Kisakoodarin käsikirja

Antti Laaksonen

10. marraskuuta 2014

Sisältö

Johdanto

Ohjelmointikisojen historia ulottuu vuosikymmenten taakse. Tunnetuimmat perinteikkäät kisat ovat lukiolaisten IOI (*International Olympiad in Informatics*) sekä yliopistojen ICPC (*International Collegiate Programming Contest*). Näiden lisäksi nettiin on ilmestynyt viime vuosina monia kaikille avoimia kisoja, joiden ansiosta kisakoodaus on suositumpaa kuin koskaan ennen.

Ohjelmointikisat mittaavat kykyä ratkaista algoritmisia ohjelmointitehtäviä. Sekä teorian että käytännön taidot ovat tarpeen kisoissa, koska tehtävän idean keksimisen jälkeen ratkaisu täytyy myös pystyä koodaamaan toimivasti. Ohjelmointikisoihin osallistuminen onkin erinomainen tapa kehittää omaa ohjelmointitaitoa sekä teorian että käytännön kannalta.

Kisakoodarin käsikirja on perusteellinen johdatus ohjelmointikisojen aiheisiin. Kirjan osiot I, II ja III kattavat yleisimmät ohjelmointikisojen aiheet, ja osiossa IV on vaikeampia ja harvemmin tarvittavia tekniikoita. Kirja olettaa, että lukija hallitsee entuudestaan ohjelmoinnin perusasiat C++-kielellä (muuttuja, ehto, silmukka, taulukko/vektori, funktio).

Jos haluat todella kehittää ohjelmointitaitoasi, on tärkeää, että osallistut säännöllisesti ohjelmointikisoihin. Tällä hetkellä netin aktiivisin kisasivusto on venäläinen Codeforces (`http://www.codeforces.com/`), joka järjestää viikoittain korkeatasoisia ohjelmointikisoja. Sivuston kautta saat tiedon myös kaikista muista merkittävistä kisoista.

Kirja on vielä keskeneräinen ja jatkuvan kehityksen alaisena. Voit lähettää palautetta kirjasta osoitteeseen `ahslaaks@cs.helsinki.fi`.

Osa I

Perusasiat

Luku 1

Kisakoodaus

Kisakoodauksen erityisluonteen vuoksi osa ohjelmoinnissa yleisesti käytetyistä periaatteista sopii siihen huonosti. Kisoissa tarvittavat ohjelmat ovat lyhyitä, niitä ei tarvitse jatkokehittää eikä muiden tarvitse ymmärtää koodia. Lisäksi kisoissa on yleensä hyvin vähän aikaa ohjelman tekemiseen.

Oleellista kisoissa on, että koodi on toimiva ja tehokas ja sen saa kirjoitettua nopeasti. Hyvä kisakoodi on suoraviivaista ja tiivistä. Ohjelmointikielen valmiskirjastoja kannattaa opetella hyödyntämään, koska ne voivat säästää paljon aikaa. Joissakin kisoissa on mahdollista katsella kisan jälkeen muiden lähettämiä ratkaisuja. Niistä voi oppia paljon kisakoodauksesta.

1.1 Kielen valinta

Eri ohjelmointikisoissa voi käyttää eri ohjelmointikieliä. IOI:ssä sallitut kielet ovat olleet pitkään C, C++ ja Pascal. ICPC:ssä sallitut kielet taas ovat C, C++ ja Java. Joissakin nettikisoissa on valittavana huomattavasti laajempi määrä kieliä, ja Google Code Jamissa voi käyttää mitä tahansa kieltä. Mikä sitten olisi paras kielivalinta ohjelmointikisaan?

Monen kisakoodarin valinta on kieli, josta voi käyttää nimeä ”C/C++”. Ideana on koodata C-tyylistä suoraviivaista koodia, mutta käyttää C++:n ominaisuuksia tarvittaessa. Erityisesti C++:n tietorakenteet ja algoritmit ovat erittäin hyödyllisiä. C++ on tehokas kieli ja yleensä aina saatavilla. Monen mielestä sen ominaisuudet soveltuvat parhaiten kisakoodaamiseen.

Paras ratkaisu on silti hallita useita kieliä ja tuntea niiden hyvät ja huonot puolet. Esimerkiksi jos tehtävässä täytyy käsitellä suuria kokonaislukuja, Python voi olla hyvä valinta. Javassa taas on valmis kirjasto päivämäärien käsittelyyn. Toisaalta tehtävien suunnittelijat yrittävät yleensä laatia sellaisia tehtäviä, ettei tietyn kielen käyttämisestä ole kohtuutonta hyötyä.

Kaikki tämän kirjan esimerkit on kirjoitettu C++:lla, ja niissä on käytetty runsaasti C++:n tietorakenteita ja algoritmeja. Käytössä on C++:n standardi C++11, jota voi nykyään käyttää useimmissa kisoissa.

1.2 Syöte ja tuloste

Nykyään useimmissa kisoissa käytetään standardivirtoja (C++:ssa `cin` ja `cout`) syötteen lukemiseen ja tulostamiseen. Tämä on syrjäyttänyt aiemman käytännön, jossa syöte ja tuloste olivat nimetyissä tiedostoissa.

Tarkastellaan seuraavaksi standardivirtojen käyttämistä C++:ssa. Ohjelmalle tuleva syöte muodostuu yleensä luvuista ja merkkijonoista, joiden välissä on välilyöntejä ja rivinvaihtoja. Niitä voi lukea `cin`-virrasta näin:

```
int a, b;
string c;
cin >> a >> b >> c;
```

Tämä tapa toimii riippumatta siitä, miten luettavat tiedot on jaettu eri riveille ja missä kohdin syötettä on välilyöntejä. Joskus taas syötteestä täytyy lukea kokonainen rivi tietoa, jolloin voi käyttää funktiota `getline`:

```
string c;
getline(cin, c);
```

Vastaavasti tulostaminen tapahtuu `cout`-virran kautta:

```
int a = 123, b = 5;
string c = "Uolevi";
cout << a << " " << b << "\n";
cout << c << "\n";
```

Joskus liukuluku täytyy tulostaa tietyllä tarkkuudella. Tulostettavien desimaalien määrän voi valita funktiolla `setprecision`:

```
double x = 1.23456789;
cout << setprecision(5); // 5 desimaalia
cout << x << "\n";
```

Jos syötteen tai tulosteen määrä on suuri, niiden käsittely voi muodostua ohjelman pullonkaulaksi. Helppo tapa nopeuttaa ohjelmaa on lisätä alkuun:

```
ios_base::sync_with_stdio(0);
```

Huomaa myös, että `endl` rivinvaihtona on hidas:

```
cout << x << endl;
```

Tämä johtuu siitä, että `endl` aiheuttaa aina flush-operaation.

Vaihtoehtoinen tehokas tapa käsitellä syötettä ja tulostetta on käyttää C:n funktioita `scanf` ja `printf`.

1.3 Lukujen käsittely

Yleisin kisaohjelmoinnissa tarvittava lukutyyppi on `int`, joka on 32-bittinen kokonaislukutyyppi. Tällöin muuttujan pienin ja suurin mahdollinen arvo ovat -2^{31} ja $2^{31}-1$ eli noin $-2 \cdot 10^9$ ja $2 \cdot 10^9$. Joskus kuitenkaan `int`-tyyppi ei riitä, ja seuraavaksi tarkastellaan näitä tilanteita.

Suuret kokonaisluvut

Tyyppi `long long` on 64-bittinen kokonaislukutyyppi, jonka pienin ja suurin mahdollinen arvo ovat -2^{63} ja $2^{63}-1$ eli noin $-2 \cdot 10^{18}$ ja $2 \cdot 10^{18}$. Jos tehtävässä tarvitsee suuria kokonaislukuja, `long long` riittää yleensä.

On kätevää antaa tyypille lyhyt nimi esimerkiksi näin:

```
#define ll long long
```

Tämän jälkeen `ll` tarkoittaa samaa kuin `long long` ja esimerkiksi muuttujia voi määritellä näin:

```
ll a, b, c;
```

Yleinen virhe `ll`-tyypin käytössä on, että jossain kohtaa koodia käytetään kuitenkin `int`-tyyppiä. Esimerkiksi tässä koodissa on salakavala virhe:

```
int a = 123456789;
ll b = a*a;
```

Vaikka muuttuja `b` on `ll`-tyyppinen, laskussa `a*a` molemmat osat ovat `int`-tyyppisiä ja myös laskun tulos on `int`-tyyppinen. Tämän vuoksi muuttujaan `b` ilmestyy luku -1757895751 oikean luvun 15241578750190521 sijasta. Ongelman voi korjata vaihtamalla muuttujan `a` tyypiksi `ll` tai kirjoittamalla `(ll)a*a`.

Jos tehtävässä tarvitsee `ll`-tyyppiä suurempia kokonaislukuja, on kaksi vaihtoehtoa: toteuttaa tarvittava suurten lukujen käsittely itse tai käyttää C++:n sijasta toista kieltä, jossa suurten lukujen käsittely on helpompaa. Esimerkiksi Python tukee suoraan mielivaltaisen suuria kokonaislukuja.

Vastaus modulona

Joskus tehtävän vastaus on hyvin suuri kokonaisluku, mutta vastaus riittää tulostaa ”modulo M” eli vastauksen jakojäännös luvulla M (esimerkiksi ”modulo 10^9+7”). Ideana on, että vaikka todellinen vastaus voi olla suuri luku, tehtävässä riittävät tyypit `int` ja `ll`, ja modulo kertoo, toimiiko koodi oikein.

Seuraava rivi on hyödyllinen moduloa käyttäessä:

```
#define M 1000000007
```

Tärkeä modulon ominaisuus on, että yhteen- ja kertolaskussa modulon voi laskea luvuista erikseen jo ennen laskutoimitusta. Toisin sanoen seuraavat yhtälöt pitävät paikkansa:

- $(a+b)\%M = (a\%M + b\%M)\%M$
- $(a \cdot b)\%M = (a\%M \cdot b\%M)\%M$

Esimerkiksi $(342 \cdot 177)\%7 = 60534\%7 = 5$ ja toisella tavalla $342\%7 = 6$, $177\%7 = 2$ ja $(6 \cdot 2)\%7 = 5$. Tämän ansiosta jos vastauksen laskeminen muodostuu yhteen- ja kertolaskuista, jokaisen yksittäisen laskun jälkeen voi ottaa jakojäännöksen eivätkä luvut kasva liian suuriksi.

Esimerkiksi seuraava koodi laskee luvun 1000 kertoman modulo M:

```
ll v = 1;
for (int i = 1; i <= 1000; i++) {
    v = (v*i)%M;
}
cout << v << endl;
```

Modulon laskeminen negatiivisesta luvusta voi tuottaa yllätyksiä. Jos n on negatiivinen, jakojäännös $n\%M$ on C++:ssa negatiivinen. Esimerkiksi $-18\%7 = -4$. Usein on kuitenkin kätevää, että jakojäännös on aina välillä $0 \ldots M-1$. Tämä onnistuu käyttämällä kaavaa $(n\%M + M)\%M$. Nyt $(-18\%7 + 7)\%7 = 3$.

Liukuluvut

Yleensä ohjelmointikisojen tehtävissä riittää käyttää kokonaislukuja. Esimerkiksi IOI:ssä on ollut käytäntönä, ettei tehtävissä tarvitse liukulukuja. Liukulukujen käyttämisessä on ongelmana, ettei kaikkia lukuja voi esittää tarkasti ja tapahtuu pyöristysvirheitä. Näin käy seuraavassa koodissa:

```
double a = 0.3*3+0.1;
double b = 1.0;
if (a != b) cout << "a ei ole b\n";
cout << setprecision(20);
cout << a << " " << b << "\n";
```

Ohjelman tulostus on seuraava:

```
a ei ole b
0.99999999999999988898 1
```

Muuttuja a lasketaan kaavalla $0{,}3 \cdot 3 + 0{,}1$, joten sen tulisi olla 1. Näin ei ole kuitenkaan pyöristysvirheiden vuoksi, ja a on hieman alle 1. Niinpä vertailussa vaikuttaa siltä, että a ja b olisivat eri luvut.

Hyvä tapa vertailla liukulukuja on seuraava:

```
#define E 0.0000000001
```

```
if (abs(a-b) < E) cout << "a ja b ovat samat\n";
```

Ideana on tulkita kaksi liukulukua samoiksi, jos niiden etäisyys toisistaan on pienempi kuin vakio E. Vakion E valinta riippuu tehtävästä, mutta yllä olevan kaltainen valinta on yleensä hyvä.

Jos tehtävän vastaus on liukuluku, tehtävänanto kertoo yleensä vastauksen vaaditun tarkkuuden tai paljonko se saa erota oikeasta vastauksesta.

1.4 Standardikirjasto

Tärkeä osa kisakoodausta on tuntea hyvin kielen standardikirjasto tietorakenteiden ja algoritmien osalta. Tutustumme pikku hiljaa C++:n standardikirjaston sisältöön oppaan kuluessa.

Kisoissa tärkeimmät C++:n tietorakenteet ovat:

- `vector`: muuttuvan kokoinen taulukko
- `deque`: taulukko, jota voi muokata tehokkaasti alusta ja lopusta
- `set`: joukkorakenne, jossa tehokas lisäys, poisto ja haku
- `map`: taulukko vapaavalintaisilla indekseillä
- `unordered_set`: tehokkaampi `set` (C++11)
- `unordered_map`: tehokkaampi `map` (C++11)

Lisäksi kirjaston `algorithm` sisältö on syytä tuntea hyvin. Siellä on esimerkiksi valmis tehokas järjestämisalgoritmi `sort`.

Kisan aikana on yleensä aina saatavilla C++:n referenssi, joten kaikkea ei tarvitse opetella ulkoa, vaan riittää tietää, mitä standardikirjasto sisältää. Hyvä kuvaus kirjastosta on sivustolla `http://www.cplusplus.com/reference/`.

Luku 2

Tehokkuus

Oleellinen asia algoritmien suunnittelussa on tehokkuus. Yleensä on helppoa suunnitella algoritmi, joka ratkaisee tehtävän hitaasti, mutta vaikeus piilee siinä, kuinka saada algoritmi toimimaan nopeasti. Aikavaativuus on kätevä tapa arvioida algoritmin tehokkuutta koodin rakenteen perusteella.

2.1 Aikavaativuus

Aikavaativuus on algoritmin tehokkuusarvio, joka lasketaan syötteen koon perusteella. Aikavaativuus merkitään $O(\cdots)$, jossa kolmen pisteen tilalla on matemaattinen kaava. Yleensä muuttuja n kuvastaa syötteen kokoa. Esimerkiksi jos algoritmin syötteenä on lista lukuja, n on lukujen määrä, ja jos syötteenä on merkkijono, n on merkkijonon pituus.

Silmukat

Algoritmin ajankäyttö johtuu yleensä pohjimmiltaan algoritmin sisäkkäisistä silmukoista. Aikavaativuus antaa arvion siitä, montako kertaa sisimmässä silmukassa oleva koodi suoritetaan. Jos algoritmissa on vain yksi silmukka, joka käy syötteen läpi, niin aikavaativuus on $O(n)$:

```
for (int i = 0; i < n; i++) {
    // koodia
}
```

Jos algoritmissa on kaksi sisäkkäistä silmukkaa, aikavaativuus on $O(n^2)$:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // koodia
    }
}
```

Vastaavasti jos algoritmissa on k sisäkkäistä silmukkaa, aikavaativuus on $O(n^k)$.

Suuruusluokka

Aikavaativuus ei kerro tarkasti, montako kertaa silmukan sisällä oleva koodi suoritetaan, vaan se kertoo vain suuruusluokan. Esimerkiksi kaikkien seuraavien silmukoiden aikavaativuus on $O(n)$:

```
for (int i = 0; i < 3*n; i++) {
    // koodia
}
```

```
for (int i = 0; i < n+5; i++) {
    // koodia
}
```

```
for (int i = 0; i < n; i += 2) {
    // koodia
}
```

Seuraavan koodin aikavaativuus taas on $O(n^2)$:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= i; j++) {
        // koodia
    }
}
```

Tässä sisin silmukka suoritetaan $1+2+3+\cdots+n$ kertaa, mistä saadaan:

$$1+2+3+\cdots+n = \frac{1}{2}n(n+1) = \frac{1}{2}n^2+\frac{1}{2}n = O(n^2)$$

Peräkkäisyys

Jos koodissa on monta peräkkäistä osaa, kokonaisaikavaativuus on suurin yksittäisen osan aikavaativuus. Tämä johtuu siitä, että koodin hitain vaihe on yleensä koodin pullonkaula, ja muiden vaiheiden merkitys on pieni.

Esimerkiksi seuraava koodi muodostuu kolmesta osasta, joiden aikavaativuudet ovat $O(n)$, $O(n^2)$ ja $O(n)$. Niinpä koko koodin aikavaativuus on $O(n^2)$.

```
for (int i = 0; i < n; i++) {
    // koodia
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // koodia
    }
}
for (int i = 0; i < n; i++) {
    // koodia
}
```

Monta muuttujaa

Joskus syötteessä on monta muuttujaa, jotka vaikuttavat aikavaativuuteen. Tällöin aikavaativuuden kaavassa esiintyy useita muuttujia.

Esimerkiksi seuraavan koodin aikavaativuus on $O(nm)$:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        // koodia
    }
}
```

Rekursio

Rekursiivisen funktion aikavaativuus saadaan laskemalla, montako kertaa funktiota kutsutaan yhteensä ja mikä on yksittäisen kutsun aikavaativuus. Kokonaisaikavaativuus saadaan kertomalla nämä arvot toisillaan.

Tarkastellaan esimerkiksi seuraavaa funktiota:

```
void f(int n) {
    if (n == 1) return;
    f(n-1);
}
```

Kutsu f(n) aiheuttaa yhteensä n funktiokutsua, ja jokainen funktiokutsu vie vakioajan, joten aikavaativuus on $O(n)$.

Tarkastellaan sitten seuraavaa funktiota:

```
void g(int n) {
    if (n == 1) return;
    g(n-1);
    g(n-1);
}
```

Tässä tapauksessa funktio haarautuu kahteen osaan, joten kutsu g(n) aiheuttaa kaikkiaan seuraavat kutsut:

kutsu	kerrat
g(n)	1
g($n-1$)	2
g($n-2$)	4
$\cdots$	$\cdots$
g(1)	2^{n-1}

Aikavaativuus voidaan laskea seuraavasti:

$$1+2+4+\cdots+2^{n-1} = 2^n - 1 = O(2^n)$$

2.2 Vaativuusluokat

Aikavaativuus on näppärä tapa vertailla algoritmeja keskenään, ja algoritmeja on mahdollista ryhmitellä vaativuusluokkiin niiden aikavaativuuden perusteella. Käytännössä pieni määrä aikavaativuuksia riittää useimpien algoritmien luokitteluun. Seuraavassa on joukko tavallisimpia aikavaativuuksia.

$O(1)$ (vakio)

Aikavaativuus $O(1)$ tarkoittaa, että algoritmi on vakioaikainen eli sen nopeus ei riipu syötteen koosta. Käytännössä $O(1)$-algoritmi on usein suora kaava vastauksen laskemiseen.

Esimerkiksi summan $1+2+3+\cdots+n$ voi laskea ajassa $O(n)$ silmukalla

```
int s = 0;
for (int i = 1; i <= n; i++) {
    s += i;
}
```

mutta myös $O(1)$-ratkaisu on mahdollinen käyttämällä kaavaa:

```
int s = n*(n+1)/2;
```

$O(\log n)$ (logaritminen)

Logaritminen aikavaativuus $O(\log n)$ syntyy usein siitä, että algoritmi puolittaa syötteen koon joka askeleella. Tämä perustuu siihen, että luvusta n laskettu 2-kantainen logaritmi $\log_2 n$ kertoo, montako kertaa luku n täytyy puolittaa, ennen kuin päästään lukuun 1. Esimerkiksi $\log_2 32 = 5$, koska 5 puolitusta riittää:

$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$

Seuraavan algoritmin aikavaativuus on $O(\log n)$:

```
for (int i = n; i >= 1; i /= 2) {
    // koodia
}
```

Kerroin $\log n$ esiintyy usein tehokkaiden algoritmien aikavaativuudessa. Käytännön esimerkkejä tästä tulee heti kirjan seuraavissa luvuissa.

$O(n)$ (lineaarinen)

Lineaarinen aikavaativuus $O(n)$ tarkoittaa, että algoritmi käy syötteen läpi vakiomäärän kertoja. Lineaarinen aikavaativuus on usein paras mahdollinen, koska tavallisesti syöte täytyy käydä kokonaan läpi ainakin kerran, ennen kuin algoritmi voi ilmoittaa vastauksen.

Seuraava lineaarinen algoritmi kääntää merkkijonon s toisinpäin:

```
int n = s.size();
for (int i = 0; i < n/2; i++) {
    swap(s[i], s[n-1-i]);
}
```

Ideana on vaihtaa keskenään ensimmäinen ja viimeinen merkki, toinen ja toiseksi viimeinen merkki jne.

$O(n \log n)$ (järjestäminen)

Aikavaativuus $O(n \log n)$ johtuu usein siitä, että algoritmin osana on syötteen järjestäminen. Tehokkaat järjestämisalgoritmit, kuten C++:n algoritmi `sort`, toimivat ajassa $O(n \log n)$. Järjestämisen jälkeen halutun asian laskeminen syötteestä on usein helpompaa.

Seuraava algoritmi tarkistaa, onko vektorissa v kahta samaa lukua:

```
sort(v.begin(), v.end());
bool ok = false;
for (int i = 0; i+1 < v.size(); i++) {
    if (v[i] == v[i+1]) ok = true;
}
if (ok) cout << "kaksi samaa lukua";
else cout << "ei samoja lukuja";
```

Algoritmi järjestää ensin vektorin ajassa $O(n \log n)$. Tämän jälkeen algoritmi tarkistaa ajassa $O(n)$ kaikki vektorin vierekkäiset luvut. Jos vektorissa on samoja lukuja, nämä ovat vierekkäin järjestämisen jälkeen.

$O(n^2)$ (neliöllinen)

Neliöllinen aikavaativuus $O(n^2)$ tarkoittaa, että algoritmi voi käydä läpi kaikki tavat valita syötteestä kaksi alkiota:

```
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // koodia
    }
}
```

$O(n^3)$ (kuutiollinen)

Kuutiollinen aikavaativuus $O(n^3)$ tarkoittaa, että algoritmi voi käydä läpi kaikki tavat valita syötteestä kolme alkiota:

```
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = j+1; k < n; k++) {
            // koodia
        }
    }
}
```

$O(2^n)$ (osajoukot)

Aikavaativuus $O(2^n)$ voi syntyä siitä, että algoritmi käy läpi syötteen osajoukot. Esimerkiksi listan [1,2,3] osajoukot ovat [], [1], [2], [3], [1,2], [1,3], [2,3] sekä [1,2,3]. Osajoukkoja on yhteensä 2^n, koska jokainen alkio voidaan joko valita tai jättää valitsematta osajoukkoon.

$O(n!)$ (permutaatiot)

Aikavaativuus $O(n!)$ voi syntyä siitä, että algoritmi käy läpi kaikki syötteen permutaatiot. Esimerkiksi listan [1,2,3] permutaatiot ovat [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2] sekä [3,2,1]. Permutaatioita on yhteensä $n!$, koska ensimmäisen alkion voi valita n tavalla, seuraavan $n-1$ tavalla jne.

2.3 Nopeuden arviointi

Aikavaativuuden avulla voi päätellä, onko mielessä oleva algoritmi riittävän nopea tehtävän ratkaisemiseen. Aikavaativuuden voi yleensä arvioida helposti, ja sen avulla saa hyvän kuvan algoritmin tehokkuudesta toteuttamatta algoritmia. Toisaalta tehtävän syöterajoista voi yrittää päätellä myös käänteisesti, millaista algoritmia tehtävän laatija odottaa tehtävään.

Aikavaativuuden ja syötteen koon suhteen oppii kokemuksen kautta, mutta seuraavat arviot ovat hyviä lähtökohtia:

- jos $n \approx 10$, algoritmi voi olla $O(n!)$
- jos $n \approx 20$, algoritmi voi olla $O(2^n)$
- jos $n \approx 250$, algoritmi voi olla $O(n^3)$
- jos $n \approx 2000$, algoritmi voi olla $O(n^2)$
- jos $n \approx 10^5$, algoritmi voi olla $O(n \log n)$
- jos $n \approx 10^6$, algoritmi voi olla $O(n)$

Nämä arviot olettavat, että käytössä on tavallinen nykyaikainen tietokone ja koodi saa käyttää sekunnin aikaa syötteen käsittelyyn. Aikavaativuus ei kerro kuitenkaan kaikkea tehokkuudesta, vaan koodin yksityiskohtien optimointi vaikuttaa myös asiaan. Optimoimalla koodi voi nopeutua moninkertaisesti, vaikka aikavaativuus ei muuttuisikaan.

2.4 Esimerkki

Saman tehtävän ratkaisuun on usein olemassa monia erilaisia algoritmeja, joilla on eri aikavaativuudet. Seuraava tehtävä on klassinen algoritmiongelma, jonka yksinkertaisen ratkaisun aikavaativuus on $O(n^3)$. Algoritmia parantamalla aikavaativuudeksi saadaan ensin $O(n^2)$ ja lopulta $O(n)$.

Tehtävä: Suurin summa

Annettuna on taulukko `t`, jossa on n kokonaislukua. Tehtävänä on etsiä taulukosta suurin mahdollinen summa, joka saadaan laskemalla yhteen joukko peräkkäisiä lukuja.

Esimerkiksi seuraavassa taulukossa suurin summa on 15, joka saadaan laskemalla yhteen harmaalla taustalla merkityt luvut.

3	1	-5	4	-2	8	5	-3

Algoritmi 1

Seuraava algoritmi toimii ajassa $O(n^3)$:

```
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        int s = 0;
        for (int k = i; k <= j; k++) {
            s += t[k];
        }
        v = max(v, s);
    }
}
```

Algoritmi käy läpi kaikki mahdolliset taulukon välit, laskee kunkin välin lukujen summan ja pitää kirjaa suurimmasta summasta. Algoritmin päätteeksi suurin summa on muuttujassa `v`.

Algoritmi 2

Edellistä algoritmia on helppoa tehostaa laskemalla välin summaa suoraan samalla, kun käydään läpi tapoja valita välin takaraja. Tämän tehostuksen ansios-

ta algoritmista lähtee pois yksi for-silmukka.

Tuloksena on seuraava algoritmi, joka toimii ajassa $O(n^2)$:

```
for (int i = 0; i < n; i++) {
    int s = 0;
    for (int j = i; j < n; j++) {
        s += t[j];
        v = max(v, s);
    }
}
```

Algoritmi 3

Vielä nopeampi algoritmi saadaan muuttamalla lähestymistapaa ongelmaan: uusi idea on laskea jokaisessa taulukon kohdassa suurin summa, joka päättyy kyseiseen kohtaan. Osoittautuu, että suurin kohtaan k päättyvä summa voidaan laskea helposti, kun tiedetään suurin kohtaan $k-1$ päättyvä summa.

Tämä algoritmi toimii ajassa $O(n)$:

```
int s = 0;
for (int i = 0; i < n; i++) {
    if (s < 0) s = t[i];
    else s += t[i];
    v = max(v, s);
}
```

Algoritmi perustuu seuraavaan havaintoon: Jos edellisen kohdan suurin summa on negatiivinen, uusi suurin summa kannattaa aloittaa puhtaalta pöydältä ja se on pelkkä taulukon nykyisen kohdan luku. Muussa tapauksessa edellisen kohdan suurimpaan summaan lisätään nykyisen kohdan luku.

Tämä on optimaalinen algoritmi tehtävään, koska minkä tahansa algoritmin täytyy käydä taulukko ainakin kerran läpi.

Luku 3

Raaka voima

Raaka voima (brute force) on suoraviivainen tapa ratkaista lähes mikä tahansa tehtävä. Ideana on käydä kaikki mahdolliset ratkaisut läpi järjestelmällisesti ja valita paras ratkaisu tai laskea ratkaisujen yhteismäärä. Raa'an voiman ratkaisu on yleensä helppo toteuttaa ja se toimii varmasti. Jos syötteet ovat niin pieniä, että raaka voima riittää, se on yleensä paras tapa ratkaista tehtävä.

Tutustutaan seuraavaksi raa'an voiman käyttämiseen kolmen tehtävän avulla.

3.1 Osajoukot

Tehtävä: Osajoukko

Annettuna on taulukko `t`, jossa on n kokonaislukua. Tehtävänä on laskea, monessako taulukon osajoukossa lukujen summa on x. Esimerkiksi jos taulukko on $[3,7,4,9,1]$ ja osajoukon summa $x = 10$, niin vastaus on 2. Taulukon osajoukot, joissa lukujen summa on 10, ovat $[3,7]$ ja $[1,9]$.

Suoraviivainen ratkaisu tehtävään on käydä läpi kaikki osajoukot:

```
void haku(int k, int s) {
    if (k == n) {
        if (s == x) lkm++;
        return;
    }
    haku(k+1, s);
    haku(k+1, s+t[k]);
}
```

Haku lähtee käyntiin näin:

```
haku(0, 0);
```

Funktion parametri k on käsiteltävä kohta taulukossa, ja s on muodosteilla olevan osajoukon summa. Jokaisen luvun kohdalla haku haarautuu kahteen tapaukseen: luku valitaan osajoukkoon tai jätetään pois osajoukosta. Aina kun $k = n$, yksi jakotapa on valmis, jolloin funktio tarkistaa, onko sen summa oikea. Funktio laskee ratkaisujen määrän muuttujaan `lkm`.

Tämän ratkaisun aikavaativuus on $O(2^n)$, koska haussa tehdään n valintaa ja jokaiseen valintaan on kaksi vaihtoehtoa. Ratkaisu on tehokas, jos $n \approx 20$, mutta haku hidastuu nopeasti n:n kasvaessa.

3.2 Permutaatiot

Tehtävä: Merkkijono

Sinulle on annettu on merkkijono, ja saat järjestää merkit mihin tahansa järjestykseen. Montako tapaa on järjestää merkit niin, että merkkijonossa ei ole kahta samaa merkkiä peräkkäin? Esimerkiksi jos merkkijono on ABCA, tapoja on 6: ABAC, ABCA, ACAB, ACBA, BACA ja CABA.

Tehtävän voi ratkaista käymällä merkkijonon kaikki permutaatiot läpi ja laskemalla ne permutaatiot, joissa ei ole kahta samaa merkkiä peräkkäin. C++:ssa tämä on helpointa toteuttaa funktiolla `next_permutation`, jolla voi käydä läpi taulukon, vektorin tai merkkijonon permutaatiot.

Funktio `next_permutation` muodostaa välin aakkosjärjestyksessä seuraavan permutaation. Esimerkiksi jos funktiolle annetaan merkkijono ACAB, tuloksena on merkkijono ACBA. Funktion palautusarvo on `true`, jos muodostettu permutaatio on edellistä suurempi, ja muuten `false`. Tämän vuoksi haku tulee aloittaa pienimmästä permutaatiosta eli välin tulee olla järjestetty.

Seuraavassa koodissa `s` sisältää annetun merkkijonon, ja koodi laskee muuttujaan `lkm` tehtävän vastauksen:

```
sort(s.begin(), s.end());
do {
    bool ok = true;
    for (int i = 0; i < s.size()-1; i++) {
        if (s[i] == s[i+1]) ok = false;
    }
    if (ok) lkm++;
} while (next_permutation(s.begin(), s.end());
```

Koodin aikavaativuus on $O(n!n)$, koska se käy läpi $n!$ permutaatiota ja jokaisen permutaation käsittely vie aikaa $O(n)$. Koodi on tehokas, jos $n \approx 10$.

3.3 Peruuttava haku

Yleisempi raa'an voiman menetelmä on *peruuttava haku*, jossa on ideana muodostaa ratkaisu askel askeleelta ja käydä joka vaiheessa rekursiivisesti läpi kaikki eri tavat tehdä seuraava valinta. Haku peruuttaa takaisin aiempiin valintakohtiin, jotta se pystyy käymään läpi kaikki erilaiset ratkaisut.

Tehtävä: Ratsun reitti

Ratsu lähtee liikkeelle $n \times n$ -shakkilaudan vasemmasta yläkulmasta. Ratsu voi liikkua shakkipelin sääntöjen mukaisesti: kaksi askelta vaakasuunnassa ja askeleen pystysuunnassa tai päinvastoin. Tehtävänä on laskea sellaisten reittien määrä, jossa ratsu käy tasan kerran jokaisessa ruudussa. Esimerkiksi tapauksessa 5×5 yksi mahdollinen reitti on seuraava:

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Tehtävän ratkaisussa on idena simuloida ratsun kaikkia mahdollisia reittejä. Joka tilanteessa ratsulla on tietty määrä vaihtoehtoja reitin jatkamiseksi. Jos reitti päättyy umpikujaan, haku palaa takaisin aiempaan haarakohtaan. Jos taas reitti on käynyt kaikissa ruuduissa, yksi ratkaisu on valmis.

Seuraava koodi toteuttaa tämän idean:

```
void haku(int y, int x, int c) {
    if (y < 0 || x < 0 || y >= n || x >= n) return;
    if (s[y][x]) return;
    s[y][x] = c;
    if (c == n*n) {
        lkm++;
    } else {
        static int dy[] = {1, 2, 1, 2, -1, -2, -1, -2};
        static int dx[] = {2, 1, -2, -1, 2, 1, -2, -1};
        for (int i = 0; i < 8; i++) {
            haku(y+dy[i], x+dx[i], c+1);
        }
    }
    s[y][x] = 0;
}
```

Funktiota kutsutaan näin:

```
haku(0, 0, 1);
```

Ideana on muodostaa ratsun reitti taulukkoon s. Parametrit y ja x sisältävät ratsun sijainnin ja parametri c reitin askelten määrän. Jos reitissä on n^2 askelta, se on valmis ja muuttuja `lkm` kasvaa. Muuten käydään läpi kaikki kahdeksan vaihtoehtoa, mihin ratsu voi liikkua seuraavaksi.

Haun tehokkuutta on vaikeaa arvioida tarkasti, koska haku haarautuu eri vaiheissa eri tavalla. Yllä oleva toteutus ratkaisee tapauksen 5×5 salamannopeasti, mutta tapaus 6×6 vie jo paljon aikaa. Hakua voisi tehostaa huomattavasti pyrkimällä karsimaan sellaisia hakuhaaroja, jotka eivät voi johtaa ratkaisuun. Esimerkiksi jos laudalle jää yksinäinen ruutu, johon ei voi hypätä mistään muusta ruudusta, ratkaisua ei ole mahdollista saattaa loppuun.

Kyseessä on kuitenkin erittäin vaikea tehtävä suuremmilla n:n arvoilla. Ratkaisujen lukumäärä tapauksessa 8×8 – eli tavalliselle shakkilaudalle – on tiettävästi edelleen avoin ongelma.

Luku 4

Järjestäminen

Järjestäminen on yksi tavallinen tapa saada aikaan tehokas algoritmi. Tämä ja kaksi seuraavaa lukua esittelevät algoritmeja ja tietorakenteita, jotka liittyvät järjestämiseen. Ensimmäisenä aiheena on järjestämisalgoritmi: annettuna on n alkiota ja tehtävänä on saada ne suuruusjärjestykseen. Tähän on olemassa tehokkaita algoritmeja, jotka toimivat ajassa $O(n \log n)$.

4.1 `sort`-komento

C++:ssa on valmis tehokas `sort`-komento, joka on lähes aina paras valinta järjestämiseen. Komento toimii ajassa $O(n \log n)$, ja sille annetaan parametreiksi järjestettävän välin alku- ja loppukohta.

Seuraava koodi järjestää vektorin `v` sisällön:

```
sort(v.begin(), v.end());
```

Oletuksena järjestys on pienimmästä suurimpaan, mutta järjestyksen saa muutettua käänteiseksi näin:

```
sort(v.rbegin(), v.rend());
```

Tiedon ryhmittely

Joskus järjestettävänä on tietueita, jotka muodostuvat monesta osasta. Oletetaan esimerkiksi, että järjestettävänä on lista, jonka jokaisella rivillä on nimi ja pistemäärä. Rivit täytyy järjestää ensisijaisesti suurimmasta pienempään pistemäärän mukaan ja toissijaisesti aakkosjärjestykseen nimen mukaan.

Helpoin ratkaisu tähän on muodostaa pareja (`pair`), jotka järjestyvät automaattisesti oikein `sort`-komennolla. Ideana on laittaa järjestyksessä ensisijainen tieto parin ensimmäiseksi jäseneksi ja toissijainen tieto toiseksi jäseneksi. Lisäksi jos

järjestetään suurimmasta pienimpään, vastaavat luvut muutetaan negatiivisiksi. Tämän jälkeen `sort`-komento järjestää tietueet oikein.

Oletetaan esimerkiksi, että listan sisältö on:

nimi	pistemäärä
Uolevi	170
Maija	120
Aapeli	200
Liisa	170

Tavoitteena on järjestää lista ensisijaisesti pistemäärän mukaan suurimmasta pienimpään ja toissijaisesti nimen mukaan pienimmästä suurimpaan. Tavoitteena on siis seuraava järjestys:

nimi	pistemäärä
Aapeli	200
Liisa	170
Uolevi	170
Maija	120

Nyt lista koodataan ohjelmaan

```
vector<pair<int,string>> v;
v.push_back(make_pair(-170, "Uolevi"));
v.push_back(make_pair(-120, "Maija"));
v.push_back(make_pair(-200, "Aapeli"));
v.push_back(make_pair(-170, "Liisa"));
```

minkä jälkeen järjestämiseen riittää:

```
sort(v.begin(), v.end());
```

Pareja on mahdollista laittaa sisäkkäin, mikä mahdollistaa monimutkaisempia järjestämistapoja. Esimerkiksi rakenne

```
vector<pair<int,pair<string,string>>> v;
```

mahdollistaa järjestämisen ensin pistemäärän, sitten sukunimen ja lopuksi etunimen mukaan. Toinen lyhyempi vaihtoehto on C++11:n `tuple`-rakenne:

```
tuple<int,string,string> v;
```

Oma vertailufunktio

Joskus `sort`-komennolle on kätevää antaa oma vertailufunktio. Tällöin `sort`-komento käyttää annettua funktiota tutkiessaan järjestämisen aikana, mikä on kahden alkion suuruusjärjestys. Vertailufunktion täytyy olla `bool`-tyyppinen ja ottaa parametreina kaksi alkiota. Jos ensimmäinen alkio on toista pienempi, funktion tulee palauttaa `true`, ja muuten `false`.

Oletetaan, että haluamme järjestää merkkijonoja ensisijaisesti niiden pituuden mukaan ja toissijaisesti aakkosjärjestykseen. Tässä on tätä järjestämistapaa vastaava vertailufunktio:

```
bool vrt(string a, string b) {
    if (a.size() == b.size()) return a < b;
    return a.size() < b.size();
}
```

Jos merkkijonot ovat yhtä pitkät, funktio vertailee niiden aakkosjärjestystä. Muuten funktio vertailee niiden pituutta.

Tämän jälkeen vektorin voi järjestää näin:

```
sort(v.begin(), v.end(), vrt);
```

4.2 $O(n^2)$-algoritmit

Vaikka järjestämisalgoritmia ei yleensä tarvitse kirjoittaa itse `sort`-komennon ansiosta, on hyvä tuntea erilaisia tekniikoita järjestämiseen. Tutustutaan ensin muutamaan yksinkertaiseen menetelmään, jotka toimivat ajassa $O(n^2)$.

Kuplajärjestäminen

Kuplajärjestäminen käy toistuvasti taulukkoa läpi vasemmalta oikealle. Jos jossain kohtaa kaksi vierekkäistä alkiota ovat väärin päin, algoritmi korjaa niiden järjestyksen. Taulukko on järjestyksessä viimeistään n kierroksen jälkeen.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (t[j] > t[j+1]) swap(t[j], t[j+1]);
    }
}
```

Lisäysjärjestäminen

Lisäysjärjestäminen käy läpi taulukon vasemmalta oikealle ja siirtää jokaisen alkion oikealle paikalleen taulukon alkuosaan askel kerrallaan. Joka vaiheessa taulukon alkuosa käsittelykohtaan asti on järjestyksessä.

```
for (int i = 0; i < n; i++) {
    for (int j = i-1; j >= 0; j--) {
        if (t[j] > t[j+1]) swap(t[j], t[j+1]);
    }
}
```

Valintajärjestäminen

Valintajärjestäminen käy läpi taulukon vasemmalta oikealle ja etsii jokaisessa kohdassa pienimmän alkion taulukon loppuosassa. Algoritmi vaihtaa keskenään käsiteltävän alkion ja pienimmän alkion.

```
for (int i = 0; i < n; i++) {
    int p = t[i], k = i;
    for (int j = i+1; j < n; j++) {
        if (t[j] < p) {
            p = t[j]; k = j;
        }
    }
    swap(t[i], t[k]);
}
```

4.3 $O(n \log n)$-algoritmit

Nopeimmat yleiskäyttöiset järjestämisalgoritmit ovat $O(n \log n)$-aikaisia. Nämä algoritmit ovat hankalampia toteuttaa kuin $O(n^2)$-algoritmit ja usein rekursiivisia. Tutustumme seuraavaksi kahteen $O(n \log n)$-aikaiseen järjestämisalgoritmiin. Tämän jälkeen todistamme, miksi mikään vertailuun perustuva järjestämisalgoritmi ei voi toimia nopeammin kuin ajassa $O(n \log n)$.

Lomitusjärjestäminen

Lomitusjärjestäminen jakaa taulukon kahteen yhtä suureen alitaulukkoon ja järjestää molemmat osat ensin rekursiivisesti kutsumalla itseään. Tämän jälkeen algoritmi muodostaa lopullisen järjestetyn taulukon lomittamalla eli siirtämällä siihen alkioita alitaulukoista. Tämä on helppoa, koska joka vaiheessa pienin alkio on jommankumman alitaulukon alussa.

Algoritmi jakaa joka vaiheessa taulukon puoliksi, joten rekursiotasojen määrä on $O(\log n)$. Jokaisella tasolla lomittamiset vievät aikaa $O(n)$, minkä vuoksi algoritmin kokonaisaikavaativuus on $O(n \log n)$.

Pikajärjestäminen

Pikajärjestäminen valitsee taulukosta jakoalkion ja siirtää kaikki jakoalkiota pienemmät alkiot taulukon vasempaan osaan ja kaikki jakoalkiota suuremmat alkiot taulukon oikeaan osaan. Tämän jälkeen se järjestää vasemman ja oikean osan rekursiivisesti kutsumalla itseään.

Pikajärjestämisen aikavaativuus riippuu siitä, kuinka hyvin jakoalkion valinta onnistuu. Jakoalkion tulisi jakaa taulukko mahdollisimman tasaisesti puoliksi, jotta algoritmi olisi tehokas. Jos jaot onnistuvat hyvin, pikajärjestämisen aikavaativuus on $O(n \log n)$ samasta syystä kuin lomitusjärjestämisessä, mutta pahimmassa tapauksessa aikavaativuus voi olla $O(n^2)$.

Pikajärjestäminen on käytännössä hyvin tehokas järjestämisalgoritmi, koska algoritmi on kevyt ja sen pahimmat tapaukset ovat harvinaisia.

Alarajatodistus

$O(n \log n)$ on paras mahdollinen aikavaativuus vertailuihin perustuvalle järjestämisalgoritmille. Kun taulukossa on n alkiota, algoritmin täytyy pystyä tuottamaan $n!$ erilaista lopputulosta – kaikki mahdolliset taulukon alkioiden permutaatiot. Jokainen algoritmin aikana tapahtuva vertailu taas kaksinkertaistaa mahdollisten lopputulosten määrän. Niinpä vertailuja tarvitaan

$$\log(n!) = \log(1 \cdot 2 \cdots n) = \log 1 + \log 2 + \cdots + \log n = O(n \log n).$$

4.4 Laskemisjärjestäminen

Laskemisjärjestäminen on ajassa $O(n)$ toimiva järjestämisalgoritmi, joka olettaa, että järjestettävät luvut ovat melko pieniä kokonaislukuja. Ideana on laskea, montako kertaa kukin luku esiintyy taulukossa, ja muodostaa tämän kirjanpidon perusteella järjestetty taulukko.

Oletetaan, että taulukko `t` sisältää n lukua, joista jokainen on kokonaisluku välillä $0 \ldots 99$. Nyt algoritmin voi toteuttaa seuraavasti:

```
int c[100] = {0};
for (int i = 0; i < n; i++) c[t[i]]++;
int k = 0;
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < c[i]; j++) {
        t[k] = i;
        k++;
    }
}
```

Algoritmi laskee ensin taulukkoon `c`, montako kertaa kukin luku välillä $0 \ldots 99$ esiintyy taulukossa `t`. Tämän jälkeen se käy läpi luvut $0 \ldots 99$ ja lisää jokaista lukua taulukkoon `t` niin monta kertaa kuin lukee taulukossa `c`.

Laskemisjärjestämisen aikavaativuus on $O(n)$ olettaen, että taulukossa olevat luvut ovat pienempiä kuin jokin vakio. Vastaavaa laskemisideaa voi soveltaa monessa muussakin algoritmissa.

Luku 5

Binäärihaku

Yksinkertainen tapa etsiä luku k taulukosta t on tehdä silmukka, joka käy taulukon läpi alusta loppuun:

```
for (int i = 0; i < n; i++) {
    if (t[i] == k) cout << "löytyi!";
}
```

Tämä menetelmä toimii ajassa $O(n)$ ja on paras mahdollinen yleinen menetelmä. Jos taulukossa voi olla mitä tahansa lukuja, ainoa mahdollisuus on käydä kaikki luvut läpi. Mutta jos taulukko on järjestyksessä, luvun etsiminen onnistuu paljon tehokkaammin ajassa $O(\log n)$ binäärihaulla.

5.1 Algoritmi

Binäärihaku tarkistaa ajassa $O(\log n)$, esiintyykö annettu luku järjestetyssä taulukossa. Oletetetaan tästä lähtien, että taulukko on järjestetty pienimmästä suurimpaan. Esimerkiksi taulukossa

i	0	1	2	3	4	5	6	7
t[i]	2	2	3	5	7	8	8	9

luku 5 esiintyy kohdassa t[3], luku 8 esiintyy kohdissa t[5] ja t[6], mutta taulukossa ei esiinny lukua 6.

Seuraavaksi esitellään kaksi erilaista menetelmää binäärihaun toteuttamiseen.

Menetelmä 1

Ensimmäinen menetelmä on toteuttaa binäärihaku taulukon läpikäynnin tehostuksena. Ideana on käydä taulukkoa läpi hyppien: aluksi hypyn pituus on $n/2$ askelta, sitten $n/4$ askelta, sitten $n/8$ askelta jne. Mitä lähemmäs etsittävää lukua päästään, sitä pienempiä hypyt ovat.

Tässä on tämän menetelmän toteutus:

```
int x = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (x+b < n && t[x+b] <= k) x += b;
}
if (t[x] == k) cout << "löytyi!";
```

Muuttujassa x on nykyinen kohta taulukossa ja muuttujassa b on hypyn pituus. Jokaisella b:n arvolla liikutaan taulukossa eteenpäin, kunnes hyppy veisi taulukon rajojen ulkopuolelle tai alkioon, joka on etsittävää alkiota k suurempi. Aikavaativuus on $O(\log n)$, koska hypyn pituus puolittuu joka vaiheessa.

Menetelmä 2

Perinteisempi menetelmä on jäljitellä nimen etsimistä puhelinluettelosta. Ideana on aloittaa haku taulukon keskeltä ja siirtyä siitä vasempaan tai oikeaan osaan sen perusteella, miten suuri taulukon keskellä oleva luku on. Tämä jatkuu, kunnes joko luku löytyy tai etsittävästä välistä tulee tyhjä.

```
int a = 0, b = n-1;
while (a <= b) {
    int c = (a+b)/2;
    if (t[c] == k) {
        cout << "löytyi!";
        break;
    }
    if (t[c] > k) b = c-1;
    if (t[c] < k) a = c+1;
}
```

Algoritmin joka kierroksella hakuväli on $a \ldots b$ ja c on välin keskikohta. Jos etsittävä luku on kohdassa c, haku päättyy. Muuten joko a tai b siirtyy c:n viereen riippuen keskikohdan luvusta. Aikavaativuus on $O(\log n)$, koska jäljellä olevan hakuvälin pituus $b-a+1$ puolittuu joka askeleella.

5.2 Muutoskohta

Käytännössä binäärihakua tarvitsee harvoin luvun etsimiseen taulukosta, koska C++:n valmiit tietorakenteet tekevät sen tarpeettomaksi. Esimerkiksi seuraavassa luvussa esiteltävä `set`-rakenne ylläpitää joukkoa, jonka operaatioita ovat tehokas alkion lisääminen, poistaminen ja etsiminen. Sitäkin tärkeämpi binäärihaun käyttökohde on funktion muutoskohdan etsiminen.

Oletetaan, että haluamme löytää pienimmän arvon k, joka on kelvollinen ratkaisu ongelmaan. Käytössämme on funktio `ok`(x), joka palauttaa `true`, jos x on kelvollinen ratkaisu, ja muuten `false`. Lisäksi tiedämme, että `ok`(x) on `false` aina kun $x < k$ ja `true` aina kun $x \geq k$. Toisin sanoen haluamme löytää funktion `ok` muutoskohdan, jossa arvosta `false` tulee arvo `true`.

Tilanne on siis seuraava:

x	0	1	$\cdots$	$k-1$	k	$k+1$	$\cdots$
ok(x)	false	false	$\cdots$	false	true	true	$\cdots$

Nyt muutoskohta on mahdollista etsiä käyttämällä binäärihakua:

```
int x = -1;
for (int b = n/2; b >= 1; b /= 2) {
    while (x+b < n && !ok(x+b)) x += b;
}
int k = x+1;
```

Muuttujassa x on lopuksi suurin arvo, jolla ok(x) on false, joten tästä seuraava arvo $x+1$ on pienin arvo, jolla ok(x) on true.

Esimerkki

Tarkastellaan esimerkiksi seuraavaa tehtävää:

Tehtävä: Ohjelmointikerho

Ohjelmointikerhossa on n lasta, ja lapsi i haluaa saada c[i] karkkia. Kerhoa varten hankitaan m karkkipussia, joissa jokaisessa on k karkkia. Lapset saavat karkit järjestyksessä: ensin lapsi 1 saa c[1] karkkia, sitten lapsi 2 saa c[2] karkkia jne. Joka hetkellä yksi karkkipussi on auki. Jos vuorossa olevalle lapselle on riittävästi karkkeja avatussa pussissa, lapsi ottaa ne siitä. Muussa tapauksessa avattu karkkipussi heitetään roskiin ja avataan uusi, josta lapsi saa karkkinsa. Mikä on pienin karkkien määrä k, jolla kaikki lapset saavat karkkinsa?

Tämän tehtävän voi ratkaista etsimällä pienin karkkien määrä binäärihaulla. Tässä tapauksessa funktion ok(x) voi toteuttaa näin:

```
bool ok(int x) {
    int a = x, t = 1;
    for (int i = 1; i <= n; i++) {
        if (c[i] > x) return false;
        if (c[i] <= a) a -= c[i];
        else {a = x-c[i]; t++;}
    }
    return t <= m;
}
```

Funktio ok(x) testaa, mitä tapahtuu, jos joka pussissa on x karkkia. Funktio käy läpi lapset järjestyksessä ja jakaa niille karkkeja kuvatulla tavalla. Muuttujassa a on avatun pussin karkkien määrä, ja muuttujaan t lasketaan tarvittavien pussien yhteismäärä. Jos karkkien jakamisen jälkeen pusseja on kulunut korkeintaan m, valinta x on kelvollinen ratkaisu.

Binäärihakua varten tarvitaan vielä yläraja karkkipussin koolle. Yksi luonteva yläraja on lasten haluamien karkkien yhteismäärä $z = c[1]+c[2]+\cdots+c[n]$. Tämä on varmasti riittävä karkkien määrä yhdessä pussissa.

Tuloksena olevan algoritmin aikavaativuus on $O(n \log z)$. Tämä johtuu siitä, että funktiota `ok`(x) kutsutaan $O(\log z)$ kertaa binäärihaun aikana, ja jokainen funktion kutsu vie aikaa $O(n)$, koska se käy kaikki lapset läpi silmukassa.

5.3 Huippuarvo

Binäärihakua voi soveltaa myös tilanteessa, jossa taulukon alkuosassa arvot kasvavat ja loppuosassa arvot laskevat. Toisin sanoen $t[x] < t[x+1]$, kun $x < k$, ja $t[x] > t[x+1]$, kun $x >= k$. Tehtävänä on etsiä kohta, jossa arvo $t[x]$ on suurin. Huomaa, että toisin kuin tavallisessa binäärihaussa, tässä ei ole sallittua, että peräkkäiset arvot olisivat yhtä suuria.

Esimerkiksi taulukossa

i	0	1	2	3	4	5	6	7
t[i]	2	3	5	8	9	7	4	2

suurin arvo on 9 kohdassa t[4].

Ideana on etsiä binäärihaulla taulukosta viimeinen kohta x, jossa pätee $t[x] < t[x+1]$. Tästä seuraavan arvon $x+1$ täytyy olla taulukon suurin arvo. Seuraava koodi etsii taulukon suurimman arvon muuttujaan s:

```
int x = -1;
for (int b = n/2; b >= 1; b /= 2) {
    while (x+b+1 < n && t[x+b] < t[x+b+1]) x += b;
}
int s = t[x+1];
```

Luku 6

Joukkorakenteet

Joukkorakenne mahdollistaa tehokkaasti seuraavat operaatiot:

- lisää alkio x joukkoon
- poista alkio x joukosta
- tutki, onko alkio x joukossa

Joukkorakenteen toteuttamiseen on kaksi tavallista lähestymistapaa: tasapainoitettu puurakenne ja hajautustaulu. Molemmat näistä ovat melko työläitä koodata itse. Onneksi C++ sisältää näistä rakenteista tehokkaat valmiit toteutukset, jotka ovat tämän luvun aiheena.

6.1 `set`-rakenne

C++:n `set`-rakenne pitää yllä alkiojoukkoa. Alkion lisääminen, poistaminen ja hakeminen onnistuvat kaikki ajassa $O(\log n)$. Rakenteen taustalla on tasapainotettu puurakenne, joka pitää järjestyksessä joukon alkioita.

Seuraava koodi määrittelee joukon `s`, johon voi tallentaa merkkijonoja. Sitten koodi lisää joukkoon kolme merkkijonoa funktiolla `insert`.

```
set<string> s;

s.insert("apina");
s.insert("banaani");
s.insert("cembalo");
```

Funktio `count` kertoo, onko alkio joukossa:

```
if (s.count("banaani")) {
    // koodia
}
```

Funktio erase poistaa alkion joukosta:

```
s.erase("apina");
```

Seuraava koodi tulostaa kaikki joukossa olevat alkiot järjestyksessä:

```
for (auto x : s) {
    cout << x << "\n";
}
```

Kuten matematiikassa yleensä, sama alkio voi esiintyä joukossa vain kerran. Esimerkiksi seuraava koodi lisää merkkijonon ”apina” vain kerran:

```
s.insert("apina");
s.insert("apina");
s.insert("apina");
```

Rakenne multiset on muuten samanlainen kuin set, mutta siinä sama alkio voi esiintyä monta kertaa. Nyt count kertoo, montako kertaa alkio esiintyy joukossa, ja erase poistaa alkion kaikki esiintymiskerrat.

6.2 map-rakenne

Toisenlainen joukkorakenne on map, joka muistuttaa taulukkoa. Se pitää muistissa avain-arvopareja, jossa avain vastaa taulukon indeksiä. Kuitenkin taulukosta poiketen avaimet voivat olla mitä tahansa, esimerkiksi merkkijonoja.

Tässä on sanakirjan määrittely map-rakenteen avulla:

```
map<string,string> m;

m["apina"] = "monkey";
m["banaani"] = "banana";
m["cembalo"] = "harpsichord";
```

Seuraava koodi tarkistaa, onko sana sanakirjassa, ja tarjoaa käännöksen:

```
if (m.count("cembalo")) {
    cout << "käännös: " << m["cembalo"] << "\n";
} else {
    cout << "käännös puuttuu\n";
}
```

Koko sanakirjan sisällön saa listattua näin:

```
for (auto x : m) {
    cout << x.first << ": " << x.second << "\n";
}
```

6.3 Iteraattorit

Iteraattori on keskeinen työkalu joukkorakenteen käsittelyssä. Iteraattori on osoitin johonkin joukon alkioon. Esimerkiksi iteraattori `s.begin()` osoittaa joukon `s` ensimmäiseen alkioon ja iteraattori `s.end()` osoittaa viimeisen alkion jälkeiseen kohtaan. Huomaa epäsymmetria: väli on puoliavoin eli `s.begin()` osoittaa joukkoon mutta `s.end()` joukon ulkopuolelle.

Tilanne on siis tällainen:

```
{  3,  4,  6,  8,  12,  13,  14,  17  }
   ↑                                   ↑
   s.begin()                           s.end()
```

Seuraava koodi määrittelee iteraattorin `it`, joka osoittaa joukon alkuun:

```
set<int>::iterator it = s.begin();
```

Myös seuraava lyhennysmerkintä on mahdollinen C++11:ssä:

```
auto it = s.begin();
```

Iteraattoria vastaavaan joukon alkioon pääsee käsiksi `*`-merkinnällä, ja iteraattoria pystyy liikuttamaan eteen- ja taaksepäin operaatioilla `++` ja `--`. Seuraava koodi tulostaa joukon ensimmäisen ja viimeisen alkion – eli pienimmän ja suurimman alkion, koska joukko on järjestyksessä.

```
auto a = s.begin();
auto b = s.end(); b--;
cout << *a << " " << *b << "\n";
```

Funktio `find` palauttaa iteraattorin annettuun alkioon joukossa. Mutta jos alkiota ei esiinny joukossa, iteraattoriksi tulee `s.end()`.

```
if (s.find(5) != s.end()) cout << "on joukossa";
else cout << "ei ole joukossa";
```

Iteraattorin voi antaa myös funktiolle `erase`:

```
s.erase(s.find(5));
```

Tämä on hyödyllinen `multiset`-rakenteessa, koska tällä tavalla joukosta voi poistaa yksittäisen alkion esiintymiskerran.

Funktio `lower_bound`(x) palauttaa iteraattorin joukon ensimmäiseen alkioon, joka on ainakin yhtä suuri kuin x. Vastaavasti `upper_bound`(x) palauttaa iteraattorin ensimmäiseen alkioon, joka on suurempi kuin x. Jos tällaista alkiota ei ole joukossa, funktiot palauttavat arvon `s.end()`.

Esimerkiksi seuraava koodi etsii joukosta alkion, joka on lähinnä lukua x:

```
auto a = s.lower_bound(x);
if (a == s.begin()) {
    cout << *a << "\n";
} else if (a == s.end()) {
    a--;
    cout << *a << "\n";
} else {
    auto b = a; b--;
    if (x-*b < *a-x) cout << *b << "\n";
    else cout << *a << "\n";
}
```

Koodin alussa iteraattori a osoittaa joukon ensimmäiseen alkioon, joka on ainakin yhtä suuri kuin x. Jos tämä on joukon ensimmäinen alkio, tämä on x:ää lähin alkio. Jos tällaista alkiota ei ole, x:ää lähin alkio on tätä edellinen alkio. Muussa tapauksessa x:ää lähin alkio on joko tämä alkio tai tätä edellinen alkio.

Kuten tästä esimerkistä voi huomata, iteraattorien käsittely vaatii tarkkuutta, koska niihin liittyy usein poikkeustilanteita.

6.4 Hajautusrakenteet

Vaihtoehtona `set`- ja `map`-rakenteille ovat hajautusrakenteet `unordered_set` ja `unordered_map`. Nämä C++11:n rakenteet perustuvat hajautustauluun, minkä ansiosta lisäämisen, poistamisen ja hakemisen aikavaativuus on keskimäärin $O(1)$. Rakenteet ovatkin käytännössä selvästi nopeampia kuin tasapainoisiin puihin perustuvat `set` ja `map`, joiden aikavaativuus on $O(\log n)$.

Hajautusrakenteita käytetään täysin samalla tavalla kuin puurakenteita. Ainoa ero on, että nimien edessä lukee `unordered_`. Esimerkki selventää asiaa:

```
unordered_set<string> s;

s.insert("apina");
s.insert("banaani");
s.insert("cembalo");
```

Hajautusrakenteissa on kuitenkin yksi tärkeä ero liittyen puurakenteisiin: ne eivät pidä yllä alkioiden järjestystä, vaan alkiot ovat sekaisin hajautustaulussa. Tästä syystä hajautusrakenteita ei voi käyttää silloin, kun joukkoon kohdistuu alkioiden järjestystä koskevia kyselyitä. Esimerkiksi `unordered_set` ei sisällä funktioita `lower_bound` ja `upper_bound`.

Luku 7

Taulukkokikkoja

Tämä luku esittelee joitakin taulukon käsittelyyn liittyviä tekniikoita. Menetelmissä on yhteistä, että ne pitävät yllä tietoa taulukon väleistä. Summataulukko mahdollistaa minkä tahansa välin summan laskemisen vakioajassa. Kahden osoittimen tekniikat siirtävät vuorotellen välin reunoja vastaavia osoittimia. Liukuva ikkuna taas käy läpi taulukon kaikki tietyn kokoiset välit.

7.1 Summataulukko

Taulukon t summataulukko s kertoo jokaiselle taulukon luvulle, mikä on taulukon alkuosan lukujen summa kyseiseen lukuun mennessä.

Tässä on esimerkki taulukosta ja summataulukosta:

i	0	1	2	3	4	5	6	7
t[i]	2	7	6	1	2	1	5	3
s[i]	2	9	15	16	18	19	24	27

Seuraava koodi muodostaa summataulukon:

```
s[0] = t[0];
for (int i = 1; i < n; i++) {
    s[i] = s[i-1]+t[i];
}
```

Summataulukon hyötynä on, että sen avulla pystyy laskemaan minkä tahansa taulukon välin summan ajassa $O(1)$. Ideana on laskea summa kahdessa osassa: ensin lasketaan summa taulukon alusta halutun välin loppuun asti, sitten vähennetään tästä summa taulukon alusta halutun välin alkuun asti.

Tarkemmin sanoen summa välillä $a \ldots b$ saadaan laskemalla s[b] – s[$a-1$]. Esimerkiksi yllä olevassa taulukossa summa välillä $2 \ldots 5$ on $6+1+2+1=10$. Tämä saadaan summataulukosta laskemalla s[5] – s[1] = 19 – 9 = 10. Poikkeuksena on tapaus $a=0$, jolloin summan saa suoraan summataulukosta.

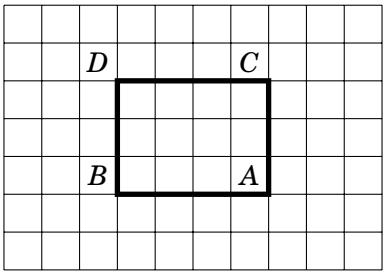
Seuraava koodi toteuttaa äskeisen idean:

```
int summa(int a, int b) {
    if (a == 0) return s[b];
    else return s[b]-s[a-1];
}
```

Summataulukon idea on mahdollista yleistää myös korkeampiin ulottuvuuksiin. Tarkastellaan esimerkkinä kaksiulotteista summataulukkoa, joka mahdollistaa minkä tahansa taulukon suorakulmaisen alueen summan laskemisen ajassa $O(1)$.

Nyt summataulukon jokaisessa kohdassa on sellaisen suorakulmion summa, joka alkaa taulukon vasemmasta yläkulmasta ja päättyy kyseiseen kohtaan. Tämän jälkeen minkä tahansa suorakulmion summan saa laskettua neljän vasemmasta yläkulmasta lähtevän suorakulmion avulla.

Lasketaan seuraavaksi lukujen summa merkityn suorakulmion alueella:



Merkityn suorakulmion summan saa laskettua kaavalla

$$S(A) - S(B) - S(C) + S(D),$$

missä $S(X)$ tarkoittaa summaa vasemmasta yläkulmasta kirjaimen X osoittamaan kohtaan asti.

7.2 Kaksi osoitinta

Tässä tekniikassa on ideana ylläpitää taulukkoon kahta osoitinta, joita siirretään vuorotellen algoritmin kuluessa. Kumpikin osoitin liikkuu vain yhteen suuntaan algoritmin aikana, minkä ansiosta osoittimet liikkuvat yhteensä vain $O(n)$ kertaa. Tämän vuoksi algoritmin kokonaisaikavaativuus on myös $O(n)$. Tutustutaan menetelmään kahden esimerkin avulla.

Esimerkki 1

Annettuna on taulukko `t`, jossa on positiivisia lukuja. Tehtävänä on selvittää, onko taulukossa väliä, jossa lukujen summa on k. Esimerkiksi jos taulukko on $[3,2,7,1,5,9]$ ja $k = 10$, tällainen väli on $[2,7,1]$.

Seuraava algoritmi käy läpi mahdollisia välejä kahden osoittimen avulla, jotka säilyttävät tietoa välin rajoista:

```
int a = 0, b = 0, s = t[0];
bool ok = false;
while (true) {
    if (s == k) {ok = true; break;}
    if (b == n-1) break;
    if (s < k) {b++; s += t[b];}
    if (s > k) {a++; s -= a[b];}
}
```

Osoitin a vastaa välin vasenta reunaa ja osoitin b välin oikeaa reunaa. Lisäksi muuttuja s pitää kirjaa välin summasta. Jos summa on liian pieni, oikea reuna siirtyy eteenpäin, ja jos summa on liian suuri, vasen reuna siirtyy eteenpäin. Muuttuja `ok` sisältää tiedon, onko haluttu väli löytynyt.

Esimerkki 2

Annettuna on taulukko `t`, joka sisältää lukuja järjestyksessä. Tehtävänä on selvittää, onko taulukossa kahta lukua, joiden summa on k. Esimerkiksi jos taulukko on $[2,3,5,6,6,8]$ ja $k = 9$, voidaan valita luvut 3 ja 6.

Nyt ideana on, että toinen osoitin liikkuu taulukossa vasemmalta oikealle ja toinen oikealta vasemmalle:

```
int a = 0, b = n-1;
bool ok = false;
while (true) {
    if (a >= b) break;
    if (t[a]+t[b] == k) {ok = true; break;}
    if (t[a]+t[b] > k) b--;
    if (t[a]+t[b] < k) a++;
}
```

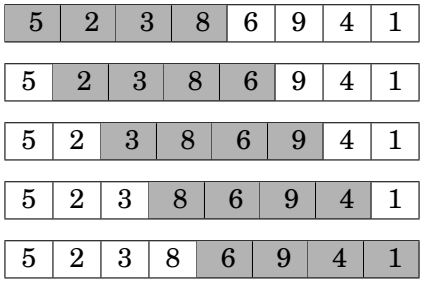
Osoitin a liikkuu vasemmalta oikealle, ja osoitin b liikkuu oikealta vasemmalle. Jos summa on liian pieni, vasen osoitin siirtyy oikealle, ja jos summa on liian suuri, oikea osoitin siirtyy vasemmalle. Jos osoittimet kohtaavat, haluttua lukuparia ei ole ja haku päättyy.

Vertailu binäärihakuun

Monen tehtävän voi ratkaista joko binäärihaulla ajassa $O(n \log n)$ tai kahdella osoittimella ajassa $O(n)$. Esimerkin 1 voisi ratkaista myös muodostamalla ensin summataulukon ja etsimällä sitten jokaisesta luvusta lähtien binäärihaulla välin oikeaa reunaa. Esimerkin 2 voisi ratkaista myös käymällä läpi kaikki taulukon luvut ja etsimällä jokaisen luvun vastaparia binäärihaulla taulukosta.

7.3 Liukuva ikkuna

Tarkastellaan tilannetta, jossa taulukossa on n alkiota, ja taulukon yli kulkee liukuva ikkuna, jonka leveys on k alkiota. Seuraavassa on esimerkki liukuvan ikkunan kulkemisesta, kun $n = 8$ ja $k = 4$:



Ikkunasta voidaan laskea summa ja minimi sen jokaisessa sijainnissa. Yllä olevassa esimerkissä summa on aluksi $5+2+3+8 = 18$ ja minimi on 2. Ikkunan liikkuessa summista muodostuu sarja 18, 19, 26, 27, 20 ja minimeistä muodostuu sarja 2, 2, 3, 4, 1. Summien laskeminen on helppoa ajassa $O(n)$, koska joka askeleella riittää lisätä ikkunaan tuleva luku summaan ja poistaa ikkunasta lähtevä luku summasta. Myös minimit voi laskea ajassa $O(n)$ seuraavasti.

Ideana on pitää yllä nousevaa jonoa minimeistä ikkunan alueella. Jonon ensimmäinen luku on tämänhetkinen minimi ja seuraavat luvut ovat mahdollisia myöhempiä minimejä. Kun luku tulee ikkunaan, jonon lopusta poistetaan kaikki luvut, jotka eivät ole sitä pienempiä, minkä jälkeen luku lisätään jonoon. Kun luku poistuu ikkunasta, se poistetaan jonon alusta, jos se on vielä siellä.

Yllä olevassa esimerkissä jonon sisältö on ensin $[2,3,8]$. Sitten luku 6 tulee ikkunaan, jolloin jonosta tulee $[2,3,6]$. Tämän jälkeen 9 tulee ikkunaan ja 2 lähtee ikkunasta, jolloin jonosta tulee $[3,6,9]$. Kahdessa viimeisessä vaiheessa jonon sisältönä on vain $[4]$ ja $[1]$, koska viimeksi ikkunaan tullut luku on minimi.

Seuraava toteutus perustuu deque-rakenteeseen, joka mahdollistaa ensimmäisen ja viimeisen alkion käsittelyn vakioajassa. Jokainen alkio on pari, jonka ensimmäinen jäsen on taulukon luku ja toinen jäsen on taulukon indeksi. Toista jäsentä tarvitaan ikkunasta poistuvan luvun käsittelyssä.

```
deque<pair<int,int>> d;
for (int i = 0; i < n; i++) {
    while (d.size() > 0 &&
           d.back().first >= t[i]) d.pop_back();
    d.push_back(make_pair(t[i],i));
    if (d.front().second == i-k) d.pop_front();
    if (i+1 >= k) cout << d.front().first << "\n";
}
```

Algoritmin kokonaisaikavaativuus on $O(n)$, koska jokainen taulukon luku lisätään kerran jonoon ja poistetaan kerran jonosta.

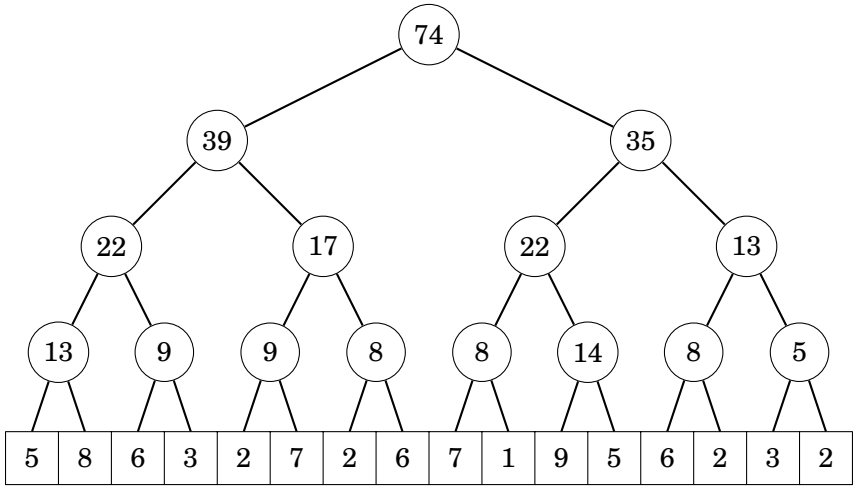
Luku 8

Segmenttipuu

Segmenttipuu on monikäyttöinen tietorakenne, joka mahdollistaa erilaisten välikyselyiden toteuttamisen muokattavaan taulukkoon. Rakenne perustuu binääripuuhun, johon on tallennettu tietoa taulukon eripituisista väleistä. Aloitamme segmenttipuuhun tutustumisen taulukon välien summien laskemisesta. Tämän jälkeen yleistämme segmenttipuun idean myös muihin välikyselyihin.

8.1 Rakenne

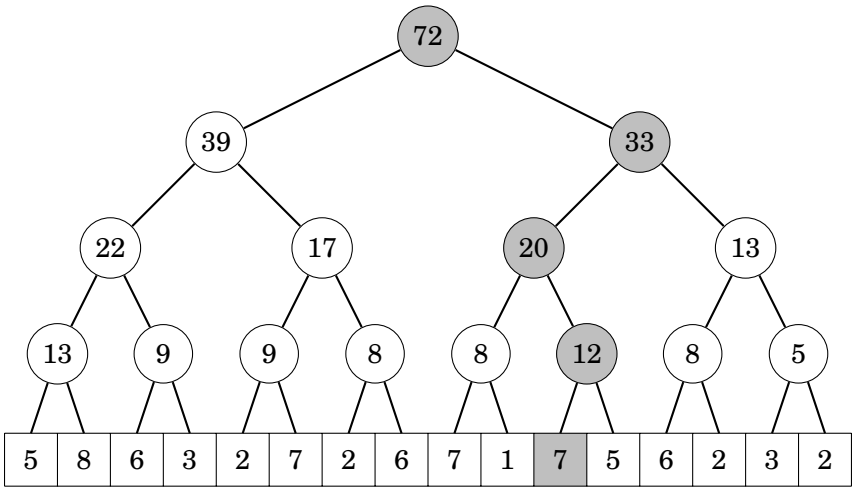
Segmenttipuu on puurakenne, jonka alimmalla tasolla on taulukon sisältö ja ylemmillä tasoilla on välikyselyihin tarvittavaa tietoa. Seuraavassa kuvassa on 16-alkioista taulukkoa vastaava segmenttipuu, jota voidaan käyttää summien laskemiseen. Jokainen puun alkio on summa kahdesta alemman tason alkioista.



Segmenttipuu on mukavinta rakentaa niin, että taulukon koko on 2:n potenssi, koska silloin tuloksena on täydellinen binääripuu. Jatkossa oletamme aina, että taulukko täyttää tämän vaatimuksen. Jos taulukon koko ei ole 2:n potenssi, sen loppuun voi lisätä tyhjää niin, että koosta tulee 2:n potenssi.

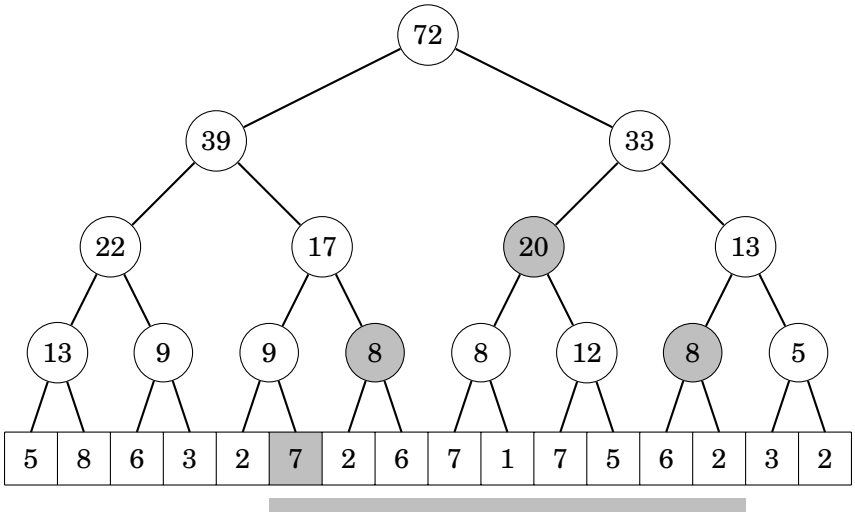
Muuttaminen

Kun taulukkoa muutetaan, myös muutoskohtaan liittyviä solmuja ylempänä puussa täytyy päivittää. Tämä tapahtuu kulkemalla puuta ylöspäin huipulle asti ja tekemällä muutokset. Seuraava kuva näyttää edellisen puun muuttumisen, kun taulukon luvusta 9 tulee 7:



Välikysely

Välikysely laskee annetun taulukon välin summan. Ideana on muodostaa välin summa käyttäen mahdollisimman korkealla puussa olevia summia. Seuraava kuva näyttää, kuinka taulukon alapuolelle harmaalla merkityn välin summan laskeminen tapahtuu. Välin summan laskemiseksi riittää hakea neljä osasummaa segmenttipuusta, ja summaksi tulee $7 + 8 + 20 + 8 = 43$.



8.2 Toteutus

Tehokas tapa toteuttaa segmenttipuu on tallentaa puu taulukkoon. Oletetaan, että segmenttipuuhun täytyy tallentaa N lukua (N on 2:n potenssi), ja varataan segmenttipuulle taulukko p, johon mahtuu $2N$ alkiota:

```
#define N 16
int p[2*N];
```

Segmenttipuun sisältö tallennetaan taulukkoon huipulta lähtien niin, että p[1] on ylin summa, p[2] ja p[3] ovat seuraavan tason summat jne. Segmenttipuun alin taso eli taulukon sisältö tallennetaan kohdasta p[N] eteenpäin. Huomaa, että kohta p[0] on käyttämätön, koska puussa on yhteensä $2N-1$ alkiota.

Funktio `muuta` muuttaa kohdan k arvoksi x:

```
void muuta(int k, int x) {
    k += N;
    p[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        p[k] = p[2*k]+p[2*k+1];
    }
}
```

Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasummat puun huipulle asti. Taulukon p indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k+1$.

Funktio `summa` laskee summan välillä $a \ldots b$:

```
void summa(int a, int b) {
    a += N; b += N;
    int q = 0;
    while (a <= b) {
        if (a%2 == 1) q += p[a++];
        if (b%2 == 0) q += p[b--];
        a /= 2; b /= 2;
    }
    return q;
}
```

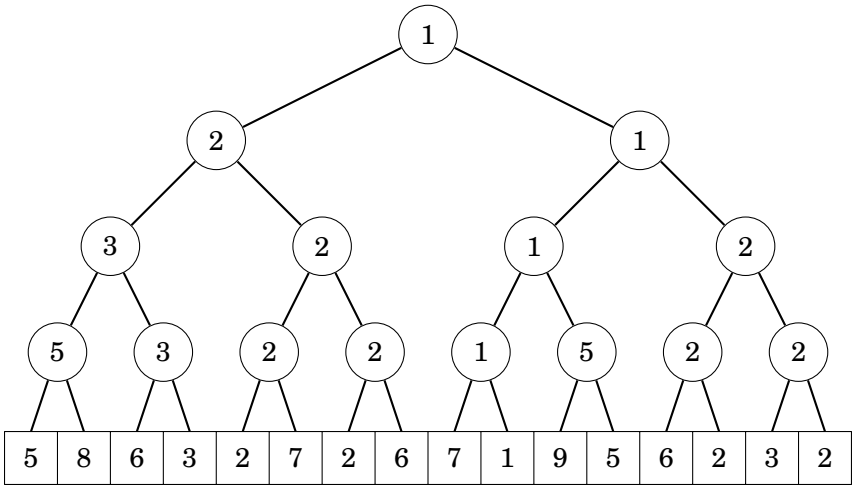
Ideana on aloittaa summan laskeminen segmenttipuun pohjalta ja liikkua askel kerrallaan ylemmälle tasolle. Funktio laskee summan muuttujaan s yhdistämällä puussa olevia osasummia. Osasumma lisätään summaan silloin, kun ylemmälle tasolle siirtyminen toisi summaan mukaan väliin kuulumattomia lukuja.

Molemmat segmenttipuun operaatiot toimivat ajassa $O(\log n)$, koska n alkiota sisältävässä segmenttipuussa on $O(\log n)$ tasoa ja operaatiot siirtyvät askel kerrallaan segmenttipuun tasoja ylöspäin.

8.3 Kyselytyypit

Segmenttipuu mahdollistaa summan lisäksi minkä tahansa välikyselyn, jonka pystyy laskemaan osissa summaa vastaavasti. Toisin sanoen kyselyn tulee vastata liitännäistä kaksipaikkaista funktiota. Tällaisia kyselyitä ovat esimerkiksi minimi ja maksimi, suurin yhteinen tekijä sekä bittioperaatiot and, or ja xor.

Esimerkiksi tässä on aiemmasta taulukosta tehty minimipuu:



Ja tässä on minimipuun käsittelyyn tarvittava koodi:

```
void muuta(int k, int x) {
    k += N;
    p[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        p[k] = min(p[2*k],p[2*k+1]);
    }
}
```

```
void minimi(int a, int b) {
    a += N; b += N;
    int q = p[a];
    while (a <= b) {
        if (a%2 == 1) q = min(q,p[a++]);
        if (b%2 == 0) q = min(q,p[b--]);
        a /= 2; b /= 2;
    }
    return q;
}
```

Huomaa, että tässä kyselyssä muuttujaa q ei voi alustaa nollaksi, koska todellinen välin minimi voi hyvin olla nollaa suurempi.

8.4 Välin muuttaminen

Tähän mennessä olemme tehneet segmenttipuita, joiden operaatiot ovat yksittäisen arvon muuttaminen ja kysely väliltä. Tutkitaan lopuksi tilannetta, jossa pitääkin muuttaa välejä ja kysellä yksittäisiä arvoja. Tarkemmin sanoen haluamme toteuttaa operaation ”kasvata kaikkia välin arvoja x:llä”, missä x on mikä tahansa luku (voi olla myös negatiivinen).

Yllättävää kyllä, voimme käyttää aiemman kaltaista segmenttipuuta myös tässä tilanteessa. Tämä vaatii, että muutamme taulukkoa niin, että jokainen taulukon arvo kertoo *muutoksen* edelliseen arvoon nähden. Esimerkiksi taulukosta

i	0	1	2	3	4	5	6	7
$t[i]$	3	3	1	1	1	5	2	2

tulee seuraava:

i	0	1	2	3	4	5	6	7
$t[i]$	3	0	−2	0	0	4	−3	0

Minkä tahansa vanhan arvon saa uudesta taulukosta laskemalla summan taulukon alusta kyseiseen kohtaan asti. Esimerkiksi kohdan 6 vanha arvo 2 saadaan summana $3-2+4-3=2$.

Uuden tallennustavan etuna on, että välin muuttamiseen riittää muuttaa kahta taulukon kohtaa. Esimerkiksi jos välille $1\ldots4$ lisätään luku 2, taulukon kohtaan 1 lisätään 2 ja taulukon kohdasta 5 poistetaan 2. Tulos on tässä:

i	0	1	2	3	4	5	6	7
$t[i]$	3	2	−2	0	0	2	−3	0

Yleisemmin kun taulukon välille $a\ldots b$ lisätään x, kohtaan $\texttt{t}[a]$ lisätään x ja kohdasta $\texttt{t}[b+1]$ vähennetään x. Näin ollen tarvittavat operaatiot ovat summan laskeminen taulukon alusta tiettyyn kohtaan sekä yksittäisen alkion muuttaminen. Voimme käyttää siis segmenttipuuta samaan tapaan kuin ennenkin.

Luku 9

Pyyhkäisyviiva

Pyyhkäisyviiva on tekniikka, jossa on ideana muuntaa ongelma yksittäisiksi tapahtumiksi, jotka voidaan järjestää esimerkiksi ajanhetken tai sijainnin mukaan. Tapahtumien järjestäminen auttaa ongelman ratkaisemista tehokkaasti, koska ne muuttavat silloin kokonaisuutta helposti hallittavalla tavalla. Tutustumme tässä luvussa pyyhkäisyviivaan kahden tehtävän avulla.

9.1 Aikavälit

Tehtävä: Ravintola

Ravintolassa käy illan mittaan n henkilöä, ja tiedossasi on jokaisen henkilön tuloaika (taulukko `t`) ja poistumisaika (taulukko `p`). Tehtäväsi on laskea, montako henkilöä ravintolassa on illan mittaan enimmillään. Voit olettaa, että vektoreissa ei esiinny kahta samaa ajanhetkeä.

Tästä tehtävästä syntyy luontevasti kahdenlaisia tapahtumia: (1) henkilö tulee ravintolaan, (2) henkilö poistuu ravintolasta.

Jokainen tapahtuma 1 kasvattaa ravintolassa olijoiden määrää yhdellä, ja vastaavasti jokainen tapahtuma 2 vähentää määrää yhdellä. Seuraava ratkaisu tallentaa tapahtumat vektoriin `v` pareina, joiden ensimmäinen jäsen on tapahtuma-aika ja toinen jäsen on 1 (henkilö tulee) tai -1 (henkilö poistuu).

```
vector<pair<int,int>> v;
for (int i = 0; i < n; i++) {
    v.push_back(make_pair(t[i], 1));
    v.push_back(make_pair(p[i], -1));
}
sort(v.begin(), v.end());
int c = 0, s = 0;
for (int i = 0; i < 2*n; i++) {
    c += v[i].second;
    s = max(s, c);
}
```

Tapahtumien järjestämisen jälkeen riittää käydä läpi tapahtumat järjestyksessä ja ylläpitää laskuria henkilömäärästä. Muuttujassa c on tämänhetkinen henkilömäärä ja muuttujassa s on tähän mennessä suurin henkilömäärä.

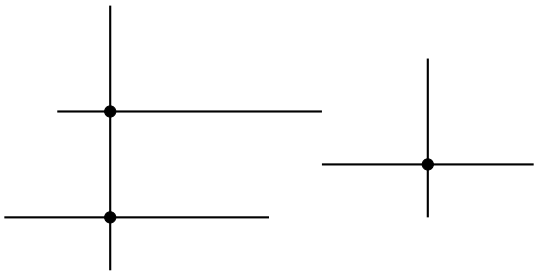
Koodin aikavaativuus on $O(n \log n)$, koska koodin työläin vaihe on tapahtumien järjestäminen.

9.2 Leikkauspisteet

Tehtävä: Leikkauspisteet

Annettuna on n janaa, joista osa on vaakasuuntaisia ja osa pystysuuntaisia. Tehtäväsi on laskea, montako leikkauspistettä janoilla on. Voit olettaa, että mitkään samansuuntaiset janat eivät mene toistensa päälle. Lisäksi voit olettaa, että sama piste ei ole koskaan monen janan päätepiste.

Esimerkiksi seuraavassa tilanteessa leikkauspisteitä on 3:



Tämä tehtävä on helppoa ratkaista ajassa $O(n^2)$: käydään läpi kaikki mahdolliset janaparit ja tutkitaan, moniko leikkaa toisiaan. Seuraavaksi ratkaisemme tehtävän ajassa $O(n \log n)$ pyyhkäisyviivan avulla.

Ideana on tarkastella tehtävää kolmenlaisina tapahtumina x-akselilla: (1) vaakajana alkaa, (2) vaakajana päättyy, (3) pystyjana.

Tapahtumat käydään läpi vasemmalta oikealle x-koordinaatin mukaan. Vaakajanoista ylläpidetään tietorakennetta, josta pystyy tarkistamaan, missä y-koordinaateissa on tällä hetkellä aktiivisia janoja. Jokaisen pystyjanan kohdalla taas lasketaan kyseiseen janaan liittyvät leikkauspisteet.

Oletetaan aluksi, että janojen y-koordinaatit ovat kokonaislukuja välillä $1 \ldots 10^5$. Nyt sopiva tietorakenne aktiivisten vaakajanojen tallentamiseen on edellisen luvun mukainen segmenttipuu.

Segmenttipuussa jokaiselle y-koordinaatille on oma kohtansa. Kun vaakajana alkaa, kohtaa k kasvatetaan, missä k on janan y-koordinaatti. Kun taas vaakajana päättyy, kohtaa k vähennetään. Pystyjanan kohdalla lasketaan segmenttipuusta summa janan päätepisteiden välistä. Tämä kertoo kyseiseen janaan liittyvien leikkauspisteiden määrän.

9.3 Pakkaaminen

Jos janojen koordinaatit voivat olla mitä tahansa, edellinen ratkaisu ei toimi, koska segmenttipuun tekeminen ei onnistu. Ongelman voi kuitenkin kiertää koordinaattien pakkaamisen avulla.

Pakkaaminen perustuu yksinkertaiseen ideaan: syötteessä on n janaa, joten syötteessä on enintään $2n$ erilaista x- ja y-koordinaattia. Tehtävän ratkaisussa on olennaista vain koordinaattien järjestys toisiinsa nähden, joten voimme valita koordinaatit uudestaan, kunhan niiden järjestys säilyy. Luonnollinen tapa on valita koordinaateiksi luonnollisia lukuja $1, 2, 3, \ldots$

Oletetaan esimerkiksi, että haluamme pakata koordinaatit $(423, 791)$, $(-233, 820)$ sekä $(599, 125)$. Järjestetyt x-koordinaatit ovat -233, 423 ja 599, ja järjestetyt y-koordinaatit ovat 125, 791 ja 820. Tämän perusteella vastaavat pakatut koordinaatit ovat $(2, 2)$, $(1, 3)$ ja $(3, 1)$.

Koordinaattien pakkauksen jälkeen pystymme löytämään leikkauspisteet ajassa $O(n \log n)$, koska segmenttipuun kooksi riittää $2n$.

Luku 10

Lukuteoria

Lukuteoria on kokonaislukuja tutkiva matematiikan ala. Tämä luku esittelee lukuteorian perusalgoritmeja, jotka liittyvät lukujen jaollisuuteen. Ensin käymme läpi algoritmeja, joilla voi tarkistaa, onko luku alkuluku, sekä etsiä luvun alkutekijät. Tämän jälkeen aiheena on suurimman yhteisen tekijän laskeminen.

10.1 Alkuluvut

Positiivinen kokonaisluku on alkuluku, jos se on vähintään 2 ja jaollinen vain 1:llä ja itsellään. Ensimmäiset alkuluvut ovat:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47

Seuraava funktio tutkii, onko luku k alkuluku:

```
bool tutki(int k) {
    for (int i = 2; i < k; i++) {
        if (k%i == 0) return false;
    }
    return true;
}
```

Funktio käy läpi kaikki kokonaisluvut välillä $2 \ldots k-1$ ja tutkii k:n jaollisuutta niillä. Jos jokin luvuista jakaa k:n, k ei ole alkuluku, ja muuten k on alkuluku. Tämän menetelmän aikavaativuus on $O(k)$.

Helppo tapa nopeuttaa tarkistusta on tämä:

```
bool tutki(int k) {
    for (int i = 2; i*i <= k; i++) {
        if (k%i == 0) return false;
    }
    return true;
}
```

Funktio on muuten samanlainen kuin ennenkin, mutta se käy välin $2 \ldots k-1$ sijasta läpi vain välin $2 \ldots \sqrt{k}$. Funktion toiminta perustuu siihen, että jos k ei ole alkuluku eli $k = a \cdot b$, pätee varmasti $a \leq \sqrt{k}$ tai $b \leq \sqrt{k}$. Jos näin ei olisi eli $a > \sqrt{k}$ ja $b > \sqrt{k}$, niin $a \cdot b > k$, mikä ei ole mahdollista. Tämän muutoksen jälkeen alkuluvun tarkistus vie aikaa vain $O(\sqrt{k})$.

10.2 Eratostheneen seula

Eratostheneen seula on tehokas tapa etsiä kaikki alkuluvut välillä $2 \ldots n$. Algoritmi tuottaa taulukon, joka kertoo jokaisesta välin luvusta, onko kyseinen luku alkuluku. Ideana on olettaa aluksi, että jokainen välin luku on alkuluku, ja sitten poistaa kirjanpidosta lukuja, jotka ovat jaollisia pienemmillä alkuluvuilla.

Seuraava toteutus tuottaa taulukon a, jossa a[k] = 0, jos luku k on alkuluku, ja a[k] = 1, jos luku k ei ole alkuluku.

```
for (int i = 2; i <= n; i++) {
    if (a[i]) continue;
    for (int j = i*i; j <= n; j += i) {
        a[j] = 1;
    }
}
```

Algoritmi käy läpi kaikki välin luvut. Jos luku k on alkuluku, algoritmi merkitsee taulukkoon, että luvut $2k, 3k, 4k, \ldots$ eivät ole alkulukuja. Tämän jälkeen välin $2 \ldots n$ luvusta k voi tarkistaa näin, onko se alkuluku:

```
if (a[k]) cout << "ei alkuluku";
else cout << "alkuluku";
```

Eratostheneen seula toimii erittäin nopeasti, vaikka siinä onkin kaksi silmukkaa sisäkkäin. Lasketaan seuraavaksi algoritmin aikavaativuus. Oletetaan ensin yksinkertaistetusti, että sisäsilmukka suoritettaisiin jokaiselle i:lle riippumatta siitä, onko i alkuluku vai ei. Tällöin sisäsilmukkaa suoritetaan n/i kierrosta ja silmukan suorituskertojen määrä on *harmoninen summa*:

$$\sum_{i=2}^{n} n/i = n/2 + n/3 + n/4 + \cdots + n/n = O(n \log n)$$

Yläraja $O(n \log n)$ saadaan korvaamalla summassa jokainen n:n jakaja edellisellä 2:n potenssilla. Tällöin summa muuttuu muotoon $2 \cdot n/2 + 4 \cdot n/4 + 8 \cdot n/8 + \ldots$ ja se on suurempi kuin alkuperäinen summa. Nyt summa muodostuu $O(\log n)$ osiosta, joista jokaisen suuruus on n, joten summa on yhteensä $O(n \log n)$

Eratostheneen seulan aikavaativuuden arvio on siis $O(n \log n)$. Todellisuudessa algoritmi on vielä nopeampi, koska sisäsilmukka suoritetaan vain silloin, kun i on alkuluku. On mahdollista osoittaa, että välillä $2 \ldots n$ on $O(n/\log n)$ alkulukua ja Eratostheneen seulan aikavaativuus on vain $O(n \log \log n)$.

10.3 Jako tekijöihin

Minkä tahansa kokonaisluvun voi esittää alkulukujen tulona eli alkutekijöinä. Esimerkiksi $60 = 2^2 \cdot 3 \cdot 5$, eli luvun 60 alkutekijät ovat 2, 2, 3 ja 5.

Seuraava koodi jakaa luvun k alkutekijöihin:

```
for (int i = 2; i*i <= k; i++) {
    while (k%i == 0) {
        cout << i << "\n";
        k /= i;
    }
}
if (k > 1) cout << k << "\n";
```

Koodin toiminta muistuttaa alkulukutarkistusta. Koodi käy läpi mahdollisia k:n tekijöitä ja joka vaiheessa tulostaa tekijän ja jakaa k:n sillä niin kauan kuin k on jaollinen tekijällä. Jos silmukan jälkeen $k > 1$, tämä tarkoittaa, että yksi alkuperäisen k:n tekijöistä on suurempi kuin $\sqrt{k}$.

Eratostheneen seulaa on myös mahdollista laajentaa niin, että sen avulla voi jakaa minkä tahansa välin $2 \ldots n$ luvun tehokkaasti alkutekijöihin. Tämä onnistuu lisäämällä algoritmiin taulukon b, jossa b[k] sisältää jonkin luvun k alkutekijän siinä tapauksessa, että k ei ole alkuluku:

```
for (int i = 2; i <= n; i++) {
    if (a[i]) continue;
    for (int j = i*i; j <= n; j += i) {
        a[j] = 1;
        b[j] = i;
    }
}
```

Tämän jälkeen luvun k voi jakaa tekijöihin näin:

```
while (a[k]) {
    cout << b[k] << "\n";
    k /= b[k];
}
cout << k << "\n";
```

10.4 Suurin yhteinen tekijä

Kokonaislukujen a ja b *suurin yhteinen tekijä* eli syt(a,b) on suurin kokonaisluku, joka jakaa sekä a:n että b:n. Esimerkiksi lukujen 24 ja 36 suurin yhteinen tekijä on 12. Suoraviivainen tapa etsiä suurin yhteinen tekijä on jakaa luvut alkutekijöiksi ja valita suurin yhteinen potenssi jokaista alkutekijää Esimerkiksi $24 = 2^3 \cdot 3$ ja $36 = 2^2 \cdot 3^2$, joten syt$(24,36) = 2^2 \cdot 3 = 12$.

Tehokas tapa laskea suurin yhteinen tekijä on käyttää Eukleideen algoritmia:

```
int syt(int a, int b) {
    if (b == 0) return a;
    return syt(b, a%b);
}
```

Esimerkiksi syt(24,36) = syt(36,24) = syt(24,12) = syt(12,0) = 12. Eukleideen algoritmin aikavaativuus on $O(\log n)$, jossa $n = \min(a,b)$, ja tutustumme myöhemmin tarkemmin algoritmin toimintaan.

Suurin yhteinen tekijä on mahdollista laskea myös useammalle luvulle ketjuttamalla syt-funktiota. Esimerkiksi lukujen 6, 12, 14 ja 20 suurin yhteinen tekijä on syt(syt(syt(6,12),14),20) = syt(syt(6,14),20) = syt(2,20) = 2.

Suurimman yhteisen tekijän sukulainen on a:n ja b:n *pienin yhteinen moninkerta* eli pym(a,b). Se on pienin kokonaisluku, joka on jaollinen sekä a:lla että b:llä. Esimerkiksi pym(4,6) = 12. Pienimmän yhteisen moninkerran saa laskettua suurimman yhteisen tekijän avulla kaavalla pym$(a,b) = ab/\text{syt}(a,b)$. Esimerkiksi pym$(24,36) = 24 \cdot 36/\text{syt}(24,36) = 72$.

Luku 11

Bittien käsittely

Tietokone käsittelee lukuja sisäisesti bittimuodossa, ja tätä pystyy hyödyntämään myös ohjelmoinnissa. Tämä luku esittelee C++:n bittioperaatiot, jotka käsittelevät lukua bitteinä. Näiden avulla syntyy tietorakenne bittitaulukko, joka mahdollistaa tehokkaan osajoukkojen käsittelyn.

11.1 Luku bitteinä

Binäärijärjestelmässä luvut esitetään käyttäen kahta numeroa eli bittiä. Ideana on muodostaa luvut samalla tavalla kuin tutussa kymmenjärjestelmässä, mutta käytettävissä ovat vain numerot 0 ja 1. Seuraavassa taulukossa on lukujen 0–31 bittiesitykset:

luku	bitteinä	luku	bitteinä	luku	bitteinä	luku	bitteinä
0	0	8	1000	16	10000	24	11000
1	1	9	1001	17	10001	25	11001
2	10	10	1010	18	10010	26	11010
3	11	11	1011	19	10011	27	11011
4	100	12	1100	20	10100	28	11100
5	101	13	1101	21	10101	29	11101
6	110	14	1110	22	10110	30	11110
7	111	15	1111	23	10111	31	11111

Luvun bittiesityksen saa laskettua kätevästi jakamalla lukua 2:lla, kunnes luvusta tulee 0. Luvun bittiesitys muodostuu näin saaduista jakojäännöksistä käänteisessä järjestyksessä. Esimerkiksi luvun 22 bittiesitys 10110 syntyy näin:

- 22/2 = 11, jää 0
- 11/2 = 5, jää 1
- 5/2 = 2, jää 1
- 2/2 = 1, jää 0
- 1/2 = 0, jää 1

Bittiesityksen saa takaisin luvuksi kertomalla jokainen bitti sen kohtaa vastaavalla 2:n potenssilla. Esimerkiksi bittiesityksestä 10110 tulee

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 22.$$

11.2 Bittioperaatiot

C++:n bittioperaatiot ovat and (&), or (|), xor (^), negaatio (~) sekä bittisiirrot (<< ja >>). Nämä operaatiot tekevät muutoksia lukujen bitteihin. Katsotaan seuraavaksi, miten operaatiot toimivat tarkkaan ottaen.

And-operaatio

And-operaatio a & b tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa sekä a:ssa ja b:ssä on ykkösbitti. Esimerkiksi 22 & 26 = 18:

	22	10110
	26	11010
&	18	10010

Or-operaatio

Or-operaatio a | b tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa a:ssa tai b:ssä on ykkösbitti. Esimerkiksi 22 | 26 = 30:

	22	10110
	26	11010
|	30	11110

Xor-operaatio

Xor-operaatio a ^ b tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa joko a:ssa tai b:ssä (ei molemmissa) on ykkösbitti. Esimerkiksi 22 ^ 26 = 12:

	22	10110
	26	11010
^	12	01100

Negaatio

Negaatio ~a kääntää luvun bitit, eli jokaisesta nollabitistä tulee ykkösbitti ja päinvastoin. Negaation tulkinta lukuna riippuu luvun tyypistä. Tavallisesti luvun bittiesityksen ensimmäinen bitti ilmaisee, onko luku positiivinen vain negatiivinen, minkä vuoksi negaatio muuttaa luvun merkkiä.

Esimerkiksi ~22 tuottaa tuloksen −23, jos luvun tyyppinä on `int`. Tämä johtuu siitä, että `int`-luvut tallennetaan sisäisesti kahden komplementtina, jolloin $\sim a = -a - 1$. Bittimuodossa tilanne näyttää seuraavalta:

$$\begin{aligned} 22 \quad & 00000000000000000000000000010110 \\ -23 \quad & 11111111111111111111111111101001 \end{aligned}$$

Bittejä on yhteensä 32, koska `int`-tyyppi on 32-bittinen.

Bittisiirto

Bittisiirrot siirtävät luvun bittejä vasemmalle tai oikealle. Operaatio $a << k$ siirtää bittejä k askelta vasemmalle, ja operaatio $a >> k$ siirtää bittejä k askelta oikealle. Vasemmalle siirtäessä oikealle ilmestyy nollabittejä, ja oikealle siirtäessä oikealta poistuu bittejä.

Esimerkiksi 5 << 2 = 20, koska 5 on bittimuodossa 101 ja siirron jälkeen tuloksena on 10100 eli lukuna 20. Vastaavasti 26 >> 3 = 3, koska 26 on bittimuodossa 11010 ja siirron tuloksena on 11 eli 3.

Sovelluksia

Seuraava koodi tarkistaa, onko luvun a bitti k oikealta laskien ykkösbitti:

```
if (a&(1<<k)) {
```

Seuraava koodi muuttaa luvun a bitin k ykkösbitiksi:

```
a |= (1<<k);
```

Seuraava koodi muuttaa luvun a bitin k nollabitiksi:

```
a &= (~(1<<k));
```

Seuraava koodi muuttaa luvun a bitin k päinvastaiseksi:

```
a ^= (1<<k);
```

Huomaa myös, että luvun viimeinen bitti kertoo parillisuuden eli a & 1 on 0, jos a on parillinen, ja 1, jos a on pariton. Lisäksi $a << k$ vastaa luvun kertomista 2:lla k kertaa ja $a >> k$ vastaa luvun jakamista 2:lla k kertaa.

Bittioperaatioiden yhteydessä kannattaa käyttää runsaasti sulkuja, koska C++:n käyttämä laskentajärjestys voi yllättää. Esimerkiksi merkintä `a&1<<k` tulkitaan `(a&1)<<k` eikä `a&(1<<k)`.

11.3 Bittitaulukko

Yksi tapa ajatella luvun bittejä on tulkita luku taulukkona, jossa jokainen alkio on 0 tai 1. Taulukon koko riippuu luvun tyypistä: esimerkiksi 32-bittistä `int`-tyyppiä käyttäessä taulukossa on 32 alkiota. Luku 1234 on bitteinä 10011010010, joten se vastaa seuraavaa taulukkoa:

kohta	0	1	2	3	4	5	6	7	8	9	10	11	⋯	31
arvo	0	1	0	0	1	0	1	1	0	0	1	0	⋯	0

Bittitaulukko tulkitaan yleensä yllä olevalla tavalla, eli luvun viimeinen bitti on taulukon ensimmäinen alkio, toiseksi viimeinen bitti on toinen alkio jne.

Kätevä bittitaulukon sovellus on joukon osajoukon esittäminen bittitaulukkona. Ideana on, että ykkösbitti tarkoittaa kyseisen luvun kuulumista osajoukkoon. Esimerkiksi `int`-luvun avulla voi esittää minkä tahansa joukon $\{0,1,\ldots,31\}$ osajoukon. Luku 1234 tarkoittaa näin ollen osajoukkoa $\{1,4,6,7,10\}$.

Nyt bittioperaatioilla voi toteuttaa tehokkaasti joukko-operaatioita:

- a & b on joukkojen a ja b leikkaus (tämä sisältää alkiot, jotka ovat kummassakin joukossa)
- a | b on joukkojen a ja b yhdiste (tämä sisältää alkiot, jotka ovat ainakin toisessa joukossa)
- a & ($\sim b$) on joukkojen a ja b erotus (tämä sisältää alkiot, jotka ovat joukossa a mutta eivät joukossa b)

Lisäksi kaikki tietyn kokoisen joukon osajoukot voi käydä läpi for-silmukalla. Seuraava koodi käy läpi kaikki k alkion joukon osajoukot:

```
for (int b = 0; b < (1<<k); b++) {
    // osajoukon b käsittely
}
```

Oletetaan esimerkiksi, että taulukko `t` sisältää k kokonaislukua, ja haluamme etsiä tavan jakaa luvut kahteen ryhmään niin, että molemmissa ryhmissä lukujen summa on sama. Tämä onnistuu käymällä kaikki osajoukot läpi:

```
for (int b = 0; b < (1<<k); b++) {
    int s1 = 0, s2 = 0;
    for (int i = 0; i < k; i++) {
        if (b&(1<<i)) s1 += t[i];
        else s2 += t[i];
    }
    if (s1 == s2) cout << "ok\n";
}
```

Koodi laskee muuttujaan `s1` osajoukossa olevien lukujen summan ja muuttujaan `s2` muiden lukujen summan. Jos summat ovat samat, jako on onnistunut.

Osa II

Verkkoalgoritmit

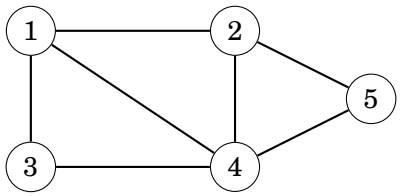
Luku 12

Verkkojen perusteet

Monen ohjelmointitehtävän voi ratkaista tulkitsemalla tehtävän verkko-ongelmana ja käyttämällä sopivaa verkkoalgoritmia. Esimerkki verkosta on tieverkosto, jonka rakenne muistuttaa luonnostaan verkkoa. Joskus taas verkko kätkeytyy syvemmälle ongelmaan ja sitä voi olla vaikeaa huomata. Tämän luvun aiheina ovat verkkoihin liittyvä käsitteistö sekä verkon tallentaminen ohjelmassa.

12.1 Määritelmiä

Verkko on tietorakenne, joka muodostuu *solmuista* ja niitä yhdistävistä *kaarista*. Esimerkiksi tieverkostossa verkon solmut ovat kaupunkeja ja kaaret ovat niiden välisiä teitä. Tässä on verkko, johon kuuluu 5 solmua ja 7 kaarta:

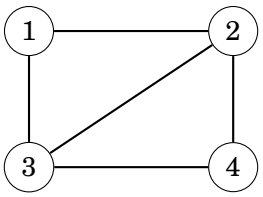


Polku on verkon kaaria pitkin kulkeva reitti kahden verkon solmun välillä. Yllä olevassa verkossa solmusta 1 solmuun 5 voi kulkea esimerkiksi polkua $1 \to 4 \to 5$ tai $1 \to 3 \to 4 \to 2 \to 5$. Solmun *naapuri* on toinen solmu, johon solmusta pääsee kaarta pitkin, ja solmun *aste* on sen naapurien määrä. Yllä olevassa verkossa solmun 2 aste on 3 ja sen naapurit ovat 1, 4 ja 5.

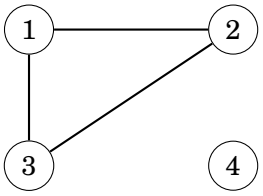
Yhtenäisyys

Verkko on *yhtenäinen*, jos minkä tahansa kahden solmun välillä on polku. Esimerkiksi tieverkosto on yhtenäinen, jos kaikista verkostoon kuuluvista kaupungeista pääsee toisiinsa teitä pitkin.

Tässä on esimerkki yhtenäisestä verkosta:

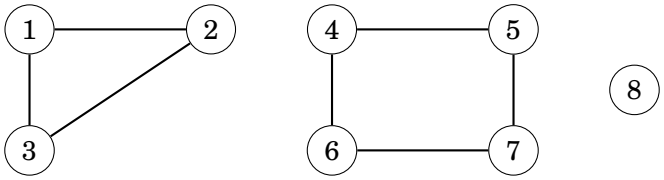


Seuraava verkko taas ei ole yhtenäinen, koska solmu 4 on erillään muista.



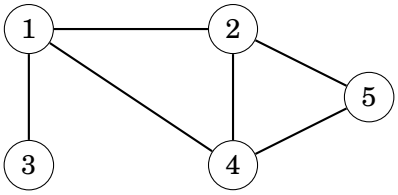
Komponentit

Verkon yhtenäiset osat muodostavat sen *komponentit*. Esimerkiksi seuraavassa verkossa on kolme komponenttia: {1, 2, 3}, {4, 5, 6, 7} ja {8}.

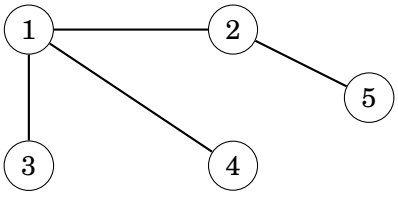


Syklit

Sykli on polku, jonka alku- ja loppusolmu on sama ja jossa ei ole samaa kaarta monta kertaa. Verkko on *syklinen*, jos siinä on sykli, ja muuten *syklitön*. Seuraava verkko on syklinen, koska siinä on sykli $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$.

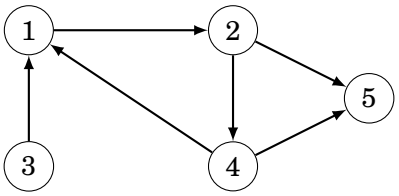


Seuraava verkko taas on syklitön:



Kaarten suunnat

Verkko on *suunnattu*, jos verkon kaaria pystyy kulkemaan vain niiden merkittyyn suuntaan. Seuraavassa on esimerkki suunnatusta verkosta:

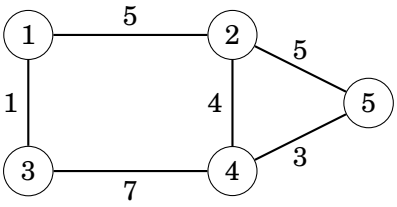


Tässä verkossa on esimerkiksi polku $3 \to 1 \to 2 \to 5$ sekä sykli $2 \to 4 \to 1 \to 2$. Solmuun 3 ei pääse mistään muusta solmusta, ja vastaavasti solmusta 5 ei pääse mihinkään muuhun solmuun.

Jos verkon kaarilla ei ole suuntaa, niin verkko on *suuntaamaton*. Toisaalta suuntaamattoman verkon voi tulkita suunnatuksi verkoksi niin, että jokaisen kaaren tilalla on kaaret molempiin suuntiin.

Kaarten painot

Verkko on *painotettu*, jos verkon kaariin liittyy painoja. Tavallinen tulkinta on, että painot kuvaavat kuvaavat matkoja solmujen välillä. Seuraavassa on esimerkki painotetusta verkosta:



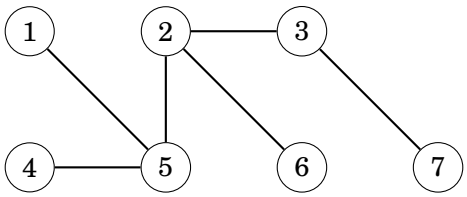
Polun *pituus* saadaan laskemalla yhteen siihen kuuluvien kaarten painot. Esimerkiksi polun $1 \to 2 \to 4$ pituus on $5+4=9$ ja polun $1 \to 3 \to 4$ pituus on $1+7=8$. Jälkimmäinen polku on *lyhin* polku solmusta 1 solmuun 4.

Jos verkon kaarilla ei ole painoja, niin verkko on *painottamaton*. Tilannetta voi ajatella myös niin, että jokaisen kaaren paino on 1.

Puu

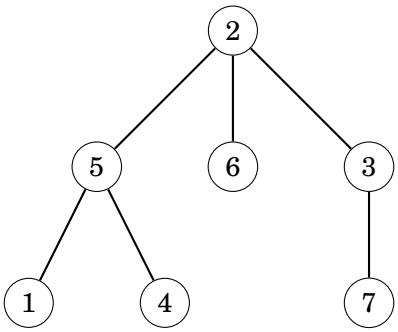
Verkko on *puu*, jos se on yhtenäinen, syklitön ja suuntaamaton. Toisin sanoen verkko on puu, jos jokaisen kahden solmun välillä on yksikäsitteinen polku.

Esimerkiksi seuraava verkko on puu:



Puussa kaarten määrä on aina yhden pienempi kuin solmujen määrä: esimerkiksi yllä olevassa puussa on 7 solmua ja 6 kaarta. Jos puuhun lisää yhden kaaren, siihen tulee sykli, ja jos siitä poistaa yhden kaaren, se ei ole enää yhtenäinen.

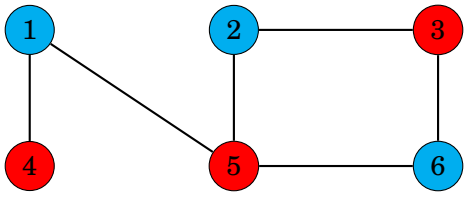
Puu esitetään usein niin, että yksi solmuista nostetaan puun *juureksi* ja muut sijoittuvat tasoittain sen alapuolelle. Esimerkiksi jos äskeisessä verkossa solmusta 2 tehdään juuri, tulos on tämä:



Kaksijakoisuus

Verkko on *kaksijakoinen*, jos sen solmut voi värittää kahdella värillä niin, ettei minkään kaaren molemmissa päissä ole samanväristä solmua.

Esimerkiksi seuraava verkko on kaksijakoinen, koska verkosta voi värittää solmut 1, 2 ja 6 sinisiksi ja solmut 3, 4 ja 5 punaisiksi.

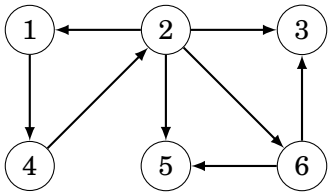


Verkko on kaksijakoinen tarkalleen silloin, kun siinä ei ole parittoman pituista sykliä. Tällaista sykliä ei ole mahdollista värittää kahdella värillä niin, että kaikki vierekkäin olevat solmut olisivat eri värisiä.

12.2 Verkon tallennus

Verkon tallennukseen ohjelmassa on monia eri tapoja. Sopiva tallennustapa riippuu siitä, miten verkkoa täytyy pystyä käsittelemään algoritmissa. Seuraavaksi esitellään kolme yleistä tapaa verkon tallentamiseen.

Esimerkkinä käytetään seuraavaa verkkoa:



Tämä verkko on suunnattu, eli kaaret ovat yksisuuntaisia. Jos verkko on suuntaamaton, niin yleensä hyvä ratkaisu on tallentaa jokainen sen kaari kahtena yksisuuntaisena kaarena.

Tästä lähtien oletuksena on, että verkko sisältää N solmua, jotka on numeroitu $1,2,\ldots,N$. C++:ssa taulukoiden indeksointi alkaa 0:sta, minkä vuoksi taulukon koon täytyy olla yhtä suurempi kuin verkon koon.

Vieruslistat

Verkon vieruslistaesityksessä jokaisella solmulla on vieruslista, joka sisältää kaikki solmut, joihin kyseisestä solmusta pääsee kaarta pitkin. Kätevä tapa tallentaa vieruslistat on luoda taulukko, jossa jokaiselle solmulle on oma vektori:

```
vector<int> v[N+1];
```

Tämän jälkeen verkon kaaret lisätään näin:

```
v[1].push_back(4);
v[2].push_back(1);
v[2].push_back(3);
v[2].push_back(5);
v[2].push_back(6);
v[4].push_back(2);
v[6].push_back(3);
v[6].push_back(5);
```

Vieruslistaesityksen etuna on, että sen avulla on nopeaa käydä läpi solmusta lähtevät kaaret. Tämä onnistuu käytännössä seuraavasti for-silmukalla:

```
cout << "solmusta " << k << " lähtee ";
cout << v[k].size() << " kaarta:\n";
for (int i = 0; i < v[k].size(); i++) {
    cout << v[k][i] << "\n";
}
```

Jos verkko on painotettu, vieruslistoihin täytyy liittää myös tieto kaarten painoista. Yksi hyvä ratkaisu on tallentaa vieruslistoihin pareja, joissa on solmun tunnus sekä siihen johtavan kaaren paino:

```
vector<pair<int,int>> v[N+1];
```

Useimmat verkkoalgoritmit pystyy toteuttamaan tehokkaasti käyttäen verkon vieruslistaesitystä, minkä vuoksi vieruslistat ovat tavallisesti hyvä valinta verkon tallennusmuodoksi.

Vierusmatriisi

Toinen tapa tallentaa verkko on muodostaa vierusmatriisi, jossa kerrotaan jokaisesta mahdollisesta kaaresta, onko se mukana verkossa vai ei. Vierusmatriisina voi käytännössä käyttää kaksiulotteista taulukkoa:

```
int verkko[N+1][N+1];
```

Ideana on, että verkko[a][b] on 1, jos solmusta a on kaari solmuun b, ja muuten 0. Seuraava koodi lisää esimerkin kaaret verkkoon:

```
verkko[1][4] = 1;
verkko[2][1] = 1;
verkko[2][3] = 1;
verkko[2][5] = 1;
verkko[2][6] = 1;
verkko[4][2] = 1;
verkko[6][3] = 1;
verkko[6][5] = 1;
```

Jos verkko on painotettu, luvun 1 voi luontevasti korvata kaaren painolla.

Vierusmatriisin etuna on, että kahden solmun välisen kaaren tarkastaminen sekä kaaren lisääminen tai poistaminen onnistuvat nopeasti. Vierusmatriisi vie kuitenkin paljon tilaa, jos verkko on suuri. Luvun 17 Floyd-Warshallin algoritmi perustuu verkon käsittelyyn vierusmatriisina.

Kaarilista

Kolmas tapa tallentaa verkko on muodostaa lista, joka sisältää kaikki verkon kaaret. Tämän voi käytännössä toteuttaa vektorina, joka sisältää kaaripareja:

```
vector<pair<int,int>> v;
```

Kaaret lisätään listalle näin:

```
v.push_back(make_pair(1,4));
v.push_back(make_pair(2,1));
v.push_back(make_pair(2,3));
v.push_back(make_pair(2,5));
v.push_back(make_pair(2,6));
v.push_back(make_pair(4,2));
v.push_back(make_pair(6,3));
v.push_back(make_pair(6,5));
```

Seuraava koodi tulostaa kaikki verkon kaaret:

```
for (auto x : v) {
    cout << x.first << " " << x.second << "\n";
}
```

Jos verkko on painotettu, kaarilistaa voi laajentaa näin:

```
vector<pair<pair<int,int>,int>> v;
```

Nyt kaarilista muodostuu pareista, joiden ensimmäinen jäsen on kaaren solmupari ja toinen jäsen on kaaren paino.

Kaarilista on hyvä tapa tallentaa verkko, jos algoritmissa täytyy käydä läpi kaikki verkon kaaret eikä kaaria tarvitse etsiä sen perusteella, mihin solmuihin ne liittyvät. Luvun 16 Bellman-Fordin algoritmi sekä luvun 19 Kruskalin algoritmi ovat mukavasti toteutettavissa käyttäen kaarilistaa.

Luku 13

Syvyyshaku

Syvyyshaku on yksinkertainen menetelmä verkon läpikäyntiin. Ideana on lähteä liikkeelle jostain verkon solmusta ja käydä kaikissa solmuissa, joihin kyseisestä solmusta pääsee jotain reittiä. Syvyyshaun avulla voi esimerkiksi tutkia, onko verkossa reittiä kahden solmun välillä, etsiä verkon yhtenäiset komponentit sekä tarkistaa, onko verkossa sykliä.

13.1 Algoritmi

Helpoin tapa toteuttaa syvyyshaku perustuu rekursioon. Oletetaan, että verkko on tallennettu vieruslistoina taulukkoon v ja lisäksi taulukossa t pidetään kirjaa siitä, missä solmuissa syvyyshaku on jo käynyt:

```
vector<int> v[N+1];
int t[N+1];
```

Syvyyshaun toteutus on seuraavanlainen:

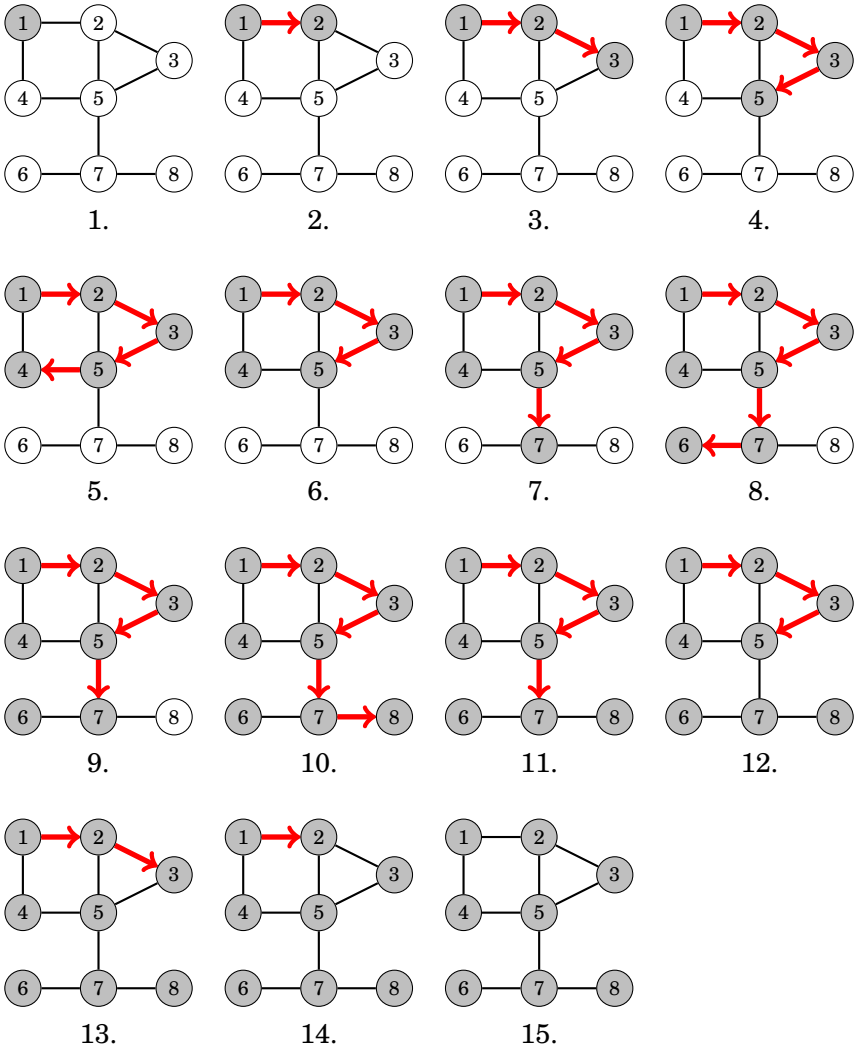
```
void haku(int s) {
    if (t[s]) return;
    t[s] = 1;
    for (int i = 0; i < v[s].size(); i++) {
        haku(v[s][i]);
    }
}
```

Funktio haku saa parametrinaan tutkittavan solmun. Aluksi funktio tarkistaa, onko solmussa jo käyty, ja poistuu saman tien tässä tapauksessa. Sitten funktio merkitsee solmussa käynnin ja käy läpi rekursiivisesti kaikki solmut, joihin kyseisestä solmusta pääsee.

Syvyyshaun solmusta 1 lähtien voisi aloittaa näin:

```
haku(1);
```

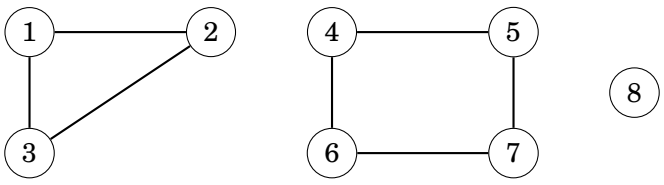
Seuraava kuvasarja esittää syvyyshaun suoritusta verkon solmusta 1 alkaen. Harmaat solmut ovat tutkittuja solmuja ja punaiset nuolet kuvaavat sillä hetkellä muistissa olevaa polkua.



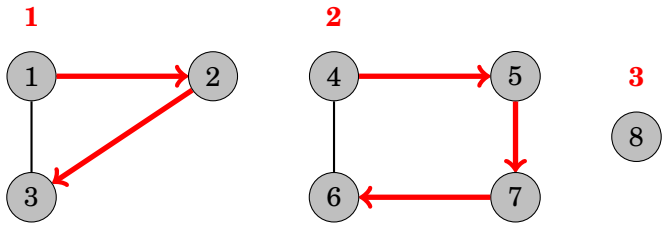
13.2 Verkon komponentit

Syvyyshaun yksi tavallinen sovellus on verkon jakaminen komponentteihin. Jokainen verkon komponentti on yhtenäinen solmujoukko, eli kaikista komponentin solmuista pääsee toisiinsa. Ideana on suorittaa jokaiselle komponentille erillinen syvyyshaku, joka etsii komponentissa olevat solmut.

Esimerkiksi verkon



komponentit löytyvät kolmella syvyyshaulla näin:



Numeroidaan verkon komponentit $1, 2, \ldots, c$, jossa c on komponenttien määrä. Taulukon t tulkintana on nyt, että siinä oleva arvo on komponentin numero. Arvo 0 tarkoittaa kuitenkin tutkimatonta solmua kuten ennenkin.

Syvyyshaun toteutus on lähes samanlainen kuin ennenkin. Erona kuitenkin funktiossa haku on uusi parametri c, joka sisältää komponentin numeron. Syvyyshaku merkitsee jokaisen saavuttamansa solmun komponentiksi c.

```
void haku(int s, int c) {
    if (t[s]) return;
    t[s] = c;
    for (int i = 0; i < v[s].size(); i++) {
        haku(v[s][i], c);
    }
}
```

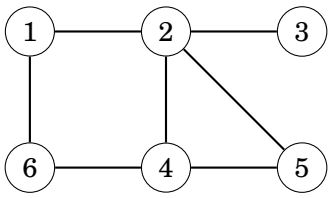
Tämän jälkeen komponentit löytyvät näin:

```
int c = 0;
for (int i = 1; i <= N; i++) {
    if (t[i]) continue;
    c++;
    haku(i, c);
}
```

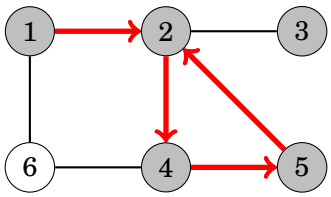
13.3 Syklin etsiminen

Verkossa olevan syklin tunnistaa siitä, että syvyyshaku saapuu uutta reittiä solmuun, jossa se on jo käynyt aiemmin. Tällöin syklin muodostaa syvyyshaun käsiteltävänä olevan reitin loppuosa, jonka alku- ja loppusolmu on sama.

Esimerkiksi verkossa



sykli $2 \to 4 \to 5 \to 2$ löytyy seuraavasti:



Syklin etsimiseen riittää pieni muutos syvyyshaun koodiin:

```
void haku(int s, int q) {
    if (t[s]) {
        cout << "sykli löytyi!";
        return;
    }
    t[s] = 1;
    for (int i = 0; i < v[s].size(); i++) {
        if (v[s][i] == q) continue;
        haku(v[s][i], s);
    }
}
```

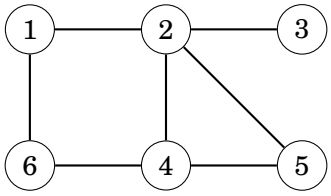
Erona aiempaan syvyyshaussa on uusi parametri q, joka sisältää edellisen solmun tunnuksen. Tämän parametrin ansiosta syvyyshaku haarautuu vain niihin suuntiin, joista se ei ole tullut, jolloin aiemmin käytyyn solmuun palaaminen tarkoittaa, että verkossa on sykli.

Huomaa, että yllä oleva koodi ei välttämättä löydä verkon kaikkia syklejä, mutta se löytää ainakin yhden niistä, jos syklejä on olemassa.

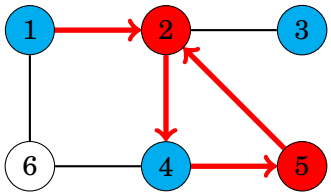
13.4 Kaksijakoisuus

Verkon kaksijakoisuuden tarkistaminen perustuu ovelaan ideaan: väritetään syvyyshaun lähtösolmu siniseksi ja siitä lähtien joka toinen solmu punaiseksi ja siniseksi. Mutta jos jossain kohtaa verkkoa syntyy ristiriita, verkko ei ole kaksijakoinen. Nyt verkon solmun mahdolliset tilat ovat: 0 (solmussa ei ole käyty), 1 (solmu on sininen) ja 2 (solmu on punainen).

Esimerkiksi verkko



ei ole kaksijakoinen, koska syvyyshaussa käy seuraavasti:



Ongelmaksi muodostuu, että solmun 2 väriksi on valittu punainen, mutta siihen pääsee myös punaisen solmun 5 kautta, joten sen värin tulisi olla sininen.

Seuraavassa toteutuksessa funktiossa `haku` on uusi parametri x, joka on 1 tai 2: solmun tuleva väri. Jos solmu on jo oikean värinen, sille ei tehdä mitään. Jos taas solmu on väärän värinen, on selvää, että verkko ei ole kaksijakoinen.

```
void haku(int s, int x) {
    if (t[s] == x) return;
    if (t[s] == 3-x) {
        cout << "ei kaksijakoinen";
        return;
    }
    t[s] = x;
    for (int i = 0; i < v[s].size(); i++) {
        haku(v[s][i], 3-x);
    }
}
```

Kaavan $3-x$ tarkoituksena on muuttaa luku 1 luvuksi 2 ja päinvastoin.

Luku 14

Leveyshaku

Leveyshaku on toinen perusmenetelmä verkon läpikäyntiin. Se käy läpi verkon solmut lähtösolmusta alkaen järjestyksessä sen mukaan, kuinka kaukana solmut ovat lähtösolmusta. Leveyshaun etuna onkin, että se löytää jokaiseen solmuun lyhimmän polun lähtösolmusta. Algoritmi on kuitenkin vaikeampi toteuttaa kuin syvyyshaku, minkä vuoksi sitä kannattaa käyttää vain silloin, kun verkosta täytyy löytää lyhimmät polut solmuihin.

14.1 Algoritmi

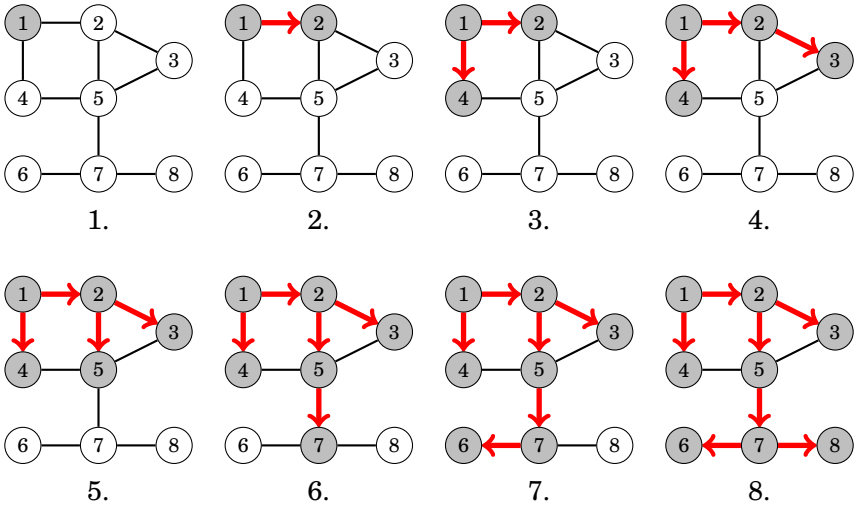
Leveyshaku kerää solmuja listaan lähtösolmusta alkaen. Aluksi listan ainoana solmuna on lähtösolmu. Sitten algoritmi valitsee joka vaiheessa käsiteltäväksi seuraavan uuden solmun listasta. Solmun käsittelyssä leveyshaku lisää listan loppuun kaikki uudet solmut, joihin solmusta pääsee.

Oletetaan taas, että taulukko v sisältää verkon vieruslistoina. Seuraava toteutus kerää solmut listaksi vektoriin z. Taulukko t kertoo, onko solmu lisätty jo listaan, ja taulukko e sisältää solmun etäisyyden lähtösolmusta. Verkon läpikäynti alkaa solmusta a, jonka käsittely on ennen silmukkaa.

```
vector<int> v[N+1];
vector<int> z;
int t[N+1], e[N+1];
```

```
t[a] = 1;
e[a] = 0;
z.push_back(a);
for (int i = 0; i < z.size(); i++) {
    for (int j = 0; j < v[z[i]].size(); j++) {
        int u = v[z[i]][j];
        if (t[u]) continue;
        t[u] = 1;
        e[u] = e[z[i]]+1;
        z.push_back(u);
    }
}
```

Seuraava kuvasarja esittää leveyshaun suoritusta verkon solmusta 1 alkaen. Punaiset nuolet kuvaavat kaaria, joita seuraamalla leveyshaku on löytänyt lyhimmät polut verkon solmuihin.



Algoritmin suorituksen päätteeksi taulukon e sisältö on

i	1	2	3	4	5	6	7	8
e[i]	0	1	2	1	2	4	3	4

eli esimerkiksi solmuun 7 on 3 askeleen polku $1 \to 2 \to 5 \to 7$.

14.2 Polun selvitys

Välillä lyhimmän polun pituuden lisäksi täytyy selvittää polkuun kuuluvat solmut. Tämä onnistuu lisäämällä leveyshakuun taulukon p, joka kertoo jokaiselle solmulle edellisen solmun tähän solmuun johtavalla lyhimmällä polulla.

```
int p[N+1];
```

Taulukkoa päivitetään leveyshaussa yhtä aikaa kuin taulukkoa e:

```
        e[u] = e[z[i]]+1;
        p[u] = z[i];
```

Edellisessä esimerkissä taulukon sisällöksi tulisi

i	1	2	3	4	5	6	7	8
p[i]	0	1	2	1	2	7	5	7

eli esimerkiksi solmuun 7 johtava lyhin polkua menee solmua 7 ennen solmuun 5, solmua 5 ennen solmuun 2 ja solmua 2 ennen solmuun 1.

Seuraava funktio `polku` tulostaa koko polun lähtösolmusta parametrina annettuun solmuun asti:

```
function polku(int s) {
    if (p[s] != 0) polku(p[s]);
    cout << s << "\n";
}
```

Funktio kutsuu ensin itseään rekursiivisesti parametrina polun edellinen solmu ja tulostaa sitten nykyisen solmun tunnuksen. Tämän ansiosta polun solmut tulostuvat oikeassa järjestyksessä lähtösolmusta päätesolmuun.

14.3 Haku ruudukossa

Tyypillinen esimerkki leveyshaun soveltamisesta on lyhimmän reitin etsiminen kahden ruudukon ruudun välillä. Esimerkiksi ruudukossa

```
##########
#........#
#.#.####.#
#A#..B...#
##########
```

lyhin reitti A:sta B:hen on

```
##########
#***.....#
#*#*####.#
#A#**B...#
##########
```

johon kuuluu 8 askelta.

Tämän tehtävän voi ratkaista leveyshaulla tulkitsemalla ruudukon verkkona niin, että jokainen lattiaruutu on verkon solmu ja jokaisen vierekkäisen lattiaruudun välillä on kaari. Kuitenkaan ruudukkoa ei tarvitse muuttaa erikseen verkoksi, vaan leveyshaun voi toteuttaa suoraan ruudukossa.

Oletetaan, että ruudukon korkeus on N ja leveys on M. Taulukossa `r` on ruudukon sisältö merkkeinä: mahdolliset merkit ovat `#`, `.`, `A` sekä `B`. Vektorissa `z` on lista tutkituista ruuduista `int`-pareina. Lisäksi taulukossa `e` on etäisyys ruudusta `A` kaikkiin muihin ruutuihin.

```
char r[N][M];
vector<pair<int,int>> z;
int e[N][M];
```

Leveyshaun toteutus on nyt seuraavanlainen:

```
int y1, x1, y2, x2;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (r[i][j] == 'A') {y1 = i; x1 = j;}
        if (r[i][j] == 'B') {y2 = i; x2 = j;}
    }
}
z.push_back(make_pair(y1,x1));
r[y1][x1] = '#';
for (int i = 0; i < z.size(); i++) {
    int y = z[i].first;
    int x = z[i].second;
    static int dy[] = {1, 0, -1, 0};
    static int dx[] = {0, 1, 0, -1};
    for (int j = 0; j < 4; j++) {
        int uy = y+dy[j];
        int ux = x+dx[j];
        if (r[uy][ux] == '#') continue;
        z.push_back(make_pair(uy,ux));
        r[uy][ux] = '#';
        e[uy][ux] = e[y][x]+1;
    }
}
if (r[y2][x2] == 'B') cout << "ei polkua";
else cout << "lyhin polku: " << e[y2][x2] << " askelta";
```

Koodin alussa etsitään A:n ja B:n sijainti ruudukossa. Tämän jälkeen alkaa varsinainen leveyshaku.

Tämä toteutus täyttää ruudukon lattioita merkillä # aloitusruudusta lähtien. Tämän ansiosta ei tarvita erillistä taulukkoa kertomaan, missä ruuduissa on käyty, mutta toisaalta ruudukon sisältö tuhoutuu haun aikana.

Jos haun päätteeksi lopetusruudussa on merkki #, haku on päässyt sinne ja lyhimmän polun pituus on taulukossa e.

Luku 15

Dijkstran algoritmi

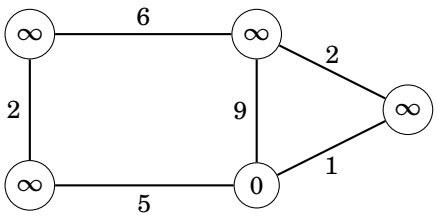
Leveyshaku on hyvä tapa etsiä lyhin polku verkossa, jos verkko on painottamaton eli jokaisen kaaren paino on 1. Mutta jos verkon kaarilla on painot, leveyshaku ei enää toimi. Dijkstran algoritmi etsii tehokkaasti lyhimmän polun lähtösolmusta kaikkiin muihin solmuihin olettaen, että jokaisen kaaren paino on epänegatiivinen. Tämä oletus pätee useimpiin verkkoihin – esimerkiksi tieverkostossa ei ole mielekästä ajatella, että tien pituus voisi olla negatiivinen.

15.1 Algoritmi

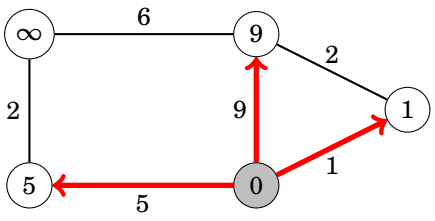
Dijkstran algoritmi pitää yllä verkon jokaiselle solmulle kahta arvoa: tieto siitä, onko solmu jo käsitelty, sekä etäisyysarvio lähtösolmusta kyseiseen solmuun. Haun alussa lähtösolmun etäisyysarviona on 0 ja kaikkien muiden solmujen etäisyysarviona on ∞.

Algoritmi valitsee joka askeleella solmun, jota ei ole vielä käsitelty ja jossa on mahdollisimman pieni etäisyysarvio. Tässä vaiheessa solmun etäisyysarviosta tulee lopullinen etäisyys kyseiseen solmuun. Lisäksi algoritmi parantaa niiden solmujen etäisyysarvioita, joihin pääsee käsiteltävän solmun kautta aiempaa lyhyempää reittiä.

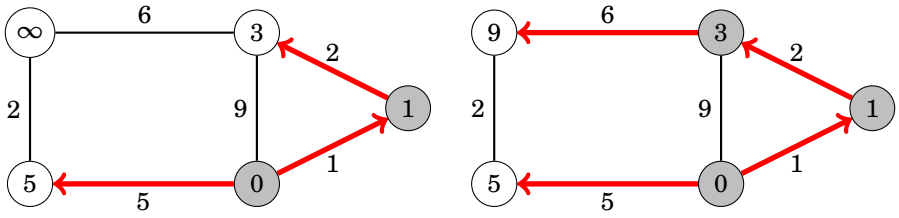
Seuraava kuvasarja esittää Dijsktran algoritmin suoritusta. Solmun sisällä on sen etäisyysarvio ja harmaat solmut ovat käsiteltyjä solmuja. Punaiset nuolet kuvaavat lyhimpiä reittejä lähtösolmusta muihin solmuihin.



1.

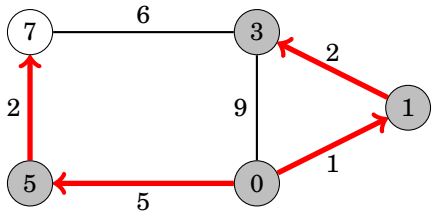


2.

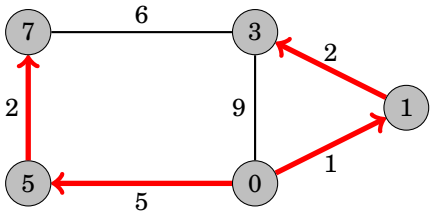


3.

4.



5.



6.

Jotta Dijkstran algoritmin toteutus olisi tehokas, siinä täytyy pystyä etsimään joka askeleella nopeasti solmu, jota ei ole käsitelty ja jossa on mahdollisimman pieni etäisyysarvio. Tähän soveltuu hyvin seuraavaksi esiteltävä kekorakenne.

15.2 Kekorakenne

Keko on tietorakenne, joka toteuttaa seuraavat operaatiot:

- lisää alkio joukkoon ajassa $O(\log n)$
- etsi pienin/suurin alkio joukosta ajassa $O(1)$
- poista pienin/suurin alkio joukosta ajassa $O(\log n)$

Keosta on olemassa kaksi versiota: minimikeko ja maksimikeko. Minimikeossa voi etsiä ja poistaa pienimmän alkion ja maksimikeossa suurimman alkion. Toisin kuin yleisestä joukosta (`set`), keosta ei voi siis etsiä tai poistaa muita kuin pienimmän tai suurimman alkion. Keon etuna on kuitenkin, että se on hyvin kevyt tietorakenne ja toimii selvästi yleistä joukkorakennetta nopeammin.

Keon toteutus perustuu tavallisesti taulukkoon tallennettuun binääripuuhun, jossa solmut ovat järjestyksessä niiden arvojen mukaan. Kekoa ei tarvitse kuitenkaan toteuttaa itse, koska C++ sisältää valmiin keon toteutuksen. C++:n keon toteutuksen nimi on `priority_queue` eli prioriteettijono. Se on oletuksena maksimikeko ja sisältää mm. seuraavat funktiot:

- `push` lisää alkion kekoon
- `top` etsii keon suurimman alkion
- `pop` poistaa keon suurimman alkion

Seuraavaksi nähdään, kuinka Dijkstran algoritmin voi toteuttaa tehokkaasti C++:n prioriteettijonon avulla.

15.3 Toteutus

Oletetaan, että taulukko v sisältää verkon vieruslistat pareina, joiden ensimmäinen alkio on naapurisolmu ja toinen alkio on kaaren paino. Prioriteettijono q sisältää pareja, joiden ensimmäinen alkio on solmun etäisyysarvio *negatiivisena* ja toinen alkio on solmun tunnus. Taulukko t sisältää tiedon, onko solmun etäisyysarvio lopullinen, ja taulukko e sisältää solmujen etäisyysarviot. Lisäksi määritellään vakio INF, joka kuvastaa ääretöntä.

```
vector<pair<int,int>> v[N+1];
priority_queue<pair<int,int>> q;
int t[N+1], e[N+1];
#define INF 999999999
```

Etäisyysarviot tallennetaan prioriteettijonoon negatiivisina, koska C++:n prioriteettijono on oletuksena maksimikeko. Dijkstran algoritmin täytyy löytää kulloinkin pienin etäisyysarvio, joka on sama kuin suurin negatiivinen etäisyysarvio. Algoritmin toteutus on tässä:

```
for (int i = 1; i <= N; i++) e[i] = INF;
e[a] = 0;
q.push(make_pair(0,a));
while (!q.empty()) {
    int x = -q.top().first;
    int s = q.top().second;
    q.pop();
    if (t[s]) continue;
    t[s] = 1;
    for (int i = 0; i < v[s].size(); i++) {
        int u = v[s][i].first;
        if (x+v[s][i].second < e[u]) {
            e[u] = x+v[s][i].second;
            q.push(make_pair(-e[u],u));
        }
    }
}
```

Alussa kaikki muut solmut paitsi aloitussolmu a saavat etäisyysarvion ∞. Tämän jälkeen koodi lisää solmun a prioriteettijonoon ja aloittaa silmukan, joka ottaa joka kierroksella käsittelyyn solmun, jolla on pienin etäisyysarvio prioriteettijonossa. Jos solmun avulla pystyy parantamaan muiden solmujen etäisyysarvioita, algoritmi lisää uudet etäisyysarviot taulukkoon e sekä prioriteettijonoon.

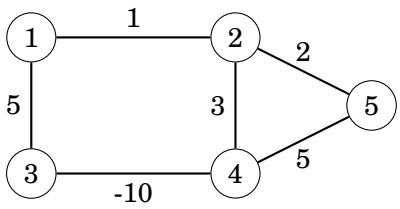
Algoritmin kuluessa samasta solmusta voi olla monta etäisyysarviota prioriteettijonossa, jos solmun etäisyysarvio paranee askeleittain. Tämän vuoksi ennen solmun käsittelyä algoritmi tarkistaa taulukosta t, ettei solmua ole jo käsitelty pienemmällä etäisyysarviolla.

Jos lyhimmän polun solmut täytyy saada selville, algoritmia voi laajentaa samaan tapaan kuin leveyshakua lisäämällä solmuihin tiedon edellisestä solmusta lyhimmällä polulla.

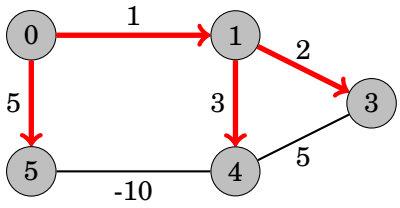
15.4 Analyysi

Dijkstran algoritmi ei ole yhtä suoraviivainen kuin syvyyshaku ja leveyshaku: miksi se toimii ja kuinka nopeasti se toimii?

Tärkeä oletus Dijkstran algoritmissa on, että verkossa ei ole negatiivisia kaaria. Esimerkiksi seuraavassa verkossa Dijkstran algoritmi ei toimi oikein negatiivisen kaaren vuoksi:



Jos haku alkaa solmusta 1, Dijkstran algoritmi tuottaa seuraavat etäisyydet:



Tämä johtuu siitä, että Dijkstran algoritmi käsittelee solmun 3 vasta viimeisenä eikä pysty hyödyntämään enää siitä lähtevää negatiivista kaarta.

Jos verkossa ei ole negatiivisia kaaria, on turvallista valita pienin etäisyysarvio solmun lopulliseksi etäisyydeksi. Solmuun ei voi olla toista lyhyempää polkua muiden solmujen kautta, koska kaikissa muissa käsittelemättömissä solmuissa etäisyysarvio on yhtä suuri tai suurempi. Tämän ominaisuuden ansiosta Dijkstran algoritmi löytää aina lyhimmät polut.

Dijkstran algoritmin aikavaativuus on $O((n+m)\log n)$, jossa n on solmujen määrä ja m on kaarten määrä. Jokainen keko-operaatio vie aikaa $O(\log n)$ ja jokaista solmua ja kaarta kohden tulee korkeintaan yksi lisäys ja poisto kekoon.

Huomaa, että keossa voi olla samaan aikaan $O(m)$ etäisyysarviota, koska samalle solmulle voi olla monta etäisyysarviota useita eri kaaria pitkin. Kuitenkin $O(\log m) = O(\log n)$, koska $m = O(n^2)$ ja $\log(n^2) = 2\log n$.

Luku 16

Bellman-Fordin algoritmi

Bellman-Fordin algoritmi on vaihtoehto Dijkstran algoritmille lyhimpien polkujen etsimiseen lähtösolmusta. Toisin kuin Dijkstran algoritmi, Bellman-Fordin algoritmi toimii myös silloin, kun verkossa on negatiivisia kaaria. Lisäksi algoritmi on helpompi toteuttaa kuin Dijkstran algoritmi. Tämän kääntöpuolena on kuitenkin, että algoritmi on hidas suurilla verkoilla.

16.1 Algoritmi

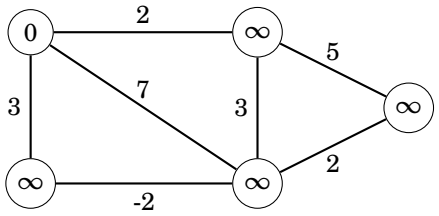
Bellman-Fordin algoritmi pitää yllä solmuille etäisyysarvioita kuten Dijkstran algoritmi. Algoritmin suoritus muodostuu joukosta kierroksia, ja joka kierroksella algoritmi käy läpi kaikki verkon kaaret ja parantaa kaikkia etäisyysarvioita, joita pystyy parantamaan sillä hetkellä. Osoittautuu, että riittävä määrä kierroksille on N, jossa N on verkon solmujen määrä.

Seuraava Bellman-Fordin algoritmin toteutus olettaa, että vektorissa `v` on verkon kaarilista, jossa joka alkion ensimmäinen jäsen on kaaren solmupari ja toinen jäsen on kaaren paino. Taulukossa `e` on solmujen etäisyysarviot ja vakio `INF` kuvastaa äärettömyyttä.

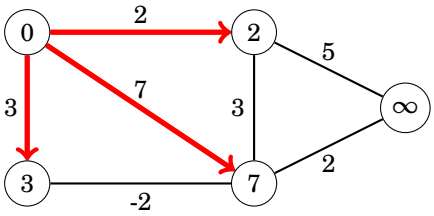
```
vector<pair<pair<int,int>,int>> v;
int e[N+1];
#define INF 999999999
```

```
for (int i = 1; i <= N; i++) e[i] = INF;
e[a] = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 0; j < v.size(); j++) {
        int a = v[j].first.first;
        int b = v[j].first.second;
        int p = v[j].second;
        e[b] = min(e[b], e[a]+p);
    }
}
```

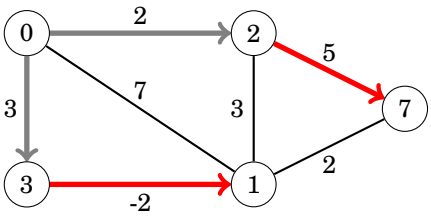
Seuraava kuvasarja esittää Bellman-Fordin algoritmin suoritusta. Punaiset nuolet kuvaavat uusia yhteyksiä, jotka parantavat etäisyysarviota. Harmaat nuolet ovat taas aiemmin muodostettuja yhteyksiä.



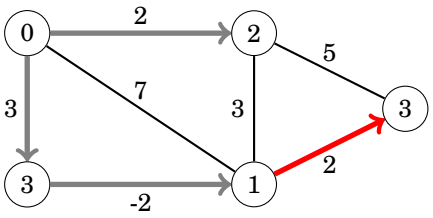
1.



2.



3.



4.

Tämän jälkeen uudet kierrokset eivät enää muuta etäisyysarvioita, joten viimeisessä vaiheessa olevat etäisyysarviot ovat lopulliset etäisyydet. Yleensä lopulliset etäisyydet vakiintuvat nopeammin kuin N kierroksen jälkeen, mutta viimeistään N kierroksen jälkeen haun voi turvallisesti lopettaa.

16.2 Negatiivinen sykli

Erikoistapaus lyhimmän polun etsimisessä on tilanne, jossa verkossa on negatiivinen sykli. Tämä tarkoittaa, että jostain solmusta pääsee takaisin itseensä kulkemalla polkua, jonka kokonaispituus on negatiivinen. Tällöin ei ole mielekästä puhua lyhimmästä polusta, koska polkua voi lyhentää loputtomiin toistamalla negatiivista sykliä.

Bellman-Fordin algoritmi pystyy havaitsemaan negatiivisen syklin. Jos verkossa on negatiivinen sykli, se ilmenee niin, että N kierroksen jälkeen jotain etäisyysarviota pystyy edelleen parantamaan. Niinpä negatiivisen syklin pystyy tunnistamaan seuraavasti Bellman-Fordin algoritmin päätteeksi:

```
for (int j = 0; j < v.size(); j++) {
    int a = v[j].first.first;
    int b = v[j].first.second;
    int p = v[j].second;
    if (e[a]+p < e[b]) cout << "negatiivinen sykli";
}
```

16.3 Analyysi

Miksi Bellman-Fordin algoritmissa riittää käydä N kertaa verkon kaaret läpi? Tämän voi ymmärtää tarkastelemalla mitä tahansa lyhintä polkua lähtösolmusta johonkin muuhun solmuun. Joka kierroksella algoritmi löytää yhden kaaren verran kyseistä polkua eteenpäin. Lisäksi lyhimmässä polussa on varmasti alle N kaarta, koska samaan solmuun ei ole järkeä mennä monta kertaa.

Yllä oleva päättely ei päde silloin, kun verkossa on negatiivinen sykli. Tällöin solmussa käyminen monta kertaa polun aikana lyhentää polkua, minkä vuoksi polku lyhentyy myös N kierroksen jälkeen.

Bellman-Fordin algoritmin aikavaativuus on $O(nm)$, jossa n on solmujen määrä ja m on kaarten määrä. Jos verkko ei ole liian suuri, algoritmi on hyvä valinta Dijkstran sijaan myös silloin, kun verkossa ei voi olla negatiivisia kaaria, koska algoritmin toteuttaminen on Dijkstraa helpompaa.

Luku 17

Floyd-Warshallin algoritmi

Dijkstran algoritmi ja Bellman-Fordin algoritmi etsivät lyhimmät polut tietystä lähtösolmusta muihin solmuihin. Floyd-Warshallin algoritmi etsii sen sijaan yhdellä kertaa lyhimmät polut kaikkien solmujen välillä. Algoritmi toimii myös silloin, kun verkossa on negatiivisia kaaria, ja se on hyvin helppo toteuttaa.

17.1 Algoritmi

Toisin kuin useimmat muut verkkoalgoritmit, Floyd-Warshallin algoritmi olettaa, että verkko on tallennettu vierusmatriisina. Algoritmin tuloksena on etäisyysmatriisi, jonka sisältönä on lyhimmän polun pituus jokaisen verkon solmuparin välillä. Aluksi algoritmi alustaa etäisyysmatriisin verkon vierusmatriisin perusteella. Tämän jälkeen algoritmi käy läpi kaikki verkon solmut ja parantaa etäisyyksiä käyttämällä kutakin solmua välisolmuna.

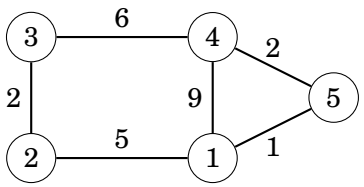
Seuraavassa toteutuksessa taulukko `v` on verkon vierusmatriisi ja taulukko `d` on uusi etäisyysmatriisi. Algoritmin runko muodostuu kolmesta for-silmukasta, jossa uloimman silmukan valitsema solmu k toimii välisolmuna.

```
int v[N+1][N+1];
int d[N+1][N+1];
```

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        if (i == j) d[i][j] = 0;
        else if (v[i][j]) d[i][j] = v[i][j];
        else d[i][j] = INF;
    }
}
for (int k = 1; k <= N; k++) {
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
        }
    }
}
```

17.2 Esimerkki

Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



Algoritmi merkitsee aluksi taulukkoon polun pituudeksi 0 solmusta itseensä ja kaaren pituuden, jos solmujen välillä on kaari. Muiden solmuparien kohdalle polun pituudeksi tulee aluksi ääretön (∞).

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	6	∞
4	9	∞	6	0	2
5	1	∞	∞	2	0

1. kierroksella solmu 1 saa toimia välisolmuna. Tämä mahdollistaa polut solmuparien 2 ja 4 sekä 2 ja 5 välille:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	6	∞
4	9	14	6	0	2
5	1	6	∞	2	0

2. kierroksella myös solmu 2 saa toimia välisolmuna. Tämä mahdollistaa polut solmuparien 1 ja 3 sekä 3 ja 5 välille. Jälkimmäinen polku käyttää välisolmuna myös solmua 1.

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	6	8
4	9	14	6	0	2
5	1	6	8	2	0

3. kierroksella myös solmu 3 saa toimia välisolmuna. Tämä lyhentää polkua solmuparin 2 ja 4 välillä. Aiempi polku kulki solmun 1 kautta ja oli pituudeltaan 14. Uuden polun pituus on vain 8.

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

4. kierroksella myös solmu 4 saa toimia välisolmuna, mutta mikään polku ei lyhene tämän avulla ja taulukon sisältö säilyy ennallaan.

5. kierroksella myös solmu 5 saa toimia välisolmuna, minkä ansiosta solmuparin 1 ja 4 lyhimmäksi poluksi tulee 3. Nyt taulukosta ovat luettavissa lyhimmät polut kaikkien solmuparien välillä.

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

17.3 Analyysi

Floyd-Warshallin algoritmin toiminta perustuu siihen, että jokaisen algoritmin kierroksen jälkeen on saatu laskettua vastaus osaongelmaan ”mitkä ovat lyhyimmät polut kun k ensimmäistä solmua saavat toimia välisolmuina”. Algoritmin päätteeksi kaikki solmut saavat toimia välisolmuina, jolloin etäisyysmatriisi sisältää lopulliset etäisyydet kaikkien solmujen välillä.

Jos verkossa on negatiivinen sykli, jonka osana on solmu s, tämän huomaa siitä, että $\mathrm{d}[s][s]$ on negatiivinen. Toisin sanoen lyhimmän polun pituus solmusta s itseensä on negatiivinen.

Floyd-Warshallin algoritmi muodostuu kolmesta silmukasta, jotka käyvät kaikki verkon solmut läpi, joten algoritmin aikavaativuus on $O(n^3)$. Jos Dijkstran algoritmi suoritetaan jokaiselle lähtösolmulle eli n kertaa, sen aikavaativuudeksi tulee $O(n(n+m)\log n)$. Jos $m = O(n)$ eli verkko on harva, tämä aikavaativuus on $O(n^2 \log n)$, mutta jos $m = O(n^2)$ eli verkko on tiheä, aikavaativuudeksi tulee $O(n^3 \log n)$. Vastaavasti toistamalla n kertaa Bellman-Fordin algoritmia saadaan aikavaativuus $O(n^2 m)$ eli harvalle verkolle $O(n^3)$ ja tiheälle verkolle $O(n^4)$.

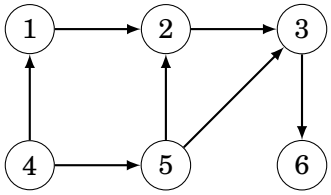
Jos verkko on niin pieni, että $O(n^3)$ ei ole liian paljon, Floyd-Warshallin algoritmi on hyvä valinta silloinkin, kun tehtävänä on etsiä pelkkä kahden solmun välinen polku. Tämä johtuu siitä, että algoritmi on helpompi toteuttaa kuin Dijkstran ja Bellman-Fordin algoritmit.

Luku 18

Topologinen järjestäminen

Suunnatun verkon topologinen järjestys on sellainen lista solmuista, että jos solmusta a on kaari solmuun b, niin a on ennen b:tä listassa. Topologinen järjestys ratkaisee esimerkiksi ongelman, jossa on annettuna joukko työvaiheita ja ehtoja muotoa ”vaihe a täytyy tehdä ennen vaihetta b”. Tulkitsemalla tilanne verkkona, jossa solmut ovat työvaiheita ja kaaret niihin liittyviä ehtoja, topologinen järjestäminen kertoo kaikki ehdot toteuttavan järjestyksen tehdä työvaiheet.

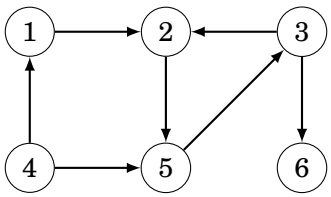
Tarkastellaan esimerkiksi seuraavaa verkkoa:



Yksi tämän verkon topologinen järjestys on 4, 1, 5, 2, 3, 6.

Topologinen järjestys on olemassa silloin, kun verkossa ei ole suunnattua sykliä. Jos verkossa on suunnattu sykli, niin topologista järjestystä ei voi muodostaa, koska mitään syklin solmuista ei voi valita järjestykseen ensimmäisenä.

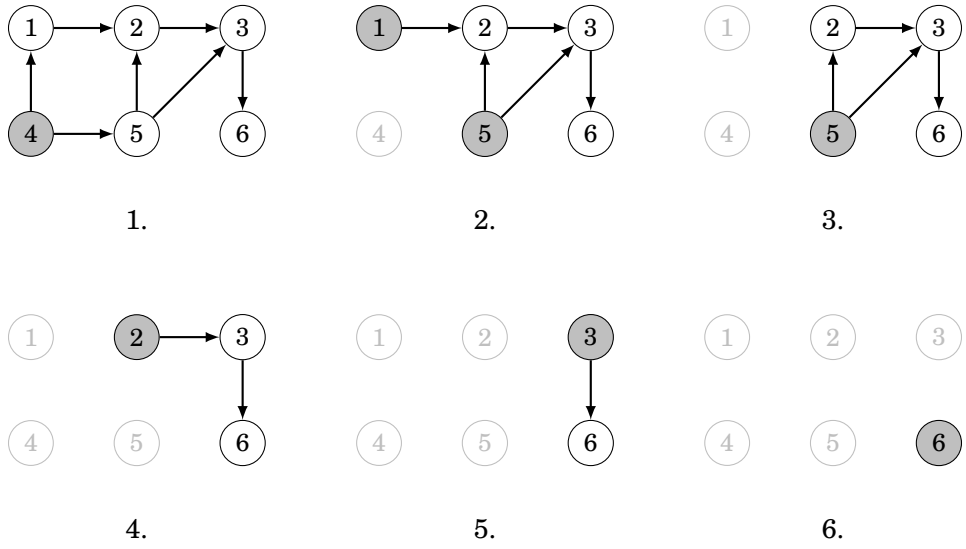
Esimerkiksi seuraavassa verkossa ei ole topologista järjestystä, koska siinä on suunnattu sykli $2 \rightarrow 5 \rightarrow 3 \rightarrow 2$.



18.1 Algoritmi

Topologisen järjestyksen voi muodostaa seuraavalla yksinkertaisella algoritmilla: valitse seuraavaksi solmuksi aina sellainen solmu, johon ei ole kaarta mistään muusta solmusta. Tällainen solmu on aina olemassa, jos verkossa ei ole suunnattua sykliä. Solmun valinnan jälkeen solmu ja kaikki siihen liittyvät kaaret poistetaan verkosta ja seuraava solmu valitaan samalla tavalla.

Esimerkkiverkossa algoritmi toimii näin:



Tuloksena on topologinen järjestys 4, 1, 5, 2, 3, 6.

Huomaa, että joskus algoritmilla on monta mahdollisuutta valita seuraava poistettava solmu. Yllä olevassa verkossa solmun 4 poiston jälkeen solmut 1 ja 5 ovat molemmat ehdokkaita seuraavaksi poistettavaksi solmuksi.

18.2 Toteutus

Topologisen järjestyksen etsiminen tehokkaasti vaatii, että algoritmi pystyy löytämään nopeasti solmun, johon ei tule kaaria muualta. Yksi ratkaisu tähän on pitää yllä jokaiselle solmulle laskuria, jossa on siihen tulevien kaarten määrä. Lisäksi tarvitaan lista solmuista, joihin ei tule kaaria. Jokaisen solmun poiston yhteydessä laskureita päivitetään ja listalle lisätään kaikki uudet solmut, joihin ei tule kaaria poiston jälkeen.

Seuraava toteutus olettaa, että verkko on tallennettu vieruslistoina taulukossa `v`. Taulukko `c` sisältää laskurit solmuihin tuleville kaarille ja vektori `u` on lista solmuista, joihin ei tule kaaria muualta.

```
vector<int> v[N+1];
int c[N+1];
vector<int> u;
```

Algoritmin toteutus on tässä:

```
for (int i = 1; i <= N; i++) {
    for (int j = 0; j < v[i].size(); i++) {
        c[v[i][j]]++;
    }
}
for (int i = 1, i <= N; i++) {
    if (c[i] == 0) u.push_back(i);
}
for (int i = 0; i < N; i++) {
    cout << u[i] << "\n";
    for (int j = 0; j < v[u[i]].size(); j++) {
        int s = v[u[i]][j];
        c[s]--;
        if (c[s] == 0) u.push_back(s);
    }
}
```

Algoritmi alustaa ensin laskurit taulukossa c verkon rakenteen mukaisesti ja lisää listalle u solmut, joihin ei tule kaarta lähtötilanteessa. Tämän jälkeen algoritmi poistaa solmuja yksi kerrallaan ja päivittää laskureita ja listaa.

18.3 Suunnattu sykli

Jos verkossa on suunnattu sykli, topologista järjestystä ei voi muodostaa. Yksi tapa tarkistaa syklin olemassaolo on yrittää muodostaa topologinen järjestys yllä olevalla algoritmilla, mutta keskeyttää muodostus, jos jossain vaiheessa ei pysty valitsemaan solmua, johon ei tule kaaria muualta.

Toinen tapa suunnatun syklin etsimiseen on soveltaa syvyyshakua. Ideana on toteuttaa haku niin, että solmulla on kolme mahdollista tilaa: 0 (ei käsitelty), 1 (käsittely kesken) tai 2 (käsittely valmis). Tila 1 aktivoituu, kun solmuun tullaan ensimmäistä kertaa, ja tilaksi tulee 2, kun kaikki solmusta lähtevät haarat on käsitelty. Verkossa on suunnattu sykli tarkalleen silloin, jos jossain vaiheessa syvyyshakua vastaan tulee tilassa 1 oleva solmu.

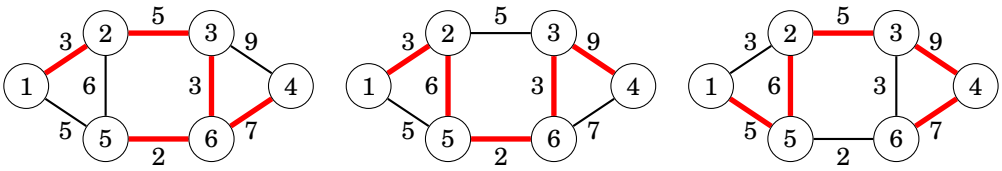
Tässä on muutettu syvyyshaun toteutus:

```
void haku(int s) {
    if (t[s] == 1) {
        cout << "suunnattu sykli löytyi";
        return;
    }
    if (t[s] == 2) return;
    t[s] = 1;
    for (int i = 0; i < v[s].size(); i++) {
        haku(v[s][i]);
    }
    t[s] = 2;
}
```

Luku 19

Virittävät puut

Verkon *virittävä puu* on kokoelma verkon kaaria, jotka kytkevät kaikki solmut toisiinsa. Kuten puut yleensä, virittävä puu on yhtenäinen ja syklitön. Samaan verkkoon voi liittyä monia virittäviä puita. Seuraavassa kuvassa on saman verkon kolme erilaista virittävää puuta.



Virittävän puun paino on siihen kuuluvien kaarten yhteispaino. Yllä vasemman puun paino on 20, keskimmäisen 23 ja oikean 32. Vasen puu on verkon pienin virittävä puu, koska minkään muun virittävän puun paino ei ole pienempi. Vastaavasti oikea puu on verkon suurin virittävä puu.

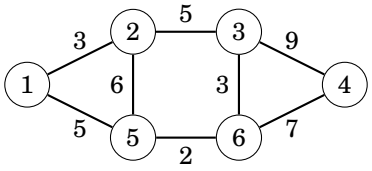
Seuraavaksi esitellään Kruskalin algoritmi, jolla voi etsiä pienimmän tai suurimman virittävän puun verkosta. Algoritmin idea on yksinkertainen, mutta sen tehokas toteutus vaatii uuden tietorakenteen.

19.1 Kruskalin algoritmi

Kruskalin algoritmi muodostaa virittävän puun käymällä läpi verkon kaaret painojärjestyksessä. Jos muodostettavana on pienin virittävä puu, algoritmi käy kaaret läpi keveimmästä raskaimpaan. Vastaavasti jos muodostettavana on suurin virittävä puu, algoritmi käy kaaret läpi raskaimmasta keveimpään. Oletetaan tästä eteenpäin, että muodostettavana on pienin virittävä puu.

Algoritmi pitää yllä tietoa, miten virittävään puuhun valitut kaaret jakavat verkon yhtenäisiin komponentteihin. Aluksi jokainen solmu on omassa komponentissaan. Sitten algoritmi käy kaaret läpi, ja jokaisen kaaren kohdalla algoritmi ottaa kaaren puuhun mukaan, jos kaari yhdistää kaksi sellaista verkon komponenttia, jotka ovat vielä erillisiä.

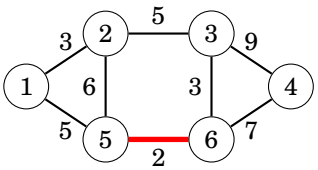
Tarkastellaan Kruskalin algoritmin toimintaa esimerkkiverkolla:



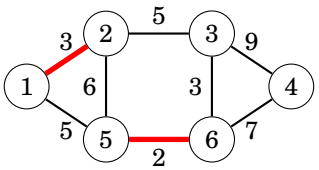
Tässä ovat verkon kaaret painojärjestyksessä:

kaari	paino
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

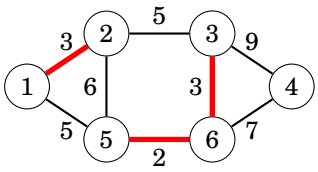
Seuraava kuvasarja esittää pienimmän virittävän puun muodostusta. Tässä tapauksessa kaaret käydään läpi keveimmästä raskaimpaan. Joka vaiheessa punainen kaari tulee mukaan puuhun ja katkoviivainen kaari jää pois.



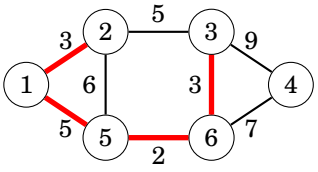
1.



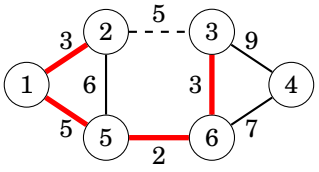
2.



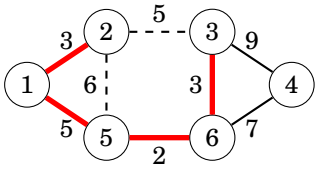
3.



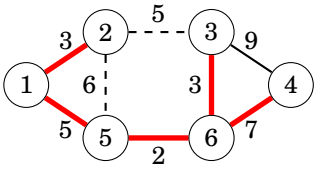
4.



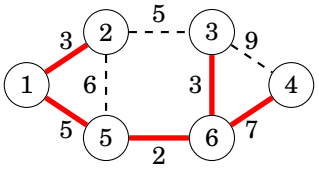
5.



6.



7.



8.

Kruskalin algoritmissa samanpainoiset kaaret voi käsitellä missä tahansa järjestyksessä. Esimerkiksi yllä olevassa tilanteessa kaaren 2–3 voisi käsitellä myös ennen kaarta 1–5. Samanpainoisten kaarten käsittelyjärjestys vaikuttaa lopputuloksena olevaan virittävään puuhun. Tässä tapauksessa pienin virittävä puu on erilainen kuin edellisen sivun esimerkissä.

Tehokas Kruskalin algoritmin toteutus vaatii, että solmuja pystyy yhdistämään komponentteihin tehokkaasti. Tätä varten tarvitsemme uuden tietorakenteen.

19.2 Union-find-rakenne

Union-find-rakenne mahdollistaa tehokkaasti seuraavat operaatiot:

- liitä alkiot a ja b samaan komponenttiin
- tarkista, ovatko alkiot a ja b samassa komponentissa

Ideana on tallentaa jokaisesta alkiosta viittaus toiseen alkioon, joka kuuluu samaan komponenttiin sen kanssa. Jos alkiosta on viittaus itseensä, alkio toimii oman komponenttinsa edustajana. Lisäksi jokaisesta komponentista merkitään muistiin sen edustajaan, montako alkiota kuuluu kyseiseen komponenttiin.

Tarkastellaan esimerkiksi tilannetta, jossa alkiot ovat $1 \ldots 8$ ja jokainen alkio on aluksi erillinen komponentti. Merkitään taulukkoon k viittaus komponentin alkioon ja taulukkoon s komponentin koko. Tilanne on nyt seuraava:

i	1	2	3	4	5	6	7	8
k[i]	1	2	3	4	5	6	7	8
s[i]	1	1	1	1	1	1	1	1

Kun kaksi komponenttia yhdistyvät, niistä kooltaan pienempi liitetään suurempaan. Jos komponentit ovat yhtä suuria, valinnalla ei ole merkitystä. Yhdistetään esimerkiksi alkiot 4 ja 7 samaan komponenttiin:

i	1	2	3	4	5	6	7	8
k[i]	1	2	3	4	5	6	**4**	8
s[i]	1	1	1	**2**	1	1	1	1

Nyt k[7] = 4, joten 7 kuuluu samaan komponenttiin kuin 4. Lisäksi s[4] = 2, joka on komponentin alkioiden määrä. Yhdistetään vielä alkiot 1 ja 2 sekä 2 ja 7 samaan komponenttiin:

i	1	2	3	4	5	6	7	8
k[i]	**2**	2	3	4	5	6	4	8
s[i]	1	**2**	1	2	1	1	1	1

i	1	2	3	4	5	6	7	8
k[i]	2	**4**	3	4	5	6	4	8
s[i]	1	2	1	**4**	1	1	1	1

Huomaa, että lopussa k[2] = 4, koska alkio 4 on komponentin edustaja.

Kulkemalla viittausketjua eteenpäin päästään aina lopulta alkioon, joka on komponentin edustaja. Kaksi alkiota ovat samassa komponentissa, jos molemmista alkioista pääsee samaan edustaja-alkioon.

19.3 Toteutus

Seuraavassa on Kruskalin algoritmin toteutus union-find-rakenteen avulla. Oletetaan, että vektori v on kaarilista, jossa joka alkion ensimmäinen jäsen on kaaren paino ja toinen jäsen on kaaren solmupari. Toisin kuin aiemmin, kaaren paino on ensin, jotta kaarilistan pystyy järjestämään helposti painon mukaan. Lisäksi taulukko k sisältää union-find-rakenteen komponenttien viittaukset ja taulukko s sisältää komponenttien koot.

```
vector<pair<int,pair<int,int>>> v;
int k[N+1], s[N+1];
```

Seuraava koodi alustaa komponentit:

```
for (int i = 1; i <= N; i++) {
    k[i] = i;
    s[i] = 1;
}
```

Funktio sama kertoo, ovatko kaksi solmua samassa komponentissa:

```
bool sama(int a, int b) {
    while (k[a] != a) a = k[a];
    while (k[b] != b) b = k[b];
    return a == b;
}
```

Ideana on seurata kummankin solmun viittauksia loppuun asti, kunnes päästään komponentin edustajasolmuun. Jos molemmista solmuista päädytään samaan edustajaan, solmut ovat samassa komponentissa.

Funktio liita yhdistää kaksi komponenttia toisiinsa:

```
void liita(int a, int b) {
    while (k[a] != a) a = k[a];
    while (k[b] != b) b = k[b];
    if (s[a] > s[b]) {
        k[b] = a;
        s[a] += s[b];
    } else {
        k[a] = b;
        s[b] += s[a];
    }
}
```

Funktio olettaa, että solmut a ja b ovat eri komponenteissa, ja etsii aluksi kummankin komponentin edustajasolmun samaan tapaan kuin aiemmin. Tämän jälkeen funktio päättää taulukon s perusteella, kummin päin komponentit yhdistetään, ja päivittää sitten taulukoita k ja s.

Tämän jälkeen algoritmin voi toteuttaa näin:

```
sort(v.begin(), v.end());
for (int i = 0; i < v.size(); i++) {
    int a = v[i].second.first;
    int b = v[i].second.second;
    if (sama(a, b)) continue;
    liita(a, b);
    cout << "valitaan kaari " << a << "-" << b << endl;
}
```

Vektorissa v parin ensimmäinen jäsen on kaaren paino, joten funktio sort järjestää kaaret automaattisesti pienimmästä suurimpaan niiden painojen mukaan. Tämän jälkeen algoritmi käy läpi kaikki kaaret ja muodostaa niistä pienimmän virittävän puun funktioiden sama ja liita avulla.

19.4 Analyysi

Kruskalin algoritmi toimii sinänsä järkevästi: kun muodostettavana on pienin virittävä puu, on hyvä idea valita mukaan keveitä kaaria. Mutta mistä voi tietää, että tulos on varmasti pienin virittävä puu eikä vain jokin melko pieni puu?

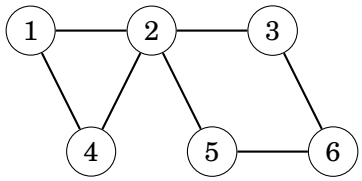
Kruskalin algoritmin toiminta pohjautuu siihen, että jos solmut eivät ole vielä samassa komponentissa, kevein mahdollinen kaari on paras valinta niiden yhdistämiseen. Jos keveintä kaarta ei valittaisi, solmut täytyisi yhdistää samaan komponenttiin joskus myöhemmin käyttäen raskaampaa kaarta. Tästä syystä algoritmi voi valita turvallisesti aina kevyimmän kaaren, jonka yhdistämät solmut ovat vielä eri komponenteissa.

Tässä esitetty union-find-rakenne toteuttaa solmun komponentin etsimisen ja komponenttien yhdistämisen ajassa $O(\log n)$, jossa n on komponenttien määrä. Tämä johtuu siitä, että pienempi komponentti yhdistetään aina suurempaan, jolloin viittausketjun pituus voi olla enintään $O(\log n)$ askelta. On myös mahdollista tehostaa union-find-rakennetta edelleen niin, että operaatioiden aikavaativuus on lähes $O(1)$, mutta tämä ei ole tarpeen kisoissa.

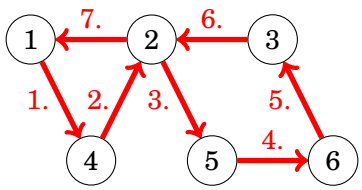
Luku 20

Eulerin kierros

Eulerin kierros on verkossa oleva polku, joka alkaa jostain solmusta, kulkee tasan kerran jokaista kaarta ja palaa lopuksi lähtösolmuun. Verkossa voi olla yksi tai useampi Eulerin kierros, mutta kaikissa verkoissa ei voi muodostaa Eulerin kierrosta. Esimerkiksi verkossa



on seuraava Eulerin kierros solmusta 1 alkaen:



Milloin verkossa on Eulerin kierros? Ensinnäkin kaikkien kaarten täytyy olla samassa verkon komponentissa, jotta yksi polku pystyy kulkemaan jokaista kaarta pitkin. Lisäksi jokaisen solmun asteen täytyy olla parillinen, koska jokainen solmussa käynti Eulerin kierroksen aikana kuluttaa kaksi kaarta. Jos nämä ehdot täyttyvät, niin verkkoon voidaan aina muodostaa Eulerin kierros.

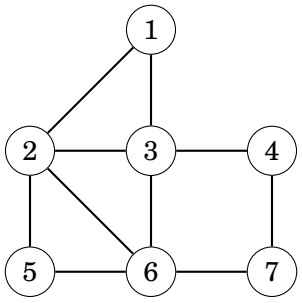
Eulerin kierroksen yleistys on *Eulerin polku*. Eulerin polku kulkee jokaista verkon kaarta tasan kerran Eulerin kierroksen tavoin, mutta polku voi päättyä eri solmuun kuin mistä se alkaa. Jos verkossa on tarkalleen kaksi paritonasteista solmua, siinä on Eulerin polku, jonka päätesolmut ovat kyseiset solmut. Tämän polun voi löytää lisäämällä verkkoon kaaren näiden solmujen välille, etsimällä Eulerin kierroksen ja poistamalla lopuksi siitä lisätyn kaaren.

20.1 Algoritmi

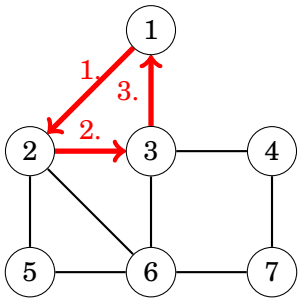
Seuraava algoritmi muodostaa Eulerin kierroksen olettaen, että kaikki verkon kaaret ovat samassa komponentissa ja jokaisen solmun aste on parillinen. Algoritmi valitsee ensin minkä tahansa verkon solmun Eulerin kierroksen aloitussolmuksi ja etsii verkosta jonkin alikierroksen, joka alkaa ja päättyy kyseisessä solmussa. Tämän jälkeen algoritmi laajentaa kierrosta lisäämällä siihen alikierroksia, kunnes kierros kattaa koko verkon.

Alikierroksen etsiminen on helppoa: riittää valita joka askeleella mikä tahansa käyttämätön kaari, kunnes ollaan taas kierroksen alkusolmussa. Haku ei voi joutua umpikujaan, koska jokaisen solmun aste on parillinen eli solmusta pääsee aina jatkamaan eteenpäin, jos alikierros ei ole vielä päättynyt.

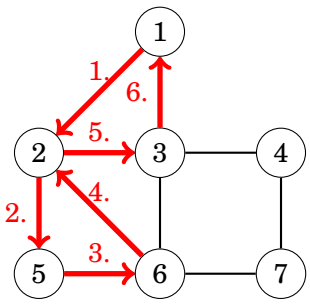
Muodostetaan seuraavaan verkkoon Eulerin kierros solmusta 1 alkaen:



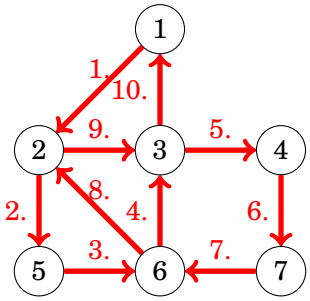
Solmusta 1 syntyy aluksi alikierros $1 \to 2 \to 3 \to 1$:



Solmu 2 on kierroksen ensimmäinen solmu, josta lähtee vielä kaaria kierroksen ulkopuolelle. Siitä syntyy alikierros $2 \to 5 \to 6 \to 2$:



Solmu 6 on kierroksen ensimmäinen solmu, josta lähtee vielä kaaria kierroksen ulkopuolelle. Siitä syntyy alikierros $6 \to 3 \to 4 \to 7 \to 6$:



Nyt kierroksessa ei ole enää solmuja, joista lähtisi kaaria kierroksen ulkopuolelle, joten Eulerin kierros on valmis.

20.2 Toteutus

Seuraava toteutus olettaa, että taulukko v sisältää verkon jokaisesta solmusta sen naapurisolmut joukkona. Tämä rakenne helpottaa Eulerin kierroksen muodostamista, koska verkosta täytyy sekä etsiä että poistaa kaaria.

```
set<int> v[N+1];
```

Algoritmin perustana on rekursiivinen funktio `kierros`, joka muodostaa alikierroksia solmusta x alkaen. Funktio tulostaa kierrokseen tulevat solmut ensimmäistä solmua lukuun ottamatta.

```
void kierros(int x) {
    vector<int> k;
    int s = x;
    do {
        int u = *(v[s].begin());
        v[s].erase(u);
        v[u].erase(s);
        s = u;
        k.push_back(s);
    } while (s != x);
    for (int s : k) {
        cout << s << "\n";
        if (v[s].size() > 0) kierros(s);
    }
}
```

Kierroksen voi aloittaa esimerkiksi solmusta 1:

```
cout << "1\n";
kierros(1);
```

Osa III

Dynaaminen ohjelmointi

Luku 21

Optimiratkaisu

Dynaaminen ohjelmointi on monikäyttöinen tekniikka, jossa on ideana jakaa ongelma osaongelmiksi, jotka voi ratkaista tehokkaasti toistensa avulla. Dynaamisen ohjelmoinnin tavalliset käyttötarkoitukset ovat optimiratkaisun etsiminen sekä kaikkien ratkaisuiden määrän laskeminen. Aloitamme dynaamiseen ohjelmointiin tutustumisen optimiratkaisun etsimisestä.

Tämän luvun aiheena on seuraava tehtävä:

Tehtävä: Kolikot

Bittimaassa on käytössä n kolikkoa, joiden arvot on annettu taulukossa a pienimmästä suurimpaan. Tehtävänä on muodostaa kolikoilla rahamäärä m. Kaikki arvot ovat kokonaislukuja, ja yksi kolikoista on aina 1. Mikä on pienin määrä kolikkoja, joilla rahamäärän voi muodostaa? Esimerkiksi jos kolikot ovat $\{1,2,5\}$ ja rahamäärä on 9, tarvitaan ainakin 3 kolikkoa: $2+2+5=9$.

21.1 Peruuttava haku

Suoraviivainen tapa ratkaista tehtävä on käyttää peruuttavaa hakua, joka käy läpi kaikki mahdollisuudet:

```
void haku(int x, int k) {
    if (x == 0) {
        p = min(p, k);
        return;
    }
    for (int i = 0; i < n; i++) {
        if (x-a[i] >= 0) haku(x-a[i], k+1);
    }
}
```

Haku käynnistyy seuraavasti:

```
haku(m, 0);
```

Funktion parametri x on rahamäärä, josta täytyy päästä eroon, ja parametri k on tähän mennessä käytettyjen kolikkojen määrä. Haku käy läpi joka tilanteessa kaikki mahdollisuudet valita seuraavaksi poistettava kolikko. Muuttuja p sisältää pienimmän määrän kolikkoja, joilla rahamäärän voi muodostaa.

Tämä on toimiva ratkaisu mutta hidas suurissa tapauksissa, koska funktio haarautuu joka vaiheessa n osaan.

21.2 Ahne algoritmi

Toinen lähestymistapa tehtävään on *ahne algoritmi*: valitaan joka vaiheessa suurin mahdollinen kolikko. Nyt ratkaisusta tulee seuraava:

```
int p = 0;
for (int i = n-1; i >= 0; i--) {
    while (a[i] <= m) {
        m -= a[i];
        p++;
    }
}
```

Koodin voi toteuttaa myös näin:

```
int p = 0;
for (int i = n-1; i >= 0; i--) {
    p += m/a[i];
    m %= a[i];
}
```

Tämä on erittäin tehokas algoritmi – mutta toimiiko se? Ahneessa algoritmissa on aina vaikea tietää, tuottavatko valinnat parasta lopputulosta.

Tämä algoritmi *ei* toimi, koska on helppoa keksiä vastaesimerkki, jossa algoritmi valitsee liikaa kolikkoja. Esimerkiksi jos kolikot ovat $\{1,4,5\}$ ja rahamäärä on 8, algoritmi muodostaa ratkaisun $5+1+1+1$, vaikka paras ratkaisu olisi $4+4$.

21.3 Dynaaminen ohjelmointi

Dynaaminen ohjelmointi yhdistää peruuttavan haun toimivuuden ja ahneen algoritmin nopeuden. Tässä tehtävässä ongelman voi jakaa osaongelmiin, jotka ovat muotoa ”mikä on pienin kolikkomäärä rahamäärän x muodostamiseen?”.

Oletetaan, että kolikot ovat $\{1,4,5\}$ ja haluamme muodostaa rahamäärän 100. On kolme vaihtoehtoa valita ensimmäinen kolikko: 1, 4 tai 5. Jos valitsemme kolikon

1, sen jälkeen täytyy muodostaa vielä rahamäärä 99. Vastaavasti kolikot 4 ja 5 jättävät jäljelle rahamäärät 96 ja 95. Näin on saatu kolme pienempää osaongelmaa: rahamäärien 99, 96 ja 95 muodostaminen.

Ideana on tallentaa vastaukset kaikkiin osaongelmiin taulukkoon, jolloin ne ovat nopeasti saatavilla tarvittaessa. Tässä tehtävässä rahamäärän m jakamiseen liittyy $m+1$ osaongelmaa: jokainen rahamäärä välillä $0 \ldots m$.

Seuraavassa koodissa taulukko `d` sisältää vastaukset osaongelmiin. Kohdassa `d`[x] lukee, montako kolikkoa tarvitaan vähintään rahamäärän x muodostamiseen. Koodin päätteeksi kohdassa `d`[m] on tehtävän ratkaisu.

```
d[0] = 0;
for (int i = 1; i <= m; i++) {
    d[i] = m+1;
    for (int j = 0; j < n; j++) {
        if (i-a[j] < 0) continue;
        d[i] = min(d[i], d[i-a[j]]+1);
    }
}
cout << d[m] << "\n";
```

Koodi laskee osaongelmien ratkaisut pienimmästä suurimpaan. Jokaisen osaongelman kohdalla koodi käy läpi kaikki vaihtoehdot valita yksi kolikko siihen. Ratkaisut pienempiin osaongelmiin ovat saatavilla suoraan taulukossa `d`.

Esimerkiksi kun kolikot ovat $\{1,4,5\}$ ja rahamäärä on 15, taulukosta tulee:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d[i]	0	1	2	3	1	1	2	3	2	2	2	3	3	3	3	3

Taulukosta nähdään esimerkiksi, että rahamäärän 8 muodostamiseen tarvitaan 2 kolikkoa ja rahamäärän 11 muodostamiseen tarvitaan 3 kolikkoa.

21.4 Ratkaisun muodostus

Seuraava askel on laajentaa dynaamista ohjelmointia niin, että voimme selvittää myös valittavat kolikot optimaalisessa ratkaisussa. Nykyisellään taulukko `d` kertoo, että rahamäärän 11 voi muodostaa 3 kolikolla, mutta mitkä kolikot ovat tarkkaan ottaen tässä tapauksessa?

Ideana on tehdä uusi taulukko `e`, joka kertoo jokaisesta osaongelmasta, mikä kolikko siitä tulee poistaa optimaalisessa ratkaisussa. Tämän taulukon avulla algoritmin päätteeksi pystyy selvittämään kaikki valittavat kolikot. Taulukon saa liitettyä äskeiseen koodiin korvaamalla rivin

```
        d[i] = min(d[i], d[i-a[j]]+1);
```

riveillä

```
        if (d[i-a[j]]+1 < d[i]) {
            d[i] = d[i-a[j]]+1;
            e[i] = i-a[j];
        }
```

Tämän jälkeen tarvittavat kolikot saa listattua näin algoritmin päätteeksi:

```
while (m > 0) {
    cout << e[m] << "\n";
    m -= e[m];
}
```

Oletetaan jälleen, että kolikot ovat $\{1,4,5\}$ ja muodostettava rahamäärä on 15. Nyt taulukot ovat seuraavat:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d[i]	0	1	2	3	1	1	2	3	2	2	2	3	3	3	3	3
e[i]	0	1	1	1	4	5	1	1	4	4	5	1	4	4	4	5

Esimerkiksi rahamäärän 11 muodostamisessa ensimmäinen askel on valita kolikko 1. Nyt jäljellä on rahamäärä 10, jossa valitaan kolikko 5. Lopuksi jäljellä on rahamäärä 5, jossa valitaan myös kolikko 5.

Luku 22

Ratkaisumäärä

Edellisessä luvussa tehtävänä oli etsiä pienin määrä kolikoita, jotka tuottavat halutun rahamäärän. Dynaamisen ohjelmoinnin toinen käyttötarkoitus on ratkaisujen yhteismäärän laskeminen, mihin tutustumme seuraavaksi. Muutetaan nyt aiempaa tehtävää seuraavasti:

Kuinka monta erilaista tapaa on muodostaa rahamäärä m?

Esimerkiksi jos kolikot ovat $\{1,2,5\}$ ja rahamäärä on 6, ratkaisuja on 15:

- $1+1+1+1+1+1=6$
- $1+1+1+1+2=6$
- $1+1+1+2+1=6$
- $1+1+2+1+1=6$
- $1+2+1+1+1=6$
- $2+1+1+1+1=6$
- $1+1+2+2=6$
- $1+2+1+2=6$
- $1+2+2+1=6$
- $2+1+1+2=6$
- $2+1+2+1=6$
- $2+2+1+1=6$
- $2+2+2=6$
- $1+5=6$
- $5+1=6$

22.1 Algoritmi

Seuraavassa koodissa d[k] sisältää ratkaisun osaongelmaan ”monellako tavalla kolikoista voi muodostaa rahamäärän k”:

```
d[0] = 1;
for (int i = 1; i <= m; i++) {
    for (int j = 0; j < n; j++) {
        if (i-a[j] < 0) continue;
        d[i] += d[i-a[j]];
    }
}
cout << d[m] << "\n";
```

Koodi perustuu samaan ideaan kuin edellisen luvun koodi: jokaisen rahamäärän kohdalla käydään läpi kaikki eri tavat valita viimeinen kolikko siihen. Jos kolikot ovat $\{1,2,5\}$ ja rahamäärä 12, taulukosta d tulee:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
d[i]	1	1	2	3	5	9	15	26	44	75	128	218	372

Esimerkiksi d[6] saadaan laskemalla d[1]+d[4]+d[5]. Tässä d[1] vastaa tapausta, jossa viimeinen kolikko on 5, ja d[4] ja d[5] vastaavat kolikoita 2 ja 1.

22.2 Vastaus modulona

Kaikkien ratkaisujen määrä on yleensä suuri, minkä vuoksi monissa tehtävissä vastaus täytyy antaa modulona. Tällöin käytännössä koodin jokaisen laskutoimituksen perään täytyy lisätä modulon laskeminen.

Muutetaan äskeistä koodia niin, että se laskee vastauksen modulo 10^9+7:

```
#define M 1000000007
```

```
d[0] = 1;
for (int i = 1; i <= m; i++) {
    for (int j = 0; j < n; j++) {
        if (i-a[j] < 0) continue;
        d[i] += d[i-a[j]];
        d[i] %= M;
    }
}
cout << d[m] << "\n";
```

Kun modulon laskee joka välissä, vastaus ei kasva koskaan niin suureksi, ettei se mahtuisi tavalliseen lukumuuttujaan.

22.3 Erilaiset ratkaisut

Tarkastellaan sitten tehtävän vaikeampaa muunnelmaa, jossa tehtävänä on laskea *erilaisten* ratkaisujen määrä. Esimerkiksi $2+2+1+1=6$ ja $1+2+1+2=6$ ovat käytännössä sama ratkaisu, vain kolikoiden järjestys eroaa.

Nyt kolikoilla $\{1,2,5\}$ voi muodostaa rahamäärän 6 vain 5 tavalla:

- $1+1+1+1+1+1=6$
- $1+1+1+1+2=6$
- $1+1+2+2=6$
- $2+2+2=6$
- $1+5=6$

Hyvä strategia tällaiseen tehtävään on laskea *järjestettyjen* ratkaisujen määrä eli vaatia, että pienempi kolikko on aina ennen suurempaa kolikkoa. Tämä vaatii kaksiulotteisen taulukon käyttämistä dynaamisessa ohjelmoinnissa.

Nyt sopiva osaongelma on ”montako tapaa on muodostaa rahamäärä i käyttäen j pienintä kolikkoa”. Seuraavassa koodissa d[i][j] vastaa tähän kysymykseen:

```
d[0][0] = 1;
for (int i = 1; i <= m; i++) {
    d[i][0] = 0;
    for (int j = 1; j <= n; j++) {
        d[i][j] += d[i][j-1];
        if (i-a[j-1] < 0) continue;
        d[i][j] += d[i-a[j-1]][j];
    }
}
cout << d[m][n] << "\n";
```

Osaongelman d[i][j] ratkaisu muodostuu seuraavasti: Jos $j = 0$ eli kolikoita ei saa käyttää, d[i][j] = 1, jos $i = 0$, ja muuten d[i][j] = 0. Jos taas $j > 0$, ratkaisu muodostuu laskemalla yhteen tapaukset d[i][$j-1$] ja d[$i-$a[$j-1$]][j]. Ensimmäinen tapaus tarkoittaa, että suurinta mahdollista kolikkoa ei käytetä vaan siirrytään pienempiin. Toinen tapaus taas tarkoittaa, että suurin kolikko valitaan ja sitä voidaan käyttää vielä myöhemmin.

Kun kolikot ovat {1,2,5} ja rahamäärä 12, taulukosta tulee:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4	5	5	6	6	7
3	1	1	2	2	3	4	5	6	7	8	10	11	13

Esimerkiksi tapaus d[6][2] tarkoittaa, että muodostettavana on rahamäärä 6 ja saamme käyttää kahta pienintä kolikkoa eli kolikoita 1 ja 2. Jos emme käytä kolikkoa 2, uuden osaongelman vastaus on kohdassa d[6][1]. Jos taas käytämme kolikon 2, vastaava osaongelma on d[4][2]. Laskemalla yhteen nämä tapaukset saadaan tulokseksi $1 + 3 = 4$.

22.4 Muistin säästäminen

Dynaamisen ohjelmoinnin muistinkulutusta voi usein vähentää pitämällä muistissa vain myöhemmin tarvittavia osaongelmien ratkaisuja. Joskus tämä vaatii algoritmin laskentajärjestyksen muuttamista. Muokataan seuraavaksi edellistä koodia niin, että $m \times n$ -kokoisen taulukon sijasta riittää $m \times 2$ -taulukko.

Tärkeä havainto on, että arvon d[i][j] laskemiseen riittää, että tiedossa on arvo d[i][$j-1$]. Muistissa ei siis tarvitse säilyttää ratkaisuja kaikille muille pienemmille j:n arvoille. Tämä edellyttää kuitenkin sitä, että koodissa ulompi silmukka käy läpi j:n arvoja ja sisempi silmukka käy läpi i:n arvoja.

Seuraava koodi käyttää taulukkoa d niin, että d[i][$j\%2$] sisältää ratkaisun rahamäärälle i käyttäen j pienintä kolikkoa. Taulukko muistaa siis vain kolikkomäärän *parillisuuden*. Koodi on muuten samanlainen kuin ennenkin, mutta silmukat ovat toisinpäin ja taulukkoa indeksoidaan j:n sijasta arvolla $j\%2$.

```
d[0][0] = 1;
for (int i = 1; i <= m; i++) d[i][0] = 0;
for (int j = 1; j <= n; j++) {
    for (int i = 1; i <= m; i++) {
        d[i][j%2] += d[i][(j-1)%2];
        if (i-a[j-1] < 0) continue;
        d[i][j%2] += d[i-a[j-1]][j%2];
    }
}
cout << d[m][n] << "\n";
```

Muistin säästäminen on joskus tarpeen kisoissa, jos tehtävässä on tiukka muistiraja. Yleensä hyvä ratkaisu on kuitenkin varata taulukko kaikille osaongelmille, jos se ei vie liikaa muistia.

Luku 23

Ruudukot

Tässä luvussa on esimerkkejä dynaamisen ohjelmoinnin soveltamisesta ruudukkojen käsittelyssä. Yleensä osaongelmana on laskea vastaus tilanteeseen, jonka kulmapisteenä on tietty ruudukon ruutu. Tyypillinen tehtävä on muodostaa ruudukon vasemmasta yläkulmasta lähteviä reittejä, joissa saa liikkua oikealle ja alaspäin. Tällöin osaongelmana on kulkea reittiä tiettyyn ruutuun asti.

23.1 Paras reitti

Ensimmäinen tehtävämme on seuraava:

Tehtävä: Paras reitti

Annettuna on $n \times m$ -ruudukko, jonka jokaisessa ruudussa on kokonaisluku. Tehtävänä on etsiä reitti ruudukon vasemmasta yläkulmasta oikeaan alakulmaan, jossa lukujen summa on mahdollisimman suuri. Saat liikkua joka askeleella yhden ruudun verran oikealle tai alaspäin.

Seuraavassa ruudukossa paras reitti on merkitty sinisellä taustalla:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Tällä reitillä lukujen summa on $3+9+8+7+9+8+5+10+8 = 67$, joka on suurin mahdollinen summa vasemmasta yläkulmasta oikeaan alakulmaan.

Tämä on kaksiulotteinen tilanne, joten myös osaongelmien tulee olla kaksiulotteisia. Nyt hyvä valinta osaongelmaksi on ”suurin summa reitillä vasemmasta yläkulmasta ruutuun (y,x)”. Reitti ruutuun tulee varmasti joko vasemmalta tai

ylhäältä, joten osaongelman vastauksen saa laskettua tehokkaasti, kunhan tiedossa on vastaukset osaongelmiin $(y-1,x)$ ja $(y,x-1)$. Poikkeustapaukset ovat vasen reuna ja yläreuna, joissa reitti voi tulla vain yhdestä suunnasta.

Seuraava koodi olettaa, että ruudukon sisältö on annettu taulukossa `t`. Koodi muodostaa dynaamisen ohjelmoinnin taulukon `d`, jossa `d`[y][x] on suurin summa reitillä ruudukon vasemmasta yläkulmasta ruutuun (y,x):

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        int s1 = 0, s2 = 0;
        if (i > 0) s1 = d[i-1][j];
        if (j > 0) s2 = d[i][j-1];
        d[i][j] = max(s1,s2)+t[i][j];
    }
}
cout << d[n-1][m-1] << "\n";
```

Ruudun (y,x) käsittelyssä muuttujiin `s1` ja `s2` haetaan suurimmat summat reiteillä ruudukon vasemmasta yläkulmasta ruutuihin $(y-1,x)$ ja $(y,x-1)$. Jos $y=0$ eli ruutu on yläreunassa, `s1` jää nollaksi, ja jos $x=0$ eli ruutu on vasemmassa reunassa, `s2` jää nollaksi. Arvoista `s1` ja `s2` suurempi toimii uuden summan pohjana, ja siihen lisätään ruudussa (y,x) oleva luku.

Jos tehtävänä on lisäksi selvittää suurinta summaa vastaava reitti, tämä onnistuu samaan tapaan kuin rahamäärän muodostamisessa lisäämällä koodiin taulukon, joka muistaa jokaisessa ruudussa reitin edellisen ruudun.

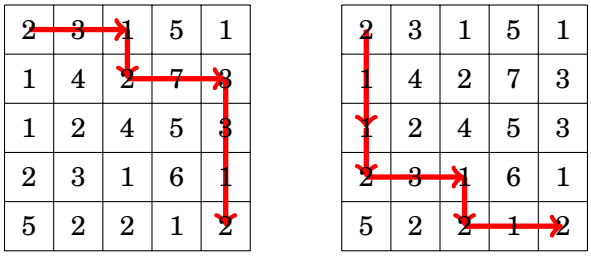
23.2 Hyppyreitti

Tarkastellaan sitten seuraavaa tehtävää:

Tehtävä: Hyppyreitti

Annettuna on $n \times m$ -ruudukko, jonka jokaisessa ruudussa on kokonaisluku. Reitti alkaa ruudukon vasemmasta ylänurkasta, ja joka askeleella saat hypätä ruudussa olevan luvun verran oikealle tai alaspäin. Kuinka monta erilaista reittiä ruudukon oikeaan alakulmaan on olemassa? Jos hyppy vie ruudukon ulkopuolelle, et voi käyttää sitä reitissä.

Tässä on esimerkiksi kaksi mahdollista reittiä ruudukossa:



Nyt osaongelma on ”montako reittiä on vasemmasta yläkulmasta ruutuun (y,x)”. Erona edelliseen tehtävään reitissä on hyppyjä, joten osaongelman ratkaisemiseksi ei riitä tarkastella edellistä ruutua vasemmalla ja ylhäällä. Yksi tapa olisi käydä läpi kaikki vasemmalla ja ylhäällä olevat ruudut, mutta suuressa ruudukossa tästä tulisi paljon lisää työtä. Osaongelmat voi ratkaista kuitenkin tehokkaasti toteuttamalla laskennan käänteisesti.

Seuraavassa koodissa on ideana lähettää joka ruudussa tietoa tuleville osaongelmille. Joka ruudussa hyppy voi suuntautua oikealle tai alaspäin, joten ruutu kasvattaa kahden myöhemmin käsiteltävän ruudun reittimääriä.

```
d[0][0] = 1;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (i+t[i][j] < n) d[i+t[i][j]][j] += d[i][j];
        if (j+t[i][j] < m) d[i][j+t[i][j]] += d[i][j];
    }
}
cout << d[n-1][m-1] << "\n";
```

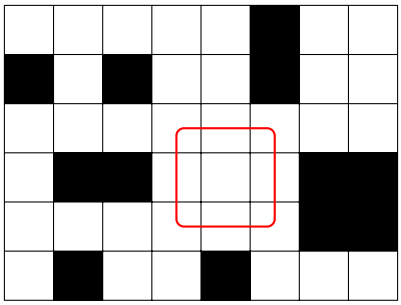
Koodi merkitsee aluksi taulukon `d` vasempaan yläkulmaan, että kyseiseen ruutuun pääsee yhdellä tavalla. Tämän jälkeen koodi käy ruudukon läpi ja välittää tietoa reittien määrästä ruudukon halki. Koodi toteuttaa vain sellaiset hypyt, jotka eivät johda ruudukon ulkopuolelle. Lopuksi kysytty reittien kokonaismäärä on taulukon `d` oikeassa alakulmassa.

23.3 Suurin neliö

Tarkastellaan lopuksi toisenlaista tehtävää:

Tehtävä: Suurin neliö

Annettuna on $n \times m$ -ruudukko, jonka jokainen ruutu on musta tai valkoinen. Tehtävänä on etsiä ruudukosta mahdollisimman suuri neliö, jonka jokainen ruutu on valkoinen. Esimerkiksi seuraavassa ruudukossa suurin neliö on kooltaan 3×3, ja se on merkitty ruudukkoon punaisella.



Tämänkin tehtävän pystyy palauttamaan osaongelmiin, joissa lasketaan tulos tiettyyn ruutuun asti. Nyt sopiva osaongelma on ”kuinka suuri on suurin neliö,

jonka oikea alakulma on ruudussa (y,x)". Koko tehtävän ratkaisu on näistä arvoista suurin koko ruudukon alueella.

Jos ruutu on musta, osaongelman ratkaisu on selvästi 0. Muussa tapauksessa osaongelman pystyy ratkaisemaan käyttäen apuna kolmea muuta osaongelmaa: $(y-1,x)$, $(y,x-1)$ ja $(y-1,x-1)$. Nämä vastaavat suurimpia neliöitä, jotka päättyvät ruudusta (y,x) yhden ruudun ylöspäin, vasemmalle ja ylävasemmalle.

Seuraava koodi olettaa, että taulukko `t` sisältää ruutujen värit (0 = valkoinen, 1 = musta). Koodi muodostaa taulukon `d`, jossa `d`[y][x] vastaa suurinta neliötä, jonka oikea alakulma on ruudussa (y,x).

```
int t = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (t[i][j] == 1) {
            d[i][j] = 0;
        } else {
            int n1 = 0, n2 = 0, n3 = 0;
            if (i > 0) n1 = d[i-1][j];
            if (j > 0) n2 = d[i][j-1];
            if (i > 0 && j > 0) n3 = d[i-1][j-1];
            d[i][j] = min(min(n1,n2),n3)+1;
            t = max(t, d[i][j]);
        }
    }
}
cout << t << "\n";
```

Jos ruutu (y,x) on tyhjä, koodi hakee muuttujiin `n1`, `n2` ja `n3` suurimman neliön ruuduissa $(y-1,x)$, $(y,x-1)$ ja $(y-1,x-1)$. Näistä arvoista pienin määrittää, kuinka suuri neliö ruudussa (y,x) voi olla.

Esimerkkitapauksessa taulukon sisällöksi tulee:

1	1	1	1	1	■	1	1
■	1	■	1	2	■	1	2
1	1	1	1	2	1	1	2
1	■	■	1	2	2	■	■
1	1	1	1	2	3	■	■
1	■	1	2	■	1	1	1

Luku 24

Merkkijonot

Moni dynaamisen ohjelmoinnin tehtävä liittyy merkkijonoihin. Usein tehtävänä on laskea, montako tietyt ehdot toteuttavaa merkkijonoa on olemassa. Tämä luku esittelee kolme merkkijonotehtävää, jotka pystyy ratkaisemaan dynaamisella ohjelmoinnilla. Tehtävistä viimeinen on selvästi haastavampi kuin aiemmat dynaamisen ohjelmoinnin esimerkit tässä kirjassa.

24.1 Porrassanat

Tehtävä: Porrassanat

Tarkastellaan merkeistä `A...Z` muodostuvia sanoja. Merkkijono on porrassana, jos joka kohdassa vierekkäin olevat merkit ovat vierekkäin aakkosissa. Esimerkiksi merkkijono `ABCBCDEF` on porrassana. Tehtävänä on laskea, montako n merkin pituista porrassanaa on olemassa.

Ratkaisu

Tässä tehtävässä luonteva osaongelma on ”laske k merkin pituiset porrassanat, joiden viimeinen merkki on x”. Esimerkiksi 8-merkkinen `D`:hen päättyvä porrassana on joko muotoa `??????CD` tai `??????ED`, jossa `?` on mikä tahansa merkki. Näin ollen tähän liittyvät osaongelmat ovat laskea, montako 7-merkkistä `C`:hen ja `E`:hen päättyvää porrassanaa on olemassa.

Seuraava koodi laskee taulukkoon `d`[k][x], montako k-merkkistä x:ään päättyvää porrassanaa on olemassa.

```
for (int j = 1; j <= 26; j++) d[1][j] = 1;
for (int i = 2; i <= n; i++) {
    for (int j = 1; j <= 26; j++) {
        if (j > 1) d[i][j] += d[i-1][j-1];
        if (j < 26) d[i][j] += d[i-1][j+1];
    }
}
```

Tämän jälkeen porrassanojen kokonaismäärän saa laskettua käymällä läpi kaikki mahdollisuudet valita viimeinen merkki:

```
int t = 0;
for (int j = 1; j <= 26; j++) t += d[n][j];
cout << t << "\n";
```

24.2 DNA-ketjut

Tehtävä: DNA-ketjut

DNA-ketju on merkkijono, joka muodostuu merkeistä A, C, G ja T. Tehtäväsi on laskea n merkin pituisten ketjujen määrä, joissa on samaa merkkiä enintään m kertaa peräkkäin. Esimerkiksi jos $n = 8$ ja $m = 3$, ketju ACGGGATG on kelvollinen, mutta ketju ACGGGGTG ei ole kelvollinen.

Ratkaisu

Tässä tehtävässä osaongelma on melko mutkikas: ”laske k-merkkiset ketjut, joissa viimeinen merkki on x, mikään merkki ei ole toistunut yli m kertaa ja viimeinen merkki on toistunut c kertaa tähän mennessä”. Parametri m on syötteenä annettu vakio, mutta kaikki muut parametrit tarvitsevat oman ulottuvuuden, joten dynaamisen ohjelmoinnin taulukko on kolmiulotteinen.

Esimerkiksi jos $k = 8$, $x =$ T ja $c = 3$, tämä vastaa ketjuja muotoa ?????TTT, missä ? on tuntematon merkki. Tämä palautuu tapaukseen $k = 7$, $x =$ T ja $c = 2$ eli ketjuihin muotoa ?????TT. Erilainen tapaus on $k = 8$, $x =$ T ja $c = 1$ eli ???????T, missä viimeinen merkki esiintyy vain kerran. Tämä palautuu mihin tahansa 7-merkkiseen ketjuun, jonka viimeinen merkki ei ole T.

Seuraava koodi muodostaa taulukon d, jossa d[k][x][c] sisältää osaongelman ratkaisun. Merkit on koodattu A = 0, C = 1, G = 2 ja T = 3.

```
for (int j = 0; j < 4; j++) d[1][j][1] = 1;
for (int i = 2; i <= n; i++) {
    for (int j = 0; j < 4; j++) {
        for (int a = 0; a < 4; a++) {
            if (j == a) continue;
            for (int b = 1; b <= m; b++) {
                d[i][j][1] += d[i-1][a][b];
            }
        }
        for (int k = 2; k <= m; k++) {
            d[i][j][k] += d[i-1][j][k-1];
        }
    }
}
```

Koodi käsittelee ensin tapauksen, jossa viimeistä merkkiä esiintyy vain kerran, jolloin ratkaisu rakentuu monista osaongelmista. Muuttuja a valitsee edellisen merkin ja b kyseisen merkin toistomäärän. Muussa tapauksessa viimeinen merkki on esiintynyt useamman kerran, jolloin osaongelman ratkaisun saa suoraan toisesta osaongelmasta.

Lopuksi tehtävän vastauksen saa laskettua näin:

```
int t = 0;
for (int j = 0; j < 4; j++) {
    for (int k = 1; k <= m; k++) {
        t += d[n][j][k];
    }
}
cout << t << "\n";
```

24.3 Alijonot

Tehtävästä on välillä vaikeaa nähdä päältä päin, että siinä voi soveltaa dynaamista ohjelmointia. Seuraavassa on yksi esimerkki tällaisesta tehtävästä. Tehtävä vaikuttaa aluksi laskennallisesti haastavalta, mutta siihen on olemassa lyhyt dynaamisen ohjelmoinnin ratkaisu.

Tehtävä: Alijonot

Merkkijonon *alijono* saadaan poistamalla merkkijonosta merkkejä ja säilyttämällä muiden merkkien järjestys ennallaan. Tehtävänä on laskea annetun merkkijonon erilaisten alijonojen määrä. Merkkijono muodostuu kirjaimista A–Z.

Esimerkiksi merkkijonon HAAPA erilaiset alijonot ovat A, AA, AAA, AAP, AAPA, AP, APA, H, HA, HAA, HAAA, HAAP, HAAPA, HAP, HAPA, HP, HPA, P ja PA. Yhteensä erilaisia alijonoja on siis 19.

Ratkaisu

Ideana on laskea jokaiselle merkkijonon merkille, montako erilaista alijonoa on olemassa, jotka päättyvät kyseiseen merkkiin. Merkkijonosta HAAPA tulisi muodostua seuraava taulukko:

H	A	A	P	A
1	2	4	6	12

Esimerkiksi P:n kohdalla on luku 6, koska P:hen päättyy 6 eri alijonoa: AP, AAP, HP, HAP, HAAP ja P.

Tarkastellaan nyt P:hen päättyvien alijonojen laskemista. Ensinnäkin yksi alijonoista on pelkkä kirjain P. Muissa alijonoissa ennen P:tä on jokin toinen kirjain.

Tässä tapauksessa kirjain on joko A tai H, koska muita kirjaimia ei ole esiintynyt ennen P:tä. AP-päätteisiä alijonoja on 4, koska A-päätteisiä alijonoja A:n viime esiintymässä on 4. Vastaavasti HP-päätteisiä alijonoja on 1.

Yleisesti jokaisen merkin kohdalla kyseiseen merkkiin päättyvien alijonojen määrän saa laskemalla yhteen luvun 1 (alijono on pelkkä merkki) sekä kunkin aakkoston merkin edellisen esiintymän alijonojen määrän. Vastaavalla idealla voi laskea myös koko merkkijonojen alijonojen määrän.

Toteutus 1

Seuraava ratkaisu laskee n-pituisen merkkijonon s alijonojen määrän. Taulukko c sisältää laskurin jokaiselle aakkoston merkille. Taulukko kertoo, montako merkkiin päättyvää alijonoa on ollut merkin viime esiintymässä. Lopuksi riittää käydä läpi kaikki vaihtoehdot valita alijonon viimeinen merkki.

```
for (int i = 0; i < n; i++) {
    int u = 1;
    for (int m = 'A'; m <= 'Z'; m++) u += c[m];
    c[s[i]] = u;
}
int t = 0;
for (char m = 'A'; m <= 'Z'; m++) t += c[m];
cout << t << "\n";
```

Toteutus 2

Sama ratkaisu on mahdollista toteuttaa lyhyemmin seuraavasti:

```
int t = 0;
for (int i = 0; i < n; i++) {
    int u = 1+t;
    t = t-c[s[i]]+u;
    c[s[i]] = u;
}
cout << t << "\n";
```

Luku 25

Osaongelmat

Tähänastisissa esimerkeissä dynaamisen ohjelmoinnin osaongelmat ovat olleet rakenteeltaan yksinkertaisia: esimerkiksi ruudukossa osaongelma on liittynyt tiettyyn ruutuun ja merkkijonossa tiettyyn merkkiin. Tämä luku esittelee tilanteita, joissa osaongelma on aiempaa monimutkaisempi.

25.1 Osajoukot

Dynaamisen ohjelmoinnin avulla saa joskus muutettua permutaatioiden läpikäynnin osajoukkojen läpikäynniksi. Permutaatioita on $n!$, kun taas osajoukkoja on vain 2^n, joten kyseessä on merkittävä tehostus. Esimerkiksi jos $n = 20$, permutaatioita on $20! = 2432902008176640000$, mikä on selvästi liian suuri määrä läpikäytäväksi, mutta osajoukkoja on vain $2^{20} = 1048576$.

Tarkastellaan esimerkkinä seuraavaa tehtävää:

Tehtävä: Elefantit

Pilvenpiirtäjän pohjakerroksessa on n elefanttia, ja he haluavat matkustaa hissillä huipulle. Taulukko `t` sisältää elefanttien painot, ja m on hissin suurin sallittu paino. Elefantit odottavat jonossa, ja hissiin menee kerrallaan mahdollisimman monta elefanttia jonon alusta. Kuitenkaan elefantit eivät halua ylittää hissin suurinta sallittua painoa. Tehtäväsi on järjestää elefantit jonoon niin, että hissimatkojen määrä on pienin mahdollinen.

Suoraviivainen ratkaisu tehtävään on käydä läpi kaikki elefanttien permutaatiot ja valita niistä optimaalinen järjestys. Tämä ratkaisu on riittävä, jos $n \approx 10$, mutta suuremmilla n:n arvoilla ratkaisu on liian hidas. Seuraava dynaamisen ohjelmoinnin ratkaisu on tehokas vielä, kun $n \approx 20$.

Ideana on käyttää dynaamista ohjelmointia osaongelmille ”mikä on paras tapa viedä ylös elefanttien osajoukko X”. Tässä paras tapa tarkoittaa, että minimoidaan ensisijaisesti hissimatkojen määrä ja toissijaisesti viimeisen hissimatkan lastin paino. Nämä osaongelmat palautuvat pienempiin osaongelmiin käymällä läpi kaikki tavat valita viimeinen hissiin laitettu elefantti osajoukosta.

Luonteva tapa käsitellä osajoukkoja on käyttää bittitaulukoita, jolloin jokaista osaongelmaa vastaa yksi kokonaisluku. Esimerkiksi jos osajoukossa ovat mukana 1., 2. ja 4. elefantti, vastaava bittitaulukko on 1011, joka on lukuna 11.

Tarkastellaan esimerkiksi tapausta, jossa elefanttien painot ovat 2, 3, 5, 6 ja hissin suurin sallittu paino on 8. Seuraava taulukko sisältää kaikki tähän tapaukseen liittyvät osaongelmat:

luku	bitteinä	osajoukko	hissimatkat	viimeinen lasti
0	0000	$\emptyset$	1	0
1	0001	{2}	1	2
2	0010	{3}	1	3
3	0011	{2,3}	1	5
4	0100	{5}	1	5
5	0101	{2,5}	1	7
6	0110	{3,5}	1	8
7	0111	{2,3,5}	2	2
8	1000	{6}	1	6
9	1001	{2,6}	1	8
10	1010	{3,6}	2	3
11	1011	{2,3,6}	2	3
12	1100	{5,6}	2	5
13	1101	{2,5,6}	2	5
14	1110	{3,5,6}	2	6
15	1111	{2,3,5,6}	2	8

Esimerkiksi osajoukon {2,3,6} käsittelyssä riittää tutkia, mitä tapahtuu, jos siitä poistaa 2:n, 3:n tai 6:n painoisen elefantin. Paras vaihtoehto on poistaa 3:n painoinen elefantti, jolloin jäljelle jää osajoukko {2,6}, jonka saa ylös 1 hissillä lastilla 8. Niinpä osajoukon {2,3,6} saa ylös 2 hissillä lastilla 3.

Seuraava koodi ratkaisee tehtävän dynaamisella ohjelmoinnilla. Taulukon d kohta d[x] sisältää parin, jonka ensimmäinen jäsen on hissimatkojen määrä ja toinen jäsen on viimeisen matkan lastin paino. Muuttuja x on bittitaulukko, joka ilmaisee osajoukkoon kuuluvat elefantit.

```
d[0] = make_pair(1,0);
for (int i = 1; i < (1<<n); i++) {
    d[i] = make_pair(n+1,0);
    for (int j = 0; j < n; j++) {
        if ((i&(1<<j)) == 0) continue;
        pair<int,int> u = d[i^(1<<j)];
        if (u.second+t[j] > m) {
            u.first++;
            u.second = 0;
        }
        u.second += t[j];
        d[i] = min(d[i], u);
    }
}
cout << d[(1<<n)-1].first << "\n";
```

Koodin alussa tyhjän osajoukon ratkaisuksi tulee $(1,0)$, mikä tarkoittaa yhtä tyhjää hissiä. Tämän jälkeen koodi käy läpi osajoukot järjestyksessä. Jokaisen osajoukon kohdalla koodi käy läpi kaikki tavat poistaa sieltä elefantti.

Muuttuja u sisältää parin, jossa on paras ratkaisu osajoukolle valitun elefantin poiston jälkeen. Jos elefantti ei mahdu nykyiseen hissiin, elefantille tarvitaan uusi hissi. Tällöin u:n ensimmäinen jäsen kasvaa ja toinen jäsen nollautuu. Sitten elefantti lisätään hissiin ja sen paino lasketaan mukaan. Osaongelman ratkaisu saadaan miniminä kaikista vaihtoehdoista poistaa elefantti.

Huomaa, että osajoukkojen läpikäynti bittien avulla takaa pienempien osajoukkojen käsittelyn ennen suurempia. Tämä johtuu siitä, että jos A ja B ovat osajoukkoja ja $A \subset B$, B:n bittiesitys on muuten samanlainen kuin A:n bittiesitys, mutta siihen on lisätty joitakin bittejä eli sitä vastaava luku on suurempi.

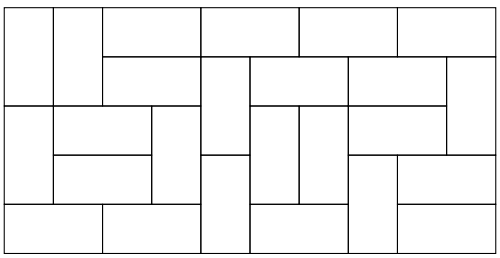
25.2 Rivi riviltä

Seuraavassa menetelmässä on idena käsitellä ruudukkoa rivi kerrallaan ja käydä läpi kaikki mahdollisuudet valita rivin sisältö. Tämä vaatii kahta asiaa tehtävältä: rivin vaihtoehtojen määrä on niin pieni, että kaikki mahdollisuudet voi käydä läpi, ja muistissa riittää pitää edellisen rivin sisältö.

Tutustutaan menetelmään seuraavan tehtävän avulla:

Tehtävä: Palikat

Tehtävänä on täyttää $5 \times n$ -ruudukko käyttämällä palikoita kokoa 1×2 ja 2×1. Montako erilaista tapaa tähän on olemassa? Esimerkiksi jos $n = 10$, yksi mahdollisuus on tämä:



Ideana on käyttää osaongelmana pystyrivin sisällön koodausta, joka on seuraavista merkeistä muodostuva merkkijono:

- `Y` (pystysuuntaisen palikan yläruutu)
- `A` (pystysuuntaisen palikan alaruutu)
- `V` (vaakasuuntaisen palikan vasen ruutu)
- `O` (vaakasuuntaisen palikan oikea ruutu)

Esimerkiksi yllä olevassa muodostelmassa ensimmäisen pystyrivin sisältö on `YAYAV`, toisen pystyrivin sisältö on `YAVVO` jne.

Seuraava koodi lisää vektoriin v kaikki mahdolliset pystyrivit:

```
vector<string> v;
```

```
void haku(string s) {
    if (s.size() == 5) v.push_back(s);
    if (s.size() >= 5) return;
    haku(s+"V");
    haku(s+"O");
    haku(s+"YA");
}
```

```
haku("");
```

Mahdollisia pystyrivejä on yhteensä 70.

Tämän jälkeen riittää käydä ruudukko läpi vasemmalta oikealle dynaamisella ohjelmoinnilla. Seuraava koodi laskee taulukkoon d[k][x] sellaisten ruudukkojen määrän, joiden leveys on k ja viimeisen pystyrivin sisältö on x. Tyhjässä ruudukosa merkkijono OOOOO kuvastaa ainoaa ratkaisua.

```
map<string,int> d[N];
```

```
d[0]["OOOOO"] = 1;
for (int i = 1; i <= n; i++) {
    for (int a = 0; a < v.size(); a++) {
         for (int b = 0; b < v.size(); b++) {
            bool ok = true;
            for (int k = 0; k < 5; k++) {
                bool v1 = v[a][k] == 'V';
                bool v2 = v[b][k] == 'O';
                if (v1 && !v2) ok = false;
                if (!v1 && v2) ok = false;
            }
            if (ok) d[i][v[b]] += d[i-1][v[a]];
        }
    }
}
```

Tämän jälkeen ruudukkojen kokonaismäärän saa laskettua näin:

```
int t = 0;
for (int x = 0; x < v.size(); x++) {
    if (v[x].find("V") != string::npos) continue;
    t += d[n][v[x]];
}
cout << t << "\n";
```

Tässä laskusta jätetään pois pystyrivit, joissa esiintyy merkki V, koska tällainen pystyrivi ei voi olla valmiin ruudukon viimeinen rivi.

25.3 Ruutualueet

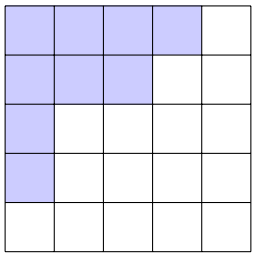
Tarkastellaan lopuksi seuraavaa tehtävää:

Tehtävä: Järjestetty ruudukko

Tehtävänä on täyttää $n \times n$ -ruudukko kokonaisluvuilla $1 \ldots n^2$ niin, että jokaisella pysty- ja vaakarivillä luvut ovat järjestyksessä pienimmästä suurimpaan. Montako tällaista ruudukkoa on olemassa? Esimerkiksi tapauksessa $n = 5$ yksi ratkaisu on seuraava:

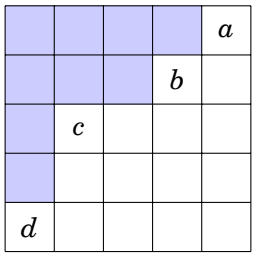
1	2	6	7	12
3	5	8	13	19
4	10	11	16	20
9	14	15	21	23
17	18	22	24	25

Oleellinen havainto on, että jokaisessa ratkaisussa k ensimmäistä lukua rajaavat yhtenäisen alueen ruudukon vasemmassa alakulmassa. Lisäksi jokaisella rivillä on yhtä suuri tai pienempi määrä lukuja kuin edellisellä rivillä. Esimerkiksi yllä olevassa ruudukossa tilanteessa $k = 9$ rajattu alue on tämä:



Tämän ansiosta dynaamisen ohjelmoinnin osaongelmaksi soveltuu ”montako tapaa on järjestää luvut annetulle alueelle”. Kätevä tapa kuvata alueen muoto on tehdä lista, joka sisältää jokaisesta rivistä tiedon, moniko rivin ruuduista kuuluu alueeseen. Esimerkiksi yllä olevan alueen voi kuvata listana $[4,3,1,1,0]$.

Uuden luvun voi sijoittaa minkä tahansa rivin loppuun, mikäli rivi ei ole vielä täynnä ja rivistä ei tule pidempi kuin edellisestä rivistä. Äskeisessä esimerkissä mahdollisia paikkoja luvulle 10 on neljä (ruudut $a \ldots d$):



Seuraava koodi toteuttaa dynaamisen ohjelmoinnin käytännössä. Koodi laskee taulukkoon d[x], montako tapaa on täyttää vektoria x vastaava alue.

```
map<vector<int>,int> d;
```

```
void haku(vector<int> &v, int k) {
    if (k == n) {
        int u = d[v];
        for (int i = 0; i < n; i++) {
            if (v[i] == n) continue;
            v[i]++; d[v] += u; v[i]--;
            if (v[i] == 0) break;
        }
        return;
    }
    int x = (k == 0) ? n : v[k-1];
    for (int i = 0; i <= x; i++) {
        v[k] = i;
        haku(v, k+1);
    }
}
```

```
vector<int> v(n);
v[0] = 1;
d[v] = 1;
haku(v, 0);
fill(v.begin(), v.end(), n);
cout << d[v] << "\n";
```

Koodi muodostaa osaongelmat rekursiivisesti, ja lähtökohtana on, että alueen $[1,0,0,\ldots,0]$ täyttämiseen on yksi tapa. Lopulta vastaus tapaukseen n saadaan taulukosta kohdasta $[n,n,n,\ldots,n]$.

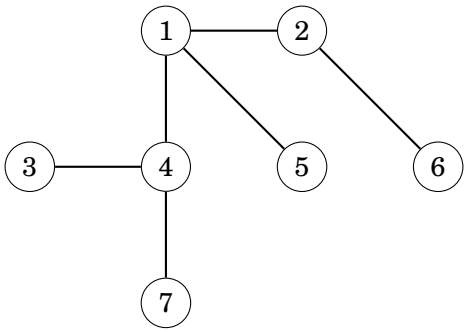
Luku 26

Puualgoritmit

Puu on yhtenäinen ja syklitön verkko, mikä tarkoittaa, että jokaisen solmuparin välillä on yksikäsitteinen polku. Puut soveltuvat hyvin dynaamiseen ohjelmointiin, koska ne jakautuvat luontevasti alipuiksi, joita voi käsitellä toisistaan riippumattomasti. Dynaamista ohjelmointia tarvitsee usein, kun jokin asia puusta täytyy laskea ajassa $O(n)$, missä n on solmujen määrä.

26.1 Läpimitta

Puun *läpimitta* tarkoittaa pisintä polkua kahden solmun välillä. Esimerkiksi seuraavan puun läpimitta on 4:



Tässä puussa on kaksi pisintä polkua:

- $3 \to 4 \to 1 \to 2 \to 6$
- $7 \to 4 \to 1 \to 2 \to 6$

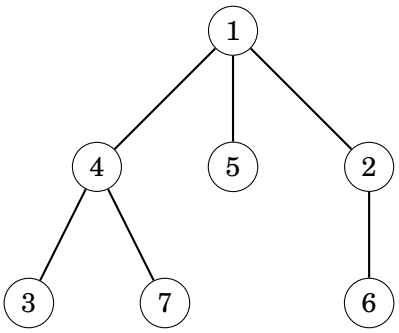
Suoraviivainen tapa laskea puun läpimitta on laskea jokaisesta solmusta pisimmät polut kaikkiin muihin solmuihin syvyyshaulla. Tässä menetelmässä kuluu kuitenkin aikaa $O(n^2)$, joten se on liian hidas suurten puiden käsittelyyn.

Tutustutaan seuraavaksi kahteen algoritmiin, joiden aikavaativuus on vain $O(n)$. Ensimmäinen algoritmi on tyypillinen dynaamisen ohjelmoinnin sovellus puun käsittelyssä. Toinen algoritmi on yllättävä ahne menetelmä, joka perustuu kahteen syvyyshakuun.

Algoritmi 1

Puun käsittelyä helpottaa usein valita yksi solmuista puun juureksi. Tämän jälkeen muut solmut järjestyvät tasoittain juuren alapuolelle, minkä jälkeen solmuja voi käsitellä luontevasti taso kerrallaan. Yleensä ei ole merkitystä, mikä solmuista valitaan puun juureksi.

Seuraavassa esimerkissä solmu 1 on valittu puun juureksi:



Nyt puun pisin polku alkaa jostain lehdestä puun pohjalta, nousee ylös puuta johonkin solmuun asti, ja palaa toista reittiä toiseen lehteen. Esimerkiksi yllä olevassa puussa polku $3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ alkaa lehdestä 3, kiipeää ylös solmuun 1, ja laskeutuu sitten lehteen 6.

Seuraava funktio laskee jokaiselle puun solmulle, mikä on pisin siitä alaspäin lähtevä polku (taulukko d) sekä mikä on pisin polku kahden lehden välillä, jonka kääntöpiste on kyseinen solmu (taulukko q). Oletuksena on, että puu on tallennettu vieruslistoina taulukkoon p. Funktion parametri *s* on käsiteltävä solmu ja *e* on edellisen tason solmu.

```
void haku(int s, int e) {
    d[s] = 0;
    int t1 = 0, t2 = 0;
    for (int i = 0; i < p[s].size(); i++) {
        if (p[s][i] == e) continue;
        haku(p[s][i], s);
        int u = d[p[s][i]]+1;
        d[s] = max(d[s], u);
        if (u > t1) {t2 = t1; t1 = u;}
        else if (u > t2) t2 = u;
    }
    q[s] = t1+t2;
}
```

Funktiota voisi kutsua näin:

```
haku(1, 0);
```

Funktio haarautuu käsiteltävästä solmusta rekursiivisesti kaikkiin alempana oleviin solmuihin. Parametrin *e* tarkoituksena on estää funktiota palautumasta puus-

sa ylempään solmuun. Funktio pitää kirjaa muuttujissa `t1` ja `t2` kahdesta pisimmästä polusta alaspäin, joista saadaan laskettua oikea arvo taulukkoon `q`.

Esimerkkipuussa taulukoista tulee:

s	1	2	3	4	5	6	7
d[s]	2	1	0	1	1	0	0
q[s]	4	1	0	2	0	0	0

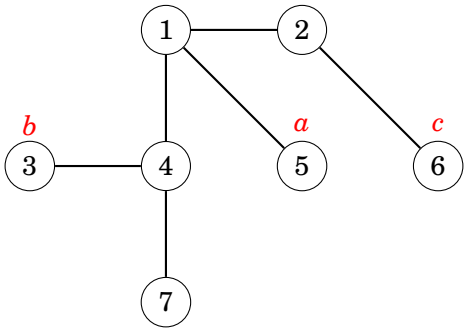
Puun läpimitta on taulukon `q` suurin arvo eli tässä 4.

Algoritmi 2

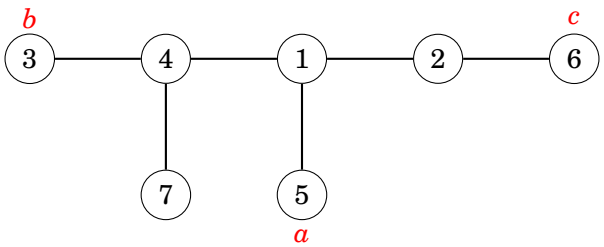
Toinen tehokas tapa laskea puun läpimitta on seuraava:

1. Valitse mikä tahansa solmu a.
2. Etsi a:sta kaukaisin solmu b (syvyyshaku).
3. Etsi b:stä kaukaisin solmu c (syvyyshaku).
4. Läpimitta on etäisyys b:n ja c:n välillä.

Esimerkissä a, b ja c voisivat olla:



Miksi tämä menetelmä sitten toimii? Tässä auttaa tarkastella puuta niin, että tuleva puun läpimittaa vastaava polku on levitetty vaakatasoon ja muut puun osat riippuvat siitä alaspäin:

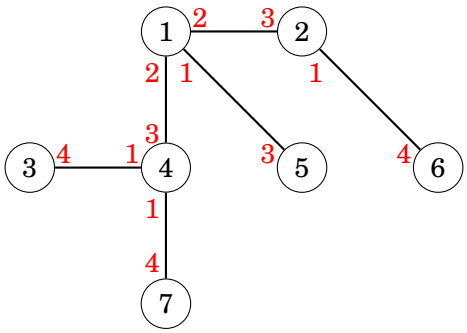


Oletetaan, että a:sta on pidempi matka b:hen kuin c:hen. Nyt jos pisin polku a:sta päätyisi johonkin toiseen solmuun x, se kulkisi ensin samaa reittiä kuin solmuun b, mutta eroaisi jossain vaiheessa. Tästä kohdasta matka b:hen ja x:ään on oltava yhtä pitkä, joten polku x:stä c:hen olisi myös yhtä pitkä kuin polku b:stä c:hen.

Tässä esimerkissä pisin polku a:sta voisi päätyä myös solmuun 7, joka vastaa toista pisintä polkua puussa.

26.2 Pisimmät polut

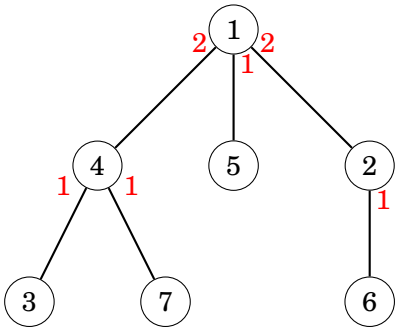
Tarkastellaan sitten vaikeampaa tehtävää, jossa tehtävänä on laskea jokaiselle puun solmulle pisimmät polut, jotka lähtevät eri suuntiin kyseisestä solmusta. Esimerkiksi seuraavassa verkossa polkujen pituudet ovat:



Tässäkin tehtävässä hyvä lähtökohta on valita jokin solmu puun juureksi. Tämän jälkeen pisimmät polut jokaisesta solmusta alaspäin saa laskettua samalla tavalla kuin läpimitan laskemisessa algoritmissa 1. Seuraavassa koodissa taulukko q tulee sisältämään kustakin kaaresta alkavan pisimmän polun pituuden samassa järjestyksessä kuin kaaret ovat taulukossa p:

```
void haku(int s, int e) {
    d[s] = 0;
    q[s].resize(p[s].size());
    for (int i = 0; i < p[s].size(); i++) {
        if (p[s][i] == e) continue;
        haku(p[s][i], s);
        int u = d[p[s][i]]+1;
        d[s] = max(d[s], u);
        q[s][i] = u;
    }
}
```

Esimerkkitapauksessa tuloksena on seuraava puu:



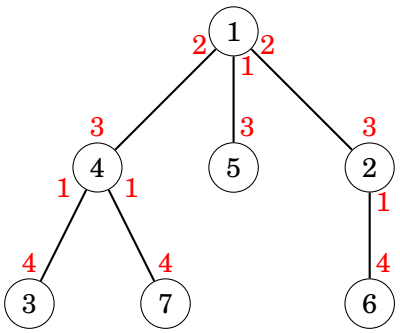
Jäljelle jäävä tehtävä on laskea jokaiselle solmulle juurta lukuun ottamatta pisin polku ylöspäin. Tämä onnistuu suorittamalla toinen puun läpikäynti juuresta alkaen, jossa uutena parametrina (x) on pisin ylhäältä päin tulevan polun pituus:

```
void haku2(int s, int e, int x) {
    int t1 = x, t2 = 0;
    for (int i = 0; i < p[s].size(); i++) {
        if (p[s][i] == e) {
            q[s][i] = x;
            continue;
        }
        int u = q[s][i];
        if (u > t1) {t2 = t1; t1 = u;}
        else if (u > t2) {t2 = u;}
    }
    for (int i = 0; i < p[s].size(); i++) {
        if (p[s][i] == e) continue;
        int u = t1;
        if (t1 == q[s][i]) u = t2;
        haku2(p[s][i], s, u+1);
    }
}
```

Funktiota kutsutaan näin:

```
haku2(1, 0, 0);
```

Nyt kaikkien kaarten tulokset ovat selvillä:

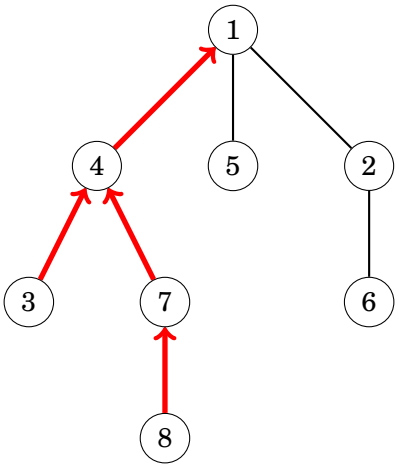


Funktio käy solmusta lähtevät kaaret läpi kaksi kertaa. Ensimmäisellä kerralla funktio merkitsee muistiin solmusta ylöspäin lähtevän kaaren pisimmän polun pituuden sekä laskee muuttujiin `t1` ja `t2` kaksi pisintä solmusta lähtevää polkua. Toisella kerralla funktio jatkaa hakua rekursiivisesti alaspäin puussa. Ylhäältä tulevan polun pituus on yleensä yhden suurempi kuin `t1`, mutta jos `t1` on sama kuin käsiteltävän kaaren polun pituus, käytetään arvoa `t2` (pisin polku ei voi olla sellainen, joka kulkee samaa kaarta monta kertaa).

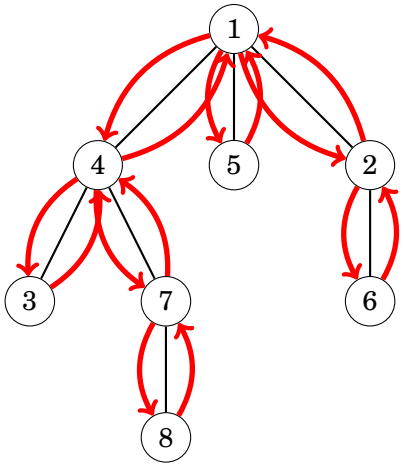
26.3 Alin yhteinen vanhempi

Tarkastellaan lopuksi tehtävää, jossa puu on juurrettu ja tehtävänä on vastata nopeasti kyselyihin muotoa ”mikä on solmujen a ja b alin yhteinen vanhempi?” Tämä tarkoittaa sellaista solmua, joka kuuluu polkuun kummastakin solmusta juureen ja on mahdollisimman matalalla puussa.

Esimerkiksi seuraavassa puussa 3:n ja 8:n alin yhteinen vanhempi on 4:



Tutustutaan seuraavaksi menetelmään, jolla $O(n \log n)$ esikäsittelyn jälkeen minkä tahansa kahden solmun alin yhteinen vanhempi selviää ajassa $O(\log n)$. Ideana on muodostaa ensin taulukko solmuista ja niiden syvyyksistä siinä järjestyksessä kuin ne tulevat vastaan puun läpikäynnissä. Esimerkiksi läpikäynnistä



muodostuu seuraava taulukko:

solmu	1	4	3	4	7	8	7	4	1	5	1	2	6	2	1
syvyys	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Nyt kun halutaan selvittää kahden solmun alin yhteinen vanhempi, riittää etsiä taulukosta alue, jonka reunasolmuina ovat kyseiset solmut. Alin yhteinen vanhempi on tällä alueella oleva solmu, jonka syvyys on pienin.

Esimerkiksi solmujen 3 ja 8 alin yhteinen vanhempi selviää seuraavasti:

solmu	1	4	3	4	7	8	7	4	1	5	1	2	6	2	1
syvyys	0	1	2	1	2	3	2	1	0	1	0	1	2	1	0

Tällä alueella pienin syvyys on 2, jota vastaa solmu 4. Jos mahdollisia alueita on useita, mikä tahansa kelpaa.

Tehokas tapa toteuttaa pienimmän syvyyden etsiminen on tehdä taulukon arvoista segmenttipuu. Jokaisen taulukon kohdan voi esittää parina, jonka ensimmäinen jäsen on solmun syvyys ja toinen jäsen on solmun tunnus. Läpikäynnin seurauksena taulukkoon tulee $2n-1$ paria ja minkä tahansa alueen alin yhteinen vanhempi löytyy ajassa $O(\log n)$.

Seuraava koodi käy puun läpi ja muodostaa segmenttipuun `q`. Parametri c on solmun syvyys ja globaali muuttuja z on solmun kohta taulukossa. Lisäksi taulukkoon `w` merkitään jokaisesta solmusta, missä kohtaa jokin sen esiintymä on segmenttipuussa.

```
void haku(int s, int e, int c) {
    w[s] = z;
    for (int i = 0; i < p[s].size(); i++) {
        muuta(z++, make_pair(c, s));
        if (p[s][i] == e) continue;
        haku(p[s][i], s, c+1);
    }
    muuta(z++, make_pair(c, s);
}
```

Tässä on vielä segmenttipuun toteutus:

```
void muuta(int k, pair<int,int> x) {
    k += N;
    q[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        q[k] = min(q[2*k], q[2*k+1]);
    }
}

pair<int,int> pienin(int a, int b) {
    a += N; b += N;
    pair<int,int> t = q[a];
    while (a <= b) {
        if (a%2 == 0) t = min(t, q[a++]);
        if (b%2 == 1) t = min(t, q[b--]);
        a /= 2; b /= 2;
    }
    return t;
}
```

Nyt solmujen a ja b alimman yhteisen vanhemman saa selville näin:

```
haku(1, 0, 0);
```

```
int x = w[a];
int y = w[b];
if (x > y) swap(x, y);
auto z = pienin(x, y);
cout << z.second << "\n";
```

Yksi kätevä sovellus alimman yhteisen vanhemman etsimiselle on a:n ja b:n välisen polun pituuden laskeminen. Tämän pystyy laskemaan, kun tiedetään solmujen syvyydet sekä alimman yhteisen vanhemman syvyys:

```
int s1 = pienin(x,x).first;
int s2 = pienin(y,y).first;
int s3 = pienin(x,y).first;
cout << s1+s2-2*s3 << "\n";
```

Esimerkissä solmun 3 syvyys on 2, solmun 8 syvyys on 3 ja yhteisen vanhemman syvyys on 1, joten solmujen välisen polun pituus on $2+3-2 \cdot 1 = 3$.

Luku 27

Kombinatoriikka

Kombinatoriikka on matematiikan ala, joka tutkii yhdistelmien määrän laskemista. Usein haasteena on muotoilla kombinatorinen ongelma rekursiivisesti niin, että vastauksen pystyy laskemaan ongelman pienemmistä tapauksista. Tällöin vastauksen laskemiseen voi käyttää myös luontevasti dynaamista ohjelmointia.

27.1 Kombinatoriset funktiot

Kombinatorinen funktio kertoo yhdistelmien määrän tietyssä tilanteessa. Kombinatoriset funktiot voi yleensä määritellä rekursiivisesti, jolloin niiden arvoja voi laskea tehokkaasti dynaamisella ohjelmoinnilla. Palautamme aluksi mieleen muutaman perusfunktion, jotka ovat tuttuja muista yhteyksistä.

Kertoma

Luvun n kertoma $n!$ määritellään näin:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Kertoma ilmaisee, montako permutaatiota eli erilaista järjestystä voi muodostaa n alkion joukosta. Esimerkiksi joukossa $\{1,2,3\}$ on 3 alkiota, joten siitä voi muodostaa $3! = 6$ permutaatiota: $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ sekä $(3,2,1)$.

Seuraava koodi laskee taulukkoon `k` lukujen $0 \ldots n$ kertomat:

```
k[0] = 1;
for (int i = 1; i <= n; i++) {
    k[i] = i*k[i-1];
}
```

Potenssilasku

Potenssilasku k^n määritellään näin:

$$\begin{aligned} k^0 &= 1 \\ k^n &= k \cdot k^{n-1} \end{aligned}$$

Potenssilasku kertoo yhdistelmien määrän, kun tehdään n valintaa ja jokaiseen valintaan on k vaihtoehtoa. Esimerkiksi 3-merkkisiä bittijonoja on $2^3 = 8$, koska bittejä on 3 ja jokaisen valintaan on 2 vaihtoehtoa (0 tai 1). Tässä tapauksessa bittijonot ovat 000, 001, 010, 011, 100, 101, 110 ja 111.

Potensseja voi laskea samaan tapaan kuin kertomia:

```
p[0] = 1;
for (int i = 1; i <= n; i++) {
    p[i] = k*p[i-1];
}
```

Jos laskettavana on yksittäinen potenssi k^n, tämä on mahdollista laskea tehokkaammin ajassa $O(\log n)$ seuraavan rekursion avulla:

$$\begin{aligned} k^0 &= 1 \\ k^n &= k \cdot k^{n-1} && \text{jos } n \text{ on pariton} \\ k^n &= k^{n/2} \cdot k^{n/2} && \text{jos } n \text{ on parillinen} \end{aligned}$$

Viimeisen kaavan ansiosta potenssin k^n laskemiseksi tarvitsee laskea vain $O(\log n)$ pienempää potenssia. Käytännössä tämän voi koodata näin:

```
int pot(int k, int n) {
    if (n == 0) return 1;
    int x = pot(k,n/2);
    if (n%2 == 0) return x*x;
    else return k*x*x;
}
```

Huomaa, että tässä on oleellista laskea välitulos muuttujaan x, jotta välitulos lasketaan vain kerran.

Fibonaccin luvut

Fibonaccin luvut määritellään näin:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Ensimmäiset Fibonaccin luvut ovat:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

Fibonaccin luvuilla on monia sovelluksia kombinatoriikassa. Esimerkiksi F_{n+2} kertoo n merkin bittijonojen määrän, joissa ei ole kahta ykkösbittiä peräkkäin. Tapauksessa $n = 4$ näitä on $F_6 = 8$: 0000, 0001, 0010, 0100, 0101, 1000, 1001 ja 1010. Tämä johtuu siitä, että jokaisen nollabitin kohdalla joko edellinen nollabitti on vieressä tai välissä on yksi ykkösbitti.

Myös Fibonaccin lukuja voi laskea dynaamisella ohjelmoinnilla:

```
f[0] = 0;
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = f[i-2]+f[i-1];
}
```

27.2 Binomikerroin

Binomikerroin määritellään näin:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Esimerkiksi

$$\binom{5}{2} = \frac{5!}{2!3!} = \frac{120}{12} = 10.$$

Binomikerroin kertoo, montako erilaista k henkilön ryhmää voidaan muodostaa n henkilön joukosta. Esimerkiksi 5 henkilön joukosta $\{A,B,C,D,E\}$ voidaan muodostaa 10 ryhmää, joissa on 2 henkilöä: $\{A,B\}$, $\{A,C\}$, $\{A,D\}$, $\{A,E\}$, $\{B,C\}$, $\{B,D\}$, $\{B,E\}$, $\{C,D\}$, $\{C,E\}$ sekä $\{D,E\}$.

Binomikertoimen voi myös laskea rekursiivisesti seuraavasti:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Tämän kaavan voi perustella seuraavasti: Oletetaan, että Uolevi on yksi henkilöistä. Nyt kun n henkilöstä valitaan k hengen ryhmä, on kaksi mahdollisuutta: joko Uolevi tulee mukaan ryhmään tai ei. Jos Uolevi tulee mukaan, pitää valita vielä $n-1$ henkilöstä $k-1$ hengen ryhmä. Jos taas Uolevi ei tule mukaan, pitää valita vielä $n-1$ henkilöstä k hengen ryhmä.

Lisäksi on kaksi reunatapausta:

$$\binom{n}{0} = 1$$

$$\binom{n}{k} = 0, \text{ jos } k > n$$

Ensimmäinen kaava johtuu siitä, että on aina 1 tapa muodostaa 0 hengen ryhmä: eli valita ketään ryhmään. Toinen kaava johtuu siitä, että ryhmän ei voi valita enempää henkilöitä kuin mitä on olemassa.

Huomaa myös seuraava kaava:

$$\binom{n}{k} = \binom{n}{n-k}$$

Tämä johtuu siitä, että jokaisessa valinnassa k henkilön ryhmäksi ulkopuolelle jää toinen $n-k$ henkilön ryhmä.

Seuraava koodi laskee dynaamisella ohjelmoinnilla kaikki binomikertoimet, joissa alkioiden määrä on $0 \ldots n$. Koodi olettaa, että ennen silmukoiden suoritusta jokainen arvo taulukossa `d` on 0.

```
for (int i = 0; i <= n; i++) {
    d[i][0] = 1;
    for (int j = 1; j <= i; j++) {
        d[i][j] = d[i-1][j-1]+d[i-1][j];
    }
}
```

Nimi ”binomikerroin” johtuu siitä, että laskussa $(a+b)^n$ binomikertoimet ovat muuttujien a ja b eri tulojen kertoimet. Esimerkiksi

$$(a+b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$$

eli

$$(a+b)^5 = \binom{5}{0}a^5 + \binom{5}{1}a^4b + \binom{5}{2}a^3b^2 + \binom{5}{3}a^2b^3 + \binom{5}{4}ab^4 + \binom{5}{5}b^5.$$

Binomikertoimet esiintyvät myös Pascalin kolmiossa, jonka reunalla on ykkösiä ja sisällä jokainen luku on kahden ylemmän luvun summa:

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
```

27.3 Catalanin luvut

Catalanin luvut määritellään näin:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Ensimmäiset Catalanin luvut ovat:

1, 2, 5, 14, 42, 132, 429, 1430, 4862

Luku C_n kertoo esimerkiksi, monellako tavalla voidaan muodostaa n sulkuparista koostuva merkkijono, jotka on sulutettu oikein. Esimerkiksi $C_3 = 5$, koska mahdollisuudet ovat `()()()`, `(())()`, `()(())`, `(()())` sekä `((()))`.

Catalanin luvut voi määritellä myös rekursiivisesti näin:

$$\begin{aligned} C_0 &= 1 \\ C_n &= \sum_{i=0}^{n-1} C_i C_{n-1-i} \end{aligned}$$

Tämä kertoo selkeämmin, miksi tuloksena on sulkuparien määrä. Jokaisessa oikein sulutetussa merkkijonossa ensimmäinen merkki on `(` ja jossain myöhemmin on tämän vastapari `)`. Nyt sekä merkkien `(` ja `)` välissä että merkin `)` jälkeen täytyy olla oikein sulutettu merkkijono. Tapauksen C_n summa käy läpi kaikki eri mahdollisuudet valita ensimmäisen sulun `(` vastapari `)`.

Catalanin luvut voi laskea koodissa näin:

```
c[0] = 1;
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= i-1; j++) {
        c[i] += c[j]*c[i-1-j];
    }
}
```

27.4 Stirlingin luvut

Stirlingin luvut määritellään näin:

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$$

Lisäksi reunatapaukset ovat:

$$\left\{ {0 \atop 0} \right\} = 1$$

$$\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 0, \text{ jos } n > 0$$

Stirlingin luvut kertovat, monellako tavalla n henkilön joukko voidaan jakaa k ryhmäksi. Esimerkiksi

$$\left\{ {4 \atop 3} \right\} = 6,$$

koska joukko $\{A,B,C,D\}$ voidaan jakaa 6 tavalla 3 ryhmäksi:

- $\{A\}$, $\{B\}$ ja $\{C,D\}$
- $\{A\}$, $\{B,C\}$ ja $\{D\}$
- $\{A\}$, $\{B,D\}$ ja $\{C\}$
- $\{A,B\}$, $\{C\}$ ja $\{D\}$
- $\{A,C\}$, $\{B\}$ ja $\{D\}$
- $\{A,D\}$, $\{B\}$ ja $\{C\}$

Rekursiivisen kaavan yleisen tapauksen

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$$

voi ymmärtää näin: Oletetaan, että Uolevi on yksi henkilöistä. Jos Uolevi on samassa ryhmässä jonkun muun kanssa, sitä ennen $n-1$ henkilöä on jaettu k ryhmäksi ja Uolevilla on k tapaa valita, mihin ryhmään hän menee. Jos taas Uolevi on omassa ryhmässään, sitä ennen $n-1$ henkilöä on jaettu $k-1$ ryhmäksi.

Tässä on vielä koodi, joka laskee Stirlingin lukuja:

```
s[0][0] = 1;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        s[i][j] = j*s[i-1][j]+s[i-1][j-1];
    }
}
```

Luku 28

Peliteoria

Peliteoria on matematiikan ala, jossa tavoitteena on analysoida pelin rakenne sekä selvittää, mikä on paras tapa pelata peliä. Tyypillinen peliteorian tehtävä on selvittää pelin voittaja, jos molemmat pelaajat toimivat optimaalisesti. Toinen mahdollinen tavoite on pyrkiä maksimoimaan pelaajan tulos pelissä.

28.1 Tikkupeli

Tarkastellaan aluksi seuraavaa peliä:

Pöydällä on kasa tikkuja, ja kaksi pelaajaa poistaa siitä vuorotellen tikkuja. Joka vuorolla saa poistaa 1, 2 tai 3 tikkua. Pelin voittaa se, joka pystyy poistamaan viimeisen tikun pöydältä.

Esimerkiksi jos tikkuja on aluksi 10 ja pelaajat ovat Uolevi ja Maija, peli voisi edetä seuraavasti:

- Uolevi poistaa 2 tikkua, ja pöydälle jää 8 tikkua.
- Maija poistaa 3 tikkua, ja pöydälle jää 5 tikkua.
- Uolevi poistaa 1 tikun, ja pöydälle jää 4 tikkua.
- Maija poistaa 2 tikkua, ja pöydälle jää 2 tikkua.
- Uolevi poistaa 2 tikkua ja voittaa pelin.

Peli muodostuu *tiloista*, joiden muutoksiin pelaajat vaikuttavat siirroillaan. Tässä tapauksessa pelin tila on tikkujen määrä. Esimerkin alussa pelin tila on 10 ja Uolevin ensimmäinen siirto muuttaa pelin tilaksi 8.

Pelin tilat voidaan luokitella kahteen ryhmään:

- *voittotila*: tässä tilassa pelaaja voittaa, jos hän pelaa optimaalisesti
- *häviötila*: tässä tilassa pelaaja häviää, jos vastustaja pelaa optimaalisesti

Tässä pelissä tila 1 on selvästi voittotila, samoin tilat 2 ja 3. Näissä tiloissa oleva pelaaja pystyy poistamaan pöydältä viimeisen tikun yhdellä siirrolla. Tila 4 taas on häviötila, koska mikä tahansa siirto johtaa tilaan 1, 2 tai 3.

Seuraava taulukko luokittelee pelin tilat 0–15:

tila	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
luokka	H	V	V	V	H	V	V	V	H	V	V	V	H	V	V	V

Tässä V tarkoittaa voittotilaa ja H häviötilaa. Esimerkiksi tila 10 on voittotila, koska poistamalla 2 tikkua päästään tilaan 8, joka on häviötila. Tila 12 taas on häviötila, koska poistamalla 1–3 tikkua tuloksena on voittotila.

Pelin *voittostrategia* tarkoittaa taktiikkaa, jolla pelin pystyy voittamaan varmasti voittotilasta lähtien. Tästä pelistä paljastuu yksinkertainen kuvio: tila on voittotila tarkalleen silloin, kun tikkujen määrä ei ole jaollinen 4:llä. Niinpä voittostrategia on varmistaa, että vastustajalle jää 4:llä jaollinen määrä tikkuja.

Seuraava funktio on optimaalinen tekoäly tikkupeliin:

```
int siirto(int n) {
    if (n%4 == 0) return 1;
    else return n%4;
}
```

Jos tikkujen määrä on jaollinen 4:llä, kyseessä on häviötila. Tällöin funktio poistaa yhden tikun toiveena, että vastustaja tekee virheen. Muuten kyseessä on voittotila, ja tekoäly tulee voittamaan varmasti. Silloin riittää poistaa tikkuja jakojäännöksen 4:llä verran.

Tarkastellaan vielä yleistä tapausta, jossa joukko P sisältää mahdolliset poistomäärät. Esimerkiksi jos $P = \{1,3,5,8\}$, pelaaja saa poistaa 1, 3, 5 tai 8 tikkua. Nyt täydellisen tekoälyn voi toteuttaa dynaamisella ohjelmoinnilla. Ensimmäinen vaihe on luokitella tilat voitto- ja häviötiloiksi.

Seuraava koodi merkitsee taulukkoon `t` arvon 'H' tai 'V' tilan tyypin mukaan. Mahdollisia poistomääriä on k erilaista ja ne on annettu taulukossa `p`.

```
t[0] = 'H';
for (int i = 1; i <= n; i++) {
    t[i] = 'H';
    for (int j = 0; j < k; j++) {
        if (i-p[j] >= 0 && t[i-p[j]] == 'H') {
            t[i] = 'V';
            break;
        }
    }
}
```

Jokaisen tilan kohdalla koodi olettaa ensin, että tila on häviötila. Sitten koodi käy läpi poistomäärät, ja jos löytyy tapa poistaa tikkuja, joka johtaa häviötilaan, kyseinen tila muuttuu voittotilaksi.

Tämän jälkeen täydellisen tekoälyn pystyy toteuttamaan näin:

```
int siirto(int n) {
    if (t[i] == 'H') return p[0];
    for (int i = 0; i < k; i++) {
        if (n-p[i] >= 0 && t[i-p[i]] == 'H') {
            return p[i];
        }
    }
}
```

Jos tekoäly on häviötilassa, se valitsee poistomääräksi ensimmäisen mahdollisuuden taulukosta p. Muuten se käy kaikki poistomäärät läpi ja etsii sen, jonka avulla vastustaja joutuu häviötilaan.

28.2 Lukupeli

Tarkastellaan sitten seuraavaa peliä:

Lista sisältää n kokonaislukua. Pelissä on kaksi pelaajaa, jotka poistavat vuorollaan yhden luvun joko listan alusta tai lopusta. Pelin päätyttyä voittaja on se, jonka poistamien lukujen summa on suurempi.

Esimerkiksi jos listan sisältö on $[4,3,2,5]$, pelaaja voi joko poistaa luvun 4 listan alusta tai luvun 5 listan lopusta. Ensimmäisessä tapauksessa jäljelle jää lista $[3,2,5]$ ja toisessa tapauksessa jäljelle jää lista $[4,3,2]$.

Nyt pelin tilana on alkuperäisen listan jäljellä oleva lukuväli. Esimerkiksi aloitustila on $(1,4)$ ja siitä pääsee tiloihin $(2,4)$ ja $(1,3)$.

Tässä pelissä tiloihin liittyy *arvo*: kuinka suuren summan tilassa oleva pelaaja voi saada korkeintaan, jos vastustaja pelaa täydellisesti. Pelaajan kannattaa tehdä sellainen valinta, että vastustajan tilan arvo on mahdollisimman pieni.

Jos tilaan kuuluu vain yksi luku, sen arvo on suoraan kyseinen luku. Esimerkiksi tilan $(2,2)$ arvo on 3, koska listassa kohdassa 2 on luku 3. Muussa tapauksessa tilan arvon saa selville käymällä läpi mahdolliset siirrot. Esimerkiksi tilasta $(1,4)$ pääsee tiloihin $(2,4)$ ja $(1,3)$, joten tilan arvo saadaan laskemalla summa listan luvuista $1 \ldots 4$ ja poistamalla siitä pienempi tilojen $(2,4)$ ja $(1,3)$ arvoista.

Seuraava taulukko sisältää esimerkin kaikkien tilojen arvot:

tila	arvo	tila	arvo	tila	arvo	tila	arvo
$(1,1)$	4	$(1,2)$	4	$(1,3)$	6	$(1,4)$	8
$(2,2)$	3	$(2,3)$	3	$(2,4)$	7		
$(3,3)$	2	$(3,4)$	5				
$(4,4)$	5						

Esimerkiksi tilan $(1,4)$ arvo on 8, koska välin lukujen summa on $4+3+2+5 = 14$ ja minimi tilojen $(1,3)$ ja $(2,4)$ arvoista on 6. Niinpä optimaalinen ratkaisu alussa on valita luku 5 oikealta, minkä seurauksena omaksi summaksi tulee 8 ja vastustajan summaksi tulee 6.

Pelin tilojen kokonaismäärä on $O(n^2)$, joten jokaisen tilan arvon voi laskea dynaamisella ohjelmoinnilla. Seuraavassa koodissa annettuna on taulukko `x`, joka sisältää listalla olevat luvut. Koodi muodostaa taulukot `s` ja `t`: `s`[a][b] on summa välillä $a \ldots b$ ja `t`[a][b] on kyseinen tilan arvo.

```
for (int i = 1; i <= n; i++) {
    u = 0;
    for (int j = i; j <= n; j++) {
        u += x[j];
        s[i][j] = u;
    }
}
for (int i = 1; i <= n; i++) t[i][i] = x[i];
for (int k = 2; k <= n; k++) {
    for (int i = 1; i+k-1 <= n; i++) {
        int z = min(t[i][i+k-2], t[i+1][i+k-1]);
        t[i][i+k-1] = s[i][i+k-1]-z;
    }
}
```

Taulukon `t` avulla on helppoa toteuttaa optimaalinen tekoäly: paras valinta on aina sellainen, jonka jälkeen vastustajan tilan arvo on mahdollisimman pieni.

Jos n on parillinen, peliin on olemassa myös yksinkertainen strategia, jota seuraamalla oma summa on ainakin yhtä suuri kuin vastustajan summa. Tämä johtuu siitä, että pelin aloittaja pystyy halutessaan saamaan kaikki parittomissa kohdissa tai kaikki parillisissa kohdissa listaa olevat luvut.

Esimerkiksi listassa $[4,3,2,5]$ luvut 4 ja 2 ovat parittomissa kohdissa ja 3 ja 5 parillisissa kohdissa. Jos pelaaja poistaa aina parillisen kohdan luvun, vastustaja on seuraavaksi tilanteessa, jossa molemmat luvut ovat parittomissa kohdissa, ja vastaavasti toisinpäin. Niinpä riittää laskea lukujen summat parittomissa ja parillisissa kohdissa ja valita näistä suurempi.

Osa IV

Erikoisaiheita

Luku 29

Matriisit

Matriisi on kaksiulotteista taulukkoa vastaava matemaattinen käsite. Esimerkiksi seuraavassa on 3×4 -kokoinen matriisi A:

$$A = \begin{pmatrix} 8 & 17 & 7 & 4 \\ 7 & 19 & 5 & 10 \\ 19 & 9 & 4 & 18 \end{pmatrix}$$

Merkitään $A_{i,j}$ matriisin A rivillä i sarakkeessa j olevaa arvoa. Esimerkiksi yllä olevassa matriisissa $A_{2,3} = 5$ ja $A_{3,4} = 18$.

29.1 Laskutoimitukset

Matriisien A ja B *yhteenlasku* $A + B$ on määritelty, jos kummankin matriisin korkeus ja leveys on sama. Tällöin uusi matriisi saadaan laskemalla yhteen kaikki matriisien vastaavissa kohdissa olevat luvut.

Seuraavassa on esimerkki matriisien yhteenlaskusta:

$$A = \begin{pmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{pmatrix} \qquad B = \begin{pmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{pmatrix} = \begin{pmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{pmatrix}$$

Matriisien A ja B *kertolasku* AB on määritelty, jos vasemman matriisin leveys on sama kuin oikean matriisin korkeus. Toisin sanoen matriisin A koon tulee olla $a \times n$ ja matriisin B koon tulee olla $n \times b$.

Ideana on muodostaa uusi $a \times b$ -matriisi seuraavasti:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Seuraavassa on esimerkkejä matriisien kertolaskuista:

$$A = \begin{pmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 6 \\ 2 & 9 \end{pmatrix}$$

$$AB = \begin{pmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{pmatrix} = \begin{pmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{pmatrix}$$

$$A = \begin{pmatrix} 9 & 6 & 5 \\ 4 & 1 & 8 \end{pmatrix} \qquad B = \begin{pmatrix} 3 \\ 5 \\ 8 \end{pmatrix}$$

$$AB = \begin{pmatrix} 9 \cdot 3 + 6 \cdot 5 + 5 \cdot 8 \\ 4 \cdot 3 + 1 \cdot 5 + 8 \cdot 8 \end{pmatrix} = \begin{pmatrix} 97 \\ 81 \end{pmatrix}$$

Tavallisesta kertolaskusta poiketen matriisien kertolasku ei ole vaihdannainen, eli ei ole voimassa $AB = BA$. Kuitenkin matriisien kertolasku on liitännäinen eli $A(BC) = (AB)C$. Matriisien potenssilasku määritellään saman tapaan kuin tavallinen potenssilasku, esimerkiksi $A^3 = AAA$.

29.2 Nopea potenssilasku

Jos matriisit A ja B ovat $n \times n$ -kokoisia, kertolaskun AB aikavaativuus on $O(n^3)$ suoraviivaisesti toteutettuna. Seuraava koodi laskee tulon AB matriisiin C:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        C[i][j] = 0;
        for (int k = 1; k <= n; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

Potenssilaskun A^k aikavaativuus on $O(n^3 k)$, jos sen toteuttaa k kertolaskuna. Mutta potenssilaskun voi toteuttaa myös ajassa $O(n^3 \log k)$ samalla tekniikalla kuin yksittäisen luvun tehokkaan potenssilaskun.

Esimerkiksi lasku

$$\begin{pmatrix} 1 & 4 \\ 5 & 3 \end{pmatrix}^{16}$$

jakaantuu osiin

$$\begin{pmatrix} 1 & 4 \\ 5 & 3 \end{pmatrix}^{8} \cdot \begin{pmatrix} 1 & 4 \\ 5 & 3 \end{pmatrix}^{8},$$

eli ongelman koko puoliintuu, kun potenssi on parillinen.

29.3 Dynaaminen ohjelmointi

Matriisien ja tehokkaan potenssilaskun hyötynä on, että tietyt dynaamisen ohjelmoinnin ratkaisut voi esittää matriisimuodossa. Niinpä tehokkaan potenssilaskun avulla aikavaativuuden lineaarisesta kertoimesta tulee logaritminen.

Tarkastellaan esimerkkinä tästä Fibonaccin lukujen laskemista. Tavallinen rekursiivinen kaava on:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Matriisimuodossa asian voi esittää näin:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

$$X \cdot \begin{pmatrix} F_k \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} F_{k+1} \\ F_{k+2} \end{pmatrix}$$

Ideana on, että jos 2×1 -matriisissa on peräkkäiset Fibonaccin luvut F_k ja F_{k+1}, kertomalla tämän matriisilla X syntyy uusi matriisi, jossa on peräkkäiset Fibonaccin luvut F_{k+1} ja F_{k+2}. Esimerkiksi

$$X \cdot \begin{pmatrix} F_5 \\ F_6 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 8 \end{pmatrix} = \begin{pmatrix} 0 \cdot 5 + 1 \cdot 8 \\ 1 \cdot 5 + 1 \cdot 8 \end{pmatrix} = \begin{pmatrix} 8 \\ 13 \end{pmatrix} = \begin{pmatrix} F_6 \\ F_7 \end{pmatrix}.$$

Tämän ansiosta arvon F_n sisältävän matriisin saa laskettua

$$\begin{pmatrix} F_n \\ F_n + 1 \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

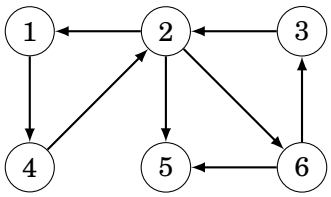
Matriisin potenssilasku onnistuu ajassa $O(\log n)$, joten tällä tekniikalla Fibonaccin luvun F_n pystyy laskemaan ajassa $O(\log n)$.

Samaa ideaa voi soveltaa aina, kun rekursiivisen funktion seuraava arvo saadaan laskemalla yhteen kiinteä määrä edellisiä arvoja sopivilla kertoimilla.

29.4 Verkko matriisina

Matriisin potenssilaskulla on mielenkiintoinen vaikutus verkon vierusmatriisin sisältöön. Oletetaan ensin, että vierusmatriisi V kuvaa verkon ja verkon kaarilla ei ole painoja eli vierusmatriisin jokainen arvo on 0 tai 1. Nyt V^n kertoo, montako n:n pituista polkua eri solmuista on toisiinsa.

Esimerkiksi verkon



vierusmatriisi on

$$V = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

Nyt matriisi V^3 kertoo, montako 3:n pituista polkua solmuista on toisiinsa:

$$V^3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

Esimerkiksi $(V^3)_{1,5} = 1$, koska solmusta 1 pääsee solmuun 5 kulkemalla polkua $1 \to 4 \to 2 \to 5$. Vastaavasti $(V^3)_{2,2} = 2$, koska solmusta 2 pääsee itseensä kulkemalla polkuja $2 \to 1 \to 4 \to 2$ sekä $2 \to 6 \to 3 \to 2$.

Yhdistämällä tämän nopeaan potenssilaskuun k:n pituisten polkujen määrät n:n solmun verkossa pystyy laskemaan ajassa $O(n^3 \log k)$.

Samantapaista ideaa voi käyttää myös laskemaan painotetussa verkossa, mikä on lyhin k kaaren pituinen polku kunkin solmun välillä. Tämän saavuttamiseksi riittää muuttaa matriisikertolaskun kaavassa yhteenlasku minimiksi ja kertolasku yhteenlaskuksi, jolloin kaavasta tulee

$$(AB)_{i,j} = \min_{k=1}^{n} A_{i,k} + B_{k,j}.$$

Tämä on lähellä Floyd-Warshallin algoritmissa käytettyä kaavaa: ainoana erona on, että halutaan laskea lyhimmän tarkalleen k kaarta sisältävän polun pituus eikä minkä tahansa lyhimmän polun pituus.

Luku 30

Nim-peli

Nim-peli on kahden pelaajan peli, jossa pelaajat poistavat vuorotellen tikkuja kasoista. Vuorossa olevan pelaajan tulee valita yksi kasoista ja poistaa siitä mikä tahansa määrä tikkuja. Pelin voittaja on se, joka onnistuu poistamaan viimeisen tikun. Seuraavassa on esimerkki pelin kulusta:

- Kasoja on 3 ja tikkumäärät ovat aluksi $[12,5,9]$.
- Uolevi poistaa 10 tikkua kasasta 1, ja tilanne on $[2,5,9]$.
- Maija poistaa 5 tikkua kasasta 2, ja tilanne on $[2,0,9]$.
- Uolevi poistaa 6 tikkua kasasta 3, ja tilanne on $[2,0,3]$.
- Maija poistaa 2 tikkua kasasta 1, ja tilanne on $[0,0,3]$.
- Uolevi poistaa 2 tikkua kasasta 3, ja tilanne on $[0,0,1]$.
- Maija poistaa 1 tikun kasasta 3 ja voittaa pelin.

Tämä luku esittelee voittostrategian nim-peliin. Sen avulla voi ratkaista monta muutakin peliä, jotka eivät ulkoisesti muistuta nim-peliä.

30.1 Voittostrategia

Voittostrategian etsimisessä tehtävänä on selvittää, mitkä ovat pelin voitto- ja häviötilat ja miten niistä pääsee toisiinsa.

Tila $[0,0,\ldots,0]$ on selvästi häviötila. Jos taas kasoja on tarkalleen yksi, tämä tila on voittotila, koska pelaaja pystyy poistamaan kaikki tikut. Jos kasoja on kaksi ja molemmissa on saman verran tikkuja, tämä tila on häviötila, koska vastustaja pystyy ”kumoamaan” jokaisen siirron poistamalla saman verran tikkuja toisesta kasasta. Muissa tilanteissa pelin analysointi alkaa olla vaikeampaa.

Yllättävää kyllä, nim-pelin tilan luonne paljastuu laskemalla xor-summa kaikista tikkujen määristä kasoissa. Jos xor-summa on 0, tila on häviötila, ja muuten tila on voittotila. Esimerkiksi tila $[12,5,9]$ on voittotila, koska 12 xor 5 xor 9 = 6. Mutta miten xor-summa liittyy nim-peliin?

Tarkastellaan ensin häviötiloja. Tila $[0,0,\ldots,0]$ on häviötila ja sen xor-summa todellakin on 0. Muissa häviötiloissa minkä tahansa siirron tulee tuottaa vastustajalle voittotila. Tämä toteutuu, koska tikkujen poistaminen yhdestä kasasta muuttaa yhtä arvoa xor-summassa. Niinpä jossain kohdassa bitistä 0 tulee bitti 1 tai bitistä 1 tulee bitti 0, joten uusi xor-summa ei ole enää 0.

Tarkastellaan sitten voittotiloja. Voittotilassa täytyy olla olemassa siirto, joka tuottaa vastustajalle häviötilan. Ideana on etsiä kasa, jonka tikkumäärän bittiesityksessä on ykkösbitti samassa kohdassa kuin xor-summan vasemmanpuoleisin ykkösbitti. Tästä kasasta voi aina poistaa niin monta tikkua, että uudeksi xor-summaksi tulee 0. Tarkemmin sanoen jos xor-summa on x ja valitussa kasassa on y tikkua, kasaan tulee jäädä siirron jälkeen x xor y tikkua.

Esimerkiksi tila $[12,5,9]$ xor-summa 6 muodostuu seuraavasti:

12	1010
5	0101
9	1001
6	0110

Tässä tapauksessa 5 tikun kasa on ainoa, jonka bittiesityksessä on ykkösbitti samassa kohdassa kuin xor-summan vasemmanpuoleisin ykkösbitti:

12	1010
5	0(1)01
9	1001
6	0(1)10

Kasan uudeksi sisällöksi täytyy saada 6 xor 5 = 3 tikkua, mikä onnistuu poistamalla 2 tikkua. Tämän jälkeen tilanne on:

12	1010
3	0011
9	1001
0	0000

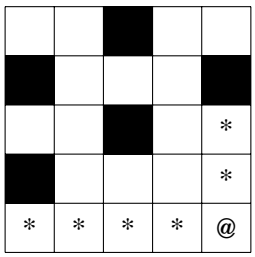
Huomaa, että x xor y on aina pienempi kuin y. Tämä johtuu siitä, että x:n vasemmanpuoleisinta ykkösbittiä vastaava ykkösbitti y:ssä muuttuu nollabitiksi ja mikään bitti sen vasemmalla puolella ei muutu.

30.2 Muunnos nim-peliksi

Nim-pelin hienoutena on, että monen muunkin pelin pystyy muuntamaan nim-peliksi tulkitsemalla pelin tilat sopivasti tikkukasoina. Tämän jälkeen pelin pystyy voittamaan samalla tavalla kuin alkuperäisen nim-pelin.

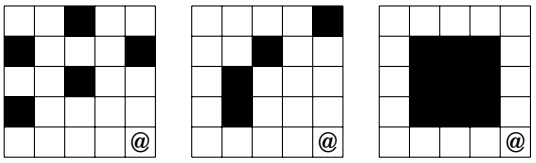
Tarkastellaan nyt sokkelopeliä, jossa pelaajat liikuttavat hahmoja sokkeloissa vuorotellen. Aluksi jokaisen sokkelon hahmo on oikeassa alakulmassa. Jokaisella siirrolla pelaaja saa siirtää yhdessä sokkelossa hahmoa minkä tahansa verran vasemmalle tai ylöspäin. Pelin voittaa se, joka tekee viimeisen siirron.

Seuraavassa on esimerkki sokkelon alkutilanteesta. Merkki @ näyttää hahmon nykyisen sijainnin, ja merkit * tarkoittavat ruutuja, joihin hahmo voi siirtyä.

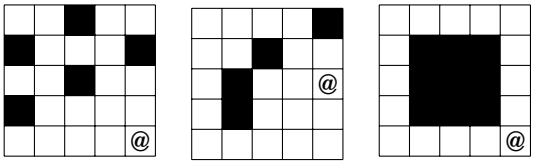


Tässä sokkelossa alussa on kuusi eri siirtomahdollisuutta: neljä mahdollista siirtoa vasemmalle ja kaksi mahdollista siirtoa ylöspäin.

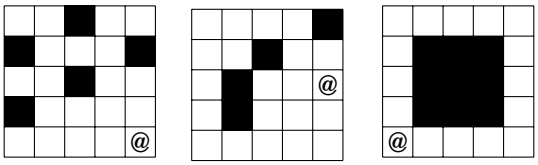
Tarkastellaan sitten peliä, jossa on kolme sokkeloa. Aloitustilanne on:



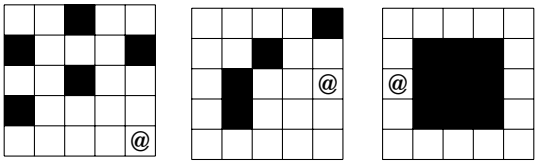
Ensimmäisellä siirrolla Uolevi siirtää sokkelon 2 hahmoa kaksi askelta ylöspäin:



Toisella siirrolla Maija siirtää sokkelon 3 hahmoa neljä askelta vasemmalle:



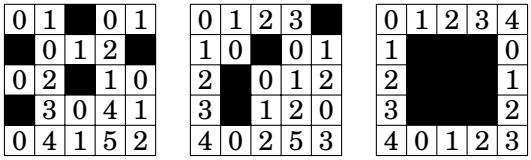
Kolmannella siirrolla Uolevi siirtää sokkelon 3 hahmoa kaksi askelta ylöspäin:



Peli jatkuu samaan tapaan, kunnes missään sokkelossa ei pysty tekemään siirtoa. Silloin viimeisen siirron tehnyt on pelin voittaja.

Sokkelopelin saa muutettua nim-peliksi kuvaamalla jokaisen sokkelon tikkukasaksi. Tämä onnistuu liittämällä jokaiseen sokkelon lattiaruutuun pienin ei-negatiivinen kokonaisluku, jota ei esiinny ruuduissa, johon kyseisestä ruudusta pääsee. Nämä nim-luvut tulevat vastaamaan nim-pelin kasojen tikkumääriä.

Esimerkiksi äskeisessä pelissä sokkeloihin liittyvät seuraavat nim-luvut:



Ideana on, että jos ruudussa on nim-luku x, siitä pääsee ruutuihin, joissa esiintyvät nim-luvut $0, 1, \ldots, x-1$. Jos taas ruudussa on nim-luku 0, siitä ei pääse ruutuun, jossa olisi nim-luku 0. Niinpä ruuduissa olevat nim-luvut vastaavat nim-pelin kasojen tikkumääriä. Esimerkiksi pelin alkutilanteen xor-summa on 2 xor 3 xor 3 = 2, joten pelin aloittajalla on voittostrategia.

Tässä on yksi tärkeä ero verrattuna alkuperäiseen nim-peliin: jos ruudussa on nim-luku x, siitä voi päästä ruutuun, jossa on suurempi nim-luku kuin x. Kasan tikkujen määrä voi siis ”kasvaa” joissain tilanteissa. Tällä ei ole kuitenkaan vaikutusta peliin, koska vastustaja pystyy aina peruuttamaan tällaisen siirron ja palauttamaan nim-luvun samaksi kuin se oli ennen siirtoa.

Tässä esitetyllä tavalla minkä tahansa vastaavanlaisen kahden pelaajan pelin voi muuttaa nim-peliksi kuvaamalla jokaisen alipelin tilat nim-luvuiksi.

30.3 Uudet alipelit

Joskus peli on luonteeltaan sellainen, että sen aikana pelialue jakautuu osiin ja syntyy uusia alipelejä. Jos nämä pelit ovat toisistaan riippumattomia, niitä voi käsitellä erillisinä nim-peleinä.

Tarkastellaan esimerkiksi peliä, jossa lähtötilanteena on merkkijono. Pelin aikana merkkijono jakautuu osiin. Jokaisella siirrolla pelaaja poistaa yhdestä merkkijonosta osan, jonka molemmissa päissä on sama merkki. Tästä jäljelle jäävät osat jäävät peliin. Pelin voittaja on se, joka tekee viimeisen jaon.

Oletetaan esimerkiksi, että merkkijono on ABBACCB. Nyt peli voi edetä seuraavasti:

- Uolevi poistaa merkkijonosta ABBACCB osan CC. Jäljelle jäävät merkkijonot ABBA ja B.
- Maija poistaa merkkijonosta ABBA osan BB. Nyt jäljellä ovat A, A ja B.
- Uolevi ei pysty tekemään mitään, joten Maija on voittaja.

Pelin analyysissa on ideana käydä läpi jokaisesta pelin aikana mahdollisesti esiintyvästä merkkijonosta kaikki mahdolliset siirrot. Tällä tavalla merkkijonoille saa laskettua nim-pelin tikkukasoja vastaavat luvut. Ensinnäkin jos merkkijonosta ei ole mahdollista poistaa mitään, sen nim-luku on 0. Muuten merkkijonon nim-luku on pienin ei-negatiivinen kokonaisluku, jota ei esiinny sen alipeleissä.

Esimerkiksi merkkijono ABBACCB mahdollistaa viisi siirtoa:

1. ABBA pois, jää CCB
2. BB pois, jää A ja ACCB
3. CC pois, jää ABBA ja B
4. BACCB pois, jää AB
5. BBACCB pois, jää A

Merkkijonon ABBACCB nim-luvun laskemiseksi täytyy laskea jokaisen tapauksen nim-luku ja valita pienin ei-negatiivinen kokonaisluku, jota ei esiinny näissä nim-luvuissa. Tapausten nim-luvut pystyy laskemaan rekursiivisesti, ja jos tapaukseen liittyy kaksi merkkijonoa, niiden yhteinen nim-luku on xor-summa niiden omista nim-luvuista.

Tapauksen 1 nim-luku on 1, koska merkkijonosta CCB voi poistaa osan CC, jolloin tuloksena on merkkijono B, jonka nim-luku on 0.

Tapauksen 2 nim-luku on 1, koska merkkijonon A nim-luku on 0, merkkijonon ACCB nim-luku on 1, ja 0 xor 1 = 1.

Tapauksen 3 nim-luku on 1, koska merkkijonon ABBA nim-luku on 1, merkkijonon B nim-luku on 0, ja 1 xor 0 = 1.

Tapausten 4 ja 5 nim-luku on 0, koska merkkijonoista ei voi poistaa mitään.

Merkkijonosta ABBACCB syntyvissä alipeleissä esiintyy nim-lukuja 0 ja 1, joten merkkijonon oma nim-luku on 2. Niinpä tässä tilanteessa pelin aloittajalla on voittostrategia. Käytännössä aloittajan kannattaa tuottaa tapaus 4 tai 5, minkä jälkeen peli päättyy heti siihen, ettei vastustaja pysty tekemään mitään.

Luku 31

Merkkijonohajautus

Merkkijonojen hajautuksen avulla $O(n)$-aikaisen esikäsittelyn jälkeen pystyy tarkistamaan $O(1)$-ajassa mistä tahansa kahdesta merkkijonon osasta, ovatko ne samat. Ideana on vertailla merkkijonon osien kaikkien merkkien sijasta niiden hajautusarvoja, jotka ovat yksittäisiä lukuja.

31.1 Hajautusarvo

Oletetaan, että merkkijonon pituus on n merkkiä ja sen merkkien merkkikoodit ovat $s_0, s_1, \ldots, s_{n-1}$. Ideana on laskea merkkijonosta hajautusarvo kaavalla

$$(s_0 \cdot A^{n-1} + s_1 \cdot A^{n-2} + \cdots + s_{n-2} \cdot A^1 + s_{n-1} \cdot A^0) \bmod B.$$

Tässä A ja B ovat satunnaisesti valittuja kokonaislukuja. Kaavan tavoitteena on, että jokainen merkkijono saadaan muutettua yksittäiseksi luvuksi, joka on merkkijonon hajautusarvo.

Esimerkiksi merkkijonossa APINA merkkikoodit ovat 65, 80, 73, 78 ja 65. Jos $A = 97$ ja $B = 151$, hajautusarvoksi tulee

$$(65 \cdot 97^4 + 80 \cdot 97^3 + 73 \cdot 97^2 + 78 \cdot 97^1 + 65 \cdot 97^0) \bmod 151 = 78.$$

Käytännössä merkkijonon s hajautusarvon voi laskea näin:

```
h = 0;
for (int i = 0; i < n; i++) {
    h = h*A+s[i];
    h %= B;
}
```

Todellisessa sovelluksessa lukujen A ja B tulee olla suuria satunnaisesti valittuja lukuja, jotta kahdelle eri merkkijonolle ei tulisi samaa hajautusarvoa. Käytännössä riittää yleensä tallentaa hajautusarvot `long long` -tyyppisinä ja valita vakiot A ja B lähelle `int`-tyypin ylärajaa.

31.2 Esikäsittely

Äskeisellä tavalla muodostettu hajautusarvo on mahdollista laskea mille tahansa merkkijonon osalle $O(1)$-ajassa sopivan esikäsittelyn jälkeen. Ideana on menetellä samoin kuin summataulukossa: jokaiselle merkkijonon alkuosalle lasketaan hajautusarvo, ja minkä tahansa osajonon hajautusarvo saadaan laskettua tehokkaasti näiden avulla.

Oletetaan esimerkiksi, että meillä on tiedossa merkkijonon ISOAPINATALO jokaisen alkuosan hajautusarvo. Tästä pystymme laskemaan merkkijonon APINA hajautusarvon ottamalla lähtökohdaksi merkkijonon ISOAPINA hajautusarvon ja poistamalla siitä merkkijonon ISO hajautusarvon sopivasti kerrottuna.

Jos aiemman mukaisesti $A = 97$ ja $B = 151$, merkkijonon ISOAPINA hajautusarvo on 116 ja merkkijonon ISO hajautusarvo on 85. Jos merkkijonon APINA hajautusarvo on x, täytyy päteä $(85 \cdot A^5 + x) \bmod B = 116$, koska APINA tuo merkkijonon ISO perään 5 uutta merkkiä. Toisin sanoen $x = (116 - 85 \cdot A^5) \bmod B = 78$, mikä vastaa aiemmin laskettua arvoa.

Yleisemmin jos p_k sisältää alkuosan hajautusarvon merkistä s_0 merkkiin s_k ja haluamme laskea hajautusarvon merkistä s_a merkkiin s_b, tämä onnistuu kaavalla $(p_b - p_{a-1} A^{b-a+1}) \bmod B$. Käytännön toteutus on helpoin laskemalla etukäteen sekä kaikki alkuosien hajautusarvot että A:n potenssit modulo B:

```
p[0] = s[0];
x[0] = 1;
for (int i = 1; i < n; i++) {
    p[i] = (p[i-1]*A+s[i])%B;
    x[i] = (x[i-1]*A)%B;
}
```

Tämän jälkeen funktio `h` palauttaa hajautusarvon merkistä a merkkiin b:

```
ll h(int a, int b) {
    ll t = p[b];
    if (a > 0) t -= (p[a-1]*x[b-a+1])%B;
    if (t < 0) t += B;
    return t;
}
```

31.3 Esimerkki

Tehtävä: Toistot

Sinulle on annettu merkkijono, ja tehtäväsi on etsiä mahdollisimman lyhyt merkkijono, jota toistamalla saat alkuperäisen merkkijonon. Esimerkiksi jos merkkijono on APINAAPINAAPINA, vastaus on APINA. Jos taas merkkijono on BANAANI, vastaus on BANAANI.

Tehtävän voi ratkaista suoraviivaisesti merkkijonojen hajautuksen avulla. Kun merkkijonon pituus on n merkkiä, ideana on käydä läpi kaikki mahdolliset toiston pituudet välillä $1,2,\ldots,n$, jotka jakavat luvun n tasan. Jokaisen toiston pituuden kohdalla riittää verrata, onko jokainen toisto-osa sama kuin ensimmäinen toisto-osa.

```
for (int i = 1; i <= n; i++) {
    if (n%i != 0) continue;
    bool ok = true;
    for (int j = i; j < n; j += i) {
        if (h(0, i-1) != h(j, j+i-1)) {
            ok = false;
            break;
        }
    }
    if (ok) {
        cout << s.substr(0, i) << "\n";
        break;
    }
}
```

Algoritmin aikavaativuus on $O(n \log n)$, koska sisemmän silmukan suorituskertojen määrä on enintään harmoninen summa

$$n + n/2 + n/3 + \cdots + n/n = O(n \log n).$$

31.4 Törmäykset

Hajautusta käyttävän algoritmin riskinä ovat *törmäykset*: erilaisia hajautusarvoja on vähemmän kuin erilaisia merkkijonoja, minkä vuoksi väistämättä kahdelle eri merkkijonolle voi tulla sama hajautusarvo. Niinpä algoritmi voi pitää kahta merkkijonoa samana, vaikka ne todellisuudessa olisivat eri merkkijonot.

Tästä riskistä huolimatta hajautus toimii lähes aina hyvin, kunhan vakiot A ja B ovat suuria lukuja ja ne on valittu satunnaisesti. Hyvä tapa valita vakiot on näpytellä umpimähkään yhdeksän numeroa. Tällöin vakiot ovat riittävän suuria ja ne eivät ole varmasti yleisesti käytettyjä lukuja (esim. muotoa 2^x tai $10^9 + 7$), joita vastaan tehtävässä voi olla erikoistestejä.

Luku 32

Z-algoritmi

Z-algoritmi muodostaa taulukon, jossa jokaisesta merkkijonon loppuosasta lukee, kuinka monta yhteistä merkkiä loppuosalla ja merkkijonolla on alusta laskien. Esimerkiksi merkkijonosta ABABABCABA (s) syntyy seuraava taulukko z:

i	0	1	2	3	4	5	6	7	8	9
s[i]	A	B	A	B	A	B	C	A	B	A
z[i]	–	0	4	0	2	0	0	3	0	1

Esimerkiksi luku 4 kohdassa z[2] johtuu siitä, että loppuosassa ABABCABA ensimmäiset 4 merkkiä täsmäävät merkkijonoon ABABABCABA, mutta sitten loppuosassa tulee merkki C, vaikka merkkijonossa on merkki A.

Suoraviivainen tapa muodostaa taulukko on verrata jokaisen loppuosan kohdalla loppuosaa ja merkkijonoa toisiinsa merkki kerrallaan. Tämän menetelmän aikavaativuus on $O(n^2)$, koska jokainen vertailu vie aikaa $O(n)$. Z-algoritmi muodostaa kuitenkin taulukon käyttäen aikaa yhteensä vain $O(n)$.

32.1 Algoritmi

Z-algoritmin ideana on täyttää taulukko vasemmalta oikealle ja pitää joka vaiheessa yllä lukuväliä $[a,b]$. Tämä vastaa sellaista aiempaa loppuosaa, joka alkaa indeksistä a, täsmää merkkijonoon indeksiin b asti ja b on mahdollisimman suuri. Merkkijonon ABABABCABA tapauksessa välit ovat:

i	0	1	2	3	4	5	6	7	8	9
s[i]	A	B	A	B	A	B	C	A	B	A
z[i]	–	0	4	0	2	0	0	3	0	1
$[a,b]$	[0,0]	[0,0]	[2,5]	[2,5]	[2,5]	[2,5]	[2,5]	[7,9]	[7,9]	[7,9]

Välin $[a,b]$ hyötynä on, että sen avulla taulukon z seuraavan arvon $z[i]$ voi laskea tehokkaasti. Jos $i > b$ eli minkään loppuosan täsmäävä osa ei ole jatkunut näin pitkälle, uusi z-arvo lasketaan alusta alkaen merkki kerrallaan. Muuten käytetään hyväksi arvoa z[$i-a$], joka kertoo suoraan, miten pitkälle nykyinen loppuosa täsmää merkkijonoon rajaan b asti.

Jos $i+\text{z}[i-a] \leq b$ eli kohdan i loppuosa ei täsmää rajaan b asti, saadaan suoraan z[i] = z[$i-a$]. Jos taas $i+\text{z}[i-a] > b$ eli kohdan i loppuosa täsmää ainakin rajaan

b asti, z-arvo lasketaan ottamalla pohjaksi arvo $b-i+1$ ja vertaamalla sitten b:n jälkeistä osaa merkki kerrallaan. Kaikissa tapauksissa väli $[a,b]$ muuttuu, jos nykyinen loppuosa täsmää aiempaa pidemmälle.

Tarkastellaan Z-algoritmin toimintaa yllä olevassa esimerkissä. Arvon z[3] laskemisessa välinä on $[2,5]$ ja $3 \leq 5$, eli voimme hyödyntää aiempaa tietoa. Koska $z[3-2] = 0$, niin myös $z[3] = 0$. Arvon z[4] laskemisessa taas $z[4-2] = 4$, joten loppuosa täsmää kohtaan 5 asti ja täytyy tutkia merkkejä kohdasta 6 alkaen. Kuitenkin kohdassa 6 on C, joten loppuosa ei täsmää pidemmälle ja $z[4] = 2$.

Z-algoritmin aikavaativuus on $O(n)$, koska se aloittaa merkkien vertaamisen yksi kerrallaan aina merkkijonon b:n jälkeisestä osasta. Aina kun merkki täsmää, b siirtyy eteenpäin eikä tätä merkkiä tarvitse verrata enää koskaan.

32.2 Toteutus

Seuraava koodi toteuttaa Z-algoritmin aiemman kuvauksen mukaisesti:

```
int a = 0, b = 0;
for (int i = 1; i < n; i++) {
    if (i > b) {
        z[i] = 0;
        for (int j = i; j < n; j++) {
            if (s[j-i] == s[j]) z[i]++;
            else break;
        }
    } else if (i+z[i-a] <= b) {
        z[i] = z[i-a];
    } else {
        z[i] = b-i+1;
        for (int j = b+1; j < n; j++) {
            if (s[j-i] == s[j]) z[i]++;
            else break;
        }
     }
     if (i+z[i]-1 > b) {
         a = i; b = i+z[i]-1;
     }
}
```

32.3 Käyttö

Z-algoritmin avulla voi etsiä tehokkaasti merkkijonon x esiintymiä merkkijonossa s. Ideana on muodostaa merkkijono x#s, jossa # on merkkijonoissa esiintymätön erikoismerkki. Esimerkiksi jos merkkijonosta KALAKUKKO etsitään merkkijonoa LAKU, kokonaismerkkijono on LAKU#KALAKUKKO. Nyt z-taulukon sisältö #:n jälkeen kertoo jokaisesta s:n kohdasta, miten monta x:n merkkiä täsmää siihen kohtaan. Jos z-arvo on x:n pituus, x esiintyy siinä kohdassa merkkijonoa s. Tämän aikavaativuus on $O(n+m)$, missä n on s:n pituus ja m on x:n pituus.

Luku 33

Loppuosataulukko

Merkkijonon *loppuosataulukko* sisältää viittaukset merkkijonon loppuosiin aakkosjärjestyksessä. Esimerkiksi merkkijonon BANAANI loppuosat ja niiden alkukohtien indeksit ovat:

loppuosa	indeksi
BANAANI	0
ANAANI	1
NAANI	2
AANI	3
ANI	4
NI	5
I	6

Aakkosjärjestyksessä loppuosat ovat:

loppuosa	indeksi
AANI	3
ANAANI	1
ANI	4
BANAANI	0
I	6
NAANI	2
NI	5

Tästä saadaan merkkijonon BANAANI loppuosataulukko `t`:

i	0	1	2	3	4	5	6
`t`$[i]$	3	1	4	0	6	2	5

33.1 Muodostus

Yksinkertaisin tapa muodostaa loppuosataulukko on järjestää merkkijonon indeksit sisältävä taulukko niin, että vertailufunktio vertaa indeksien osoittamissa kohdissa olevia loppuosia toisiinsa. Tämän menetelmän aikavaativuus on kuitenkin $O(n^2 \log n)$, koska järjestämisen runko vie aikaa $O(n \log n)$ ja kahden loppuosan vertailu voi viedä aikaa $O(n)$. Seuraavassa on kaksi kisoihin soveltuvaa menetelmää, joiden aikavaativuus on vain $O(n \log^2 n)$.

Menetelmä 1

Ideana on vertailla ensin merkkijonon 1:n pituisia osajonoja, sitten 2:n pituisia, sitten 4:n pituisia, sitten 8:n pituisia jne., kunnes osajonon pituus ylittää merkkijonon pituuden. Jokaisessa vaiheessa peräkkäisistä osajonoista muodostetaan pari, jossa on osajonon alkuosan ja loppuosan koodi. Tämän jälkeen parit järjestetään ja niistä muodostetaan uudet koodit pidemmille osajonoille.

Tarkastellaan esimerkiksi merkkijonon BANAANI käsittelyä. Ensimmäisessä vaiheessa 1:n pituisten osajonojen koodeina voi käyttää suoraan merkkikoodeja:

B	A	N	A	A	N	I
66	65	78	65	65	78	73

Tämän jälkeen muodostetaan pareja, joista jokainen koodaa 2:n pituisen osajonon. Jokainen pari sisältää kahden vierekkäisen merkin koodin:

B	A	N	A	A	N	I
$(66,65)$	$(65,78)$	$(78,65)$	$(65,65)$	$(65,78)$	$(78,73)$	$(73,-1)$

Huomaa, että viimeisessä parissa toinen osa on -1, koska tämä menee merkkijonon ulkopuolelle. Seuraavaksi parit koodataan yksittäisiksi kokonaisluvuiksi niiden järjestyksen mukaan:

pari	$(65,65)$	$(65,78)$	$(66,65)$	$(73,-1)$	$(78,65)$	$(78,73)$
koodi	0	1	2	3	4	5

Tuloksena on koodaus 2:n pituisille osajonoille:

B	A	N	A	A	N	I
2	1	4	0	1	5	3

Osajono AN esiintyy merkkijonossa kahdesti, joten myös sitä vastaava koodi 1 esiintyy kaksi kertaa. Kaikki muut 2:n pituiset osajonot ovat eri osajonoja, joten niillä on eri koodi.

Tämän jälkeen muodostetaan pareja, joista jokainen koodaa 4:n pituisen osajonon. Esimerkiksi osajono BANA muodostuu osista BA ja NA, joiden koodit ovat 2 ja 4. Osajonoja vastaavat parit ovat seuraavat:

B	A	N	A	A	N	I
$(2,4)$	$(1,0)$	$(4,1)$	$(0,5)$	$(1,3)$	$(5,-1)$	$(3,-1)$

Näistä pareista syntyvät seuraavat koodit:

B	A	N	A	A	N	I
3	1	5	0	2	6	4

Nyt jokaisella osajonolla on eri koodi, minkä vuoksi pidempien osajonojen käsittely ei enää muuta tilannetta. Tästä taulukosta saa loppuosataulukon muuttamalla sen käänteiseksi: kohdassa 0 on 3, joten kohtaan 3 tulee 0, kohdassa 4 on 2, joten kohtaan 2 tulee 4 jne. Tuloksena on loppuosataulukko:

i	0	1	2	3	4	5	6
t[i]	3	1	4	0	6	2	5

Menetelmän aikavaativuus on $O(n \log^2 n)$, koska parit muodostetaan $O(\log n)$ kertaa ja jokainen muodostus vie aikaa $O(n \log n)$ järjestämisen takia.

Menetelmä 2

Toinen menetelmä on tehostaa suoraviivaista loppuosien järjestämistä käyttämällä merkkijonojen hajautusta ja binäärihakua. Tämän avulla kahden merkkijonon vertailun saa tehtyä ajassa $O(\log n)$: etsitään viimeinen kohta, johon asti merkkijonot ovat samat, ja vertaillaan sen jälkeistä merkkiä. Tämän ansiosta aikavaativuudeksi tulee $O(n \log^2 n)$.

33.2 Toteutus

Yllä esitetyistä menetelmistä ensimmäinen on käytännössä parempi, koska se on helpompi toteuttaa ja se ei perustu hajautukseen vaan toimii varmasti. Seuraavassa on menetelmän toteutus:

```
for (int i = 0; i < n; i++) k[i] = s[i];
for (int x = 1; x <= n; x *= 2) {
    vector<pair<pair<int,int>,int>> v;
    for (int i = 0; i < n; i++) {
        int u = (i+x < n) ? k[i+x] : -1;
        v.push_back(make_pair(make_pair(k[i],u),i));
    }
    sort(v.begin(), v.end());
    int c = 0;
    for (int i = 0; i < n; i++) {
        if (i != 0 && v[i-1].first != v[i].first) c++;
        k[v[i].second] = c;
    }
    if (c == n-1) break;
}
for (int i = 0; i < n; i++) t[k[i]] = i;
```

Tässä `s` on käsiteltävä merkkijono, n merkkijonon pituus, `k` sisältää osajonojen koodit algoritmin aikana ja `t` on lopullinen loppuosataulukko.

33.3 Käyttö

Loppuosataulukolla voi Z-algoritmin tavoin etsiä merkkijonon `x` alkuosia ja esiintymiä merkkijonossa `s`. Kuitenkin erona Z-algoritmiin merkkijonoa `x` ei tarvitse päättää ennen loppuosataulukon luomista, vaan loppuosataulukosta voi etsiä mitä tahansa merkkijonoa tehokkaasti.

Ideana on käydä merkkijono `x` läpi merkki kerrallaan vasemmalta oikealle, ja pitää yllä loppuosataulukon väliä, joka täsmää nykyiseen merkkijonon `x` alkuosaan. Tämä onnistuu tehokkaasti binäärihaulla, koska loppuosat ovat järjestyksessä loppuosataulukossa. Aikavaativuus on loppuosataulukon muodostamisen jälkeen $O(m \log n)$, missä n on `s`:n pituus ja m on `x`:n pituus.

Luku 34

Eukleideen algoritmi

Eukleideen algoritmi tuli tutuksi jo kirjan alkuosassa käyttötarkoituksena lukujen a ja b suurimman yhteisen tekijän syt(a,b) laskeminen. Nyt on aika tutustua algoritmin laajennettuun muotoon ja perustella algoritmin toiminta.

Laajennettu Eukleideen algoritmi etsii lukujen a ja b suurimman yhteisen tekijän lisäksi luvut x ja y, jotka toteuttavat yhtälön

$$ax + by = \text{syt}(a,b).$$

Esimerkiksi jos $a = 39$ ja $b = 15$, niin syt$(a,b) = 3$, $x = 2$ ja $y = -5$, koska

$$39 \cdot 2 + 15 \cdot (-5) = 3.$$

Huomaa, että on olemassa monta tapaa valita luvut x ja y, ja Eukleideen algoritmi tuottaa yhden tavoista.

34.1 Algoritmi

Eukleideen algoritmin syötteenä on kokonaisluvut a ja b. Algoritmin alussa $c_a = a$ ja $c_b = b$ ja joka vaiheessa c_a:sta tulee c_b ja c_b:stä tulee $c_a \% c_b$. Algoritmi päättyy, kun $c_b = 0$, jolloin c_a on syt(a,b).

Laajennettu algoritmi pitää lisäksi yllä tietoa, miten c_a ja c_b voidaan esittää a:n ja b:n avulla. Ideana on, että $c_a = x_a a + y_a b$ ja $c_b = x_b a + y_b b$, ja algoritmin päättyessä $x = x_a$ ja $y = y_a$. Joka vaiheessa x_a:sta tulee x_b ja y_a:sta tulee y_b. Uudet arvot x_b ja y_b lasketaan kaavoilla $x_b = x_a - (c_a/c_b)x_b$ ja $y_b = y_a - (c_a/c_b)y_b$.

Seuraava taulukko esittää algoritmin toimintaa, kun $a = 39$ ja $b = 15$.

c_a	c_b	c_a/c_b	$c_a \% c_b$	x_a	y_a	x_b	y_b
39	15	2	9	1	0	0	1
15	9	1	6	0	1	1	−2
9	6	1	3	1	−2	−1	3
6	3	2	0	−1	3	2	−5
3	0	–	–	2	−5	−5	13

Esimerkiksi kolmannella rivillä $x_a = 1$ ja $y_a = -2$, koska luvun $c_a = 9$ pystyy esittämään lukujen $a = 39$ ja $b = 15$ avulla laskemalla $1 \cdot 39 + (-2) \cdot 15$. Vastaavasti $x_b = -1$ ja $y_b = 3$, koska $c_b = 6 = (-1) \cdot 39 + 3 \cdot 15$.

34.2 Toteutus

Tässä on laajennetun Eukleideen algoritmin toteutus:

```
int syt(int ca, int cb, int xa, int xb) {
    if (cb == 0) {
        x = xa;
        y = (b == 0) ? 0 : (ca-a*xa)/b;
        return ca;
    }
    return syt(cb, ca%cb, xb, xa-(ca/cb)*xb);
}
```

Funktiolle annetaan parametrina luvut a, b, 1 ja 0, ja funktio palauttaa a:n ja b:n suurimman yhteisen tekijän syt(a,b). Lisäksi funktio asettaa muuttujiin x ja y arvot, jotka toteuttavat yhtälön

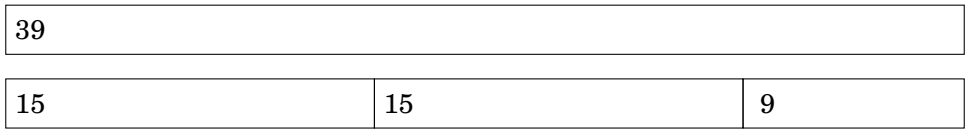
$$ax + by = \text{syt}(a,b).$$

Toisin kuin äskeisessä esimerkissä, funktio pitää yllä vain arvoja x_a ja x_b eikä arvoja y_a ja y_b. Tämä johtuu siitä, että y_a:n ja y_b:n voi laskea x_a:n ja x_b:n perusteella. Tämä on tarpeen algoritmin lopussa, jotta y:n arvo saadaan laskettua.

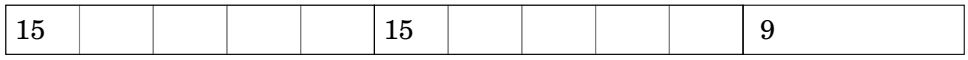
Eukleideen algoritmia voi käyttää myös silloin, kun a ja/tai b ovat negatiivisia. Tällöin tuloksena oleva syt(a,b) on negatiivinen. Jos tämä ei ole haluttu tulos, x:n, y:n ja syt(a,b):n voi kertoa -1:llä.

34.3 Analyysi

Keskeinen kysymys Eukleideen algoritmin toiminnassa on, miksi a:n ja b:n suurin yhteinen tekijä on sama kuin b:n ja $a\%b$:n suurin yhteinen tekijä. Tarkastellaan esimerkiksi algoritmin alkutilannetta, jossa $a = 39$ ja $b = 15$. Algoritmi jakaa 39:n kahteen 15:n kokoiseen osaan ja lisäksi yli jää 9:



Nyt koska syt$(39,15)$ on 15:n tekijä, sen täytyy jakaa 15:n kokoiset osat tasan:



Mutta syt(39, 15) on myös 39:n tekijä, joten sen täytyy jakaa myös 9 tasan, jotta se jakaisi tasan kokonaisuuden $15 + 15 + 9 = 39$:

15	15	9		

Niinpä syt(39, 15) on suurin luku, joka jakaa tasan 15:n ja 9:n – eli syt(15, 9).

Voidaan osoittaa, että Eukleideen algoritmi on hitaimmillaan silloin, kun syötteenä on kaksi peräkkäistä Fibonaccin lukua. Tällöin algoritmi käy läpi kaikki pienemmät Fibonaccin luvut yksi kerrallaan. Esimerkiksi kun $a = 55$ ja $b = 34$, algoritmi toimii seuraavasti:

c_a	c_b	c_a / c_b	$c_a \% c_b$	x_a	y_a	x_b	y_b
55	34	1	21	1	0	0	1
34	21	1	13	0	1	1	−1
21	13	1	8	1	−1	−1	2
13	8	1	5	−1	2	2	−3
8	5	1	3	2	−3	−3	5
5	3	1	2	−3	5	5	−8
3	2	1	1	5	−8	−8	13
2	1	2	0	−8	13	13	−21
1	0	–	–	13	−21	−34	55

Fibonaccin luvut kasvavat eksponentiaalisesti, minkä seurauksena Eukleideen algoritmin aikavaativuus on logaritminen.

Luku 35

Modulon jakolasku

Yhteenlasku, vähennyslasku ja kertolasku ovat helppoja toteuttaa moduloilla: riittää laskea ne samaan tapaan kuin yleensäkin ja ottaa lopuksi jakojäännös. Modulon jakolaskun määrittely on vaikeampaa, mutta silti mahdollista. Ideana on esittää jakolasku a/b muodossa ab^{-1}, jossa b^{-1} on luvun b modulon käänteisluku. Näin jakolasku muuttuu tutuksi kertolaskuksi.

Ainoa ongelma on, kuinka luvusta b saa käänteisluvun b^{-1}. Käänteisluvun tulisi toteuttaa yhtälö

$$(bb^{-1})\%M = 1,$$

missä M on haluttu modulo. Esimerkiksi jos $b = 7$ ja $M = 10$, niin voidaan valita $b^{-1} = 3$, koska $(7 \cdot 3)\%10 = 1$. Tämän jälkeen esimerkiksi lasku $35/7 = 5$ tapahtuu moduloilla $(35 \cdot 3)\%10 = 5$.

Toisin kuin tavallinen jakolasku, modulon jakolasku ei ole aina mahdollinen, koska käänteisluku on olemassa vain silloin, kun b ja M ovat suhteellisia alkulukuja eli $\text{syt}(b, M) = 1$. Esimerkiksi jos $b = 2$ ja $M = 10$, $\text{syt}(b, M) = 2$ eikä käänteislukua b^{-1} ole olemassa. Tämä johtuu siitä, että minkään luvun b^{-1} kertominen 2:lla ei voi tuottaa paritonta lukua, jonka jakojäännös 10:llä olisi 1.

Tämä luku esittelee kaksi menetelmää modulon käänteisluvun etsimiseen: Eukleideen algoritmi sekä Eulerin teoreema.

35.1 Eukleideen algoritmi

Eukleideen algoritmi soveltuu modulon käänteisluvun etsimiseen, koska kaavan

$$(bb^{-1})\%M = 1$$

voi muuttaa muotoon

$$Mx + bb^{-1} = 1,$$

joka vastaa Eukleideen algoritmin tuottamaa esitystä

$$ax + by = \text{syt}(a, b).$$

valitsemalla $a = M$ ja $y = b^{-1}$. Esimerkiksi tapauksessa $b = 7$ ja $M = 10$ Eukleideen algoritmi tuottaa tuloksen

$$10 \cdot (-2) + 7 \cdot 3 = 1,$$

josta saadaan käänteisluvuksi $y = b^{-1} = 3$.

35.2 Eulerin lause

Toinen tapa etsiä modulon käänteisluku on käyttää Eulerin lausetta

$$(x^{\varphi(M)}) \% M = 1,$$

missä $\varphi(n)$ tarkoittaa, montako lukua välillä $1 \ldots n$ on suhteellisia alkulukuja n:n kanssa. Esimerkiksi $\varphi(10) = 4$, koska luvut 1, 3, 7 ja 9 ovat suhteellisia alkulukuja 10:n kanssa.

Modulon käänteisluvun kaavan

$$(bb^{-1}) \% M = 1$$

saa vastaamaan tähän muotoon merkitsemällä

$$(bb^{\varphi(M)-1}) \% M = 1,$$

minkä seurauksena

$$b^{-1} = b^{\varphi(M)-1}.$$

Esimerkiksi kun $b = 7$ ja $M = 10$, käänteisluvuksi tulee $7^{\varphi(10)-1} = 7^3 = 343$, joka on modulo 10 sama kuin 3.

Eulerin teoreema on erityisen kätevä silloin, kun M on alkuluku, jolloin $\varphi(M) = M - 1$. Yleisessä tapauksessa arvon $\varphi(n)$ saa laskettua kaavalla

$$\varphi(n) = n(1 - 1/p_1)(1 - 1/p_2) \cdots (1 - 1/p_m),$$

jossa luvut $p_1, p_2, \ldots, p_m$ ovat luvun n eri alkutekijät. Esimerkiksi

$$\varphi(20) = 20(1 - 1/2)(1 - 1/5) = 8.$$

Tämän kaavan voi ymmärtää *todennäköisyyksien* kautta: esimerkiksi tapauksessa $n = 20$ väliltä $1 \ldots 20$ satunnaisesti valittu luku ei ole parillinen todennäköisyydellä $1 - 1/2$ ja ei ole 5:llä jaollinen todennäköisyydellä $1 - 1/5$.

35.3 Esimerkki

Tehtävä: Ryhmän valinta

Huoneessa on n henkilöä, ja sinun tulee muodostaa heistä m henkilön ryhmä. Kuinka monta vaihtoehtoa tähän on? Voit olettaa, että $0 \le m \le n \le 10^5$. Ilmoita tulos modulo 10^9+7.

Tehtävä ratkeaa sinänsä suoraan kaavalla $\binom{n}{m}$, mutta ongelmana on lukujen n ja m suuruus. Aiemmin esitetyn dynaamisen ohjelmoinnin algoritmin aikavaativuus on $O(nm)$, joten se on tähän liian hidas.

Ratkaisu on käyttää kaavaa

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

ja laskea jakolaskut käyttäen modulon käänteislukua. Luku 10^9+7 on alkuluku, joten Eulerin lause sopii hyvin tähän tehtävään.

Seuraavassa koodissa funktio `pot` laskee tehokkaasti potenssilaskun x^n modulo M ja funktio `inv` laskee x:n käänteisluvun.

```
#define ll long long
#define M 1000000007

ll pot(ll x, int n) {
    if (n == 0) return 1;
    if (n%2 == 1) return (pot(x,n-1)*x)%M;
    ll t = pot(x,n/2);
    return (t*t)%M;
}

ll inv(ll x) {
    return pot(x, M-2);
}
```

Tämän jälkeen riittää koodata kaava ohjelmaan:

```
ll t = 1;
for (int i = 1; i <= n; i++) t = (t*i)%M;
for (int i = 1; i <= m; i++) t = (t*inv(i))%M;
for (int i = 1; i <= n-m; i++) t = (t*inv(i))%M;
cout << t << "\n";
```