Kisakoodarin käsikirja

Antti Laaksonen

10. heinäkuuta 2016

Sisältö

Alkusanat

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

Johdanto

Kisakoodaus eroaa monella tavalla tavallisesta ohjelmoinnista. Koodit ovat lyhyitä, syötteet ja tulosteet on määritelty tarkasti, eikä koodeja tarvitse ymmärtää eikä jatkokehittää kisan jälkeen. Lisäksi kisoissa on yleensä hyvin vähän aikaa ohjelmointiin. Niinpä monet ohjelmistokehityksen periaatteet sopivat huonosti kisakoodaukseen.

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 pystyy katsomaan jälkeenpäin muiden lähettämiä ratkaisuja, joista voi oppia paljon kisakoodauksesta.

1.1 Ohjelmointikieli

Tällä hetkellä yleisimmät koodauskisoissa käytetyt kielet ovat C++, Python ja Java. Esimerkiksi vuoden 2016 Google Code Jamissa 3000 parhaan osallistujan joukossa 73 % käytti C++:aa, 15 % käytti Pythonia ja 10 % käytti Javaa. Jotkut käyttivät myös useita näistä kielistä.

Monen mielestä C++ on paras valinta kisakoodauksen kieleksi, ja se on yleensä aina käytettävissä kisajärjestelmissä. C++:n etuja ovat, että sillä toteutettu koodi on hyvin tehokasta ja kielen standardikirjastoon kuuluu kattava valikoima valmiita tietorakenteita ja algoritmeja.

Paras ratkaisu on silti hallita useita kieliä ja tuntea niiden edut. Esimerkiksi jos tehtävässä täytyy käsitellä suuria kokonaislukuja, Python voi sopia ratkaisuun paremmin kuin C++, koska Pythonissa on sisäänrakennettuna suurten kokonaislukujen käsittely. Toisaalta tehtävät yritetään yleensä laatia niin, ettei tietyn kielen käyttämisestä ole merkittävää hyötyä.

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

1.2 Koodipohja

Tyypillinen C++-koodin pohja kisakoodausta varten näyttää seuraavalta:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // koodi tulee tähän
}
```

Koodin alussa oleva `#include`-rivi on GCC:n `g++`-kääntäjän tarjoama tapa sisällyttää kaikki standardikirjaston sisältö. Tämän ansiosta koodissa ei tarvitse erikseen sisällyttää tiedostoja `iostream`, `vector`, `algorithm`, jne., vaan ne kaikki ovat käytettävissä automaattisesti.

Seuraavana oleva `using`-rivi määrittää, että standardikirjaston sisältöä voi käyttää suoraan koodissa. Ilman `using`-riviä koodissa pitäisi kirjoittaa esimerkiksi `std::cout`, mutta `using`-rivin ansiosta riittää kirjoittaa `cout`.

Koodin voi kääntää esimerkiksi seuraavalla komennolla:

```
g++ -std=c++11 -O2 -Wall koodi.cpp -o koodi
```

Komento kääntää tiedoston `koodi.cpp` binääritiedostoksi `koodi`. Komento kääntää koodin C++11-standardin mukaisesti (`-std=c++11`), optimoi sitä käännöksen aikana (`-O2`) ja näyttää varoituksia mahdollisista virheistä (`-Wall`).

1.3 Syöte ja tuloste

Nykyään useimmissa kisoissa käytetään standardivirtoja syötteen lukemiseen ja tulostamiseen. C++:ssa standardivirrat ovat `cin` lukemiseen ja `cout` tulostamiseen. Tämän lisäksi voi käyttää C:n funktioita `scanf` ja `printf`.

1.3.1 `cin` ja `cout`

Ohjelmalle tuleva syöte muodostuu yleensä luvuista ja merkkijonoista, joiden välissä on välilyöntejä ja rivinvaihtoja. Niitä voi lukea `cin`-virrasta näin:

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

Tämä tapa toimii riippumatta siitä, miten luettavat tiedot on jaettu riveille ja missä kohdin syötettä on välilyöntejä.

Vastaavasti tulostaminen tapahtuu `cout`-virran kautta:

```
cout << c << "\n";
```

Jos syötteen tai tulosteen määrä on suuri, seuraavat rivit koodin alussa voivat nopeuttaa ohjelman toimintaa merkittävästi:

```
ios_base::sync_with_stdio(0);
cin.tie(0);
```

Huomaa myös, että rivinvaihto `"\n"` toimii tulostuksessa nopeammin kuin `endl`, koska `endl` aiheuttaa aina flush-operaation.

1.3.2 `scanf` ja `printf`

C++:ssa voi käyttää myös C:n funktioita `scanf` ja `printf` syötteen lukemiseen ja tulostamiseen. Nämä funktiot ovat joskus nopeampia kuin C++:n standardivirrat, mutta niiden käyttäminen on hieman hankalampaa.

Seuraava koodi lukee kaksi kokonaislukua syötteestä:

```
int a, b;
scanf("%d %d", &a, &b);
```

Seuraava koodi taas tulostaa kokonaisluvun:

```
printf("%d\n", c);
```

Yksi tilanne, jossa funktio `printf` on erityisen kätevä, on liukuluvun tulostaminen tietyllä tarkkuudella. Seuraava koodi tulostaa muuttujassa `x` olevan liukuluvun 9 desimaalin tarkkuudella:

```
printf("%.9f\n", x);
```

1.3.3 Rivin lukeminen

Joskus ohjelman täytyy lukea syötteestä kokonainen rivi tietoa välittämättä rivin välilyönneistä. Tämä onnistuu seuraavasti funktiolla `getline`:

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

1.3.4 Tiedostot

Joissakin kisajärjestelmissä syöte täytyy lukea tiedostosta ja tuloste täytyy kirjoittaa tiedostoon. Helppo ratkaisu tähän on kirjoittaa koodi tavallisesti standardivirtoja käyttäen, mutta kirjoittaa alkuun seuraavat rivit:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Tämän seurauksena koodi lukee syötteen tiedostosta ”input.txt” ja kirjoittaa tulosteen tiedostoon ”output.txt”.

1.4 Lukujen käsittely

Yleisin kisakoodauksessa tarvittava lukutyyppi on `int`, joka on useimmissa ympäristöissä 32-bittinen kokonaislukutyyppi, eli pienin ja suurin mahdollinen arvo ovat -2^{31} ja $2^{31}-1$ eli noin $-2 \cdot 10^9$ ja $2 \cdot 10^9$. Joskus kuitenkin tehtävissä esiintyy lukuja, jotka ovat `int`-tyypin arvoalueen ulkopuolella.

1.4.1 Suuret kokonaisluvut

Tyyppi `long long` on 64-bittinen kokonaislukutyyppi, jonka pienin ja suurin mahdollinen arvo ovat -2^{63} ja $2^{63}-1$ eli noin $-9 \cdot 10^{18}$ ja $9 \cdot 10^{18}$. Jos tehtävässä tarvitsee suuria kokonaislukuja, `long long` riittää yleensä.

Esimerkiksi seuraava koodi määrittelee `long long` -muuttujan:

```
long long x = 123456789123456789LL;
```

Luvun lopussa oleva merkintä `LL` ilmaisee, että luku on `long long` -tyyppinen.

Tyyppinimi `long long` on pitkä, minkä vuoksi tavallinen tapa on antaa sille lyhyempi nimi kuten `ll`:

```
typedef long long ll;
```

Tämän jälkeen `long long` -tyyppisen muuttujan voi määritellä näin:

```
ll x;
```

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

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

Vaikka muuttuja `b` on `long long` -tyyppinen, laskussa `a*a` molemmat osat ovat `int`-tyyppisiä ja myös laskun tulos on `int`-tyyppinen. Tämän vuoksi muuttujaan `b` ilmestyy väärä luku. Ongelman voi korjata vaihtamalla muuttujan `a` tyypiksi `long long` tai kirjoittamalla lasku muodossa `(long long)a*a`.

1.4.2 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 riittää käyttää tyyppejä `int` ja `long long`.

Luvun x jakojäännöstä M:llä merkitään $x \bmod M$. Esimerkiksi $12 \bmod 5 = 2$, koska 12:n jakojäännös 5:llä on 2.

Tärkeä modulon ominaisuus on, että yhteen-, vähennys- ja kertolaskussa modulon voi laskea ennen laskutoimitusta, eli seuraavat kaavat pätevät:

$$\begin{aligned}(a+b) \bmod M &= (a \bmod M + b \bmod M) \bmod M \\ (a-b) \bmod M &= (a \bmod M - b \bmod M) \bmod M \\ (a \cdot b) \bmod M &= (a \bmod M \cdot b \bmod M) \bmod M\end{aligned}$$

Näiden kaavojen ansiosta jokaisen laskun vaiheen jälkeen voi ottaa modulon eivätkä luvut kasva liian suuriksi.

Esimerkiksi seuraava koodi laskee luvun 1000 kertoman modulo M:

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

Yleensä on tarkoitus, että modulo olisi aina välillä $0 \ldots M-1$. Kuitenkin C++:ssa ja useimmissa muissa ohjelmointikielissä negatiivisen luvun modulo voi olla negatiivinen. Yksi ratkaisu ongelmaan on laskea ensin luvun modulo ja lisätä sitten M jos tulos on negatiivinen:

```
x = x%M;
if (x < 0) x += M;
```

Tämä on tarpeen kuitenkin vain silloin, kun modulo voi olla negatiivinen koodissa olevien vähennyslaskujen vuoksi.

1.4.3 Liukuluvut

Yleensä ohjelmointikisojen tehtävissä riittää käyttää kokonaislukuja. Esimerkiksi IOI:ssä on ollut käytäntönä, että tehtävät voi ratkaista ilman liukulukuja. Liukulukujen käyttämisessä on ongelmana, että kaikkia lukuja ei voi esittää tarkasti liukulukuina vaan tapahtuu pyöristysvirheitä.

Esimerkiksi seuraava koodi tuottaa yllättävän tuloksen:

```
double x = 0.3*3+0.1;
printf("%.20f\n", x);
```

Koodin tulostus on seuraava:

```
0.99999999999999988898
```

Pyöristysvirheen vuoksi muuttujan `x` sisällöksi tulee hieman alle 1, vaikka sen arvo tarkasti laskettuna olisi 1.

Liukulukuja on vaarallista vertailla ==-merkinnällä, koska vaikka luvut olisivat todellisuudessa samat, niissä voi olla pientä eroa pyöristysvirheiden vuok-

si. Parempi tapa vertailla liukulukuja on tulkita kaksi lukua samoiksi, jos niiden erona on ε, jossa ε on sopiva pieni luku.

Käytännössä vertailun voi toteuttaa seuraavan tyylisesti ($\varepsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a ja b ovat samat
}
```

1.5 Lyhennysmakrot

Kisakoodauksessa lyhyt koodi on tavoiteltava asia, koska koodi täytyy pystyä kirjoittamaan nopeasti. Jotkut kisakoodarit käyttävät lyhennysmakroja, joiden avulla koodin saa kirjoitettua lyhyemmin.

Esimerkiksi luokkien `vector` ja `pair` avuksi voi määritellä makrot

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

joiden avulla koodin

```
v.push_back(make_pair(a,b));
int c = v[i].first+v[i].second;
```

saa kirjoitettua lyhyemmin näin:

```
v.PB(MP(a,b));
int c = v[i].F+v[i].S;
```

Makrojen avulla voi myös lyhentää komentorakenteita. Esimerkiksi makron

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

avulla silmukan

```
for (int i = 1; i <= n; i++) check(i);
```

voi kirjoittaa lyhyemmin näin:

```
REP(i,1,n) check(i);
```

Luku 2

Aikavaativuus

Kisakoodauksessa oleellinen asia on algoritmien tehokkuus. Yleensä on helppoa suunnitella algoritmi, joka ratkaisee tehtävän hitaasti, mutta todellinen vaikeus piilee siinä, kuinka keksiä nopeasti toimiva algoritmi. Jos algoritmi on liian hidas, se tuottaa vain osan pisteistä tai ei pisteitä lainkaan.

Aikavaativuus (*time complexity*) on kätevä tapa arvioida, kuinka nopeasti algoritmi toimii. Se arvio algoritmin tehokkuutta funktiona, jonka parametrina on syötteen koko. Aikavaativuuden avulla algoritmista voi päätellä ennen koodaamista, onko se riittävän tehokas tehtävän ratkaisuun.

2.1 Laskusäännöt

Algoritmin aikavaativuus merkitään $O(\cdots)$, jossa kolmen pisteen tilalla on kaava, joka kuvaa algoritmin ajankäyttöä. Yleensä muuttuja n esittää syötteen kokoa. Esimerkiksi jos algoritmin syötteenä on taulukko lukuja, n on lukujen määrä, ja jos syötteenä on merkkijono, n on merkkijonon pituus.

Silmukat

Algoritmin ajankäyttö johtuu usein pohjimmiltaan silmukoista, jotka käyvät syötettä läpi. Mitä enemmän sisäkkäisiä silmukoita algoritmissa on, sitä hitaampi se on. Jos sisäkkäisiä silmukoita on k, aikavaativuus on $O(n^k)$.

Esimerkiksi seuraavan algoritmin aikavaativuus on $O(n)$:

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

Vastaavasti seuraavan algoritmin aikavaativuus on $O(n^2)$:

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

Suuruusluokka

Aikavaativuus ei kerro tarkasti, montako kertaa silmukan sisällä oleva koodi suoritetaan, vaan se kertoo vain suuruusluokan. Esimerkiksi seuraavissa esimerkeissä silmukat suoritetaan $3n$, $n+5$ ja $\lfloor n/2 \rfloor$ kertaa, mutta kunkin koodin aikavaativuus on sama $O(n)$.

```
for (int i = 1; i <= 3*n; i++) {
    // koodia
}
```

```
for (int i = 1; i <= n+5; i++) {
    // koodia
}
```

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

Seuraavan koodin aikavaativuus on $O(n^2)$, koska silmukoiden sisällä oleva koodi suoritetaan $1+2+\cdots+n = \frac{n(n+1)}{2}$ kertaa.

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

Peräkkäisyys

Jos koodissa on peräkkäisiä osia, 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 koodin aikavaativuus on $O(n^2)$.

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

Monta muuttujaa

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

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

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

Rekursio

Rekursiivisen funktion aikavaativuuden määrittää, 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}

Tämän perusteella kutsun g(n) aikavaativuus on

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

2.2 Vaativuusluokat

Aikavaativuuksien avulla algoritmeja voi ryhmitellä luokkiin, joissa olevilla algoritmeilla on sama aikavaativuus. Seuraavassa listassa on tärkeimpiä algoritmien aikavaativuuksia järjestyksessä nopeimmasta hitaimpaan.

$O(1)$ (vakioaikainen)

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

$O(\log n)$ (logaritminen)

Logaritminen aikavaativuus $O(\log n)$ syntyy usein siitä, että algoritmi puolittaa syötteen koon joka askeleella, koska logaritmi $\log_2 n$ ilmaisee, montako kertaa luku n täytyy puolittaa, ennen kuin tuloksena on 1.

$O(\sqrt{n})$ (neliöjuuri)

Aikavaativuus $O(\sqrt{n})$ on melko harvinainen ja sijoittuu vaativuuksien $O(\log n)$ ja $O(n)$ välimaastoon. Neliöjuuren erityinen ominaisuus on, että $\sqrt{n} = n/\sqrt{n}$, joten se osuu tietyllä tavalla syötteen puoliväliin.

$O(n)$ (lineaarinen)

Lineaarinen algoritmi käy syötteen läpi kiinteän määrän kertoja. Tämä on usein paras mahdollinen aikavaativuus, koska syöte täytyy käydä kokonaan läpi ainakin kerran, ennen kuin algoritmi voi ilmoittaa vastauksen.

$O(n \log n)$ (järjestäminen)

Aikavaativuus $O(n \log n)$ viittaa usein syötteen järjestämiseen, koska tehokkaat järjestämisalgoritmit toimivat ajassa $O(n \log n)$. Toinen mahdollisuus on, että algoritmi käyttää tietorakennetta, jonka operaatiot ovat $O(\log n)$-aikaisia.

$O(n^2)$ (neliöllinen)

Neliöllinen aikavaativuus $O(n^2)$ voi syntyä siitä, että algoritmissa on kaksi sisäkkäistä silmukkaa. Neliöllinen algoritmi voi käydä läpi kaikki tavat valita taulukosta kaksi alkiota.

$O(n^3)$ (kuutiollinen)

Kuutiollinen aikavaativuus $O(n^3)$ voi syntyä siitä, että algoritmissa on kolme sisäkkäistä silmukkaa. Kuutiollinen algoritmi voi käydä läpi kaikki tavat valita taulukosta kolme alkiota.

$O(2^n)$ **(osajoukot)**

Aikavaativuus $O(2^n)$ tarkoittaa usein, että algoritmi käy läpi kaikki syötteen osajoukot. Esimerkiksi lukujen $\{1,2,3\}$ osajoukot ovat $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1,2,3\}$, missä $\emptyset$ on tyhjä joukko.

$O(n!)$ **(permutaatiot)**

Aikavaativuus $O(n!)$ voi syntyä siitä, että algoritmi käy läpi kaikki syötteen permutaatiot. Esimerkiksi lukujen $\{1,2,3\}$ permutaatiot ovat $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ sekä $(3,2,1)$.

Algoritmin on polynominen, jos sen aikavaativuus on korkeintaan $O(n^k)$, kun k on vakio. Tässä mainituista aikavaativuuksista kaikki ovat polynomisia paitsi $O(2^n)$ ja $O(n!)$, jotka ovat eksponentiaalisia. Tavoitteena on yleensä saada aikaan polynominen algoritmi, koska eksponentiaalisen algoritmin ajankäyttö kasvaa räjähdysmäisesti syötteen koon n kasvaessa.

Useimmat tässä kirjassa esitettävät algoritmit ovat polynomisia. Silti on paljon ongelmia, joihin ei tunneta polynomista algoritmia eli ei mitään tehokasta ratkaisutapaa. Esimerkiksi NP-ongelmien joukko sisältää monia käytännössä esiintyviä ongelmia, joihin ei tiedetä polynomista algoritmia.

2.2.1 Nopeuden arviointi

Aikavaativuuden avulla voi arvioida *ennen* algoritmin koodaamista, onko algoritmi riittävän nopea tehtävän ratkaisuun. Arviota varten tehtävänannosta täytyy katsoa, miten suuria syötteitä algoritmin täytyy pystyä käsittelemään ja paljonko aikaa algoritmilla on käytössään.

Syötteen koon ja aikavaativuuden suhteen oppii kokemuksen kautta, mutta seuraavat arviot ovat hyviä lähtökohtia:

syötteen koko (n)	haluttu aikavaativuus
$n \leq 10^{18}$	$O(1)$ tai $O(\log n)$
$n \leq 10^{12}$	$O(\sqrt{n})$
$n \leq 10^6$	$O(n)$ tai $O(n \log n)$
$n \leq 5000$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$
$n \leq 10$	$O(n!)$

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. Optimoinnin avulla koodi voi nopeutua moninkertaisesti, vaikka aikavaativuus ei muuttuisikaan.

Algoritmin nopeuden arviointi on tärkeää aina ennen koodin toteuttamista, koska jos algoritmin idea on liian hidas, hyväkään toteutus ei pelasta asiaa.

2.2.2 Esimerkki

Tehtävän ratkaisuun on usein monia algoritmeja, joilla on eri aikavaativuudet. Tarkastellaan esimerkkinä seuraavaa tehtävää:

> *Tehtävä:* Annettuna on n lukua $x_1, x_2, \ldots, x_n$. Tehtäväsi on tarkastaa, onko lukujen joukossa kahta samaa lukua. Suunnittele tehokas algoritmi, kun syötteen koko on (a) 1000 (b) 10^5.

Tapauksessa (a) syötteen koko on 1000, jolloin suoraviivainen $O(n^2)$-algoritmi riittää tehtävän ratkaisuun. Algoritmi käy läpi kaikki parit, joita luvuista voi muodostaa, ja tarkastaa, onko joukossa samoja lukuja.

```
bool ok = false;
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        if (x[i] == x[j]) ok = true;
    }
}
```

Tapauksessa (b) syötteen koko on 10^5, jolloin $O(n^2)$-algoritmi on liian hidas tehtävän ratkaisuun. Tämä syötekoko viittaa yleensä siihen, että tehtävään haetaan $O(n)$- tai $O(n \log n)$-algoritmia.

Yksi tapa ratkaista tehtävä tehokkaasti on järjestää ensin luvut, missä kuluu aikaa $O(n \log n)$. Järjestämisen hyötynä on, että sen jälkeen mahdolliset yhtä suuret luvut ovat peräkkäin ja ne pystyy tunnistamaan $O(n)$-ajassa.

Seuraava koodi olettaa, että käytössä on funktio `sort`, joka järjestää taulukon sisällön $O(n \log n)$-ajassa.

```
sort(x,n);
bool ok = false;
for (int i = 1; i <= n-1; i++) {
    if (x[i] == x[i+1]) ok = true;
}
```

Tehtävä on mahdollista ratkaista myös $O(n)$-ajassa pitämällä yllä kirjanpitoa, joka kertoo, montako kertaa kukin luku on esiintynyt. Seuraavassa koodissa kirjanpito on taulukossa `c` Aikavaativuus on vain $O(n)$, kunhan luvut ovat niin pieniä, että niitä voi käyttää taulukon indekseinä.

```
bool ok = false;
for (int i = 1; i <= n; i++) {
    c[x[i]]++;
    if (c[x[i]] == 2) ok = true;
}
```

Luku 3

Järjestäminen

Järjestäminen (*sorting*) on keskeinen algoritmiikan ongelma, ja moni tehokas algoritmi perustuu järjestämiseen. Tehokkaat yleiset järjestämisalgoritmit toimivat ajassa $O(n \log n)$, ja tämä aikavaativuus esiintyykin usein järjestämiseen perustuvissa algoritmeissa.

Järjestämiseen on kehitetty monia algoritmeja, jotka tarjoavat hyviä esimerkkejä algoritmien suunnittelun tekniikoista. Käytännössä järjestämisalgoritmia ei kuitenkaan tarvitse yleensä toteuttaa itse, koska C++:n ja muiden kielten standardikirjastoissa on hyvät valmiit algoritmit järjestämiseen.

3.1 Järjestämisen teoriaa

Järjestämisen perusongelma on seuraava:

> *Tehtävä:* Annettuna on taulukko, jossa on n alkiota $x_1, x_2, \ldots, x_n$. Tehtäväsi on järjestää alkiot pienimmästä suurimpaan.

Esimerkiksi taulukko

1	3	6	2	8	2	5	9

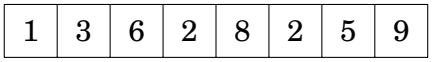
on järjestettynä seuraava:

1	2	2	3	5	6	8	9

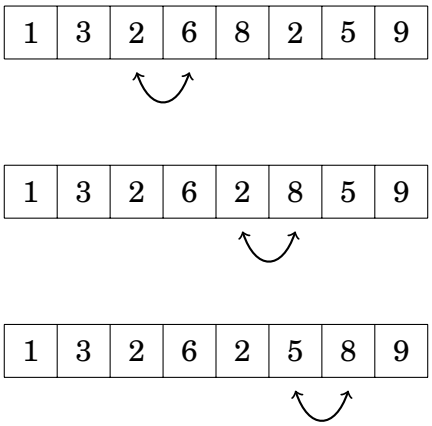
3.1.1 Kuplajärjestäminen

Kuplajärjestäminen (*bubble sort*) on yksinkertainen järjestämisalgoritmi, jonka toiminta perustuu peräkkäisiin taulukon läpikäynteihin. Algoritmin jokainen läpikäynti tarkastaa, onko taulukossa vierekkäisiä alkioita, jotka ovat väärässä järjestyksessä, ja korjaa tällaisten alkioiden järjestyksen.

Esimerkiksi taulukossa



kuplajärjestämisen ensimmäinen läpikäynti tekee seuraavat vaihdot:

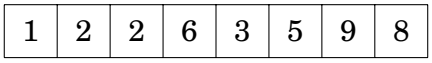


Läpikäynnit jatkuvat, kunnes kaikki vierekkäiset alkiot ovat oikein päin, mikä tarkoittaa, että taulukko on järjestyksessä. Tämä tapahtuu aina viimeistään $n-1$ läpikäynnin jälkeen, joten algoritmin aikavaativuus on $O(n^2)$.

3.1.2 Inversiot

Kuplajärjestäminen on esimerkki algoritmista, joka perustuu taulukon vierekkäisten alkioiden vaihtamiseen. Osoittautuu, että tällaisen algoritmin aikavaativuus ei voi olla parempi kuin $O(n^2)$, koska tarvittava alkioiden vaihtojen määrä saattaa olla luokkaa $O(n^2)$.

Hyödyllinen käsite vierekkäisiä alkioita vaihtavien järjestämisalgoritmien analyysissa on inversio (*inversion*), joka on taulukossa väärin päin oleva (ei välttämättä vierekkäinen) alkiopari. Esimerkiksi taulukossa

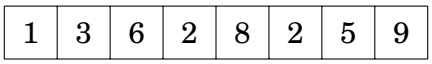


inversiot ovat $(6,3)$, $(6,5)$ ja $(9,8)$. Inversioiden määrä kuvaa, miten lähellä järjestystä taulukko on. Jokainen väärin päin olevien vierekkäisten alkioiden vaihto poistaa taulukosta yhden inversion, joten inversioiden määrä on sama kuin järjestämiseen tarvittava vaihtojen määrä.

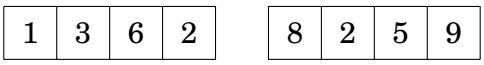
Suurin määrä inversioita syntyy silloin, kun taulukossa ei ole samoja alkioita ja alkioiden järjestys on käänteinen. Silloin taulukon jokainen alkiopari on inversio ja inversioiden määrä on $\frac{n(n-1)}{2} = O(n^2)$. Niinpä vierekkäisiä alkioita vaihtavan algoritmin aikavaativuus on aina ainakin $O(n^2)$, koska sen täytyy tehdä pahimmassa tapauksessa $O(n^2)$ vaihtoa.

3.1.3 Lomitusjärjestäminen

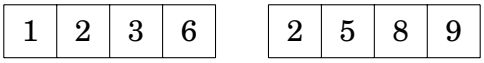
Lomitusjärjestäminen (*mergesort*) on tehokas järjestämisalgoritmi, jonka toiminta on rekursiivinen. Algoritmi jakaa järjestettävän taulukon kahdeksi osataulukoksi, järjestää osataulukot rekursiivisesti ja yhdistää lopuksi osataulukot järjestetyksi taulukoksi. Esimerkiksi taulukko



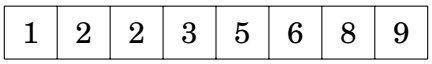
jakautuu osataulukoiksi



jotka ovat järjestettyinä



Osataulukot yhdistämällä syntyy järjestetty taulukko



Algoritmin pohjatapauksena on taulukko, jossa on vain yksi alkio. Tällainen taulukko on valmiiksi järjestyksessä, joten sille ei tarvitse tehdä mitään ja rekursiivinen jakautuminen päättyy.

Lomitusjärjestämisen aikavaativuus on $O(n \log n)$, koska algoritmin aikana osataulukoista muodostuu $O(\log n)$ tasoa ja jokaisella tasolla osataulukoiden yhdistäminen vie yhteensä $O(n)$ aikaa. Osataulukoiden yhdistäminen onnistuu tehokkaasti sen ansiosta, että ne ovat järjestyksessä.

3.1.4 Järjestämisen alaraja

Lomitusjärjestämisen lisäksi on useita muitakin $O(n \log n)$-aikaisia järjestämisalgoritmeja, kuten pikajärjestäminen (*quicksort*) ja kekojärjestäminen (*heapsort*). Kukaan ei ole kuitenkaan onnistunut keksimään yleistä järjestämisalgoritmia, joka toimisi nopeammin kuin ajassa $O(n \log n)$.

Tähän on yllättävä syy: on mahdollista todistaa, että järjestämistä ei voi toteuttaa nopeammin kuin ajassa $O(n \log n)$.

Todistuksen ideana on tarkastella järjestämistä prosessina, jossa jokainen kahden alkion vertailu antaa lisää tietoa taulukon sisällöstä. Prosessilla on $n!$ mahdollista tulosta: kaikki mahdolliset järjestykset, joita taulukon alkoista voi muodostaa. Koska jokainen vertailu antaa kyllä/ei-vastauksen, tarvittava vertailuiden määrä on vähintään

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n) = O(n \log n).$$

3.1.5 Laskemisjärjestäminen

Järjestämisen alaraja $O(n \log n)$ ei koske algoritmeja, jotka eivät perustu alkioiden vertailuun vaan hyödyntävät jotain muuta tietoa alkioista. Esimerkki tällaisesta algoritmista on laskemisjärjestäminen (*counting sort*), joka järjestää kokonaisluvuista koostuvan taulukon $O(n)$-ajassa.

Algoritmin ideana on luoda kirjanpito, josta selviää, montako kertaa mikäkin alkio esiintyy taulukossa. Esimerkiksi taulukosta

1	3	6	2	8	2	5	9

syntyvä kirjanpito on seuraava:

1	2	3	4	5	6	7	8	9	10
1	2	1	0	1	1	0	1	1	0

Kirjanpidon muodostus vie aikaa $O(n)$, ja tämän jälkeen järjestetyn taulukon luominen vie myös aikaa $O(n)$, koska kunkin alkion määrän saa selville suoraan kirjanpidosta.

Laskemisjärjestäminen on hyvin tehokas algoritmi, mutta sen käyttäminen vaatii, että taulukon sisältönä on niin pieniä kokonaislukuja, että niitä voi käyttää kirjanpidon taulukon indeksöinnissä.

3.2 Järjestäminen C++:ssa

Käytännössä järjestämistä ei kannata yleensä toteuttaa itse, vaan parempi ratkaisu on käyttää ohjelmointikielen standardikirjastoon kuuluvaa järjestämisalgoritmia. Esimerkiksi C++:ssa on tehokas `sort`-funktio, jonka avulla pystyy järjestämään helposti taulukoita ja muita tietorakenteita.

3.2.1 Peruskäyttö

Seuraava koodi järjestää taulukon `t`, jossa on n alkiota:

```
sort(t, t+n);
```

Vastaavasti seuraava koodi järjestää vektorin `v`:

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

Jos järjestettävänä on lukuja, ne järjestyvät pienimmästä suurimpaan. Jos taas järjestettävänä on merkkijonoja, ne järjestyvät aakkosjärjestykseen.

Seuraava koodi järjestää merkkijonon `s`:

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

Merkkijonon järjestäminen tarkoittaa, että sen merkit järjestetään aakkosjärjestykseen. Esimerkiksi merkkijono ”apina” on järjestettynä ”aainp”.

Vektorin ja merkkijonon tapauksessa järjestyksen saa myös käänteiseksi muuttamalla sanat `begin` ja `end` muotoon `rbegin` ja `rend`:

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

Toinen mahdollisuus on kääntää järjestys järjestämisen jälkeen:

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

3.2.2 Parien järjestäminen

Jos järjestettävät alkiot ovat pareja (`pair`), niiden järjestys määräytyy ensisijaisesti ensimmäisen kentän (`first`) mukaan ja toissijaisesti toisen kentän (`second`) mukaan. Seuraava koodi esittelee asiaa:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

Koodin suorituksen jälkeen parien järjestys on $(1,2)$, $(1,5)$, $(2,3)$.

Järjestäminen toimii automaattisesti vastaavasti myös silloin, kun alkiot muodostuvat sisäkkäisistä pareista. Samoin tuplet (`tuple`) järjestyvät automaattisesti ensimmäisen eroavan kentän mukaan.

3.2.3 Tietueiden järjestäminen

Jos järjestettävät alkiot ovat tietueita (`struct`), niiden järjestyksen määrää operaattori <. Operaattorin tulee palauttaa `true`, jos oma alkio on pienempi kuin parametrialkio, ja muuten `false`. Järjestämisen aikana `sort`-funktio käyttää operaattoria < selvittääkseen, mikä on kahden alkion järjestys.

Seuraava tietue `P` sisältää pisteen x- ja y-koordinaatit. Pisteet järjestyvät ensisijaisesti x-koordinaatin ja toissijaisesti y-koordinaatin mukaan.

```
struct P {
    int x, y;

    bool operator<(const p& a) {
        if (this.x != a.x) return this.x < a.x;
        else return this.y < a.y;
    }
};
```

3.2.4 Vertailufunktio

On myös mahdollista antaa `sort`-funktiolle ulkopuolinen vertailufunktio. Esimerkiksi seuraava vertailufunktio järjestää merkkijonot ensisijaisesti pituuden mukaan ja toissijaisesti aakkosjärjestyksen mukaan:

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

Tämän jälkeen merkkijonovektorin voi järjestää näin:

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

3.3 Binäärihaku

Binäärihaku (*binary search*) on algoritmi, joka etsii tehokkaasti alkiota järjestetystä taulukosta. Algoritmi hyödyntää tietoa siitä, että taulukko on järjestyksessä, minkä ansiosta aikaa kuluu vain $O(\log n)$. Tämä on merkittävä tehostus verrattuna tavalliseen $O(n)$-algoritmiin, joka käy kaikki alkiot läpi.

3.3.1 Toteutus

Binäärihaun toteutukseen on monia lähestymistapoja. Seuraavaksi esitettävä toteutus käy taulukkoa läpi vasemmalta oikealle hyppien. Aluksi hypyn pituus on $n/2$, sitten $n/4$, sitten $n/8$ jne., kunnes pituus on lopuksi 1. Hyppyjen jälkeen joko haettava alkio on löytynyt tai selviää, että sitä ei ole taulukossa.

Seuraava funktio `contains` etsii vektorista v alkiota x binäärihaulla. Funktio palauttaa `true`, jos alkio x on vektorissa, ja muuten `false`. Funktio edellyttää, että vektorin alkiot on järjestetty pienimmästä suurimpaan.

```
bool contains(vector<int> v, int x) {
    int n = v.size();
    int k = 0;
    for (int b = n/2; b >= 1; b /= 2) {
        while (k+b < n && v[k+b] <= x) k += b;
    }
    return v[k] == x;
}
```

Muuttuja k osoittaa vektorin alkioon ja muuttuja b määrittää hypyn pituuden. Algoritmi etsii suurimman k:n arvon, jolle pätee $\mathtt{v}[k] \leq x$, joten jos alkio x on vektorissa, niin algoritmin päätteeksi $\mathtt{v}[k] = x$.

For-silmukan sisällä oleva koodi suoritetaan $O(\log n)$ kertaa, koska hypyn pituus b puolittuu joka vaiheessa. While-silmukka suoritetaan korkeintaan kaksi kertaa kullakin b:n arvolla. Niinpä algoritmin aikavaativuus on $O(\log n)$.

Samalla idealla voi myös laskea, montako kertaa alkio x esiintyy vektorissa:

```
int count(vector<int> v, int x) {
    int n = v.size();
    int k1 = -1, k2 = -1;
    for (int b = n/2; b >= 1; b /= 2) {
        while (k1+b < n && v[k1+b] < x) k1 += b;
        while (k2+b < n && v[k2+b] <= x) k2 += b;
    }
    return k2-k1;
}
```

Nyt algoritmi etsii kaksi kohtaa vektorista: k_1 on viimeinen kohta, jossa $\texttt{v}[k_1] < x$, ja k_2 on viimeinen kohta, jossa $\texttt{v}[k_2] \leq x$. Erotus $k_2 - k_1$ kertoo, montako kertaa alkio x esiintyy vektorissa.

3.3.2 Muutoskohta

Käytännössä binäärihakua tarvitsee koodata harvoin alkion etsimiseen taulukosta, koska sen sijasta voi käyttää standardikirjastoa. Esimerkiksi C++:n funktiot `lower_bound` ja `upper_bound` toteuttavat binäärihaun ja tietorakenne `set` ylläpitää joukkoa, jonka operaatiot ovat $O(\log n)$-aikaisia.

Sitäkin tärkeämpi binäärihaun käyttökohde on funktion muutoskohdan etsiminen. Oletetaan, että haluamme löytää pienimmän arvon k, joka on kelvollinen ratkaisu ongelmaan. Käytössämme on funktio $\texttt{ok}(x)$, joka palauttaa `true`, jos x on kelvollinen ratkaisu, ja muuten `false`. Lisäksi tiedämme, että $\texttt{ok}(x)$ 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 näyttää seuraavalta:

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

Nyt muutoskohta on mahdollista etsiä käyttämällä binäärihakua:

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

Haku etsii suurimman x:n arvon, jolla $\texttt{ok}(x)$ on `false`. Niinpä tästä seuraava arvo $k = x+1$ on pienin arvo, jolla $\texttt{ok}(k)$ on `true`. Hypyn aloituspituus z tulee olla sopiva suuri luku, esimerkiksi sellainen, jolla $\texttt{ok}(z)$ on varmasti `true`.

Algoritmi kutsuu $O(\log z)$ kertaa funktiota `ok`, joten kokonaisaikavaativuus riippuu siitä, kauanko funktion `ok` suoritus kestää. Usein ratkaisun kelvollisuuden voi tarkastaa ajassa $O(n)$, jolloin kokonaisaikavaativuus on $O(n \log z)$.

3.3.3 Huippukohta

Binäärihaulla voi myös etsiä suurimman alkion taulukossa, jonka alkuosa on nouseva ja loppuosa on laskeva. Toisin sanoen tehtävänä on etsiä taulukon kohta k niin, että $t[x] < t[x+1]$, kun $x < k$, ja $t[x] > t[x+1]$, kun $x >= k$.

Esimerkiksi taulukossa

2	4	5	6	7	4	2	1

suurin alkio on 7.

Ideana on etsiä binäärihaulla taulukosta viimeinen kohta x, jossa pätee $t[x] < t[x+1]$. Tällöin $k = x+1$, koska joko k on taulukon viimeinen kohta tai pätee $t[x+1] > t[x+2]$. Seuraava koodi toteuttaa haun:

```
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 k = x+1;
```

Huomaa, että toisin kuin tavallisessa binäärihaussa, tässä ei ole sallittua, että peräkkäiset arvot olisivat yhtä suuria. Silloin ei olisi mahdollista tietää, mihin suuntaan hakua tulee jatkaa.

Luku 4

Tietorakenteet

Tietorakenne (*data structure*) tarkoittaa tapaa säilyttää tietoa tietokoneen muistissa. Sopivan tietorakenteen valinta on tärkeää, koska kullakin rakenteella on omat vahvuutensa ja heikkoutensa. Tietorakenteen valinnassa oleellinen kysymys on, mitkä operaatiot rakenne toteuttaa tehokkaasti.

Tämä luku esittelee keskeisimmät C++:n standardikirjaston tietorakenteet. Valmiita tietorakenteita kannattaa käyttää aina kun mahdollista, koska se säästää paljon aikaa toteutuksessa. Myöhemmin kirjassa tutustumme erikoisempiin rakenteisiin, joita ei ole valmiina C++:ssa.

4.1 Dynaaminen taulukko

Dynaaminen taulukko on taulukko, jonka kokoa pystyy muuttamaan. C++:n tavallinen dynaaminen taulukko on vektori (`vector`). Vektoria pystyy käsittelemään tehokkaasti sen lopusta: alkion lisäys loppuun ja alkion poisto lopusta vievät aikaa keskimäärin $O(1)$.

Seuraava koodi luo tyhjän vektorin ja lisää siihen kolme lukua:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

Tämän jälkeen vektorin sisältöä voi käsitellä taulukon tavoin:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Funktio `size` kertoo, montako alkiota vektorissa on. Seuraava koodi tulostaa kaikki vektorin alkiot:

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

Vektorin voi käydä myös läpi lyhyemmin näin:

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

Funktio `back` hakee vektorin viimeisen alkion, ja funktio `pop_back` poistaa vektorin viimeisen alkion:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

Vektorin sisällön voi antaa myös sen luonnissa:

```
vector<int> v = {2, 4, 2, 5, 1};
```

Kolmas tapa luoda vektori on ilmoittaa vektorin koko ja alkuarvo:

```
// koko 100, alkuarvo 0
vector<int> v(100);
// koko 10, alkuarvo 5
vector<int> w(100, 5)
```

Hyödyllisiä funktioita ovat myös `clear`, joka tyhjentää vektorin, sekä `resize`, joka muuttaa vektorin kokoa.

Merkkijono

Myös merkkijono (`string`) on dynaaminen taulukko, jota pystyy käsittelemään lähes samaan tapaan kuin vektoria. Merkkijonon käsittelyyn liittyy lisäksi erikoissyntaksia ja funktioita, joita ei ole muissa tietorakenteissa.

Merkkijonoja voi yhdistää toisiinsa +-merkin avulla. Funktio `substr`(k,x) erottaa merkkijonosta osajonon, joka alkaa kohdasta k ja jonka pituus on x. Funktio `find(t)` etsii kohdan, jossa osajono `t` esiintyy merkkijonossa.

Seuraava koodi esittelee merkkijonon käyttämistä:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

4.2 Joukko

Joukko (*set*) on tietorakenne, joka vastaa matemaattista joukkoa. Sen tärkeimmät operaatiot ovat alkion lisäys, haku ja poisto.

C++ sisältää kaksi toteutusta joukolle: `set` ja `unordered_set`. Rakenne `set` perustuu tasapainoiseen binääripuuhun, ja sen operaatioiden aikavaativuus on $O(\log n)$. Rakenne `unordered_set` pohjautuu hajautustauluun, ja sen operaatioiden aikavaativuus on keskimäärin $O(1)$.

Usein on makuasia, kumpaa joukon toteutusta käyttää. Rakenteen `set` etuna on, että se säilyttää joukon alkioita järjestyksessä ja tarjoaa järjestykseen liittyviä funktioita, joita `unordered_set` ei sisällä. Toisaalta `unordered_set` on usein hieman nopeampi rakenne.

Seuraava koodi luo lukuja sisältävän joukon ja esittelee sen käyttämistä. Funktio `insert` lisää joukkoon alkion, funktio `count` laskee alkion määrän joukossa ja funktio `erase` poistaa alkion joukosta.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

C++:n joukko vastaa siinä mielessä matemaattista joukkoa, että sama alkio voi esiintyä siinä vain kerran. Niinpä funktio `count` palauttaa aina arvon 0 tai 1 ja funktio `insert` ei lisää joukkoa uudestaan alkioon, jos se on siellä valmiina. Seuraava koodi havainnollistaa asiaa:

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ sisältää kuitenkin myös rakenteet `multiset` ja `unordered_multiset`, jotka toimivat muuten kuin `set` ja `unordered_set`, mutta sama alkio voi esiintyä niissä monta kertaa. Muuttamalla äskeisen koodin rakenteeksi `multiset` kaikki luvun 5 kopiot lisätään joukkoon:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Komento `erase` poistaa kaikki alkion esiintymät `multiset`-rakenteessa:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Usein kuitenkin tulisi poistaa vain yksi esiintymä, mikä onnistuu näin:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

4.3 Hakemisto

Hakemisto (*map*) on taulukon yleistys, joka sisältää joukon avain-alkio-pareja. Siinä missä taulukossa avaimet ovat aina peräkkäiset kokonaisluvut $0, 1, 2, \ldots$, hakemistossa ne voivat olla mitä tahansa arvoja.

Joukkoa vastaavasti C++ sisältää sekä binääripuuhun että hajautustauluun perustuvat hakemistot `map` ja `unordered_map`, joissa alkion käsittely vie aikaa vastaavasti $O(\log n)$ ja keskimäärin $O(1)$.

Seuraava koodi toteuttaa hakemiston, jossa avaimet ovat merkkijonoja ja alkiot ovat kokonaislukuja:

```
map<string,int> m;
m["apina"] = 4;
m["banaani"] = 3;
m["cembalo"] = 9;
cout << m["banaani"] << "\n"; // 3
```

Jos hakemistosta hakee avainta, jota ei ole siinä, avain lisätään hakemistoon automaattisesti oletusarvolla. Esimerkiksi seuraavassa koodissa hakemistoon ilmestyy avain ”aybabtu”, jonka alkiona on 0:

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

Komennolla `count` voi tutkia, esiintyykö avain hakemistossa:

```
if (m.count("aybabtu")) {
    cout << "avain on hakemistossa";
}
```

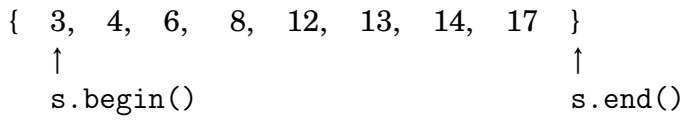
Seuraava koodi listaa hakemiston kaikki avaimet ja alkiot:

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

4.4 Iteraattorit ja välit

Monet C++:n standardikirjaston funktiot käsittelevät tietorakenteiden iteraattoreita ja niiden määrittelemiä välejä. Iteraattori (*iterator*) on muuttuja, joka osoittaa tiettyyn tietorakenteen alkioon.

Usein tarvittavat iteraattorit ovat `begin` ja `end`, jotka rajaavat välin, joka sisältää kaikki tietorakenteen alkiot. Iteraattori `begin` osoittaa tietorakenteen ensimmäiseen alkioon, kun taas iteraattori `end` osoittaa tietorakenteen viimeisen alkion jälkeiseen kohtaan. Tilanne on siis tällainen:



Huomaa epäsymmetria iteraattoreissa: `s.begin()` osoittaa tietorakenteen alkioon, kun taas `s.end()` osoittaa tietorakenteen ulkopuolelle. Iteraattoreiden rajaama joukon väli on siis puoliavoin.

Taulukon välit

Iteraattoreita tarvitsee C++:n standardikirjaston funktioissa, jotka käsittelevät tietorakenteen välejä. Yleensä halutaan käsitellä tietorakenteiden kaikkia alkioita, jolloin funktiolle annetaan iteraattorit `begin` ja `end`.

Seuraava koodi järjestää vektorin funktiolla `sort`, kääntää sitten alkioiden järjestyksen funktiolla `reverse` ja sekoittaa lopuksi alkioiden järjestyksen funktiolla `random_shuffle`.

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

Samoja funktioita voi myös käyttää tavallisen taulukon yhteydessä, jolloin iteraattorin sijasta annetaan osoitin taulukkoon:

```
sort(t, t+n);
reverse(t, t+n);
random_shuffle(t, t+n);
```

Joukon iteraattorit

Iteraattoreita tarvitsee usein joukon alkioiden käsittelyssä. Seuraava koodi määrittelee iteraattorin `it`, joka osoittaa joukon `s` alkuun:

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

Koodin voi kirjoittaa myös lyhyemmin näin:

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

Iteraattoria vastaavaan joukon alkioon pääsee käsiksi *-merkinnällä. Esimerkiksi seuraava koodi tulostaa joukon ensimmäisen alkion:

```
auto it = s.begin();
cout << *it << "\n";
```

Iteraattoria pystyy liikuttamaan operaatioilla ++ (eteenpäin) ja -- (taaksepäin). Tällöin iteraattori siirtyy seuraavaan tai edelliseen alkioon joukossa.

Seuraava koodi tulostaa joukon kaikki alkiot:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Seuraava koodi taas tulostaa joukon viimeisen alkion:

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

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

```
auto it = s.find(x);
if (it == s.end()) cout << "x puuttuu joukosta";
```

Funktio `lower_bound`(x) palauttaa iteraattorin joukon pienimpään alkioon, joka on ainakin yhtä suuri kuin x. Vastaavasti `upper_bound`(x) palauttaa iteraattorin pienimpään alkioon, joka on suurempi kuin x. Jos tällaisia alkioita 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";
}
```

Iteraattori `a` osoittaa pienimpään 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 joukon viimeinen alkio. Muussa tapauksessa x:ää lähin alkio on joko a:n osoittama alkio tai tätä edellinen alkio.

4.5 Muita tietorakenteita

4.5.1 Pino ja jono

Pino (stack) on tietorakenne, joka sisältää kaksi operaatiota: alkion lisäys pinon päälle ja alkion poisto pinon päältä. Pinossa ei ole mahdollista käsitellä muita alkioita kuin pinon päällimmäistä alkiota.

Seuraava koodi esittelee pinon käyttämistä:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Jono (queue) on tietorakenne, jossa alkion lisäys tapahtuu jonon loppuun mutta alkion poisto tapahtuu jonon alusta. Jonossa ei ole mahdollista käsitellä muita alkioita kuin ensimmäistä ja viimeistä.

Seuraava koodi esittelee jonon käyttämistä:

```
queue<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.front(); // 3
s.pop();
cout << s.front(); // 2
```

Pino ja jono ovat harvoin tarvittavia tietorakenteita, koska niiden sijasta voi käyttää esimerkiksi vektoria. Pinon ja jonon ideat ovat kuitenkin tärkeitä algoritmien suunnittelussa.

4.5.2 Pakka

Pakka (deque) on kuin vektori, mutta siinä taulukon kokoa pystyy muuttamaan tehokkaasti sekä taulukon alussa että lopussa. Tämän mahdollistamiseksi pakka sisältää funktioiden push_back ja pop_back lisäksi myös funktiot push_front ja pop_front.

Seuraava koodi esittelee pakan käyttämistä:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

Pakan sisäinen toteutus on monimutkaisempi kuin vektorissa, minkä vuoksi se on jonkin verran vektoria raskaampi rakenne. Kuitenkin vektorin tavoin lisäyksen ja poiston aikavaativuus on keskimäärin $O(1)$.

4.5.3 Prioriteettijono

Prioriteettijono (`priority_queue`) pitää yllä joukkoa alkioista. Sen operaatiot ovat alkion lisäys ja jonon tyypistä riippuen joko pienimmän alkion haku ja poisto tai suurimman alkion haku ja poisto. Lisäyksen ja poiston aikavaativuus on $O(\log n)$ ja haun aikavaativuus on $O(1)$.

Prioriteettijonon operaatiot ja enemmänkin pystyy toteuttamaan myös `set`-rakenteella. Prioriteettijonon etuna on kuitenkin, että sen kekoon perustuva sisäinen toteutus on yksinkertaisempi kuin `set`-rakenteen binääripuu, minkä vuoksi rakenne on kevyempi ja operaatiot ovat tehokkaampia.

Seuraava koodi esittelee prioriteettijonon käyttämistä. Oletuksena prioriteettijono toimii niin, että siitä voi hakea ja poistaa suurimman alkion.

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Seuraava koodi luo käänteisen prioriteettijonon, jossa voi hakea ja poistaa pienimmän alkion:

```
priority_queue<int,vector<int>,greater<int>> q;
```

Luku 5

Peruuttava haku

Peruuttava haku (*backtracking*) on systemaattinen tapa käydä läpi kaikki ratkaisut, jotka voi muodostaa annetuista aineksista. Haku rakentaa ratkaisua askel kerrallaan ja haarautuu joka vaiheessa sen mukaan, millä kaikilla tavoilla ratkaisua voi jatkaa eteenpäin. Muodostettuaan yhden ratkaisun haku peruuttaa takaisin muodostamaan muita ratkaisuja.

Peruuttava haku on hyvä menetelmä, jos kaikki ratkaisut ehtii käydä läpi, koska haku on yleensä suoraviivainen toteuttaa ja se antaa varmasti oikean vastauksen. Jos peruuttava haku on liian hidas, seuraavien lukujen ahneet algoritmit tai dynaaminen ohjelmointi voivat soveltua tehtävään.

5.1 Osajoukot

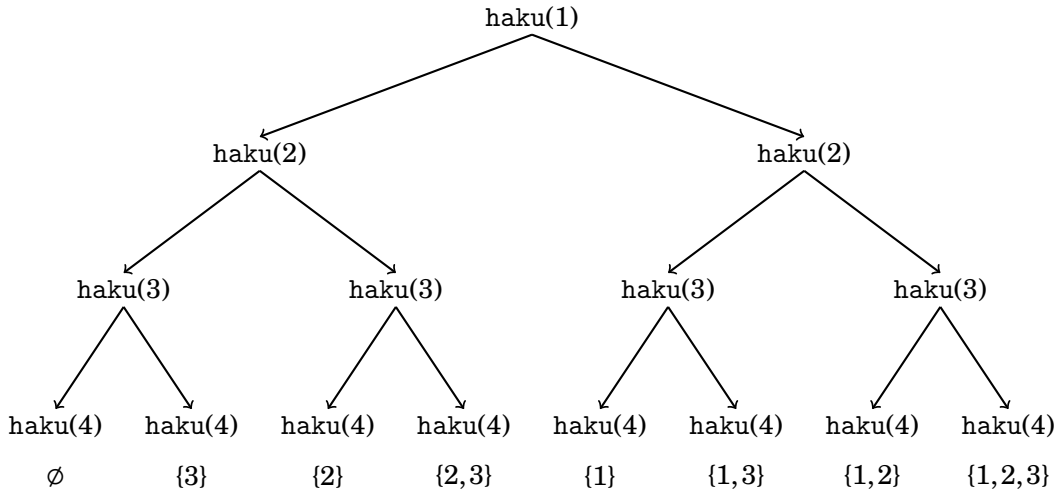
Yksi tavallinen peruuttava haun käyttötarkoitus on joukon osajoukkojen muodostaminen. Esimerkiki joukon $\{1,2,3\}$ osajoukot ovat $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ ja $\{1,2,3\}$, missä $\emptyset$ on tyhjä joukko.

Seuraava funktio `haku` muodostaa joukon $\{1,2,\ldots,n\}$ osajoukot. Funktio pitää yllä vektoria `v`, joka sisältää osajoukossa olevat luvut, ja parametri k ilmaisee käsittelyvuorossa olevan luvun.

```
void haku(int k) {
    if (k == n+1) {
        // käsittele osajoukko
    } else {
        haku(k+1);
        v.push_back(k);
        haku(k+1);
        v.pop_back();
    }
}
```

Funktiota kutsutaan `haku(1)`, minkä jälkeen se muodostaa joukon osajoukot rekursiivisesti. Joka kutsulla funktio haarautuu kahteen tapaukseen: joko luku k tulee tai ei tule mukaan osajoukkoon. Aina kun $k = n+1$, kaikki luvut on käyty läpi ja yksi osajoukko on muodostettu.

Esimerkiksi kun $n = 3$, funktion suoritus etenee seuraavan kuvan mukaisesti. Joka kutsussa oikea haara lisää luvun osajoukkoon ja vasen haara jättää luvun pois joukosta.



Joukosta $\{1,2,\ldots,n\}$ voidaan muodostaa kaikkiaan 2^n osajoukkoa, koska jokainen alkio joko valitaan osajoukkoon tai jää osajoukon ulkopuolelle. Muodostus vie aikaa $O(2^n)$, koska jokainen funktion kutsu on vakioaikainen.

5.2 Permutaatiot

Peruuttavan haun avulla voi myös muodostaa joukon alkioiden permutaatiot. Esimerkiksi joukon $\{1,2,3\}$ alkioiden permutaatiot ovat $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ ja $(3,2,1)$.

Seuraava funktio `haku` muodostaa joukon $\{1,2,\ldots,n\}$ permutaatiot. Funktio muodostaa permutaation vektoriin `v`. Lisäksi funktio pitää yllä taulukkoa `p`, joka kertoo, onko tietty luku mukana permutaatiossa. Jos `p`$[k] = 0$, luku k ei ole mukana, ja jos `p`$[k] = 1$, luku k on mukana.

```
void haku() {
    if (v.size() == n) {
        // käsittele permutaatio
    } else {
        for (int i = 1; i <= n; i++) {
            if (p[i]) continue;
            p[i] = 1;
            v.push_back(i);
            haku();
            p[i] = 0;
            v.pop_back();
        }
    }
}
```

Haku alkaa kutsumalla funktiota `haku()`. Funktiolla ei ole parametreja, koska se selvittää vektorin `v` koosta, onko permutaatio tullut valmiiksi.

Joukon $\{1,2,\ldots,n\}$ permutaatioiden määrä on $n!$, koska permutaation ensimmäisen luvun voi valita n tavalla, toisen voi valita $n-1$ tavalla jne. Tässä esitetty toteutus muodostaa permutaatiot ajassa $O(n!n)$, koska jokainen funktion kutsu käy läpi luvut $1 \ldots n$.

Funktio `next_permutation`

Vaihtoehtoinen tapa käydä läpi permutaatiot on käyttää C++:n standardikirjastoon kuuluvaa funktiota `next_permutation`. Se muuttaa taulukossa olevan permutaation seuraavaksi järjestyksessä olevaksi permutaatioksi.

Seuraava koodi muodostaa joukon $\{1,2,\ldots,n\}$ permutaatiot käyttäen apuna funktiota `next_permutation`:

```
vector<int> v;
for (int i = 1; i <= n; i++) {
    v.push_back(i);
}
do {
    // käsittele permutaatio
} while (next_permutation(v.begin(),v.end()));
```

Funktion `next_permutation` kutsu vie aikaa $O(n)$, joten myös tämän toteutuksen aikavaativuus on $O(n!n)$.

5.3 Ratsutehtävä

Tarkastellaan lopuksi seuraavaa tehtävää:

> *Tehtävä:* Tehtäväsi on laskea, montako reittiä on olemassa, joissa ratsu lähtee liikkelle $n \times n$ -kokoisen shakkilaudan vasemmasta yläkulmasta ja käy kerran jokaisessa ruudussa?

Esimerkiksi 5×5 -shakkilaudassa on yhteensä 304 erilaista ratkaisua. Yksi ratkaisuista 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

Tehtävä ratkeaa simuloimalla ratsun liikkeitä peruuttavan haun avulla. Ratsu aloittaa reittinsä ruudukon vasemmasta yläkulmasta ja joka siirrolla on erilaisia vaihtoehtoja, minne ratsu voi siirtyä seuraavaksi. Jos ruudukko on täynnä, yksi ratsun reitti on löytynyt.

Seuraava koodi laskee reittien määrän globaaliin muuttujaan c:

```
void haku(int y, int x, int k) {
    if (y < 1 || x < 1 || y > n || x > n) return;
    if (s[y][x]) return;
    s[y][x] = k;
    if (k == n*n) {
        c++;
    } 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], k+1);
        }
    }
    s[y][x] = 0;
}
```

Haku alkaa kutsumalla funktiota `haku(1,1,1)`. Funktion parametrit tarkoittavat, että ratsu on ruudukossa rivillä y sarakkeessa x ja se on kulkenut k askelta. Funktio tarkastaa aluksi, että ratsu on laudan sisällä eikä se ole käynyt samassa ruudussa aiemmin. Tämän jälkeen funktio käy läpi kaikki vaihtoehdot, miten ratsu voi jatkaa matkaansa.

Funktion haarautuminen riippuu siitä, missä kohtaa ruudukossa ratsu on ja mitkä ruudut ovat vielä vapaina. Tämän vuoksi ei ole suoraviivaista arvioida, miten kauan reittien määrän laskeminen kestää. Osoittautuu, että tapaus $n = 5$ on nopea laskea, mutta tätä suuremmissa tapauksissa reittien määrä on niin suuri, että niiden laskeminen vaatisi kehittyneemmän hakualgoritmin.

Luku 6

Ahneet algoritmit

Ahne algoritmi (*greedy algorithm*) muodostaa ongelman ratkaisun tekemällä joka askeleella ahneen valinnan eli sillä hetkellä parhaalta näyttävän valinnan. Toisin kuin peruuttava haku, ahne algoritmi ei koskaan peruuta tekemiään valintoja vaan muodostaa ratkaisun suoraan valmiiksi. Tämän ansiosta ahneet algoritmit ovat yleensä hyvin tehokkaita.

Vaikeutena ahneissa algoritmeissa on keksiä toimiva ahne strategia, joka tuottaa optimiratkaisun tehtävään. Ahneen algoritmin tulee olla siis sellainen, että kulloinkin parhaalta näyttävät valinnat tuottavat myös parhaan kokonaisuuden. Tämän perusteleminen voi olla vaikeaa, vaikka ahne algoritmi olisi toiminnaltaan yksinkertainen.

6.1 Kolikkotehtävä

Aloitamme ahneisiin algoritmeihin tutustumisen seuraavasta tehtävästä:

> *Tehtävä:* Kolikoiden arvot ovat $\{c_1, c_2, \ldots, c_k\}$, ja tehtäväsi on muodostaa kolikoista rahamäärä x. Jokaista kolikkoa on saatavilla rajattomasti. Mikä on pienin määrä kolikoita, joilla rahamäärän voi muodostaa?

Esimerkiksi jos kolikot ovat eurokolikot eli sentteinä

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

ja muodostettava rahamäärä on 520, kolikoita tarvitaan vähintään 4. Optimiratkaisu on valita kolikot $200 + 200 + 100 + 20$.

6.1.1 Ahne algoritmi

Luonteva ahne algoritmi tehtävän ratkaisuun on poistaa rahamäärästä aina mahdollisimman suuri kolikko, kunnes rahamäärä on 0. Tämä algoritmi toimii esimerkissä, koska rahamäärästä 520 poistetaan ensin kahdesti 200, sitten 100 ja lopuksi 20. Mutta toimiiko ahne algoritmi aina oikein?

Osoittautuu, että eurokolikoiden tapauksessa ahne algoritmi toimii aina oikein, eli se tuottaa aina ratkaisun, jossa on pienin määrä kolikoita. Algoritmin toimivuuden voi perustella seuraavasti:

Kutakin kolikkoa 1, 5, 10, 50 ja 100 on optimiratkaisussa enintään yksi. Tämä johtuu siitä, että jos ratkaisussa olisi kaksi tällaista kolikkoa, saman ratkaisun voisi muodostaa käyttäen vähemmän kolikoita. Esimerkiksi jos ratkaisussa on kolikot $5+5$, ne voi korvata kolikolla 10.

Vastaavasti kumpaakin kolikkoa 2 ja 20 on optimiratkaisussa enintään kaksi, koska kolikot $2+2+2$ voi korvata kolikoilla $1+5$ ja kolikot $20+20+20$ voi korvata kolikoilla $10+50$. Lisäksi ratkaisussa ei voi olla yhdistelmiä $1+2+2$ ja $10+20+20$, koska ne voi korvata kolikoilla 5 ja 50.

Näiden havaintojen perusteella jokaiselle eurokolikolle x pätee, että x:ää pienemmistä eurokolikoista ei ole mahdollista saada aikaan summaa x tai suurempaa summa optimaalisesti. Esimerkiksi jos $x = 100$, pienemmistä kolikoista saa korkeintaan summan $50+20+20+5+2+2 = 99$. Niinpä ahne algoritmi, joka valitsee aina suurimman kolikon, tuottaa optimiratkaisun.

Kuten tästä esimerkistä huomaa, ahneen algoritmin toimivuuden perusteleminen voi olla vaikeaa, vaikka kyseessä olisi yksinkertainen algoritmi.

6.1.2 Yleinen tapaus

Yleisessä tapauksessa kolikot voivat olla mitä tahansa. Tällöin suurimman kolikon valitseva ahne algoritmi ei välttämättä tuota optimiratkaisua.

Jos ahne algoritmi ei toimi, tämän voi osoittaa näyttämällä vastaesimerkin, jossa algoritmi antaa väärän vastauksen. Tässä tehtävässä vastaesimerkki on helppoa keksiä: jos kolikot ovat $\{1,3,4\}$ ja muodostettava rahamäärä on 6, ahne algoritmi tuottaa ratkaisun $1+1+4$, kun taas optimiratkaisu on $3+3$.

Yleisessä tapauksessa tehtävän ratkaisuun ei tunneta ahnetta algoritmia, mutta palaamme tehtävään seuraavassa luvussa. Tehtävään on nimittäin olemassa dynaamista ohjelmointia käyttävä algoritmi, joka tuottaa optimiratkaisun millä tahansa kolikoilla ja rahamäärällä.

6.2 Aikataulutus

Monet aikataulutukseen liittyvät ongelmat ratkeavat ahneilla algoritmeilla. Tällaisissa ongelmissa on usein monta luontevaa ahnetta ratkaisua ja vaikeutena on selvittää, mikä niistä tuottaa aina optimiratkaisun. Tutustumme seuraavaksi kahteen klassiseen aikataulutusongelmaan.

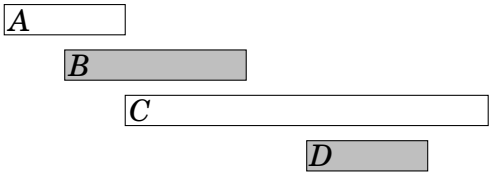
6.2.1 Tapahtumat

Tehtävä: Annettuna on n tapahtumaa, jotka alkavat ja päättyvät tiettyinä hetkinä. Tehtäväsi on suunnitella aikataulu, jota seuraamalla pystyt osallistumaan mahdollisimman moneen tapahtumaan. Et voi osallistua tapahtumaan vain osittain.

Esimerkiksi jos tapahtumat ovat

tapahtuma	alkuaika	loppuaika
A	1	3
B	2	5
C	3	9
D	6	8

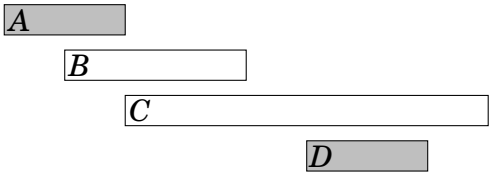
niin on mahdollista osallistua korkeintaan kahteen tapahtumaan. Yksi mahdollisuus on osallistua tapahtumiin B ja D seuraavasti:



Tehtävän ratkaisuun on mahdollista keksiä useita ahneita algoritmeja, mutta mikä niistä toimii kaikissa tapauksissa?

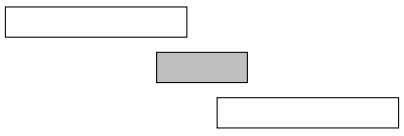
Algoritmi 1

Ensimmäinen idea on valita ratkaisuun mahdollisimman lyhyitä tapahtumia. Esimerkin tapauksessa tällainen algoritmi valitsee tapahtumat



ja tuottaa optimiratkaisun.

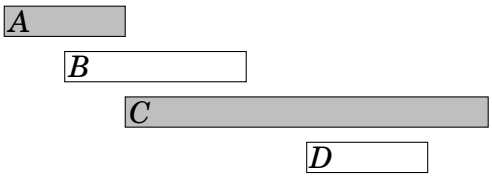
Lyhimpien tapahtumien valinta ei ole kuitenkaan aina toimiva strategia. Algoritmi epäonnistuu esimerkiksi seuraavassa tilanteessa:



Kun lyhyt tapahtuma valitaan mukaan, on mahdollista osallistua vain yhteen tapahtumaan. Kuitenkin valitsemalla pitkät tapahtumat olisi mahdollista osallistua kahteen tapahtumaan.

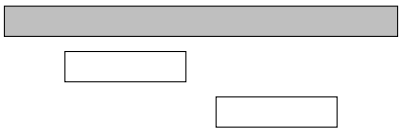
Algoritmi 2

Toinen idea on valita aina seuraavaksi tapahtuma, joka alkaa mahdollisimman aikaisin. Tämä algoritmi valitsee esimerkissä tapahtumat



ja tuottaa optimiratkaisun.

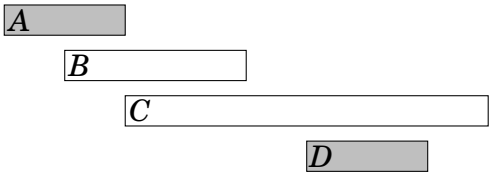
Tämä algoritmi ei kuitenkaan toimi esimerkiksi seuraavassa tilanteessa:



Kun ensimmäisenä alkava tapahtuma valitaan mukaan, mitään muuta tapahtumaa ei ole mahdollista valita. Kuitenkin olisi mahdollista osallistua kahteen tapahtumaan valitsemalla kaksi myöhempää tapahtumaa.

Algoritmi 3

Kolmas idea on valita aina seuraavaksi tapahtuma, joka päättyy mahdollisimman aikaisin. Tämä algoritmi valitsee esimerkissä tapahtumat



ja tuottaa optimiratkaisun.

Osoittautuu, että tämä ahne algoritmi tuottaa *aina* optimiratkaisun. Algoritmin toimii, koska on aina kokonaisuuden kannalta optimaalista valita ensimmäiseksi tapahtumaksi mahdollisimman aikaisin päättyvä tapahtuma. Tämän jälkeen on taas optimaalista valita seuraava aikatauluun sopiva mahdollisimman aikaisin päättyvä tapahtua, jne.

Yksi tapa perustella valintaa on miettiä, mitä tapahtuu, jos ensimmäiseksi tapahtumaksi valitaan jokin muu kuin mahdollisimman aikaisin päättyvä tapahtuma. Oleellinen havainto on, että tällainen valinta ei ole koskaan parempi, koska myöhemmin päättyvän tapahtuman jälkeen on joko yhtä paljon tai vähemmän mahdollisuuksia valita seuraavia tapahtumia.

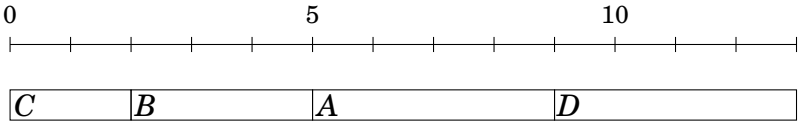
6.2.2 Tehtävät

Tehtävä: Annettuna on n tehtävää, joista jokaisella on kesto ja deadline. Tehtäväsi on valita järjestys, jossa suoritat tehtävät. Saat kustakin tehtävästä $d - x$ pistettä, missä d on tehtävän deadline ja x on tehtävän valmistumishetki. Mikä on suurin mahdollinen pistesumma?

Esimerkiksi jos tehtävät ovat

tehtävä	kesto	deadline
A	4	2
B	3	5
C	2	7
D	4	5

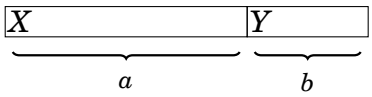
niin optimaalinen ratkaisu on suorittaa tehtävät seuraavasti:



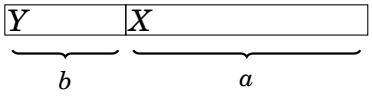
Tässä ratkaisussa C tuottaa 5 pistettä, B tuottaa 0 pistettä, A tuottaa -7 pistettä ja D tuottaa -8 pistettä, joten pistesumma on -10.

Yllättävää kyllä, tehtävän optimaalinen ratkaisu ei riipu lainkaan deadlineista. Toimiva ahne strategia on suorittaa tehtävät järjestyksessä keston mukaan lyhimmästä pisimpään. Syynä tähän on, että jos missä tahansa vaiheessa suoritetaan peräkkäin kaksi tehtävää, joista ensimmäinen kestää toista kauemmin, tehtävien järjestyksen vaihtaminen parantaa ratkaisua.

Esimerkiksi jos peräkkäin ovat tehtävät



ja $a > b$, niin järjestyksen muuttaminen muotoon



antaa X:lle b pistettä vähemmän ja Y:lle a pistettä enemmän, joten kokonaismuutos pistemäärään on $a - b > 0$. Optimiratkaisussa kaikille peräkkäin suoritettaville tehtäville tulee päteä, että lyhyempi tulee ennen pidempää, mistä seuraa, että tehtävät tulee suorittaa järjestyksessä keston mukaan.

6.3 Keskiluvut

Tarkastelemme lopuksi kahta tehtävää, jossa annettuna on joukko lukuja ja tehtävänä on etsiä keskiluku, joka on niitä mahdollisimman lähellä. Ensin minimoimme erotusten itseisarvoa ja tämän jälkeen erotusten neliötä.

6.3.1 Itseisarvosumma

> *Tehtävä:* Annettuna on n lukua $[a_1, a_2, \ldots, a_n]$. Tehtäväsi on etsiä luku x, joka minimoi summan $|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|$.

Esimerkiksi jos luvut ovat $[1,2,9,2,6]$, niin paras ratkaisu on $x = 2$, jolloin summaksi tulee

$$|1-2| + |2-2| + |9-2| + |2-2| + |6-2| = 12.$$

Yleisessä tapauksessa paras valinta x:n arvoksi on lukujen *mediaani* eli keskimmäinen luku järjestyksessä. Esimerkiksi luvut $[1,2,9,2,6]$ ovat järjestyksessä $[1,2,2,6,9]$, joten mediaani on 2.

Mediaanin valinta on paras ratkaisu, koska jos x on mediaania pienempi, x:n suurentaminen pienentää summaa. Vastaavasti jos x on mediaania suurempi, x:n pienentämien pienentää summaa. Niinpä x:ää kannattaa siirtää mahdollisimman lähelle mediaania, eli optimiratkaisu on valita x mediaaniksi.

Jos n on parillinen ja mediaaneja on kaksi, kumpikin mediaani sekä kaikki niiden välillä olevat luvut tuottavat optimaalisen ratkaisun.

6.3.2 Neliösumma

> *Tehtävä:* Annettuna on n lukua $[a_1, a_2, \ldots, a_n]$. Tehtäväsi on etsiä luku x, joka minimoi summan $(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2$.

Esimerkiksi jos luvut ovat $[1,2,9,2,6]$, niin paras ratkaisu on $x = 4$, jolloin summaksi tulee

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46.$$

Nyt paras valinta x:n arvoksi on lukujen *keskiarvo*. Esimerkissä lukujen keskiarvo on $(1+2+9+2+6)/5 = 4$.

Tämän tuloksen voi johtaa järjestämällä summan uudestaan muotoon

$$(a_1^2 + a_2^2 + \cdots + a_n^2) - 2x(a_1 + a_2 + \cdots + a_n) + nx^2.$$

Ensimmäinen osa ei riipu x:stä, joten sen voi unohtaa. Jäljelle jäävistä osista muodostuu funktio $nx^2 - 2xs$, missä $s = a_1 + a_2 + \cdots + a_n$. Tämä on ylöspäin aukeava paraabeli, jonka nollakohdat ovat $x = 0$ ja $x = 2s/n$ ja pienin arvo on näiden keskikohta $x = s/n$ eli taulukon lukujen keskiarvo.

Luku 7

Dynaaminen ohjelmointi

Dynaaminen ohjelmointi (*dynamic programming*) on tekniikka, joka yhdistää peruuttavan haun toimivuuden ja ahneiden algoritmien tehokkuuden. Ideana on käydä läpi kaikki ongelman ratkaisut siten, että haun aikana sama osaratkaisu käsitellään vain kerran. Dynaaminen ohjelmointi toimii tehokkaasti, jos erilaisia osaratkaisuja on riittävän pieni määrä.

Dynaamisen ohjelmoinnin ymmärtäminen on yksi merkkipaalu jokaisen kisakoodarin uralla, ja tutustumme tässä luvussa dynaamiseen ohjelmointiin perusesimerkkien kautta. Dynaaminen ohjelmointi on monipuolinen tekniikka, ja palaamme siihen monta kertaa myöhemmin kirjassa erilaisissa tilanteissa.

7.1 Optimiratkaisun etsiminen

Dynaaminen ohjelmointi liittyy usein optimiratkaisun etsimiseen. Näin on esimerkiksi seuraavassa tutussa tehtävässä:

> *Tehtävä:* Kolikoiden arvot ovat $\{c_1, c_2, \ldots, c_k\}$, ja tehtäväsi on muodostaa kolikoista rahamäärä x. Jokaista kolikkoa on saatavilla rajattomasti. Mikä on pienin määrä kolikoita, joilla rahamäärän voi muodostaa?

Luvussa 6 ratkaisimme tehtävän ahneella algoritmilla, joka muodostaa rahamäärän valiten mahdollisimman suuria kolikoita. Ahne algoritmi toimii esimerkiksi silloin, kun kolikot ovat eurokolikot, mutta yleisessä tapauksessa ahne algoritmi ei välttämättä valitse pienintä määrää kolikoita.

Nyt on aika ratkaista tehtävä tehokkaasti dynaamisella ohjelmoinnilla niin, että algoritmi toimii millä tahansa kolikoilla.

7.1.1 Rekursiivinen esitys

Dynaamisessa ohjelmoinnissa on ideana esittää ongelma rekursiivisesti niin, että ongelman ratkaisun voi laskea saman ongelman pienempien tapausten ratkaisuista. Tässä tehtävässä luonteva ongelma on seuraava: mikä on pienin määrä kolikoita, joilla voi muodostaa rahamäärän x?

Merkitään $f(x)$ funktiota, joka antaa vastauksen ongelmaan, eli $f(x)$ on pienin määrä kolikoita, joilla voi muodostaa rahamäärän x. Funktion arvot riippuvat siitä, mitkä kolikot ovat käytössä. Esimerkiksi jos kolikot ovat $\{1,3,4\}$, funktion ensimmäiset arvot ovat:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 1 \\ f(4) &= 1 \\ f(5) &= 2 \\ f(6) &= 2 \\ f(7) &= 2 \\ f(8) &= 2 \\ f(9) &= 3 \end{aligned}$$

Funktion arvo $f(0) = 0$, koska jos rahamäärä on 0, ei tarvita yhtään kolikkoa. Vastaavasti $f(3) = 1$, koska rahamäärän 3 voi muodostaa kolikolla 3, ja $f(5) = 2$, koska rahamäärän 5 voi muodostaa kolikoilla 1 ja 4.

Oleellinen ominaisuus funktiossa on, että arvon $f(x)$ pystyy laskemaan rekursiivisesti käyttäen pienempiä funktion arvoja. Esimerkiksi jos kolikot ovat $\{1,3,4\}$, on kolme tapaa alkaa muodostaa rahamäärää x: valitaan kolikko 1, 3 tai 4. Jos valitaan kolikko 1, täytyy muodostaa vielä rahamäärä $x-1$. Vastaavasti jos valitaan kolikko 3 tai 4, täytyy muodostaa rahamäärä $x-3$ tai $x-4$.

Niinpä rekursiivinen kaava on

$$f(x) = \min(f(x-1), f(x-3), f(x-4)) + 1,$$

missä funktio min valitsee pienimmän parametreistaan. Yleisemmin jos kolikot ovat $\{c_1, c_2, \ldots, c_k\}$, rekursiivinen kaava on

$$f(x) = \min(f(x-c_1), f(x-c_2), \ldots, f(x-c_k)) + 1.$$

Funktion pohjatapauksena on $f(0) = 0$. Lisäksi on hyvä määritellä $f(x) = \infty$, jos $x < 0$. Tämän ideana on, että negatiivisen rahamäärän muodostaminen vaatii äärettömästi kolikoita, mikä estää sen, että rekursio muodostaisi ratkaisun, johon kuuluu negatiivinen rahamäärä.

C++:lla funktion määrittely näyttää seuraavalta:

```
int f(int x) {
    if (x == 0) return 0;
    if (x < 0) return 1e9;
    int u = 1e9;
    for (int i = 1; i <= k; i++) {
        u = min(u, f(x-c[i])+1);
    }
    return u;
}
```

Koodi olettaa, että käytettävät kolikot ovat c[1], c[2], ..., c[n], ja arvo 10^9 kuvastaa ääretöntä. Tämä on toimiva funktio, mutta se ei ole vielä tehokas, koska funktio käy läpi valtavasti erilaisia tapoja muodostaa rahamäärä. Seuraavaksi esiteltävä muistitaulukko tekee funktiosta tehokkaan.

7.1.2 Muistitaulukko

Dynaaminen ohjelmointi tehostaa rekursiivisen funktion laskentaa tallentamalla funktion arvoja muistitaulukkoon. Taulukon avulla funktion arvo tietyllä parametrilla riittää laskea vain kerran, minkä jälkeen sen voi hakea suoraan taulukosta. Tämä muutos nopeuttaa algoritmia huomattavasti.

Tässä tehtävässä muistitaulukoksi sopii taulukko

```
int d[N];
```

ja ideana on, että kohtaan d[x] lasketaan funktion arvo $f(x)$. Vakio N valitaan niin, että kaikki laskettavat funktion arvot mahtuvat taulukkoon.

Tämän jälkeen funktion voi toteuttaa tehokkaasti näin:

```
int f(int x) {
    if (x == 0) return 0;
    if (x < 0) return 1e9;
    if (d[x]) return d[x];
    int u = 1e9;
    for (int i = 1; i <= k; i++) {
        u = min(u, f(x-c[i])+1);
    }
    d[x] = u;
    return d[x];
}
```

Funktio käsittelee pohjatapaukset $x = 0$ ja $x < 0$ kuten ennenkin. Sitten funktio tarkastaa, onko $f(x)$ laskettu jo taulukkoon d[x]. Jos $f(x)$ on laskettu, funktio palauttaa sen suoraan. Muussa tapauksessa funktio laskee arvon rekursiivisesti ja tallentaa sen kohtaan d[x].

Muistitaulukon ansiosta funktio toimii nopeasti, koska sen tarvitsee laskea vastaus kullekin x:n arvolle vain kerran rekursiivisesti. Heti kun arvo $f(x)$ on tallennettu muistitaulukkoon, sen saa haettua sieltä suoraan, kun funktiota kutsutaan seuraavan kerran parametrilla x.

Tuloksena olevan algoritmin aikavaativuus on $O(xk)$, kun rahamäärä on x ja kolikoiden määrä on k. Kiinnostava piirre aikavaativuudessa on, että rahamäärän suuruus vaikuttaa algoritmin tehokkuuteen.

7.1.3 Silmukkatoteutus

Dynaamisen ohjelmoinnin ratkaisu on mahdollista toteuttaa rekursion sijasta myös silmukalla, joka muodostaa taulukon d. Siinä missä rekursio laskee arvoja ”ylhäältä alaspäin”, silmukka laskee niitä ”alhaalta ylöspäin”.

Tässä tehtävässä silmukasta tulee:

```
d[0] = 0;
for (int i = 1; i <= x; i++) {
    int u = 1e9;
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        u = min(u, d[i-c[j]]+1);
    }
    d[i] = u;
}
```

Silmukan jälkeen taulukko d sisältää vastaukset rahamäärille $0, 1, \ldots, x$. Silmukkatoteutus on lyhyempi ja hieman tehokkaampi kuin rekursiototeutus, minkä vuoksi kokeneet kisakoodarit toteuttavat dynaamisen ohjelmoinnin laskennan usein silmukan avulla.

7.1.4 Ratkaisun muodostus

Joskus optimiratkaisun arvon lisäksi pitää muodostaa yksi mahdollinen ratkaisu. Tässä tehtävässä tämä tarkoittaa, että ohjelman täytyy antaa esimerkki tavasta valita kolikot, joista muodostuu rahamäärä x.

Ratkaisun muodostus onnistuu lisäämällä koodiin uuden taulukon, joka kertoo kullekin x:n arvolle, mikä kolikko siitä kannattaa poistaa. Seuraavassa koodissa taulukko e huolehtii asiasta:

```
d[0] = 0;
for (int i = 1; i <= x; i++) {
    d[i] = 1e9;
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        int u = d[i-c[j]]+1;
        if (u < d[i]) {
            d[i] = u;
            e[i] = c[j];
        }
    }
}
```

Tämän jälkeen rahamäärän x muodostavat kolikot voi tulostaa näin:

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

7.2 Ratkaisumäärän laskeminen

Toinen tavallinen dynaamisen ohjelmoinnin käyttötarkoitus on ratkaisuiden määrän laskeminen. Esimerkiksi kolikkotehtävästä on luonteva muunnelma, jossa tulee selvittää ratkaisuiden määrä optimiratkaisun sijaan:

> *Tehtävä:* Kolikoiden arvot ovat $\{c_1, c_2, \ldots, c_k\}$, ja tehtäväsi on muodostaa kolikoista rahamäärä x. Jokaista kolikkoa on saatavilla rajattomasti. Monellako eri tavalla voit muodostaa rahamäärän?

Esimerkiksi jos kolikot ovat $\{1,3,4\}$ ja rahamäärä on 5, niin ratkaisuja on 6:

- $1+1+1+1+1$
- $1+1+3$
- $1+3+1$
- $3+1+1$
- $1+4$
- $4+1$

Ratkaisumäärän laskeminen tapahtuu melko samalla tavalla kuin optimiratkaisun etsiminen. Erona on, että optimiratkaisun etsivässä rekursiossa valitaan yleensä pienin tai suurin aiempi arvo, kun taas ratkaisumäärän laskevassa rekursiossa lasketaan yhteen kaikki vaihtoehdot.

Tässä tapauksessa voimme muodostaa funktion $f(x)$, joka kertoo, monellako tavalla rahamäärän x voi muodostaa kolikoista. Esimerkiksi $f(5) = 6$, kun kolikot ovat $\{1,3,4\}$.

Funktion $f(x)$ saa laskettua rekursiivisesti kaavalla

$$f(x) = f(x-c_1) + f(x-c_2) + \cdots + f(x-c_k),$$

koska rahamäärän x muodostamiseksi pitää valita jokin kolikko c_i ja muodostaa sen jälkeen rahamäärä $x-c_i$. Pohjatapauksina ovat $f(0) = 1$, koska rahamäärä 0 syntyy ilman yhtään kolikkoa, sekä $f(x) = 0$, kun $x < 0$, koska negatiivista rahamäärää ei ole mahdollista muodostaa.

Yllä olevassa esimerkissä funktioksi tulee

$$f(x) = f(x-1) + f(x-3) + f(x-4),$$

ja funktion ensimmäiset arvot ovat:

$$\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(2) &= 1 \\
f(3) &= 2 \\
f(4) &= 4 \\
f(5) &= 6 \\
f(6) &= 9 \\
f(7) &= 15 \\
f(8) &= 25 \\
f(9) &= 40
\end{aligned}$$

Seuraava koodi laskee funktion $f(x)$ arvon dynaamisella ohjelmoinnilla täyttämällä taulukon d rahamäärille $0 \ldots x$:

```
d[0] = 1;
for (int i = 1; i <= x; i++) {
    for (int j = 1; j <= k; j++) {
        if (i-c[j] < 0) continue;
        d[i] += d[i-c[j]];
    }
}
```

Usein ratkaisumäärä on niin suuri, että sitä ei tarvitse laskea kokonaan vaan riittää ilmoittaa vastaus modulo M, missä esimerkiksi $M = 10^9 + 7$. Tämä onnistuu muokkaamalla koodia niin, että kaikki laskutoimitukset lasketaan modulo M. Tässä tapauksessa riittää lisätä rivin

```
        d[i] += d[i-c[j]];
```

jälkeen rivi

```
        d[i] %= M;
```

7.3 Esimerkkejä

Olemme nyt käyneet läpi tärkeimmät ideat, joihin dynaamisen ohjelmoinnin ratkaisut perustuvat. Haasteena dynaamisessa ohjelmoinnissa on oppia soveltamaan sitä erilaisissa tehtävissä. Seuraavat esimerkit antavat näytteitä dynaamisen ohjelmoinnin mahdollisuuksista.

7.3.1 Pisin nouseva alijono

Tehtävä: Taulukossa on n kokonaislukua $x_1, x_2, \ldots, x_n$. Taulukon nouseva alijono muodostuu valitsemalla taulukosta järjestyksessä lukuja niin, että jokainen luku on edellistä suurempi. Tehtäväsi on selvittää, kuinka pitkä on taulukon pisin nouseva alijono.

Esimerkiksi taulukossa

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

pisin nouseva alijono sisältää 4 lukua:

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

Ongelman rekursiivinen esitys on laskea, kuinka pitkä on pisin taulukon kohtaan k päättyvä nouseva alijono. Olkoon $f(k)$ pisimmän kohtaan k päättyvän nousevan alijonon pituus. Tätä funktiota käyttäen ratkaisu tehtävään on suurin arvoista $f(1), f(2), \ldots, f(n)$.

Esimerkkitaulukossa funktion arvot ovat:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 1 \\ f(3) &= 2 \\ f(4) &= 1 \\ f(5) &= 3 \\ f(6) &= 2 \\ f(7) &= 4 \\ f(8) &= 2 \end{aligned}$$

Esimerkiksi $f(1) = 1$, koska pisin kohtaan 0 päättyvä nouseva alijono on [6]. Vastaavasti $f(6) = 2$, mikä vastaa alijonoja [2,4] ja [1,4]. Funktion suurin arvo on kohdassa 7, koska kohtaan 7 päättyy alijono [2,5,7,8].

Arvon $f(k)$ laskemisessa on kaksi tapausta. Yksinkertainen tapaus on, että kohtaan k päättyvässä alijonossa on vain kohdan k luku x_k, jolloin $f(k) = 1$. Rekursiivisessa tapauksessa kohtaan k päättyvä alijono muodostuu yhdistämällä kohtaan $i < k$ päättyvä nouseva alijono sekä luku x_k. Koska alijonon tulee olla nouseva, taulukon luvuille täytyy päteä ehto $x_i < x_k$.

Rekursiivisessa tapauksessa $f(k)$:n voi laskea arvoista $f(1), f(2), \ldots, f(k-1)$ käymällä läpi kaikki vaihtoehdot kohdalle $i < k$. Jos $x_i < x_k$, syntyy nouseva alijono, jonka pituus on $f(i)+1$, ja arvoksi $f(k)$ tulee valita näistä arvoista suurin. Algoritmin aikavaativuus on $O(n^2)$, koska jokaisessa kohdassa täytyy käydä läpi kaikki aiemmat kohdat.

Yllättävää kyllä, tehtävään on olemassa myös $O(n \log n)$-aikainen ratkaisu. Keksitkö, miten tämä onnistuu?

7.3.2 Reitti ruudukossa

Tehtävä: Annettuna on $n \times n$ -kokoinen ruudukko, jonka jokaisessa ruudussa on luku. Tehtäväsi on etsiä reitti ruudukon vasemmasta yläkulmasta oikeaan alakulmaan niin, että lukujen summa reitillä on mahdollisimman suuri. Saat liikkua joka askeleella yhden ruudun oikealle tai alaspäin.

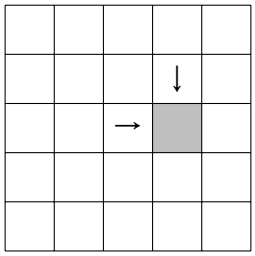
Seuraavassa ruudukossa paras reitti on merkitty harmaalla taustalla:

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.

Hyvä lähestymistapa tehtävään on laskea kuhunkin ruutuun (y,x) suurin summa reitillä vasemmasta yläkulmasta kyseiseen ruutuun. Merkitään tätä suurinta summaa $f(y,x)$, jolloin $f(n,n)$ on suurin summa reitillä vasemmasta yläkulmasta oikeaan alakulmaan.

Rekursio syntyy havainnosta, että ruutuun (y,x) saapuvan reitin täytyy tulla joko vasemmalta ruudusta $(y,x-1)$ tai ylhäältä ruudusta $(y-1,x)$:



Kun $r(y,x)$ on ruudukon luku kohdassa (y,x), rekursion pohjatapaukset ovat seuraavat:

$$\begin{aligned} f(1,1) &= r(1,1) \\ f(1,x) &= f(1,x-1)+r(1,x) \\ f(y,1) &= f(y-1,1)+r(y,1) \end{aligned}$$

Yleisessä tapauksessa valittavana on kaksi reittiä, joista kannattaa valita se, joka tuottaa suuremman summan:

$$f(y,x) = \max(f(y,x-1), f(y-1,x)) + r(y,x)$$

7.3.3 Repunpakkaus

Tehtävä: Sinulla on n tavaraa, joiden painot ovat $w_1, w_2, \ldots, w_n$ ja arvot ovat $a_1, a_2, \ldots, a_n$. Tehtäväsi on valita reppuun pakattavat tavarat niin, että painojen summa on enintään p ja arvojen summa on mahdollisimman suuri.

Esimerkiksi jos tavarat ovat

tavara	paino	arvo
A	5	1
B	6	3
C	8	5
D	5	3

ja suurin sallittu paino on 12, niin paras ratkaisu on pakata reppuun tavarat B ja D. Niiden yhteispaino $6+5=11$ on pienempi kuin 12 ja arvojen summa on $3+3=6$, mikä on paras mahdollinen tulos.

On houkuttelevaa koettaa ratkaista tehtävää ahneesti valitsemalla tavaroita, joiden arvon ja painon suhde on mahdollisimman suuri. Tämä onkin usein hyvä valintaperuste, mutta osoittautuu, että ahne algoritmi ei toimi oikein kaikissa tapauksissa.

Dynaaminen ohjelmointi löytää sen sijaan aina parhaan ratkaisun. Ideana on määritellä funktio $f(k,x)$, joka kertoo pienimmän yhteispainon, kun k ensimmäisen tavaran joukosta valitaan tavaroita niin, että arvojen summa on x. Jos valinta ei ole mahdollinen, määritellään $f(k,x) = \infty$. Ratkaisu tehtävään on suurin x:n arvo, jolle $f(n,x) \leq p$.

Rekursiivinen kaava funktion laskemiseksi on

$$f(k,x) = \min(f(k-1,x), f(k-1,x-a_k) + w_k),$$

koska kohdan k tavara joko otetaan tai oteta mukaan ratkaisuun. Pohjatapauksina $f(k,0) = 0$ ja $f(0,x) = \infty$, kun $x \neq 0$.

Seuraava taulukko näyttää funktion $f(k,x)$ arvot yllä olevassa esimerkissä. Suurin mahdollinen tavaroiden yhteisarvo on summa $\sum_{i=1}^{n} a_i = 12$, ja parhaan ratkaisun tuottava arvo $f(4,6)$ on alleviivattu.

$k \setminus x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	0	5	∞	6	11	∞	∞	∞	∞	∞	∞	∞	∞
3	0	5	∞	6	11	8	13	∞	14	19	∞	∞	∞
4	0	5	∞	5	10	8	<u>11</u>	16	13	18	∞	19	24

7.3.4 Editointietäisyys

Editointietäisyys (*edit distance*) on pienin määrä operaatioita, joilla merkkijonon saa muutettua toiseksi. Sallitut operaatiot ovat:

- merkin lisäys (esim. `ABC` → `ABCA`)
- merkin poisto (esim. `ABC` → `AC`)
- merkin muutos (esim. `ABC` → `ADC`)

Esimerkiksi merkkijonojen `TALO` ja `PALLO` editointietäisyys on 2, koska voi tehdä seuraavat operaatiot: `TALO` → `TALLO` → `PALLO`.

Merkkijonojen x ja y editointietäisyyden voi laskea tehokkaasti dynaamisella ohjelmoinnilla. Ideana on määritellä funktio $f(a,b)$, joka on editointietäisyys x:n a ensimmäisen merkin sekä y:n b:n ensimmäisen merkin välillä.

Funktion pohjatapaukset ovat

$$\begin{aligned} f(0,b) &= b \\ f(a,0) &= a \end{aligned}$$

ja yleisessä tapauksessa pätee kaava

$$f(a,b) = \min(f(a,b-1)+1, f(a-1,b)+1, f(a-1,b-1)+c),$$

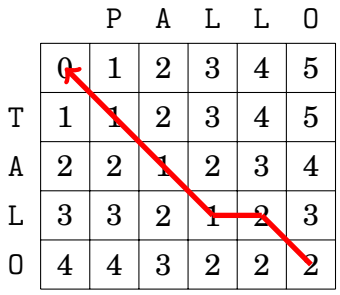
missä $c = 0$, jos x:n merkki a ja y:n merkki b ovat samat, ja muussa tapauksessa $c = 1$. Kaava käy läpi mahdollisuudet lyhentää merkkijonoja:

- $f(a, b-1)$ tarkoittaa, että x:ään lisätään merkki
- $f(a-1, b)$ tarkoittaa, että x:stä poistetaan merkki
- $f(a-1, b-1)$ tarkoittaa, että x:ssä ja y:ssä on sama merkki ($c = 0$) tai x:n merkki muutetaan y:n merkiksi ($c = 1$)

Seuraava taulukko sisältää funktion f arvot esimerkin tapauksessa:

		P	A	L	L	O
	0	1	2	3	4	5
T	1	1	2	3	4	5
A	2	2	1	2	3	4
L	3	3	2	1	2	3
O	4	4	3	2	2	2

Taulukosta pystyy myös lukemaan, miten pienimmän editointietäisyyden voi saavuttaa. Esimerkissä mahdollinen polku on seuraava:



Merkkijonojen PALLO ja TALO viimeinen merkki on sama, joten niiden editointietäisyys on sama kuin merkkijonojen PALL ja TAL. Nyt voidaan poistaa viimeinen L merkkijonosta PAL, mistä tulee yksi operaatio. Editointietäisyys on siis yhden suurempi kuin merkkijonoilla PAL ja TAL, jne.

Luku 8

Kyselyrakenteet

Kyselyrakenteet mahdollistavat tehokkaat kyselyt taulukkoon tai muuhun tietorakenteeseen. Tyypillinen esimerkki kyselystä on välikysely (*range query*), jossa annettuna on taulukon väli ja tehtävänä on laskea nopeasti lukujen summa, minimi tai muu tieto väliltä.

Tehokkaat kyselyt perustuvat yleensä esikäsittelyyn, mikä tarkoittaa, että ennen kyselyitä käydään läpi koko tietorakenne ja koostetaan sen perusteella kyselyiden tekemistä helpottava kyselyrakenne. Tämän jälkeen vastaus mihin tahansa kyselyyn selviää kyselyrakenteesta.

8.1 Summakysely

Summakyselyssä tehtävänä on laskea taulukon välin alkioiden summa. Summakyselyyn on mahdollista vastata $O(1)$-ajassa luomalla taulukosta summataulukko (*prefix sum array*). Summataulukko kertoo jokaiselle taulukon kohdalle, mikä on lukujen summa kyseiseen kohtaan mennessä.

Esimerkiksi taulukon

1	3	4	8	6	1	4	2

summataulukko on:

1	4	8	16	22	23	27	29

Summataulukon voi muodostaa $O(n)$-ajassa alkuperäisestä taulukosta. Tämän jälkeen minkä tahansa taulukon välin summan voi laskea $O(1)$-ajassa kahdesta summataulukon arvosta. Riittää näet vähentää välin lopussa olevasta summasta välin alkua edeltävä summa.

Esimerkiksi välin

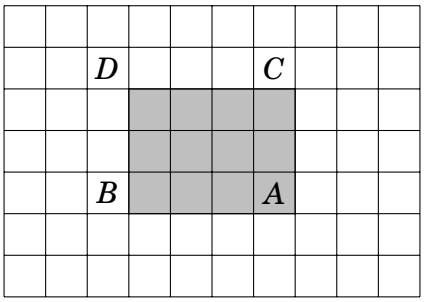
1	3	4	**8**	**6**	**1**	**4**	2

summan $8 + 6 + 1 + 4 = 19$ saa summataulukosta laskemalla $27 - 8 = 19$:

1	4	**8**	16	22	23	**27**	29

Summataulukon idean voi yleistää myös kaksiulotteiseen taulukkoon, jolloin summataulukon avulla voi laskea minkä tahansa suorakulmaisen alueen summan $O(1)$-ajassa. Ideana on tallentaa summataulukkoon summia alueista, jotka alkavat taulukon vasemmasta yläkulmasta.

Seuraava ruudukko havainnollistaa asiaa:



Harmaan 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.

8.2 Minimikysely

Minimikyselyssä tehtävänä on selvittää taulukon välin pienin alkio. Myös minimikyselyyn on mahdollista vastata $O(1)$-ajassa sopivan esikäsittelyn avulla, joskin tämä on vaikeampaa kuin summakyselyssä.

Ideana on laskea esikäsittelynä minimi jokaisesta taulukon välistä, jonka pituus on muotoa 2^k. Tällaisia minimeitä on yhteensä $O(n \log n)$, koska taulukon joka kohdasta alkaa $O(\log n)$ 2^k-väliä. Tämän jälkeen minkä tahansa välin minimin saa laskettua miniminä välin alussa ja lopussa olevasta 2^k-välistä.

Esimerkiksi taulukosta

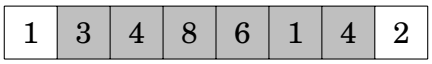
1	3	4	8	6	1	4	2

lasketaan esikäsittelynä seuraavat minimit:

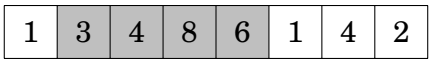
2^0	1	3	4	8	6	1	4	2
2^1	1	3	4	6	1	1	2	–
2^2	1	3	1	1	1	–	–	–
2^3	1	–	–	–	–	–	–	–

Nyt välin $[a,b]$ minimin saa laskettua miniminä kahdesta 2^k-välistä, joista ensimmäinen alkaa kohdasta a ja toinen päättyy kohtaan b. Pituus 2^k on valittu niin, että se on suurin mahdollinen pituus, joka ei ylitä välin pituutta $b-a+1$. Tämän ansiosta kaksi 2^k-kokoista väliä kattaa koko välin $[a,b]$.

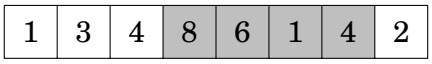
Esimerkiksi välin



minimi lasketaan 4:n pituisten välien



ja



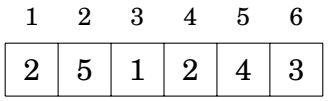
minimeistä. Ensimmäisen välin minimi on 3 ja toisen välin minimi on 1, joten koko välin minimi on 1.

Taulukon esikäsittely vie aikaa $O(n \log n)$, koska jokaisen 2^k-välin minimin saa laskettua miniminä siihen kuuluvista 2^{k-1}-väleistä. Esikäsittelyn jälkeen minkä tahansa välin minimin saa selville $O(1)$-ajassa.

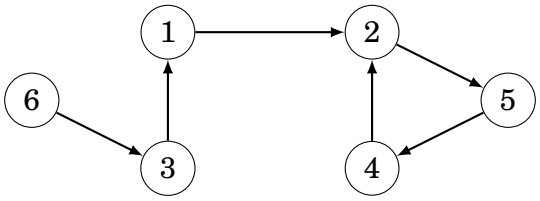
8.3 Hyppytaulukko

Oletetaan, että n solmun välillä on linkkejä niin, että jokaisesta solmusta on tarkalleen yksi linkki ulospäin toiseen solmuun. Tällaisen rakenteen kuvaa taulukko, jossa on n alkiota ja jokainen alkio on luku välillä $1 \ldots n$.

Esimerkiksi taulukko



kuvaa rakenteen



Tehtävänä on laskea tehokkaasti arvo $f(x,m)$: mihin solmuun päätyy solmusta x kulkemalla m askelta. Esimerkiksi yllä olevassa tilanteessa $f(3,5) = 2$, koska muodostuu ketju $3 \to 1 \to 2 \to 5 \to 4 \to 2$.

Solmun $f(x,m)$ laskeminen tehokkaasti onnistuu muodostamalla hyppytaulukko, johon on laskettu etukäteen kaikki arvot muotoa $f(x,2^k)$ riittävän suureen k-arvoon asti. Tämän jälkeen solmun $f(x,m)$ saa laskettua ajassa $O(\log m)$ kulkemalla hyppytaulukossa mahdollisimman suuria 2^k-hyppyjä.

Hyppytaulukon pystyy muodostamaan tehokkaasti laskemalla arvot kasvavassa k:n järjestyksessä, koska $f(x,2^k) = f(f(x,2^{k-1}),2^{k-1})$. Esimerkiksi yllä olevassa tilanteessa muodostuu seuraava hyppytaulukko:

	1	2	3	4	5	6
2^0	2	5	1	2	4	3
2^1	5	4	2	5	2	1
2^2	2	5	4	2	4	5
2^3	5	4	2	5	2	4

Solmu $f(x,m)$ selviää hyppytaulukosta $O(\log m)$-ajassa jakamalla m:n askeleen reitti mahdollisimman suuriin 2^k-pituisiin osiin. Esimerkiksi solmu $f(3,5)$ selviää hyppytaulukosta laskemalla reitti osissa $f(f(3,4),1) = f(4,1) = 2$.

8.4 Seuraajataulukko

Seuraajataulukko kertoo jokaiselle taulukon luvulle, mikä on seuraava sitä suurempi luku taulukossa. Poikkeuksena jos taulukossa ei ole seuraavaa suurempaa lukua, niin seuraajataulukossa on merkintä –.

Esimerkiksi taulukon

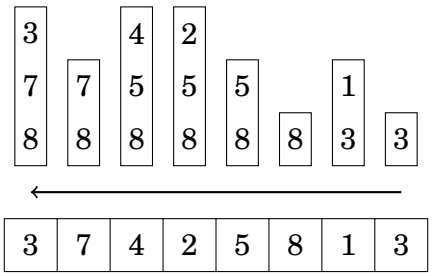
3	7	4	2	5	8	1	3

seuraajataulukko on

7	8	5	5	8	–	3	–

Seuraajataulukko on mahdollista rakentaa tehokkaasti pinon avulla käymällä läpi taulukko lopusta alkuun. Joka vaiheessa pinosta poistetaan lukuja, kunnes vastaan tulee luku, joka on käsiteltävää lukua suurempi. Tämä luku on seuraava suurempi luku taulukossa, tai jos pino on tyhjä, niin seuraavaa suurempaa lukua ei ole. Lopuksi käsiteltävä luku lisätään pinoon.

Esimerkin taulukossa pinon sisältö muuttuu näin:



3	7	4	2	5	8	1	3

Algoritmin aikavaativuus on $O(n)$, koska jokainen taulukon luku lisätään korkeintaan kerran pinoon ja poistetaan korkeintaan kerran pinosta, minkä ansiosta pino-operaatioita tehdään yhteensä $O(n)$.

Luku 9

Segmenttipuu

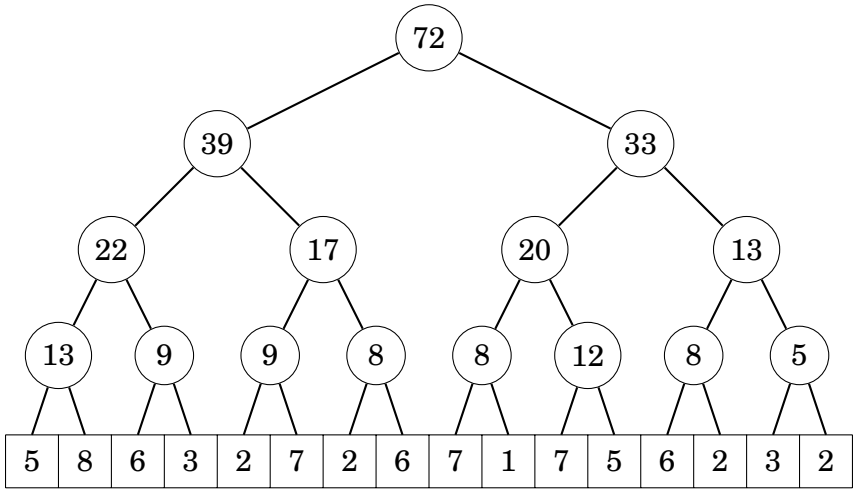
Segmenttipuu (*segment tree*) on tietorakenne, jonka avulla taulukkoon voi toteuttaa tehokkaasti monenlaisia välikyselyitä. Tavallisia välikyselyitä ovat taulukon välin summan, minimin ja maksimin laskeminen. Segmenttipuu mahdollistaa sekä välikyselyn että taulukon muuttamisen ajassa $O(\log n)$.

Segmenttipuuta voi ajatella luvun 8.1 summataulukon yleistyksenä. Siinä on kaksi etua summataulukkoon nähden. Ensinnäkin segmenttipuu sallii taulukon muuttamisen, mikä ei ole mahdollista summataulukossa. Lisäksi segmenttipuun avulla voi toteuttaa muitakin välikyselyitä kuin summan laskemisen.

9.1 Rakenne

Segmenttipuun alimmalla tasolla on taulukon sisältö ja ylemmillä tasoilla on välikyselyihin tarvittavaa tietoa. Oletamme seuraavaksi, että taulukko sisältää lukuja ja välikysely laskee välin lukujen summan.

Seuraavassa kuvassa on 16-alkioista taulukkoa vastaava segmenttipuu, joka on tarkoitettu summakyselyihin. Jokainen puun solmu sisältää kahden alemman tason solmun summan, ja puun lehdet ovat taulukon alkioita.



Segmenttipuu on mukavinta rakentaa niin, että taulukon koko on 2:n potenssi, jolloin 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.

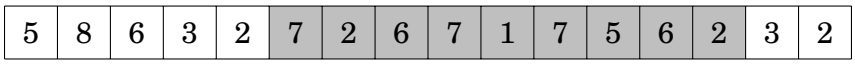
9.2 Operaatiot

Segmenttipuun operaatiot ovat välikyselyn suorittaminen sekä taulukon muuttaminen. Puurakenteen ansiosta kumpikin operaatio on mahdollista toteuttaa tehokkaasti ajassa $O(\log n)$. Tutustumme nyt operaatioihin käyttäen esimerkkinä edelleen summan laskemista taulukossa.

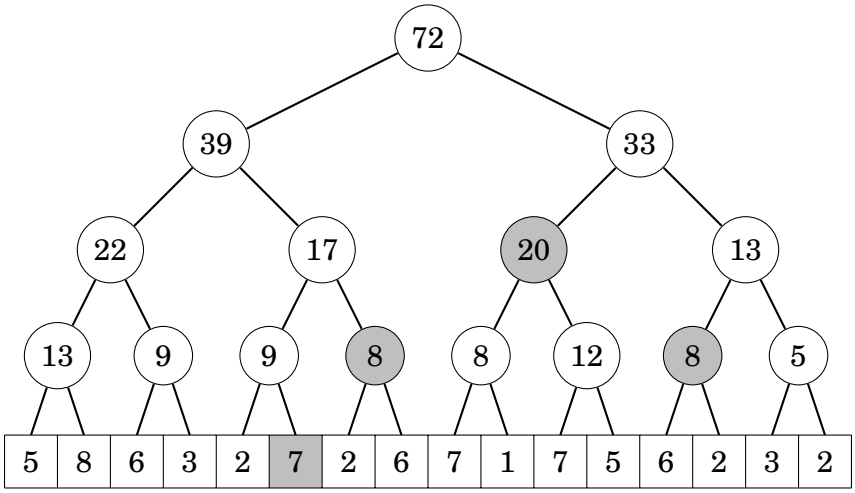
9.2.1 Välikysely

Välikysely laskee annetun taulukon välin lukujen summan käyttäen apuna välien osasummia segmenttipuun solmuissa. Välikyselyssä on ideana muodostaa summa mahdollisimman korkealla puussa olevista osasummista, jotta summan saa laskettua tehokkaasti.

Esimerkiksi taulukon välin



summa muodostuu seuraavista osista:

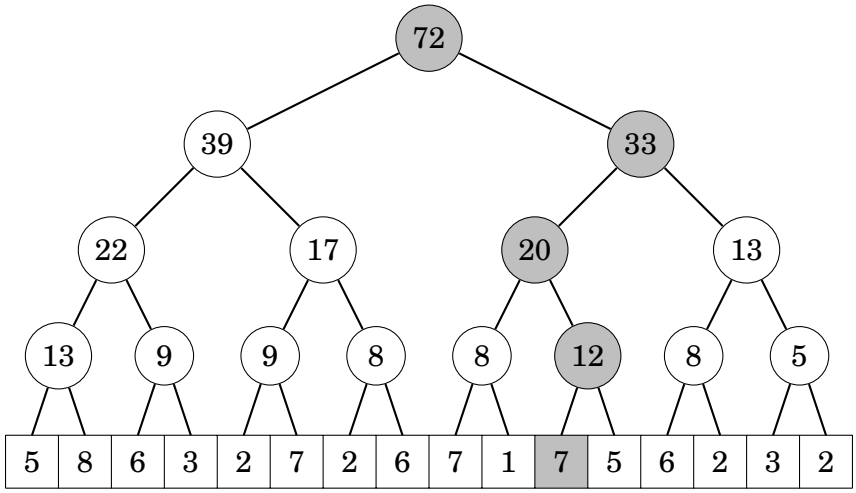


Taulukon välin summa saadaan laskemalla yhteen osien summat, eli tässä tapauksessa summaksi tulee $7+8+20+8=43$.

Osoittautuu, että kun taulukon välin summa lasketaan mahdollisimman korkealla segmenttipuussa olevista osasummista, summaan kuuluu enintään kaksi osasummaa jokaiselta segmenttipuun tasolta. Tämän ansiosta osasummien yhteismäärä on luokkaa $O(\log n)$.

9.2.2 Muuttaminen

Kun taulukon arvo muuttuu, segmenttipuussa täytyy päivittää kaikkia solmuja, joiden arvo riippuu muutetusta taulukon kohdasta. Tämä tapahtuu kulkemalla puuta ylöspäin huipulle asti ja tekemällä muutokset. Seuraava kuva näyttää, mihin puun solmuihin taulukossa oleva arvo vaikuttaa.



9.2.3 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:

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

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

Ideana on aloittaa summan laskeminen segmenttipuun pohjalta ja liikkua askel kerrallaan ylemmälle tasolle. Funktio laskee summan muuttujaan s yh-

distämällä puussa olevia osasummia. Välin reunalla oleva osasumma lisätään summaan aina silloin, kun se ei kuulu ylemmän tason osasummaan.

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];
    }
}
```

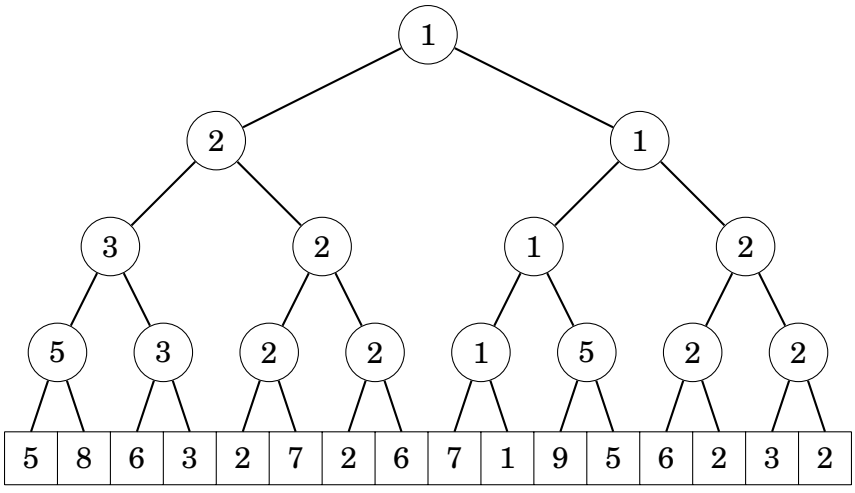
Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasummat puun huipulle asti. Taulukon `p` indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k+1$.

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.

9.2.4 Muut kyselyt

Segmenttipuu mahdollistaa summan lisäksi minkä tahansa välikyselyn, jonka pystyy laskemaan osissa summaa vastaavasti. Kyselyn voi toteuttaa segmenttipuuna, jos vierekkäisten välien $[a,b]$ ja $[c,d]$ tuloksista pystyy laskemaan tehokkaasti välin $[a,d]$ tuloksen. Tällaisia kyselyitä ovat esimerkiksi minimi ja maksimi, suurin yhteinen tekijä sekä bittioperaatiot and, or ja xor.

Esimerkiksi tässä on taulukosta tehty minimipuu:



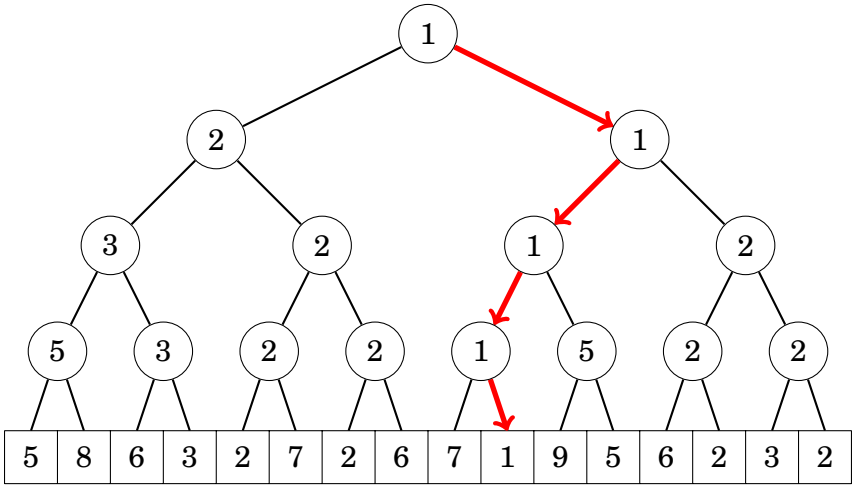
Tässä segmenttipuussa jokainen puun solmu kertoo, mikä on pienin luku sen alapuolella olevassa taulukon osassa. Segmenttipuun ylin luku on pienin luku koko taulukon alueella. Puun toteutus on samanlainen kuin summan laskemisessa, mutta joka kohdassa pitää laskea summan sijasta lukujen minimi.

9.3 Lisätekniikoita

9.3.1 Binäärihaku puussa

Segmenttipuun sisältämää tietoa voi käyttää binäärihaun kaltaisesti aloittamalla haun puun huipulta. Näin on mahdollista selvittää esimerkiksi minimisegmenttipuusta $O(\log n)$-ajassa, missä kohdassa on taulukon pienin luku.

Esimerkiksi seuraavassa puussa pienin alkio on 1, jonka sijainti löytyy kulkemalla puussa huipulta alaspäin:



9.3.2 Indeksien pakkaus

Segmenttipuun rajoituksena on, että se on rakennettu taulukon päälle ja alkiot on indeksoitu kokonaisluvuin $0, 1, 2, \ldots, N-1$. Tästä seuraa ongelmia, jos tarvittavat indeksit ovat suuria. Esimerkiksi indeksin 10^9 käyttäminen vaatisi, että taulukossa on 10^9 alkiota, mikä ei ole realistista.

Tätä rajoitusta on kuitenkin mahdollista kiertää usein käyttämällä indeksien pakkausta. Se tarkoittaa, että n indeksiä jaetaan uudestaan niin, että ne ovat välillä $0, 1, 2, \ldots, n-1$. Tämä onnistuu silloin, jos kaikki algoritmin aikana tarvittavat indeksit ovat tiedossa algoritmin alussa.

Ideana on korvata jokainen alkuperäinen indeksi x indeksillä $p(x)$, missä p jakaa indeksit uudestaan. Vaatimuksena on, että indeksien järjestys ei muutu, eli jos $a < b$, niin $p(a) < p(b)$, minkä ansiosta segmettipuuta voi käyttää melko tavallisesti indeksien pakkauksesta huolimatta.

Esimerkiksi jos alkuperäiset indeksit ovat 555, 10^9 ja 8, ne muuttuvat näin:

$$\begin{aligned} p(8) &= 0 \\ p(555) &= 1 \\ p(10^9) &= 2 \end{aligned}$$

9.3.3 Välin muuttaminen

Tähän mennessä segmenttipuun operaatiot ovat olleet välikysely ja yksittäisen arvon muuttaminen. Tarkastellaan lopuksi tilannetta, jossa pitääkin muuttaa välejä ja kysellä yksittäisiä arvoja. Tarkemmin sanoen haluamme toteuttaa operaation, joka kasvattaa kaikkia välin arvoja x:llä.

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

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

tulee seuraava:

1	2	3	4	5	6	7	8
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 5 saadaan summana $3-2+4=5$.

Uuden tallennustavan etuna on, että välin muuttamiseen riittää muuttaa kahta taulukon kohtaa. Esimerkiksi jos välille $2\ldots5$ lisätään luku 5, taulukon kohtaan 2 lisätään 5 ja taulukon kohdasta 6 poistetaan 5. Tulos on tässä:

1	2	3	4	5	6	7	8
3	5	−2	0	0	−1	−3	0

Yleisemmin kun taulukon välille $a\ldots b$ lisätään x, taulukon kohtaan a lisätään x ja taulukon kohdasta $b+1$ vähennetään x. Tarvittavat operaatiot ovat summan laskeminen taulukon alusta tiettyyn kohtaan sekä yksittäisen alkion muuttaminen, eli tutut segmenttipuun operaatiot.

Luku 10

Bittien käsittely

Tietokone käsittelee tietoa sisäisesti bitteinä eli lukuina 0 ja 1. Tässä luvussa tutustumme kokonaisluvun bittiesitykseen sekä bittioperaatioihin, jotka muokkaavat tehokkaasti luvun bittejä. Osoittautuu, että näistä operaatioista on monenlaista hyötyä algoritmien ohjelmoinnissa.

Tyypillinen tekniikka on luvun bittiesityksen rinnastaminen joukon osajoukkoon. Tällöin joukko-operaatiot voi toteuttaa tehokkaasti bittioperaatioina ja bittiesitystä voi käyttää esimerkiksi dynaamisen ohjelmoinnin tilana.

10.1 Bittiesitys

Kokonaisluvun bittiesitys ilmaisee, mistä 2:n potensseista luku muodostuu. Esimerkiksi luvun 43 bittiesitys on 101011, koska $43 = 2^5 + 2^3 + 2^1 + 2^0$ eli oikealta lukien bitit 0, 1, 3 ja 5 ovat ykkösiä ja kaikki muut bitit ovat nollia.

Tietokoneessa luvut tallennetaan niin, että käytössä on tietty määrä bittejä. Esimerkiksi C++:n `int`-tyyppi on tavallisesti 32-bittinen, jolloin `int`-luku tallennetaan 32 bittinä. Tällöin luvun 43 esitys `int`-lukuna on

$$00000000000000000000000000101011.$$

Luvun bittiesitys on joko etumerkillinen (*signed*) tai etumerkitön (*unsigned*). Etumerkillisen bittiesityksen ensimmäinen bitti on etumerkki (+ tai −) ja n bitillä voi esittää luvut $-2^{n-1} \ldots 2^{n-1} - 1$. Jos taas bittiesitys on etumerkitön, kaikki bitit kuuluvat lukuun ja n bitillä voi esittää luvut $0 \ldots 2^n - 1$.

Etumerkillisessä bittiesityksessä ei-negatiivisen luvun ensimmäinen bitti on 0 ja negatiivisen luvun ensimmäinen bitti on 1. Bittiesityksenä on kahden komplementti (*two's complement*), jossa positiivisesta luvusta saa negatiivisen muuttamalla kaikki bitit käänteiseksi ja lisäämällä tulokseen yksi.

Esimerkiksi luvun -43 esitys `int`-lukuna on

$$11111111111111111111111111010101.$$

Etumerkillisen ja etumerkittömän bittiesityksen yhteys on, että etumerkillisen luvun $-x$ ja etumerkittömän luvun $2^n - x$ bittiesitykset ovat samat. Niinpä yllä oleva bittiesitys tarkoittaa etumerkittömänä lukua $2^{32} - 43$.

C++:ssa luvut ovat oletuksena etumerkillisiä, mutta avainsanan `unsigned` avulla luvusta saa etumerkittömän. Esimerkiksi koodissa

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

etumerkillistä lukua $x = -43$ vastaa etumerkitön luku $y = 2^{32} - 43$.

Jos luvun suuruus menee käytössä olevan bittiesityksen ulkopuolelle, niin luku pyörähtää ympäri. Etumerkillisessä bittiesityksessä luvusta $2^{n-1} - 1$ seuraava luku on -2^{n-1} ja vastaavasti etumerkittömässä bittiesityksessä luvusta $2^n - 1$ seuraava luku on 0. Esimerkiksi koodissa

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

muuttuja x pyörähtää ympäri luvusta $2^{31} - 1$ lukuun -2^{31}.

10.2 Bittioperaatiot

C++:n bittioperaatiot ovat:

- x & y (and) tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa molemmissa luvuissa x ja y on ykkösbitti
- x | y (or) tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa ainakin toisessa luvuista x ja y on ykkösbitti
- x ^ y (xor) tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa tarkalleen yhdessä luvuista x ja y on ykkösbitti
- ~x (not) tuottaa luvun, jossa kaikki x:n bitit on muutettu käänteisiksi
- x << k tuottaa luvun, jossa x:n bittejä on siirretty k askelta vasemmalle (eli luvun loppuun tulee k nollabittiä)
- x >> k tuottaa luvun, jossa x:n bittejä on siirretty k askelta oikealle (eli luvun lopusta lähtee pois k viimeistä bittiä)

Seuraava koodi esittelee and-, or- ja xor-operaatioita:

```
int x = 22;   // 22 = 10110
int y = 26;   // 26 = 11010
cout << (x&y) << "\n"; // 18 = 10010
cout << (x|y) << "\n"; // 30 = 11110
cout << (x^y) << "\n"; // 12 = 01100
```

Not-operaatio (~) vaikuttaa luvun kaikkiin bitteihin, joten sen toiminta riippuu luvun tyypistä. Operaatiolle pätee kaava $\sim x = -x - 1$.

```
int x = 22;              // 22 = 00000000000000000000000000010110
cout << (~x) << "\n"; // -23 = 11111111111111111111111111101001
```

Bittisiirto $x << k$ vastaa luvun x kertomista luvulla 2^k, ja bittisiirto $x >> k$ vastaa luvun x jakamista luvulla 2^k alaspäin pyöristäen.

```
int x = 5; // 5 = 101
cout << (x<<1) << "\n"; // 10 = 1010
cout << (x<<2) << "\n"; // 20 = 10100
cout << (x<<3) << "\n"; // 40 = 101000
int y = 142; // 142 = 10001110
cout << (y>>1) << "\n"; // 71 = 1000111
cout << (y>>2) << "\n"; // 35 = 100011
cout << (y>>3) << "\n"; // 17 = 10001
```

Sovelluksia

Luku x on parillinen tarkalleen silloin, kun $x \,\&\, 1 = 0$. Yleisemmin luku x on jaollinen luvulla 2^k tarkalleen silloin, kun $x \,\&\, ((1 << k) - 1) = 0$.

Esimerkiksi koodin

```
if (x%2 == 0) {
```

voi kirjoittaa myös

```
if ((x&1) == 0) {
```

Luvun bitit indeksoidaan oikealta vasemmalle nollasta alkaen. Luvun x bitti k on ykkösbitti, jos $x \,\&\, (1 << k)$ on positiivinen. Lauseke $x \mid (1 << k)$ asettaa luvun x bitin k ykköseksi, lauseke $x \,\&\, \sim(1 << k)$ asettaa luvun x bitin k nollaksi ja lauseke $x \,\hat{}\, (1 << k)$ muuttaa luvun x bitin k käänteiseksi.

Seuraava koodi muuttaa luvun bittejä:

```
int x = 181; // 10110101
cout << (x|(1<<2)) << "\n"; // 181 = 10110101
cout << (x|(1<<3)) << "\n"; // 189 = 10111101
cout << (x&~(1<<2)) << "\n"; // 177 = 10110001
cout << (x&~(1<<3)) << "\n"; // 181 = 10110101
cout << (x^(1<<2)) << "\n"; // 177 = 10110001
cout << (x^(1<<3)) << "\n"; // 189 = 10111101
```

Lauseke $x \,\&\, (x-1)$ muuttaa luvun x viimeisen ykkösbitin nollaksi, ja lauseke $x \,\&\, -x$ nollaa luvun x kaikki bitit paitsi viimeisen ykkösbitin. Lauseke $x \mid (x-1)$ vuorostaan muuttaa kaikki viimeisen ykkösbitin jälkeiset bitit ykkösiksi. Huomaa myös, että positiivinen luku x on muotoa 2^k, jos $x \,\&\, (x-1) = 0$.

Seuraava koodi esittelee operaatioita:

```
int x = 168; // 10101000
cout << (x&(x-1)) << "\n"; // 160 = 10100000
cout << (x&-x) << "\n"; // 8 = 00001000
cout << (x|(x-1)) << "\n"; // 175 = 10101111
```

Lisäfunktiot

GCC:n g++-kääntäjä sisältää mm. seuraavat funktiot bittien käsittelyyn:

- `__builtin_clz`(x): nollien määrä bittiesityksen alussa
- `__builtin_ctz`(x): nollien määrä bittiesityksen lopussa
- `__builtin_popcount`(x): ykkösten määrä bittiesityksessä
- `__builtin_parity`(x): ykkösten määrän parillisuus

Yllä olevat funktiot käsittelevät `int`-lukuja, mutta funktioista on myös `long long` -versiot, joiden lopussa on pääte `ll`.

Seuraava koodi esittelee funktioiden käyttöä:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

10.3 Bittitaulukko

Bittitaulukko (*bit array*) on tietorakenne, joka perustuu luvun bittiesitykseen. Ideana on, että kukin luvun biteistä on yksi taulukon alkio, jolloin taulukon alkioiden määrä on sama kuin luvun bittien määrä. Esimerkiksi 32-bittistä `int`-tyyppiä käyttäessä bittitaulukon koko on 32 alkiota.

Bittitaulukon indeksit ovat $0 \ldots n-1$, missä n on luvun bittien määrä. Luvun bitit on indeksoitu bittiesityksen lopusta alkaen. Esimerkiksi luvun 1234 bittiesitys on 10011010010, joten vastaava `int`-tyyppinen bittitaulukko on:

kohta	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$	31
arvo	0	1	0	0	1	0	1	1	0	0	1	0	$\cdots$	0

Bittitaulukon alkioita käsitellään bittioperaatioiden avulla. Esimerkiksi seuraava koodi sijoittaa taulukon `t` kohtaan 5 arvon 1:

```
t |= (1<<5);
```

10.3.1 Osajoukot

Kätevä bittitaulukon käyttötarkoitus on joukon osajoukon esittäminen. Tässä esityksessä joukossa on yhtä monta alkiota kuin luvussa on bittejä, ja ykkösbitti tarkoittaa, että alkio kuuluu osajoukkoon.

Esimerkiksi `int`-luvun avulla voi esittää joukon $\{0,1,\ldots,31\}$ osajoukon. Äskeisen esimerkin mukaisesti 1234 tarkoittaa osajoukkoa $\{1,4,6,7,10\}$.

Bittioperaatioilla voi toteuttaa tehokkaasti joukko-operaatioita:

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

Osajoukkojen läpikäynti

Seuraava koodi käy läpi joukon $\{0,1,\ldots,n-1\}$ osajoukot:

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

Seuraava koodi käsittelee osajoukot, joissa on k alkiota:

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

Seuraava koodi etsii vektorin v osajoukot, joissa lukujen summa on x:

```
vector<int> v = {3,2,5,4,7};
int n = v.size();
for (int b = 0; b < (1<<n); b++) {
    int s = 0;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) s += v[i];
    }
    if (s == x) {
        // osajoukko löytyi
    }
}
```

10.3.2 Dynaaminen ohjelmointi

Bittitaulukkoa voi myös käyttää dynaamisen ohjelmoinnin tilan osana. Näin on esimerkiksi seuraavassa tehtävässä:

> *Tehtävä:* Montako permutaatiota voit muodostaa luvuista $1,2,\ldots,n$ niin, että missään kohdassa ei ole kahta peräkkäistä lukua? Esimerkiksi kun $n = 4$, ratkaisuja on 2: $(2,4,1,3)$ ja $(3,1,4,2)$.

Suoraviivainen ratkaisutapa on käydä läpi lukujen $1,2,\ldots,n$ permutaatiot, missä kuluu aikaa $O(n!)$. Kuitenkin dynaamisen ohjelmoinnin avulla on mahdollista toteuttaa $O(2^n n^2)$-aikainen ratkaisu, joka käy läpi lukujen $1,2,\ldots,n$ osajoukot ja eri vaihtoehdot valita viimeinen luku.

Ideana on merkitä $f(x,k)$:llä, monellako tavalla osajoukon x luvut voi järjestää niin, että viimeinen luku on k ja missään kohdassa ei ole kahta peräkkäistä lukua. Funktion f avulla ratkaisu tehtävään on summa

$$\sum_{i=1}^{n} f(\{1,\ldots,n\},i).$$

Jatkossa indeksoimme lukuväliä $0,1,\ldots,n-1$, koska tämä helpottaa bittien käsittelyä. Dynaamisen ohjelmoinnin tilat voi tallentaa seuraavasti:

```
ll d[1<<n][n];
```

Perustapauksena $f(\{k\},k) = 1$ kaikilla k:n arvoilla:

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

Tämän jälkeen muut funktion arvot saa laskettua seuraavasti:

```
for (int b = 0; b < (1<<n); b++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (abs(i-j) > 1 && (b&(1<<i)) && (b&(1<<j))) {
                d[b][i] += d[b^(1<<i)][j];
            }
        }
    }
}
```

Koodissa b on osajoukon bittiesitys ja j ja i ovat kaksi viimeistä lukua osajoukkoa vastaavassa permutaation alkuosassa. Vaatimukset ovat, että lukujen i ja j etäisyyden tulee olla yli 1 ja lukujen tulee olla osajoukossa b.

Lopuksi ratkaisujen määrän saa laskettua näin muuttujaan s:

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

Osa II

Verkkoalgoritmit

Luku 11

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.

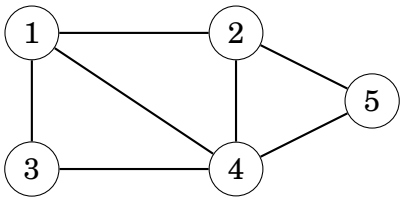
Tutustumme kirjan tässä osassa moniin verkko-ongelmiin sekä tehokkaisiin algoritmeihin niiden ratkaisemiseen. Ensin on kuitenkin paikallaan luoda katsaus siihen, mitä verkot ovat ja miten niitä voi käsitellä algoritmeissa.

11.1 Määritelmiä

Verkko (*graph*) on tietorakenne, joka muodostuu solmuista (*vertex*) ja kaaristä (*edge*). Esimerkiksi tieverkostossa verkon solmut ovat kaupunkeja ja kaaret ovat niiden välisiä teitä.

Tässä kirjassa kirjain n ilmaisee verkon solmujen määrän, ja solmut on numeroitu $1,2,\ldots,n$. Kirjain m taas ilmaisee verkon kaarten määrän.

Esimerkiksi seuraavassa verkossa on 5 solmua ja 7 kaarta:

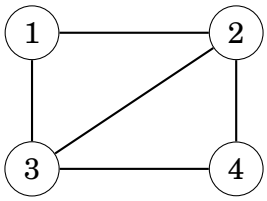


Polku (*path*) on kaaria pitkin kulkeva reitti kahden solmun välillä. Yllä olevassa verkossa solmusta 1 solmuun 5 voi kulkea esimerkiksi polkua $1 \to 4 \to 5$ tai $1 \to 3 \to 4 \to 2 \to 5$. Solmun naapuri (*neighbor*) on toinen solmu, johon solmusta pääsee kaarta pitkin, ja solmun aste (*degree*)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

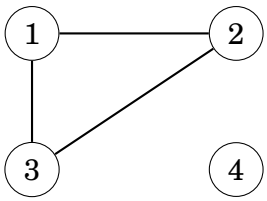
11.1.1 Yhtenäisyys

Verkko on yhtenäinen (*connected*), 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:

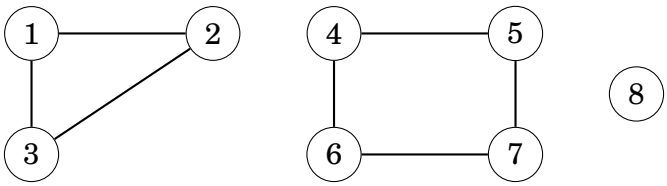


Seuraava verkko taas ei ole yhtenäinen, koska solmu 4 on erillään muista.



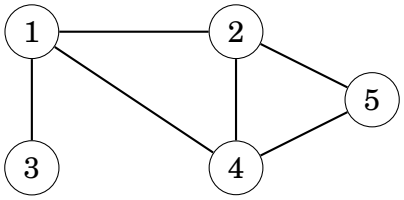
11.1.2 Komponentit

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

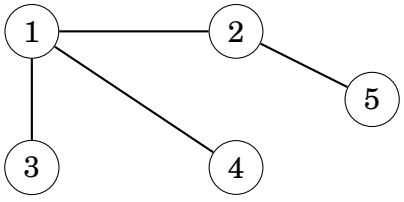


11.1.3 Syklit

Sykli (*cycle*) on polku, jonka alku- ja loppusolmu on sama ja jossa ei ole samaa kaarta monta kertaa. Verkko on syklinen (*cyclic*), jos siinä on sykli. Esimerkiksi seuraava verkko on syklinen, koska siinä on sykli $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$.

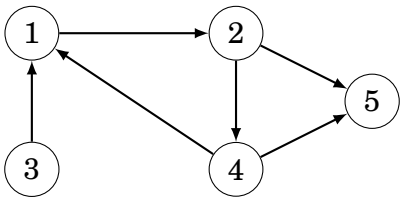


Seuraava verkko taas on syklitön (*acyclic*), eli siinä ei ole sykliä:



11.1.4 Kaarten suunnat

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

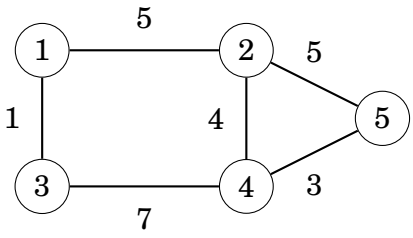


Tässä verkossa on esimerkiksi polku $3 \to 1 \to 2 \to 5$ sekä sykli $2 \to 4 \to 1 \to 2$. Solmuun 3 ei pääse mistään muusta solmusta, ja vastaavasti solmusta 5 ei pääse mihinkään muuhun solmuun.

Jos verkon kaarilla ei ole suuntaa, verkko on suuntaamaton (*undirected*).

11.1.5 Kaarten painot

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



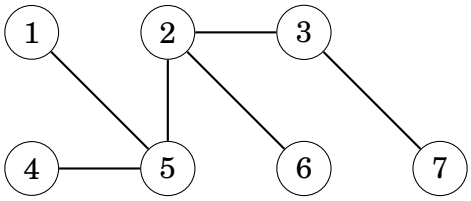
Polun pituus (*path length*) on sen kaarten painojen summa. Esimerkiksi polun $1 \to 2 \to 4$ pituus on $5 + 4 = 9$ ja polun $1 \to 3 \to 4$ pituus on $1 + 7 = 8$. Jälkimmäinen polku on lyhin polku (*shortest path*) solmusta 1 solmuun 4.

Jos verkon kaarilla ei ole painoja, verkko on painottamaton (*unweighted*).

11.1.6 Puu

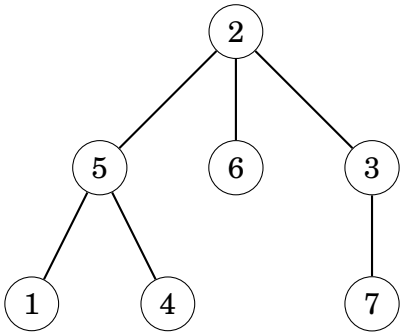
Puu (*tree*) on yhtenäinen, syklitön ja suuntaamaton verkko. Jos verkko on puu, niin sen jokaisen kahden solmun välillä on yksikäsitteinen polku.

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.

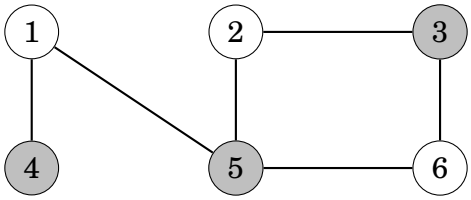
Puu esitetään usein niin, että yksi solmuista nostetaan puun juureksi (*root*) ja muut sijoittuvat tasoittain sen alapuolelle. Esimerkiksi jos äskeisessä verkossa solmusta 2 tehdään juuri, tulos on tämä:



11.1.7 Kaksijakoisuus

Verkko on kaksijakoinen (*bipartite*), 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 valkoisiksi ja solmut 3, 4 ja 5 harmaiksi.

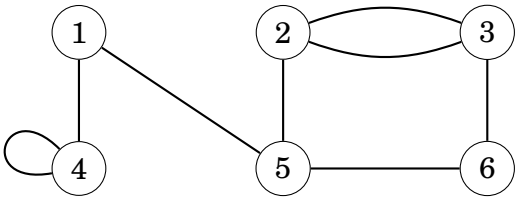


Vaihtoehtoinen määritelmä on, että verkko on kaksijakoinen tarkalleen silloin, kun siinä ei ole parittoman pituista sykliä.

11.1.8 Yksinkertaisuus

Verkko on yksinkertainen (*simple*), jos mistään solmusta ei ole kaarta itseensä eikä minkään kahden solmun välillä ole monta kaarta samaan suuntaan.

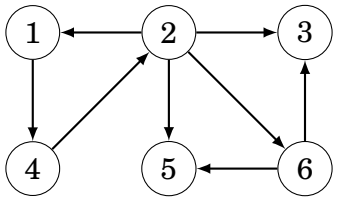
Esimerkiksi seuraava verkko ei ole yksinkertainen, koska solmusta 4 on kaari itseensä ja solmujen 2 ja 3 välillä on kaksi kaarta:



11.2 Verkon esitys

On monia tapoja pitää verkkoa muistissa koodissa. Sopiva tietorakenne riippuu siitä, kuinka suuri verkko on ja millä tavoin algoritmi käsittelee verkkoa. Seuraavaksi käymme läpi kolme tavallista vaihtoehtoa.

Esimerkkinä on seuraava verkko:



11.2.1 Vieruslistat

Vieruslista (*adjacency list*) sisältää kaikki solmut, joihin tietystä solmusta pääsee suoraan kaarella. Useimmat verkkoalgoritmit pystyy toteuttamaan tehokkaasti käyttäen verkon vieruslistaesitystä, minkä vuoksi vieruslistat ovat tavallisesti hyvä valinta verkon tallennusmuodoksi.

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 s lähtevät kaaret. Tämä onnistuu käytännössä seuraavasti for-silmukalla:

```
for (int i = 0; i < v[s].size(); i++) {
    // käsittele kaari solmuun v[s][i]
}
```

Jos verkko on suuntaamaton, hyvä ratkaisu on tallentaa jokainen kaari kahteen kertaan vieruslistoihin molempiin suuntiin.

Jos verkko on painotettu, kaarten painot voi tallentaa esimerkiksi toiseen taulukkoon vektoreita samassa järjestyksessä kuin kaarten päätesolmut.

11.2.2 Vierusmatriisi

Vierusmatriisi (*adjacency matrix*) kertoo jokaisesta mahdollisesta kaaresta, onko se mukana verkossa vai ei. Matriisin avulla on tehokasta käsitellä solmujen välillä olevia kaaria. Toisaalta matriisi vie paljon tilaa, jos verkko on suuri.

Vierusmatriisi on yleensä järkevää tallentaa kaksiulotteisena taulukkona:

```
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.

11.2.3 Kaarilista

Kaarilista (*edge list*) sisältää kaikki verkon kaaret. Kaarilista on hyvä tapa tallentaa verkko, jos algoritmissa täytyy käydä läpi kaikki verkon kaaret eikä ole tarvetta etsiä kaarta alkusolmun perusteella.

Yksi tapa toteuttaa kaarilista on käyttää vektoria, joka sisältää solmupareja:

```
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));
```

Jos verkko on painotettu, listan alkoita voi laajentaa niin, että niissä on solmuparin lisäksi myös kaaren paino.

Toinen tapa toteuttaa kaarilista on tallentaa tiedot kaarten alku- ja loppusolmut taulukkoihin:

```
int a[m+1]; // alkusolmu
int b[m+1]; // loppusolmu
```

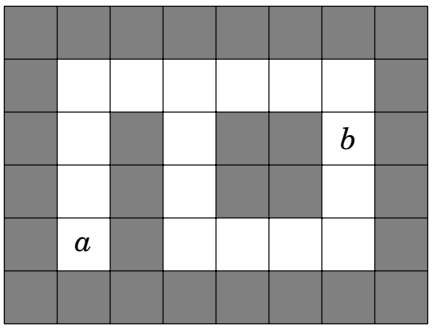
Tämän jälkeen kaaret lisätään näin:

```
a[1] = 1; b[1] = 4;
a[2] = 2; b[2] = 1;
a[3] = 2; b[3] = 3;
a[4] = 2; b[4] = 5;
a[5] = 2; b[5] = 6;
a[6] = 4; b[6] = 2;
a[7] = 6; b[7] = 3;
a[8] = 6; b[8] = 5;
```

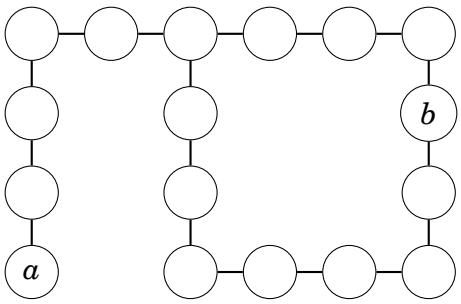
11.3 Epäsuora esitys

Joskus verkkoa on kätevää käsitellä epäsuoran esityksen (*implicit representation*) kautta. Tämä tarkoittaa, että muistissa ei ole suoraan verkon solmuja ja kaaria vaan verkko on kuvattu jollakin toisella tavalla.

Tyypillinen esimerkki epäsuorasta verkosta on labyrintti:



Labyrintti vastaa verkkoa, jossa jokainen lattiaruutu on solmu ja vierekkäisten lattiaruutujen välissä on kaari. Verkko näyttää tältä:



Kätevä tapa tallentaa labyrintti on luoda taulukko merkkijonoista. Jokainen merkkijono vastaa yhtä labyrintin riviä:

```
s[0] = "########";
s[1] = "#......#";
s[2] = "#.#.##b#";
s[3] = "#.#.##.#";
s[4] = "#a#....#";
s[5] = "########";
```

Luku 12

Verkon läpikäynti

Syvyyshaku ja leveyshaku ovat keskeisiä menetelmiä verkon läpikäyntiin. Molemmat algoritmit lähtevät liikkeelle tietystä verkon solmusta ja käyvät läpi kaikki solmut, joihin aloitussolmusta pääsee. Algoritmien erona on, missä järjestyksessä ne etenevät verkon solmuja.

Syvyyshakua ja leveyshakua voi käyttää sekä suuntaamattomassa että suunnatussa verkossa. Tutustumme ensin algoritmeihin tilanteissa, joissa verkko on suuntaamaton. Myöhemmissä luvuissa käytämme algoritmeja myös suunnatun verkon läpikäynneissä.

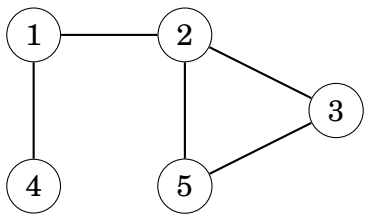
12.1 Syvyyshaku

Syvyyshaku (*depth-first search*) on suoraviivainen menetelmä verkon läpikäyntiin. Syvyyshaku lähtee liikkeelle tietystä verkon solmusta ja etenee siitä kaikkiin solmuihin, jotka ovat saavutettavissa kaaria kulkemalla.

Syvyyshaku etenee verkossa syvyyssuuntaisesti eli valitsee aina yhden suunnan ja kulkee eteenpäin niin kauan kuin vastaan tulee uusia solmuja. Tämän jälkeen haku peräätyy kokeilemaan muita polkuja. Haku pitää kirjaa vierailemistaan solmuista, jotta se käsittelee kunkin solmun vain kerran.

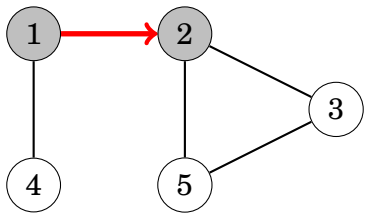
12.1.1 Toiminta

Tarkastellaan syvyyshaun toimintaa seuraavassa verkossa:

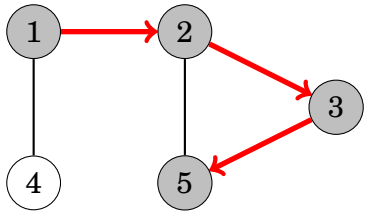


Syvyyshaku voi lähteä liikkeelle mistä tahansa solmusta, mutta oletetaan nyt, että haku lähtee liikkeelle solmusta 1.

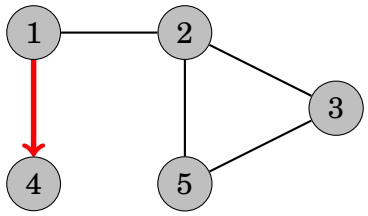
Solmun 1 naapurit ovat solmut 2 ja 4, joista haku etenee ensin solmuun 2:



Tämän jälkeen haku etenee vastaavasti solmuihin 3 ja 5:



Solmun 5 naapurit ovat 2 ja 3, mutta haku on käynyt jo molemmissa. Niinpä haku alkaa peruuttaa taaksepäin. Myös solmujen 3 ja 2 naapurit on käyty, joten haku peruuttaa solmuun 1 asti. Siitä lähtee kaari, josta pääsee solmuun 4:



Tämän jälkeen haku päättyy, koska se on käynyt kaikissa solmuissa.

Syvyyshaun aikavaativuus on $O(n+m)$, missä n on solmujen määrä ja m on kaarten määrä, koska haku käsittelee kerran jokaisen solmun ja kaaren.

12.1.2 Toteutus

Syvyyshaku on yleensä mukavinta toteuttaa rekursiolla. Seuraava toteutus olettaa, että verkon solmut ovat $1,2,\ldots,n$ ja verkko on tallennettu vieruslistoina taulukkoon `v`. Taulukko `z` kertoo, onko solmussa käyty haun aikana.

```
int z[n+1];

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

Haku alkaa kutsumalla funktiota `haku` parametrina aloitussolmu.

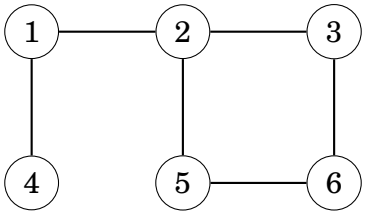
12.2 Leveyshaku

Leveyshaku (*breadth-first search*) käy solmut läpi järjestyksessä sen mukaan, kuinka kaukana ne ovat aloitussolmusta. Niinpä leveyshaun avulla pystyy laskemaan etäisyyden aloitussolmusta kaikkiin muihin solmuihin. Leveyshaku on kuitenkin vaikeampi toteuttaa kuin syvyyshaku.

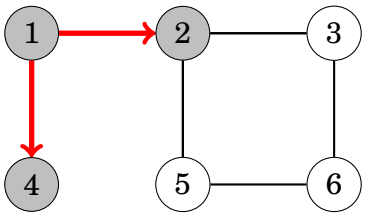
Leveyshakua voi ajatella niin, että se käy solmuja läpi kerros kerrallaan. Ensin haku käy läpi solmut, joihin pääsee yhdellä kaarella alkusolmusta. Tämän jälkeen vuorossa ovat solmut, joihin pääsee kahdella kaarella alkusolmusta jne. Sama jatkuu, kunnes uusia käsiteltäviä solmuja ei enää ole.

12.2.1 Toiminta

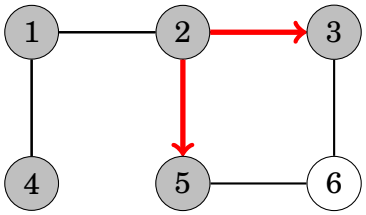
Tarkastellaan leveyshaun toimintaa seuraavassa verkossa:



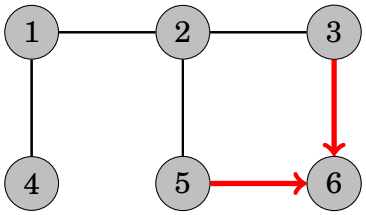
Oletetaan jälleen, että haku alkaa solmusta 1. Haku etenee ensin kaikkiin solmuihin, joihin pääsee alkusolmusta:



Seuraavaksi haku etenee solmuihin 3 ja 5:



Viimeisenä haku etenee solmuun 6:



Leveyshaun tuloksena on etäisyys kuhunkin verkon solmuun alkusolmusta. Etäisyys on sama kuin kerros, jossa solmu käsiteltiin haun aikana:

solmu	etäisyys
1	0
2	1
3	2
4	1
5	2
6	3

Leveyshaun aikavaativuus on syvyyshaun tavoin $O(n+m)$, missä n on solmujen määrä ja m on kaarten määrä.

12.2.2 Toteutus

Leveyshaun toteutus on syvyyshakua monimutkaisempi, koska haku käy läpi solmuja verkon eri puolilta niiden etäisyyden mukaan. Tyypillinen toteutus on pitää yllä jonoa käsiteltävistä solmuista. Joka askeleella otetaan käsittelyyn seuraava solmu jonosta ja uudet solmut lisätään jonon perälle.

Seuraava koodi toteuttaa leveyshaun. Taulukko z kertoo, onko haku käynyt solmussa, ja taulukkoon e lasketaan etäisyydet alkusolmusta. Vektori q toteuttaa jonon, joka sisältää käsiteltävät solmut etäisyysjärjestyksessä.

```
int z[n+1], e[n+1];
vector<int> q;
int s = 1; // alkusolmu
z[s] = 1;
q.push_back(s);
for (int i = 0; i < q.size(); i++) {
    for (int j = 0; j < v[q[i]].size(); j++) {
        int u = v[q[i]][j];
        if (z[u]) continue;
        z[u] = 1;
        e[u] = e[q[i]]+1;
        q.push_back(u);
    }
}
```

12.3 Sovelluksia

Verkon läpikäynnin avulla saa selville monia asioita verkon rakenteesta. Seuraavat sovellukset olettavat, että käsiteltävänä on suuntaamaton verkko. Kaikissa sovelluksissa voi käyttää joko syvyyshakua tai leveyshakua, mutta syvyyshaku on käytännössä parempi valinta, koska sen toteutus on helpompi.

12.3.1 Yhtenäisyys

Verkko on yhtenäinen, jos mistä tahansa solmuista pääsee kaikkiin muihin solmuihin. Niinpä verkon yhtenäisyys selviää aloittamalla läpikäynti jostakin verkon solmusta ja tarkastamalla, pääseekö siitä kaikkiin solmuihin. Jos kaikki solmut tulevat vastaan läpikäynnin aikana, niin verkko on yhtenäinen.

12.3.2 Komponentit

Jos verkko ei ole yhtenäinen, se muodostuu useammasta komponentista. Kaikki tiettyyn komponenttiin kuuluvat solmut löytyvät aloittamalla läpikäynti jostakin komponentin solmusta.

Verkon komponentit saa selville käymällä läpi verkon solmut ja pitämällä kirjaa komponenteista. Jos solmu ei kuulu vielä komponenttiin, se aloittaa uuden komponentin, johon kuuluvat kaikki solmut, joihin solmusta pääsee.

12.3.3 Kaksijakoisuus

Verkko on kaksijakoinen, jos sen solmut voi värittää kahdella värillä niin, että kahta samanväristä solmua ei ole vierekkäin. Kaksijakoisuus on yllättävän helppoa selvittää verkon läpikäynnin avulla.

Ideana on värittää yksi solmuista värillä 1, sen kaikki naapurit värillä 2, näiden solmujen kaikki naapurit värillä 1 jne. Jos jossain vaiheessa ilmenee ristiriita (saman solmun tulisi olla sekä väriä 1 että väriä 2), verkko ei ole kaksijakoinen. Muuten verkko on kaksijakoinen ja yksi väritys on muodostunut.

Tämä algoritmi toimii, koska kun värejä on vain kaksi, ensimmäisen solmun värin valinta määrittää kaikkien muiden samassa komponentissa olevien solmujen värin. Ei ole merkitystä, kumman värin ensimmäinen solmu saa.

Luku 13

Lyhimmät polut

Lyhin polku kahden verkon solmun välillä on kaaria pitkin kulkeva reitti alkusolmusta loppusolmuun, jossa kaarten painojen summa on mahdollisimman pieni. Tässä luvussa tutustumme algoritmeihin, joiden avulla voi selvittää lyhimmän polun verkossa.

Jos verkon kaarissa ei ole painoja, polun pituus on sama kuin kaarten määrä polulla, jolloin lyhimmän polun voi etsiä leveyshaulla. Tässä luvussa keskitymme kuitenkin tapaukseen, jossa kaarilla on painot. Tällöin lyhimpien polkujen etsimiseen tarvitaan kehittyneempiä algoritmeja.

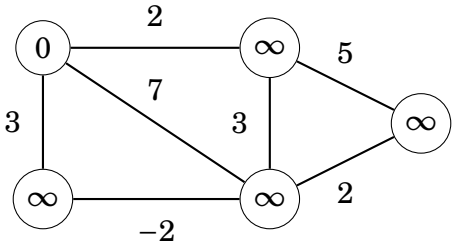
13.1 Bellman-Fordin algoritmi

Bellman-Fordin algoritmi etsii lyhimmän polun alkusolmusta kaikkiin muihin verkon solmuihin. Algoritmi on helppo toteuttaa, ja sitä voi käyttää kaikenlaisissa verkoissa. Algoritmi pystyy myös tunnistamaan, onko verkossa sykliä, jonka pituus on negatiivinen.

Algoritmi pitää yllä etäisyysarvioita alkusolmusta jokaiseen muuhun solmuun. Alussa jokainen etäisyysarvio on ääretön, ja algoritmi parantaa arvioita pikkuhiljaa suorituksensa aikana. Kun algoritmi päättyy, jokaiseen solmuun on saatu laskettua pienin etäisyys alkusolmusta.

13.1.1 Toiminta

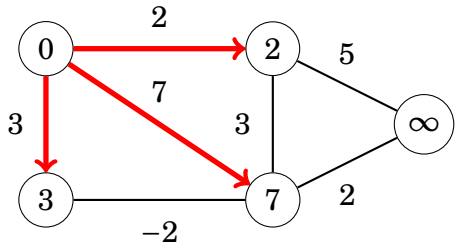
Tarkastellaan Bellman-Fordin algoritmin toimintaa seuraavassa verkossa:



Verkon jokaiseen solmuun on merkitty etäisyysarvio. Alussa alkusolmun etäisyysarvio on 0 ja kaikkien muiden solmujen etäisyysarvio on ääretön (∞).

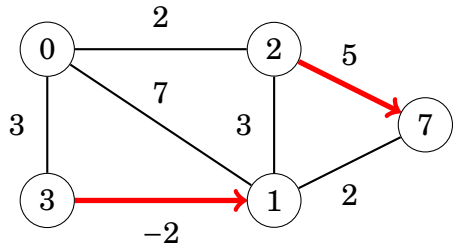
Algoritmin toiminta muodostuu peräkkäisistä kierroksista. Jokaisella kierroksella algoritmi käy kaikki verkon kaaret läpi. Jos kaaren avulla pystyy parantamaan jotain etäisyysarviota, algoritmi tekee näin.

Tässä verkossa ensimmäinen kierros parantaa arvioita näin:

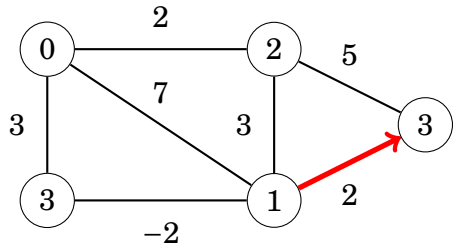


Esimerkiksi alkusolmusta lähtevä 2:n pituinen kaari parantaa sen kohdesolmun etäisyysarviota, koska vanha arvio oli ääretön, mutta kulkemalla tätä kaarta alkusolmusta etäisyys on vain 2.

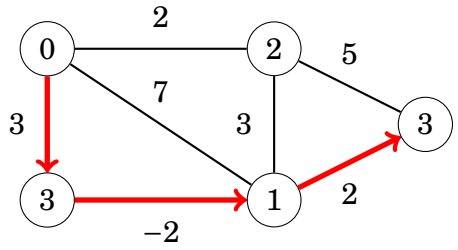
Seuraavalla kierroksella arviot paranevat näin:



Sitten tulee vielä yksi parannus:



Tämän jälkeen mikään kaari ei paranna etäisyysarvioita. Tämä tarkoittaa, että etäisyydet ovat lopulliset, eli joka solmussa on nyt lyhin etäisyys alkusolmusta kyseiseen solmuun. Esimerkiksi lyhin etäisyys 3 oikeanpuolimmaiseen solmuun toteutuu käyttämällä seuraavaa reittiä:



13.1.2 Toteutus

Bellman-Fordin algoritmi on mukavinta toteuttaa niin, että verkko on tallennettu kaarilistana. Seuraavassa koodissa taulukot `a` ja `b` kertovat kaaren alku- ja loppusolmun ja taulukko `w` kertoo kaaren painon.

Koodi pitää yllä etäisyysarvioita solmuihin taulukossa `e`. Koodi päättyy, kun mikään arvio ei muutu kierroksen aikana.

```
int e[n+1];
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[s] = 0; // alkusolmu s
while (true) {
    bool x = false;
    for (int i = 1; i <= m; i++) {
        if (e[a[i]]+w[i] < e[b[i]]) {
            e[b[i]] = e[a[i]]+w[i];
            x = true;
        }
    }
    if (!x) break;
}
```

Osoittautuu, että Bellman-Fordin algoritmi pysähtyy aina viimeistään n kierroksen jälkeen. Tämä johtuu siitä, että jokainen lyhin polku on enintään n kaaren mittainen ja algoritmi muodostaa polkuja ainakin kaaren verran edemmäs joka kierroksella. Niinpä algoritmin aikavaativuus on $O(nm)$.

13.1.3 Negatiivinen sykli

Jos verkossa on negatiivinen sykli, lyhimpien polkujen etsiminen ei ole mielekästä, koska negatiivisen syklin sisältävää polkua voi lyhentää loputtomasti kiertämällä sykliä uudestaan ja uudestaan.

Verkossa olevan negatiivisen syklin voi tunnistaa muunnetulla Bellman-Fordin algoritmilla. Ideana on suorittaa algoritmia n kierrosta ja tarkastaa sen jälkeen, pystyykö vielä jotain polkua lyhentämään. Jos polun lyhentäminen on mahdollista, verkossa on negatiivinen sykli.

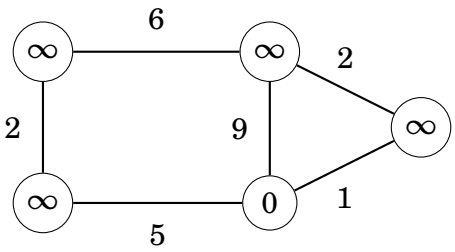
13.2 Dijkstran algoritmi

Dijkstran algoritmi etsii Bellman-Fordin algoritmin tavoin lyhimmät polut alkusolmusta kaikkiin muihin solmuihin. Dijkstran algoritmi on tehokkaampi kuin Bellman-Fordin algoritmi, minkä ansiosta se soveltuu suurien verkkojen käsittelyyn. Algoritmi vaatii kuitenkin, ettei verkossa ole negatiivisia kaaria.

Dijkstran algoritmi pitää Bellman-Fordin algoritmin tavoin yllä etäisyysarvioita solmuihin ja parantaa niitä algoritmin aikana. Algoritmin tehokkuus perustuu siihen, että sen riittää käydä läpi verkon kaaret vain kerran hyödyntämällä tietoa, ettei verkossa ole negatiivisia kaaria.

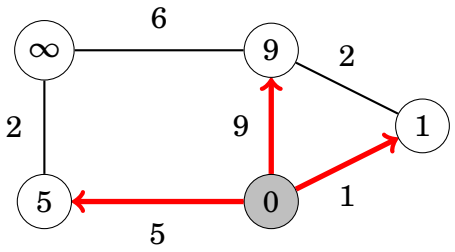
13.2.1 Toiminta

Tarkastellaan Dijkstran algoritmin toimintaa seuraavassa verkossa:



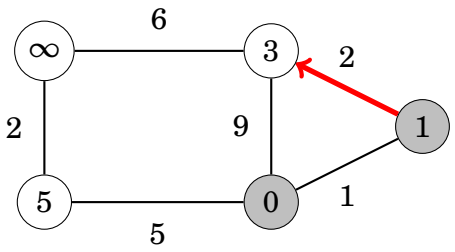
Dijkstran algoritmi ottaa joka askeleella käsittelyyn sellaisen solmun, jota ei ole vielä käsitelty ja jonka etäisyysarvio on mahdollisimman pieni. Alussa tällainen solmu on alkusolmu, jonka etäisyysarvio on 0.

Kun solmu tulee käsittelyyn, algoritmi käy läpi kaikki siitä lähtevät kaaret ja parantaa etäisyysarvioita niiden avulla:

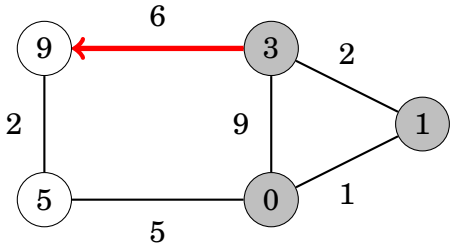


Alkusolmun käsittely paransi etäisyyksiä kolmen muuhun solmuun, joiden uudet etäisyydet ovat nyt 1, 5 ja 9.

Seuraavaksi käsittelyyn tulee solmu, jonka etäisyys on 1:

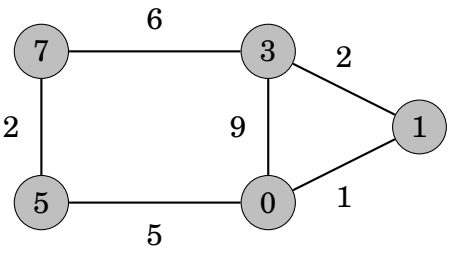


Tämän jälkeen vuorossa on solmu 3:



Dijkstran algoritmissa on hienoutena, että aina kun solmu tulee käsittelyyn, sen etäisyysarvio on siitä lähtien lopullinen. Esimerkiksi tässä vaiheessa etäisyydet 0, 1 ja 3 ovat lopulliset etäisyydet käsiteltyihin solmuihin.

Algoritmi käsittelee vastaavasti vielä kaksi viimeistä solmua, minkä jälkeen algoritmin päätteeksi etäisyydet ovat:



13.2.2 Toteutus

Dijkstran algoritmin tehokas toteutus vaatii, että verkosta pystyy paikantamaan tehokkaasti seuraavaksi käsiteltävän solmun. Sopiva tietorakenne tähän on keko, joka sisältää etäisyysarvioita.

Seuraavassa koodissa keko `q` sisältää pareja, joiden ensimmäinen jäsen on etäisyysarvio ja toinen jäsen on solmun tunnus. Taulukko `e` sisältää kunkin solmun etäisyysarvion, ja taulukko `z` kertoo, onko solmu jo käsitelty.

Koodi olettaa, että verkko on tallennettu vieruslistoina taulukkoihin `v` ja `w`.

```
priority_queue<pair<int,int>> q;
int e[n+1], z[n+1];
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[s] = 0;
q.push({0,s});
while (!q.empty()) {
    int x = q.top().second;
    q.pop();
    if (z[x]) continue;
    z[x] = 1;
    for (int i = 0; i < v[x].size(); i++) {
        int u = v[x][i];
        if (e[x]+w[x][i] < e[u]) {
            e[u] = e[x]+w[x][i];
            q.push({-e[u],u});
        }
    }
}
```

C++:n kekorakenne `priority_queue` on oletuksena maksimikeko, eli siitä pystyy kysymään suurimpia alkioita. Dijkstran algoritmissa kuitenkin täytyy saada selville joka vaiheessa pienin etäisyysarvio. Tämän vuoksi koodi tallentaa etäisyysarviot kekoon negatiivisina.

Dijkstran algoritmin yllä olevan toteutuksen aikavaativuus on $O(n+m \log m)$. Aikavaativuuden osa $m \log m$ johtuu siitä, että keossa on yhteensä enintään m

etäisyysarviota, yksi jokaista verkon kaarta kohti. Algoritmi laittaa jokaisen arvion kerran kekoon ja poistaa kerran keosta.

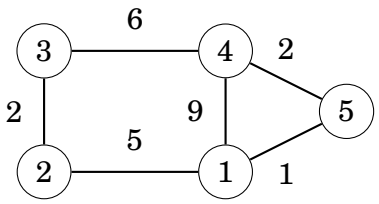
13.3 Floyd-Warshallin algoritmi

Floyd-Warshallin algoritmi on hyvin erilainen lähestymistapa lyhimpien polkujen etsimiseen kuin Bellman-Fordin ja Dijkstran algoritmit. Siinä missä muut algoritmit etsivät lyhimpiä polkuja tietystä solmusta alkaen, Floyd-Warshallin algoritmi etsii lyhimmän polun jokaisen verkon solmuparin välillä.

Algoritmi ylläpitää kaksiulotteista taulukkoa etäisyyksistä solmujen välillä. Ensin taulukkoon on merkitty etäisyydet käyttäen vain solmujen välisiä kaaria. Tämän jälkeen algoritmi päivittää etäisyyksiä, kun yksi verkon solmuista saa kulloinkin toimia välisolmuna poluilla.

13.3.1 Toiminta

Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



Algoritmi merkitsee aluksi taulukkoon etäisyyden 0 jokaisesta solmusta itseensä sekä etäisyyden x, jos solmuparin välillä on kaari, jonka pituus on x. Muiden solmuparien etäisyys on aluksi ääretön.

Tässä verkossa taulukosta tulee:

	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

Algoritmin toiminta muodostuu peräkkäisistä kierroksista. Jokaisella kierroksella uusi solmu saa toimia välisolmuna poluilla, ja algoritmi parantaa taulukon etäisyyksiä käyttäen tätä solmua.

Ensimmäisellä kierroksella solmu 1 saa toimia välisolmuna. Tämä lyhentää etäisyyksiä solmuparien 2 ja 4 sekä 2 ja 5 välillä:

	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

Toisella myös solmu 2 saa toimia välisolmuna. Tämä mahdollistaa uudet polut solmuparien 1 ja 3 sekä 3 ja 5 välille:

	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

Algoritmin toiminta jatkuu samalla tavalla niin, että kukin solmu tulee vuorollaan välisolmuksi. Algoritmin päätteeksi taulukko sisältää lyhimmän etäisyyden minkä tahansa solmuparin 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

13.3.2 Toteutus

Floyd-Warshallin algoritmi on luontevaa toteuttaa käyttäen verkon vierusmatriisiesitystä. Seuraava koodi laskee etäisyydet taulukkoon `d`:

```
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] = 1e9;
    }
}
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]);
        }
    }
}
```

Koodin alkuosa merkitsee taulukkoon alkuetäisyydet, minkä jälkeen koodin loppuosa käy läpi välisolmut muuttujalla k ja päivittää etäisyyksiä.

Floyd-Warshallin algoritmin aikavaativuus on $O(n^3)$, koska sen työläin vaihe muodostuu kolmesta sisäkkäisestä silmukasta, jotka käyvät läpi verkon solmut. Algoritmin toteutus on yksinkertainen, minkä ansiosta se on hyvä valinta myös yksittäisen lyhimmän polun etsimiseen pienessä verkossa.

13.4 Yhteenveto

Olemme nyt käsitelleet neljä algoritmia lyhimpien polkujen etsimiseen:

algoritmi	toiminta	aikavaativuus	edellytys
leveyshaku	polut alkusolmusta	$O(n+m)$	ei kaarten painoja
Bellman-Ford	polut alkusolmusta	$O(nm)$	–
Dijkstra	polut alkusolmusta	$O(n+m\log m)$	ei negatiivisia kaaria
Floyd-Warshall	kaikki polut	$O(n^3)$	–

Jokaisessa algoritmissa on omat hyvät ja huonot puolensa, jotka täytyy ottaa huomioon algoritmin valinnassa.

Luku 14

Puiden käsittely

Puu on yhtenäinen, syklitön ja suuntaamaton verkko, mikä tarkoittaa, että jokaisen kahden solmun välillä on yksikäsitteinen polku. Puun yksinkertainen rakenne mahdollistaa sen käsittelyn yleistä verkkoa tehokkaammin. Tässä luvussa aloitamme tutustumisen puiden käsittelyyn liittyviin tekniikoihin.

Yksi keskeinen tekniikka puiden käsittelyssä on dynaaminen ohjelmointi. Puut soveltuvat hyvin dynaamiseen ohjelmointiin, koska ne jakautuvat luontevasti alipuiksi, joita voi käsitellä toisistaan riippumattomasti.

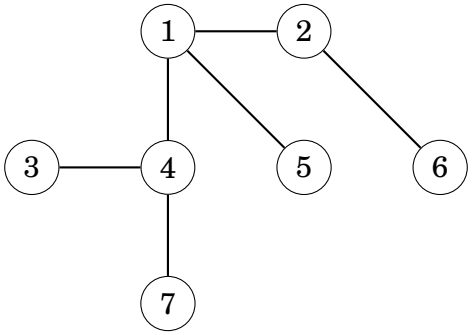
14.1 Perusteet

Käymme aluksi läpi puihin liittyvää sanastoa sekä puiden käsittelyn perustekniikoita, joita tarvitsee usein eri algoritmien osana.

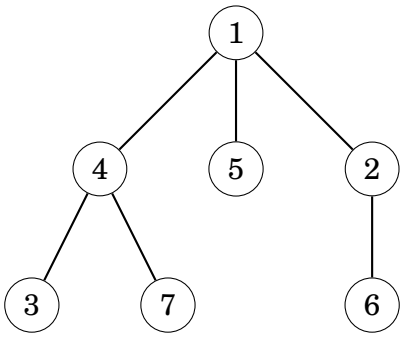
14.1.1 Juuren valinta

Tavallinen tapa käsitellä puuta on valita yksi puun solmuista juureksi (*root*), jonka alapuolelle kaikki muut solmut asettuvat. Usein ei ole merkitystä, mikä puun solmuista valitaan juureksi.

Tarkastellaan esimerkkinä seuraavaa puuta:



Kun solmu 1 valitaan juureksi, puu asettuu seuraavaan muotoon:



Solmun vanhempi (*parent*) on ylemmän tason solmu, johon pääsee kaarella. Solmun lapsia (*children*) ovat alemman tason solmut, joihin pääsee kaarella. Esimerkiksi solmun 4 vanhempi on solmu 1 ja lapset ovat solmut 3 ja 7.

Jokainen puun solmuista muodostaa alipuun (*subtree*), jonka juurena on solmu itse. Esimerkiksi solmun 4 alipuussa ovat solmut 4, 3 ja 7. Puun lehtiä (*leaf*) ovat solmut, joilla ei ole lapsia, eli tässä puussa solmut 3, 7, 5 ja 6.

14.1.2 Läpikäynti

Seuraava rekursiivinen koodi käy läpi vieruslistoina tallennetun puun:

```
void haku(int s, int e) {
    // solmun s käsittely
    for (int i = 0; i < v[s].size(); i++) {
        if (v[s][i] == e) continue;
        haku(v[s][i], s);
    }
}
```

Funktion parametrit ovat käsiteltävä solmu s sekä tätä solmua ylempi solmu e. Parametrin e ideana on pakottaa, että puuta käydään läpi vain ylhäältä alaspäin. Seuraava kutsu käy läpi puun niin, että juurena on solmu 1:

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

Juurisolmulla ei ole ylempää solmua, joten parametri e on 0.

14.1.3 Dynaaminen ohjelmointi

Puun läpikäyntiin voi yhdistää myös dynaamista ohjelmointia ja laskea sen avulla jotain tietoa puusta. Dynaamisen ohjelmoinnin avulla voi esimerkiksi laskea jokaiseen solmuun, montako solmua sen alipuussa on tai kuinka pitkä on pisin solmusta alaspäin jatkuva polku puussa.

Lasketaan esimerkiksi solmujen määrät alipuissa. Ideana on liittää puun läpikäyntiin koodi, joka laskee jokaisen alipuun solmujen määrän sen lasten alipuiden solmujen määrästä.

Tarvittava koodi on tässä:

```
int d[n+1];

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

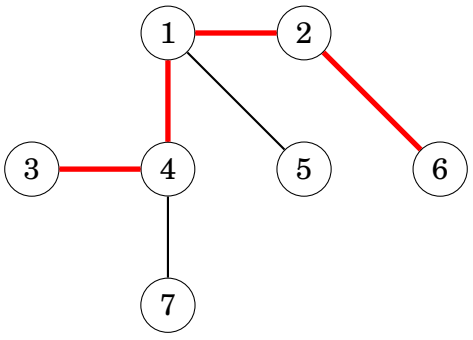
Funktion kutsun jälkeen d[s] kertoo solmun s alipuun solmujen määrän.

14.2 Etäisyydet

Seuraava tavoitteemme on laskea tehokkaasti etäisyyksiä puun solmujen välillä. Osoittautuu, että puun erityisen rakenteen ansiosta muutama puun läpikäynti riittää kaikkien solmujen etäisyyksien laskemiseen.

14.2.1 Läpimitta

Puun läpimitta (*diameter*) on pisin polku kahden puussa olevan solmun välillä. Esimerkiksi seuraavassa puussa läpimitta on 4:



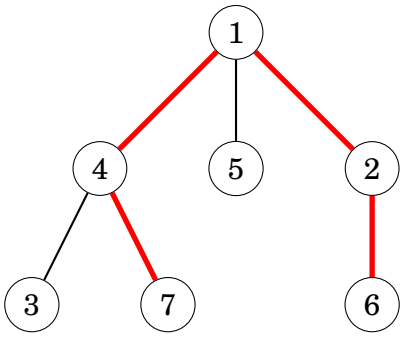
Läpimitta on 4, koska polku solmusta 3 solmuun 6 on pituudeltaan 4. Läpimittaa vastaava polku ei ole välttämättä yksikäsitteinen. Esimerkiksi tässä puussa myös polku solmusta 7 solmuun 6 on pituudeltaan 4.

Käymme seuraavaksi läpi kaksi tehokasta algoritmia puun läpimitan laskeminen. Ensimmäinen algoritmi perustuu dynaamiseen ohjelmointiin, ja toinen algoritmi etsii kaukaisimmat solmut syvyyshakujen avulla.

Algoritmi 1

Algoritmin alussa yksi solmuista valitaan puun juureksi. Tämän jälkeen algoritmi laskee jokaiseen solmuun, kuinka pitkä on pisin polku, joka alkaa lehdestä, nousee kyseiseen solmuun asti ja laskeutuu toiseen lehteen. Yksi näistä poluista on pisin kahden solmun välinen polku puussa.

Esimerkissä pisin polku alkaa lehdestä 7, nousee lehteen 1 asti ja laskeutuu sitten alas lehteen 6:



Algoritmi laskee dynaamisella ohjelmoinnilla jokaiseen solmuun kaksi pisintä polkua, jotka lähtevät solmusta alaspäin. Nämä polut yhdistämällä syntyy pisin polku, jonka käännekohtana kyseinen solmu on.

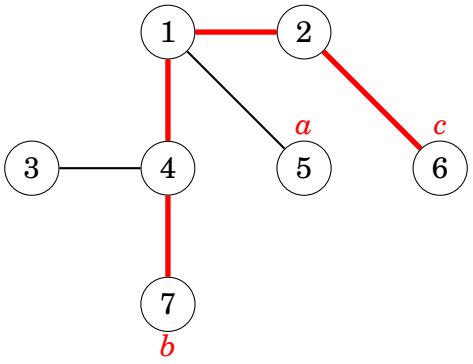
Esimerkiksi solmusta 1 lähtee kolme polkua alaspäin. Niistä kahden pituus on 2 ($1 \to 4 \to 7$ sekä $1 \to 2 \to 6$) ja yhden pituus on 1 ($1 \to 5$). Pisin polku syntyy yhdistämällä molemmat 2-pituiset polut.

Algoritmi 2

Toinen tehokas tapa laskea puun läpimitta on seuraava:

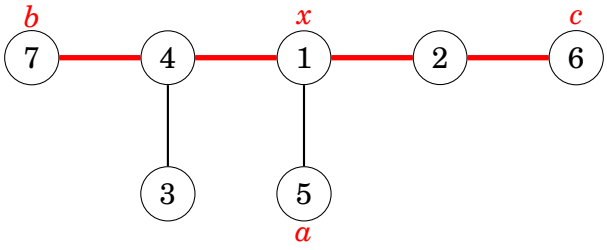
1. Valitse mikä tahansa solmu a.
2. Etsi a:sta kaukaisin solmu b (syvyyshaku).
3. Etsi b:stä kaukaisin solmu c (syvyyshaku).
4. Läpimitta on etäisyys b:n ja c:n välillä.

Esimerkissä a, b ja c voisivat olla:



Menetelmä on tyylikäs, mutta miksi se toimii?

Tässä auttaa tarkastella puuta niin, että puun läpimittaa vastaava polku on levitetty vaakatasoon ja muut puun osat riippuvat siitä alaspäin:

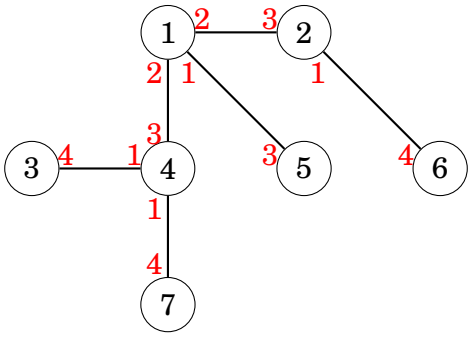


Solmu x on kohta, jossa polku solmusta a liittyy läpimittaa vastaavaan polkuun. Kaukaisin solmu a:sta on solmu b, solmu c tai jokin muu solmu, joka on ainakin yhtä kaukana solmusta x. Niinpä tämä solmu on aina sopiva valinta läpimittaa vastaavan polun toiseksi päätesolmuksi.

14.2.2 Kaikki etäisyydet

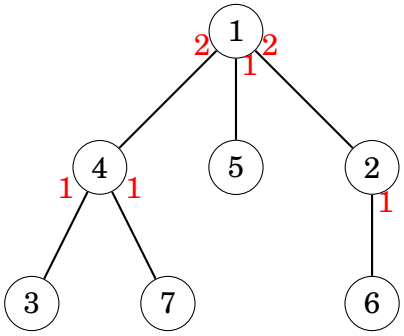
Tarkastellaan sitten vaikeampaa tehtävää, jossa tehtävänä on laskea jokaiselle puun solmulle, kuinka kaukana on kaukaisin solmu kussakin suunnassa.

Esimerkkipuussa etäisyydet ovat:



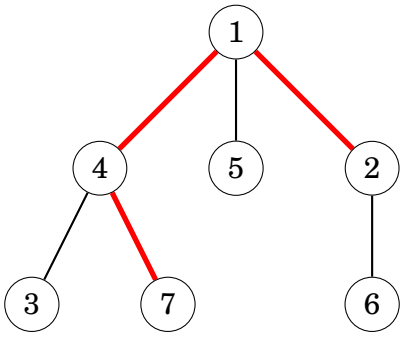
Esimerkiksi solmussa 4 kaukaisin solmu ylöspäin mentäessä on solmu 6, johon etäisyys on 3 polkua $4 \to 1 \to 2 \to 6$.

Tässäkin tehtävässä hyvä lähtökohta on valita jokin solmu puun juureksi, jolloin kaikki etäisyydet alaspäin saa laskettua suoraan dynaamisella ohjelmoinnilla:

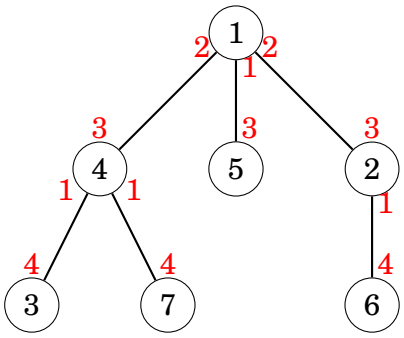


Jäljelle jäävä tehtävä on laskea etäisyydet ylöspäin. Tämä onnistuu tekemällä puuhun toinen läpikäynti, joka pitää mukana tietoa, mikä on suurin etäisyys solmun vanhemmasta johonkin toisessa suunnassa olevaan solmuun.

Esimerkiksi solmun 2 suurin etäisyys ylöspäin on yhtä suurempi kuin solmun 1 suurin etäisyys johonkin muuhun suuntaan kuin solmuun 2:



Lopputuloksena on etäisyydet kaikista solmuista kaikkiin suuntiin:



Luku 15

Virittävät puut

Virittävä puu on kokoelma verkon kaaria, joka kytkee kaikki verkon solmut toisiinsa. Kuten puut yleensä, virittävä puu on yhtenäinen ja syklitön. Virittävän puun paino on sen kaarien painojen summa, ja usein on kiinnostavaa etsiä painoltaan mahdollisimman pieni virittävä puu.

Tämä luku esittelee Kruskalin algoritmin, jota käyttämällä voi etsiä pienimmän virittävän puun verkosta. Algoritmin idea on yksinkertainen, mutta sen tehokas toteutus vaatii uuden tietorakenteen, jonka avulla pystyy pitämään yllä ja yhdistämään alkioiden muodostamia joukkoja.

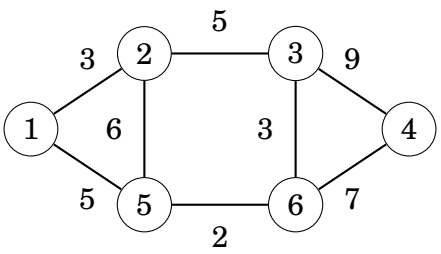
15.1 Kruskalin algoritmi

Kruskalin algoritmi muodostaa verkon pienimmän virittävän puun (*minimum spanning tree*). Algoritmi aloittaa virittävän puun rakentamisen tilanteesta, jossa puussa ei ole yhtään kaaria. Sitten algoritmi alkaa lisätä puuhun kaaria järjestyksessä kevyimmästä raskaimpaan.

Kruskalin algoritmi pitää yllä tietoa verkon komponenteista. Aluksi jokainen solmu on omassa komponentissaan, ja algoritmin aikana puuhun lisättävät kaaret yhdistävät komponentteja. Lopulta kaikki solmut ovat samassa komponentissa, jolloin pienin virittävä puu on valmis.

15.1.1 Toiminta

Tarkastellaan Kruskalin algoritmin toimintaa seuraavassa verkossa:

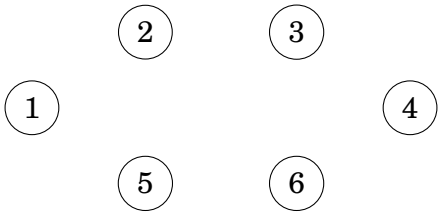


Algoritmin ensimmäinen vaihe on järjestää verkon kaaret niiden painon mukaan. Tuloksena on seuraava lista:

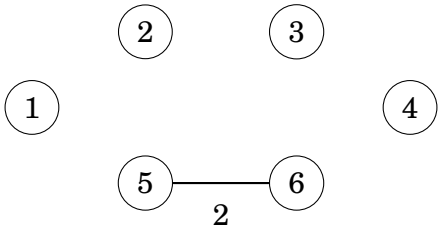
kaari	paino
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

Tämän jälkeen algoritmi käy listan läpi ja lisää kaaren puuhun, jos se yhdistää kaksi erillistä komponenttia.

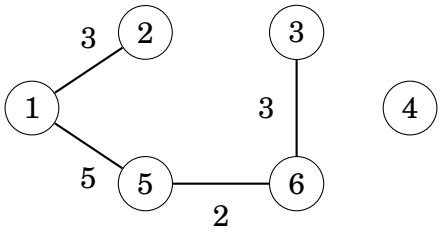
Aluksi jokainen solmu on omassa komponentissaan:



Ensimmäinen virittävään puuhun lisättävä kaari on 5–6, joka yhdistää komponentit {5} ja {6} komponentiksi {5,6}:



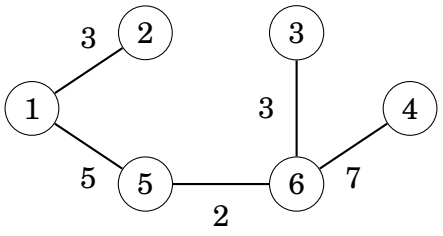
Tämän jälkeen algoritmi lisää puuhun vastaavasti kaaret 1–2, 3–6 ja 1–5:



Näiden lisäysten jälkeen monet komponentit ovat yhdistyneet ja verkossa on kaksi komponenttia: {1,2,3,5,6} ja {4}.

Seuraavaksi käsiteltävä kaari on 2–3, mutta tämä kaari ei tule mukaan puuhun, koska solmut 2 ja 3 ovat jo samassa komponentissa. Vastaavasta syystä myöskään kaari 2–5 ei tule mukaan puuhun.

Lopuksi puuhun tulee kaari 4–6, joka luo yhden komponentin:



Tuloksena on verkon pienin virittävä puu, jonka paino on $2+3+3+5+7 = 21$.

Yllättävä ominaisuus Kruskalin algoritmissa on, että sen tuloksena on aina pienin mahdollinen virittävä puu. Tämä johtuu siitä, että kahden komponentin yhdistämisessä mahdollisimman kevyt kaari on paras valinta, koska muuten komponentit täytyisi yhdistää myöhemmin käyttäen raskaampaa kaarta.

15.1.2 Toteutus

Seuraava koodi on runko Kruskalin algoritmin toteutukselle. Koodi olettaa, että verkon kaaret on tallennettu kaarilistana niin, että taulukot `a` ja `b` kertovat kaaren alku- ja loppusolmun ja taulukko `w` kertoo kaaren painon. Kaarilistan tulee olla järjestyksessä kaarten painojen mukaisesti.

Koodi käyttää kahta funktiota: funktio `sama` tutkii, ovatko solmut samassa komponentissa, ja funktio `liita` yhdistää kaksi komponenttia toisiinsa.

```
int c = 0; // virittävän puun koko
for (int i = 1; i <= m; i++) {
    if (sama(a[i],b[i])) continue;
    liita(a[i],b[i]);
    c += w[i];
}
```

Ongelmana on, kuinka toteuttaa tehokkaasti funktiot `sama` ja `liita`. Seuraavaksi esiteltävä tietorakenne ratkaisee asian. Se toteuttaa kummankin funktion ajassa $O(\log n)$, jolloin Kruskalin algoritmin aikavaativuus on $O(m \log n)$ kaarilistan järjestämisen jälkeen.

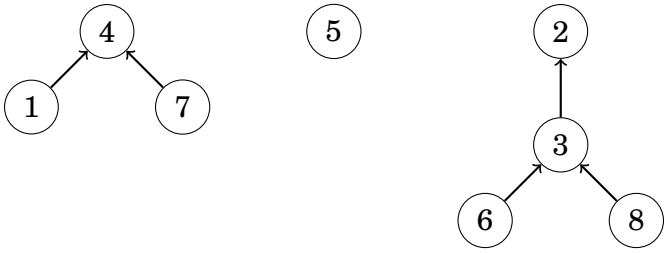
15.2 Union-find-rakenne

Union-find-rakenne pitää yllä alkiojoukkoja. Joukot ovat erillisiä, eli tietty alkio on tarkalleen yhdessä joukossa. Rakenne tarjoaa kaksi operaatiota, jotka toimivat ajassa $O(\log n)$. Ensimmäinen operaatio tarkistaa, ovatko kaksi alkiota samassa joukossa. Toinen operaatio yhdistää kaksi joukkoa toisiinsa.

15.2.1 Rakenne

Union-find-rakenteessa jokaisella joukolla on edustaja-alkio. Kaikki muut joukon alkiot osoittavat edustajaan joko suoraan tai muiden alkioiden kautta.

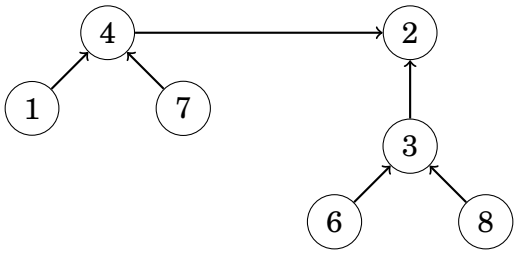
Esimerkiksi jos joukot ovat $\{1,4,7\}$, $\{5\}$ ja $\{2,3,6,8\}$, tilanne voisi olla:



Tässä tapauksessa alkiot 4, 5 ja 2 ovat joukkojen edustajat.

Minkä tahansa alkion edustaja löytyy kulkemalla polku loppuun alkiosta. Esimerkiksi alkion 6 edustaja on 2, koska polku on $6 \to 3 \to 2$. Tämän avulla voi selvittää, ovatko kaksi alkiota samassa joukossa: jos kummankin alkion edustaja on sama, alkiot ovat samassa joukossa, ja muuten eri joukoissa.

Joukkojen yhdistäminen tapahtuu valitsemalla toisen edustaja joukkojen yhteiseksi edustajaksi ja kytkemällä toinen edustaja siihen. Esimerkiksi joukot $\{1,4,7\}$ ja $\{2,3,6,8\}$ voi yhdistää näin joukoksi $\{1,2,3,4,6,7,8\}$:



Tästä lähtien alkio 2 edustaa kaikkia joukon alkioita.

Tehokkuuden kannalta oleellista on, miten yhdistäminen tapahtuu. Osoittautuu, että ratkaisu on yksinkertainen: riittää yhdistää aina pienempi joukko suurempaan, tai kummin päin tahansa, jos joukot ovat yhtä suuret.

Tätä menetelmää käyttäen pisin mahdollinen ketju alkiosta edustajaan on aina luokkaa $O(\log n)$, eli operaatiot ovat tehokkaita.

15.2.2 Toteutus

Union-find-rakenne on kätevää toteuttaa taulukoiden avulla. Seuraavassa toteutuksessa taulukko `k` viittaa seuraavaan alkioon ketjussa tai alkioon itseensä, jos alkio on edustaja. Taulukko `s` taas kertoo jokaiselle edustajalle, kuinka monta alkiota niiden joukossa on.

Aluksi jokainen alkio on omassa joukossaan, jonka koko on 1:

```
for (int i = 1; i <= n; i++) k[i] = i;
for (int i = 1; i <= n; i++) s[i] = 1;
```

Funktio sama kertoo, ovatko alkiot samassa joukossa:

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

Funktio liita taas yhdistää kaksi joukkoa toisiinsa:

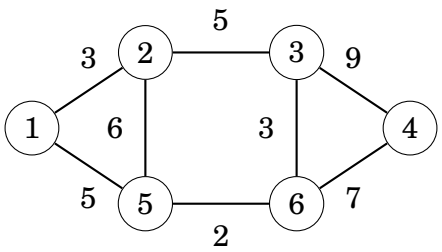
```
void liita(int a, int b) {
    while (a != k[a]) a = k[a];
    while (b != k[b]) b = k[b];
    if (s[a] > s[b]) {
        s[a] += s[b];
        k[b] = a;
    } else {
        s[b] += s[a];
        k[a] = b;
    }
}
```

15.3 Primin algoritmi

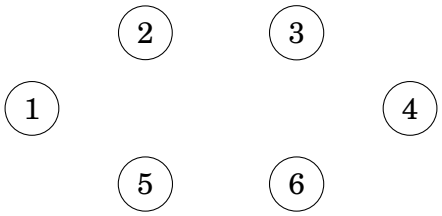
Primin algoritmi on vaihtoehtoinen menetelmä verkon pienimmän virittävän puun muodostamiseen. Algoritmi aloittaa puun muodostamisen valitusta verkon solmusta ja lisää puuhun aina kaaren, joka on mahdollisimman kevyt ja joka liittää puuhun uuden solmun.

15.3.1 Toiminta

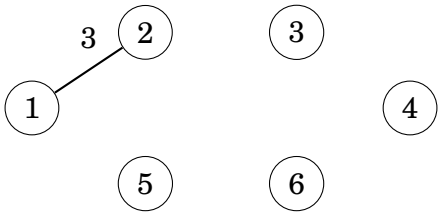
Tarkastellaan Primin algoritmin toimintaa seuraavassa verkossa:



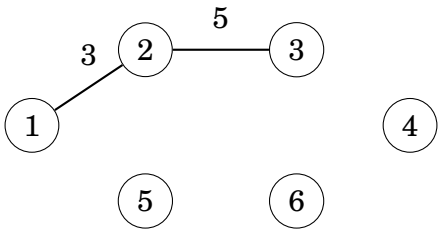
Aluksi solmujen välillä ei ole mitään kaaria:



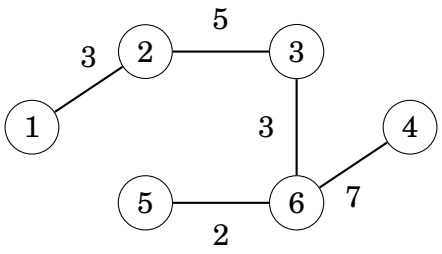
Puun muodostuksen voi aloittaa mistä tahansa solmusta, ja aloitetaan se nyt solmusta 1. Kevein kaari on painoltaan 3 ja se johtaa solmuun 2:



Nyt kevein uuteen solmuun johtavan kaaren paino on 5, ja voimme laajentaa joko solmuun 3 tai 5. Valitaan solmu 3:



Sama jatkuu, kunnes kaikki solmut ovat mukana puussa:



15.3.2 Toteutus

Primin algoritmi muistuttaa Dijkstran algoritmia, ja sen voi toteuttaa samaan tapaan keon avulla. Erona on, että Dijkstran algoritmissa valitaan kaari, jonka kautta syntyy lyhin polku alkusolmusta uuteen solmuun, mutta Primin algoritmissa valitaan vain kevein kaari, joka johtaa uuteen solmuun.

Algoritmin aikavaativuus on $O(n + m \log m)$ eli sama kuin Dijkstran algoritmissa. Käytännössä Primin algoritmi on suunnilleen yhtä nopea kuin Kruskalin algoritmi, eli on makuasia, kumpaa algoritmia käyttää.

Luku 16

Syklittömät verkot

Kun suunnatussa verkossa ei ole syklejä, sen käsittely on yleistä verkkoa helpompaa, koska solmut voi laittaa selkeään järjestykseen sen perusteella, miten ne viittaavat toisiinsa. Tämän ansiosta on mahdollista soveltaa esimerkiksi dynaamista ohjelmointia verkon polkujen käsittelyssä.

Tässä luvussa tutustumme algoritmeihin, jotka käsittelevät suunnattuja, syklittömiä verkkoja. Tämä verkkotyyppi tulee vastaan niin usein, että englannissa on sille oma lyhenne *dag*, joka tulee sanoista *directed acyclic graph*.

16.1 Syklin etsiminen

Miten voi tietää, onko suunnatussa verkossa sykliä? Yksi tehokas menetelmä on etsiä sykliä käyttäen muunnettua syvyyshakua.

Ideana on toteuttaa syvyyshaku niin, että solmulla on kolme mahdollista tilaa: 0 (ei käsitelty), 1 (käsittely kesken) tai 2 (käsittely valmis). Aluksi jokaisen solmun tilana on 0. Tila 1 aktivoituu, kun haku tulee solmuun ensimmäistä kertaa, ja tilaksi tulee 2, kun kaikki solmusta lähtevät kaaret on käsitelty.

Verkossa on suunnattu sykli tarkalleen silloin, jos jossain vaiheessa syvyyshakua vastaan tulee tilassa 1 oleva solmu. Sykli muodostuu niistä solmuista, joihin haku on edennyt tilassa 1 olevan solmun jälkeen.

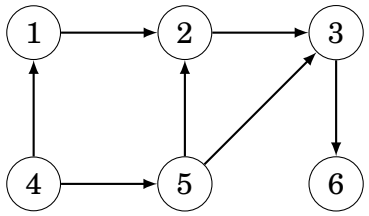
Tässä on algoritmin toteutus:

```
void haku(int s) {
    if (t[s] == 1) {
        // 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;
}
```

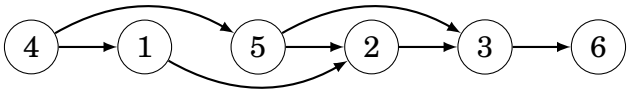
16.2 Topologinen järjestäminen

Suunnatun verkon topologinen järjestys (*topological sort*) on sellainen lista solmuista, että jos solmusta a on kaari solmuun b, niin a on ennen b:ta listassa. Yksi tulkinta on, että kaaret määrittelevät solmuille suuruusjärjestyksen ja topologisessa järjestyksessä jokainen solmu on edellistä suurempi

Tarkastellaan esimerkiksi seuraavaa verkkoa:



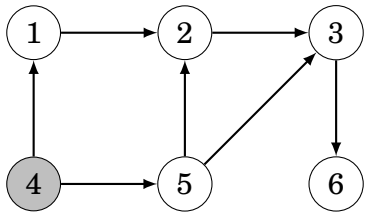
Yksi tämän verkon topologinen järjestys on 4, 1, 5, 2, 3, 6:



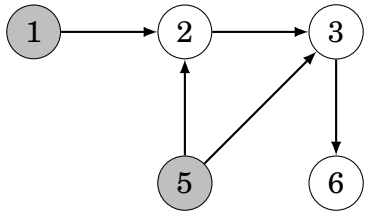
Topologinen järjestys on olemassa silloin, kun verkossa ei ole suunnattua sykliä. Jos verkossa on suunnattu sykli, topologista järjestystä ei voi muodostaa, koska mitään syklin solmuista ei voi valita järjestykseen ensimmäisenä.

Yksinkertainen tapa muodostaa topologinen järjestys on valita joka askeleella seuraavaksi solmuksi jokin solmu, johon ei tule kaaria muista solmuista. Tämän jälkeen valittu solmu ja siitä lähtevät kaaret poistetaan verkosta.

Esimerkkiverkossa ensimmäinen valittava solmu on 4:



Solmun poistamisen jälkeen solmuihin 1 ja 5 ei tule kaaria, joten jompikumpi niistä valitaan seuraavaksi topologiseen järjestykseen:



Sama jatkuu, kunnes kaikki solmut on valittu ja topologinen järjestys on valmis. Verkon syklittömyys takaa, että jokin solmu on aina mahdollista valita, koska jos kaikkiin solmuihin tulisi kaari toisesta solmusta, niin verkossa olisi sykli. Jos mahdollisia solmuja on useita, niistä voi valita minkä tahansa.

16.3 Dynaaminen ohjelmointi

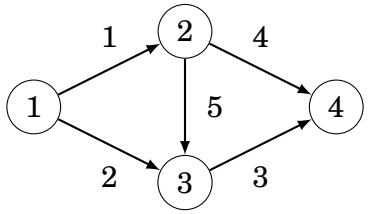
Solmujen käsittely topologisessa järjestyksessä mahdollistaa dynaamisen ohjelmoinnin käyttämisen verkossa. Seuraavaksi käymme läpi kaksi esimerkkiä dynaamisesta ohjelmoinnista polkujen etsimisessä.

16.3.1 Pisin polku

Tehtävä: Annettuna on suunnattu, syklitön, painotettu verkko. Kuinka pitkä on pisin yksinkertainen polku lähtösolmusta kohdesolmuun?

Yksinkertainen polku tarkoittaa, että sama solmu ei esiinny polussa monta kertaa. Tämä ongelma on NP-vaikea yleisessä verkossa, mutta syklittömässä verkossa tehokas ratkaisu on mahdollinen dynaamisella ohjelmoinnilla.

Esimerkiksi seuraavassa verkossa pisin polku solmusta 1 solmuun 4 kulkee kaaria $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, ja polun painona on $1 + 5 + 3 = 9$.

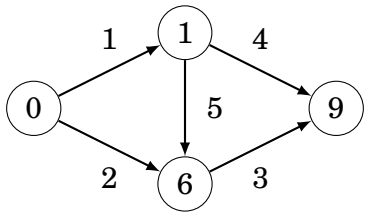


Dynaamisessa ohjelmoinnissa on ideana laskea jokaiseen solmuun, kuinka pitkä on pisin polku alkusolmusta siihen solmuun.

Jos solmu on alkusolmu, pisimmän polun pituus on 0. Muuten polun pituuden saa laskettua käymällä läpi solmuun tulevat kaaret ja valitsemalla pisimmän polun tuottava kaari. Kun solmut käsitellään topologisessa järjestyksessä, nämä kaaret lähtevät solmuista, joihin on jo laskettu pisimmän polun pituus.

Esimerkkiverkossa solmujen topologinen järjestys on 1, 2, 3, 4, joten ensin tulee laskea polun pituus solmuun 1, sitten solmuun 2 jne.

Tuloksena ovat seuraavat pituudet:



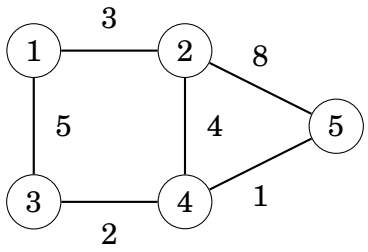
16.3.2 Lyhimmät polut

Tehtävä: Annettuna on yleinen verkko, jossa ei ole negatiivisia kaaria. Montako erilaista lyhintä polkua verkossa on lähtösolmusta kohdesolmuun?

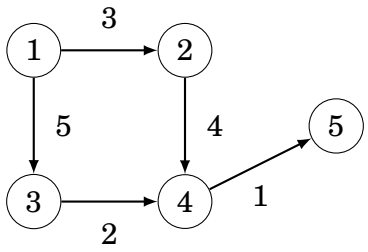
Dijkstran algoritmin tuottama kuvaus verkon lyhimmistä poluista on suunnattu, syklitön verkko, joten siihen voi käyttää dynaamista ohjelmointia.

Ideana on ottaa lyhimpien polkujen kuvaukseen mukaan kaikki kaaret, joita seuraamalla pääsee lyhintä polkua kohdesolmuun. Jos samasta solmusta lähtee useita kaaria, on monta tapaa jatkaa siitä lyhintä polkua kohdesolmuun.

Esimerkiksi verkosta

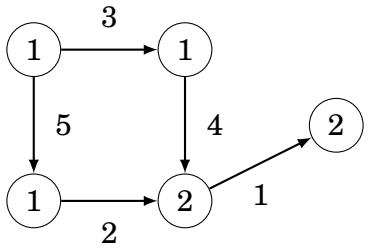


syntyy seuraava kuvaus:



Lyhin polku solmusta 1 solmuun 5 on pituudeltaan 8, ja mahdolliset polut ovat $1 \to 2 \to 4 \to 5$ sekä $1 \to 3 \to 4 \to 5$.

Dynaamisessa ohjelmoinnissa on ideana laskea jokaiseen solmuun, montako lyhintä polkua alkusolmusta siihen solmuun on olemassa:



Luku 17

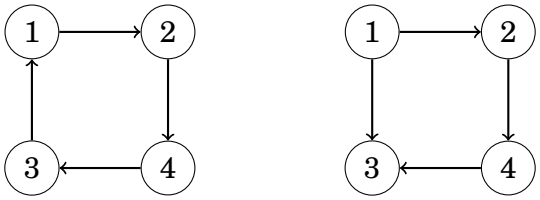
Vahvasti yhtenäisyys

Suunnatussa verkossa yhtenäisyyden käsite on monimutkaisempi kuin suuntaamattomassa verkossa, koska kaaria voi kulkea vain yhteen suuntaan. Vahvasti yhtenäisyys tarkoittaa, että kunkin solmuparin välillä on olemassa polku kumpaankin suuntaan.

Jos verkko ei ole vahvasti yhtenäinen, sen voi kuitenkin jakaa vahvasti yhtenäisiin komponentteihin. Komponenttien muodostama verkko on suunnattu syklitön verkko, joka kuvaa alkuperäisen verkon syvärakenteen.

17.1 Käsitteitä

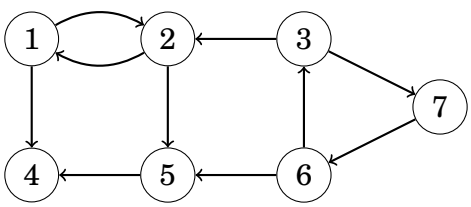
Verkko on vahvasti yhtenäinen (*strongly connected*), jos mistä tahansa solmusta on olemassa polku kaikkiin muihin solmuihin. Esimerkiksi seuraavassa kuvassa vasen verkko on vahvasti yhtenäinen, kun taas oikea verkko ei ole.



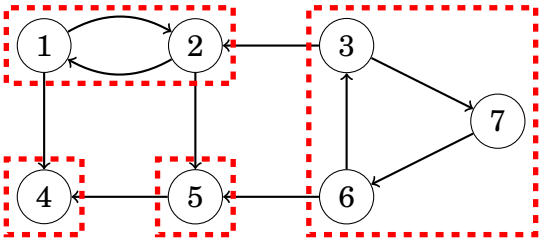
Oikeanpuoleinen verkko ei ole vahvasti yhtenäinen, koska esimerkiksi solmusta 2 ei ole polkua solmuun 1.

Verkon vahvasti yhtenäiset komponentit (*strongly connected components*) jakavat verkon solmut mahdollisimman suuriin vahvasti yhtenäisiin aliverkkoihin. Vahvasti yhtenäiset komponentit muodostavat komponenttiverkon, joka kuvaa alkuperäisen verkon syvärakennetta.

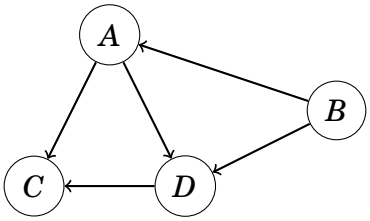
Esimerkiksi verkon



vahvasti yhtenäiset komponentit ovat



ja ne muodostavat seuraavan komponenttiverkon:



Komponentit ovat $A = \{1,2\}$, $B = \{3,6,7\}$, $C = \{4\}$ sekä $D = \{5\}$.

Komponenttiverkko on syklitön suunnattu verkko, jonka käsittely on tietyissä tapauksissa alkuperäistä verkkoa helpompaa. Esimerkiksi luvun 8.3 tyylistä dynaamista ohjelmointia voi soveltaa komponenttiverkkoon.

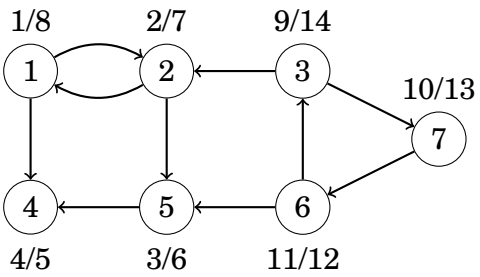
17.2 Kosarajun algoritmi

Kosarajun algoritmi on tehokas menetelmä verkon vahvasti yhtenäisten komponenttien etsimiseen. Se suorittaa verkkoon kaksi syvyyshakua, joista ensimmäinen kerää solmut listaan verkon rakenteen perusteella ja toinen muodostaa vahvasti yhtenäiset komponentit.

Syvyyshaku 1

Ensimmäinen syvyyshaku käy läpi kaikki verkon solmut ja muodostaa listan solmuista siinä järjestyksessä kuin niiden käsittely on päättynyt. Solmu lisätään listalle, kun kaikki siitä lähtevät kaaret on käsitelty.

Seuraava kuva näyttää solmujen käsittelyjärjestyksen esimerkkiverkossa:



Solmun kohdalla oleva merkintä x/y tarkoittaa, että solmun käsittely syvyyshaussa alkoi hetkellä x ja päättyi hetkellä y. Kun solmut järjestetään käsittelyn päättymisajan mukaan, tuloksena on seuraava järjestys:

solmu	päättymisaika
4	5
5	6
2	7
1	8
6	12
7	13
3	14

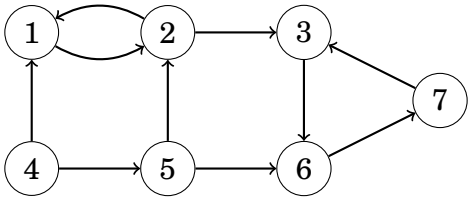
Solmujen käsittelyjärjestys algoritmin seuraavassa vaiheessa tulee olemaan tämä järjestys käänteisenä eli $[3,7,6,1,2,5,4]$.

Syvyyshaku 2

Toinen syvyyshaku muodostaa verkon vahvasti yhtenäiset komponentit.

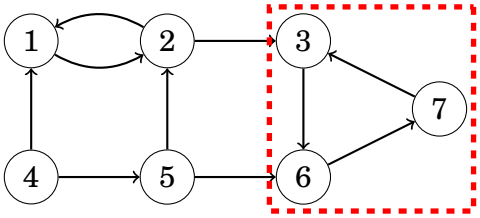
Ennen toista syvyyshakua algoritmi muuttaa jokaisen kaaren suunnan käänteiseksi. Tämä varmistaa, että toisen syvyyshaun aikana löydetään joka kerta vahvasti yhtenäinen komponentti, johon ei kuulu ylimääräisiä solmuja.

Esimerkkiverkko on käännettynä seuraava:



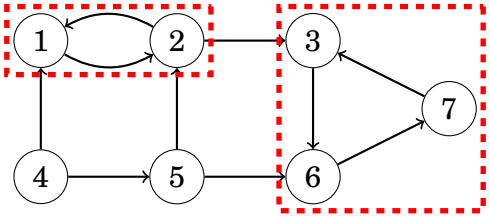
Tämän jälkeen algoritmi käy läpi solmut ensimmäisen syvyyshaun tuottamassa käsittelyjärjestyksessä. Jos solmu ei kuulu vielä komponenttiin, siitä alkaa uusi syvyyshaku käänteisessä verkossa. Solmun komponenttiin kuuluvat kaikki aiemmin käsittelemättömät solmut, joihin syvyyshaku pääsee solmusta.

Esimerkkiverkossa muodostuu ensin komponentti solmusta 3 alkaen:

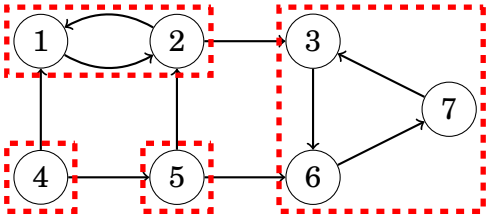


Huomaa, että kaarten kääntämisen ansiosta komponentti ei pääse ”vuotamaan” muihin verkon osiin.

Sitten listalla ovat solmut 7 ja 6, mutta ne on jo liitetty komponenttiin. Seuraava uusi solmu on 1, josta muodostuu uusi komponentti:



Viimeisenä algoritmi käsittelee solmut 5 ja 4, jotka tuottavat loput vahvasti yhtenäiset komponentit:



Algoritmin aikavaativuus on $O(n+m)$, missä n on solmujen määrä ja m on kaarten määrä. Tämä johtuu siitä, että algoritmi suorittaa kaksi syvyyshakua ja kummankin haun aikavaativuus on $O(n+m)$.

17.3 2SAT-ongelma

Vahvasti yhtenäisyys liittyy myös 2SAT-ongelman ratkaisuun. Siinä annettuna on looginen lauseke muotoa

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m)$$

ja tehtävänä on valita muuttujille arvot niin, että lauseke on tosi, tai todeta, että tämä ei ole mahdollista. Merkit ”$\wedge$” ja ”$\vee$” tarkoittavat loogisia operaatioita ”ja” ja ”tai”. Jokainen lausekkeessa esiintyvä a_i ja b_i on looginen muuttuja $(x_1, x_2, \ldots, x_n)$ tai sen negaatio $(\neg x_1, \neg x_2, \ldots, \neg x_n)$.

Esimerkiksi lauseke

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

on tosi, kun x_1 ja x_2 ovat epätosia ja x_3 ja x_4 ovat tosia. Vastaavasti lauseke

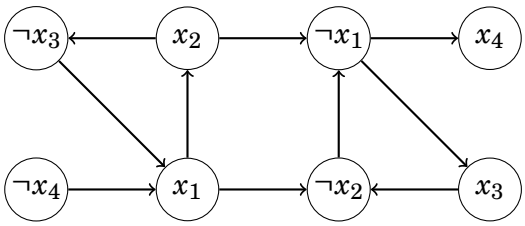
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

on epätosi riippumatta muuttujien valinnasta. Tämä johtuu siitä, että jos x_1 on tosi, pitäisi päteä sekä x_3 että $\neg x_3$, mikä on mahdotonta. Toisaalta jos x_1 on epätosi, pitäisi päteä sekä x_2 että $\neg x_2$, mikä on myös mahdotonta.

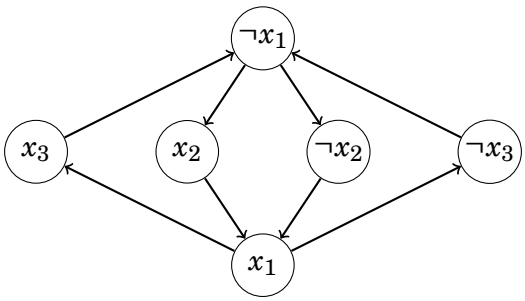
2SAT-ongelman saa muutettua verkoksi niin, että jokainen muuttuja x_i ja negaatio $\neg x_i$ on yksi verkon solmuista ja muuttujien riippuvuudet ovat kaaria.

Jokaisesta parista $(a_i \vee b_i)$ tulee kaksi kaarta: $\neg a_i \to b_i$ sekä $\neg b_i \to a_i$. Nämä tarkoittavat, että jos a_i ei päde, niin b_i:n on pakko päteä, ja päinvastoin.

Lausekkeen L_1 verkosta tulee nyt:



Lausekkeen L_2 verkosta taas tulee:



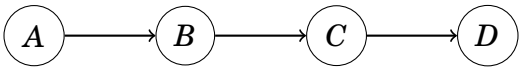
Verkon rakenne kertoo, onko 2SAT-ongelmalla ratkaisua. Jos on jokin muuttuja x_i niin, että x_i ja $\neg x_i$ ovat samassa vahvasti yhtenäisessä komponentissa, niin ratkaisua ei ole olemassa. Tällöin verkossa on polku sekä x_i:stä $\neg x_i$:ään että $\neg x_i$:stä x_i:ään, eli kumman tahansa arvon valitseminen muuttujalle x_i pakottaisi myös valitsemaan vastakkaisen arvon, mikä on ristiriita.

Lausekkeen L_1 verkossa tällaista muuttujaa x_i ei ole, mikä tarkoittaa, että ratkaisu on olemassa. Lausekkeen L_2 verkossa taas kaikki solmut kuuluvat samaan vahvasti yhtenäiseen komponenttiin, eli ratkaisua ei ole olemassa.

Jos ratkaisu on olemassa, muuttujien arvot saa selville käymällä komponenttiverkko läpi käänteisessä topologisessa järjestyksessä. Tällöin verkosta otetaan käsittelyyn ja poistetaan joka vaiheessa komponentti, josta ei lähde kaaria muihin jäljellä oleviin komponentteihin.

Jos komponentin muuttujille ei ole vielä valittu arvoja, ne saavat komponentin mukaiset arvot. Jos taas arvot on jo valittu, niitä ei muuteta. Näin jatketaan, kunnes jokainen muuttuja on saanut arvon.

Lausekkeen L_1 verkon komponenttiverkko on seuraava:



Komponentit ovat $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ sekä $D = \{x_4\}$. Ratkaisun muodostuksessa käsitellään ensin komponentti D, josta x_4 saa arvon tosi. Sitten käsitellään komponentti C, josta x_1 ja x_2 tulevat epätodeksi ja x_3 tulee todeksi. Kaikki muuttujat ovat saaneet arvon, joten myöhemmin käsiteltävät komponentit B ja A eivät vaikuta enää ratkaisuun.

Huomaa, että tämän menetelmän toiminta perustuu verkon erityiseen rakenteeseen. Jos solmusta x_i pääsee solmuun x_j, josta pääsee solmuun $\neg x_j$, niin x_i ei saa koskaan arvoa tosi. Tämä johtuu siitä, että solmusta $\neg x_j$ täytyy päästä myös solmuun $\neg x_i$, koska kaikki riippuvuudet ovat verkossa molempiin suuntiin. Niinpä sekä x_i että x_j saavat varmasti arvokseen epätosi.

2SAT-ongelman vaikeampi versio on 3SAT-ongelma, jossa jokainen lausekkeen osa on muotoa $(a_i \vee b_i \vee c_i)$. Tämän ongelman ratkaisemiseen *ei* tunneta tehokasta menetelmää, vaan kyseessä on NP-vaikea ongelma.

Luku 18

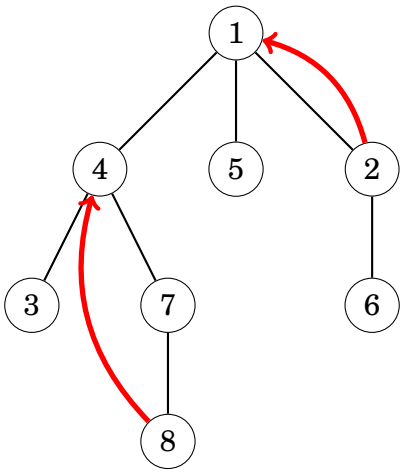
Puukyselyt

Tyypillisiä puukyselyitä ovat puun polkuihin ja alipuihin liittyvät kyselyt. Tämä luku esittelee joukon tekniikoita, joiden avulla on mahdollista toteuttaa puukyselyjä tehokkaasti. Usein esiintyvä idea on muuttaa puu jollakin tavalla taulukoksi, mikä helpottaa puun käsittelyä.

18.1 Tehokas nouseminen

Tehtävä: Annettuna on puu, jolle on valittu juurisolmu. Tehtävänä on vastata kyselyihin muotoa ”mikä solmu on k askelta ylempänä solmua s”.

Merkitään $f(s,k)$ solmua, joka on k askelta ylempänä solmua s. Esimerkiksi seuraavassa puussa $f(2,1) = 1$ ja $f(8,2) = 4$.



Suoraviivainen tapa laskea funktion $f(s,k)$ arvo on kulkea puussa k askelta ylöspäin solmusta s alkaen. Tämän aikavaativuus on kuitenkin $O(n)$, kun puussa on n solmua, koska kaikki puun solmut voivat olla samassa ketjussa.

Kuitenkin funktion $f(s,k)$ arvo on mahdollista laskea ajassa $O(\log n)$ esilaskemalla jokaiseen solmuun joitakin funktion arvoja. Ideana on laskea etukäteen kaikki arvot $f(s,k)$, joissa $k = 1,2,4,8,\ldots$ eli 2:n potenssi. Tämän jälkeen mikä tahansa askelmäärä k muodostuu $O(\log n)$ esilasketusta arvosta.

Kun $k = 1$, arvot $f(s,k)$ on helppo laskea, koska riittää mennä yksi kaari ylemmäs. Kun taas $k > 1$, pätee kaava $f(s,k) = f(f(s,k/2),k/2)$, eli k askeleen

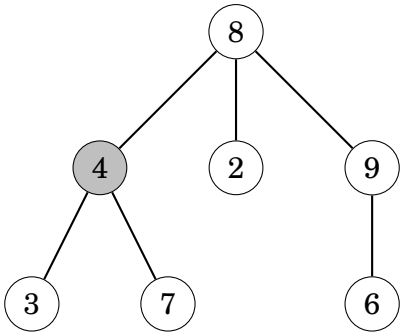
nousun pystyy jakamaan kahdeksi $k/2$ askeleen nousuksi. Esimerkiksi äskeisessä puussa $f(8,1) = 7$ ja $f(7,1) = 4$, joten $f(8,2) = f(f(8,1),1) = 4$.

Esilasketut arvot vievät tilaa $O(n \log n)$, ja niiden avulla pystyy vastaamaan mihin tahansa kyselyyn ajassa $O(\log n)$.

18.2 Alipuut taulukossa

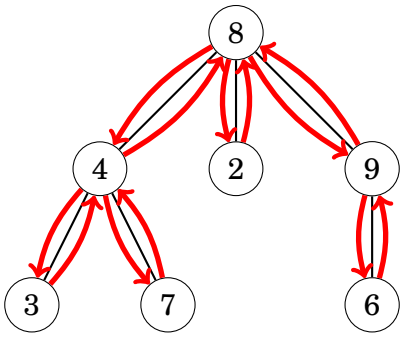
Tehtävä: Annettuna on puu, jolle on valittu juurisolmu. Jokaisella solmulla on tietty paino. Tehtävänä on käsitellä kyselyt muotoa ”muuta solmun s painoa” sekä ”mikä on pienin solmun paino solmun s alipuussa”.

Seuraavassa puussa jokaisessa solmussa lukee sen paino. Esimerkiksi pienin paino harmaalla merkityn solmun alipuussa on 3.



Alipuiden käsittelyssä kätevä tekniikka on käyttää solmutaulukoita, jotka sisältävät tietoa solmuista juuresta alkavan syvyyshaun järjestyksessä. Kukin solmu lisätään taulukkoon silloin, kun syvyyshaku saapuu solmuun.

Esimerkin puussa syvyyshaku etenee näin:



Tässä tehtävässä tarvitaan kaksi taulukkoa: a[s] kertoo solmusta s alkavan alipuun solmujen määrän ja p[s] kertoo solmun s painon. Esimerkin tapauksessa taulukot ovat seuraavat:

a	7	3	1	1	1	2	1

p	8	4	3	7	2	9	6

Nyt taulukko a kertoo mille tahansa solmulle, mikä väli taulukossa p sisältää sen alipuun solmut. Esimerkiksi harmaalla merkitty solmu vastaa taulukoiden kohtia seuraavasti:

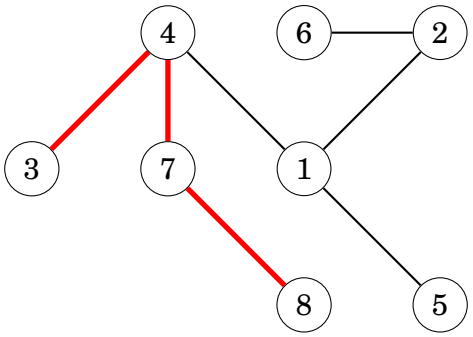
a	7	3	1	1	1	2	1
p	8	4	3	7	2	9	6

Viimeinen tarvittava askel on muodostaa taulukosta p segmenttipuu. Tämän jälkeen ajassa $O(\log n)$ pystyy muuttamaan solmun painoa sekä etsimään mistä tahansa alipuusta solmun, jonka paino on pienin.

18.3 Alin yhteinen vanhempi

Tehtävä: Annettuna on puu, jossa on n solmua. Tehtävänä on vastata kyselyihin muotoa ”kuinka pitkä on polku solmujen a ja b välillä”.

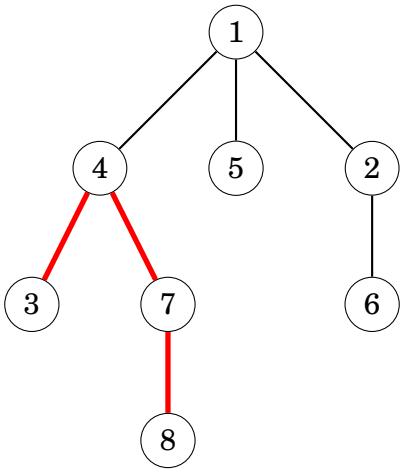
Esimerkiksi seuraavassa puussa polun pituus solmusta 3 solmuun 8 on 3, koska solmu muodostuu kaarista $3 \to 4 \to 7 \to 8$.



Ensimmäinen askel ratkaisussa on valita mikä tahansa solmu puun juureksi. Tämän jälkeen tehtävä palautuu kysymykseen, mikä on kahden solmun alin yhteinen vanhempi (*lowest common ancestor*).

Ideana on, että alin yhteinen vanhempi on käännekohtana polulla solmujen välillä. Jos solmujen a ja b alin yhteinen vanhempi on c ja $s(x)$ on solmun x syvyys puussa, niin solmujen a ja b välimatka on $s(a)+s(b)-2\cdot s(c)$.

Seuraava kuva havainnollistaa asiaa:

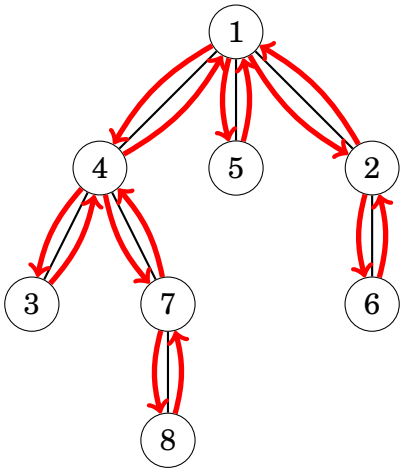


Solmujen 3 ja 8 alin yhteinen vanhempi on 4. Polku solmusta 3 solmuun 8 kulkee ensin ylöspäin solmusta 3 solmuun 4 ja sitten alaspäin solmusta 4 solmuun 8. Solmujen syvyydet ovat $s(3) = 3$, $s(8) = 4$ ja $s(4) = 2$, joten solmujen 3 ja 8 välimatka on $3 + 4 - 2 \cdot 2 = 3$.

18.3.1 Toteutus

Alimman yhteisen vanhemman haun pystyy toteuttamaan solmutaulukoiden avulla. Kuten alipuiden käsittelyssä, ideana on kerätä taulukoihin solmut juuresta alkavan syvyyshaun järjestyksessä. Erona kuitenkin solmu lisätään taulukkoon uudestaan aina silloin, kun haku palaa siihen.

Esimerkin puussa syvyyshaku etenee näin:



Kyselyitä varten tarvitaan kaksi taulukkoa: taulukko x kertoo solmun tunnuksen kussakin syvyyshaun vaiheessa ja taulukko s kertoo vastaavan solmun syvyyden puussa. Esimerkin puusta muodostuu seuraavat taulukot:

x	1	4	3	4	7	8	7	4	1	5	1	2	6	2	1
s	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Nyt solmujen a ja b alin yhteinen vanhempi selviää etsimällä taulukosta x kohdat a' ja b' niin, että $x[a'] = a$ ja $x[b'] = b$. Tämän jälkeen pienin syvyys taulukossa s välillä $a' \ldots b'$ on alimman yhteisen vanhemman kohdalla.

Esimerkiksi solmujen 3 ja 8 alin yhteinen vanhempi löytyy näin:

x	1	4	**3**	4	7	**8**	7	4	1	5	1	2	6	2	1
s	1	2	**3**	**2**	**3**	**4**	3	2	1	2	1	2	3	2	1

Solmuja 3 ja 8 vastaavan välin syvyydet ovat 3, 2, 3 ja 4, ja niistä pienin syvyys on 2. Taulukosta x selviää, että tämä syvyys on solmulla 4.

Jos puun solmu ei ole lehti, niin solmu esiintyy monta kertaa taulukossa x. Tällöin mikä tahansa valinta tuottaa oikean tuloksen.

Luku 19

Polut ja kierrokset

Tässä luvussa tutustumme Hamiltonin ja Eulerin polkuihin. Hamiltonin polku kulkee tasan kerran jokaisen solmun kautta, kun taas Eulerin polku kulkee tasan kerran jokaista kaarta pitkin. Jos polun alku- ja loppusolmu on sama, polkua kutsutaan vastaavasti kierrokseksi.

Yllättävää kyllä, vaikka Hamiltonin ja Eulerin polut muistuttavat toisiaan, niihin liittyy hyvin erilaisia laskennallisia ongelmia. Hamiltonin polun muodostamiseen ei tunneta mitään tehokasta menetelmää, kun taas Eulerin polun pystyy etsimään yksinkertaisella tehokkaalla algoritmilla.

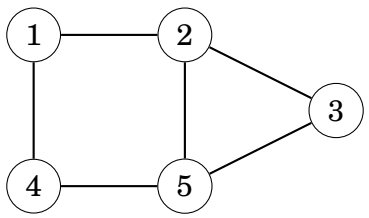
19.1 Hamiltonin polku

Hamiltonin polku (*Hamiltonian path*) on verkossa oleva polku, joka käy tarkalleen kerran jokaisessa verkon solmussa. Hamiltonin kierros (*Hamiltonian cycle*) taas on Hamiltonin polku, jonka alku- ja loppusolmu on sama.

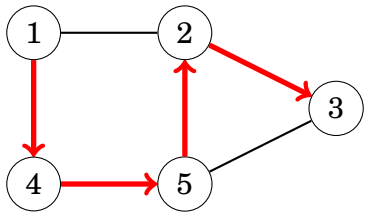
Hamiltonin polkuun liittyvät ongelmat ovat laskennallisesti vaikeita, eikä tällä hetkellä tunneta mitään tehokasta algoritmia, jolla voisi etsiä Hamiltonin polkua tai kierrosta verkosta. Kyseessä on NP-vaikea ongelma, eli luultavasti tehokasta algoritmia ei myöskään ole olemassa.

19.1.1 Esimerkkejä

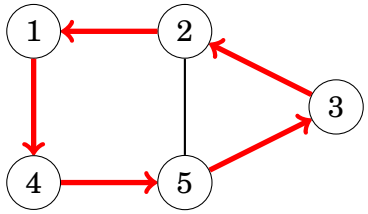
Seuraavassa verkossa on sekä Hamiltonin polku että Hamiltonin kierros:



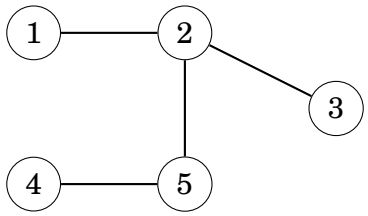
Tässä on esimerkki Hamiltonin polusta solmusta 1 solmuun 3:



Tässä taas on yksi mahdollinen Hamiltonin kierros:



Seuraavassa verkossa ei ole Hamiltonin polkua:



Hamiltonin polkua ei voi muodostaa, koska solmut 1, 3 ja 4 ovat ”umpikujia”, joista polku ei voi jatkaa eteenpäin.

19.1.2 Peruuttava haku

Yksinkertaisin tapa etsiä Hamiltonin polkua on käydä läpi kaikki vaihtoehdot peruuttavalla haulla. Mahdollisia polkuja on $O(n!)$, koska n solmulle voi muodostaa $n!$ erilaista järjestystä. Käytännössä peruuttavan haun tehokkuus riippuu verkon rakenteesta: esimerkiksi jos verkossa on paljon kaaria, jokin Hamiltonin polku löytyy yleensä nopeasti.

19.1.3 Dynaaminen ohjelmointi

Hamiltonin polkua voi etsiä myös dynaamisella ohjelmoinnilla käymällä läpi verkon solmujen osajoukkoja. Osajoukkojen määrä on $O(2^n)$, joten menetelmä on pahimmassa tapauksessa selvästi peruuttavaa hakua tehokkaampi.

Ideana on muotoilla rekursio niin, että tehtävänä on selvittää, onko tietyssä verkon solmujen osajoukossa Hamiltonin polkua, joka päättyy tiettyyn solmuun. Ongelman voi ratkaista käymällä läpi vaihtoehdot valita kaari, jonka kautta viimeiseen solmuun on tultu.

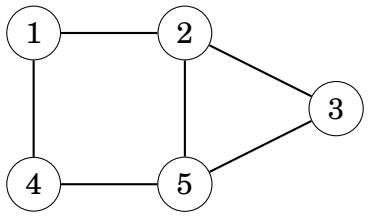
19.2 Eulerin polku

Eulerin polku (*Eulerian path*) on verkossa oleva polku, joka kulkee tarkalleen kerran jokaista verkon kaarta. Eulerin kierros (*Eulerian cycle*) taas on Eulerin polku, jonka alku- ja loppusolmu on sama.

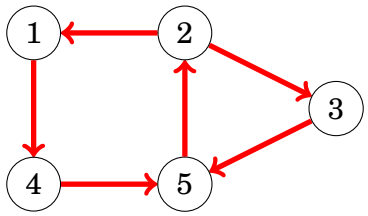
Eulerin polkuun liittyvät ongelmat ovat tehokkaasti ratkaistavissa. Eulerin polun olemassaolon voi päätellä helposti verkon rakenteesta, ja polun muodostamiseen on myös olemassa tehokkaita algoritmeja.

19.2.1 Esimerkkejä

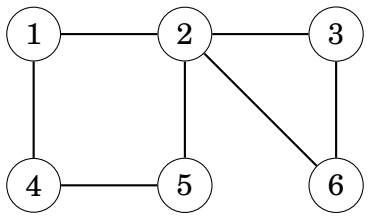
Seuraavassa verkossa on Eulerin polku:



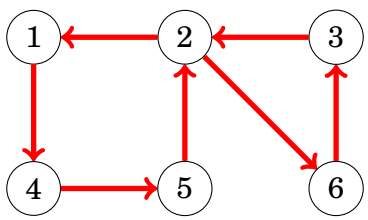
Polun voi muodostaa esimerkiksi näin solmusta 2 solmuun 5:



Seuraavassa verkossa taas on Eulerin kierros:



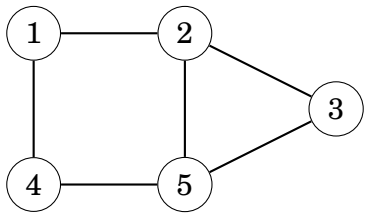
Eulerin kierros näyttää esimerkiksi seuraavalta:



19.2.2 Olemassaolo

Eulerin polun ja kierroksen olemassaolo riippuu verkon solmujen asteista. Solmun aste tarkoittaa, montako kaarta solmuun liittyy.

Olkoon p paritonasteisten solmujen määrä verkossa. Osoittautuu, että verkossa on Eulerin polku tarkalleen silloin, kun $p = 0$ tai $p = 2$, ja Eulerin kierros tarkalleen silloin, kun $p = 0$. Esimerkiksi verkossa



solmujen 1, 3 ja 4 aste on 2 ja solmujen 2 ja 5 aste on 3. Siis $p = 2$, joten verkossa on Eulerin polku mutta ei Eulerin kierrosta.

Jos $p = 2$, paritonasteiset solmut ovat Eulerin polun päätesolmut. Esimerkiksi yllä olevassa verkossa Eulerin polku kulkee solmujen 2 ja 5 välillä.

19.2.3 Muodostus

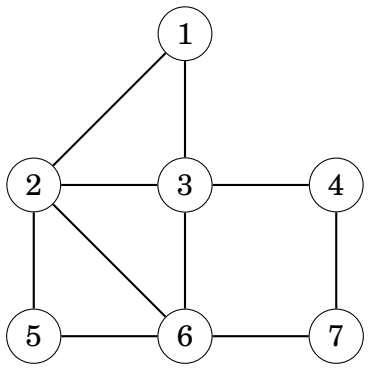
Seuraava algoritmi muodostaa Eulerin kierroksen verkossa, jossa jokaisen solmun aste on parillinen. Algoritmilla voi myös muodostaa Eulerin polun lisäämällä uuden kaaren paritonasteisten solmujen väliin.

Algoritmissa on ideana muodostaa Eulerin kierros joukkona alikierroksia. Alikierros on jokin verkossa oleva polku, jonka alku- ja loppusolmu on sama.

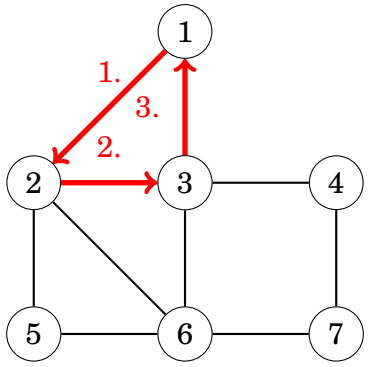
Algoritmi muodostaa ensin pohjan kierrokselle lähtemällä liikkeelle mistä tahansa alkusolmusta ja kulkemalla uusia kaaria eteenpäin, kunnes alkusolmu tulee vastaan uudestaan.

Tämän jälkeen algoritmi alkaa täydentää kierrosta lisäämällä siihen alikierroksia. Uuden alikierroksen voi aloittaa mistä tahansa solmusta, joka on osana kierrosta, mutta josta lähtee kaaria, jotka eivät ole vielä kierroksessa.

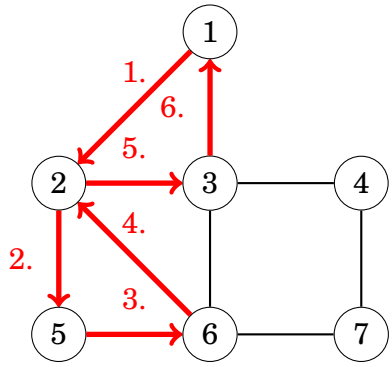
Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



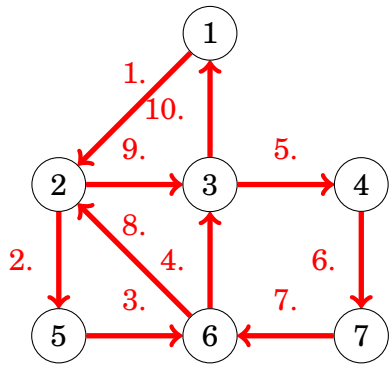
Oletetaan, että algoritmi muodostaa ensimmäisen kierroksen solmusta 1. Siitä syntyy kierros $1 \to 2 \to 3 \to 1$:



Seuraavaksi algoritmi lisää alikierroksen $2 \to 5 \to 6 \to 2$:



Lopuksi algoritmi lisää alikierroksen, $6 \to 3 \to 4 \to 7 \to 6$:



Nyt kaikki kaaret ovat kierroksessa, joten Eulerin kierros on valmis.

Algoritmin toiminta perustuu siihen, että jokaisen solmun aste on parillinen. Tämän ansiosta alikierroksen muodostus onnistuu aina, koska ennemmin tai myöhemmin vastaan tulee alikierroksen alkusolmu.

Toteuttamalla tehokkaasti yllä oleva algoritmi Eulerin kierroksen muodostaminen onnistuu ajassa $O(n+m)$.

19.3 De Bruijnin jono

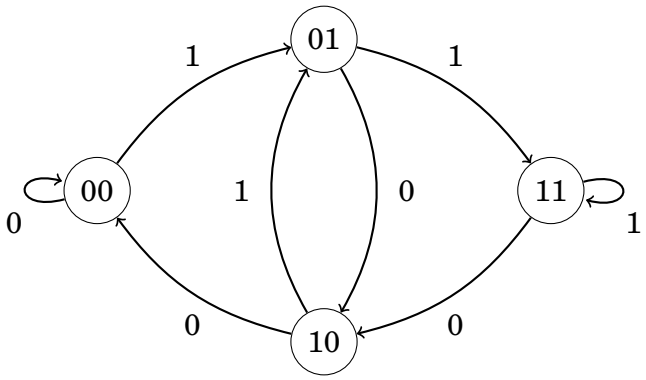
Tarkastellaan lopuksi seuraavaa tehtävää:

Tehtävä: Annettuna on merkistö, jossa on k merkkiä, sekä pituus n. Mikä on lyhin merkkijono, jossa on osana kaikki n merkin yhdistelmät?

Esimerkiksi jos merkistö on $\{0,1\}$ ja $n = 3$, lyhin merkkijono on 10-merkkinen ja yksi tapa muodostaa se on 0001011100. Merkkijonon osana ovat kaikki 3 merkin yhdistelmät 000, 001, 010, 011, 100, 101, 110 ja 111.

Osoittautuu, että lyhimmän merkkijonon pituus on aina $k^n + n - 1$ ja merkkijonon pystyy löytämään etsimällä verkosta Eulerin kierroksen. Tällaista merkkijonoa kutsutaan de Bruijnin jonoksi (*de Bruijn sequence*).

Ideana on muodostaa verkko niin, että jokaisessa solmussa on $n-1$ merkin yhdistelmä ja liikkuminen kaarta pitkin muodostaa uuden n merkin yhdistelmän. Esimerkin tapauksessa verkosta tulee seuraava:



Eulerin kierros tässä verkossa tuottaa lyhimmän merkkijonon, joka sisältää kaikki n merkin yhdistelmät.

Erona aiempaan on, että verkko on suunnattu. Suunnatussa verkossa on Eulerin kierros silloin, kun verkko on vahvasti yhtenäinen ja jokaiseen solmuun tulee yhtä monta kaarta kuin siitä lähtee. Tällöin Eulerin kierroksen voi muodostaa samalla algoritmilla kuin suuntaamattomassa verkossa.

Luku 20

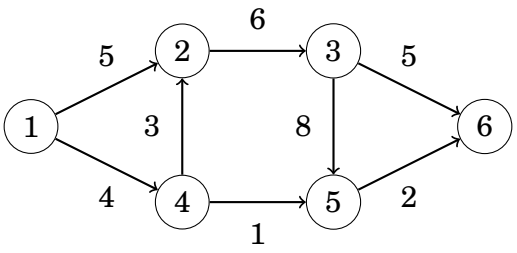
Virtauslaskenta

Virtauslaskenta on verkkoteorian osa-alue, joka tutkii verkossa kulkevia virtauksia. Keskeinen virtauslaskennan ongelma on laskea maksimivirtaus kahden verkon solmun välillä. Maksimivirtausta voi soveltaa monissa verkko-ongelmissa, kuten maksimiparituksen ja minimileikkauksen laskemisessa.

20.1 Maksimivirtaus

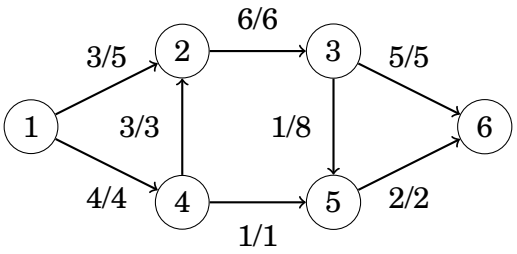
Maksimivirtaus (*maximum flow*) on alkusolmusta loppusolmuun kulkeva virtaus verkossa. Jokaisella verkon kaarella on kapasiteetti, jota kaarta pitkin kulkeva virtaus ei saa ylittää. Lisäksi jokaiseen välisolmuun saapuva virtaus tulee olla yhtä suuri kuin solmusta lähtevä virtaus.

Tarkastellaan esimerkiksi seuraavaa verkkoa:



Solmu 1 on alkusolmu ja solmu 6 on loppusolmu. Jokaiseen kaareen on merkitty kapasiteetti: suurin sallittu määrä kaarta pitkin kulkevalle virtaukselle.

Verkon maksimivirtaus on 7, joka saadaan aikaan seuraavasti:



Merkintä v/k kaaressa tarkoittaa, että kaaren kapasiteetti on k ja sen kautta kulkee virtausta v. Kaikissa välisolmuissa pätee, että solmuun saapuva virtaus on yhtä suuri kuin siitä lähtevä virtaus. Esimerkiksi solmuun 3 tulee virtaus 6 ja siitä lähtee virtaus $5 + 1 = 6$.

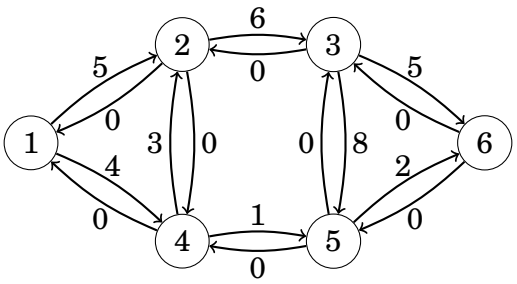
Alkusolmusta lähtevä virtaus ja loppusolmuun saapuva virtaus on verkon maksimivirtaus. Tässä tapauksessa alkusolmusta lähtee virtausta $3 + 4 = 7$ ja loppusolmuun saapuu virtausta $5 + 2 = 7$.

20.2 Ford-Fulkersonin algoritmi

Ford-Fulkersonin algoritmi etsii verkon maksimivirtauksen. Algoritmin ideana on aloittaa tilanteesta, jossa virtaus on 0, ja etsiä sitten verkosta polkuja, jotka tuottavat siihen lisää virtausta. Kun mitään polkua ei enää pysty muodostamaan, maksimivirtaus on valmis.

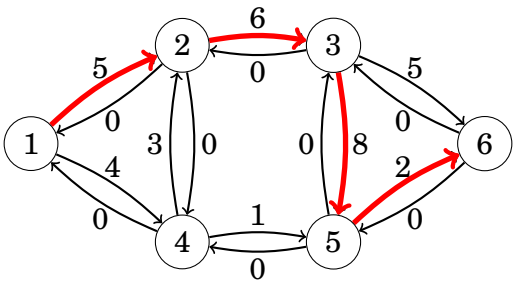
Algoritmi käsittelee verkkoa muodossa, jossa jokaiselle kaarelle on vastakkaiseen suuntaan kulkeva pari. Kaaren paino kuvastaa, miten paljon lisää virtausta sen kautta pystyisi vielä kulkemaan. Aluksi alkuperäisen verkon kaarilla on painona niiden kapasiteetti ja lisätyillä kaarilla on painona 0.

Esimerkkiverkosta syntyy seuraava verkko:



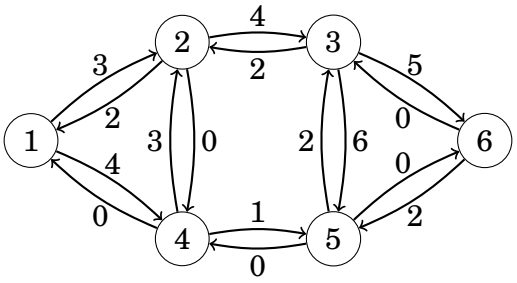
Ford-Fulkersonin algoritmi etsii verkosta joka vaiheessa polun, joka alkaa alkusolmusta, päättyy loppusolmuun ja jossa jokaisen kaaren paino on positiivinen. Jos vaihtoehtoja on useita, mikä tahansa valinta kelpaa.

Esimerkkiverkossa voimme valita vaikkapa seuraavan polun:



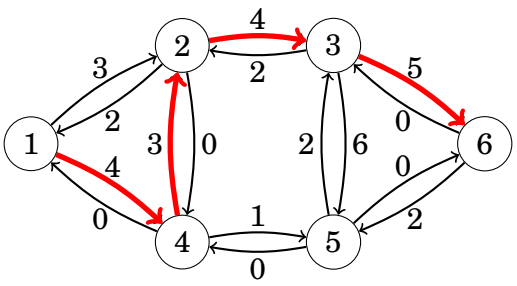
Polun valinnan jälkeen virtaus lisääntyy v yksikköä, jossa v on pienin kaaren paino polulla. Tällöin jokaisen polun osana olevan kaaren paino laskee v:llä ja jokaisen vastakkaiseen suuntaan menevän kaaren paino nousee v:llä.

Tässä tapauksessa pienin kaaren paino on 2, joten virtaus kasvaa 2:lla ja verkko muuttuu seuraavasti:

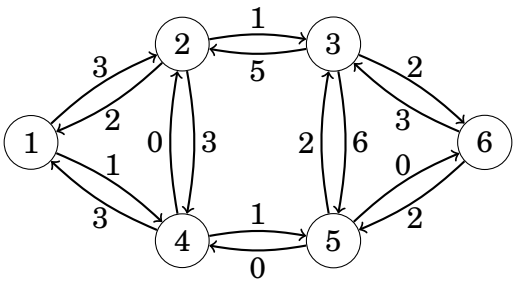


Muutoksessa on ideana, että virtauksen lisääminen vähentää polkuun kuuluvien kaarten kykyä välittää virtausta. Toisaalta virtausta on mahdollista peruuttaa myöhemmin käyttämällä vastakkaiseen suuntaan meneviä kaaria.

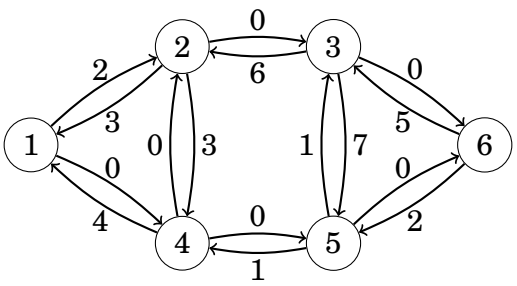
Algoritmi lisää virtausta pikkuhiljaa niin kauan, kuin polkuja alkusolmusta loppusolmuun on olemassa. Esimerkissä seuraava polku voisi olla vaikkapa:



Tämä polku kasvattaa virtausta 3:lla, joten kokonaisvirtaus olisi polun käsittelyn jälkeen 5. Nyt verkko muuttuu seuraavasti:



Maksimivirtaus tulee valmiiksi lisäämällä virtausta vielä polkujen $1 \to 2 \to 3 \to 6$ ja $1 \to 4 \to 5 \to 3 \to 6$ avulla. Molemmat polut tuottavat 1 yksikön lisää virtausta, ja lopullinen verkko on seuraava:



Nyt virtausta ei pysty enää kasvattamaan, koska verkossa ei ole mitään polkua alkusolmusta loppusolmuun, jossa jokaisen kaaren paino olisi positiivinen. Verkon maksimivirtaus on siis 7.

Ford-Fulkersonin algoritmi pysähtyy aina, koska jokainen polku kasvattaa virtausta verkossa. Polun valintatapa kuitenkin vaikuttaa siihen, kuinka tehokas algoritmi on. Pahimmassa tapauksessa jokainen polku lisää virtausta vain 1:llä, jolloin algoritmi voi toimia hitaasti.

Käytännössä hyvä tapa valita polku on käyttää leveyshakua, jolloin polkuun tulee mahdollisimman vähän kaaria. Voidaan osoittaa, että algoritmin aikavaativuus on tällöin $O(nm^2)$, missä n on solmujen määrä ja m on kaarten määrä. Tämä versio Ford-Fulkersonin algoritmista on Edmonds-Karpin algoritmi.

20.3 Sovelluksia

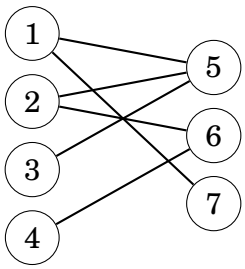
Tavallinen tapa hyödyntää virtauslaskentaa on muuttaa verkko-ongelma sopivalla tavalla maksimivirtauksen laskemiseksi. Seuraavassa on joukko esimerkkejä tällaisista ongelmista.

20.3.1 Maksimiparitus

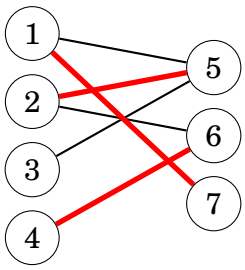
Maksimiparitus (*maximum matching*) on verkko-ongelma, jossa tehtävänä on valita verkosta mahdollisimman suuri joukko kaaria niin, että mikään solmu ei esiinny monen kaaren päätesolmuna.

Maksimiparituksen etsiminen yleisessä verkossa on vaikea ongelma, mutta kaksijakoisessa verkossa se onnistuu maksimivirtauksen avulla. Kaksijakoisessa verkossa solmut jakautuvat kahteen ryhmään, joiden välillä kulkee kaaria.

Tilanne voi näyttää esimerkiksi seuraavalta:

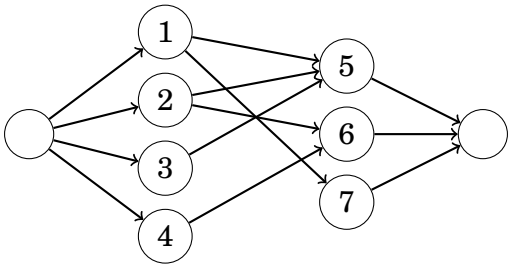


Tämän verkon maksimiparituksessa on 3 paria:



Maksimiparituksen saa muutettua maksimivirtaukseksi lisäämällä verkkoon alkusolmun ja loppusolmun. Alkusolmusta pääsee ensimmäisen ryhmän solmuihin ja loppusolmuun pääsee toisen ryhmän solmuista. Lisäksi ryhmien väliset kaaret muutetaan johtamaan ensimmäisestä ryhmästä toiseen.

Esimerkissä tuloksena on seuraava verkko:

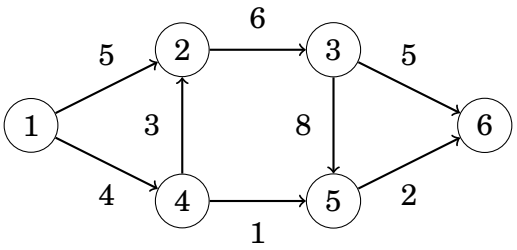


Tämän verkon maksimivirtaus on alkuperäisen verkon maksimiparitus.

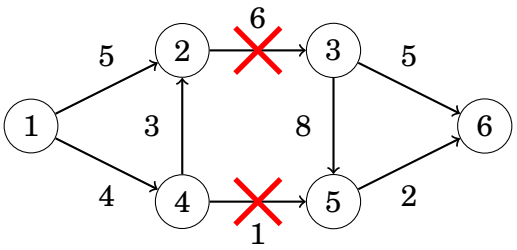
20.3.2 Minimileikkaus

Minimileikkaus (*minimum cut*) on yhteispainoltaan pienin joukko verkon kaaria, joiden poistamisen jälkeen verkossa on kaksi erillistä osaa. Osoittautuu, että maksimivirtaus ja minimileikkaus ovat saman asian kaksi eri puolta.

Esimerkiksi verkon



minimileikkaus on seuraava:

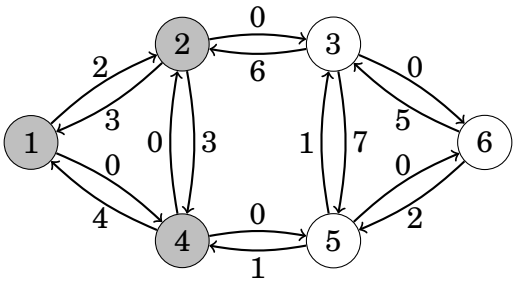


Tässä minimileikkauksessa kaarten yhteispaino on 7.

Yllättävää kyllä, verkon minimileikkaus on aina yhtä suuri kuin maksimivirtaus. Tämä johtuu siitä, että toisaalta minimileikkaus rajoittaa sitä, miten paljon virtausta verkon läpi pystyy kulkemaan, ja toisaalta minimileikkauksen täytyy estää virtauksen kulkeminen verkossa kokonaan.

Maksimivirtauksen ja minimileikkauksen yhteys näkyy Ford-Fulkersonin algoritmin tuottamassa verkossa. Olkoon X niiden solmujen joukko, joihin verkossa pääsee alkusolmusta positiivisia kaaria pitkin.

Esimerkkiverkossa X sisältää solmut 1, 2 ja 4:



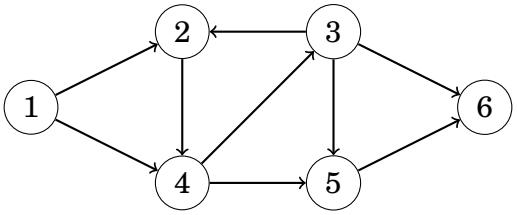
Minimileikkauksen muodostavat ne alkuperäisen verkon kaaret, jotka kulkevat joukosta X joukon X ulkopuolelle ja joiden kapasiteetti on täysin käytetty maksimivirtauksessa. Tässä verkossa kyseiset kaaret ovat $2 \to 3$ ja $4 \to 5$.

20.3.3 Polkujoukko

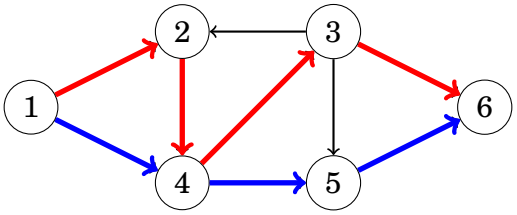
Maksimivirtauksen avulla voi myös etsiä suurimman polkujoukon, jossa jokainen polku alkaa alkusolmusta ja päättyy loppusolmuun ja polut eivät mene päällekkäin missään vaiheessa.

Tarkastellaan ensin tilannetta, jossa samaa kaarta saa käyttää vain yhdessä polussa mutta samaa solmua saa käyttää monessa polussa.

Nyt esimerkiksi verkossa



solmusta 1 solmuun 6 pystyy muodostamaan 2 polkua. Tämä toteutuu valitsemalla polut $1 \to 2 \to 4 \to 3 \to 6$ ja $1 \to 4 \to 5 \to 6$:

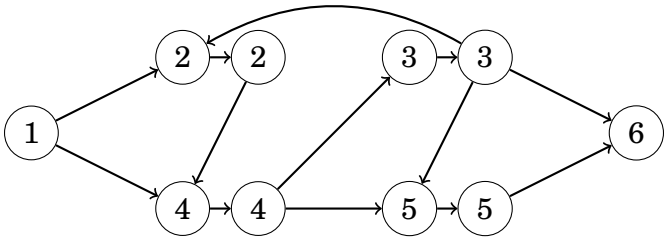


Suurin polkujoukko on sama kuin verkon maksimivirtaus, kun jokaisen kaaren paino on 1. Tämä johtuu siitä, että painon 1 ansiosta jokaista kaarta pystyy käyttämään vain kerran poluissa.

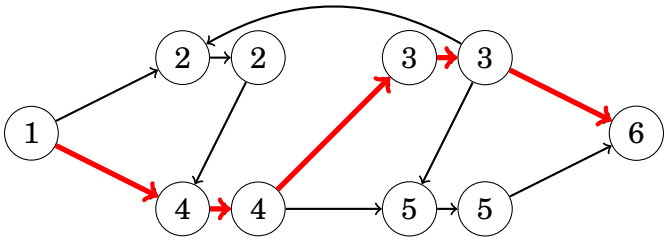
Tarkastellaan sitten tilannetta, jossa myös jokaista solmua (alku- ja loppusolmua lukuun ottamatta) saa käyttää vain kerran.

Tavallinen tapa rajoittaa solmujen kautta kulkemista virtauslaskennassa on korvata jokainen solmu kahdella solmulla, joiden välillä oleva kaari kertoo solmun kapasiteetin. Ensimmäinen solmuista on tulosolmu, johon vain tulee kaaria, ja toinen solmuista on lähtösolmu, josta vain lähtee kaaria.

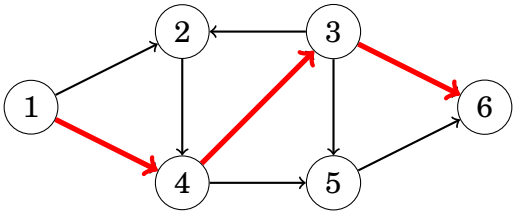
Esimerkkiverkosta tulee nyt:



Maksimivirtaus tässä verkossa on



mikä vastaa alkuperäisen verkon suurinta polkujoukkoa, jossa jokaista kaarta ja solmua saa käyttää vain kerran:



Osa III

Uusia haasteita

Luku 21

Lukuteoria

Lukuteoria on kokonaislukuja tutkiva matematiikan ala, jonka keskeinen teema on lukujen jaollisuus. Tyypillinen lukuteorian tehtävä on selvittää, onko luku alkuluku tai mitkä ovat sen alkutekijät. Tässä luvussa tutustumme kisakoodauksessa usein tarvittaviin lukuteorian algoritmeihin.

21.1 Jaollisuus ja alkuluvut

Luku a on luvun b jakaja eli tekijä, jos b on jaollinen a:lla. Jos a on b:n jakaja, niin merkitään $a \mid b$, ja muuten merkitään $a \nmid b$. Esimerkiksi luvun 24 jakajat ovat 1, 2, 3, 4, 6, 8, 12 ja 24.

Luku n on alkuluku, jos sen ainoat jakajat ovat 1 ja n. Esimerkiksi luvut 7, 19 ja 41 ovat alkulukuja. Luku 35 taas ei ole alkuluku, koska $5 \cdot 7 = 35$. Jokaiselle luvulle $n > 1$ on olemassa yksikäsitteinen alkutekijähajotelma

$$n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k},$$

jossa $p_1, p_2, \ldots, p_k$ ovat alkulukuja. Alkutekijähajotelman avulla saa laskettua luvun n jakajien määrän kaavalla,

$$m(n) = \prod_{i=1}^{k} (a_i + 1)$$

jakajien summan kaavalla

$$s(n) = \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i + 1}.$$

sekä jakajien tulon kaavalla

$$t(n) = n^{m(n)/2}.$$

Esimerkiksi luvun 1176 alkutekijähajotelma on $2^3 \cdot 3^1 \cdot 7^2$, jakajien määrä on $(3+1)(1+1)(2+1) = 24$, summa on $\frac{2^4-1}{2-1} \cdot \frac{3^2-1}{3-1} \cdot \frac{7^3-1}{7-1} = 3420$ ja tulo on 1176^{12}.

21.1.1 Perusalgoritmit

Jos luku n ei ole alkuluku, niin sen voi esittää muodossa $a \cdot b$, missä $a \leq \sqrt{n}$ tai $b \leq \sqrt{n}$, minkä ansiosta sillä on varmasti tekijä välillä $2 \ldots \sqrt{n}$. Tämän havainnon avulla voi tarkastaa ajassa $O(\sqrt{n})$, onko luku alkuluku vai ei, sekä myös selvittää ajassa $O(\sqrt{n})$ luvun alkutekijähajotelman.

Seuraava funktio `prime` selvittää, onko annettu luku n alkuluku. Funktio koettaa jakaa lukua kaikilla luvuilla välillä $2 \ldots \sqrt{n}$, ja jos mikään luvuista ei jaa n:ää, niin n on alkuluku.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

Seuraava funktio `factors` muodostaa vektorin, joka sisältää luvun n alkutekijähajotelman. Funktio jakaa n:ää sen alkutekijöillä ja lisää niitä samaan aikaan vektoriin. Prosessi päättyy, kun jäljellä on luku n, jolla ei ole tekijää välillä $2 \ldots \sqrt{n}$. Jos $n > 1$, se on alkuluku ja viimeinen tekijä.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Huomaa, että funktio lisää jokaisen alkutekijän niin monta kertaa, kuin se jakaa luvun. Esimerkiksi $24 = 2^3 \cdot 3$, joten funktio tuottaa vektorin $[2,2,2,3]$.

21.1.2 Eratostheneen seula

Eratostheneen seula on esilaskenta-algoritmi, jonka suorituksen jälkeen mistä tahansa välin $2 \ldots n$ luvusta pystyy tarkastamaan tehokkaasti, onko se alkuluku, sekä etsimään yhden luvun alkutekijän, jos luku ei ole alkuluku.

Algoritmi luo taulukon `a`, jossa `a`$[k] = 0$ tarkoittaa, että k on alkuluku, ja `a`$[k] \neq 0$ tarkoittaa, että k ei ole alkuluku. Jälkimmäisessä tapauksessa `a`$[k]$ on yksi k:n alkutekijöistä.

Eratostheneen seula käy läpi välin $2 \ldots n$ lukuja yksi kerrallaan. Aina kun uusi alkuluku x löytyy, niin algoritmi merkitsee taulukkoon, että x:n moninkerrat $2x, 3x, 4x, \ldots$ eivät ole alkulukuja, koska niillä on alkutekijä x.

Esimerkiksi jos väli on $2 \ldots 20$, taulukosta tulee:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Seuraava koodi toteuttaa Eratostheneen seulan. Koodi olettaa, että jokainen taulukon a alkio on aluksi 0.

```
for (int x = 2; x <= n; x++) {
    if (a[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        a[u] = x;
    }
}
```

Algoritmin sisäsilmukka suoritetaan n/i kertaa tietyllä i:n arvolla, joten algoritmin ajankäyttöä arvioi harmoninen summa $\sum_{i=2}^{n} n/i = n/2+n/3+n/4+\cdots+n/n$. Harmonisen summan suuruus on luokkaa $O(n \log n)$, mikä on yläraja algoritmin ajankäytölle. Todellisuudessa Eratostheneen seula on vielä nopeampi, koska sisäsilmukka suoritetaan vain, jos luku on alkuluku.

21.1.3 Eukleideen algoritmi

Lukujen a ja b suurin yhteinen tekijä eli $\text{syt}(a,b)$ on suurin luku, jolla sekä a että b on jaollinen. Esimerkiksi $\text{syt}(30,42) = 6$. Vastaavasti pienin yhteinen moninkerta eli $\text{pym}(a,b)$ on pienin luku, joka on jaollinen sekä a:lla että b:llä. Näiden lukujen välillä on yhteys $\text{pym}(a,b) = ab/\text{syt}(a,b)$.

Eukleideen algoritmi on tehokas tapa etsiä lukujen a ja b suurin yhteinen tekijä $\text{syt}(a,b)$. Se laskee suurimman yhteisen tekijän kaavalla

$$\text{syt}(a,b) = \begin{cases} a & b = 0 \\ \text{syt}(b, a \bmod b) & b \neq 0 \end{cases}$$

Esimerkiksi $\text{syt}(24,36) = \text{syt}(36,24) = \text{syt}(24,12) = \text{syt}(12,0) = 12$. Eukleideen algoritmi on tehokas algoritmi, ja on mahdollista osoittaa, että sen aikavaativuus on vain $O(\log n)$, kun $n = \min(a,b)$.

21.1.4 Eulerin totienttifunktio

Luvut a ja b ovat suhteelliset alkuluvut, jos $\text{syt}(a,b) = 1$. Eulerin totienttifunktio $\varphi(n)$ laskee luvun n suhteellisten alkulukujen määrän välillä $1 \ldots n$. Esimerkiksi $\varphi(12) = 4$, koska 1, 5, 7 ja 11 ovat suhteellisia alkulukuja 12:n kanssa.

Totienttifunktion arvon $\varphi(n)$ pystyy laskemaan luvun n alkutekijähajotelmasta kaavalla

$$\varphi(n) = \prod_{i=1}^{k} p_i^{a_i-1}(p_i - 1).$$

Esimerkiksi $\varphi(12) = 2^1 \cdot (2-1) \cdot 3^0 \cdot (3-1) = 4$. Huomaa myös, että $\varphi(n) = n-1$, jos n on alkuluku.

21.2 Modulolaskenta

Modulolaskennan peruslaskusäännöt ovat:

$$\begin{aligned}
(x+y) \bmod M &= (x \bmod M + y \bmod M) \bmod M \\
(x-y) \bmod M &= (x \bmod M - y \bmod M) \bmod M \\
(x \cdot y) \bmod M &= (x \bmod M \cdot y \bmod M) \bmod M \\
(x^k) \bmod M &= (x \bmod M)^k \bmod M
\end{aligned}$$

21.2.1 Tehokas potenssilasku

Modulolaskennassa tulee usein tarvetta laskea tehokkaasti potenssilasku x^n. Tämä onnistuu ajassa $O(\log n)$ seuraavan rekursion avulla:

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

Oleellista on, että parillisen n:n tapauksessa luku $x^{n/2}$ lasketaan vain kerran. Tämän ansiosta potenssilaskun aikavaativuus on $O(\log n)$, koska n:n koko puolittuu aina silloin, kun n on parillinen.

21.2.2 Eulerin lause

Eulerin lauseen mukaan

$$x^{\varphi(M)} \bmod M = 1,$$

kun x ja M ovat suhteelliset alkuluvut. Kun M on alkuluku, saadaan lause

$$x^{M-1} \bmod M = 1,$$

jonka seurauksena

$$(x^k) \bmod M = (x^{k \bmod (M-1)}) \bmod M,$$

kun x ja M ovat suhteelliset alkuluvut.

21.2.3 Modulon käänteisluku

Luvun x käänteisluku modulo M tarkoittaa sellaista lukua x^{-1}, että

$$xx^{-1} \bmod M = 1.$$

Esimerkiksi jos $x = 6$ ja $M = 17$, niin $x^{-1} = 3$, koska $(6 \cdot 3) \bmod 17 = 1$.

Modulon käänteisluku mahdollistaa jakolaskun laskemisen modulossa olevilla luvuilla, koska jakolasku luvulla x vastaa kertolaskua luvulla x^{-1}. Esimerkiksi lasku $36/6 = 6$ laskettuna modulo 17 on $2 \cdot 3 = 6$.

Toisin kuin tavallinen jakolasku, modulon jakolasku ei ole aina mahdollista suorittaa. Käänteisluku x^{-1} modulossa M on olemassa vain silloin, kun x ja M ovat suhteellisia alkulukuja.

Modulon käänteisluvun saa laskettua kaavalla

$$x^{-1} = x^{\varphi(M)-1}.$$

Jos M on alkuluku, kaavasta tulee

$$x^{-1} = x^{M-2}.$$

Esimerkiksi jos $x = 6$ ja $M = 17$, niin $x^{-1} = 6^{17-2} \bmod 17 = 3$.

Modulon käänteisluvun kaava perustuu Eulerin lauseeseen. Modulon käänteisluvulle täytyy päteä

$$xx^{-1} \bmod M = 1.$$

Toisaalta Eulerin kaavan perusteella

$$xx^{\varphi(M)-1} \bmod M = 1,$$

eli lukujen x^{-1} ja $x^{\varphi(M)-1}$ on oltava samat.

21.3 Yhtälönratkaisu

21.3.1 Diofantoksen yhtälö

Diofantoksen yhtälö on muotoa

$$ax + by = c,$$

missä a, b ja c ovat vakioita ja tehtävänä on ratkaista muuttujat x ja y. Kaikki luvut yhtälössä ovat kokonaislukuja. Esimerkiksi jos yhtälö on $5x+2y = 11$, yksi ratkaisu on valita $x = 3$ ja $y = -2$.

Diofantoksen yhtälön voi ratkaista tehokkaasti Eukleideen algoritmin avulla, koska Eukleideen algoritmia laajentamalla pystyy löytämään luvun $\text{syt}(a,b)$ lisäksi luvut x ja y, jotka toteuttavat yhtälön

$$ax + by = \text{syt}(a,b) \cdot z.$$

Alkuperäisen yhtälön ratkaisu on olemassa, jos c on jaollinen $\text{syt}(a,b)$:llä, ja muussa tapauksessa yhtälöllä ei ole ratkaisua.

Laajennettu Eukleideen algoritmi

Etsitään esimerkkinä luvut x ja y, jotka toteuttavat yhtälön

$$39x + 15y = 12,$$

Yhtälöllä on ratkaisu, koska $\text{syt}(39,15) = 3$ ja $3 \mid 12$. Kun Eukleideen algoritmi laskee lukujen 39 ja 15 suurimman yhteisen tekijän, syntyy ketju

$$\text{syt}(39,15) = \text{syt}(15,9) = \text{syt}(9,6) = \text{syt}(6,3) = \text{syt}(3,0) = 3.$$

Algoritmin aikana muodostuvat jakoyhtälöt ovat:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Näiden yhtälöiden avulla saadaan

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

ja kertomalla yhtälö 4:lla tuloksena on

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

joten alkuperäisen yhtälön ratkaisu on $x = 8$ ja $y = -20$.

Diofantoksen yhtälön ratkaisu ei ole yksikäsitteinen, vaan yhdestä ratkaisusta on mahdollista muodostaa äärettömästi muita ratkaisuja. Kun yhtälön ratkaisu on (x, y), niin myös $(x + kb/s, y - ka/s)$ on ratkaisu, missä $s = \text{syt}(a, b)$ ja k on mikä tahansa kokonaisluku.

21.3.2 Kiinalainen jäännöslause

Kiinalainen jäännöslause ratkaisee yhtälöryhmän muotoa

$$\begin{aligned} x &= a_1 \bmod M_1 \\ x &= a_2 \bmod M_2 \\ &\cdots \\ x &= a_n \bmod M_n \end{aligned}$$

missä kaikki parit luvuista $M_1, M_2, \ldots, M_n$ ovat suhteellisia alkulukuja.

Olkoon x_M^{-1} luvun x käänteisluku modulo M ja

$$X_k = \frac{M_1 M_2 \cdots M_n}{M_k}.$$

Näitä merkintöjä käyttäen yhtälöryhmän ratkaisu on

$$x = a_1 X_1 X_{1_{M_1}}^{-1} + a_2 X_2 X_{2_{M_2}}^{-1} + \cdots + a_n X_n X_{n_{M_n}}^{-1}.$$

Ideana on, että jokaisessa yhtälön osassa $a_k X_k X_{k_{M_k}}^{-1}$ jakojäännös M_k:lla on a_k, koska X_k ja sen käänteisluku kumoavat toisensa. Samaan aikaan kaikki muut yhtälön osat ovat jaollisia luvulla M_k, eli ne eivät vaikuta jakojäännökseen ja koko summan jakojäännös M_k:lla on a_k.

Esimerkiksi yhtälöryhmän

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

ratkaisu on

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Kun luku x on yhtälöryhmän ratkaisu, niin myös kaikki luvut muotoa $x + M_1 M_2 \cdot M_n$ ovat ratkaisuja.

Luku 22

Kombinatoriikka

Kombinatoriikka tarkoittaa yhdistelmien määrän laskemista. Tavoitteena on yleensä laskea yhdistelmät tehokkaasti niin, että jokaista yhdistelmää ei tarvitse muodostaa erikseen, vaan yhteismäärän saa selville tehokkaammin hyödyntämällä säännöllisyyksiä ja dynaamista ohjelmointia.

22.1 Perustekniikat

Yhdistelmät

Jos yhdistelmä muodostuu n osasta ja jokainen osa voidaan valita k tavalla, yhdistelmiä on yhteensä k^n. Esimerkiksi n-pituisia bittijonoja on 2^n, koska jokainen jonon bitti voidaan valita 2 tavalla.

Yleisemmin jos yhdistelmä muodostuu n osasta, joista osan 1 voi valita k_1 tavalla, osan 2 voi valita k_2 tavalla, jne., yhdistelmien määrä on $k_1 \cdot k_2 \cdots k_n$.

Permutaatiot

Permutaatio tarkoittaa joukon alkioiden järjestystä. Jos joukossa on n alkiota, niin siitä voi muodostaa $n!$ permutaatiota, koska ensimmäinen alkio voidaan valita n tavalla, toinen $(n-1)$ tavalla, jne. Esimerkiksi kirjaimet `ABC` voidaan järjestää 6 tavalla: `ABC`,`ACB`,`BAC`,`BCA`, `CAB` ja `CBA`.

Rekursiokaava

Kombinatorisen tehtävän ratkaisun pystyy usein ilmaisemaan rekursiolla, jolloin vastauksen saa laskettua tehokkaasti dynaamisella ohjelmoinnilla. Näin on esimerkiksi seuraavassa tehtävässä:

> *Tehtävä:* Tehtäväsi on laskea, monessako n bitin jonossa ei ole vierekkäin kahta ykkösbittiä. Esimerkiksi jos $n = 4$, vastaus on 8, koska bittijonot ovat `0000`, `0001`, `0010`, `0100`, `0101`, `1000`, `1001` ja `1010`.

Vastaus selviää funktiolla $f(n,k)$, jossa n on bittijonon pituus ja k on viimeinen bitti (0 tai 1). Tätä funktiota käyttäen n bitin jonojen määrä on summa $f(n,0)+f(n,1)$. Rekursiivinen kaava on seuraava:

$$f(n,k) = \begin{cases} 1 & n = 1 \\ f(n-1,0)+f(n-1,1) & k = 0 \\ f(n-1,0) & k = 1 \end{cases}$$

Kaavan ideana on, että jos viimeinen bitti on 0, sitä ennen voi olla sekä bitti 0 että bitti 1, ja jos viimeinen bitti on 1, sitä ennen on pakko olla bitti 0.

22.2 Binomikerroin

Binomikerroin $\binom{n}{k}$ ilmaisee, montako erilaista k alkion osajoukkoa n alkion joukosta voidaan muodostaa. Esimerkiksi $\binom{5}{2} = 10$, koska joukosta $\{A,B,C,D,E\}$ voidaan muodostaa 10 erilaista 2 alkion osajoukkoa:

$$\{A,B\},\{A,C\},\{A,D\},\{A,E\},\{B,C\},\{B,D\},\{B,E\},\{C,D\},\{C,E\},\{D,E\}$$

Laskutapa 1

Tavallisin tapa laskea binomikerroin on käyttää kaavaa

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

Esimerkiksi

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

Huomaa myös, että

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

koska k alkion valinta osajoukkoon tarkoittaa samalla $n-k$ alkion valintaa osajoukon ulkopuolelle.

Laskutapa 2

Binomikertoimen voi myös laskea rekursiivisesti kaavalla

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

Pohjatapauksina $\binom{n}{0} = 1$, koska tyhjän osajoukon voi muodostaa aina tasan yhdellä tavalla, ja $\binom{n}{k} = 0$, jos $k > n$, koska n alkiosta ei ole mahdollista valita yli n alkion osajoukkoa.

Rekursion voi perustella tarkastelemalla yksittäistä joukon alkiota x. Jos alkio x valitaan osajoukkoon, täytyy vielä valita $n-1$ alkiosta $k-1$ alkiota. Jos taas alkiota x ei valita, täytyy vielä valita $n-1$ alkiosta k alkiota.

Multinomikerroin

Binomikertoimen yleistys on multinomikerroin

$$\binom{n}{k_1,k_2,\ldots,k_m} = \frac{n!}{k_1!k_2!\cdots k_m!},$$

missä $k_1+k_2+\cdots+k_m = n$. Multinomikerroin ilmaisee, monellako tavalla n alkiota voidaan jakaa osajoukkoihin, joiden koot ovat $k_1,k_2,\ldots,k_m$. Jos $m = 2$, niin multinomikertoimen kaava vastaa binomikertoimen kaavaa.

Hatut ja pallot

Binomikertoimen avulla voi ratkaista myös seuraavan tehtävän:

> *Tehtävä:* Rivissä on n hattua, joihin sijoitetaan m palloa. Montako erilaista tapaa tähän on? Esimerkiksi jos $n = 3$ ja $m = 2$, niin tapoja on 6: $[0,0,2]$, $[0,1,1]$, $[0,2,0]$, $[1,0,1]$, $[1,1,0]$, $[2,0,0]$.

Ratkaisu tehtävään on $\binom{n+m-1}{m}$. Jokaisen sijoitustavan voi ilmaista merkkijonona, jossa `o` kuvaa palloa ja `|` on kahden hatun raja. Merkkijonon pituus on $n+m-1$, koska palloja on m ja hatun rajoja on $n-1$, ja siitä valitaan m paikkaa palloille. Esimerkiksi sijoitustapaa $[1,0,1]$ vastaa merkkijono `o||o`.

22.3 Muita funktioita

Catalanin luvut

Catalanin luku C_n ilmaisee, montako tapaa on muodostaa kelvollinen sulkulauseke n alkusulusta ja n loppusulusta. Esimerkiksi $C_3 = 5$, koska vastaavat sulkulausekkeet ovat `()()()`, `(())()`, `()(())`, `((()))` ja `(()())`.

Luonteva tapa laskea Catalanin lukuja on käyttää rekursiokaavaa

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1},$$

jossa pohjatapauksena on $C_0 = 1$. Catalanin luvut saa laskettua myös binomikertoimen avulla kaavalla

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Stirlingin luvut

Stirlingin luku $\left\{{n \atop k}\right\}$ ilmaisee, montako tapaa on jakaa n alkion joukko k epätyhjäksi osajoukoksi. Esimerkiksi $\left\{{4 \atop 2}\right\} = 7$, koska joukon $\{A,B,C,D\}$ jakotavat 2 epätyhjäksi osajoukoksi ovat $\{A,B,C\}$ ja $\{D\}$, $\{A,B,D\}$ ja $\{C\}$, $\{A,C,D\}$ ja $\{B\}$, $\{B,C,D\}$ ja $\{A\}$, $\{A,B\}$ ja $\{C,D\}$, $\{A,C\}$ ja $\{B,D\}$, sekä $\{A,D\}$ ja $\{B,C\}$.

Stirlingin lukuja voi laskea rekursiivisesti kaavalla

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\},$$

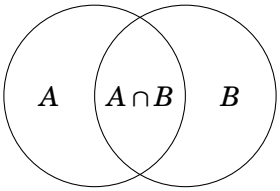
jossa pohjatapauksina $\left\{ {n \atop k} \right\} = 0$, kun $k = 0$ tai $k > n$, ja $\left\{ {n \atop k} \right\} = 1$, kun $k = 1$.

22.4 Sisään-ulos-periaate

Sisään-ulos-periaate eli inkluusio-ekskluusio on tekniikka, jonka avulla pystyy laskemaan joukkojen yhdisteen koon leikkausten kokojen perusteella ja päinvastoin. Yksinkertainen esimerkki periaatteesta on kaava

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

jossa A ja B ovat joukkoja ja $|X|$ on joukon koko. Seuraava kuva havainnollistaa kaavaa, kun joukot ovat tason ympyröitä:

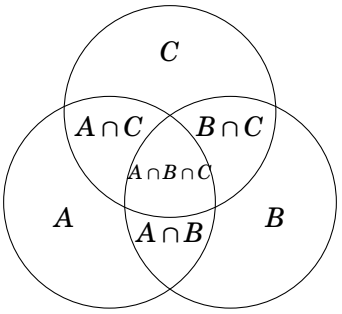


Tavoitteena on laskea, kuinka suuri on yhdiste $A \cup B$ eli alue, joka on ainakin toisen ympyrän sisällä. Kuvan mukaisesti yhdisteen $A \cup B$ koko saadaan laskemalla ensin yhteen ympyröiden A ja B koot ja vähentämällä siitä sitten leikkauksen $A \cap B$ koko.

Samaa ideaa voi soveltaa, kun joukkoja on enemmän. Kolmen joukon tapauksessa kaavasta tulee

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

ja vastaava kuva on



Yleisessä tapauksessa yhdisteen $X_1 \cup X_2 \cup \cdots \cup X_n$ koon saa laskettua käymällä läpi kaikki tavat muodostaa leikkaus joukoista $X_1, X_2, \ldots, X_n$. Parittoman määrän joukkoja sisältävät leikkaukset lasketaan mukaan positiivisina ja parillisen määrän negatiivisina.

Huomaa, että vastaavat kaavat toimivat myös käänteisesti leikkauksen koon laskemiseen yhdisteiden kokojen perusteella. Esimerkiksi

$$|A \cap B| = |A| + |B| - |A \cup B|$$

ja

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Esimerkki

> *Tehtävä:* Taulukossa on luvut $1, 2, \ldots, n$ järjestyksessä. Tehtäväsi on muuttaa järjestystä niin, että mikään luvuista ei jää alkuperäiselle paikalleen. Montako tapaa tähän on olemassa? Esimerkiksi jos $n = 3$, ratkaisuja on 2: $(2,3,1)$ ja $(3,1,2)$

Yksi tapa lähestyä tehtävää on käyttää sisään-ulos-periaatetta. Olkoon joukko X_k niiden permutaatioiden joukko, jossa kohdassa k on luku k. Esimerkiksi jos $n = 3$, niin joukot ovat seuraavat:

$$\begin{aligned} X_1 &= \{(1,2,3),(1,3,2)\} \\ X_2 &= \{(1,2,3),(3,2,1)\} \\ X_3 &= \{(1,2,3),(2,1,3)\} \end{aligned}$$

Näitä joukkoja käyttäen ratkaisujen määrä on

$$n! - |X_1 \cup X_2 \cup \cdots \cup X_n|,$$

eli riittää laskea joukkojen yhdisteen koko. Tämä palautuu sisään-ulos-periaatteen avulla joukkojen leikkausten kokojen laskemiseen, mikä onnistuu tehokkaasti. Esimerkiksi kun $n = 3$, joukon $|X_1 \cup X_2 \cup X_3|$ koko on

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ = \; & 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ = \; & 4, \end{aligned}$$

joten ratkaisujen määrä on $3! - 4 = 2$.

22.5 Burnsiden lemma

Burnsiden lemma laskee yhdistelmien määrän niin, että symmetrisistä yhdistelmistä lasketaan mukaan vain yksi edustaja. Burnsiden lemman mukaan yhdistelmien määrä on

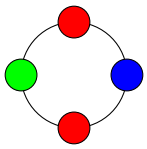
$$\sum_{k=1}^{n} \frac{c(k)}{n},$$

missä yhdistelmän asentoa voi muuttaa n tavalla ja $c(k)$ on niiden yhdistelmien määrä, jotka pysyvät ennallaan, kun asentoa muutetaan tavalla k.

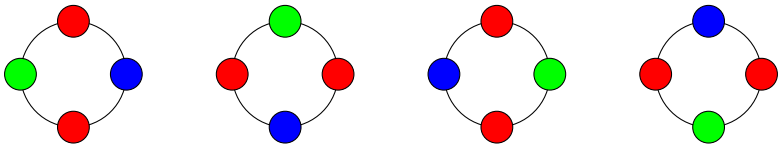
Tutustumme Burnsiden lemmaan seuraavan tehtävän kautta:

> *Tehtävä:* Helminauhassa on n helmeä, joista jokaisen väri on väliltä $1, 2, \ldots, m$. Montako erilaista helminauhaa on olemassa? Kaksi helminauhaa ovat symmetriset, jos ne voi saada näyttämään samalta pyörittämällä.

Esimerkiksi helminauhan



kanssa symmetriset helminauhat ovat seuraavat:



Tapoja muuttaa asentoa on n, koska helminauhaa voi pyörittää $0, 1, \ldots, n-1$ askelta myötäpäivään. Jos helminauhaa pyörittää 0 askelta, kaikki m^n väritystä säilyvät ennallaan. Jos taas helminauhaa pyörittää 1 askeleen, vain m yksiväristä helminauhaa säilyy ennallaan.

Yleisemmin kun helminauhaa pyörittää k askelta, ennallaan säilyvien yhdistelmien määrä on

$$m^{\mathrm{syt}(k,n)},$$

missä $\mathrm{syt}(k,n)$ on lukujen k ja n suurin yhteinen tekijä. Tämä johtuu siitä, että $\mathrm{syt}(k,n)$-kokoiset pätkät helmiä siirtyvät toistensa paikoille k askelta eteenpäin. Niinpä helminauhojen määrä on Burnsiden lemman mukaan

$$\sum_{i=0}^{n-1} \frac{m^{\mathrm{syt}(i,n)}}{n}.$$

Esimerkiksi kun helminauhan pituus on 4 ja värejä on 3, helminauhoja on

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

Luku 23

Matriisit

Matriisi on kaksiulotteinen taulukko, jolle on määritelty laskutoimituksia. Tässä luvussa näemme, miten matriisien avulla voi optimoida dynaamista ohjelmointia. Osoittautuu, että jos dynaamisen ohjelmoinnin rekursiossa lasketaan yhteen kiinteä määrä aiempia arvoja vakiokertoimilla, ratkaisun aikavaativuuden saa muutettua lineaarisesta logaritmisesta matriisien avulla.

23.1 Määritelmiä

Matriisi (*matrix*) 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 $A_{2,3} = 5$.

Yhteenlasku

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:

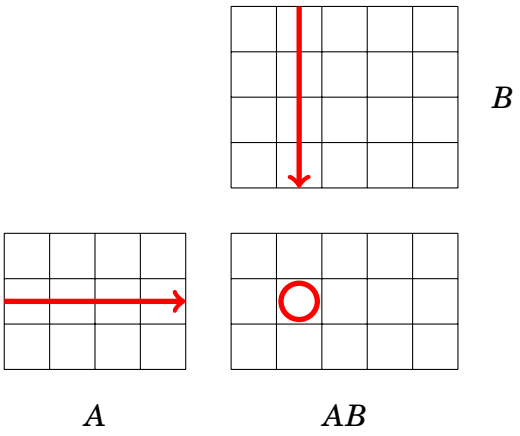
$$A = \begin{pmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{pmatrix} \qquad B = \begin{pmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{pmatrix} = \begin{pmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{pmatrix}$$

Kertolasku

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 $n \times b$. Kertolaskun tuloksena on uusi matriisi, jonka koko on $a \times b$.

Matriisien kertolaskussa jokainen tulosmatriisin luku muodostuu summana A:n ja B:n lukuparien tuloista. Summa muodostuu seuraavan kuvan tapaan:



Kertolaskun matemaattinen määritelmä on:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

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 on voimassa $A(BC) = (AB)C$.

Matriisikertolaskun aikavaativuus kolmea for-silmukkaa käyttäen on $O(n^3)$, kun matriisin koko on $n \times n$.

Potenssilasku

Matriisien potenssilasku määritellään saman tapaan kuin tavallinen potenssilasku, esimerkiksi $A^3 = AAA$.

Potenssilaskun A^k aikavaativuus on $O(n^3 k)$, jos sen toteuttaa k kertolaskuna. Mutta potenssilaskun voi toteuttaa myös ajassa $O(n^3 \log k)$ tehokkaalla potenssilaskulla (luku 21.2).

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.

23.2 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}, kertominen matriisilla X tuottaa uuden matriisin, 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

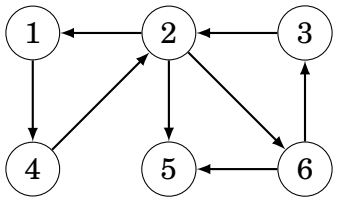
$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Matriisin potenssilasku onnistuu ajassa $O(\log n)$, joten tällä tekniikalla Fibonaccin luvun F_n pystyy laskemaan ajassa $O(\log n)$. Samaa ideaa voi myös soveltaa aina, kun rekursiivisen funktion seuraava arvo saadaan laskemalla yhteen kiinteä määrä edellisiä arvoja sopivilla kertoimilla.

23.3 Verkko matriisina

Matriisin potenssilaskulla on mielenkiintoinen vaikutus painottoman verkon vierusmatriisin sisältöön. Kun V on vierusmatriisi, jossa jokainen arvo on 0 tai 1, niin 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 esimerkiksi matriisi V^3 kertoo 3:n pituisten polkujen määrät:

$$V^3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

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 $2 \to 6 \to 3 \to 2$.

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

$$(AB)_{i,j} = \min_{k=1}^{n}(A_{i,k} + B_{k,j}).$$

Luku 24

Todennäköisyys

Tässä luvussa tutustumme todennäköisyyslaskennan perustekniikoihin, joita tarvitsee yleisimmin kisakoodauksessa. Todennäköisyyksien laskenta on pääasiassa matematiikkaa, mutta usein mukana esiintyy myös dynaamista ohjelmointia, jotta todennäköisyyden saa laskettua tehokkaasti.

24.1 Perusasiat

Todennäköisyys on luku väliltä $0 \ldots 1$, joka kuvaa sitä, miten todennäköinen jokin tapahtuma on. Varman tapahtuman todennäköisyys on 1, ja mahdottoman tapahtuman todennäköisyys on 0.

Tyypillinen esimerkki todennäköisyydestä on nopan heitto, jossa tuloksena on silmäluku väliltä $1, 2, \ldots, 6$. Yleensä oletetaan, että kunkin silmäluvun todennäköisyys on 1/6 eli kaikki tulokset ovat yhtä todennäköisiä.

Tapahtuman todennäköisyyttä merkitään $P(\cdots)$, jossa kolmen pisteen tilalla on tapahtuman kuvaus. Esimerkiksi nopan heitossa P(”silmäluku on 4”) = 1/6, P(”silmäluku ei ole 6”) = 5/6 ja P(”silmäluku on parillinen”) = 1/2.

24.1.1 Laskutavat

Tapahtuman todennäköisyyden laskemiseen on kaksi tavallista laskutapaa. Tutustumme niihin seuraavan tehtävän avulla:

Tehtävä: Korttipakassa on 52 korttia, jotka jakaantuvat neljään maahan (hertta, risti, ruutu, pata). Kussakin maassa on 13 korttia, joiden arvot ovat $1, 2, \ldots, 13$. Sinulle jaetaan kolme korttia pakasta. Mikä on todennäköisyys, että jokaisen kortin arvo on sama?

Laskutapa 1

Kombinatorisessa laskutavassa todennäköisyyden kaava on

$$\frac{\text{halutut tapaukset}}{\text{kaikki tapaukset}}.$$

Tässä tehtävässä halutut tapaukset ovat niitä, joissa jokaisen kolmen kortin arvo on sama. Tällaisia tapauksia on $13\binom{4}{3}$, koska on 13 vaihtoehtoa, mikä on kortin arvo, ja $\binom{4}{3}$ tapaa valita 3 maata 4 mahdollisesta.

Kaikkien tapausten määrä on $\binom{52}{3}$, koska 52 kortista valitaan 3 korttia. Niinpä tapahtuman todennäköisyys on

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Laskutapa 2

Toinen tapa laskea todennäköisyys on simuloida prosessia, jossa tapahtuma syntyy. Tässä tapauksessa pakasta nostetaan kolme korttia, joten prosessissa on kolme vaihetta. Ideana on vaatia, että prosessin jokainen vaihe onnistuu.

Ensimmäisen kortin nosto onnistuu varmasti, koska mikä tahansa kortti kelpaa. Tämän jälkeen kahden seuraavan kortin arvon tulee olla sama. Toisen kortin nostossa kortteja on jäljellä 51 ja niistä 3 kelpaa, joten todennäköisyys on 3/51. Vastaavasti kolmannen kortin nostossa todennäköisyys on 2/50.

Todennäköisyys koko prosessin onnistumiselle on

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.1.2 Tapahtumat

Todennäköisyyden tapahtuma voidaan tulkita joukkona, joka sisältää alkeistapauksia. Esimerkiksi nopan heitossa alkeistapaukset ovat $\{1,2,\ldots,6\}$, missä kukin alkeistapaus vastaa silmälukua ja sen todennäköisyys on 1/6.

Esimerkiksi joukko $A = \{2,4,6\}$ vastaa tapahtumaa ”silmäluku on parillinen” ja joukko $B = \{1,2,3,4\}$ vastaa tapahtumaa ”silmäluku on alle 5”. Tapahtumien todennäköisyydet ovat $P(A) = 1/2$ ja $P(B) = 2/3$.

Komplementti $\bar{A}$ tarkoittaa tapahtumaa ”A ei tapahdu”. Yhdiste $A \cup B$ tarkoittaa ”A tai B tapahtuu”, ja leikkaus $A \cap B$ tarkoittaa ”A ja B tapahtuvat”.

Komplementti

Komplementin $\bar{A}$ todennäköisyys lasketaan $P(\bar{A}) = 1 - P(A)$. Joskus tehtävän ratkaisu on kätevää laskea vastatapahtuman kautta. Esimerkiksi todennäköisyys saada 10 nopan heitossa ainakin kerran silmäluku 6 on $1 - (5/6)^{10}$, missä 5/6 on todennäköisyys, että silmäluku ei ole 6.

Yhdiste

Yhdisteen $A \cup B$ todennäköisyys lasketaan kaavalla

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Esimerkiksi jos A = ”silmäluku on parillinen” ja B = ”silmäluku on alle 5”, niin tapahtuman $A \cup B$ todennäköisyys on $P(A)+P(B)-P(A \cap B) = 1/2+2/3-1/3 = 5/6$. Joukko $A \cup B$ sisältää tapaukset $\{1,2,3,4,6\}$.

Jos tapahtumat A ja B ovat erilliset eli $A \cap B$ on tyhjä, yhdisteen $A \cup B$ todennäköisyys on yksinkertaisesti

$$P(A \cup B) = P(A) + P(B).$$

Leikkaus

Leikkauksen $A \cap B$ todennäköisyys lasketaan kaavalla

$$P(A \cap B) = P(A)P(B|A),$$

missä $P(B|A)$ on B:n todennäköisyys ehdolla, että A on tapahtunut.

Esimerkiksi jos A = ”silmäluku on parillinen” ja B = ”silmäluku ei ole 6”, niin tapahtuman $A \cap B$ todennäköisyys on $P(A)P(B|A) = 1/2 \cdot 2/3 = 1/3$. Joukko $A \cap B$ sisältää tapaukset $\{2,4\}$.

Jos tapahtumat A ja B ovat riippumattomat eli A:n tapahtuminen ei vaikuta B:n todennäköisyyteen ja päinvastoin, leikkauksen todennäköisyys on

$$P(A \cap B) = P(A)P(B),$$

koska tässä tapauksessa $P(B|A) = P(B)$.

24.2 Satunnaismuuttuja

Satunnaismuuttuja on arvo, joka syntyy satunnaisen prosessin tuloksena. Satunnaismuuttujaa merkitään yleensä suurella kirjaimella. Kahden nopan heitossa yksi mahdollinen satunnaismuuttuja on X = ”silmälukujen summa”. Esimerkiksi jos heitot ovat $[4,6]$, niin X saa arvon 10.

Merkintä $P(X = x)$ tarkoittaa todennäköisyyttä, että satunnaismuuttujan X arvo on x. Edellisessä esimerkissä $P(X = 10) = 3/36$, koska erilaisia heittotapoja on 36 ja niistä summan 10 tuottavat heitot $[4,6]$, $[5,5]$ ja $[6,4]$.

24.2.1 Jakaumat

Satunnaismuuttujan jakauma kertoo, millä todennäköisyydellä satunnaismuuttuja saa minkäkin arvon. Jakauma muodostuu arvoista $P(X = x)$. Esimerkiksi kahden nopan heitossa silmälukujen summan jakauma on:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Tutustumme seuraavaksi muutamaan usein esiintyvään jakaumaan.

Tasajakauma

Tasajakaumassa satunnaismuuttuja saa arvoja väliltä $a, a+1, \ldots, b$ ja jokaisen arvon todennäköisyys on sama. Esimerkiksi yhden nopan heitto tuottaa tasajakauman, jossa $P(X = x) = 1/6$, kun $x = 1, 2, \ldots, 6$.

Binomijakauma

Binomijakauma kuvaa tilannetta, jossa tehdään n yritystä ja joka yrityksessä onnistumisen todennäköisyys on p. Satunnaismuuttuja X on onnistuneiden yritysten määrä.

Satunnaismuuttujan arvon x todennäköisyys on

$$P(X = x) = p^x(1-p)^{n-x}\binom{n}{x},$$

missä p^x kuvaa onnistuneita yrityksiä, $(1-p)^{n-x}$ kuvaa epäonnistuneita yrityksiä ja $\binom{n}{x}$ antaa erilaiset tavat, miten yritykset sijoittuvat toisiinsa nähden.

Esimerkiksi jos heitetään 10 kertaa noppaa, todennäköisyys saada tarkalleen 3 kertaa silmäluku 6 on $(1/6)^3(5/6)^7\binom{10}{3}$.

Geometrinen jakauma

Geometrinen jakauma kuvaa tilannetta, jossa onnistumisen todennäköisyys on p ja yrityksiä tehdään, kunnes tulee ensimmäinen onnistuminen. Satunnaismuuttuja X on tarvittavien heittojen määrä.

Satunnaismuuttujan arvon x todennäköisyys on

$$P(X = x) = (1-p)^{x-1}p,$$

missä $(1-p)^{x-1}$ kuvaa epäonnistuneita yrityksiä ja p on ensimmäinen onnistunut yritys.

Esimerkiksi jos heitetään noppaa, kunnes tulee silmäluku 6, todennäköisyys heittää tarkalleen 4 kertaa on $(5/6)^3 1/6$.

24.2.2 Odotusarvo

Odotusarvo $E[X]$ kertoo, mikä satunnaismuuttujan X arvo on keskimääräisessä tilanteessa. Odotusarvo lasketaan summana

$$\sum_x P(X = x)x,$$

missä x saa kaikki mahdolliset satunnaismuuttujan arvot.

Esimerkiksi yhden nopan heitossa silmäluvun odotusarvo on $1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2$.

Lineaarisuus

Usein hyödyllinen odotusarvon ominaisuus on lineaarisuus. Sen ansiosta summa $E[X_1 + X_2 + \cdots + X_n]$ voidaan laskea $E[X_1] + E[X_2] + \cdots + E[X_n]$. Kaava pätee myös silloin, kun satunnaismuuttujat riippuvat toisistaan.

Esimerkiksi kahden nopan heitossa silmälukujen summan odotusarvo on $E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7$.

Seuraava tehtävä on hieman vaativampi:

Tehtävä: Pöydällä on n hattua ja n palloa. Jokainen pallo sijoitetaan satunnaisesti johonkin hattuun. Mikä on odotusarvo, montako hattua on tyhjiä?

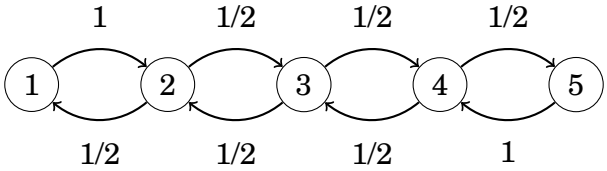
Todennäköisyys, että yksittäinen hattu on tyhjä, on $(\frac{n-1}{n})^n$, koska mikään pallo ei saa mennä sinne. Odotusarvon lineaarisuuden ansiosta tyhjien hattujen määrän odotusarvo on $n \cdot (\frac{n-1}{n})^n$.

24.3 Satunnaisprosessi

Satunnaisprosessi muodostuu tiloista, joiden välillä liikutaan kaaria pitkin. Jokaiseen kaareen liittyy todennäköisyys, joka tarkoittaa, miten todennäköisesti tilasta valitaan tämä kaari. Luonteva tapa esittää satunnaisprosessi on muodostaa verkko, jonka solmut ovat prosessin tiloja.

Tehtävä: Rakennuksessa on n kerrosta. Aloitat kerroksesta 1 ja liikut joka vuorolla satunnaisesti kerroksen ylöspäin tai alaspäin. Poikkeuksena kerroksesta 1 liikut aina ylöspäin ja kerroksesta n aina alaspäin. Mikä on todennäköisyys, että olet m vuoron jälkeen kerroksessa k?

Tehtävässä kukin rakennuksen kerros on yksi tiloista, ja kerrosten välillä liikutaan satunnaisesti. Esimerkiksi jos $n = 5$, verkosta tulee:



Satunnaisprosessin tila voidaan esittää taulukkona $[p_1, p_2, \ldots, p_n]$, missä p_k tarkoittaa todennäköisyyttä olla tällä hetkellä tilassa k. Todennäköisyyksille pätee aina $p_1 + p_2 + \cdots + p_n = 1$.

Esimerkissä tilana on ensin $[1,0,0,0,0]$, koska on varmaa, että kulku alkaa kerroksesta 1. Seuraava tila on $[0,1,0,0,0]$, koska kerroksesta 1 pääsee vain kerrokseen 2. Tämän jälkeen on mahdollisuus mennä joko ylöspäin tai alaspäin, joten seuraava tila on $[1/2,0,1/2,0,0]$ jne.

Tehokas tapa simuloida satunnaisprosessia on käyttää dynaamista ohjelmointia. Ideana on käydä joka vuorolla läpi kaikki tilat ja jokaisesta tilasta kaikki mahdollisuudet jatkaa eteenpäin. Tämän simuloinnin aikavaativuus on $O(n^2 m)$, missä n on tilojen määrä ja m on askelten määrä.

Tilasiirtymät voi esittää myös matriisina, mikä mahdollistaa tehokkaan matriisipotenssin käyttämisen. Tästä menetelmästä voi olla hyötyä, jos m on suuri, koska aikavaativuudeksi tulee $O(n^3 \log m)$.

24.4 Satunnaisalgoritmit

Joskus tehtävässä voi hyödyntää satunnaisuutta, vaikka tehtävä ei itsessään liittyisi todennäköisyyteen. Satunnaisalgoritmit ovat algoritmeja, joiden toiminta perustuu satunnaisuuteen.

Hyvä satunnaisalgoritmi tuottaa suurella todennäköisyydellä oikean vastauksen tehokkaasti. Satunnaisalgoritmin analysoinnissa tarvitsee todennäköisyyslaskentaa, jotta voi näyttää, että algoritmi toimii riittävän hyvin.

Tehtävä: Annettuna on taulukko, jossa on n lukua. Tiedät, että yli puolet taulukon luvuista on samaa lukua x, ja tehtäväsi on etsiä luku x.

Tehtävän ratkaisuun on olemassa yksinkertainen satunnaisalgoritmi: valitse taulukosta satunnainen luku ja tarkista, esiintyykö se yli $n/2$ kertaa. Jos esiintyy, x on löytynyt, ja muuten toista samaa uudestaan.

Koska ainakin puolet taulukon luvuista on lukua x, todennäköisyys, että valittu luku ei ole x, on alle 1/2. Niinpä k:n valinnan jälkeen todennäköisyys, että x on löytynyt, on ainakin $1-(1/2)^k$. Tämä lähestyy nopeasti lukua 1:

k	$1-(1/2)^k$
1	0,500000
2	0,750000
3	0,875000
4	0,937500
5	0,968750
...	
10	0,999023
...	
20	0,999999

Oikea x löytyy lähes varmasti muutaman valinnan jälkeen, joten algoritmin aikavaativuus on käytännössä $O(n)$.

Luku 25

Peliteoria

Peliteoria etsii tapoja pelata pelejä voitokkaasti. Tässä luvussa keskitymme kahden pelaajan peleihin, joissa pelaajat tekevät siirtoja vuorotellen. Osoittautuu, että monia tällaisia pelejä voi mallintaa nim-pelin avulla, johon on olemassa yksinkertainen voittostrategia.

25.1 Pelin tila

Pelin tila kertoo, missä vaiheessa peli on sillä hetkellä. Jokaiseen tilaan liittyy mahdollisia siirtoja, jotka johtavat toisiin tiloihin. Tavoitteena on usein laskea pelin tilasta, kumpi pelaaja voittaa, jos molemmat pelaavat optimaalisesti. Tutustumme seuraavaksi kahteen usein esiintyvään pelityyppiin.

25.1.1 Voitto ja häviö

Monissa peleissä pelaajat tekevät siirtoja vuorotellen, kunnes siirtoa ei pysty enää tekemään. Pelistä riippuen viimeisen siirron tekijä voittaa tai häviää pelin. Näin on esimerkiksi seuraavassa pelissä:

Tikkupeli: Pöydällä on n tikkua kasassa, ja pelaajat poistavat vuorotellen kasasta tikkuja. Pelaaja saa poistaa kerrallaan 1, 2 tai 3 tikkua. Pelin voittaa se, joka poistaa viimeisen tikun.

Tällaisessa pelissä jokainen pelin tila on joko voittotila tai häviötila. Voittotilassa oleva pelaaja pystyy voittamaan pelin, ja häviötilassa oleva pelaaja häviää pelin, jos vastustaja toimii optimaalisesti.

Tikkupelissä pelin tila on tikkujen määrä pöydällä. Tila 0 on häviötila, koska tikkuja ei voi poistaa. Tilat 1, 2 ja 3 ovat voittotiloja, koska niissä voi poistaa 1, 2 tai 3 tikkua, jolloin vastustaja häviää pelin. Tila 4 on taas häviötila, koska 1, 2 tai 3 tikun poistaminen johtaa tilaan, joka on vastustajalle voittotila.

Seuraava taulukko näyttää tikkupelin tilojen luokittelun, kun tikkuja on $0, 1, \ldots, 15$. Merkki V tarkoittaa voittotilaa ja merkki H tarkoittaa häviötilaa.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	V	V	V	H	V	V	V	H	V	V	V	H	V	V	V

Kuten taulukosta voi huomata, tikkupelissä on helppoa päätellä, onko tila voittotila vai häviötila. Jos kasassa on 4:llä jaollinen määrä tikkuja, tila on häviötila, ja muuten tila on voittotila.

Yleisessä tapauksessa pätee, että tila on voittotila, jos siitä pääsee jollakin siirrolla häviötilaan, ja tila on häviötila, jos kaikki mahdolliset siirrot vievät voittotiloihin. Tällä päättelyllä pystyy luokittelemaan minkä tahansa vastaavan pelin tilat voitto- ja häviötiloihin.

25.1.2 Pisteiden keräys

Joissakin peleissä pelaajat keräävät pisteitä pelin aikana ja tavoitteena on maksimoida tai minimoida pisteiden yhteismäärä. Tällöin pelien tilojen väliset siirrot tuottavat pisteitä. Näin on seuraavassa pelissä:

Poistopeli: Annettuna on taulukko, jossa on n lukua. Vuorossa oleva pelaaja poistaa taulukosta ensimmäisen tai viimeisen luvun, kunnes taulukko on tyhjä. Pelaajan tavoitteena on maksimoida poistettujen lukujen summa.

Tässä pelissä pelin tila on jäljellä oleva taulukon väli. Jokaiseen tilaan liittyy lisätietona suurin pistemäärä, jonka kyseisestä tilasta aloittava pelaaja voi saada. Esimerkiksi pelin tilan

4	5	1	3

suurin pistemäärä on 8. Aloittajan optimaalinen pelitapa on poistaa ensin luku 3. Sitten vastustaja poistaa luvun 1 tai 4, jonka jälkeen aloittaja saa poistettua luvun 5. Aloittaja saa siis yhteensä $3+5=8$ pistettä.

Poistopelin optimaalisen pelitavan saa selville dynaamisella ohjelmoinnilla, jossa jokaiselle taulukon välille lasketaan suurin mahdollinen aloittajan pistemäärä. Vastaavalla tavalla voi lähestyä muitakin pelejä, joissa kerätään pisteitä, jos tilojen määrä on riittävän pieni.

Huomaa, että ahne strategia, joka poistaa aina suuremman luvun, ei tuota välttämättä suurinta pistemäärää. Esimerkiksi yllä olevassa pelissä ahne strategia tuottaa pistemäärän 7, vaikka pistemäärä 8 on mahdollinen.

25.2 Nim-peli

Nim-peli on yksinkertainen peli, joka on tärkeässä asemassa peliteoriassa, koska monia pelejä voi pelata samalla strategialla kuin nim-peliä. Tutustumme aluksi nim-peliin ja yleistämme strategian sitten muihin peleihin.

Nim-peli: Pelissä on n kasaa tikkuja, joissa kussakin on tietty määrä tikkuja. Joka vuorolla pelaaja valitsee yhden epätyhjän kasan ja poistaa siitä minkä tahansa määrän tikkuja. Pelin voittaa se, joka poistaa viimeisen tikun.

Nim-pelin tila on muotoa $[x_1, x_2, \ldots, x_n]$, jossa x_k on tikkujen määrä kasassa k. Esimerkiksi $[10, 12, 5]$ tarkoittaa nim-peliä, jossa on kolme kasaa ja tikkujen määrät ovat 10, 12 ja 5. Tila $[0, 0, \ldots, 0]$ on häviötila, koska siitä ei voi poistaa mitään tikkua, ja peli päättyy aina siihen.

25.2.1 Analyysi

Osoittautuu, että nim-pelin tilan luonteen kertoo xor-summa $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, missä $\oplus$ tarkoittaa xor-operaatiota. Jos xor-summa on 0, tila on häviötila, ja muussa tapauksessa tila on voittotila. Esimerkiksi tilan $[10, 12, 5]$ xor-summa on $10 \oplus 12 \oplus 5 = 3$, joten tila on voittotila.

Mutta miten xor-summa liittyy nim-peliin? Tämä selviää tutkimalla, miten xor-summa muuttuu, kun nim-pelin tila muuttuu.

Häviötilat

Pelin päätöstila $[0, 0, \ldots, 0]$ on häviötila, ja sen xor-summa on 0, kuten kuuluukin. Muissa häviötiloissa mikä tahansa siirto johtaa voittotilaan, koska yksi luvuista x_k muuttuu ja samalla pelin xor-summa muuttuu eli siirron jälkeen xor-summasta tulee jokin muu kuin 0.

Voittotilat

Voittotilasta pääsee häviötilaan muuttamalla jonkin kasan k tikkujen määräksi $x_k \oplus s$, missä s on pelin xor-summa. Vaatimuksena on, että $x_k \oplus s < x_k$, koska kasasta voi vain poistaa tikkuja. Sopiva kasa x_k on jokin sellainen, jossa on ykkösbitti samassa kohdassa kuin s:n vasemmanpuoleisin ykkösbitti.

Esimerkki

Tila $[10, 12, 5]$ on voittotila, koska sen xor-summa on 3. Täytyy siis olla olemassa siirto, jolla tilasta pääsee häviötilaan. Selvitetään se seuraavaksi.

Pelin xor-summa muodostuu seuraavasti:

10	1010
12	1100
5	0101
3	0011

Tässä tapauksessa 10 tikun kasa on ainoa, jonka bittiesityksessä on ykkösbitti samassa kohdassa kuin xor-summan vasemmanpuoleisin ykkösbitti:

10	10(1)0
12	1100
5	0101
3	00(1)1

Kasan uudeksi sisällöksi täytyy saada $10 \oplus 3 = 9$ tikkua, mikä onnistuu poistamalla 1 tikku 10 tikun kasasta. Tämän jälkeen tilanne on:

9	1001
12	1100
5	0101
0	0000

25.2.2 Misääripeli

Misääripelissä nim-pelin tavoite on käänteinen, eli pelin häviää se, joka poistaa viimeisen tikun. Osoittautuu, että misääripeliä pystyy pelaamaan lähes samalla strategialla kuin tavallista nim-peliä.

Ideana on pelata misääripeliä aluksi kuin tavallista nim-peliä, mutta muuttaa strategiaa pelin lopussa. Käänne tapahtuu silloin, kun seuraavan siirron seurauksena kaikissa pelin kasoissa olisi 0 tai 1 tikkua.

Tavallisessa nim-pelissä tulisi nyt tehdä siirto, jonka jälkeen 1-tikkuisia kasoja on parillinen määrä. Misääripelissä tulee kuitenkin tehdä siirto, jonka jälkeen 1-tikkuisia kasoja on pariton määrä.

Tämä strategia toimii, koska käännekohta tulee aina vastaan jossakin vaiheessa peliä, ja kyseinen tila on voittotila, koska siinä on tarkalleen yksi kasa, jossa on yli 1 tikkua, joten xor-summa ei ole 0.

25.3 Muunnos nimiksi

Nim-pelin hienoutena on, että mikä tahansa peli, jossa kaksi pelaajaa tekee siirtoja vuorotellen ja viimeinen siirto ratkaisee voittajan, voidaan muuttaa nim-peliksi ja siinä pystyy käyttämään samaa strategiaa kuin nim-pelissä. Tämä tulos tunnetaan nimellä Sprague–Grundyn lause.

25.3.1 Grundy-luku

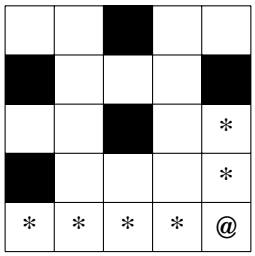
Pelin tilan Grundy-luku on pienin ei-negatiivinen kokonaisluku, joka ei ole minkään sellaisen tilan Grundy-luku, johon tilasta pääsee yhdellä siirrolla. Jos tilasta ei pääse mihinkään tilaan, sen Grundy-luku on 0.

Grundy-luku vastaa tikkujen määrää nim-kasassa. Jos Grundy-luku on 0, tilasta pääsee vain tiloihin, joiden Grundy-luku ei ole 0. Jos taas Grundy-luku on $x > 0$, siitä pääsee tiloihin, joiden Grundy-luku on $0, 1, \ldots, x-1$.

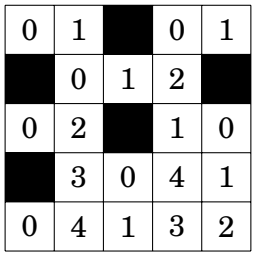
Esimerkki

Sokkelopeli: Sokkelossa on pelihahmo, jota pelaajat siirtävät vuorotellen. Jokainen sokkelon ruutu on lattiaa tai seinää. Kullakin siirrolla hahmon tulee liikkua jokin määrä askeleita vasemmalle tai jokin määrä askeleita ylöspäin. Pelin voittaja on se, joka tekee viimeisen siirron.

Esimerkiksi seuraavassa on pelin mahdollinen aloitustilanne, jossa @ on pelihahmo ja * merkitsee ruutua, johon hahmo voi siirtyä.



Sokkelopelin tiloja ovat kaikki sokkelon lattiaruudut. Äskeisessä esimerkissä tilojen Grundy-luvut ovat seuraavat:



Tämän muunnoksen jälkeen sokkelopelin tila käyttäytyy samalla tavalla kuin nim-pelin kasa. Huomaa, että toisin kuin nim-pelissä, tilasta saattaa päästä toiseen tilaan, jonka Grundy-luku on suurempi. Tällaisen siirron voi kuitenkin aina peruuttaa niin, että Grundy-luku palautuu samaksi.

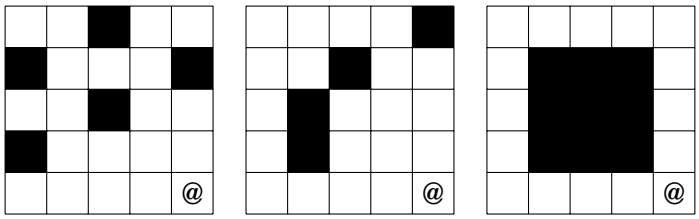
25.3.2 Alipelit

Oletetaan seuraavaksi, että peli muodostuu alipeleistä ja jokaisella vuorolla pelaaja valitsee jonkin alipeleistä ja tekee siirron siinä. Peli päättyy, kun missään alipelissä ei pysty tekemään siirtoa.

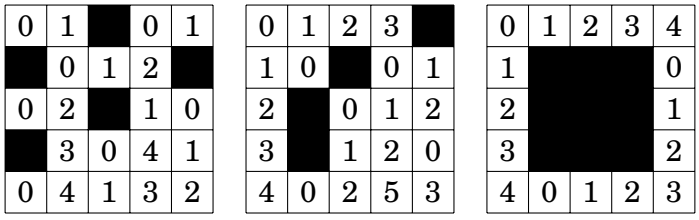
Nyt pelin tilan Grundy-luku on alipelien Grundy-lukujen xor-summa. Peliä pystyy pelaamaan nim-pelin tapaan selvittämällä kaikkien alipelien Grundy-luvut ja laskemalla niiden xor-summa.

Esimerkki

Kolmen sokkelon pelissä joka siirrolla pelaaja valitsee yhden sokkeloista ja siirtää siinä olevaa hahmoa. Pelin aloitustilanne voi olla seuraavanlainen:



Sokkeloiden ruutujen Grundy-luvut ovat:



Aloitustilanteessa Grundy-lukujen xor-summa on $2 \oplus 3 \oplus 3 = 2$, joten aloittaja pystyy voittamaan pelin. Sopiva aloitussiirto on liikkua vasemmassa sokkelossa 2 askelta ylöspäin, jolloin xor-summaksi tulee $0 \oplus 3 \oplus 3 = 0$.

25.3.3 Jakautuminen

Joskus pelissä on joukko alipelejä, joista jokainen voi jakautua uusiksi alipeleiksi. Kuten ennenkin, jokainen alipeli vastaa nim-pelin kasaa ja alipelien joukon Grundy-luku saadaan laskemalla xor-summa alipelien Grundy-luvuista.

Alipelin Grundy-luku selviää käymällä läpi kaikki tavat, miten alipeli voi jakautua. Jokainen jakotapa luo alipelien joukon, jonka Grundy-luvun saa laskettua xor-summalla. Alipelin Grundy-luku on pienin luku, joka ei ole mikään näistä xor-summista. Sama jatkuu rekursiivisesti pienempiin alipeleihin.

Esimerkki

Tarkastellaan lopuksi seuraavaa peliä:

Bittipeli: Annettuna on joukko bittijonoja. Joka vuorolla pelaaja valitsee jonkin bittijonon ja jakaa sen kahteen osaan niin, että kumpaankin osaan jää sekä bitti 0 että bitti 1. Pelin voittaja on se, joka tekee viimeisen siirron.

Esimerkiksi pelissä $\{1101001\}$ on kolme mahdollista siirtoa, jotka tuottavat alipelit $\{110,1001\}$, $\{1101,001\}$ ja $\{11010,01\}$. Pelin Grundy-luku on pienin kokonaisluku, joka ei ole minkään alipelin Grundy-luku.

Alipelin $\{110,1001\}$ Grundy-luku on alipelien $\{110\}$ ja $\{1001\}$ Grundy-lukujen xor-summa. Alipelin $\{110\}$ Grundy-luku on 0, koska mitään siirtoa ei voi tehdä. Alipelin $\{1001\}$ Grundy-luku on 1, koska ainoa siirto luo alipelin $\{10,01\}$, jonka Grundy-luku on 0. Niinpä alipelin $\{110,1001\}$ Grundy-luku on $0 \oplus 1 = 1$.

Vastaavasti saadaan, että alipelien $\{1101,001\}$ ja $\{11010,01\}$ Grundy-luvut ovat 0 ja 1. Pelin $\{1101001\}$ Grundy-luku on siis 2, koska siinä on kolme mahdollista siirtoa, jotka tuottavat alipelit, joiden Grundy-luvut ovat 1, 0 ja 1.

Tässä tapauksessa aloittaja voittaa jakamalla bittijonon $\{1101,001\}$. Tämän jälkeen vastustaja ei voi tehdä mitään siirtoa ja peli on päättynyt.

Luku 26

Merkkijonot

Tässä luvussa tutustumme tehokkaisiin perusmenetelmiin merkkijonojen käsittelyyn. Trie on puurakenne, joka pitää yllä merkkijonojoukkoa. Merkkijonohajautus ja Z-algoritmi ovat monipuolisia algoritmeja, joiden avulla voi ratkaista tehokkaasti monia merkkijonotehtäviä.

26.1 Määritelmiä

Merkkijonon s merkit ovat $s_1, s_2, \ldots, s_n$, missä n on merkkijonon pituus. Aakkosto (*alphabet*) määrittää, mitä merkkejä merkkijonossa voi esiintyä. Esimerkiksi aakkosto {A,B,...,Z} sisältää englannin kielen suuret kirjaimet.

Osajono (*substring*) on merkkijonon yhtenäinen osa. Esimerkiksi merkkijonon ABC osajonot ovat A, B, C, AB, BC ja ABC. Alijono (*subsequence*) sisältää osan merkkijonon merkeistä niiden alkuperäisessä järjestyksessä. Esimerkiksi merkkijonon ABC alijonot ovat A, B, C, AB, AC, BC ja ABC.

Alkuosa (*prefix*) on merkkijonon alusta alkava osajono, ja loppuosa (*suffix*) on merkkijonon loppuun päättyvä osajono. Esimerkiksi merkkijonon ABC alkuosat ovat A, AB ja ABC ja loppuosat ovat ABC, BC ja C. Alkuosa tai loppuosa on aito (*proper*), jos se ei ole sama kuin koko merkkijono.

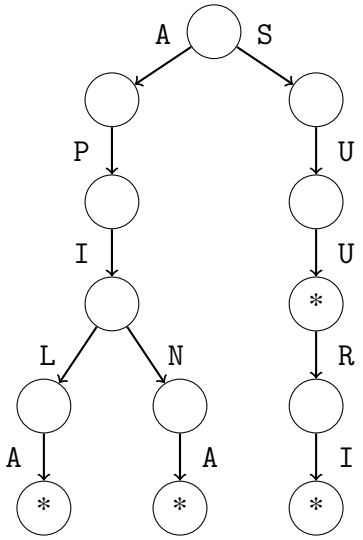
Kierto (*rotation*) syntyy, kun merkkejä siirretään yksi kerrallaan alusta loppuun. Esimerkiksi merkkijonon ABC kierrot ovat ABC, BCA ja CAB. Jakso (*period*) on alkuosa, jota toistamalla merkkijono muodostuu. Esimerkiksi merkkijonon ABCABCA lyhin jakso on ABC. Reuna (*border*) on merkkijono, joka on sekä aito alkuosa että loppuosa. Esimerkiksi merkkijonon ABADABA pisin reuna on ABA.

Merkkijonojen vertailussa käytössä on yleensä leksikografinen järjestys, joka vastaa aakkosjärjestystä. Siinä $x < y$, jos joko x on y:n aito alkuosa tai on olemassa kohta k niin, että $x_i = y_i$, kun $i < k$, ja $x_k < y_k$.

26.2 Trie-rakenne

Trie on puurakenne, joka pitää yllä merkkijonojoukkoa. Merkkijonot tallennetaan puuhun juuresta lähtevinä merkkien ketjuina. Jos useammalla merkkijonolla on sama alkuosa, niiden ketjun alkuosa on yhteinen.

Esimerkiksi joukkoa {APILA,APINA,SUU,SUURI} vastaa seuraava trie:



Merkki * solmussa tarkoittaa, että jokin merkkijono päättyy kyseiseen solmuun. Tämä merkki on tarpeen, koska merkkijono voi olla toisen merkkijonon alkuosa, kuten tässä puussa merkkijonot SUU ja SUURI.

Triessä merkkijonon lisäyksen ja hakemisen aikavaativuus on $O(n)$, kun n on merkkijonon pituus. Molemmissa operaatioissa ideana on lähteä liikkeelle juuresta ja kulkea alaspäin merkkijonon merkkien mukaisesti. Lisäyksessä puuhun lisätään tarvittaessa uusia solmuja.

Triestä on mahdollista etsiä sekä merkkijonoja että merkkijonojen alkuosia. Lisäksi puun solmuissa voi pitää kirjaa, monessako merkkijonossa on solmua vastaava alkuosa, mikä lisää trien käyttömahdollisuuksia.

26.3 Merkkijonohajautus

Merkkijonohajautus (*string hashing*) on tekniikka, jonka avulla voi esikäsittelyn jälkeen tarkistaa tehokkaasti, ovatko kaksi merkkijonoa samat. Ideana on verrata toisiinsa merkkijonojen hajautusarvoja, mikä on tehokkaampaa kuin merkkijonojen vertaaminen merkki kerrallaan.

Hajautusarvo

Merkkijonon hajautusarvo (*hash value*) on luku, joka lasketaan merkkijonon merkeistä etukäteen valitulla tavalla. Jos kaksi merkkijonoa ovat samat, myös niiden hajautusarvot ovat samat, minkä ansiosta merkkijonoja voi vertailla niiden hajautusarvojen kautta.

Tavallinen tapa toteuttaa merkkijonohajautus on käyttää polynomista hajautusta. Siinä hajautusarvo lasketaan kaavalla

$$(c_1 A^{n-1} + c_2 A^{n-2} + \cdots + c_n A^0) \bmod B,$$

missä merkkijonon merkkien koodit ovat $c_1, c_2, \ldots, c_n$ ja A ja B ovat etukäteen valitut vakiot.

Esimerkiksi merkkijonon KISSA merkkien koodit ovat:

K	I	S	S	A
75	73	83	83	65

Jos $A = 3$ ja $B = 97$, merkkijonon KISSA hajautusarvoksi tulee

$$(75 \cdot 3^4 + 73 \cdot 3^3 + 83 \cdot 3^2 + 83 \cdot 3^1 + 65 \cdot 3^0) \bmod 97 = 59.$$

Esikäsittely

Polynomisessa hajautuksessa voi esikäsittelyn jälkeen laskea minkä tahansa merkkijonon osajonon hajautusarvon $O(1)$-ajassa. Ideana on muodostaa kaksi taulukkoa: taulukko s kertoo jokaisen merkkijonon alkuosan hajautusarvon ja taulukko p sisältää lukuja muotoa $A^k \bmod B$.

Taulukko s lasketaan rekursiolla

$$\begin{aligned} s_0 &= 0 \\ s_k &= (s_{k-1}A + c_k) \bmod B \end{aligned}$$

ja taulukko p lasketaan rekursiolla

$$\begin{aligned} p_0 &= 1 \\ p_k &= (p_{k-1}A) \bmod B. \end{aligned}$$

Tämän jälkeen funktio

$$h(a,b) = (s_b - s_{a-1}p_{b-a+1}) \bmod B$$

laskee hajautusarvon osajonolle, joka alkaa kohdasta a ja päättyy kohtaan b.

Hajautuksen käyttö

Hajautusarvot tarjoavat nopean tavan merkkijonojen vertailuun. Ideana on vertailla merkkijonojen koko sisällön sijaan niiden hajautusarvoja. Jos hajautusarvot ovat samat, myös merkkijonot ovat *todennäköisesti* samat, ja jos taas hajautusarvot eivät ole samat, merkkijonot eivät ole samat.

Hajautuksen avulla pystyy usein tehostamaan suoraviivaista algoritmia niin, että siitä tulee tehokas. Näin on esimerkiksi seuraavassa tehtävässä:

> *Tehtävä:* Annettuna on merkkijono t, jossa on n merkkiä, sekä merkkijono p, jossa on m merkkiä ($m \leq n$). Tehtäväsi on selvittää, montako kertaa merkkijono p esiintyy merkkijonon t osajonona.

Suoraviivainen algoritmi tehtävään käy läpi kaikki mahdolliset kohdat, joissa p voi esiintyä t:n osajonona. Mahdollisia kohtia on $O(n)$ ja yksi vertailu vie aikaa $O(m)$, joten aikavaativuus on $O(nm)$. Hajautuksen avulla kuitenkin vertailu vie aikaa vain $O(1)$, jolloin algoritmin aikavaativuudeksi tulee $O(n)$.

Hajautuksen avulla voi myös tutkia merkkijonojen suuruusjärjestystä:

> *Tehtävä:* Annettuna on merkkijono, jossa on n merkkiä, sekä kokonaisluku $m \leq n$. Tehtäväsi on etsiä merkkijonon aakkosjärjestyksessä ensimmäinen osajono, jonka pituus on m.

Suoraviivainen $O(nm)$-aikainen algoritmi käy läpi kaikki m-pituiset osajonot ja pitää muistissa pienintä osajonoa. Hajautuksen avulla kahden osajonon suuruusjärjestyksen voi selvittää logaritmisessa ajassa etsimällä ensin binäärihaulla osajonojen yhteisen alkuosan pituuden ja vertaamalla sitten alkuosan jälkeistä merkkiä. Aikavaativuudeksi tulee $O(n \log m)$.

Törmäykset ja parametrit

Ilmeinen riski hajautusarvojen vertailussa on *törmäys*, mikä tarkoittaa, että kahdessa merkkijonossa on eri sisältö mutta niiden hajautusarvot ovat samat. Tällöin hajautusarvojen perusteella merkkijonot näyttävät samalta, vaikka todellisuudessa ne eivät ole samat, ja algoritmi voi toimia väärin.

Törmäyksen riski on aina olemassa, koska erilaisia merkkijonoja on enemmän kuin erilaisia hajautusarvoja, mutta riskin saa pieneksi valitsemalla vakiot A ja B huolellisesti. Vakioiden valinnassa on kaksi tavoitetta: hajautusarvojen tulisi jakautua tasaisesti merkkijonoille ja erilaisten hajautusarvojen määrän tulisi olla riittävän suuri.

Hyvä ratkaisu on valita vakioiksi satunnaisia suuria alkulukuja. Tyypillinen tapa on valita vakiot läheltä lukua 10^9, esimerkiksi

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Tällainen valinta takaa käytännössä sen, että hajautusarvot jakautuvat riittävän tasaisesti välille $0 \ldots B-1$. Suuruusluokan 10^9 etuna on, että `long long` -tyyppi riittää hajautusarvojen käsittelyyn koodissa, koska tulot AB ja BB mahtuvat `long long` -tyyppiin. Mutta onko 10^9 riittävä määrä hajautusarvoja?

Tarkastellaan nyt kolmea hajautuksen käyttötapaa:

Tapaus 1: Merkkijonoja x ja y verrataan toisiinsa. Törmäyksen todennäköisyys on $1/B$ olettaen, että kaikki hajautusarvot esiintyvät yhtä usein.

Tapaus 2: Merkkijonoa x verrataan merkkijonoihin $y_1, y_2, \ldots, y_n$. Yhden tai useamman törmäyksen todennäköisyys on $1-1(1-1/B)^n$.

Tapaus 3: Merkkijonoja $x_1, x_2, \ldots, x_n$ verrataan kaikkia keskenään. Yhden tai useamman törmäyksen todennäköisyys on

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Seuraava taulukko sisältää törmäyksen todennäköisyydet, kun vakion B arvo vaihtelee ja $n = 10^6$:

vakio B	tapaus 1	tapaus 2	tapaus 3
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

Taulukosta näkee, että valinta $B \approx 10^9$ riittää tapauksissa 1 ja 2, koska törmäyksen riski on vähäinen. Sen sijaan tapauksessa 3 tilanne on toinen: törmäys tapahtuu lähes varmasti vielä valinnalla $B \approx 10^9$.

Kätevä tapa pienentää törmäyksen riskiä on laskea monta hajautusarvoa eri parametreilla ja vertailla niitä kaikkia. On hyvin pieni todennäköisyys, että törmäys tapahtuisi samaan aikaan kaikissa hajautusarvoissa. Esimerkiksi kaksi hajautusarvoa parametrilla $B \approx 10^9$ vastaa yhtä hajautusarvoa parametrilla $B \approx 10^{18}$, mikä takaa hyvän suojan törmäyksiltä.

Jotkut käyttävät hajautuksessa vakioita $B = 2^{32}$ tai $B = 2^{64}$, jolloin modulo B tulee laskettua automaattisesti, kun muuttujan arvo pyörähtää ympäri. Tämä ei ole kuitenkaan hyvä valinta, koska muotoa 2^x olevaa moduloa vastaan pystyy tekemään testisyötteen, joka aiheuttaa törmäyksen.

26.4 Z-algoritmi

Usein vaihtoehtoinen tekniikka merkkijonohajautukselle on Z-algoritmi, joka laskee jokaiselle merkkijonon kohdalle, mikä on pisin kyseisestä kohdasta alkava osajono, joka on myös merkkijonon alkuosa.

Toisin kuin merkkijonohajautus, Z-algoritmi toimii varmasti oikein eikä siinä ole törmäysten riskiä. Toisaalta Z-algoritmi on vaikeampi toteuttaa eikä se sovellu kaikkeen samaan kuin hajautus.

Toiminta

Z-algoritmi muodostaa merkkijonolle Z-taulukon, jonka jokaisessa kohdassa lukee, kuinka pitkälle kohdasta alkava osajono vastaa merkkijonon alkuosaa. Esimerkiksi Z-taulukko merkkijonolle `ACBACDACBACBACDA` on seuraava:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Esimerkiksi kohdassa 7 on arvo 5, koska siitä alkava 5-merkkinen osajono `ACBAC` on merkkijonon alkuosa, mutta 6-merkkinen osajono `ACBACB` ei ole enää merkkijonon alkuosa.

Z-algoritmi käy läpi merkkijonon vasemmalta oikealle ja laskee jokaisessa kohdassa, kuinka pitkälle kyseisestä kohdasta alkava osajono täsmää merkkijonon alkuun. Algoritmi laskee yhteisen alkuosan pituuden vertaamalla merkkijonon alkua ja osajonon alkua toisiinsa.

Suoraviivaisesti toteutettuna tällaisen algoritmin aikavaativuus olisi $O(n^2)$, koska yhteiset alkuosat voivat olla pitkiä, mutta Z-algoritmissa on yksi tärkeä optimointi, jonka ansiosta algoritmin aikavaativuus on vain $O(n)$.

Ideana on pitää muistissa väliä $[x,y]$, joka on aiemmin laskettu merkkijonon alkuun täsmäävä väli, jossa y on mahdollisimman suuri. Tällä välillä olevia merkkejä ei tarvitse koskaan verrata uudestaan merkkijonon alkuun, vaan niitä koskevan tiedon saa suoraan Z-taulukon lasketusta osasta.

Z-algoritmin aikavaativuus on $O(n)$, koska algoritmi aloittaa merkki kerrallaan vertailun aina kohdasta $y+1$. Jos merkit täsmäävät, kohta y siirtyy eteenpäin eikä algoritmin tarvitse enää koskaan vertailla tätä kohtaa, vaan se pystyy hyödyntämään Z-taulukon alussa olevaa tietoa.

Esimerkki

Katsotaan nyt, miten Z-algoritmi muodostaa seuraavan Z-taulukon:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

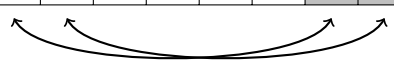
Ensimmäinen mielenkiintoinen kohta tulee, kun yhteisen alkuosan pituus on 5. Silloin algoritmi laittaa muistiin välin $[7,11]$ seuraavasti:

						x				y					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?

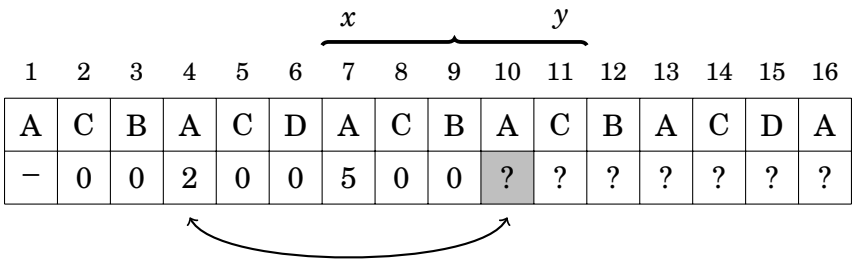
Välin $[7,11]$ hyötynä on, että algoritmi voi sen avulla laskea seuraavat Z-taulukon arvot nopeammin. Koska välin $[7,11]$ merkit ovat samat kuin merkkijonon alussa, myös Z-taulukon arvoissa on vastaavuutta.

Ensinnäkin kohdissa 8 ja 9 tulee olla samat arvot kuin kohdissa 2 ja 3, koska väli $[7,11]$ vastaa väliä $[1,5]$:

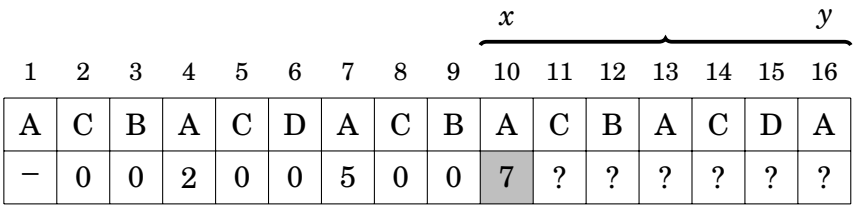
						x				y					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



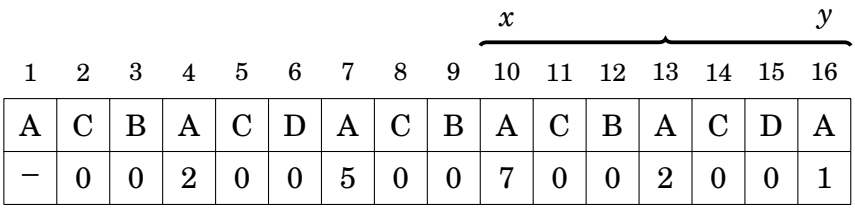
Seuraavaksi kohdasta 4 saa tietoa kohdan 10 arvon laskemiseksi. Koska kohdassa 4 on arvo 2, tämä tarkoittaa, että osajono täsmää kohtaan $y = 11$ asti, mutta sen jälkeen on tutkimatonta aluetta merkkijonossa.



Nyt algoritmi alkaa vertailla merkkejä kohdasta $y+1$ alkaen merkki kerrallaan. Algoritmi ei voi hyödyntää aiempaa tietoa Z-taulukossa, koska se ei ole vielä aiemmin tutkinut merkkijonoa näin pitkälle. Tuloksena osajonon pituudeksi tulee 7 ja väli $[x, y]$ päivittyy vastaavasti:



Tämän jälkeen kaikkien seuraavien Z-taulukon arvojen laskemisessa pystyy hyödyntämään jälleen välin $[x, y]$ antamaa tietoa ja algoritmi saa Z-taulukon loppuun tulevat arvot suoraan Z-taulukon alusta:



Z-taulukon käyttö

Z-taulukossa olevan tiedon avulla pystyy ratkaisemaan tehokkaasti monia merkkijonotehtäviä. Ratkaistaan esimerkkinä seuraava tehtävä:

> *Tehtävä:* Annettuna on merkkijono t, jossa on n merkkiä, sekä merkkijono p, jossa on m merkkiä ($m \leq n$). Tehtäväsi on selvittää, montako kertaa merkkijono p esiintyy merkkijonon t osajonona.

Ideana on luoda uusi merkkijono muotoa $p\#t$, jossa merkkijonojen p ja t välissä on erikoismerkki #, jota ei esiinny merkkijonoissa. Tämän merkkijonon Z-taulukko ilmaisee kohdat, joissa p esiintyy t:ssä, koska tarkalleen niissä kohdissa taulukossa on luku m. Algoritmin aikavaativuus on $O(n)$.

Toteutus

Seuraavassa on lyhyt Z-algoritmin toteutus, joka palauttaa Z-taulukon vektorina. Huomaa että toisin kuin Z-algoritmin käsittelyssä, tässä merkkijonossa ja taulukossa on 0-indeksointi.

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0,min(z[i-x],y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```

Luku 27

Neliöjuurialgoritmit

Neliöjuurialgoritmi on algoritmi, jonka aikavaativuudessa esiintyy neliöjuuri. Neliöjuurta voi ajatella ”köyhän miehen logaritmina”: aikavaativuus $O(\sqrt{n})$ on parempi kuin $O(n)$ mutta huonompi kuin $O(\log n)$. Toisaalta neliöjuurialgoritmit toimivat käytännössä hyvin ja niiden vakiokertoimet ovat pieniä.

27.1 Summakysely

Aloitamme tutusta ongelmasta, jossa toteutettavana on kaksi operaatiota lukutaulukkoon: alkion muuttaminen ja välin summan laskeminen. Olemme aiemmin ratkaisseet tämän tehtävän segmenttipuulla, mutta vaihtoehtoinen lähestymistapa tehtävään on käyttää neliöjuurirakennetta.

Ideana on jakaa taulukko $\sqrt{n}$-kokoisiin väleihin niin, että jokaiseen väliin tallennetaan lukujen summa välillä. Seuraavassa on esimerkki taulukosta ja sitä vastaavista $\sqrt{n}$-väleistä:

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Kun taulukon luku muuttuu, tämän yhteydessä täytyy laskea uusi summa vastaavalle $\sqrt{n}$-välille:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Välin summan laskeminen taas tapahtuu muodostamalla summa reunoissa olevista yksittäisistä luvuista sekä keskellä olevista $\sqrt{n}$-väleistä:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Luvun muuttamisen aikavaativuus on $O(1)$, koska riittää muuttaa yhden $\sqrt{n}$-välin summaa. Välin summa taas lasketaan kolmessa osassa:

- vasemmassa reunassa on $O(\sqrt{n})$ yksittäistä lukua
- keskellä on $O(\sqrt{n})$ peräkkäistä $\sqrt{n}$-väliä
- oikeassa reunassa on $O(\sqrt{n})$ yksittäistä lukua

Jokaisen osan summan laskeminen vie aikaa $O(\sqrt{n})$, joten koko summan laskemisen aikavaativuus on $O(\sqrt{n})$.

Vertailun vuoksi segmenttipuuta käyttäen sekä luvun muuttaminen että välin summan laskenta vievät aikaa $O(\log n)$. Neliöjuurirakenne mahdollistaa siis luvun muuttamisen segmenttipuuta nopeammin, mutta summan laskemiseen kuluu enemmän aikaa.

Neliöjuurialgoritmeissa parametri $\sqrt{n}$ johtuu siitä, että se saattaa kaksi asiaa tasapainoon. Käytännössä algoritmeissa ei ole kuitenkaan pakko käyttää tarkalleen parametria $\sqrt{n}$. Voi olla parempi ratkaisu valita toiseksi parametriksi k ja toiseksi n/k, missä k on pienempi tai suurempi kuin $\sqrt{n}$.

Paras parametri selviää usein kokeilemalla ja riippuu tehtävästä ja syötteestä. Esimerkiksi jos taulukkoa käsittelevä algoritmi käy usein läpi välit mutta harvoin välin sisällä olevia alkioita, taulukko voi olla järkevää jakaa $k < \sqrt{n}$ väliin, joista jokaisella on $n/k > \sqrt{n}$ alkiota.

27.2 Eräkäsittely

Eräkäsittelyssä algoritmin operaatiot jaetaan eriin, jotka käsitellään omina kokonaisuuksina. Erien välissä tehdään yksittäinen työläs toimenpide, joka auttaa tulevien operaatioiden käsittelyä.

Neliöjuurialgoritmi syntyy, kun n operaatiota jaetaan $O(\sqrt{n})$-kokoisiin eriin, jolloin sekä eriä että operaatioita kunkin erän sisällä on $O(\sqrt{n})$. Tämä tasapainottaa sitä, miten usein erien välinen työläs toimenpide tapahtuu sekä miten paljon työtä erän sisällä täytyy tehdä.

Eräkäsittelyä voi käyttää esimerkiksi seuraavassa tehtävässä:

> *Tehtävä:* Ruudukossa on $k \times k$ ruutua, jotka kaikki ovat aluksi valkoisia. Tehtäväsi on suorittaa n operaatiota ruudukkoon. Tyypin 1 operaatio värittää ruudun (y,x) mustaksi, ja tyypin 2 operaatio etsii ruudusta (y,x) lähimmän mustan ruudun. Ruutujen (y_1,x_1) ja (y_2,x_2) etäisyys on $|y_1-y_2|+|x_1-x_2|$.

Ratkaisuna on jakaa operaatiot $O(\sqrt{n})$ erään, joista jokaisessa on $O(\sqrt{n})$ operaatiota. Kunkin erän alussa jokaiseen ruudukon ruutuun lasketaan pienin etäisyys mustaan ruutuun. Tämä onnistuu ajassa $O(k^2)$ leveyshaun avulla.

Kunkin erän käsittelyssä pidetään yllä listaa ruuduista, jotka on muutettu mustaksi tässä erässä. Nyt etäisyys ruudusta lähimpään mustaan ruutuun on

joko erän alussa laskettu etäisyys tai sitten etäisyys johonkin listassa olevaan tämän erän aikana mustaksi muutettuun ruutuun.

Algoritmi vie aikaa $O((k^2+n)\sqrt{n})$, koska erien välissä tehdään $O(\sqrt{n})$ kertaa $O(k^2)$-aikainen läpikäynti, ja erissä käsitellään yhteensä $O(n)$ solmua, joista jokaisen kohdalla käydään läpi $O(\sqrt{n})$ solmua listasta.

Jos algoritmi tekisi leveyshaun jokaiselle operaatiolle, aikavaativuus olisi $O(k^2 n)$. Jos taas algoritmi kävisi kaikki muutetut ruudut läpi jokaisen operaation kohdalla, aikavaativuus olisi $O(n^2)$. Neliöjuurialgoritmi yhdistää nämä aikavaativuudet ja muuttaa kertoimen n kertoimeksi $\sqrt{n}$.

27.3 Tapauskäsittely

Tapauskäsittelyssä algoritmissa on useampia toimintatapoja, jotka aktivoituvat syötteestä riippuen. Tyypillisesti yksi algoritmin osa on tehokas pienellä parametrilla ja toinen osa on tehokas pienellä parametrilla, ja sopiva jakokohta kulkee suunnilleen arvon $\sqrt{n}$ kohdalla.

Tapauskäsittelyä voi käyttää esimerkiksi seuraavassa tehtävässä:

> *Tehtävä:* Puussa on n solmua, joista jokaisella on tietty väri. Solmujen värit on numeroitu $1,2,\ldots,m$. Tehtäväsi on etsiä puusta kaksi solmua, jotka ovat samanvärisiä ja mahdollisimman kaukana toisistaan.

Ideana on käydä läpi värit yksi kerrallaan ja etsiä kullekin värille kaksi solmua, jotka ovat mahdollisimman kaukana toisistaan. Värillä c algoritmin toiminta riippuu siitä, miten paljon puussa on c-värisiä solmuja.

Tapaus 1: $c \leq \sqrt{n}$

Jos c-värisiä solmuja on vähän, käydään läpi kaikki c-väristen solmujen parit ja valitaan pari, jonka etäisyys on suurin. Jokaisesta solmusta täytyy laskea etäisyys $O(\sqrt{n})$ muuhun solmuun, joten kaikkien tapaukseen 1 osuvien solmujen käsittely vie aikaa yhteensä $O(n\sqrt{n})$.

Tapaus 2: $c > \sqrt{n}$

Jos c-värisiä solmuja on paljon, käydään koko puu läpi ja lasketaan suurin etäisyys kahden c-värisen solmun välillä. Läpikäynnin aikavaativuus on $O(n)$, ja tapaus 2 aktivoituu korkeintaan $O(\sqrt{n})$ värille, joten tapauksen 2 solmut tuottavat aikavaativuuden $O(n\sqrt{n})$.

Algoritmin kokonaisaikavaativuus on $O(n\sqrt{n})$, koska sekä tapaus 1 että tapaus 2 vievät aikaa yhteensä $O(n\sqrt{n})$.

27.4 Mo'n algoritmi

Mo'n algoritmi soveltuu tehtäviin, joissa taulukkoon tehdään välikyselyitä ja taulukon sisältö kaikissa kyselyissä on sama. Algoritmi järjestää kyselyt uudestaan niin, että niiden käsittely on tehokasta.

Algoritmi pitää yllä taulukon väliä, jolle on laskettu kyselyn vastaus. Kyselystä toiseen siirryttäessä algoritmi muuttaa väliä askel kerrallaan niin, että vastaus uuteen kyselyyn saadaan laskettua. Algoritmin aikavaativuus on $O(n\sqrt{n}f(n))$, kun kyselyitä on n ja yksi välin muutosaskel vie aikaa $f(n)$.

Algoritmin toiminta perustuu järjestykseen, jossa kyselyt käsitellään. Kun kyselyjen välit ovat muotoa $[a,b]$, algoritmi järjestää ne ensisijaisesti arvon $\lfloor a/\sqrt{n} \rfloor$ mukaan ja toissijaisesti arvon b mukaan. Algoritmi suorittaa siis peräkkäin kaikki kyselyt, joiden alkukohta on tietyllä $\sqrt{n}$-välillä.

Osoittautuu, että tämän järjestyksen ansiosta algoritmi tekee yhteensä vain $O(n\sqrt{n})$ muutosaskelta. Tämä johtuu siitä, että välin vasen reuna liikkuu n kertaa $O(\sqrt{n})$ askelta, kun taas välin oikea reuna liikkuu $\sqrt{n}$ kertaa $O(n)$ askelta. Molemmat reunat liikkuvat siis yhteensä $O(n\sqrt{n})$ askelta.

Esimerkki

> *Tehtävä:* Annettuna on joukko välejä taulukossa. Tehtävänä on selvittää kullekin välille, montako eri lukua taulukossa on kyseisellä välillä.

Mo'n algoritmissa kyselyt järjestetään aina samalla tavalla, ja tehtävästä riippuva osa on, miten kyselyn vastausta pidetään yllä. Tässä tehtävässä luonteva tapa on pitää muistissa kyselyn vastausta sekä taulukkoa c, jossa c[x] on alkion x lukumäärä aktiivisella välillä.

Kyselystä toiseen siirryttäessä taulukon aktiivinen väli muuttuu. Esimerkiksi jos nykyinen kysely koskee väliä

ja seuraava kysely koskee väliä

niin tapahtuu kolme muutosaskelta: välin vasen reuna siirtyy askeleen oikealle ja välin oikea reuna siirtyy kaksi askelta oikealle.

Jokaisen muutosaskeleen jälkeen täytyy päivittää taulukkoa c. Jos väliin tulee alkio x, arvo c[x] kasvaa 1:llä, ja jos välistä poistuu alkio x, arvo c[x] vähenee 1:llä. Jos lisäyksen jälkeen c[x] = 1, kyselyn vastaus kasvaa 1:llä, ja jos poiston jälkeen c[x] = 0, kyselyn vastaus vähenee 1:llä.

Tässä tapauksessa muutosaskeleen aikavaativuus on $O(1)$, joten algoritmin kokonaisaikavaativuus on $O(n\sqrt{n})$.

Luku 28

Lisää segmenttipuusta

Segmenttipuu on tehokas tietorakenne, joka mahdollistaa monenlaisten kyselyiden toteuttamisen tehokkaasti. Tähän mennessä olemme käyttäneet kuitenkin segmenttipuuta melko rajoittuneesti. Nyt on aika tutustua pintaa syvemmältä segmenttipuun mahdollisuuksiin.

28.1 Kulku puussa

Tähän mennessä olemme kulkeneet segmenttipuuta alhaalta ylöspäin lehdistä juureen. Vaihtoehtoinen tapa toteuttaa puun käsittely on kulkea ylhäältä alaspäin juuresta lehtiin. Tämä kulkusuunta on usein kätevä silloin, kun kyseessä on perustilannetta monimutkaisempi segmenttipuu.

Esimerkiksi välin $[a,b]$ summan laskeminen segmenttipuussa tapahtuu alhaalta ylöspäin tuttuun tapaan näin (luku 9.2):

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

Ylhäältä alaspäin toteutettuna funktiosta tulee:

```
int summa(int a, int b, int k, int c, int d) {
    if (a > d || b < c) return 0;
    if (a == c && b == d) return p[k];
    int w = d-c+1;
    return summa(a, min(b,c+w/2-1), 2*k, c, c+w/2-1) +
           summa(max(a,c+w/2), b, 2*k+1, c+w/2, d);
}
```

Nyt välin $[a,b]$ summan saa laskettua kutsumalla funktiota näin:

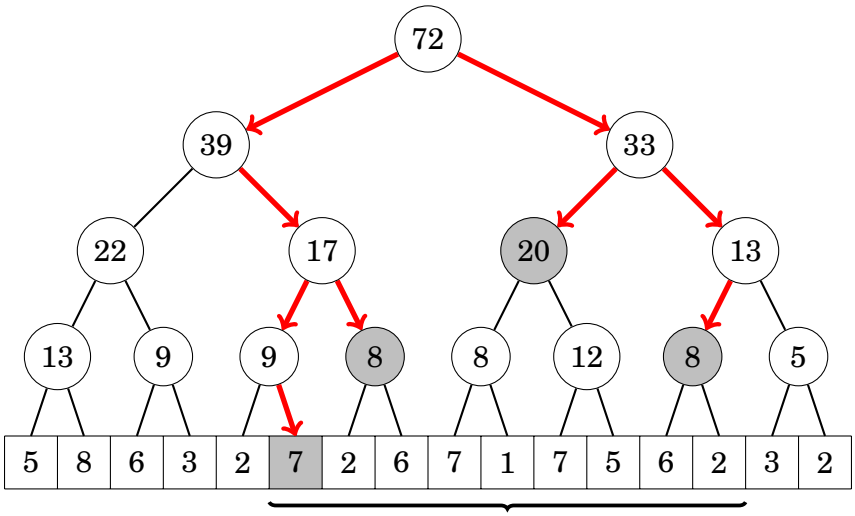
```
int s = summa(a, b, 1, 0, N-1);
```

Parametri k ilmaisee kohdan taulukossa p. Aluksi k:n arvona on 1, koska summan laskeminen alkaa segmenttipuun juuresta.

Väli $[c,d]$ on parametria k vastaava väli, aluksi koko kyselyalue eli $[0,N-1]$. Jos väli $[a,b]$ on välin $[c,d]$ ulkopuolella, välin summa on 0. Jos taas välit $[a,b]$ ja $[c,d]$ ovat samat, summan saa taulukosta p.

Jos väli $[a,b]$ on välin $[c,d]$ sisällä, haku jatkuu rekursiivisesti välin $[c,d]$ vasemmasta ja oikeasta puoliskosta. Kun w on välin $[c,d]$ pituus, vasen puolisko kattaa välin $[c,c+w/2-1]$ ja oikea puolisko kattaa välin $[c+w/2,d]$.

Seuraava kuva näyttää, kuinka haku etenee puussa, kun lasketaan puun alle merkityn välin summa. Harmaat solmut ovat kohtia, joissa rekursio päättyy ja välin summan saa taulukosta p.



Myös tässä toteutuksessa kyselyn aikavaativuus on $O(\log n)$, koska haun aikana käsiteltävien solmujen määrä on $O(\log n)$.

28.2 Laiska eteneminen

Laiska eteneminen (*lazy propagation*) mahdollistaa segmenttipuun, jossa voi sekä muuttaa väliä että kysyä tietoa väliltä ajassa $O(\log n)$. Ideana on suorittaa muutokset ja kyselyt ylhäältä alaspäin ja toteuttaa muutokset laiskasti niin, että ne välitetään puussa alaspäin vain silloin, kun se on välttämätöntä.

Laiskassa segmenttipuussa solmuihin liittyy kahdenlaista tietoa. Kuten tavallisessa segmenttipuussa, jokaisessa solmussa on sitä vastaavan välin summa tai muu haluttu tieto. Tämän lisäksi solmussa voi olla laiskaan etenemiseen liittyvää tietoa, jota ei ole vielä välitetty solmusta alaspäin.

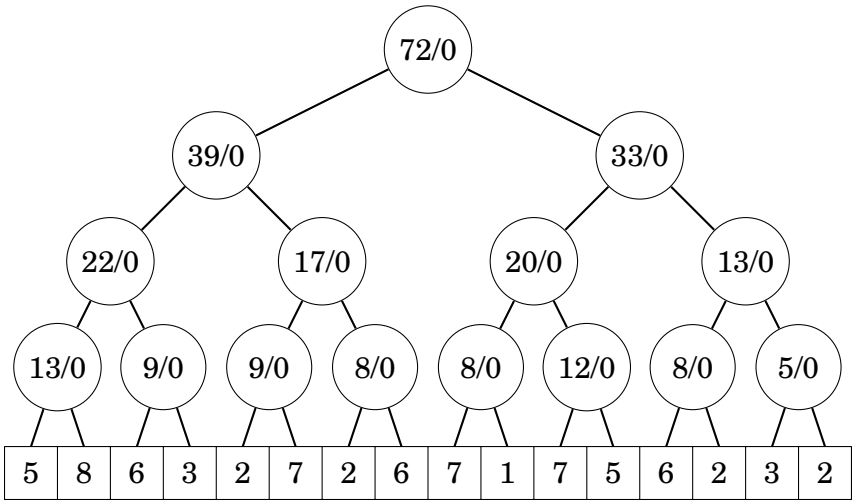
Välin muutostapa voi olla joko *lisäys* tai *asetus*. Lisäyksessä välin jokaiseen alkioon lisätään tietty arvo, ja asetuksessa välin jokainen alkio saa tietyn arvon.

Kummankin operaation toteutus on melko samanlainen, ja puu voi myös tukea samaan aikaan molempia muutostapoja.

28.2.1 Esimerkki

Tehtävä: Toteuta segmenttipuu, jonka avulla voi kasvattaa jokaista välin $[a,b]$ arvoa x:llä sekä laskea välin $[a,b]$ summan.

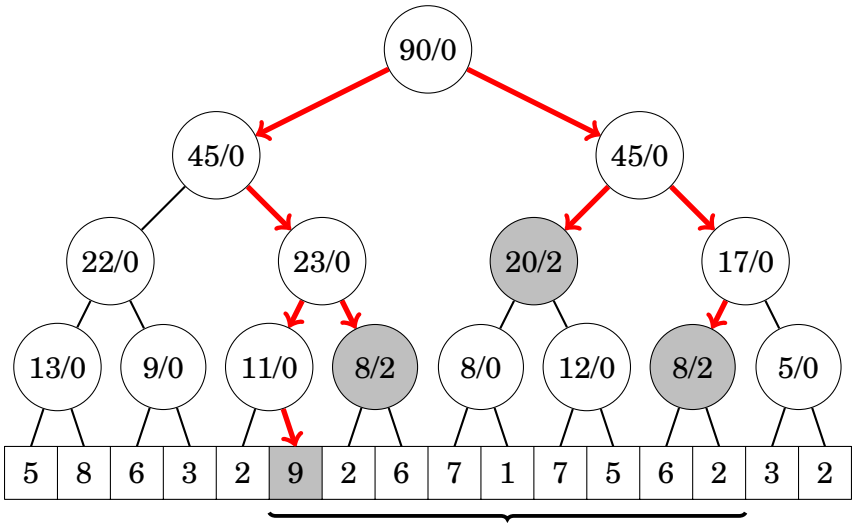
Jokaisessa puun solmussa on kaksi arvoa (s/z): välin lukujen summa s, kuten tavallisessa segmenttipuussa, sekä laiska muutos z, joka tarkoittaa, että kaikkiin välin lukuihin tulee lisätä z. Seuraavassa puussa jokaisessa solmussa $z = 0$ eli mitään muutoksia ei ole kesken.



Kun välin $[a,b]$ solmuja kasvatetaan x:llä, alkaa kulku puun juuresta lehtiä kohti. Kulun aikana tapahtuu kahdenlaisia muutoksia puun solmuihin:

Jos solmun väli $[c,d]$ kuuluu kokonaan muutettavalle välille $[a,b]$, solmun arvo z kasvaa x:llä ja kulku pysähtyy. Jos taas väli $[c,d]$ kuuluu osittain välille $[a,b]$, solmun arvo s kasvaa wx:llä, missä w on välien $[a,b]$ ja $[c,d]$ yhteisen osan pituus, ja kulku jatkuu rekursiivisesti alaspäin.

Kasvatetaan esimerkiksi puun alle merkittyä väliä 2:lla:



Välin $[a,b]$ summan laskenta tapahtuu myös kulkuna puun juuresta lehtiä kohti. Jos solmun väli $[c,d]$ kuuluu kokonaan väliin $[a,b]$, se kasvattaa kyselyn tuloksena olevaa summaa arvolla $s+(d-c+1)z$. Muussa tapauksessa kulku jatkuu rekursiivisesti alaspäin solmun lapsiin.

Laiskat muutokset välitetään puussa alaspäin samalla kun puussa kuljetaan alaspäin välin muutoksen tai summakyselyn yhteydessä. Ideana on, että laiska muutos etenee vain silloin, kun siihen on todellinen syy. Useita laiskoja muutoksia voi olla myös päällekkäin niin, että z kertoo niiden summan.

28.2.2 Polynomit

Laiskaa segmenttipuuta voi yleistää niin, että väliä muuttaa polynomi

$$p(x) = t_k x^k + t_{k-1} x^{k-1} + \cdots + t_0.$$

Ideana on, että välin ensimmäisen kohdan muutos on $p(0)$, toisen kohdan muutos on $p(1)$ jne., eli välillä $[a,b]$ kohta i muutos on $p(i-a)$. Esimerkiksi polynomin $p(x) = x+1$ lisäys välille $[a,b]$ tarkoittaa, että kohta a kasvaa 1:llä, kohta $a+1$ kasvaa 2:lla, kohta $a+2$ kasvaa 3:lla jne.

Polynomimuutoksen voi toteuttaa niin, että jokaisessa solmussa on $k+2$ arvoa, missä k on polynomin asteluku. Arvo s kertoo solmua vastaavan välin summan kuten ennenkin, ja arvot $z_0, z_1, \ldots, z_k$ ovat polynomin kertoimet, joka ilmaisee väliin kohdistuvan laiskan muutoksen.

Nyt välin $[c,d]$ summa on

$$s + \sum_{x=0}^{d-c} z_k x^k + z_{k-1} x^{k-1} + \cdots + z_0.$$

Summan saa laskettua tehokkaasti osissa summakaavalla. Esimerkiksi termin z_0 summaksi tulee $(d-c+1)z_0$ ja termin $z_1 x$ summaksi tulee

$$z_1(0+1+\cdots+d-c) = z_1 \frac{(d-c)(d-c+1)}{2}.$$

Kun muutos etenee alaspäin puussa, polynomin $p(x)$ indeksointi muuttuu, koska jokaisella välillä $[c,d]$ polynomin arvot tulevat kohdista $x = 0,1,\ldots,d-c$. Tämä ei kuitenkaan tuota ongelmia, koska $p'(x) = p(x+c)$ on samanasteinen polynomi kuin $p(x)$, kun c on vakio.

Esimerkiksi jos $p(x) = t_2 x^2 + t_1 x - t_0$, niin

$$p'(x) = t_2(x+c)^2 + t_1(x+c) - t_0 = t_2 x^2 + (2ct_2 + t_1)x + t_2 c^2 + t_1 c - t_0.$$

28.3 Dynaaminen toteutus

Tavallinen segmenttipuu on staattinen, eli jokaiselle solmulle on paikka taulukossa ja puu vie kiinteän määrän muistia. Tämä toteutus kuitenkin tuhlaa muistia, jos suurin osa puun solmuista on tyhjiä. Dynaaminen segmenttipuu varaa muistia vain niille solmuille, joita todella tarvitaan.

Solmut on kätevää tallentaa tietueina tähän tapaan:

```
struct node {
    int x;
    int a, b;
    node *l, *r;
    node(int x, int a, int b) : x(x), a(a), b(b) {}
};
```

Tässä x on solmussa oleva arvo, $[a,b]$ on solmua vastaava väli ja l ja r osoittavat solmun vasempaan ja oikeaan alipuuhun.

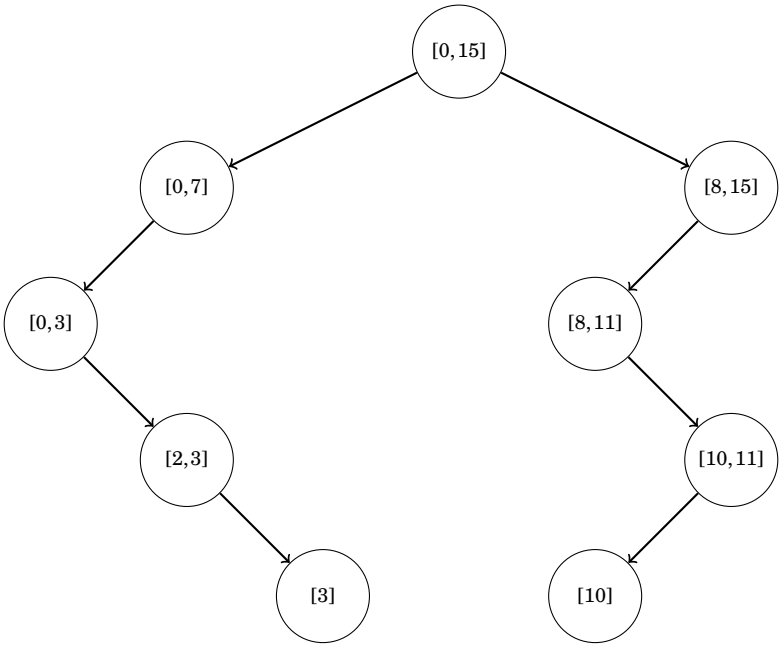
Tämän jälkeen solmuja voi käsitellä seuraavasti:

```
// uuden solmun luonti
node *s = new node(0, 0, 15);
// kentän muuttaminen
s->x = 5;
```

28.3.1 Harva indeksialue

Dynaaminen segmenttipuu on hyödyllinen, jos puun indeksialue $[0,N-1]$ on harva eli N on suuri mutta vain pieni osa indekseistä on käytössä. Siinä missä tavallinen segmenttipuu vie muistia $O(N)$, dynaaminen segmenttipuu vie muistia vain $O(n \log N)$, missä n on käytössä olevien indeksien määrä.

Ideana on, että puu on aluksi tyhjä ja sen ainoa solmu on $[0,N-1]$. Kun puu muuttuu, siihen lisätään solmuja dynaamisesti sitä mukaa kuin niitä tarvitaan uusien indeksien vuoksi. Esimerkiksi jos $N = 16$ ja indeksejä 3 ja 10 on muutettu, puu sisältää seuraavat solmut:



Reitti puun juuresta lehteen sisältää $O(\log N)$ solmua, joten jokainen muutos puuhun lisää enintään $O(\log N)$ uutta solmua puuhun. Niinpä n muutoksen jälkeen puussa on enintään $O(n \log N)$ solmua.

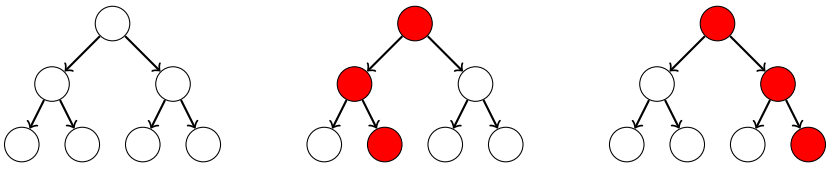
Huomaa, että jos kaikki tarvittavat indeksit ovat tiedossa algoritmin alussa, dynaamisen segmenttipuun sijasta voi käyttää tavallista segmenttipuuta ja koordinaattien pakkausta (luku 9.3). Tämä ei ole kuitenkaan mahdollista, jos indeksit syntyvät vasta algoritmin aikana.

28.3.2 Muutoshistoria

Dynaaminen segmenttipuu mahdollistaa myös puun muutoshistorian säilyttämiseen. Tämä tarkoittaa, että muistissa on jokainen segmenttipuun vaihe, joka on esiintynyt algoritmin suorituksen aikana.

Muutoshistorian hyötynä on, että kaikkia vanhoja puita voi käsitellä segmenttipuun tapaan, koska niiden rakenne on edelleen olemassa. Vanhoista puista voi myös johtaa uusia puita, joita voi muokata edelleen.

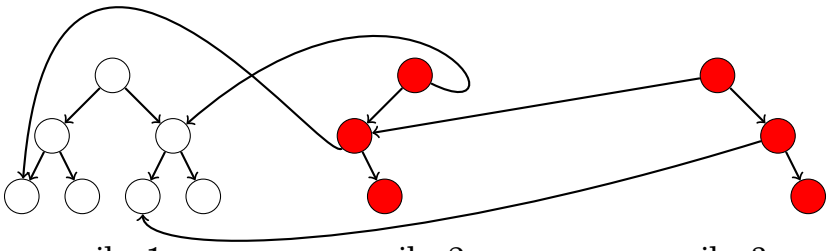
Tarkastellaan esimerkiksi seuraavaa muutossarjaa, jossa punaiset solmut muuttuvat päivityksessä ja muut solmut säilyvät ennallaan:



vaihe 1 vaihe 2 vaihe 3

Jokaisen muutoksen jälkeen suurin osa puun solmuista säilyy ennallaan. Muistia säästävä tapa tallentaa muutoshistoria onkin käyttää mahdollisimman paljon hyväksi puun vanhoja osia muutoksissa.

Tässä tapauksessa muutoshistorian voisi tallentaa seuraavasti:



vaihe 1 vaihe 2 vaihe 3

Nyt muistissa on jokaisesta puun vaiheesta puun juuri, jonka avulla pystyy selvittämään koko puun rakenteen kyseisellä hetkellä. Jokainen muutos tuo vain $O(\log N)$ uutta solmua puuhun, kun puun indeksialue on $[0, N-1]$, joten koko muutoshistorian pitäminen muistissa on mahdollista.

28.4 Tietorakenne solmussa

Segmenttipuun solmussa voi olla yksittäisen arvon sijasta myös jokin tietorakenne, joka pitää yllä tietoa solmua vastaavasta välistä. Tällöin segmenttipuun operaatiot vievät aikaa $O(f(n)\log n)$, missä $f(n)$ on yksittäisen solmun tietorakenteen käsittelyyn kuluva aika.

28.4.1 Lukumäärät

Tehtävä: Toteuta segmenttipuu, jonka avulla voi laskea, montako kertaa luku x esiintyy välillä $[a,b]$.

Ideana on, että jokainen segmenttipuun solmu sisältää tietorakenteen, josta voi kysyä, montako kertaa luku x esiintyy kyseisellä välillä. Vastaus kyselyyn syntyy laskemalla yhteen esiintymismäärät väleiltä, joista $[a,b]$ muodostuu.

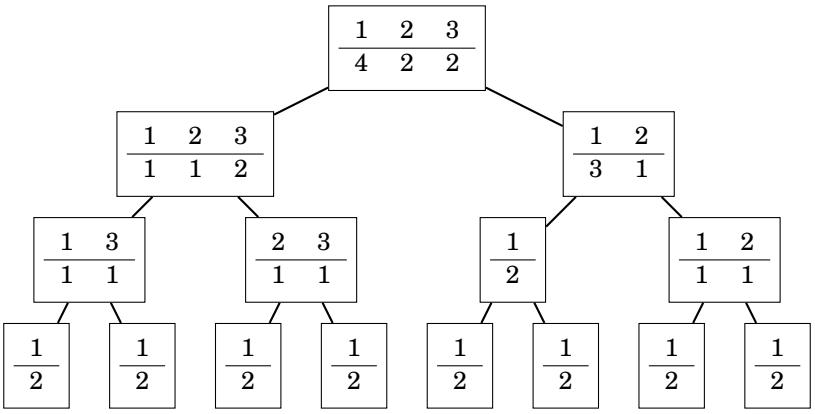
Sopiva tietorakenne tehtävään on `map`-rakenne, jonka avulla voi pitää kirjaa välillä esiintyvien lukujen määrästä. Yhden solmun käsittely vie aikaa $O(\log n)$, joten kunkin kyselyn aikavaativuus on $O(\log^2 n)$.

Solmuissa olevat tietorakenteet kasvattavat segmenttipuun muistinkäyttöä. Tässä tapauksessa segmenttipuu vie tilaa $O(n \log n)$, koska siinä on $O(\log n)$ tasoa, joista jokaisella binääripuut sisältävät $O(n)$ lukua.

Esimerkiksi taulukosta

3	1	2	3	1	1	1	2

syntyy seuraava segmenttipuu:



28.4.2 Kaksiulotteisuus

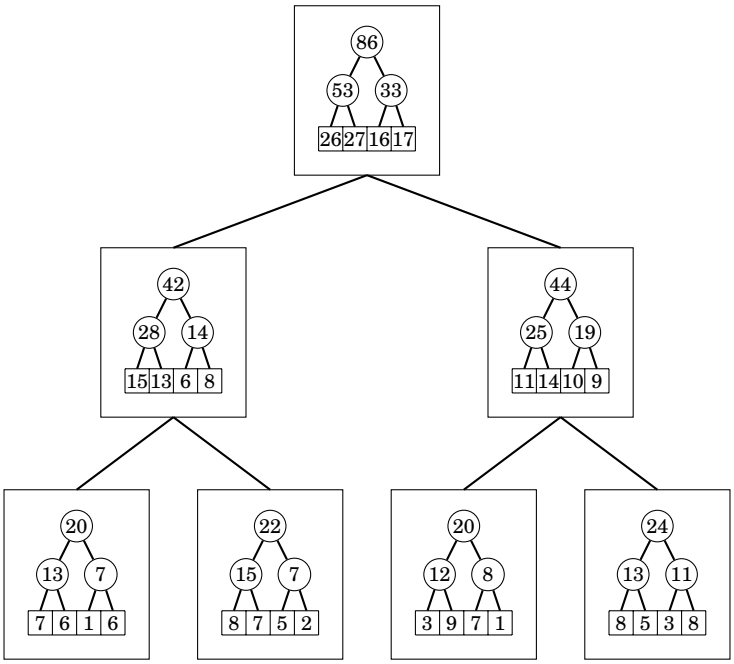
Tehtävä: Toteuta segmenttipuu, jolla pystyy laskemaan $n \times n$ -kokoisen kaksiulotteisen taulukon alitaulukon summan sekä muuttamaan taulukkoa.

Tavallinen tapa toteuttaa kaksiulotteinen segmenttipuu on luoda segmenttipuu, jonka jokaisessa solmussa on segmenttipuu. Suuri segmenttipuu vastaa taulukon rivejä, ja pienet segmenttipuut vastaavat sarakkeita.

Esimerkiksi taulukon

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

alueiden summia voi laskea seuraavasta segmenttipuusta:



Segmenttipuun operaatiot vievät aikaa $O(\log^2 n)$, koska suuressa puussa ja kussakin pienessä puussa on $O(\log n)$ tasoa. Segmenttipuu vie muistia $O(n^2)$, koska jokainen pieni puu vie muistia $O(n)$.

Vastaavalla tavalla voi luoda myös segmenttipuita, joissa on vielä enemmän ulottuvuuksia, mutta tälle on harvoin tarvetta.

Luku 29

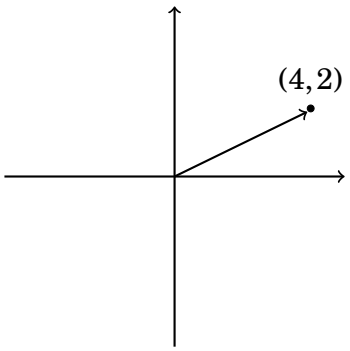
Geometria

Geometrisissä tehtävissä on usein haasteena keksiä, mistä suunnasta ongelmaa kannattaa lähestyä, jotta ratkaisun saa koodattua mukavasti ja erikoistapauksia tulee mahdollisimman vähän. Tässä luvussa tutustumme tekniikoihin, jotka helpottavat geometristen algoritmien toteutusta.

29.1 Kompleksiluvut

Kompleksiluku on luku muotoa $x + yi$, missä $i = \sqrt{-1}$ on imaginääriyksikkö. Kompleksiluvun luonteva geometrinen tulkinta on, että se esittää kaksiulotteisen tason pistettä (x, y) tai vektoria origosta pisteeseen (x, y).

Esimerkiksi luku $4 + 2i$ tarkoittaa seuraavaa pistettä ja vektoria:



C++:ssa on kompleksilukujen käsittelyyn luokka `complex`, josta on hyötyä geometriassa, koska sen avulla voi esittää pisteen tai vektorin kompleksilukuna ja luokassa on valmiita geometriaan soveltuvia työkaluja.

Seuraavassa koodissa `C` on koordinaatin tyyppi ja `P` on pisteen tai vektorin tyyppi. Lisäksi koodi määrittelee lyhennysmerkinnät `X` ja `Y`, joiden avulla pystyy viittaamaan x- ja y-koordinaatteihin.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

Esimerkiksi seuraava koodi määrittelee pisteen $p = (4,2)$ ja ilmoittaa sen x- ja y-koordinaatin:

```
P p = {4,2};
cout << p.X << " " << p.Y << "\n"; // 4 2
```

Seuraava koodi määrittelee vektorit $v = (3,1)$ ja $u = (2,2)$ ja laskee sitten niiden summan $s = v + u$:

```
P v = {3,1};
P u = {2,2};
P s = v+u;
cout << s.X << " " << s.Y << "\n"; // 5 3
```

Sopiva tyyppi koordinaatille on tilanteesta riippuen `long long` (kokonaisluku) tai `long double` (liukuluku). Kokonaislukuja kannattaa käyttää aina kun mahdollista, koska silloin laskenta on tarkkaa.

Jos koordinaatit ovat liukulukuja, niiden vertailussa täytyy ottaa huomioon epätarkkuus. Turvallinen tapa tarkistaa, ovatko liukuluvut a ja b samat on käyttää vertailua $|a-b| < \epsilon$, jossa ϵ on pieni luku (esimerkiksi $\epsilon = 10^{-9}$).

Funktioita

Seuraavissa esimerkeissä pisteen tyyppinä on `long double`:

Funktio `abs(v)` laskee vektorin $v = (x,y)$ pituuden $|v|$ kaavalla $\sqrt{x^2+y^2}$. Sillä voi laskea myös pisteiden (x_1,y_1) ja (x_2,y_2) etäisyyden, koska pisteiden etäisyys on sama kuin vektorin (x_2-x_1, y_2-y_1) pituus. Joskus hyödyllinen on myös funktio `norm(v)`, joka laskee vektorin $v = (x,y)$ pituuden neliön $|v|^2$.

Seuraava koodi laskee pisteiden $(4,2)$ ja $(3,-1)$ etäisyyden:

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.60555
```

Funktio `arg(v)` laskee vektorin $v = (x,y)$ kulman radiaaneina suhteessa x-akseliin. Radiaaneina ilmoitettu kulma r vastaa asteina kulmaa $180 \cdot r/\pi$ astetta. Jos vektori osoittaa suoraan oikealle, sen kulma on 0. Kulma kasvaa vastapäivään ja vähenee myötäpäivään liikuttaessa.

Funktio `polar(s,a)` muodostaa vektorin, jonka pituus on s ja joka osoittaa kulmaan a. Lisäksi vektoria pystyy kääntämään kulman a verran kertomalla se vektorilla, jonka pituus on 1 ja kulma on a.

Seuraava koodi laskee vektorin $(4,2)$ kulman, kääntää sitä sitten 1/2 radiaania vastapäivään ja laskee uuden kulman:

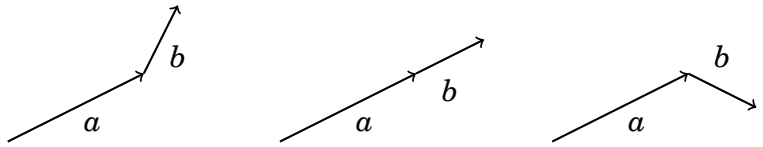
```
P v = {4,2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; // 0.963648
```

29.2 Pisteen sijainti

29.2.1 Ristitulo

Vektorien $a = (x_1, y_1)$ ja $b = (x_2, y_2)$ ristitulo $a \times b$ lasketaan kaavalla $x_1 y_2 - x_2 y_1$. Ristitulo ilmaisee, mihin suuntaan vektori b kääntyy, jos se laitetaan vektorin a perään. Positiivinen ristitulo tarkoittaa käännöstä vasemmalle, negatiivinen käännöstä oikealle, ja nolla tarkoittaa, että vektorit ovat samalla suoralla.

Seuraava kuva näyttää kolme esimerkkiä ristitulosta:



$a \times b = 6$ $a \times b = 0$ $a \times b = -8$

Luokkaa `complex` käyttäen vektorien a ja b ristitulon voi laskea näin:

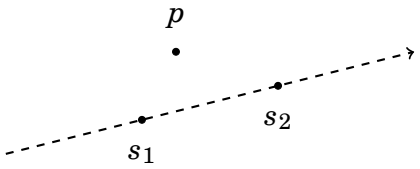
```
P a = {4,2};
P b = {1,2};
C r = (conj(a)*b).Y; // 6
```

Tämä perustuu siihen, että funktio `conj` muuttaa vektorin y-koordinaatin käänteiseksi ja kompleksilukujen kertolaskun seurauksena vektorien $(x_1, -y_1)$ ja (x_2, y_2) kertolaskun y-koordinaatti on $x_1 y_2 - x_2 y_1$.

29.2.2 Suora ja piste

Ristitulon avulla voi selvittää, kummalla puolella suoraa tutkittava piste sijaitsee. Oletetaan, että suora kulkee pisteiden s_1 ja s_2 kautta, katsontasuunta on pisteestä s_1 pisteeseen s_2 ja tutkittava piste on p.

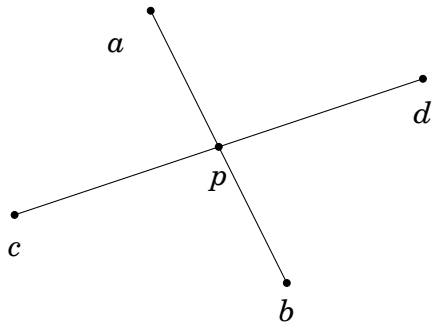
Esimerkiksi seuraavassa kuvassa piste p on suoran vasemmalla puolella:



Ristitulo $(p - s_1) \times (p - s_2)$ kertoo, kummalla puolella suoraa piste sijaitsee. Jos ristitulo on positiivinen, piste p on suoran vasemmalla puolella, ja jos ristitulo on negatiivinen, piste p on suoran oikealla puolella. Jos taas ristitulo on nolla, piste p on pisteiden s_1 ja s_2 kanssa suoralla.

29.2.3 Janojen leikkaus

Usein esiintyvä tehtävä on selvittää, leikkaavatko kaksi janaa. Esimerkiksi seuraavassa kuvassa janat ab ja cd leikkaavat pisteessä p.



Oletetaan, että janat eivät ole samalla suoralla eikä niillä ole yhteisiä päätepisteitä. Tässä tapauksessa janat leikkaavat tarkalleen silloin, kun samaan aikaan pisteet c ja d ovat eri puolilla a:sta b:hen kulkevaa suoraa ja pisteet a ja b ovat eri puolilla c:stä d:hen kulkevaa suoraa.

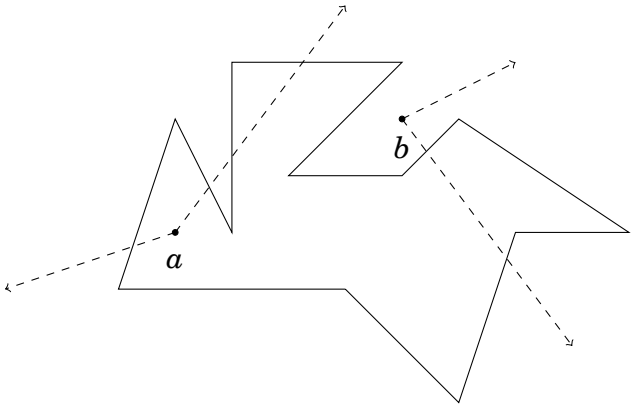
Janojen leikkauspiste p selviää etsimällä parametrit t ja u niin, että

$$p = a + t(b-a) = c + u(d-c).$$

29.2.4 Piste monikulmiossa

Kätevä keino tarkistaa, onko piste monikulmion sisällä, on lähettää pisteestä säde satunnaiseen suuntaan ja laskea, montako kertaa se osuu monikulmion reunaan. Jos kertoja on pariton määrä, piste on sisäpuolella, ja jos kertoja on parillinen määrä, piste on ulkopuolella.

Esimerkiksi seuraavassa kuvassa piste a on monikulmion sisäpuolella ja piste b on ulkopuolella. Kuvassa on myös joitakin pisteistä lähteviä säteitä.



Pisteestä a lähtevät säteet osuvat 1 ja 3 kertaa monikulmion reunaan, joten piste on sisäpuolella. Vastaavasti pisteestä b lähtevät säteet osuvat 0 ja 2 kertaa monikulmion reunaan, joten piste on ulkopuolella.

29.3 Pinta-alat

29.3.1 Kolmion pinta-ala

Kolmion pinta-alan laskemiseen on monia tapoja. Koulusta tuttu kaava on

$$\frac{bh}{2},$$

missä b on kannan pituus ja h on korkeus.

Heronin kaava

$$\sqrt{s(s-a)(s-b)(s-c)}$$

laskee pinta-alan, kun sivujen pituudet ovat a, b ja c, ja $s = (a+b+c)/2$.

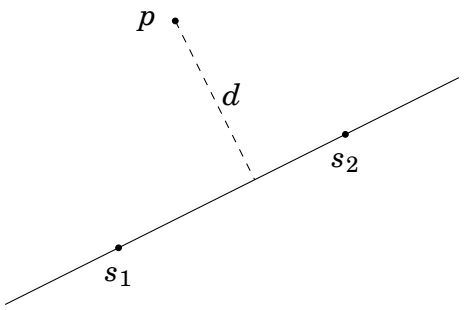
Pinta-alan voi laskea myös ristitulon avulla kaavalla

$$\frac{1}{2}((p_2 - p_1) \times (p_3 - p_1)),$$

missä kolmion kärkipisteet ovat $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ ja $p_3 = (x_3, y_3)$.

Pisteen etäisyys suorasta

Kolmion pinta-alan avulla voi selvittää, kuinka kaukana piste on suorasta. Esimerkiksi seuraavassa kuvassa d on lyhin etäisyys pisteestä p suoralle, jonka määrittävät pisteet s_1 ja s_2:



Pisteiden s_1, s_2 ja p muodostaman kolmion pinta-ala on toisaalta $\frac{1}{2}|s_2 - s_1|d$ ja toisaalta $\frac{1}{2}((s_1 - p) \times (s_2 - p))$, joten etäisyys on

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

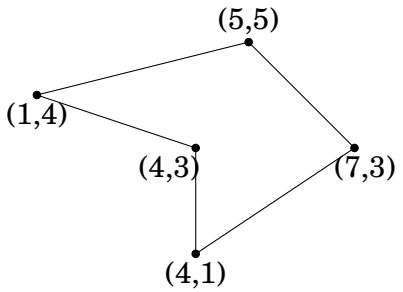
29.3.2 Monikulmion pinta-ala

Yleinen kaava monikulmion pinta-alan laskemiseen on

$$\frac{1}{2}|\sum_{i=1}^{i=n-1} (p_i \times p_{i+1})|,$$

missä monikulmion kärkipisteet ovat järjestyksessä $p_1, p_2, \ldots, p_n$ ja ensimmäinen ja viimeinen kärkipiste ovat samat eli $p_1 = p_n$.

Esimerkiksi monikulmion



pinta-ala on

$$\frac{|(4 \cdot 3 - 7 \cdot 1) + (7 \cdot 5 - 5 \cdot 3) + (5 \cdot 4 - 1 \cdot 5) + (1 \cdot 3 - 4 \cdot 4) + (4 \cdot 1 - 4 \cdot 3)|}{2} = 19/2.$$

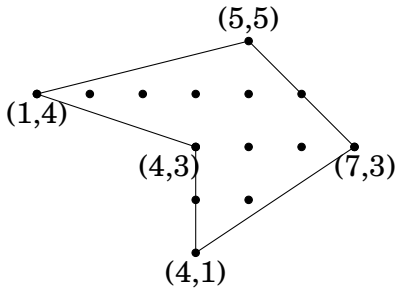
29.3.3 Pickin lause

Toinen tapa laskea monikulmion pinta-ala on käyttää Pickin lausetta. Se pätee silloin, kun kaikki kärkipisteet ovat kokonaislukupisteissä. Pickin lauseen mukaan monikulmion pinta-ala on

$$a + b/2 - 1,$$

missä a on kokonaislukupisteiden määrä monikulmion sisällä ja b on kokonaislukupisteiden määrä monikulmion reunalla.

Esimerkiksi monikulmion



pinta-ala on $7 + 7/2 - 1 = 19/2$.

Luku 30

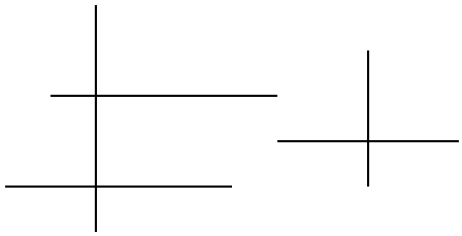
Pyyhkäisyviiva

Pyyhkäisyviiva on laskennallisen geometrian tekniikka, jossa on ideana kulkea tason halki vaaka- tai pystysuuntaisesti ja laskea kunkin pisteen kohdalla tehtävän ratkaisua eteenpäin sopivien tietorakenteiden avulla. Tämä luku esittelee tehtäviä, jotka voi ratkaista pyyhkäisyviivan avulla.

30.1 Leikkauspisteet

Tehtävä: Annettuna on n viivaa, joista jokainen on vaaka- tai pystysuuntainen. Monessako pisteessä kaksi viivaa leikkaa toisiaan?

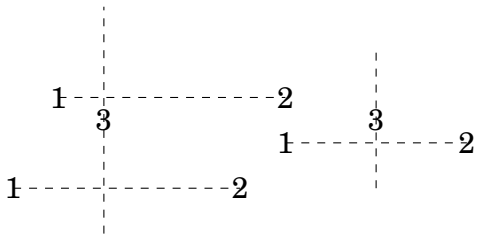
Esimerkiksi seuraavassa tilanteessa leikkauspisteitä on kolme:



Tehtävä on helppoa ratkaista ajassa $O(n^2)$, koska riittää käydä läpi kaikki mahdolliset janaparit ja tarkistaa, moniko leikkaa toisiaan. Seuraavaksi ratkaisemme tehtävän ajassa $O(n \log n)$ pyyhkäisyviivan avulla.

Ideana on luoda janoista kolmentyyppisiä pisteitä: (1) vaakajana alkaa, (2) vaakajana päättyy, (3) pystyjana.

Yllä olevassa esimerkissä pistejoukko on:



Algoritmi käy läpi pisteet vasemmalta oikealle ja pitää yllä tietorakennetta y-koordinaateista, joissa on tällä hetkellä aktiivinen vaakajana. Tapahtuman 1 kohdalla vaakajanan y-koordinaatti lisätään joukkoon ja tapahtuman 2 kohdalla vaakajanan y-koordinaatti poistetaan joukosta.

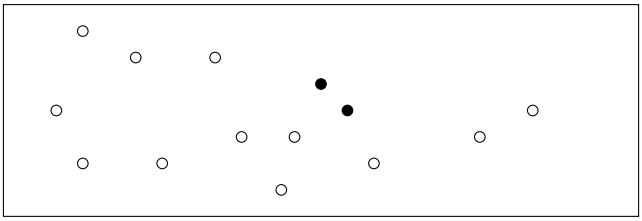
Algoritmi laskee janojen leikkauspisteet tapahtumien 3 kohdalla. Kun pystyjana kulkee y-koordinaattien $y_1 \ldots y_2$ välillä, algoritmi laskee tietorakenteesta, monessako vaakajanassa on y-koordinaatti välillä $y_1 \ldots y_2$ ja kasvattaa leikkauspisteiden määrää tällä arvolla.

Sopiva tietorakenne vaakajanojen y-koordinaattien tallentamiseen on segmenttipuu, johon on tarvittaessa yhdistetty indeksien pakkaus. Segmenttipuun avulla jokaisen pisteen käsittely vie aikaa $O(\log n)$, joten algoritmin kokonaisaikavaativuus on $O(n \log n)$.

30.2 Lähin pistepari

Tehtävä: Annettuna on n pistettä kaksiulotteisessa tasossa ja tehtäväsi on etsiä kaksi pistettä, jotka ovat mahdollisimman lähellä toisiaan.

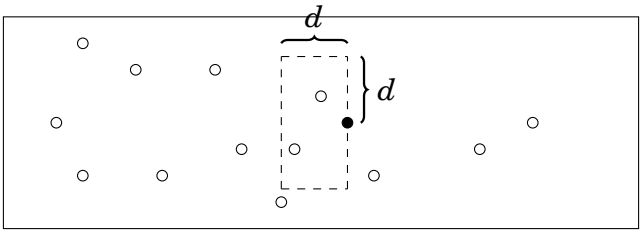
Esimerkiksi seuraavassa kuvassa ratkaisu on tummennettu pistepari:



Tämäkin tehtävä ratkeaa $O(n \log n)$-ajassa pyyhkäisyviivan avulla. Algoritmi käy pisteet läpi vasemmalta oikealle ja pitää yllä arvoa d, joka on pienin kahden pisteen etäisyys. Kunkin pisteen kohdalla algoritmi etsii lähimmän toisen pisteen vasemmalta. Jos etäisyys tähän pisteeseen on alle d, tämä on uusi pienin kahden pisteen etäisyys ja algoritmi päivittää d:n arvon.

Jos käsiteltävä piste on (x, y) ja jokin vasemmalla oleva piste on alle d:n etäisyydellä, sen x-koordinaatin tulee olla välillä $[x-d, x]$ ja y-koordinaatin tulee olla välillä $[y-d, y+d]$. Algoritmin riittää siis tarkistaa ainoastaan pisteet, jotka osuvat tälle välille, mikä tehostaa hakua merkittävästi.

Esimerkiksi seuraavassa kuvassa katkoviiva-alue sisältää pisteet, jotka voivat olla alle d:n etäisyydellä tummennetusta pisteestä.



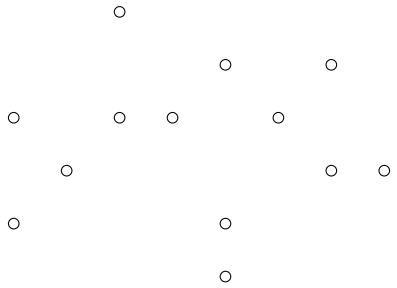
Algoritmin tehokkuus perustuu siihen, että d:n rajoittamalla alueella on aina vain $O(1)$ pistettä. Nämä pisteet pystyy käymään läpi $O(\log n)$-aikaisesti pitämällä algoritmin aikana yllä joukkoa pisteistä, joiden x-koordinaatti on välillä $[x-d,x]$ ja jotka on järjestetty y-koordinaatin mukaan.

Algoritmin aikavaativuus on $O(n \log n)$, koska se käy läpi n pistettä ja etsii jokaiselle lähimmän edeltävän pisteen ajassa $O(\log n)$.

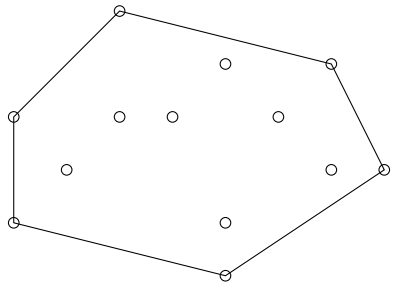
30.3 Konveksi peite

Konveksi peite (*convex hull*) on pienin konveksi monikulmio, joka ympäröi kaikki pistejoukon pisteet. Konveksius tarkoittaa, että minkä tahansa kahden kärkipisteen välinen jana kulkee monikulmion sisällä. Hyvä mielikuva asiasta on, että pistejoukko ympäröidään tiukasti viritetyllä narulla.

Esimerkiksi pistejoukon



konveksi peite on seuraava:

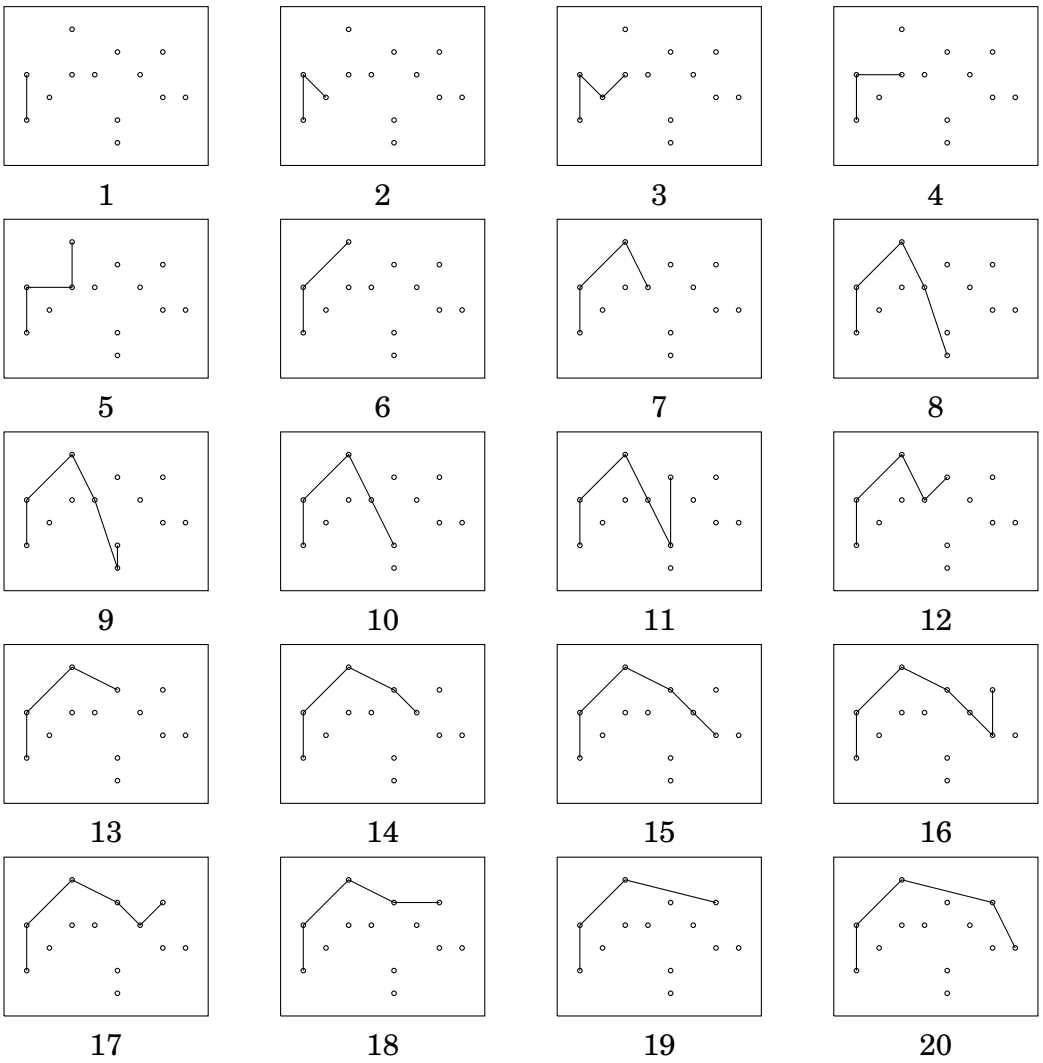


Tehokas ja helposti toteutettava tapa muodostaa konveksi peite on Andrew'n algoritmi, jonka aikavaativuus on $O(n \log n)$.

Algoritmi muodostaa konveksin peitteen kahdessa osassa: ensin peitteen yläosan ja sitten peitteen alaosan. Kummankin osan muodostaminen tapahtuu samalla tavalla, ja keskitymme nyt yläosan muodostamiseen.

Algoritmi järjestää ensin pisteet ensisijaisesti x-koordinaatin ja toissijaisesti y-koordinaatin mukaan. Tämän jälkeen se käy pisteet läpi järjestyksessä ja liittää aina uuden pisteen osaksi peitettä. Kuitenkin aina kun kolme viimeistä pistettä peitteessä muodostavat vasemmalle kääntyvän osan, algoritmi poistaa näistä keskimmäisen pisteen.

Seuraava kuvasarja esittää Andrew'n algoritmin toimintaa:



Vasemmalle kääntyvän osan tarkistus onnistuu ristitulon avulla. Algoritmin aikavaativuus on $O(n \log n)$, koska pisteiden järjestäminen vie aikaa $O(n \log n)$ ja sen jälkeen konveksin peitteen muodostaminen vie aikaa $O(n)$.