidele.
Väljundisse kirjutada minimaalne võimalik lisatasu, mida bussijuhtidele tuleb maksta.

NÄIDE 1:

```
2 20 5
10 15
10 15
```

Vastus:

```
50
```

NÄIDE 2:

```
2 20 5
10 10
10 10
```

Vastus:

```
0
```

1Dalla-Dalla nimeline buss, millega Targo Tansaanias sõitis

7.6.3 Hernehirmutised

Farmer Fredil on pikk peenar, mida ohustavad varesed. Peenar on jagatav N lõiguks (1≤N≤100), millest osadele on midagi külvatud, osadele mitte. Peenardele saab panna ka hernehirmutisi, mis kaitsevad vareste eest seda lõiku, millel nad ise seisavad, ning lisaks ühte naaberlõiku kummalgi pool.
Leida, mitut hernehirmutist on vaja, et kaitsta ära terve peenar.
Sisendi esimesel real on arv N. Teisel real on string pikkusega N, mis koosneb märkidest '.' (tähistab külvatud lõiku) ja '#' (tähistab külvamata lõiku).
Väljundisse kirjutada täpselt üks arv: mitut hernehirmutist on vaja.

NÄIDE 1:

```
3
.#.
```

Vastus:

```
1
```

NÄIDE 2

```
11
...##....##
```

Vastus:

```
3
```

7.6.4 Kolimine

Artur kolis just uude majja ja nüüd on tal üle N kuubikujulist pappkasti (1≤N≤10 000). Ruumi kokku hoidmiseks katsub Artur panna need kastid üksteise sisse. Selleks, et üht kasti teise sisse panna, peab see olema teisest rangelt väiksem. Leida strateegia, mis muudaks nähtavate kastide arvu võimalikult väikeseks.
Sisendi esimesel real on kastide arv N ja teisel real N positiivset täisarvu, mis tähistavad kastide suurusi.
Väljundi ainsale reale kirjutada minimaalne võimalik näha jäävate kastide arv M.

NÄIDE:

6
1 1 2 2 2 3
Vastus:
3

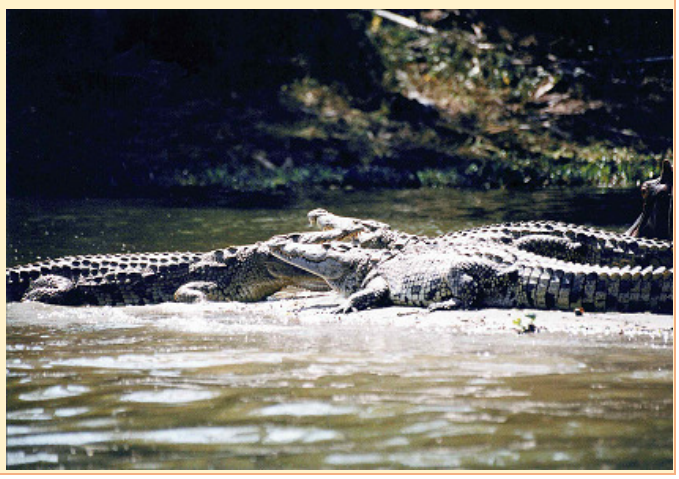
Selle vastuse saamiseks on kastide paigutamiseks mitu võimalust. Näiteks sobib nii paigutus
1→2, 1→2, 2→3 kui ka 1→2→3, 1→2, 2 (kus a→b tähistab, et kast suurusega a on kasti suurusega b sees)

7.6.5 Krokodillid

Indiana Jones seisab jõe vasakul kaldal ja tal tuleb teiselt kaldalt ära tuua kullast iidolikuju. Jões on rida kive ja krokodille, mille otsa Jones saab jõe ületamisel hüpata. Krokodilli peale hüppamisel ujub ta minema ja tagasitulekul ei saa enam seda krokodilli kasutada. Kivid jäävad alati paigale. Kuna pikad hüpped on ohtlikumad, katsub Indiana Jones jõge ületada nii, et tema pikim hüpe jääks minimaalseks. Leida, mis on pikim hüpe, mille ta siiski peab tegema.
Sisendi esimesel real on kivide arv N (1≤N≤100), krokodillide arv M (1≤M≤100)ja jõe laius D ($1 \le D \le 10^9$). Teisel real on N täisarvu, mis tähistavad kivide kaugust jõe vasakust kaldast ning kolmandal real M täisarvu, mis tähistavad krokodillide kaugusi.
Väljundisse kirjutada üks arv, mis tähistab Jonesi pikima vajaliku hüppe pikkust.

NÄIDE 1:

1 1 10
5
7
Vastus:
5

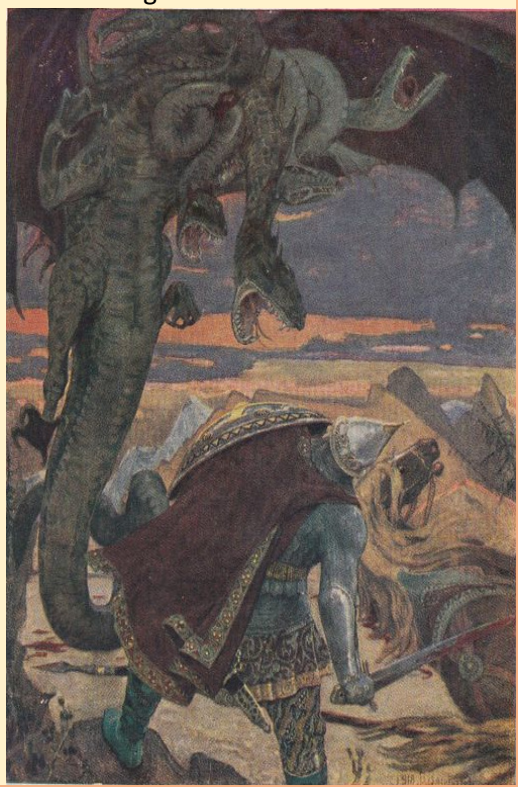


NÄIDE 2:

1 1 10
3
6
Vastus:
7

7.6.6 Lohe Gorõnitš

Vanal Kiievi-Venemaal oli suureks nuhtluseks lohe Gorõnitš. Bõliinadest tuntud vägilane Dobrõnja Nikititš palkab endale parajasti abiks teisi kangelasi, et lohe vastu võitlusse minna. Lohel on hulk erineva suurusega päid ja tema tapmiseks on vaja kõik need pead maha raiuda. Iga kangelane jaksab raiuda ülimalt ühe pea, aga ainult sellise, mis ei ole temast endast suurem. Samas tuleb kangelastele maksta tasu vastavalt nende enda suurusele. Leida minimaalne tasu, mis tuleb kangelastele lohe võitmise eest maksta.



Sisendi esimesel real on kaks täisarvu: lohe peade arv `N` ja riigis elavate kangelaste arv `M` (`1≤M, N≤20 000`). Teisel real on `N` täisarvu, mis tähistavad lohe peade suurusi ja kolmandal real `M` täisarvu, mis kangelaste suurused. Väljundisse kirjutada minimaalne makstav palk. Kui kangelastel pole võimalik lohet võita, sööb lohe nad kõik ära ja palka ei maksta. Sel juhul tuleb väljundisse kirjutada `0`.

NÄIDE 1:

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

Vastus:

```
11
```

(palkame esimese ja kolmanda kangelase)

NÄIDE 2:

```
2 1
5 5
10
```

Vastus:

```
0
```

7.6.7 Fraktsioonid

N-liikmeline parlament jaguneb fraktsioonideks, mis peavad olema erineva suurusega. Iga fraktsioon saadab kord päevas ühe liikme fraktsioonidevahelisele nõupidamisele. Et saaks arutada erinevaid asju, peab nõupidamisel iga kord olema erinev koosseis, vastasel korral arvab rahvas, et parlament ei tee midagi kasulikku ning korraldatakse uued valimised. Kuna parlamendi liikmed ei soovi oma kohast loobuda, katsuvad nad jaotada end fraktsioonideks nii, et nõupidamisi saaks korraldada võimalikult kaua. Kui antud on parlamendi suurus, siis leida, milline fraktsioonideks jaotus võimaldaks parlamendil võimalikult kaua püsida.

Sisendi ainsal real on täpselt üks arv N ($5 \le N \le 1000$) – parlamendi liikmete arv.

Väljundi esimesele reale kirjutada fraktsioonide arv M ja teisele reale kasvavas järjestuses M tühikutega eraldatud täisarvu, mis tähistavad fraktsioonide suurusi.

NÄIDE 1:

7

Vastus:

2

3 4 (parlament töötab 3*4=12 päeva)

NÄIDE 2:

31

Vastus:

6

2 3 5 6 7 8

(parlament töötab 10080 päeva)

Kreeka parlament

7.6.8 Bitimask

Raamatu esimeses osas oli muuhulgas juttu bitioperatsioonidest ja bitimaskidest. Bitimaskid võimaldavad kergesti manipuleerida mõne arvu üksikuid bitte. Kui meil on näiteks vaja 32-bitise arvu alumised neli bitti ära nullida, tuleb sooritada bitikaupa AND operatsioon bitimaskiga 0xFFFFFFF0 (kõik bitid peale alumise nelja on ühed). Siin ülesandes tuleb leida bitimask M, mis:

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

Sisendi ainsal real on kolm 32-bitist täisarvu N, L ja U, kus $L \le U$.

Väljundisse kirjutada arv M. Kui võimalikke vastuseid on mitu, väljastada neist väikseim.

NÄIDE 1:

100 50 60

Vastus:

59 (59 | 100 = 127)

NÄIDE 2:

100 0 100

Vastus:

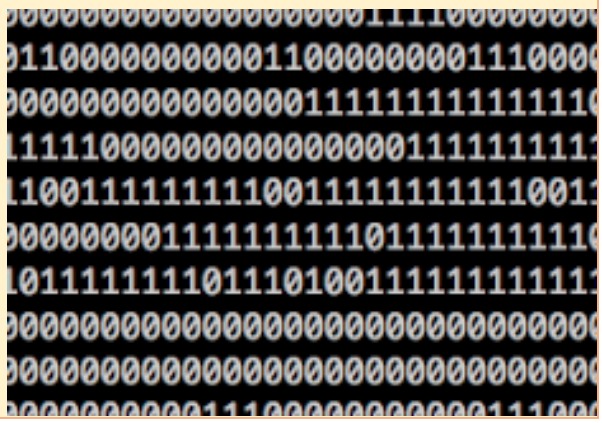
27 (27 | 100 = 127)

NÄIDE 3:

1 0 100

Vastus:

100



7.6.9 LED-lambid

Hullul elektroonikul Romanil on ristkülikukujuline paneel, mis on täidetud LED-lampidega. Alguses on kõik lambid välja lülitatud. Paneeli juhtimiseks on Roman koostanud mikroskeemi, mis võimaldab etteantud suurusega alamristkülikul lampe ümber lülitada – need, mis olid enne väljas, on nüüd sees ja vastupidi.
Ülesande sisendis on toodud pilt, mis on vaja lõpuks saavutada, leida, mitu lülitamist on selleks vaja. Sisendi esimesel real on neli täisarvu: paneeli kõrgus `N` ja laius `M` ning ümberlülitatava piirkonna kõrgus `A` ja laius `B` (`1≤A≤N≤100, 1≤B≤M≤100`). Järgmisel `N` real on igaühel `M` märgist koosnev string, mis näitab, millised lambis on soovitud pildil sisse lülitatud – `0` tähistab väljalülistatud ja `1` sisselülitatud lampi. Väljundisse kirjutada vajalike lülituste arv. Kui nõutud lülitamine pole võimalik, väljastada `-1`.

NÄIDE 1:

```
3 3 1 1
010
101
010
```

Vastus:

```
4
```

NÄIDE 2:

```
4 3 2 1
011
110
011
110
```

Vastus:

```
6
```

NÄIDE 3:

```
3 4 2 2
0110
0111
0000
```

Vastus:

```
-1
```

7.6.10 Antimonotoonne jada

Käesoleva raamatu viiendas peatükis uuriti muuhulgas etteantud arvude seast pikima kasvava või kahaneva osajada (nn monotoonse osajada) leidmist. Siin ülesandes tuleb leida jada `A` pikim antimonotoonne osajada `B`, s.t selline jada, kus `B[0] > B[1] < B[2] > B[3]`…
Osajadas peavad elemendid esinema samas järjekorras, mis algses jadas, kuid need ei pea olema algses jadas järjest. Näitekst jada 1 3 5 on jada 1 2 3 4 5 osajada.
Sisendi esimesel real on jada `A` pikkus `N` (`1≤N≤30 000`). Teisel real on `N` positiivset täisarvu.
Väljundisse kirjutada pikima võimaliku antimonotoonse osajada pikkus.

NÄIDE 1:

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

Vastus:

```
1
```

NÄIDE 2:

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

Vastus:

```
5
```

NÄIDE 3:

```
7
3 5 3 2 4 4 3
```

Vastus:

```
4 (Nt jada 5 3 4 3)
```

7.7 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk7 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
MinenurkaPikk.cpp	Tankiülesande funktsioon mineNurka esialgne variant
MinenurkaLyhem.cpp	Nurka minemise meetodid ümber kirjutatuna
Mortimer2.cpp, Mortimer2.java, Mortimer2.py	Ülesande “Miljonär ja vaeslapsed” jõumeetodiga lahendus (testimiseks)
MortimerGen.cpp, MortimerGen.java, MortimerGen.py	Testigeneraator ülesande “Miljonär ja vaeslapsed” testide koostamiseks
MortimerVal.cpp, MortimerVal.java, MortimerVal.py	Validaator ujukomaarvude võrdlemiseks testimisel
Myndid.cpp, Myndid.java, Myndid.py	Mündiülesanne ahne algoritmiga
Kingid.cpp, Kingid.java, Kingid.py	Jaotamise ülesanne ahne algoritmiga
Majakavahid.cpp, Majakavahid.java, Majakavahid.py	Vahemike kattuvuse ülesanne ahne algoritmiga
Suveniirid.cpp, Suveniirid.java, Suveniirid.py	Ahne algoritmiga lahenduv keerukam ülesanne