

Kisakoodarin käsikirja

Antti Laaksonen

22. marraskuuta 2016

Sisältö

Alkusanat	v
I Perusasiat	1
1 Johdanto	3
2 Aikavaativuus	15
3 Järjestäminen	23
4 Tietorakenteet	33
5 Täydellinen haku	43
6 Ahneet algoritmit	51
7 Dynaaminen ohjelmointi	57
8 Tasoitettu analyysi	69
9 Välikyselyt	75
10 Bittien käsittely	87
II Verkkoalgoritmit	95
11 Verkkojen perusteet	97
12 Verkon läpikäynti	105
13 Lyhimmät polut	111
14 Puiden käsittely	121
15 Virittävät puut	127
16 Suunnatut verkot	135
17 Vahvasti yhtenäisyys	143

18 Puukyselyt	149
19 Polut ja kierrokset	157
20 Virtauslaskenta	163
 III Uusia haasteita	 175
21 Lukuteoria	177
22 Kombinatoriikka	183
23 Matriisit	193
24 Todennäköisyys	197
25 Peliteoria	203
26 Merkkijonot	209
27 Neliöjuorialgoritmit	217
28 Lisää segmenttipuusta	221
29 Geometria	229
30 Pyyhkäisyviiva	235

Alkusanat

Tämän kirjan tarkoituksena on antaa sinulle perusteellinen johdatus kisakoodauksen maailmaan. Kirja olettaa, että osaat ennestään ohjelmoinnin perusasiat, mutta aiempaa kokemusta kisakoodauksesta ei vaadita.

Kirja on tarkoitettu erityisesti Datatähti-kilpailun osallistujille, jotka haluavat oppia algoritmiaa ja mahdollisesti osallistua kansainvälisiin tietotekniikan olympialaisiin (IOI). Lisäksi kirja sopii myös yliopistojen opiskelijoille ja kaikille muille kisakoodauksesta kiinnostuneille.

Kirjan lukuihin liittyy kokoelma harjoitustehtäviä, jotka ovat saatavilla osoitteessa <https://cses.fi/dt/list/>.

Hyväksi kisakoodariksi kehittyminen vie paljon aikaa, mutta se on samalla mahdollisuus oppia paljon. Voit olla varma, että tulet saamaan hyvän ymmärryksen algoritmia perusteista kirjaa lukemalla ja tehtäviä ratkomalla.

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

Osa I

Perusasiat

Luku 1

Johdanto

Kisakoodauksessa yhdistyy kaksi asiaa: (1) algoritmien suunnittelu ja (2) algoritmien toteutus.

Algoritmien suunnittelu on loogista ongelmanratkaisua, joka on lähellä matematiikkaa. Se vaatii kykyä analysoida ongelmia ja ratkaista niitä luovasti. Tehtävän ratkaisevan algoritmin tulee olla sekä toimiva että tehokas, ja usein nimenomaan tehokkaan algoritmin keksiminen on tehtävän ydin.

Teoreettiset tiedot algoritmikasta ovat kisakoodarille kullannarvoisia. Usein tehtävän ratkaisu on yhdistelmä tunnettuja tekniikoita sekä omia oivalluksia. Kisakoodauksessa esiintyvät menetelmät ovat myös algoritmikan tieteellisen tutkimuksen perusta.

Algoritmien toteutus edellyttää hyvää ohjelmointitaitoa. Kisakoodauksessa tehtävän ratkaisun arvostelu tapahtuu testaamalla toteutettua algoritmia joukolla testisyötteitä. Ei siis riitä, että algoritmin idea on oikea, vaan algoritmi pitää myös onnistua toteuttamaan virheettömästi.

Hyvä kisakoodi on suoraviivaista ja tiivistä. Ratkaisu täytyy pystyä kirjoittamaan nopeasti, koska kisoissa on vain vähän aikaa. Toisin kuin tavallisessa ohjelmistokehityksessä, ratkaisut ovat lyhyitä (yleensä enintään joitakin satoja rivejä) eikä koodia tarvitse kehittää kilpailun jälkeen.

1.1 Ohjelmointikielet

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 ¹. Jotkut osallistajat 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.

Toisaalta on hyvä hallita useita kieliä ja tuntea niiden edut. Esimerkiksi jos tehtävässä esiintyy suuria kokonaislukuja, Python voi olla hyvä valinta, kos-

¹<https://www.go-hero.net/jam/16>

ka kielessä on sisäänrakennettuna suurten kokonaislukujen käsittely. Toisaalta tehtävät yritetään yleensä laatia niin, ettei tietyn kielen ominaisuuksista ole kohtuutonta hyötyä tehtävän ratkaisussa.

Kaikki tämän kirjan esimerkit on kirjoitettu C++:lla, ja niissä on käytetty runsaasti C++:n valmiita tietorakenteita ja algoritmeja. Käytössä on C++:n standardi C++11, jota voi nykyään käyttää useimmissa kisoissa. Jos et vielä osaa ohjelmoida C++:lla, nyt on hyvä hetki alkaa opetella.

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 g++-kääntäjän tarjoama tapa ottaa mukaan kaikki standardikirjaston sisältö. Tämän ansiosta koodissa ei tarvitse erikseen ottaa mukaan 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 tuottaa kooditiedostosta `koodi.cpp` binääritiedoston `koodi`. Kääntäjä noudattaa C++11-standardia (`-std=c++11`), optimoi koodia käännöksen aikana (`-O2`) ja näyttää varoituksia mahdollisista virheistä (`-Wall`).

1.3 Syöte ja tuloste

Useimmissa kisoissa käytetään standardivirtoja syötteen lukemiseen ja tulosteen kirjoittamiseen. C++:ssa standardivirrät ovat `cin` lukemiseen ja `cout` tulostamiseen. Lisäksi voi käyttää C:n funktioita `scanf` ja `printf`.

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;
```

Tällainen koodi toimii aina, kunhan jokaisen luettavan alkion välissä on ainakin yksi rivinvaihto tai välilyönti. Esimerkiksi yllä oleva koodi hyväksyy molemmat seuraavat syötteet:

```
123 456 apina
```

```
123    456
apina
```

Tulostaminen tapahtuu seuraavasti cout-virran kautta:

```
int a = 123, b = 456;
string x = "apina";
cout << a << " " << b << " " << x << "\n";
```

Syötteen ja tulosteen käsittely on joskus pullonkaula ohjelmassa. Seuraavat rivit koodin alussa tehostavat syötteen ja tulosteen käsittelyä:

```
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.

C:n funktiot scanf ja printf ovat vaihtoehto C++:n standardivirroille. Ne ovat yleensä hieman nopeampia, mutta toisaalta vaikeakäyttöisempiä. Seuraava koodi lukee kaksi kokonaislukua syöttestä:

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

Seuraava koodi taas tulostaa kaksi kokonaislukua:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

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

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

Jos syötteessä olevan tiedon määrä ei ole tiedossa etukäteen, seuraavanlainen silmukka on kätevä:

```
while (cin >> x) {
    // koodia
}
```

Tämä silmukka lukee syötettä alkio kerrallaan, kunnes syöte loppuu.

Joissakin kisajärjestelmissä syötteen ja tulosteen käsittelyyn käytetään tiedostoja. 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

1.4.1 Kokonaisluvut

Tavallisin kokonaislukutyyppi kisakoodauksessa on `int`. Tämä on 32-bittinen tyyppi, jonka sallittu lukuväli on $-2^{31} \dots 2^{31} - 1$ eli noin $-2 \cdot 10^9 \dots 2 \cdot 10^9$. Jos tyyppi `int` ei riitä, sen sijaan voi käyttää 64-bittistä tyyppiä `long long`, jonka lukuväli on $-2^{63} \dots 2^{63} - 1$ eli noin $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$.

Seuraava koodi määrittelee `long long`-muuttujan:

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

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

Yleinen virhe `long long`-tyypin käytössä on, että jossain kohtaa käytetään kuitenkin `int`-tyyppejä. 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` tyyppiä `long long` tai kirjoittamalla laskun muodossa `(long long)a*a`.

Yleensä tehtävät laaditaan niin, että tyyppi `long long` riittää niiden ratkaisuun. Kuitenkin `g++`-kääntäjä sisältää myös 128-bittisen tyypin `__int128_t`, jonka lukuväli on $-2^{127} \dots 2^{127} - 1$ eli noin $-10^{38} \dots 10^{38}$. Tämä tyyppi ei kuitenkaan ole käytettävissä kaikissa kisajärjestelmissä.

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 tyyppiejä `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 n kertoman modulo M :

```
long long x = 1;
for (int i = 2; i <= n i++) {
    x = (x*i)%M;
}
cout << x << "\n";
```

Yleensä vastaus modulona tulee antaa niin, että se on aina välillä $0 \dots M - 1$. Kuitenkin C++:ssa ja muissa kielissä negatiivisen luvun modulo on nolla tai negatiivinen. Helppo tapa varmistaa, että modulo ei ole negatiivinen, on laskea ensin 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

Tavalliset liukulukutyypit kisakoodauksessa ovat 64-bittinen double sekä g++-kääntäjän laajennoksena 80-bittinen long double. Yleensä double riittää, mutta long double tuo tarvittaessa lisää tarkkuutta.

Vastauksena oleva liukuluku täytyy yleensä tulostaa tietyllä tarkkuudella. Tämä onnistuu helpoiten printf-funktiolla, jolle voi antaa desimaalien määrän. Esimerkiksi seuraava koodi tulostaa luvun 9 desimaalin tarkkuudella:

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

Liukulukujen käyttämisessä on hankaluutena, 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:

```
0.999999999999999988898
```

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

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

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

Vaikka liukuluvut ovat epätarkkoja, niillä voi esittää tarkasti kokonaislukuja tiettyyn rajaan asti. Esimerkiksi `double`-tyypillä voi esittää tarkasti kaikki kokonaisluvut, joiden itseisarvo on enintään 2^{53} .

1.5 Koodin lyhentäminen

1.5.1 Tyypinimet

Komennolla `typedef` voi antaa lyhyemmän nimen tyyppille. Esimerkiksi nimi `long long` on pitkä, joten tyyppille voi antaa lyhyemmän nimen `ll`:

```
typedef long long ll;
```

Tämän jälkeen seuraavat kaksi koodia tarkoittavat samaa:

```
long long f(long long x) {  
    long long u = (x == 0) ? 1 : f(x/2);  
    return u*u;  
}
```

```
ll f(ll x) {  
    ll u = (x == 0) ? 1 : f(x/2);  
    return u*u;  
}
```

Komentoa `typedef` voi käyttää myös monimutkaisempien tyyppien kanssa. Esimerkiksi seuraava koodi antaa nimen `vi` kokonaisluvuihin muodostuvalle vektorille sekä nimen `pi` kaksi kokonaislukua sisältävälle parille.

```
typedef vector<int> vi;  
typedef pair<int,int> pi;
```

1.5.2 Makrot

Esikäntäjän komennolla `#define` voi määritellä makroja, joiden avulla voi lyhentää koodia. Yksinkertaisimmillaan makro antaa lyhyemmän version jollekin koodissa esiintyvälle sanalle.

Määritellään esimerkiksi seuraavat makrot:

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

Tämän jälkeen seuraavat koodit vastaavat toisiaan:

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

Makro on mahdollista määritellä myös niin, että sille voi antaa parametreja. Tämän ansiosta makrolla voi lyhentää esimerkiksi komentorakenteita.

Määritellään esimerkiksi seuraava makro:

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

Tämän jälkeen seuraavat koodit vastaavat toisiaan:

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

```
REP(i,1,n) {
    haku(i);
}
```

Huomaa, että toisin kuin funktioissa, makron parametri sijoitetaan lausekkeeseen sellaisenaan. Tästä voi tulla joskus yllättäviä ongelmia. Näin on esimerkiksi seuraavassa makrossa:

```
#define SQR(x) x*x
```

Nyt seuraavat koodit vastaavat toisiaan:

```
cout << SQR(1+2) << "\n";
```

```
cout << 1+2*1+2 << "\n";
```

Tuloksena koodi laskee siis $1 + (2 \cdot 1) + 2$, vaikka tarkoitus olisi laskea $(1 + 2) \cdot (1 + 2)$. Yleensä helppo korjaus ongelmaan on lisätä makron lausekkeeseen sulkuja, tässä tapauksessa seuraavasti:

```
#define SQR(x) (x)*(x)
```

1.6 Virheen etsiminen

Yleensä kilpailun aikana ei saa tietää, millä syötteillä koodia testataan. Niinpä jos koodi toimii väärin, syy virheeseen täytyy löytää itse. Virhe voi selvitä helposti koodia tutkimalla, mutta virhettä voi olla myös vaikeaa löytää.

Järjestelmällinen tapa etsiä virhettä on koodata kaksi ratkaisua: tehokas palautettavaksi tarkoitettu ratkaisu sekä toinen raa'an voiman ratkaisu, joka toimii varmasti oikein mutta ei ole tehokas. Tämän jälkeen riittää etsiä syöte, jossa nämä koodit toimivat eri tavalla.

Seuraavassa on esimerkki Python-koodista, joka etsii virheellistä syötettä automaattisesti. Tehokas ratkaisu on nimeltään fast ja raa'an voiman ratkaisu on nimeltään brute. Syötteessä annetaan ensin lukujen määrä n ja sen jälkeen n lukua väliltä $1 \dots r$.

```
from os import system
from random import randint

n = 10 # lukujen määrä
r = 5 # lukujen yläraja

while True:
    f = open("input.txt","w")
    f.write(str(n) + "\n")
    u = []
    for i in range(n):
        u.append(str(randint(1,r)))
    f.write(" ".join(u) + "\n")
    f.close()
    system("./fast < input.txt > output1.txt")
    system("./brute < input.txt > output2.txt")
    o1 = open("output1.txt").readlines()
    o2 = open("output2.txt").readlines()
    if o1 == o2:
        print "ok"
    else:
        print "virhe!"
        exit(0)
```


Koodi muodostaa satunnaisen syötteen tiedostoon "input.txt" ja suorittaa sitten kummankin koodin tälle syötteelle niin, että tulosteet ohjautuvat tiedostoihin "output1.txt" ja "output2.txt". Tämän jälkeen koodi vertaa tulostiedostoja ja pysähtyy, jos ne eivät ole samat.

Tällä lähestymistavalla koodissa oleva virhe löytyy lähes varmasti. Lisäksi yleensä jos koodissa on virhe, se esiintyy jollakin pienellä syötteellä, mikä helpottaa virheen etsimistä koodista.

1.7 Matematiikka

1.7.1 Merkintöjä

Summa $\sum_{x=a}^b$ käy läpi välin $a \dots b$ kokonaisluvut. Jokaisen luvun kohdalla summaan lisätään yleensä x :stä riippuvan lausekkeen arvo. Esimerkiksi $\sum_{x=1}^n x$ vastaa summaa $1 + 2 + 3 + \dots + n$ ja $\sum_{x=1}^n x^2$ vastaa summaa $1^2 + 2^2 + 3^2 + \dots + n^2$.

Vastaavasti tulo $\prod_{x=a}^b$ käy läpi välin $a \dots b$ kokonaisluvut. Esimerkiksi $\prod_{x=1}^n x$ vastaa tuloa $1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ (eli kertomaa $n!$).

Merkinnät $\lfloor x \rfloor$ ja $\lceil x \rceil$ pyöristävät luvun x kokonaisluvuksi alaspäin ja ylöspäin. Esimerkiksi $\lfloor \pi \rfloor = 3$ ja $\lceil \pi \rceil = 4$, missä $\pi = 3,14159\dots$

Merkinnät $\min(a_1, a_2, \dots, a_n)$ ja $\max(a_1, a_2, \dots, a_n)$ valitsevat pienimmän ja suurimman alkioista a_1, a_2, \dots, a_n .

1.7.2 Joukot

Joukko on kokoelma alkioita. Esimerkiksi joukko $\{2, 4, 7\}$ sisältää alkioit 2, 4 ja 7. Tietty alkio voi esiintyä joukossa enintään kerran. Merkintä \emptyset tarkoittaa tyhjää joukkoa. Joukon S koko eli alkoiden määrä on $|S|$, esimerkiksi $|\{2, 4, 7\}| = 3$.

Merkintä $x \in S$ tarkoittaa, että alkio x on joukossa S , ja merkintä $x \notin S$ tarkoittaa, että alkio x ei ole joukossa S . Esimerkiksi $4 \in \{2, 4, 7\}$, mutta $5 \notin \{2, 4, 7\}$.

Leikkaus $A \cap B$ sisältää alkioit, jotka ovat molemmissa joukoista A ja B . Yhdiste $A \cup B$ sisältää alkioit, jotka ovat ainakin toisessa joukoista A ja B . Erotus $A \setminus B$ sisältää alkioit, jotka ovat joukossa A mutta eivät joukossa B . Esimerkiksi jos $A = \{1, 2, 5\}$ ja $B = \{2, 4\}$, niin $A \cap B = \{2\}$, $A \cup B = \{1, 2, 4, 5\}$ ja $A \setminus B = \{1, 5\}$.

Merkintä $A \subset S$ tarkoittaa, että A on S :n osajoukko eli jokainen A :n alkio esiintyy S :ssä. Esimerkiksi joukon $\{2, 4, 7\}$ osajoukot ovat \emptyset , $\{2\}$, $\{4\}$, $\{7\}$, $\{2, 4\}$, $\{2, 7\}$, $\{4, 7\}$ ja $\{2, 4, 7\}$. Joukon S osajoukkojen määrä on $2^{|S|}$.

Usein esiintyviä joukkoja ovat \mathbb{N} (luonnolliset luvut), \mathbb{Z} (kokonaisluvut), \mathbb{Q} (rationaaliluvut) sekä \mathbb{R} (reaaliluvut). Joukko \mathbb{N} voidaan määritellä kahdella tavalla: joko $\mathbb{N} = \{0, 1, 2, \dots\}$ tai $\mathbb{N} = \{1, 2, 3, \dots\}$.

1.7.3 Logiikka

Loogisen lausekkeen arvo on joko 1 (tosi) tai 0 (epätosi). Tärkeimmät loogiset operaatiot ovat \neg (negaatio), \wedge (konjunktio), \vee (disjunktio), \Rightarrow (implikaatio) sekä \Leftrightarrow (ekvivalenssi). Seuraava taulukko näyttää operaatioiden merkityksen:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Negaatio $\neg A$ muuttaa lausekkeen käänteiseksi. Lauseke $A \wedge B$ on tosi, jos molemmat A ja B ovat tosia, ja lauseke $A \vee B$ on tosi, jos A tai B tai molemmat ovat tosia. Lauseke $A \Rightarrow B$ on tosi, jos A :n ollessa tosi myös B :n on aina tosi. Lauseke $A \Leftrightarrow B$ on tosi, jos A :n ja B :n totuusarvo on sama.

Predikaatti on lauseke, jonka arvo on tosi tai epätosi riippuen sen parametreista. Yleensä predikaattia merkitään suurella kirjaimella. Esimerkiksi voidaan määritellä predikaatti $P(x)$, joka on tosi tarkalleen silloin, kun x on alkuluku. Tällöin esimerkiksi $P(7)$ on tosi, kun taas $P(8)$ on epätosi.

Kvanttoreita ovat \forall (kaikille) ja \exists (on olemassa). Esimerkiksi $\forall x(\exists y(y > x))$ tarkoittaa kokonaislukujen joukossa, että jokaiselle luvulle x on olemassa jokin luku y niin, että y on x :ää suurempi. Tämän lausekkeen arvo on tosi, koska kokonaislukuja on äärettömästi.

Näiden merkintöjä avulla on mahdollista esittää monenlaisia loogisia väitteitä. Esimerkiksi $\forall x(\neg P(x) \Rightarrow (\exists a(\exists b(x = ab \wedge a > 1 \wedge b > 1))))$ tarkoittaa, että jos luku x ei ole alkuluku, niin on olemassa luvut a ja b , joiden tulo on x ja jotka molemmat ovat suurempia kuin 1. Tämänkin lausekkeen arvo on tosi.

1.7.4 Logaritmi

Logaritmi merkitään $\log_k(x)$, missä k on logaritmin kantaluku. Logaritmi on potenssilaskun käänteisoperaatio: $\log_k(x) = a$ tarkalleen silloin, kun $k^a = x$.

Algoritmiikassa hyödyllinen tulkinta on, että logaritmi $\log_k(x)$ ilmaisee, montako kertaa lukua x täytyy jakaa k :lla, ennen kuin tulos on 1. Esimerkiksi $\log_2(32) = 5$, koska 2:lla jakaminen muuttaa lukua 32 seuraavasti:

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

Logaritmi tulee usein vastaan algoritmiikassa, koska monessa tehokkaassa algoritmissa jokin asia puolittuu joka askeleella. Niinpä logaritmin avulla voi arvioida algoritmin tehokkuutta.

Logaritmi muuttaa kertolaskun yhteenlaskuksi ja jakolaskun vähennyslaskuksi: $\log_k(a \cdot b) = \log_k(a) + \log_k(b)$ ja $\log_k(a/b) = \log_k(a) - \log_k(b)$. Lisäksi on voimassa kaava $\log_u(x) = \log_k(x)/\log_k(u)$. Tämän ansiosta logaritmeja voi laskea mille tahansa kantaluvulle kiinteän kantaluvun funktiolla.

Yksi logaritmin ominaisuus on, että se kertoo luvun numeroiden määrän halutussa lukujärjestelmässä. Esimerkiksi luvun x numeroiden määrä 10-järjestelmässä on $\lfloor \log_{10}(x) \rfloor + 1$.

1.7.5 Summakaavat

Aritmeettinen summa

Aritmeettisessa summassa jokaisen vierekkäisen summattavan luvun erotus on vakio. Esimerkiksi $3+7+11+15$ on aritmeettinen summa, jossa vakio on 4. Siinä pätee siis $7-3=4$, $11-7=4$ ja $15-11=4$.

Aritmeettinen summa voidaan laskea kaavalla

$$\frac{n(a+b)}{2},$$

missä summan ensimmäinen luku on a , viimeinen luku on b ja lukujen määrä on n . Esimerkiksi

$$3+7+11+15 = \frac{4 \cdot (3+15)}{2} = 36$$

Kaava perustuu siihen, että summa muodostuu n luvusta ja luvun suuruus on keskimäärin $(a+b)/2$.

Aritmeettisen summan erikoistapaus on

$$1+2+3+\dots+n = \frac{n(n+1)}{2}.$$

Geometrinen summa

Geometrisessa summassa jokaisen vierekkäisen summattavan luvun suhde on vakio. Esimerkiksi $3+6+12+24$ on geometrinen summa, jossa vakio on 2. Siinä pätee $6/3=2$, $12/6=2$ ja $24/12=2$.

Geometrinen summa voidaan laskea kaavalla

$$\frac{bx-a}{x-1},$$

missä summan ensimmäinen luku on a , viimeinen luku on b ja vierekkäisten lukujen suhde on x . Esimerkiksi

$$3+6+12+24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Geometrisen summan kaavan voi johtaa merkitsemällä

$$S = a + ax + ax^2 + \dots + b.$$

Kertomalla molemmat puolet x :llä saadaan

$$xS = ax + ax^2 + ax^3 + \dots + bx,$$

josta kaava seuraa ratkaisemalla yhtälön

$$xS - S = bx - a.$$

Geometrisen summan erikoistapaus on usein kätevä kaava

$$1+2+4+8+\dots+2^n = 2^{n+1} - 1.$$

Harmoninen summa

Harmoninen summa on

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}.$$

Yläraja harmonisen summan suuruudelle on $\log_2(n) + 1$. Summaa voi näet arvioida ylöspäin muuttamalla jokaista termiä $1/k$ niin, että k :ksi tulee alempi 2 :n potenssi. Esimerkiksi tapauksessa $n = 6$ arvioksi tulee

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Tämän seurauksena summa jakaantuu $\log_2(n) + 1$ samaa rationaalilukua toistamaan osaan, joista jokaisen summa on enintään 1.

Muita kaavoja

Jokaiselle muotoa $\sum_{x=1}^n x^k$ olevalle summalle on olemassa oma kaavansa, kun k on vakio. Mitä suurempi k on, sitä monimutkaisempi kaava on.

Esimerkiksi

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

ja

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}.$$

Geometrisen summan sukulainen on

$$x + 2x^2 + 3x^3 + \dots + kx^k = \frac{kx^{k+2} - (k+1)x^{k+1} + x}{(x-1)^2}.$$

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 arvioi 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 $O(\dots)$, 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 algoritmista on, sitä hitaampi se on. Jos sisäkkäisiä silmukoita on k , aikavaativuus on $O(n^k)$.

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

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

Vastaavasti seuraavan koodin 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 $1 + 2 + \dots + n = \frac{n(n+1)}{2}$ 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. Kokonais-aikavaativuus 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
...	...
$g(1)$	2^{n-1}

Tämän perusteella kutsun $g(n)$ aikavaativuus on

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

2.2 Vaativuusluokkia

$O(1)$ (vakioaikainen)

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

$O(\log n)$ (logaritminen)

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

$O(\sqrt{n})$ (neliöjuuri)

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

$O(n)$ (lineaarinen)

Lineaarinen algoritmi käy syötteen läpi kiinteän määrän kertoja. Tämä on usein paras mahdollinen aikavaativuus, koska yleensä syöte täytyy käydä 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 joukosta kaksi alkia.

$O(n^3)$ (kuutiollinen)

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

$O(2^n)$ (osajoukot)

Aikavaativuus $O(2^n)$ tarkoittaa usein, että algoritmi käy läpi kaikki syötteen osajoukot. Esimerkiksi joukon $\{1, 2, 3\}$ 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 joukon $\{1, 2, 3\}$ permutaatiot ovat $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ sekä $(3, 2, 1)$.

Algoritmi on *polynominen*, jos sen aikavaativuus on korkeintaan $O(n^k)$, kun k on vakio. Edellä mainituista aikavaativuuksista kaikki paitsi $O(2^n)$ ja $O(n!)$ ovat polynomisia. Käytännössä vakio k on yleensä pieni, minkä ansiosta polynomisuus kuvastaa sitä, että algoritmi on tehokas.

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 tärkeitä ongelmia, joihin ei tiedetä polynomista algoritmia.

2.3 Tehokkuuden arviointi

Aikavaativuuden hyötynä on, että sen avulla voi arvioida ennen algoritmin toteuttamista, onko algoritmi riittävän nopea tehtävän ratkaisemiseen. Lähtökohtana arviossa on, että nykyaikainen tietokone pystyy suorittamaan sekunnissa sadasta miljoonasta miljardiin koodissa olevaa komentoa.

Oletetaan esimerkiksi, että tehtävän aikaraja on yksi sekunti ja syötteen koko on $n = 10^5$. Jos algoritmin aikavaativuus on $O(n^2)$, algoritmi suorittaa noin $(10^5)^2 = 10^{10}$ komentoa. Tähän kuluu aikaa ainakin kymmenen sekunnin luokkaa, joten algoritmi vaikuttaa liian hitaalta tehtävän ratkaisemiseen.

Käänteisesti syötteen koosta voi päätellä, kuinka tehokasta algoritmia tehtävän laatija odottaa ratkaisijalta. Seuraavassa taulukossa on joitakin hyödyllisiä arvioita, jotka olettavat, että tehtävän aikaraja on yksi sekunti.

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 $O(n \log n)$
$n \leq 5000$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$
$n \leq 10$	$O(n!)$

Esimerkiksi jos syötteen koko on $n = 10^5$, tehtävän laatija odottaa luultavasti algoritmia, jonka aikavaativuus on $O(n)$ tai $O(n \log n)$. Tämä tieto helpottaa algoritmin suunnittelua, koska se rajaa pois monia lähestymistapoja, joiden tuloksena olisi hitaampi aikavaativuus.

Aikavaativuus ei kerro kuitenkaan kaikkea algoritmin tehokkuudesta, koska se kätkee toteutuksessa olevat vakiokertoimet. Esimerkiksi aikavaativuuden $O(n)$ algoritmi voi tehdä käytännössä $n/2$ tai $5n$ operaatiota. Tällä on merkittävä vaikutus algoritmin todelliseen ajankäyttöön.

2.4 Esimerkki

Yleensä ohjelmointitehtävän ratkaisuun on monta luontevaa algoritmia, joiden aikavaativuudet eroavat. Tutustumme seuraavaksi klassiseen ongelmaan, jonka suoraviivaisen ratkaisun aikavaativuus on $O(n^3)$, mutta algoritmia parantamalla aikavaativuudeksi tulee ensin $O(n^2)$ ja lopulta $O(n)$.

Tehtävä: Annettuna on taulukko, jossa on n kokonaislukua. Tehtäväsi on etsiä alitaulukko (taulukon yhtenäinen väli), jonka lukujen summa on mahdollisimman suuri.

Tehtävän kiinnostavuus on siinä, että taulukossa saattaa olla negatiivisia lukuja. Vaikka negatiiviset luvut pienentävät summaa, niitä kannattaa joskus ottaa mukaan ratkaisuun, koska negatiivisen luvun kummallakin puolella voi olla positiivisia lukuja, jotka kumoavat negatiivisen luvun vaikutuksen.

Esimerkiksi taulukossa

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

optimiratkaisu on valita alitaulukko seuraavasti:

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

Tämän alitaulukon lukujen summa on $2 + 4 + (-3) + 5 + 2 = 10$, joka on suurin mahdollinen. Keskellä oleva negatiivinen luku -3 kannattaa ottaa mukaan, koska sen kummallakin puolella olevat luvut kasvattavat summaa yli 3:lla.

Ratkaisu 1 ($O(n^3)$)

Suoraviivainen ratkaisu tehtävään on käydä läpi kaikki tavat valita alitaulukko taulukosta, laskea jokaisesta vaihtoehdosta lukujen summa ja pitää muistissa suurinta summaa. Seuraava koodi toteuttaa tämän algoritmin:

```
int p = 0;
for (int a = 1; a <= n; a++) {
    for (int b = a; b <= n; b++) {
        int s = 0;
        for (int c = a; c <= b; c++) {
            s += t[c];
        }
        p = max(p,s);
    }
}
cout << p << "\n";
```

Koodi olettaa, että luvut on tallennettu taulukkoon t , jota indeksoidaan $1 \dots n$. Muuttujat a ja b valitsevat alitaulukon ensimmäisen ja viimeisen luvun, ja alitaulukon summa lasketaan muuttujaan s . Muuttujassa p on taas paras haun aikana löydetty summa.

Algoritmin aikavaativuus on $O(n^3)$, koska siinä on kolme sisäkkäistä silmukkaa ja jokainen silmukka käy läpi $O(n)$ lukua.

Ratkaisu 2 ($O(n^2)$)

Äskeistä ratkaisua on helppoa tehostaa hankkiutumalla eroon sisimmästä silmukasta. Tämä on mahdollista laskemalla summaa samalla, kun alitaulukon oikea reuna liikkuu eteenpäin. Tuloksena on seuraava koodi:

```
int p = 0;
for (int a = 1; a <= n; a++) {
    int s = 0;
    for (int b = a; b <= n; b++) {
        s += t[b];
        p = max(p,s);
    }
}
cout << p << "\n";
```

Tämän muutoksen jälkeen koodin aikavaativuus on $O(n^2)$, koska siinä on kaksi sisäkkäistä silmukkaa.

Ratkaisu 3 ($O(n)$)

Yllättävää kyllä, tehtävään on olemassa myös $O(n)$ -aikainen ratkaisu eli koodista pystyy karsimaan vielä yhden silmukan. Ratkaisun ideana on laskea taulukon jokaiseen kohtaan, mikä on suurin mahdollinen summa kyseiseen kohtaan päättyvässä alitaulukossa, ja valita suurin näistä summista.

Kun alitaulukon loppukohta k on valittu, sen muodostamiseen on kaksi vaihtoehtoa. Yksi mahdollisuus on, että alitaulukossa on vain yksi luku: kohdassa k oleva luku. Muussa tapauksessa siinä on ensin kohtaan $k - 1$ päättyvä alitaulukko, johon on yhdistetty kohdassa k oleva luku.

Koska tavoitteena on saada aikaan mahdollisimman suuri summa, jälkimmäisessä tapauksessa myös kohtaan $k - 1$ päättyvä alitaulukko tulee valita niin, että sen summa on mahdollisimman suuri. Niinpä tehokas ratkaisu syntyy käymällä läpi kaikki loppukohdat k järjestyksessä.

Seuraava koodi toteuttaa ratkaisun:

```
int p = 0, s = 0;
for (int k = 1; k <= n; k++) {
    s = max(t[k], s+t[k]);
    p = max(p,s);
}
cout << p << "\n";
```

Algoritmissa on vain yksi silmukka, joka käy läpi taulukon luvut, joten sen aikavaativuus on $O(n)$. Tämä on myös paras mahdollinen aikavaativuus, koska minkä tahansa algoritmin täytyy käydä läpi ainakin kerran taulukon sisältö.

Tehokkuusvertailu

On kiinnostavaa tutkia, kuinka tehokkaita algoritmit ovat käytännössä. Seuraava taulukko näyttää, kuinka nopeasti äskeiset ratkaisut toimivat eri n :n arvoilla (testikoneena Intel Core i7-2677M, 1,80 GHz).

Jokainen syöte on muodostettu satunnaisesti, ja taulukon luvut ovat välillä $-10^9 \dots 10^9$. Ajankäyttöön ei ole laskettu syötteen lukemiseen kuluvaa aikaa.

taulukon koko n	ratkaisu 1	ratkaisu 2	ratkaisu 3
10^2	0,0 s	0,0 s	0,0 s
10^3	0,1 s	0,0 s	0,0 s
10^4	> 10,0 s	0,1 s	0,0 s
10^5	> 10,0 s	5,3 s	0,0 s
10^6	> 10,0 s	> 10,0 s	0,0 s
10^7	> 10,0 s	> 10,0 s	0,0 s

Vertailu osoittaa, että pienillä syötteillä kaikki algoritmit ovat tehokkaita, mutta suuremmat syötteen tuovat esille huomattavia eroja algoritmien suoritussajassa. $O(n)$ -aikainen ratkaisu 3 on ainoa, joka pystyy ratkaisemaan kaikki syötteen alle 10 sekunnissa.

Luku 3

Järjestäminen

Järjestäminen (*sorting*) on keskeinen algoritmiikan ongelma. Moni tehokas algoritmi perustuu järjestämiseen, koska järjestetyn tiedon käsittely on usein helpompaa kuin sekalaisessa järjestyksessä olevan.

Esimerkiksi kysymys ”onko taulukossa kahta samaa alkiota?” ratkeaa tehokkaasti järjestämisen avulla. Jos taulukossa on kaksi samaa alkiota, ne ovat järjestämisen jälkeen peräkkäin, jolloin niiden löytäminen on helppoa. Samaan tapaan ratkeaa myös kysymys ”mikä on yleisin alkio taulukossa?”.

Järjestämiseen on kehitetty monia algoritmeja, jotka tarjoavat hyviä esimerkkejä algoritmien suunnittelun tekniikoista. Tehokkaat yleiset järjestämisalgoritmit toimivat ajassa $O(n \log n)$, ja tämä aikavaativuus on myös monella järjestämisestä käytävällä algoritmilla.

3.1 Järjestämisen teoriaa

Järjestämisen perusongelma on seuraava:

Tehtävä: Annettuna on taulukko, jossa on n alkiota. Tehtäväsi on järjestää alkiot pienimmästä suurimpaan.

Esimerkiksi taulukko

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

on järjestettynä seuraava:

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

3.1.1 Kuplajärjestäminen

Kuplajärjestäminen (*bubble sort*) on yksinkertainen järjestämisalgoritmi, joka muodostuu taulukon läpikäynneistä. Jokainen läpikäynti etsii taulukosta vierekkäisiä alkiopareja, jotka ovat väärässä järjestyksessä, ja korjaa näiden alkioparien järjestyksen. Algoritmi päättyy, kun läpikäynnin aikana ei tule mitään muutoksia taulukkoon, jolloin taulukko on järjestyksessä.

Kuplajärjestämisen voi toteuttaa seuraavasti, kun järjestettävä taulukko muodostuu alkiosta $t[1], t[2], \dots, t[n]$:

```
bool stop = false;
while (!stop) {
    stop = true;
    for (int i = 1; i <= n-1; i++) {
        if (t[i] > t[i+1]) {
            swap(t[i], t[i+1]);
            stop = false;
        }
    }
}
```

Esimerkiksi taulukossa

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

kuplajärjestämisen ensimmäinen läpikäynti tekee seuraavat vaihdot:

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

↖ ↗

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

↖ ↗

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

↖ ↗

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

↖ ↗

Kuplajärjestämisessä ensimmäisen läpikäynnin jälkeen suurin alkio on paikallaan, toisen läpikäynnin jälkeen kaksi suurinta alkioita on paikallaan, jne. Niinpä kuplajärjestäminen päättyy aina viimeistään n läpikäynnin jälkeen. Koska jokainen läpikäynti vie aikaa $O(n)$, algoritmin aikavaativuus on $O(n^2)$.

3.1.2 Inversiot

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

Hyödyllinen käsite järjestämisalgoritmien analyysissä on inversio. Se on indeksipari (a, b) , joille $a < b$ ja $t[a] > t[b]$ eli kaksi taulukon alkioita, jotka ovat väärässä järjestyksessä. Esimerkiksi taulukossa

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

inversiot ovat $(4, 5)$, $(4, 6)$ ja $(7, 8)$. Inversioiden määrä kuvaa, miten lähellä järjestystä taulukko on. Järjestetyn taulukon inversioiden määrä on 0, ja käänteisesti järjestetyn taulukon inversioiden määrä on $1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2}$.

Jos vierekkäiset alkioita ovat väärässä järjestyksessä, niiden vaihtaminen keskenään poistaa taulukosta yhden inversion. Niinpä inversioiden määrä on sama kuin taulukon järjestämiseen tarvittava vaihtojen määrä. Vierekkäisiä alkioita vaihtavan algoritmin aikavaativuus on aina ainakin $O(n^2)$, koska sen täytyy tehdä pahimmassa tapauksessa $\frac{n(n-1)}{2} = O(n^2)$ vaihtoa.

3.1.3 Lomitusjärjestäminen

Lomitusjärjestäminen (*mergesort*) on tehokas $O(n \log n)$ -aikainen järjestämisalgoritmi. Algoritmi jakaa taulukon kahteen yhtä suureen osaan, järjestää osataulukot rekursiivisesti ja muodostaa sitten järjestetyn taulukon yhdistämällä osataulukot. Algoritmin runko on seuraava:

```
void mergesort(int a, int b) {
    if (a == b) return;
    int c = (a+b)/2;
    mergesort(a, c);
    mergesort(c+1, b);
    merge(a, c, c+1, b);
}
```

Funktio *mergesort* järjestää taulukon välin $a \dots b$ alkioita. Jos $a = b$, välillä on vain yksi alkio, joten se on valmiiksi järjestyksessä. Muuten algoritmi jakaa välin kahteen osaan: vasen osa on väli $a \dots c$ ja oikea osa on väli $c + 1 \dots b$, missä $c = (a + b)/2$. Algoritmi järjestää osat rekursiivisesti kutsumalla itseään.

Algoritmi kutsuu funktiota *merge*, joka lomittaa välin vasemman ja oikean osan alkioita. Tämä tarkoittaa, että alkioita kerätään yhteen niin, että koko taulukon väli on järjestyksessä. Lomitus on mahdollista toteuttaa ajassa $O(n)$ valitsemalla alkioita järjestyksessä vasemman ja oikean osan alusta alkaen.

Esimerkiksi taulukko

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

jakautuu osataulukoiksi

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

jotka ovat järjestettyinä:

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

Osataulukot lomittamalla syntyy järjestetty taulukko

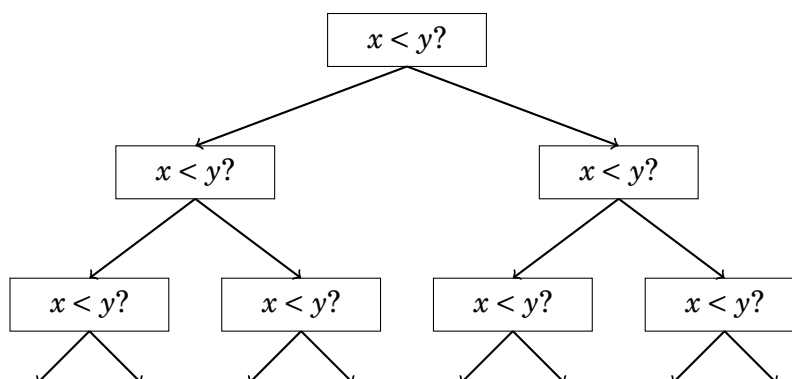
1	2	3	4	5	6	7	8
1	2	2	3	5	6	8	9

Lomitusjärjestämisen aikavaativuus on $O(n \log n)$, koska algoritmin aikana osataulukoista muodostuu $O(\log n)$ tasoa ja kullakin tasolla osataulukoiden lomitus vie yhteensä $O(n)$ aikaa.

3.1.4 Järjestämisen alaraja

Lomitusjärjestämisen lisäksi on olemassa 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 $O(n \log n)$.

Tähän on yllättävä syy: on mahdollista todistaa, että yleistä järjestämisalgoritmia ei voi toteuttaa nopeammin kuin ajassa $O(n \log n)$. Ideana on tarkastella järjestämistä prosessina, jossa jokainen kahden alkion vertailu antaa lisää tietoa taulukon sisällöstä. Prosessista muodostuu seuraavanlainen puu:



Merkintä " $x < y$?" tarkoittaa taulukon alkioiden x ja y vertailua. Jos $x < y$, prosessi jatkaa vasemmalle, ja muuten oikealle. Prosessin tulokset ovat taulukon mahdolliset järjestykset, joita on kaikkiaan $n!$ erilaista. Puun korkeuden tulee olla tämän vuoksi vähintään

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

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 kokonaisluvusta koostuvan taulukon $O(n)$ -ajassa.

Algoritmin ideana on luoda kirjanpito, josta selviää, montako kertaa mikäkin alkio esiintyy taulukossa. Kirjanpito on taulukko, jonka indeksit ovat alkuperäisen taulukon alkioita. Jokaisen indeksin kohdalla lukee, montako kertaa kyseinen alkio esiintyy alkuperäisessä taulukossa.

Esimerkiksi taulukosta

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

syntyy seuraava kirjanpito:

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

Esimerkiksi kirjanpidossa lukee indeksin 3 kohdalla 2, koska luku 3 esiintyy kahdesti alkuperäisessä taulukossa (indekseissä 2 ja 6).

Kirjanpidon muodostus vie aikaa $O(n)$, koska riittää käydä taulukko läpi kerran. Tämän jälkeen järjestetyn taulukon luominen vie myös aikaa $O(n)$, koska kunkin alkion määrän saa selville suoraan kirjanpidosta. Niinpä laskemisjärjestämisen kokonaisaikavaativuus on $O(n)$.

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

C++:n standardikirjastossa on `sort`-funktio, jonka avulla voi järjestää helposti taulukoita ja muita tietorakenteita. Tätä funktiota kannattaa käyttää yleensä aina järjestämiseen algoritmeissa, koska funktio on toteutettu tehokkaasti ja toimivasti. Tutustumme seuraavaksi funktion käyttämiseen tarkemmin.

3.2.1 Peruskäyttö

Seuraava koodi järjestää vektorin `v` luvut pienimmästä suurimpaan:

```
vector<int> v = {4,2,5,3,5,8,3};  
sort(v.begin(), v.end());
```

Järjestämisen jälkeen vektorin sisältö on {2,3,3,4,5,5,8}.

Oletuksena järjestys tapahtuu pienimmästä suurimpaan, mutta järjestyksen saa käänteiseksi näin:

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

Tavallisen taulukon voi järjestää seuraavasti:

```
int n = 7; // taulukon koko  
int t[] = {4,2,5,3,5,8,3};  
sort(t, t+n);
```

Seuraava koodi järjestää merkkijonon `s`:

```
string s = "apina";  
sort(s.begin(), s.end());
```

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

3.2.2 Vertailuoperaattori

Funktion `sort` käyttäminen vaatii, että järjestettävien alkioiden tietotyyppille on määritelty vertailuoperaattori `<`, jonka avulla voi selvittää, mikä on kahden alkion järjestys. Järjestämisen aikana `sort`-funktio käyttää operaattoria `<` aina, kun sen täytyy vertailla järjestettäviä alkioita.

Vertailuoperaattori on määritelty valmiiksi useimmille C++:n tietotyypeille, minkä ansiosta niitä pystyy järjestämään automaattisesti. Jos järjestettävänä on lukuja, ne järjestyvät suuruusjärjestykseen, ja jos järjestettävänä on merkkijonoja, ne järjestyvät aakkosjärjestykseen.

Parit (`pair`) järjestyvät ensisijaisesti ensimmäisen kentän (`first`) mukaan. Jos kuitenkin parien ensimmäiset kentät ovat samat, järjestys määräytyy 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).

Jos järjestettävänä on omia tietueita, niiden vertailuoperaattori täytyy toteuttaa itse. Operaattori määritellään tietueen sisään `operator<`-nimisenä funktiona, jonka parametrina on toinen alkio. Operaattorin tulee palauttaa `true`, jos oma alkio on pienempi kuin parametrialkio, ja muuten `false`.

Esimerkiksi seuraava tietue `P` sisältää pisteen `x`- ja `y`-koordinaatit. Vertailuoperaattori on toteutettu niin, että 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;
    }
};
```

3.2.3 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

Tavallinen tapa etsiä alkioa `x` taulukosta `t` on käyttää `for`-silmukkaa, joka käy läpi taulukon sisällön:

```
for (int i = 1; i <= n; i++) {
    if (t[i] == x) // alkio x löytyi kohdasta i
}
```

Tämän menetelmän aikavaativuus on $O(n)$, koska pahimmassa tapauksessa koko taulukko täytyy käydä läpi. Jos taulukon sisältö voi olla mitä tahansa, tämä on kuitenkin tehokkain mahdollinen menetelmä, koska saatavilla ei ole lisätietoa siitä, mistä päin taulukkoa alkioa `x` kannattaa etsiä.

Tilanne on toinen silloin, kun taulukko on järjestyksessä. Seuraavaksi käsiteltävä binäärihaku (*binary search*) on tehokas menetelmä etsiä alkioa järjes-

tetystä taulukosta. Algoritmi hyödyntää tietoa taulukon järjestyksestä, minkä ansiosta hakuun kuluu aikaa vain $O(\log n)$.

3.3.1 Binäärihaun toteutus

Menetelmä 1

Perinteinen tapa toteuttaa binäärihaku muistuttaa sanan etsimistä sanakirjasta. Haku puolittaa joka askeleella hakualueen taulukossa, kunnes lopulta etsittävä alkio löytyy tai osoittautuu, että sitä ei ole taulukossa.

Haku tarkistaa ensin taulukon keskimmäisen alkion. Jos keskimäinen alkio on etsittävä alkio, haku päättyy. Muuten haku jatkuu taulukon vasempaan tai oikeaan osaan sen mukaan, onko keskimäinen alkio suurempi vain pienempi kuin etsittävä alkio.

Yllä olevan idean voi toteuttaa seuraavasti:

```
int a = 1, b = n;
while (a <= b) {
    int k = (a+b)/2;
    if (t[k] == x) // alkio x löytyi kohdasta k
    if (t[k] > x) b = k-1;
    else a = k+1;
}
```

Algoritmi pitää yllä väliä $a \dots b$, joka on jäljellä oleva hakualue taulukossa. Aluksi väli on $1 \dots n$ eli koko taulukko. Välin koko puolittuu algoritmin joka vaiheessa, joten aikavaativuus on $O(\log n)$.

Menetelmä 2

Vaihtoehtoinen tapa toteuttaa binäärihaku perustuu taulukon tehostettuun läpikäyntiin. Ideana on käydä taulukkoa läpi hyppien ja hidastaa vauhtia, kun etsittävä alkio lähestyy.

Haku käy taulukkoa läpi vasemmalta oikealle aloittaen hypyn pituudesta n . Joka vaiheessa hypyn pituus puolittuu: ensin $n/2$, sitten $n/4$, sitten $n/8$ jne., kunnes lopulta hypyn pituus on 1. Hyppyjen jälkeen joko haettava alkio on löytynyt tai selviää, että sitä ei ole taulukossa.

Seuraava koodi toteuttaa äskeisen idean:

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

Muuttuja k on läpikäynnin kohta taulukossa ja muuttuja b on hypyn pituus. Jos alkio x esiintyy taulukossa, sen kohta on muuttujassa k algoritmin päätteeksi. Algoritmin aikavaativuus on $O(\log n)$, koska while-silmukassa oleva koodi suoritetaan aina enintään kahdesti.

3.3.2 Muutoskohdan etsiminen

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

Toisin sanoen haluamme löytää funktion ok muutoskohdan, jossa arvosta `false` tulee arvo `true`. Tilanne näyttää seuraavalta:

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

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 $ok(x)$ on `false`. Niinpä tästä seuraava arvo $k = x+1$ on pienin arvo, jolla $ok(k)$ on `true`. Hypyn aloituspituus z tulee olla sopiva suuri luku, esimerkiksi sellainen, jolla $ok(z)$ on varmasti `true`.

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

3.3.3 Huippuarvon etsiminen

Binäärihaulla voi myös etsiä suurimman arvon funktiolle, joka on ensin kasvava ja sitten laskeva. Toisin sanoen tehtävänä on etsiä arvo k niin, että

- $f(x) < f(x+1)$, kun $x < k$, ja
- $f(x) > f(x+1)$, kun $x \geq k$.

Ideana on etsiä binäärihaulla viimeinen kohta x , jossa pätee $f(x) < f(x+1)$. Tällöin $k = x+1$, koska pätee $f(x+1) > f(x+2)$. Seuraava koodi toteuttaa haun:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(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*) on tapa 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

4.1.1 Vektori

Vektori (*vector*) on C++:n tavallisin dynaaminen taulukko. Se on kuin taulukko, mutta alkioden määrää voi muuttaa suorituksen aikana.

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

Vektori on toteutettu sisäisesti tavallisena taulukkona. Jos vektorin koko kasvaa ja taulukko jää liian pieneksi, varataan uusi suurempi taulukko, johon kopioidaan vektorin sisältö. Näin tapahtuu kuitenkin niin harvoin, että vektorin funktion `push_back` aikavaativuus on keskimäärin $O(1)$.

4.1.2 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 muodostuu siinä olevista alkioista. Joukon perusoperaatiot 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
```

Tärkeä joukon ominaisuus on, että tietty alkio voi esiintyä siinä vain kerran. Niinpä funktio `count` palauttaa aina arvon 0 (alkiota ei ole joukossa) tai 1 (alkio on joukossa) ja funktio `insert` ei lisää alkiota uudestaan joukkoon, 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 myös rakenteet `multiset` ja `unordered_multiset`, jotka toimivat muuten samalla tavalla kuin `set` ja `unordered_set`, mutta sama alkio voi esiintyä monta kertaa joukossa. Esimerkiksi seuraavassa koodissa kaikki luvun 5 kopiot lisätään joukkoon:

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

Funktio `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-arvo-pareja. Siinä missä taulukossa avaimet ovat aina peräkkäiset kokonaisluvut $0, 1, 2, \dots$, hakemistossa ne voivat olla mitä tahansa tyyppiä.

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 arvot 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 "aybaltu", jonka arvona on 0:

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

Komennolla `count` voi tutkia, esiintyykö avain hakemistossa:

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

Seuraava koodi listaa hakemiston kaikki avaimet ja arvot:

```
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:

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

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

4.4.1 Taulukon välit

Iteraattoreita tarvitsee C++:n standardikirjaston funktioissa, jotka käsittelevät tietorakenteen välejä. Yleensä halutaan käsitellä tietorakenteiden kaikkia alkiota, 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);
```

4.4.2 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 alkia ei esiinny joukossa, iteraattoriksi tulee `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 `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 alkia 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 Bittijoukko

Bittijoukko (bitset) on taulukko, jonka jokaisen alkion arvo on 0 tai 1. Bittijoukon etuna on, että jokainen alkio vie vain yhden bitin tilaa muistissa. Niinpä bittijoukon käyttäminen säästää muistia, jos arvot 0 ja 1 riittävät.

Esimerkiksi jos taulukossa on n arvoa tallennettuna `int`-lukuina, jokainen arvo vie tilaa 32 bittiä ja taulukko vie muistia $32n$ bittiä. Käyttämällä bittijoukkoa tilaa kuluu vain n bittiä eli 32 kertaa vähemmän.

Seuraava koodi luo bittijoukon, jossa on 10 alkiota (indeksointi $0 \dots 9$). Bittijoukkoa voi käyttää samalla tavalla kuin taulukkoa. Lisäksi funktio `count` kertoo ykkösbittien määrän bittijoukossa.

```
bitset<10> s;  
s[2] = 1;  
s[5] = 1;  
cout << s[4] << "\n"; // 0  
cout << s[5] << "\n"; // 1  
cout << s.count() << "\n"; // 2
```

Bittijoukon toinen etu muistinkäytön lisäksi on, että sen sisältöä voi käsitellä suoraan bittiopeeraatioilla. Seuraava koodi näyttää esimerkkejä tästä:

```
bitset<10> a(string("0010110110"));  
bitset<10> b(string("1011011000"));  
cout << (a&b) << "\n"; // 0010010000  
cout << (a|b) << "\n"; // 1011111110  
cout << (a^b) << "\n"; // 1001101110
```

4.5.2 Pakka

Pakka (deque) on dynaaminen taulukko, jonka kokoa pystyy muuttamaan tehokkaasti sekä alku- että loppupäässä. Pakka sisältää vektorin tavoin funktiot `push_back` ja `pop_back`, mutta siinä on lisäksi myös funktiot `push_front` ja `pop_front`, jotka käsittelevät taulukon alkua.

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 vektoria raskaampi rakenne. Kuitenkin lisäyksen ja poiston aikavaativuus on keskimäärin $O(1)$ molemmissa päissä.

4.5.3 Pino ja jono

Pino

Pino (stack) on tietorakenne, joka tarjoaa kaksi $O(1)$ -aikaista operaatiota: alkion lisäys pinon päälle ja alkion poisto pinon päältä. Pinossa ei ole mahdollista käsitellä muita alkioita kuin pinon päällimmäistä alkioita.

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

Jono (queue) on kuin pino, mutta alkion lisäys tapahtuu jonon loppuun ja alkion poisto tapahtuu jonon alusta. Jonossa on mahdollista käsitellä vain alussa ja lopussa olevaa alkioita.

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
```

Huomaa, että rakenteiden stack ja queue sijasta voi aina käyttää rakenteita vector ja deque, joilla voi tehdä kaiken saman ja enemmän. Kuitenkin stack ja queue ovat kevyempiä ja hieman tehokkaampia rakenteita, jos niiden operaatiot riittävät algoritmin toteuttamiseen.

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

C++:n prioriteettijono toimii oletuksena niin, että alkiot ovat järjestyksessä suurimmasta pienimpään ja jonosta pystyy hakemaan ja poistamaan suurimman alkion. Seuraava koodi esittelee prioriteettijonon käyttämistä:

```
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 määrittely luo käänteisen prioriteettijonon, jossa alkiot ovat järjestyksessä pienimmästä suurimpaan ja jonosta pystyy hakemaan ja poistamaan pienimmän alkion:

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

4.6 Erilaiset ratkaisutavat

Monen tehtävän voi ratkaista tehokkaasti joko sopivilla tietorakenteilla tai järjestämisellä. Vaikka erilaiset ratkaisutavat olisivat kaikki periaatteessa tehokkaita, niissä voi olla käytännössä merkittäviä eroja. Näin on esimerkiksi seuraavassa tehtävässä:

Tehtävä: Annettuna on kokonaisluku n sekä listat A ja B , joista kummallakin on n lukua. Mikään luku ei esiinny monta kertaa samalla listalla. Tehtäväsi on selvittää, moniko luku esiintyy kummallakin listalla.

Ratkaisu 1

Ensimmäinen ratkaisu on tallentaa listan A luvut joukkoon ja käydä sitten läpi listan B luvut ja tarkistaa jokaisesta, esiintyykö se myös listalla A . Joukon ansiosta on tehokasta tarkastaa, esiintyykö listan B luku listalla A . Kun joukko toteutetaan set-rakenteella, algoritmin aikavaativuus on $O(n \log n)$.

Ratkaisu 2

Joukon ei tarvitse säilyttää lukuja järjestyksessä, joten set-rakenteen sijasta voi käyttää myös unordered_set-rakennetta. Tämä on usein hyvä tapa paran-

taa algoritmin käytännön tehokkuutta. Algoritmin toteutus säilyy samana ja vain tietorakenne vaihtuu. Uuden algoritmin aikavaativuus on $O(n)$.

Ratkaisu 3

Vaihtoehtoinen lähestymistapa on käyttää tietorakenteiden sijasta järjestämistä. Järjestetään ensin listat A ja B , minkä jälkeen yhteiset luvut voi löytää käymällä listat rinnakkain läpi. Järjestämisen aikavaativuus on $O(n \log n)$ ja läpikäynnin aikavaativuus on $O(n)$, joten kokonaisaikavaativuus on $O(n \log n)$.

Vertailu

Seuraavassa taulukossa on mittaustuloksia äskeisten algoritmien tehokkuudesta, kun n vaihtelee ja listojen luvut ovat välillä $1 \dots 10^9$:

n	ratkaisu 1	ratkaisu 2	ratkaisu 3
10^6	1,5 s	0,3 s	0,2 s
$2 \cdot 10^6$	3,7 s	0,8 s	0,3 s
$3 \cdot 10^6$	5,7 s	1,3 s	0,5 s
$4 \cdot 10^6$	7,7 s	1,7 s	0,7 s
$5 \cdot 10^6$	10,0 s	2,3 s	0,9 s

Osoittautuu, että järjestämistä käyttävä ratkaisu 3 on selvästi nopeampi kuin joukkoa käyttävät ratkaisut 1 ja 2. Syynä tähän on, että järjestäminen on kevyt operaatio ja se tehdään vain kerran ratkaisun 3 alussa. Ratkaisut 1 ja 2 taas joutuvat pitämään käsittelemään jatkuvasti joukkoa.

Tärkeä havainto on, että vaikka sekä ratkaisun 1 että ratkaisun 3 aikavaativuus on $O(n \log n)$, ratkaisu 3 on todellisuudessa noin 10 kertaa tehokkaampi kuin ratkaisu 1. Lisäksi ratkaisun 2 `unordered_set`-rakenne on tässä tapauksessa noin 5 kertaa tehokkaampi kuin ratkaisun 1 `set`-rakenne.

Luku 5

Täydellinen haku

Täydellinen haku (*complete search*) on yleispätevä tapa ratkaista lähes mikä tahansa ohjelmointitehtävä. Ideana on käydä läpi raa'alla voimalla kaikki mahdolliset tehtävän ratkaisut ja tehtävästä riippuen valita paras ratkaisu tai laskea ratkaisuiden yhteismäärä.

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

5.1 Osajoukkojen läpikäynti

Joukon osajoukkoja ovat kaikki mahdolliset tavat valita osa joukon alkioista. Kun joukossa on n alkioita, sillä on 2^n osajoukkoa. Esimerkiksi 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. Tutustumme seuraavaksi kahteen menetelmään osajoukkojen läpikäyntiin.

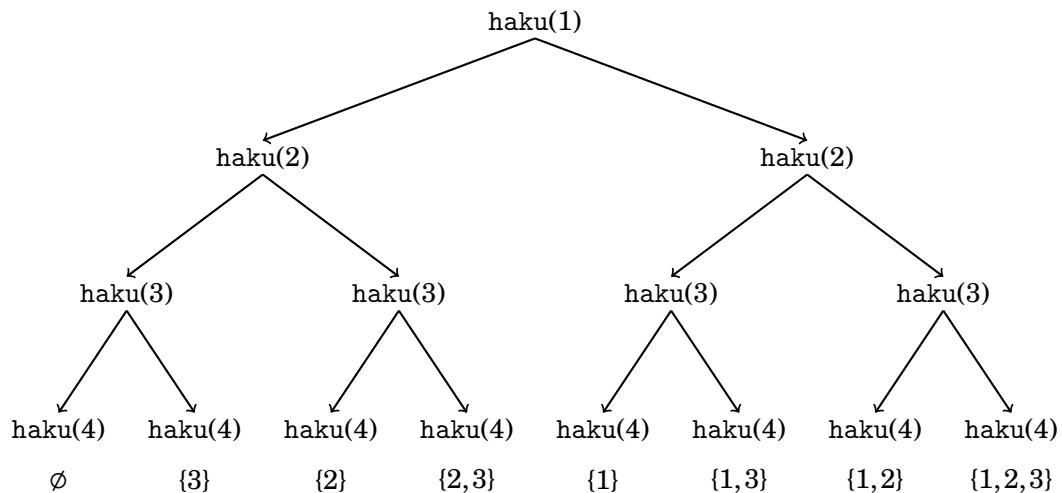
Menetelmä 1

Kätevä tapa käydä läpi osajoukot on käyttää rekursiota. Seuraava funktio haku muodostaa joukon $\{1, 2, \dots, n\}$ osajoukot. Funktio pitää yllä vektoria v , johon se kokoaa osajoukossa olevat luvut. Osajoukkojen muodostus alkaa kutsumalla funktiota `haku(1)`.

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

Funktion parametri k on luku, joka on ehdolla lisättäväksi osajoukkoon seuraavaksi. Joka kutsulla funktio haarautuu kahteen tapaukseen: joko luku k lisätään tai ei lisätä 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 vasen haara jättää luvun pois osajoukosta ja oikea haara lisää sen osajoukkoon.



Menetelmä 2

Toinen tapa käydä osajoukot läpi on hyödyntää kokonaislukujen bittiesitystä. Jokainen n alkion osajoukko voidaan esittää n bitin jonona, joka taas vastaa lukua väliltä $0 \dots 2^n - 1$. Bittiesityksen ykkösbitit ilmaisevat, mitkä joukon alkiot on valittu osajoukkoon.

Tarkastellaan esimerkiksi joukkoa $\{1, 2, 3, 4, 5\}$. Nyt jokaista joukon osajoukkoa vastaa jokin 5 bitin jono. Esimerkiksi osajoukon $\{2, 3, 5\}$ bittiesitys on 01101, jossa bitit 1 ja 4 ovat nollia ja bitit 2, 3, ja 5 ovat ykkösiä.

Seuraava koodi käy läpi n alkion joukon osajoukkojen bittiesitykset:

```

for (int b = 0; b < (1<<n); b++) {
    // käsittele osajoukko b
}

```

Merkintä $1 << n$ on bittisiirto, joka tarkoittaa samaa kuin 2^n .

Seuraava koodi muodostaa jokaisen osajoukon kohdalla vektorin v , joka sisältää osajoukossa olevat luvut. Ne saadaan selville tutkimalla, mitkä bitit ovat ykkösiä osajoukon bittiesityksessä.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> v;
    for (int i = 0; i < n; i++) {
        if (b & (1<<i)) v.push_back(i+1);
    }
}

```

5.2 Permutaatioiden läpikäynti

Joukon permutaatiot ovat alkioiden mahdolliset järjestykset. Kun joukossa on n alkioita, permutaatioita on kaikkiaan $n!$. Esimerkiksi joukon $\{1, 2, 3\}$ permutaatiot ovat $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ ja $(3, 2, 1)$. Tutustumme seuraavaksi kahteen tapaan permutaatioiden läpikäyntiin.

Menetelmä 1

Osajoukkojen tavoin permutaatioita voi muodostaa rekursiivisesti. Seuraava funktio haku käy läpi joukon $\{1, 2, \dots, n\}$ permutaatiot. Funktio muodostaa kunkin permutaation vuorollaan vektoriin v . Permutaatioiden muodostus alkaa kutsumalla funktiota ilman parametreja.

```
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();
        }
    }
}
```

Funktion jokainen kutsu lisää uuden luvun permutaatioon vektoriin v . Taulukko p kertoo, mitkä luvut on jo valittu permutaatioon. Jos $p[k] = 0$, luku k ei ole mukana, ja jos $p[k] = 1$, luku k on mukana. Jos vektorin v koko on sama kuin joukon koko n , permutaatio on tullut valmiiksi.

Menetelmä 2

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 $\{1, 2, \dots, n\}$ 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()));
```

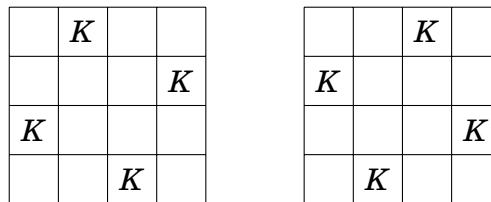
5.3 Peruuttava haku

Peruuttava haku (*backtracking*) aloittaa ratkaisun etsimisen tyhjästä ja laajentaa ratkaisua askel kerrallaan. Joka askeleella haku haarautuu kaikkiin mahdollisiin suuntiin, joihin ratkaisua voi laajentaa. Haaran tutkimisen jälkeen haku peruuttaa takaisin ja jatkaa hakua muihin suuntiin.

Tarkastellaan esimerkkinä seuraavaa tehtävää:

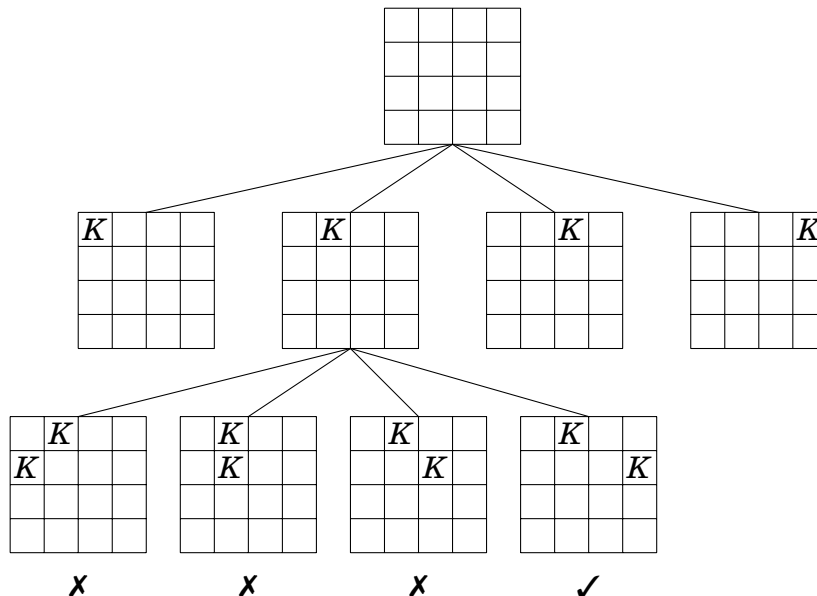
Tehtävä: Montako tapaa on asettaa n kuningatarta $n \times n$ -shakkilaudalle niin, että mitkään kaksi kuningatarta eivät uhkaa toisiaan?

Esimerkiksi kun $n = 4$, mahdolliset ratkaisut ovat seuraavat:



Tehtävän voi ratkaista peruuttavalla haulla muodostamalla ratkaisua rivi kerrallaan. Jokaisella rivillä täytyy valita yksi ruuduista, johon sijoitetaan kuningatar niin, ettei se uhkaa mitään aiemmin lisättyä kuningatarta. Ratkaisu on valmis, kun viimeisellekin riville on lisätty kuningatar.

Esimerkiksi kun $n = 4$, osa peruuttavan haun muodostamasta puusta näyttää seuraavalta:



Kuvan alimmalla tasolla kolme ensimmäistä osaratkaisua eivät kelpaa, koska niissä kuningattaret uhkaavat toisiaan. Sen sijaan neljäs osaratkaisu kelpaa, ja sitä on mahdollista laajentaa loppuun asti kokonaiseksi ratkaisuksi.

Seuraava koodi toteuttaa peruuttavan haun:

```
void haku(int y) {
    if (y == n) {
        c++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (xx[x] || d1[x+y] || d2[x-y+n-1]) continue;
        xx[x] = d1[x+y] = d2[x-y+n-1] = 1;
        haku(y+1);
        xx[x] = d1[x+y] = d2[x-y+n-1] = 0;
    }
}
```

Haku alkaa kutsumalla funktiota `haku(0)`. Laudan koko on muuttujassa n , ja koodi laskee ratkaisuiden määrän muuttujaan c .

Koodi olettaa, että laudan vaaka- ja pystyrivit on numeroitu 0:sta alkaen. Funktio asettaa kuningattaren vaakariville y , kun $0 \leq y < n$. Jos taas $y = n$, yksi ratkaisu on valmis ja funktio kasvattaa muuttujaa c .

Taulukko `xx` pitää kirjaa, millä laudan pystyriveillä on jo kuningatar. Vastaavasti taulukot `d1` ja `d2` pitävät kirjaa vinoriveistä. Tällaisille riveille ei voi laittaa enää toista kuningatarta. Esimerkiksi 4×4 -laudan tapauksessa vinorivit on numeroitu seuraavasti:

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

d1

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

d2

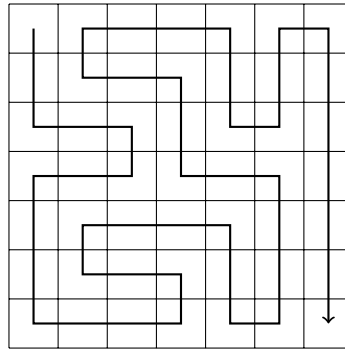
Koodin avulla selviää esimerkiksi, että tapauksessa $n = 8$ on 92 tapaa sijoittaa 8 kuningatarta 8×8 -laudalle. Kun n kasvaa, koodi hidastuu nopeasti, koska ratkaisujen määrä kasvaa räjähdysmäisesti. Tapauksen $n = 16$ laskeminen vie jo noin minuutin nykyaikaisella tietokoneella (14772512 ratkaisua).

5.4 Haun optimointi

Peruuttavaa hakua on usein mahdollista tehostaa huomattavasti erilaisten optimointien avulla. Tarkastellaan esimerkkinä seuraavaa tehtävää:

Tehtävä: Aloitat $n \times n$ -ruudukon vasemmasta yläkulmasta ja tehtäväsi on päästä oikeaan alakulmaan. Saat liikkua joka vuorolla askeleen ylöspäin, alaspäin, vasemmalle tai oikealle. Sinun tulee käydä reitin aikana tasan kerran kussakin ruudussa. Montako erilaista reittiä on olemassa?

Esimerkiksi 7×7 -ruudukossa on 111712 mahdollista reittiä vasemmasta yläkulmasta oikeaan alakulmaan, joista yksi on seuraava:



Keskitymme seuraavaksi nimenomaan tapaukseen 7×7 , koska se on laskennallisesti sopivan haastava. Lähdemme liikkeelle suoraviivaisesta peruuttavaa hakua käyttävästä algoritmista ja teemme siihen pikkuhiljaa optimointeja, jotka nopeuttavat hakua eri tavoin.

Mittaamme jokaisen optimoinnin jälkeen algoritmin suoritusajan sekä rekursiokutsujen yhteismäärän, jotta näemme selvästi, mikä vaikutus kullakin optimoinnilla on haun tehokkuuteen.

Algoritmi

Algoritmin ensimmäisessä versiossa ei ole mitään optimointeja, vaan peruuttava haku käy läpi kaikki mahdolliset tavat muodostaa reitti ruudukon vasemmasta yläkulmasta oikeaan alakulmaan.

- suoritus aika: 483 sekuntia
- rekursiokutsuja: 76 miljardia

Optimointi 1

Reitin ensimmäinen askel on joko alaspäin tai oikealle. Tästä valinnasta seuraavat tilanteet ovat symmetrisiä ruudukon lävistäjän suhteen. Niinpä voimme mennä aina ensin alaspäin ja kertoa lopuksi reittien määrä 2:lla.

- suoritus aika: 244 sekuntia
- rekursiokutsuja: 38 miljardia

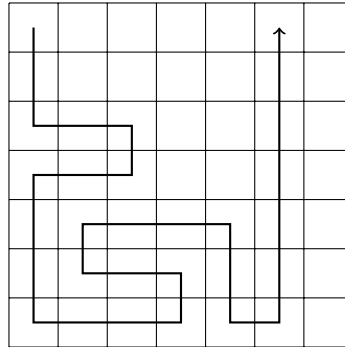
Optimointi 2

Jos reitti menee oikean alakulman ruutuun ennen kuin se on käynyt kaikissa muissa ruuduissa, siitä ei voi mitenkään enää saada kelvollista ratkaisua. Niinpä voimme hylätä haun aikana kaikki tällaiset reitit.

- suoritus aika: 119 sekuntia
- rekursiokutsuja: 20 miljardia

Optimointi 3

Jos reitti osuu seinään niin, että kummallakin puolella on ruutu, jossa reitti ei ole vielä käynyt, ruudukko jakautuu kahteen osaan. Näin on esimerkiksi seuraavassa tilanteessa:



Nyt ei ole enää mahdollista käydä kaikissa ruuduissa, joten voimme hylätä kaikki tällaiset reitit. Tämä optimointi on hyvin hyödyllinen:

- suoritus aika: 1,8 sekuntia
- rekursiokutsuja: 221 miljoonaa

Optimointi 4

Äskeisen optimoinnin ideaa voi yleistää: jos nykyisen ruudun ylä- ja alapuolella on seinä tai aiemmin käyty ruutu sekä vasemmalla ja oikealla on vielä käymätön ruutu (tai päinvastoin), voimme hylätä reitin.

- suoritus aika: 0,6 sekuntia
- rekursiokutsuja: 69 miljoonaa

5.5 Puolivälihaku

Puolivälihaku (*meet in the middle*) on tekniikka, jossa hakutehtävä jaetaan kahteen yhtä suureen osaan. Kumpaankin osaan tehdään erillinen haku, ja lopuksi hakujen tulokset yhdistetään. Puolivälihaun hyötynä on, että sen avulla voi parantaa haun aikavaativuutta.

Tutustumme puolivälihakuun seuraavan tehtävän kautta:

Tehtävä: Annettuna on lista, jossa on n lukua, sekä lisäksi kokonaisluku x . Tehtäväsi on selvittää, voiko listan luvuista valita osajoukon niin, että osajoukon lukujen summa on x .

Tavanomainen ratkaisu tehtävään on käydä kaikki listan alkuioiden osajoukot läpi ja tarkastaa, onko jonkin osajoukon summa x . Tällainen ratkaisu kuluttaa aikaa $O(2^n)$, koska erilaisia osajoukkoja on 2^n . Puolivälihaun avulla on kuitenkin mahdollista luoda tehokkaampi $O(2^{n/2})$ -aikainen ratkaisu.

Ideana on jakaa syötteenä oleva lista kahteen listaan A ja B , joista kumpikin sisältää noin puolet alkioista. Ensimmäinen haku muodostaa kaikki osajoukot listan A luvuista ja laittaa muistiin niiden summat listaan S_A . Toinen haku käsittelee vastaavasti listan B luvut ja laittaa niiden summat listaan S_B .

Tämän jälkeen riittää tarkastaa, onko mahdollista valita yksi luku listasta S_A ja toinen luku listasta S_B niin, että lukujen summa on x . Tämän on mahdollista tarkalleen silloin, kun alkuperäisen listan luvuista saa summan x .

Tarkastellaan esimerkiksi, jossa lista on $\{2, 4, 5, 9\}$ ja $x = 15$. Puolivälihaku muodostaa listat $A = \{2, 4\}$ ja $B = \{5, 9\}$ sekä summalistat $S_A = \{0, 2, 4, 6\}$ ja $S_B = \{0, 5, 9, 14\}$. Summa $x = 15$ on mahdollista muodostaa, koska voidaan valita S_A :sta luku 6 ja S_B :stä luku 9. Tämä vastaa ratkaisua $\{2, 4, 9\}$.

Ratkaisun aikavaativuus on huolellisesti toteutettuna vain $O(2^{n/2})$. Listat S_A ja S_B on kumpikin mahdollista muodostaa ajassa $O(2^{n/2})$ niin, että niiden luvut ovat järjestyksessä. Tämän jälkeen on mahdollista tutkia ajassa $O(2^{n/2})$, voiko lukua x muodostaa valitsemalla kummastakin listasta yksi luku.

Vaikka aikavaativuudet $O(2^n)$ ja $O(2^{n/2})$ muistuttavat toisiaan, niiden ero on merkittävä. Vakiokertoimilla on siis väliä, jos ne esiintyvät potenssin eksponentissa. Esimerkiksi jos $n = 40$, $O(2^n)$ -algoritmin suoritus veisi tunteja aikaa, kun taas $O(2^{n/2})$ valmistuu sekunnin murto-osassa.

Luku 6

Ahneet algoritmit

Ahne algoritmi (*greedy algorithm*) muodostaa ongelman ratkaisun tekemällä joka askeleella sillä hetkellä parhaalta näyttävän valinnan. 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 aina optimaalisen ratkaisun tehtävään. Ahneen algoritmin tulee olla sellainen, että kulloinkin parhaalta näyttävät valinnat tuottavat myös parhaan kokonaisuuden. Tämän perusteleva on usein hankalaa.

6.1 Kolikkotehtävä

Aloitamme ahneisiin algoritmeihin tutustumisen seuraavasta tehtävästä:

Tehtävä: Kolikoiden arvot ovat $\{c_1, c_2, \dots, c_k\}$, 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 $200 + 200 + 100 + 20$.

Ahne algoritmi

Luonteva ahne algoritmi tehtävään 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 kolikolle x pätee, että x :ää pienemmistä kolikoista ei ole mahdollista saada aikaan summaa x tai suurempaa summaa 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.

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 monta luontevaa ahnetta ratkaisua, mutta useimmat niistä eivät tuota aina optimiratkaisua. Tutustumme seuraavaksi kahteen klassiseen aikataulutukseen.

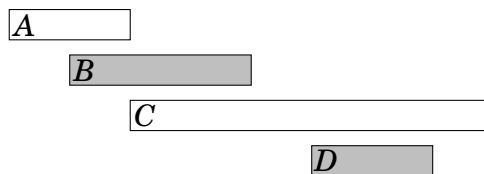
6.2.1 Tapahtumien valinta

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
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

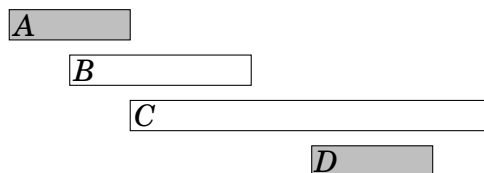
niin on mahdollista osallistua korkeintaan kahteen tapahtumaan. Yksi mahdollisuus on osallistua tapahtumiin *B* ja *D* seuraavasti:



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

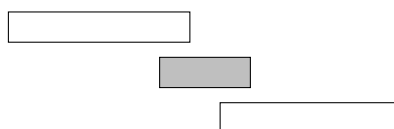
Algoritmi 1

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



ja tuottaa optimiratkaisun.

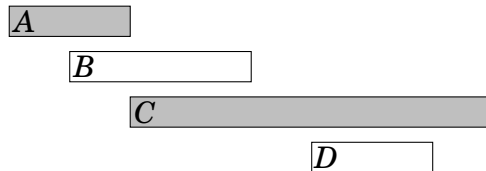
Lyhimpien tapahtumien valinta ei ole kuitenkaan aina toimiva strategia. Algoritmi epäonnistuu esimerkiksi seuraavassa tilanteessa:



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

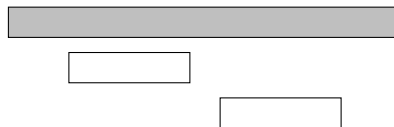
Algoritmi 2

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



ja tuottaa optimiratkaisun.

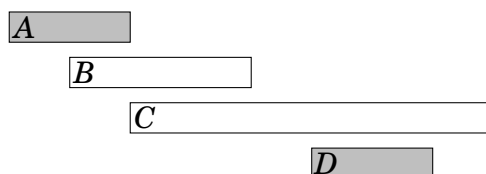
Tämä algoritmi ei kuitenkaan toimi esimerkiksi seuraavassa tilanteessa:



Kun ensimmäisenä alkava tapahtuma valitaan mukaan, mitään muuta tapahtumaa ei ole mahdollista valita. Kuitenkin olisi mahdollista osallistua kahden 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. Algoritmi 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 tapahtuma, jne.

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

6.2.2 Tehtävien järjestys

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

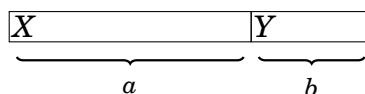
niin optimaalinen ratkaisu on suorittaa tehtävät seuraavasti:



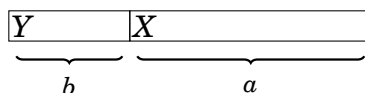
Tässä ratkaisussa C tuottaa 5 pistettä, B tuottaa 0 pistettä, A tuottaa -7 pistettä ja D tuottaa -8 pistettä, joten pistesumma on -10 .

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

6.3.1 Itseisarvosumma

Tehtävä: Annettuna on n lukua a_1, a_2, \dots, a_n . Tehtäväsi on etsiä luku x , joka minimoi summan $|a_1 - x| + |a_2 - x| + \dots + |a_n - x|$.

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 keskimäinen 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 pienentäminen pienentää summaa. Niinpä x 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, \dots, a_n . Tehtäväsi on etsiä luku x , joka minimoi summan $(a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2$.

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 yleisessä tapauksessa 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

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

Ensimmäinen osa ei riipu x :stä, joten sen voi jättää huomiotta. Jäljelle jäävistä osista muodostuu funktio $nx^2 - 2xs$, missä $s = a_1 + a_2 + \dots + a_n$. 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 täydellisen haun toimivuuden ja ahneiden algoritmien tehokkuuden. Dynaamisen ohjelmoinnin käyttäminen edellyttää, että tehtävä jakautuu osaongelmiin, jotka voidaan käsitellä toisistaan riippumattomasti.

Dynaamisella ohjelmoinnilla on kaksi käyttötarkoitusta: (1) optimiratkaisun etsiminen ja (2) ratkaisujen määrän laskeminen. Tutustumme dynaamiseen ohjelmointiin ensin optimiratkaisun etsimisen kautta ja käytämme sitten samaa ideaa ratkaisujen määrän laskemiseen.

Dynaamisen ohjelmoinnin ymmärtäminen on yksi merkkipaalu jokaisen kisakoodarin uralla. Vaikka menetelmän perusidea on yksinkertainen, haasteena on oppia soveltamaan sitä sujuvasti erilaisissa tehtävissä. Tämä luku esittelee joukon perusesimerkkejä, joista on hyvä lähteä liikkeelle.

7.1 Optimiratkaisun etsiminen

Tehtävä: Kolikoiden arvot ovat $\{c_1, c_2, \dots, c_k\}$, 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 ratkaisuihin. 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 $\{c_1, c_2, \dots, c_k\}$, rekursiivinen kaava on

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

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 $c[1], c[2], \dots, c[n]$, 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 ratkaisevasti.

Tässä tehtävässä muistitaulukoksi sopii taulukko

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

jonka 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 . Huomaa kiinnostava piirre aikavaativuudessa: algoritmin tehokkuuteen vaikuttaa, kuinka suuri luku rahamäärä x on. Käytännössä x :n tulee olla niin pieni, että on mahdollista varata sen kokoinen taulukko.

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 $0, 1, \dots, x$. 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 rahamäärälle, mikä kolikko siitä tulee poistaa optimiratkaisussa. 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 Ratkaisujen määrän laskeminen

Tehtävä: Kolikoiden arvot ovat $\{c_1, c_2, \dots, c_k\}$, 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:

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

Ratkaisujen määrän laskeminen tapahtuu melko samalla tavalla kuin optimiratkaisun etsiminen. Erona on, että optimiratkaisun etsivässä rekursiossa valitaan yleensä pienin tai suurin aiempi arvo, kun taas ratkaisujen määrän 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

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

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 täytämällä taulukon d rahamäärille $0 \dots x$:

```
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 ratkaisujen määrä 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 riviin

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

jälkeen rivi

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

7.3 Esimerkkejä

7.3.1 Pisin nouseva alijono

Tehtävä: Taulukossa on n kokonaislukua x_1, x_2, \dots, 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

pin 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 $f(1), f(2), \dots, f(n)$.

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 päättyvä 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 $f(1), f(2), \dots, f(k-1)$ 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.

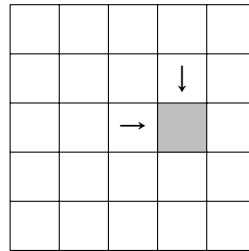
Seuraavassa ruudukossa paras reitti on merkitty harmaalla taustalla:

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) , rekursioiden 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 p_1, p_2, \dots, p_n ja arvot ovat a_1, a_2, \dots, a_n . Tehtäväsi on valita reppuun pakattavat tavarat niin, että painojen summa on enintään x 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 yhteispaino on 12, niin paras ratkaisu on pakata reppuun tavarat B ja D . Niiden yhteispaino $6 + 5 = 11$ ei ylitä rajaa 12 ja arvojen summa on $3 + 3 = 6$, mikä on paras mahdollinen tulos.

Tämä tehtävä on mahdollista ratkaista kahdella eri tavalla dynaamisella ohjelmoinnilla riippuen siitä, tarkastellaanko ongelmaa maksimointina vai minimointina. Käymme seuraavaksi läpi molemmat ratkaisut.

Ratkaisu 1

Maksimointi: Määritellään funktio $f(k, u)$: suurin mahdollinen tavaroiden yhteisarvo, kun reppuun pakataan jokin osajoukko k ensimmäisestä tavarasta niin, että tavaroiden yhteispaino on u . Ratkaisu tehtävään on suurin arvoista $f(n, u)$, kun $0 \leq u \leq x$.

Rekursiivinen kaava funktion laskemiseksi on

$$f(k, u) = \max(f(k-1, u), f(k-1, u - p_k) + a_k),$$

koska viimeinen tavara joko otetaan mukaan tai ei oteta mukaan ratkaisuun. Pohjatapauksina $f(0, 0) = 0$ ja $f(0, u) = -\infty$, kun $u \neq 0$. Ratkaisun aikavaativuus on $O(nx)$.

Ratkaisu 2

Minimointi: Määritellään funktio $f(k, u)$: pienin mahdollinen tavaroiden yhteispaino, kun reppuun pakataan jokin osajoukko k ensimmäisestä tavarasta ja niiden yhteisarvo on u . Ratkaisu tehtävään on suurin arvoista u , kun $0 \leq u \leq s$ ja $f(n, u) \leq x$, missä $s = \sum_{i=1}^n a_i$.

Rekursiivinen kaava funktion laskemiseksi on

$$f(k, u) = \min(f(k-1, u), f(k-1, u - a_k) + p_k),$$

samaan tapaan kuin edellisessä ratkaisussa. Pohjatapauksina $f(0, 0) = 0$ ja $f(0, u) = \infty$, kun $u \neq 0$. Ratkaisun aikavaativuus on $O(ns)$.

Kiinnostava seikka on, että ratkaisuilla on eri aikavaativuus. Ratkaisussa 1 tavaroiden painot vaikuttavat aikavaativuuteen mutta arvot eivät vaikuta. Ratkaisussa 2 vuorostaan tavaroiden arvot vaikuttavat aikavaativuuteen mutta painot eivät vaikuta.

7.3.4 Editointietäisyys

Editointietäisyys (*edit distance*) kuvaa, kuinka kaukana kaksi merkkijonoa ovat toisistaan. Se on pienin määrä editointioperaatioita, joilla ensimmäisen merkkijonon saa muutettua toiseksi. Sallitut operaatiot ovat:

- merkin lisäys (esim. ABC \rightarrow ABCA)
- merkin poisto (esim. ABC \rightarrow AC)
- merkin muutos (esim. ABC \rightarrow ADC)

Esimerkiksi merkkijonojen TALO ja PALLO editointietäisyys on 2, koska voi tehdä ensin operaation TALO \rightarrow TALLO (merkin lisäys) ja sen jälkeen operaation TALLO \rightarrow PALLO (merkin muutos). Tämä on pienin mahdollinen määrä operaatioita, koska selvästikään yksi operaatio ei riitä.

Tutustumme editointietäisyyteen seuraavan tehtävän kautta:

Tehtävä: Annettuna on merkkijono x , jossa on n merkkiä, sekä merkkijono y , jossa on m merkkiä. Mikä on merkkijonojen editointietäisyys ja miten editoinnin voi suorittaa?

Merkkijonojen x ja y editointietäisyys on mahdollista laskea dynaamisella ohjelmoinnilla ajassa $O(nm)$. Ideana on merkitä funktiolla $f(a, b)$ editointietäisyyttä x :n a ensimmäisen merkin sekä y :n b :n ensimmäisen merkin välillä. Osoittautuu, että funktion f avulla saa sekä laskettua tehokkaasti editointietäisyyden että selvitettyä, mitkä ovat tarvittavat editointioperaatiot.

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 meriksi ($c = 1$)

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	3	2	1	2
O	4	4	4	3	2	2

Taulukon oikean alanurkan ruutu kertoo, että merkkijonojen TALO ja PALLO editointietäisyys on 2. Taulukosta pystyy myös lukemaan, miten pienimmän editointietäisyyden voi saavuttaa. Tässä tapauksessa polku on seuraava:

		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	3	2	1	2
O	4	4	4	3	2	2

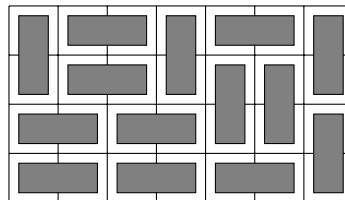
Merkkijonojen PALLO ja TALO viimeinen merkki on sama, joten niiden editointietäisyys on sama kuin merkkijonojen PALL ja TAL. Nyt voidaan poistaa viimeinen L merkkijonosta PAL, mistä tulee yksi operaatio. Editointietäisyys on siis yhden suurempi kuin merkkijonoilla PAL ja TAL, jne.

7.3.5 Laatoitukset

Joskus dynaamisen ohjelmoinnin tila on monimutkaisempi kuin kiinteä yhdistelmä lukuja. Näin on seuraavassa tehtävässä:

Tehtävä: Sinulla on käytettävissäsi rajaton määrä laattoja kokoa 1×2 ja 2×1 , ja tehtäväsi on täyttää niillä $n \times m$ -kokoinen ruudukko. Kuinka monta mahdollista tapaa tähän on?

Esimerkiksi kun ruudukon koko on 4×7 , yksi mahdollinen ratkaisu on



ja ratkaisujen kokonaismäärä on 781.

Yksi tapa ratkaista tehtävä dynaamisella ohjelmoinnilla on käydä läpi ruudukkoa rivi riviltä. Jokainen ratkaisun rivi pelkistyy merkkijonoksi, jossa on m merkkiä joukosta $\{\square, \sqcup, \sqsubset, \sqsupset\}$. Esimerkiksi yllä olevassa ratkaisussa on 4 riviä, jotka vastaavat merkkijonoja

- $\square \sqsubset \square \square \sqsubset \square$,
- $\sqcup \sqsubset \sqcup \square \square \sqcup$,
- $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \square$ ja
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup$.

Tehtävään sopiva dynaamisen ohjelmoinnin funktio on $f(k, x)$, joka laskee, montako tapaa on muodostaa ratkaisu ruudukon ylimmältä riviltä riville k niin, että viimeistä riviä vastaa merkkijono x . Dynaaminen ohjelmointi on mahdollista, koska jokaisen rivin sisältöä rajoittaa vain edellisen rivin sisältö.

Riveistä muodostuva kokonaisratkaisu on kelvollinen, jos ensimmäisellä rivillä ei ole merkkiä \sqcup , viimeisellä rivillä ei ole merkkiä \square ja kaikki peräkkäiset rivit ovat keskenään yhteensopivat. Esimerkiksi rivit $\sqcup \sqsubset \sqcup \square \square \sqcup$ ja $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \square$ ovat yhteensopivat, kun taas rivit $\square \sqsubset \square \sqsubset \square$ ja $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup$ eivät ole yhteensopivat.

Koska rivillä on m merkkiä ja jokaiselle merkille on 4 vaihtoehtoa, erilaisia rivejä on korkeintaan 4^m . Käytännössä tämä määrä on pienempi, koska rivillä

olevaa merkkiä \sqsubset täytyy aina seurata merkki \sqsupset , joten osa riveistä ei voi koskaan esiintyä kelvollisessa ratkaisussa.

Ratkaisun aikavaativuus on $O(n4^{2m})$, koska joka rivillä käydään läpi $O(4^m)$ vaihtoehtoa rivin sisällölle ja jokaista vaihtoehtoa kohden $O(4^m)$ vaihtoehtoa edellisen rivin sisällölle. Käytännössä ruudukko kannattaa kääntää niin päin, että pienempi luku on $m:n$ roolissa, koska $m:n$ suuruus on ratkaiseva ajankäytön kannalta.

Ratkaisua on mahdollista tehostaa parantamalla rivien esitystapaa merkkijonoina. Itse asiassa ainoa seuraavalla rivillä tarvittava tieto on, missä kohtaa riviltä lähtee laattoja alaspäin. Niinpä rivin voikin tallentaa käyttämällä vain merkkejä \sqsubset ja \sqsupset , missä \sqsupset kokoaa yhteen vanhat merkit \sqsubset , \sqsupset ja \sqsupset . Tämän muutoksen ansiosta erilaisia rivejä on vain 2^m ja aikavaativuudeksi tulee $O(n2^{2m})$.

Mainittakoon lopuksi, että laatoitusten määrän laskemiseen on myös yllättävä suora kaava

$$\prod_{a=1}^{\lfloor n/2 \rfloor} \prod_{b=1}^{\lfloor m/2 \rfloor} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right).$$

Tämä kaava on sinänsä hyvin tehokas, koska se laskee laatoitusten määrän ajassa $O(nm)$, mutta käytännön ongelma kaavan käyttämisessä on välitulosten tallentaminen riittävän tarkkoina lukuina.

Luku 8

Tasoitettu analyysi

Monen algoritmin aikavaativuuden pystyy laskemaan suoraan katsomalla algoritmin rakennetta: mitä silmukoita algoritmissa on ja miten monta kertaa niitä suoritetaan. Joskus kuitenkin näin suoraviivainen analyysi ei riitä antamaan todellista kuvaa algoritmin tehokkuudesta.

Tasoitettu analyysi (*amortized analysis*) soveltuu algoritmeihin, joiden osana on jokin operaatio, jonka ajankäyttö vaihtelee. Ideana on tarkastella yksittäisen operaation sijasta kaikkia operaatioita algoritmin aikana ja laskea niiden ajankäytölle yhteinen raja.

8.1 Kaksi osoitinta

Kahden osoittimen tekniikka on taulukon käsittelyssä käytettävä tekniikka, jossa taulukkoa käydään läpi kahden osoittimen avulla. Molemmat osoittimet liikkuvat algoritmin aikana, mutta rajoituksena on, että ne voivat liikkua vain yhteen suuntaan, mikä takaa, että algoritmi toimii tehokkaasti.

Tutustumme seuraavaksi kahden osoittimen tekniikkaan kahden esimerkkitehtävän kautta.

8.1.1 Välin summa

Tehtävä: Annettuna on taulukko, jossa on n positiivista kokonaislukua. Tehtäväsi on tutkia, onko taulukossa väliä, jossa lukujen summa on x .

Ideana on käydä taulukkoa läpi kahden osoittimen avulla, jotka rajaavat välin taulukosta. Joka vuorolla vasen osoitin liikkuu yhden askeleen eteenpäin, kun taas oikea osoitin liikkuu niin kauan eteenpäin kuin summa on enintään x . Jos välin summaksi tulee tarkalleen x , ratkaisu on löytynyt.

Tarkastellaan esimerkkinä algoritmin toimintaa seuraavassa taulukossa, kun tavoitteena on muodostaa summa $x = 8$:

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

Aluksi osoittimet rajaavat taulukosta välin, jonka summa on $1 + 3 + 2 = 6$. Väli ei voi olla tätä suurempi, koska seuraava luku 5 veisi summan yli x :n.

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

↑
↑

Seuraavaksi vasen osoitin siirtyy askeleen eteenpäin. Oikea osoitin säilyy paikallaan, koska muuten summa kasvaisi liian suureksi.

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

↑
↑

Vasen osoitin siirtyy taas askeleen eteenpäin ja tällä kertaa oikea osoitin siirtyy kolme askelta eteenpäin. Muodostuu summa $2 + 5 + 1 = 8$ eli taulukosta on löytynyt väli, jonka lukujen summa on x .

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

↑
↑

Algoritmin toteutus näyttää seuraavalta:

```
int s = 0, b = 0;
for (int a = 1; a <= n; a++) {
    while (b < n && s + t[b+1] <= x) {
        b++;
        s += t[b];
    }
    if (s == x) {
        // ratkaisu löytyi
    }
    s -= t[a];
}
```

Muuttujat a ja b sisältävät vasemman ja oikean osoittimen kohdan. Muuttuja s taas laskee lukujen summan välillä. Joka askeleella a liikkuu askeleen eteenpäin ja b liikkuu niin kauan kuin summa on enintään x .

Algoritmin aikavaativuus riippuu siitä, kauanko while-silmukan suoritus vie aikaa. Tämä vaihtelee, koska oikea osoitin voi liikkua minkä tahansa matkan eteenpäin taulukossa. Kuitenkin oikea osoitin liikkuu *yhteensä* $O(n)$ askelta algoritmin aikana, koska se voi liikkua vain eteenpäin.

Koska sekä vasen että oikea osoitin liikkuvat $O(n)$ askelta algoritmin aikana, algoritmin aikavaativuus on $O(n)$.

8.1.2 Kahden luvun summa

Tehtävä: Annettuna on taulukko, jossa on n kokonaislukua. Tehtäväsi on tutkia, voiko taulukosta valita kaksi lukua niin, että niiden summa on x .

Tämänkin tehtävän voi ratkaista kahden osoittimen tekniikalla, kuitenkin niin, että osoittimet liikkuvat eri suuntiin. Ideana on järjestää ensin taulukon luvut pienimmästä suurimpaan ja sitten käydä taulukkoa läpi kahdella osoittimella, jotka lähtevät liikkelle sen molemmista päistä.

Vasen osoitin aloittaa taulukon alusta ja liikkuu joka vaiheessa askeleen eteenpäin. Oikea osoitin taas aloittaa taulukon lopusta ja peruuttaa niin kauan, kunnes osoitinten määrittämän välin lukujen summa on enintään x . Jos summa on tarkalleen x , ratkaisu on löytynyt.

Tarkastellaan esimerkkinä algoritmin toimintaa seuraavassa taulukossa, kun tavoitteena on muodostaa summa $x = 12$:

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

Seuraavassa on algoritmin aloitustilanne. Lukujen summa on $1 + 10 = 11$, joka on pienempi kuin x :n arvo.

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

↑ ↑

Seuraavaksi vasen osoitin liikkuu askeleen eteenpäin. Oikea osoitin peruuttaa kolme askelta, minkä jälkeen summana on $4 + 7 = 11$.

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

↑ ↑

Sitten vasen osoitin siirtyy jälleen askeleen eteenpäin. Oikea osoitin pysyy paikallaan ja ratkaisu $5 + 7 = 12$ on löytynyt.

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

↑
↑

Algoritmin alussa taulukon järjestäminen vie aikaa $O(n \log n)$. Tämän jälkeen vasen osoitin liikkuu $O(n)$ askelta eteenpäin ja oikea osoitin liikkuu $O(n)$ askelta taaksepäin, mihin kuluu aikaa $O(n)$. Algoritmin kokonaisaikaavaativuus on siis $O(n \log n)$.

Huomaa, että tehtävän voi ratkaista myös toisella tavalla ajassa $O(n \log n)$ binäärihaun avulla. Tässä ratkaisussa jokaiselle taulukon luvulle etsitään binäärihaulla toista lukua niin, että lukujen summa olisi yhteensä x . Binäärihaku suoritetaan n kertaa ja jokainen binäärihaku vie aikaa $O(\log n)$.

8.2 Tietorakenteen operaatiot

Tasoitettun analyysin avulla arvioidaan usein tietorakenteeseen tehtävien operaatioiden määrää. Algoritmin operaatiot voivat jakautua epätasaisesti niin, että useimmat operaatiot tehdään tietyssä algoritmin vaiheessa ja muuten niitä tehdään vähän. Oleellinen asia on operaatioiden kokonaismäärä.

Tutustumme seuraavaksi kahteen klassiseen algoritmiin, joiden tehokkuuden arvioinnissa on hyötyä tasoitetusta analyysistä. Molemmat algoritmit käyvät läpi taulukon ja pitävät yllä tietorakennetta, jossa on ketju taulukon lukuja laskevassa järjestyksessä.

8.2.1 Lähin pienempi edeltäjä

Tehtävä: Annettuna on taulukko, jossa on n lukua. Tehtäväsi on etsiä jokaiselle luvulle sitä lähinnä oleva pienempi luku taulukon alkuosassa tai todeta, että tällaista lukua ei ole olemassa.

Tehokas ratkaisu tehtävään on käydä taulukko läpi alusta loppuun ja pitää samalla yllä ketjua, jonka ensimmäinen luku on käsiteltävä taulukon luku ja jokainen seuraava luku on luvun lähin pienempi edeltäjä. Jos ketjussa on vain yksi luku, käsiteltävällä luvulla ei ole pienempää edeltäjää.


Joka askeleella ketjun alusta poistetaan lukuja niin kauan, kunnes ketjun ensimmäinen luku on pienempi kuin käsiteltävä taulukon luku tai ketju on tyhjä. Tämän jälkeen käsiteltävä luku lisätään ketjun alkuun.

Tarkastellaan esimerkkinä algoritmin toimintaa seuraavassa taulukossa:

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


Aluksi luvut 1, 3 ja 4 liittyvät ketjuun, koska jokainen luku on edellistä suurempi. Siis luvun 4 lähin pienempi edeltäjä on luku 3, jonka lähin pienempi edeltäjä on puolestaan luku 1. Tilanne näyttää tältä:

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



Taulukon seuraava luku 2 on pienempi kuin ketjun kaksi ensimmäistä lukua 4 ja 3. Niinpä luvut 4 ja 3 poistetaan ketjusta, minkä jälkeen luku 2 lisätään ketjun alkuun. Sen lähin pienempi edeltäjä on luku 1:

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



Seuraava luku 5 on suurempi kuin luku 2, joten se lisätään suoraan ketjun alkuun ja sen lähin pienempi edeltäjä on luku 2:

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

Algoritmi jatkaa samalla tavalla taulukon loppuun ja selvittää jokaisen luvun lähimmän pienemmän edeltäjän. Mutta kuinka tehokas algoritmi on?

Algoritmin tehokkuus riippuu siitä, kauanko ketjun käsittelyyn kuluu aikaa yhteensä. Jos uusi luku on suurempi kuin ketjun ensimmäinen luku, se vain lisätään ketjun alkuun, mikä on tehokasta. Joskus taas ketjussa voi olla useita suurempia lukuja, joiden poistaminen vie aikaa.

Oleellista on kuitenkin, että jokainen taulukossa oleva luku liittyy tarkalleen kerran ketjuun ja poistuu korkeintaan kerran ketjusta. Niinpä jokainen luku aiheuttaa $O(1)$ ketjuun liittyvää operaatiota, ja tämän seurauksena algoritmin kokonaisaikavaativuus on $O(n)$.

8.2.2 Liukuvan ikkunan minimi

Tehtävä: Annettuna on taulukko, jossa on n lukua, sekä kokonaisluku k . Taulukon halki kulkee ikkuna, joka käy läpi taulukon kaikki k -kokoiset välit. Tehtäväsi on ilmoittaa jokaisesta ikkunasta pienin välillä oleva luku.

Tämän tehtävän voi ratkaista lähes samalla tavalla kuin edellisen tehtävän. Ideana on pitää yllä ketjua, jonka alussa on ikkunan viimeinen luku ja jossa jokainen luku on edellistä pienempi. Joka vaiheessa ketjun viimeinen luku on ikkunan pienin luku.

Kun liukuva ikkuna liikkuu eteenpäin ja välille tulee uusi luku, ketjusta poistetaan kaikki luvut, jotka ovat uutta lukua suurempia. Tämän jälkeen uusi luku lisätään ketjun alkuun. Lisäksi jos ketjun viimeinen luku ei enää kuulu välille, se poistetaan ketjusta.

Tarkastellaan esimerkkinä, kuinka algoritmi selvittää minimi seuraavassa taulukossa, kun ikkunan koko $k = 4$.


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

Liukuva ikkuna aloittaa matkansa taulukon vasemmasta reunasta. Ensimmäisessä ikkunan sijainnissa pienin luku on 1:

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


Kun ikkuna siirtyy eteenpäin, mukaan tulee luku 3, joka on pienempi kuin luvut 5 ja 4 ketjun alussa. Niinpä luvut 5 ja 4 poistuvat ketjusta ja luku 3 siirtyy sen alkuun. Pienin luku on edelleen 1.

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



Ikkuna siirtyy taas eteenpäin, jonka seurauksena pienin luku 1 putoaa pois ikkunasta. Niinpä se poistetaan ketjun lopusta ja uusi pienin luku on 3. Lisäksi uusi ikkunaan tuleva luku 4 lisätään ketjun alkuun.

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




Seuraavaksi ikkunaan tuleva luku 1 on pienempi kuin kaikki ketjussa olevat luvut. Tämän seurauksena koko ketju tyhjäntyy ja siihen jää vain luku 1:

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

•

Lopuksi ikkuna saapuu viimeiseen sijaintiinsa. Luku 2 lisätään ketjun alkuun, mutta ikkunan pienin luku on edelleen 1.

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



Tässäkin algoritmissa jokainen taulukon luku lisätään ketjuun tarkalleen kerran ja poistetaan ketjusta korkeintaan kerran, joko ketjun alusta tai ketjun lopusta. Niinpä algoritmin kokonaisaikaavaativuus on $O(n)$.

Luku 9

Välikyselyt

Välikysely (*range query*) kohdistuu taulukkoon, jonka sisältönä on lukuja. Kyselyssä annetaan tietty taulukon väli $[a, b]$ ja tehtävänä on laskea haluttu tieto välillä olevista luvuista. Tavallisia välikyselyitä ovat:

- **summakysely**: laske välin $[a, b]$ lukujen summa
- **minimikysely**: etsi pienin luku välillä $[a, b]$
- **maksimikysely**: etsi suurin luku välillä $[a, b]$

Tarkastellaan esimerkkinä seuraavan taulukon väliä $[4, 7]$:

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

Välin $[4, 7]$ summa on $4 + 6 + 1 + 3 = 14$, minimi on 1 ja maksimi on 6.

Helppo tapa vastata välikyselyyn on käydä läpi kaikki välin luvut silmukalla. Esimerkiksi seuraava funktio toteuttaa summakyselyn:

```
int summa(int a, int b) {  
    int s = 0;  
    for (int i = a; i <= b; i++) {  
        s += t[i];  
    }  
    return s;  
}
```

Yllä oleva funktio toteuttaa summakyselyn ajassa $O(n)$, mikä on hidasta, jos taulukko on suuri ja kyselyitä tulee paljon. Tässä luvussa opimme, miten välikyselyitä pystyy toteuttamaan huomattavasti nopeammin.

9.1 Staattisen taulukon kyselyt

Aloitamme yksinkertaisesta tilanteesta, jossa taulukko on staattinen eli sen sisältö ei muutu kyselyiden välillä. Tällöin riittää muodostaa ennen kyselyitä taulukon pohjalta tietorakenne, josta voi selvittää tehokkaasti vastauksen mihin tahansa väliin kohdistuvaan kyselyyn.

9.1.1 Summakysely

Summakyselyyn on mahdollista vastata tehokkaasti muodostamalla taulukosta etukäteen summataulukko, jonka kohdassa k on taulukon välin $[1, k]$ summa. Tämän jälkeen minkä tahansa välin $[a, b]$ summan saa laskettua $O(1)$ -ajassa summataulukkoa käyttäen.

Esimerkiksi taulukon

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

summataulukko on seuraava:

1	2	3	4	5	6	7	8
1	4	8	16	22	23	27	29

Seuraava koodi muodostaa taulukosta t summataulukon s ajassa $O(n)$:

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

Tämän jälkeen summakyselyyn voi vastata ajassa $O(1)$ seuraavasti:

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

Ideana on laskea välin $[a, b]$ summa laskemalla ensin välin $[1, b]$ summa ja poistamalla siitä sitten välin $[1, a - 1]$ summa. Summataulukosta riittää hakea kaksi arvoa ja aikaa kuluu vain $O(1)$. Huomaa, että 1-indeksoinnin ansiosta yllä oleva toteutus toimii myös tapauksessa $a = 1$, kunhan $s[0] = 0$.

Tarkastellaan esimerkiksi väliä $[4, 7]$:

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

Välin $[4, 7]$ summa on $8 + 6 + 1 + 4 = 19$. Tämän saa laskettua tehokkaasti summataulukosta etsimällä välien $[1, 3]$ ja $[1, 7]$ summat:

1	2	3	4	5	6	7	8
1	4	8	16	22	23	27	29

Välin $[4, 7]$ summa on siis $27 - 8 = 19$.

Kaksiulotteinen taulukko

Summataulukon idean voi yleistää myös kaksiulotteiseen taulukkoon, jolloin summataulukosta 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:

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

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.

9.1.2 Minimikysely

Myös minimikyselyyn on mahdollista vastata $O(1)$ -ajassa sopivan esikäsittelyn avulla, joskin tämä on vaikeampaa kuin summakyselyssä. Huomaa, että minimikysely ja maksimikysely on mahdollista toteuttaa aina samalla tavalla, joten riittää keskittyä minimikyselyn toteutukseen.

Ideana on laskea etukäteen taulukon jokaiselle 2^k -kokoiselle välille, mikä on kyseisen välin minimi. Esimerkiksi taulukosta

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

lasketaan seuraavat minimi:

väli	koko	minimi	väli	koko	minimi	väli	koko	minimi
[1,1]	1	1	[1,2]	2	1	[1,4]	4	1
[2,2]	1	3	[2,3]	2	3	[2,5]	4	3
[3,3]	1	4	[3,4]	2	4	[3,6]	4	1
[4,4]	1	8	[4,5]	2	6	[4,7]	4	1
[5,5]	1	6	[5,6]	2	1	[5,8]	4	1
[6,6]	1	1	[6,7]	2	1	[1,8]	8	1
[7,7]	1	4	[7,8]	2	2			
[8,8]	1	2						

Taulukon 2^k -välien määrä on $O(n \log n)$, koska jokaisesta taulukon kohdasta alkaa $O(\log n)$ väliä. Kaikkien 2^k -välien minimi pystytään laskemaan ajassa $O(n \log n)$, koska jokainen 2^k -väli muodostuu kahdesta 2^{k-1} välistä ja 2^k -välin minimi on pienempi 2^{k-1} -välien minimeistä.

Tämän jälkeen minkä tahansa välin $[a, b]$ minimin saa laskettua $O(1)$ -ajassa miniminä kahdesta 2^k -välistä, missä $k = \lfloor \log_2(b - a + 1) \rfloor$. Ensimmäinen väli alkaa kohdasta a ja toinen väli päättyy kohtaan b . Parametri k on valittu niin, että kaksi 2^k -kokoista väliä kattaa koko välin $[a, b]$.

Tarkastellaan esimerkiksi väliä $[2, 7]$:

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

Välin $[2, 7]$ pituus on 6 ja $\lfloor \log_2(6) \rfloor = 2$. Niinpä välin minimin saa selville kahden 4-pituksen välin minimistä. Välit ovat $[2, 5]$ ja $[4, 7]$:

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

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

Välin $[2, 5]$ minimi on 3 ja välin $[4, 7]$ minimi on 1. Tämän seurauksena välin $[2, 7]$ minimi on pienempi näistä eli 1.

Mainittakoon, että $O(1)$ -aikaiset minimikyselyt pystyy toteuttamaan myös niin, että esikäsitteily vie aikaa vain $O(n)$ eikä $O(n \log n)$. Tämä on kuitenkin selvästi vaikeampaa, eikä sillä ole merkitystä kisakoodauksessa.

9.2 Binääri-indeksipuu

Binääri-indeksipuu¹ (*binary indexed tree*) on summataulukkoa muistuttava tietorakenne, joka toteuttaa kaksi operaatiota: taulukon välin $[a, b]$ summakysely sekä taulukon kohdassa k olevan luvun päivitys. Kummankin operaation aika-vaativuus on $O(\log n)$.

Binääri-indeksipuun etuna summataulukkoon verrattuna on, että taulukkoa pystyy päivittämään tehokkaasti summakyselyiden välissä. Summataulukossa tämä ei olisi mahdollista, vaan koko summataulukko tulisi muodostaa uudestaan $O(n)$ -ajassa taulukon päivityksen jälkeen.

Rakenne

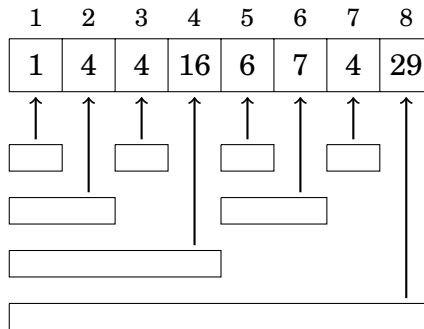
Binääri-indeksipuu on taulukko, jonka kohdassa k on kohtaan k päättyvän välin lukujen summa alkuperäisessä taulukossa. Välin pituus on suurin 2:n potenssi, jolla k on jaollinen. Esimerkiksi jos $k = 6$, välin pituus on 2, koska 6 on jaollinen 2:lla mutta ei ole jaollinen 4:llä.

¹Tämä tietorakenne tunnetaan myös nimellä Fenwick-puu (*Fenwick tree*).

Esimerkiksi taulukkoa

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

vastaava binääri-indeksipuu on seuraava:

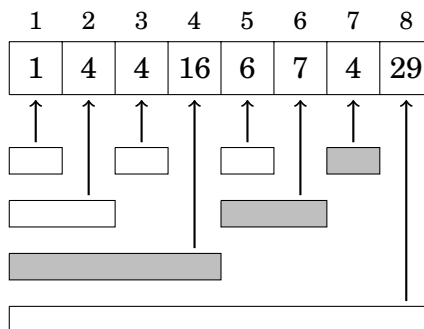


Esimerkiksi binääri-indeksipuun kohdassa 6 on luku 7, koska välin $[5, 6]$ lukujen summa on $6 + 1 = 7$.

Summakysely

Binääri-indeksipuun perusoperaatio on välin $[1, k]$ summan laskeminen, missä k on mikä tahansa taulukon kohta. Tällaisen summan pystyy muodostamaan aina laskemalla yhteen puussa olevia välien summia.

Esimerkiksi välin $[1, 7]$ summa muodostuu seuraavista summista:

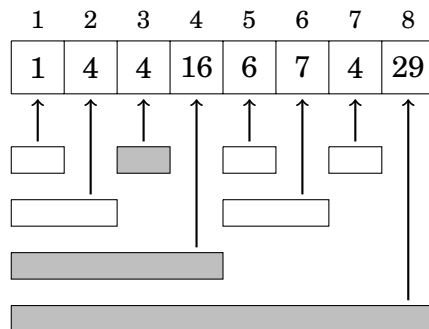


Välin $[1, 7]$ summa on siis $16 + 7 + 4 = 27$. Binääri-indeksipuun rakenteen ansiosta jokainen summaan kuuluva väli on eripituinen. Niinpä summa muodostuu aina $O(\log n)$ välin summasta.

Summataulukon tavoin binääri-indeksipuusta voi laskea tehokkaasti mikä tahansa taulukon välin summan, koska välin $[a, b]$ summa saadaan vähentämällä välin $[1, b]$ summasta välin $[1, a - 1]$ summa. Aikavaativuus on edelleen $O(\log n)$, koska riittää laskea kaksi $[1, k]$ -välin summaa.

Taulukon päivitys

Kun taulukon kohdassa k oleva luku muuttuu, tämä vaikuttaa useaan binääri-indeksipuussa olevaan summaan. Esimerkiksi jos kohdassa 3 oleva luku muuttuu, seuraavat välien summat muuttuvat:



Myös tässä tapauksessa muutos kohdistuu $O(\log n)$ kohtaan binääri-indeksi-
puussa, koska kaikki välit, joihin muutos vaikuttaa, ovat eripituisia.

Toteutus

Binääri-indeksipuun operaatiot on mahdollista toteuttaa lyhyesti ja tehokkaasti bittien käsittelyn avulla. Oleellinen bittioperaatio on $k \& -k$, joka eristää luvusta k viimeisenä olevan ykkösbitin. Esimerkiksi $6 \& -6 = 2$, koska luku 6 on bittimuodossa 110 ja luku 2 on bittimuodossa 010.

Osoittautuu, että summan laskemisessa binääri-indeksipuun kohtaa k tulee muuttaa joka askeleella niin, että siitä poistetaan luku $k \& -k$. Vastaavasti taulukon päivityksessä kohtaa k tulee muuttaa joka askeleella niin, että siihen lisätään luku $k \& -k$.

Seuraavat funktiot olettavat, että binääri-indeksipuu on tallennettu taulukon b ja se muodostuu kohdista $1 \dots n$.

Funktio `summa` laskee välin $[1, k]$ summan:

```
int summa(int k) {
    int s = 0;
    while (k >= 1) {
        s += b[k];
        k -= k & -k;
    }
    return s;
}
```

Funktio `lisaa` kasvattaa taulukon kohtaa k arvolla x :

```
void lisaa(int k, int x) {
    while (k <= n) {
        b[k] += x;
        k += k & -k;
    }
}
```

Kummankin yllä olevan funktion aikavaativuus on $O(\log n)$, koska jokainen käsiteltävä taulukon väli on eripituinen ja yksittäinen bittioperaatio riittää siirtymään seuraavaan väliin.

9.3 Segmenttipuu

Segmenttipuu (*segment tree*) on tietorakenne, jonka operaatiot ovat taulukon välin $[a, b]$ välikysely sekä kohdan k arvon päivitys. Segmenttipuun avulla voi toteuttaa summakyselyn, minimikyselyn ja monia muitakin kyselyitä niin, että kummankin operaation aikavaativuus on $O(\log n)$.

Segmenttipuun etuna binääri-indeksipuuhun verrattuna on, että se on yleisempi tietorakenne. Binääri-indeksipuulla voi toteuttaa vain summakyselyn, mutta segmenttipuu sallii muitakin kyselyitä. Toisaalta segmenttipuu vie enemmän muistia ja on hieman vaikeampi toteuttaa kuin binääri-indeksipuu.

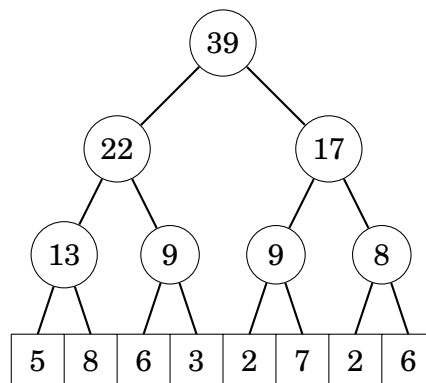
Rakenne

Segmenttipuussa on $2n - 1$ solmua niin, että alimmalla tasolla on n solmua, jotka kuvaavat taulukon sisällön, ja ylemmillä tasoilla on välikyselyihin tarvittavaa tietoa. Segmenttipuun sisältö riippuu siitä, mikä välikysely puun tulee toteuttaa. Oletamme aluksi, että välikysely on tuttu summakysely.

Esimerkiksi taulukkoa

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

vastaa seuraava segmenttipuu:



Jokaisessa segmenttipuun solmussa on tietoa 2^k -kokoisesta välistä taulukossa. Tässä tapauksessa solmussa oleva arvo kertoo, mikä on taulukon lukujen summa solmua vastaavalla välillä. Kunkin solmun arvo saadaan laskemalla yhteen solmun alapuolella vasemmalla ja oikealla olevien solmujen arvot.

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.

Välikysely

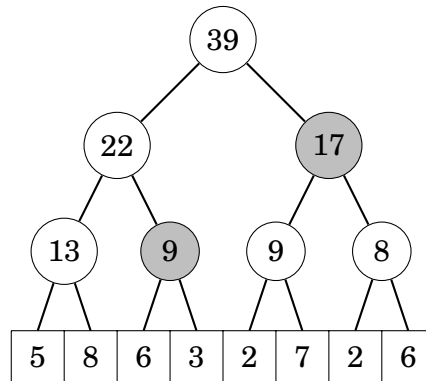
Segmenttipuussa vastaus välikyselyyn lasketaan väliin kuuluvista solmuista, jotka ovat mahdollisimman korkealla puussa. Jokainen solmu antaa vastauk-

sen väliin kuuluvalla osavälillä, ja vastaus kyselyyn selviää yhdistämällä segmenttipuusta saadut osavälilejät koskevat tiedot.

Tarkastellaan esimerkiksi seuraavaa taulukon väliä:

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

Lukujen summa välillä $[3, 8]$ on $6 + 3 + 2 + 7 + 2 + 6 = 26$. Segmenttipuusta summa saadaan laskettua seuraavien osasummien avulla:



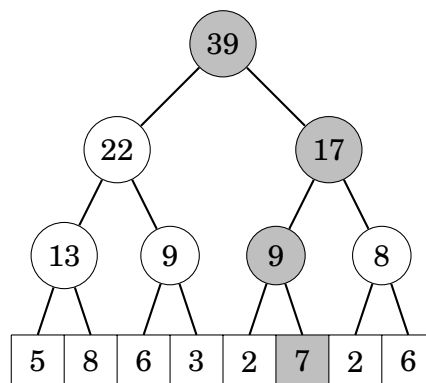
Taulukon välin summaksi tulee osasummista $9 + 17 = 26$.

Kun vastaus välikyselyyn lasketaan mahdollisimman korkealla segmenttipuussa olevista solmuista, väliin kuuluu enintään kaksi solmua jokaiselta segmenttipuun tasolta. Tämän ansiosta välikyselyssä tarvittavien solmujen yhteismäärä on vain $O(\log n)$.

Taulukon päivitys

Kun taulukossa oleva 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, mitkä solmut segmenttipuussa muuttuvat, jos taulukon luku 7 muuttuu.

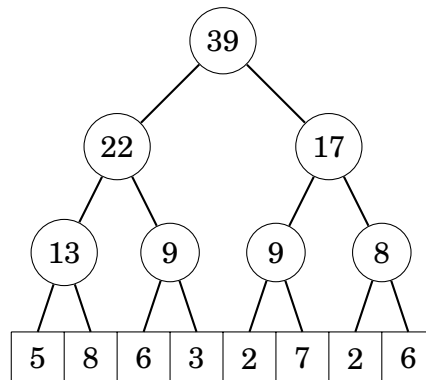


Polku segmenttipuun pohjalta huipulle muodostuu aina $O(\log n)$ solmusta, joten taulukon arvon muuttuminen vaikuttaa $O(\log n)$ solmuun puussa.

Puun tallennus

Tavallinen tapa tallentaa segmenttipuu muistiin on luoda taulukko, jossa on $2n - 1$ alkia. Taulukon kohdassa 1 on puun ylimmän solmun arvo, kohdat 2 ja 3 sisältävät seuraavan tason solmujen arvot, jne. Segmenttipuun alin taso eli alkuperäisen taulukon sisältö tallennetaan kohdasta n eteenpäin.

Esimerkiksi segmenttipuun



voi tallentaa taulukkoon seuraavasti:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Tätä tallennustapaa käyttäen puussa liikkuminen on helppoa. Kohdassa k olevasta solmusta ylempi solmu on kohdassa $\lfloor k/2 \rfloor$ ja alemmat solmut ovat kohdissa $2k$ (vasen) ja $2k + 1$ (oikea). Lisäksi parillisten kohtien solmut ovat vasemmalla ja parittomien kohtien solmut oikealla.

Toteutus

Tarkastellaan seuraavaksi välikyselyn ja päivityksen toteutusta segmenttipuuhun. Seuraavat funktiot olettavat, että segmenttipuu on tallennettu $2n - 1$ -kokoiseen taulukkoon p edellä kuvatulla tavalla.

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

```
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;
}
```

Funktio aloittaa summan laskeminen segmenttipuun pohjalta ja liikkuu askel kerrallaan ylemmille tasoille. Funktio laskee välin summan muuttujaan s yhdistämällä puussa olevia osasummia. Välin reunalla oleva osasumma lisätään summaan aina silloin, kun se ei kuulu ylemmän tason osasummaan.

Funktio lisää kasvattaa kohdan k arvoa x :llä:

```
void lisaa(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];
    }
}
```

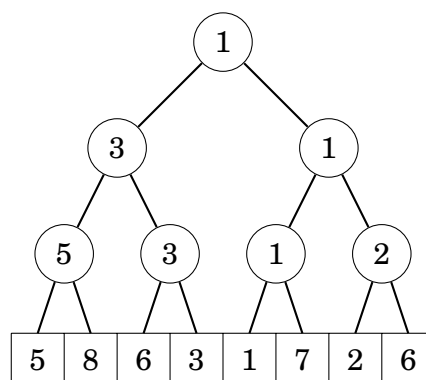
Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasumat puun huipulle asti. Taulukon p indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k + 1$.

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

Muut kyselyt

Segmenttipuu mahdollistaa summan lisäksi minkä tahansa välikyselyn, jossa 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 seuraavan segmenttipuun avulla voi laskea taulukon välien minimejä:

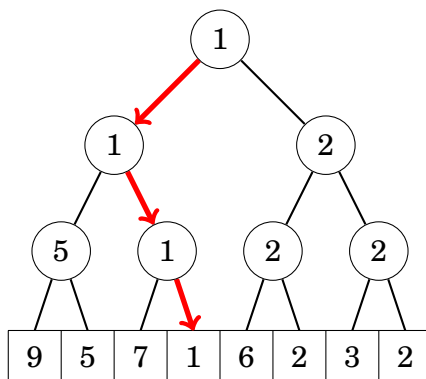


Tässä segmenttipuussa jokainen puun solmu kertoo, mikä on pienin luku sen alapuolella olevassa taulukon osassa. Segmenttipuun ylin luku on pienin luku koko taulukon alueella. Puun toteutus on samanlainen kuin summan laskemisessa, mutta joka kohdassa pitää laskea summan sijasta lukujen minimi.

Binäärihaku puussa

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

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



9.4 Lisäteknikoita

9.4.1 Indeksien pakkaus

Tämän luvun tietorakenteiden rajoituksena on, että ne on rakennettu taulukon päälle ja alkiot on indeksoitu kokonaisluvuin $1, 2, \dots, n$. Tästä seuraa ongelmia, jos tarvittavat indeksit ovat suuria. Esimerkiksi indeksin 10^9 käyttäminen vaatisi, että taulukossa olisi 10^9 alkioita, mikä ei ole realistista.

Tätä rajoitusta on kuitenkin mahdollista kiertää usein käyttämällä indeksien pakkausta. Se tarkoittaa, että n indeksia jaetaan uudestaan niin, että ne ovat välillä $1, 2, \dots, n$. Tämä on mahdollista silloin, kun 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 kyselyitä voi tehdä melko tavallisesti indeksien pakkauksesta huolimatta.

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

$$\begin{aligned} p(8) &= 1 \\ p(555) &= 2 \\ p(10^9) &= 3 \end{aligned}$$

9.4.2 Välin muuttaminen

Tähän asti tavoitteemme on ollut toteuttaa tehokkaasti välikyselyjä ja muuttaa yksittäisiä taulukon arvoja. Tarkastellaan lopuksi käänteistä tilannetta, jossa pitääkin muuttaa välejä ja kysellä yksittäisiä arvoja. Tavoitteena on nyt toteuttaa operaatio, joka kasvattaa kaikkia välin $[a, b]$ arvoja x :llä.

Yllättävää kyllä, voimme käyttää tämän luvun tietorakenteita 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 $2 \dots 5$ 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 $a \dots b$ 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, joten voimme käyttää tuttuja menetelmiä tässäkin tilanteessa.

Hankalampi tilanne on, jos samaan aikaan pitää pystyä sekä kysymään tietoa väleiltä että muuttamaan välejä. Myöhemmin luvussa 28 tulemme näkemään, että tämäkin on mahdollista.

Luku 10

Bittien käsittely

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

10.1 Luvun bittiesitys

Luvun 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 luvun bittiesityksen bittien määrä on kiinteä ja riippuu käytetystä tietotyypistä. Esimerkiksi C++:n int-tyyppi on tavallisesti 32-bittinen, jolloin int-luku tallennetaan 32 bittinä. Tällöin esimerkiksi luvun 43 bittiesitys int-lukuna on seuraava:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1

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

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 seuraava:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1

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 pyörrähtää 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 pyörrähtää ympäri luvusta $2^{31} - 1$ lukuun -2^{31} .

10.2 Bittiooperaatiot

And-operaatio

And-operaatio $x \& y$ tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa molemmissa luvuissa x ja y on ykkösbitti. Esimerkiksi $22 \& 26 = 18$, koska

$$\begin{array}{rcl} & 10110 & (22) \\ \& & 11010 & (26) \\ \hline = & 10010 & (18) \end{array}$$

And-operaation avulla voi tarkastaa luvun parillisuuden, koska $x \& 1 = 0$, jos luku on parillinen, ja $x \& 1 = 1$, jos luku on pariton.

Or-operaatio

Or-operaatio $x \mid y$ tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa ainakin toisessa luvuista x ja y on ykkösbitti. Esimerkiksi $22 \mid 26 = 30$, koska

$$\begin{array}{rcl} & 10110 & (22) \\ \mid & 11010 & (26) \\ \hline = & 11110 & (30) \end{array}$$

Xor-operaatio

Xor-operaatio $x \wedge y$ tuottaa luvun, jossa on ykkösbitti niissä kohdissa, joissa tarkalleen toisessa luvuista x ja y on ykkösbitti. Esimerkiksi $22 \wedge 26 = 12$, koska

$$\begin{array}{rcl} & 10110 & (22) \\ \wedge & 11010 & (26) \\ \hline = & 01100 & (12) \end{array}$$

Not-operaatio

Not-operaatio $\sim x$ tuottaa luvun, jossa kaikki x :n bitit on muutettu käänteisiksi. Operaatiolle pätee kaava $\sim x = -x - 1$, esimerkiksi $\sim 29 = -30$.

Not-operaation toiminta bittitasolla riippuu siitä, montako bittiä luvun bittiesityksessä on, koska operaatio vaikuttaa kaikkiin luvun bitteihin. Esimerkiksi 32-bittisenä int-lukuna tilanne on seuraava:

```
x    = 29  000000000000000000000000000011101
~x   = 30  111111111111111111111111111110010
```

Bittisiirrot

Vasen bittisiirto $x \ll k$ tuottaa luvun, jossa luvun x bittejä on siirretty k askelta vasemmalle (luvun loppuun tulee k nollabittiä). Oikea bittisiirto $x \gg k$ tuottaa puolestaan luvun, jossa luvun x bittejä on siirretty k askelta oikealle (luvun lopusta lähtee pois k viimeistä bittiä).

Esimerkiksi $14 \ll 2 = 56$, koska 14 on bitteinä 1110, josta tulee bittisiirron jälkeen 111000 eli 56. Vastaavasti $49 \gg 3 = 6$, koska 49 on bitteinä 110001, josta tulee bittisiirron jälkeen 110 eli 6.

Huomaa, että vasen bittisiirto $x \ll k$ vastaa luvun x kertomista 2^k :lla ja oikea bittisiirto $x \gg k$ vastaa luvun x jakamista 2^k :lla alaspäin pyöristäen.

Bittien käsittely

Luvun bitit indeksoidaan oikealta vasemmalle nollasta alkaen. Luvussa $1 \ll k$ on tarkalleen yksi ykkösbitti kohdassa k , joten sen avulla voi käsitellä muiden lukujen yksittäisiä bittejä.

Luvun x bitti k on ykkösbitti, jos $x \& (1 \ll k) = (1 \ll k)$. Lauseke $x \mid (1 \ll k)$ asettaa luvun x bitin k ykköseksi, lauseke $x \& \sim(1 \ll k)$ asettaa luvun x bitin k nollassa ja lauseke $x \wedge (1 \ll k)$ muuttaa luvun x bitin k käänteiseksi.

Lauseke $x \& (x-1)$ muuttaa luvun x viimeisen ykkösbitin nollassa, ja lauseke $x \& -x$ nollassa 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 $x \& (x-1) = 0$.

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

Nämä funktiot käsittelevät int-lukuja, mutta funktioista on myös long long -versiot, joiden lopussa on päätte ll.

Seuraava koodi esittelee funktioiden käyttöä:

```
int x = 5328; // 000000000000000000001010011010000
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 Joukon bittiesitys

Joukon $\{0, 1, 2, \dots, n-1\}$ jokaista osajoukkoa vastaa n -bittinen luku, jossa ykkös-bitit ilmaisevat, mitkä alkiot ovat mukana osajoukossa. Esimerkiksi joukkoa $\{1, 3, 4, 8\}$ vastaa bittiesitys 100011010 eli luku $2^8 + 2^4 + 2^3 + 2^1 = 282$.

Joukon bittiesitys vie vähän muistia, koska tieto kunkin alkion kuulumisesta osajoukkoon vie vain yhden bitin tilaa. Lisäksi bittimuodossa tallennettua joukkoa on tehokasta käsitellä bittioperaatioilla.

10.3.1 Joukon käsittely

Seuraavan koodin muuttuja x sisältää joukon $\{0, 1, 2, \dots, 31\}$ osajoukon. Koodi lisää luvut 1, 3, 4 ja 8 joukkoon ja tulostaa joukon sisällön.

```
// x on tyhjä joukko
int x = 0;
// lisätään luvut 1, 3, 4 ja 8 joukkoon
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
// tulostetaan joukon sisältö
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
cout << "\n";
```

Koodin tulostus on seuraava:

```
1 3 4 8
```

Nyt joukko-operaatiot voi toteuttaa bittioperaatioilla:

- $a \& b$ on joukkojen a ja b leikkaus $a \cap b$ (tämä sisältää alkiot, jotka ovat kummassakin joukossa)
- $a \mid b$ on joukkojen a ja b yhdiste $a \cup b$ (tämä sisältää alkiot, jotka ovat ainakin toisessa joukossa)

- $a \& (\sim b)$ on joukkojen a ja b erotus $a \setminus b$ (tämä sisältää alkiot, jotka ovat joukossa a mutta eivät joukossa b)

Seuraava koodi muodostaa joukkojen $\{1, 3, 4, 8\}$ ja $\{3, 6, 8, 9\}$ yhdisteen:

```
// joukko {1,3,4,8}
int x = (1<<1)+(1<<3)+(1<<4)+(1<<8);
// joukko {3,6,8,9}
int y = (1<<3)+(1<<6)+(1<<8)+(1<<9);
// joukkojen yhdiste
int z = x|y;
// tulostetaan yhdisteen sisältö
for (int i = 0; i < 32; i++) {
    if (z&(1<<i)) cout << i << " ";
}
cout << "\n";
```

Koodin tulostus on seuraava:

```
1 3 4 6 8 9
```

10.3.2 Osajoukkojen läpikäynti

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

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

Seuraava koodi käy läpi osajoukot, joissa on k alkia:

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

Seuraava koodi käy läpi bittiesitystä x vastaavan joukon osajoukot:

```
int b = 0;
do {
    // osajoukon käsittely
} while (b=b-x&x);
```

Yllä olevien koodien tavoin tämä koodi käy osajoukot läpi bittiesityksen suuruusjärjestyksessä.

10.4 Dynaaminen ohjelmointi

10.4.1 Permutaatioista osajoukoiksi

Dynaamisen ohjelmoinnin avulla on usein mahdollista muuttaa permutaatioiden läpikäynti osajoukkojen läpikäynniksi. Tällöin dynaamisen ohjelmoinnin tilana on joukon osajoukko sekä mahdollisesti muuta tietoa.

Tekniikan hyötynä on, että n -alkioisen joukon permutaatioiden määrä ($n!$) on selvästi suurempi kuin osajoukkojen määrä (2^n). Esimerkiksi jos $n = 20$, niin $n! = 2432902008176640000$, kun taas $2^n = 1048576$. Niinpä sopivilla n :n arvoilla permutaatioita ei ehdi käydä läpi mutta osajoukot ehtii käydä läpi.

Tutustumme tekniikkaan seuraavan tehtävän kautta:

Tehtävä: Montako permutaatiota voit muodostaa luvuista $\{0, 1, \dots, n-1\}$ niin, että missään kohdassa ei ole kahta peräkkäistä lukua? Esimerkiksi kun $n = 4$, ratkaisuja on 2: $(1, 3, 0, 2)$ ja $(2, 0, 3, 1)$.

Merkitään $f(x, k)$:llä, monellako tavalla osajoukon x luvut voi järjestää niin, että viimeinen luku on k ja missään kohdassa ei ole kahta peräkkäistä lukua. Esimerkiksi $f(\{0, 1, 3\}, 1) = 1$, koska voidaan muodostaa permutaatio $(0, 3, 1)$, ja $f(\{0, 1, 3\}, 3) = 0$, koska 0 ja 1 eivät voi olla peräkkäin alussa.

Funktion f avulla ratkaisu tehtävään on summa

$$\sum_{i=0}^{n-1} f(\{0, 1, \dots, n-1\}, i).$$

Dynaamisen ohjelmoinnin tilat voi tallentaa seuraavasti:

```
long long 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];
            }
        }
    }
}
```

Muuttujassa b on osajoukon bittiesitys, ja osajoukon luvuista muodostettu permutaatio on muotoa (\dots, j, i) . 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 :

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

10.4.2 Osajoukkojen määrät

Dynaamisen ohjelmoinnin avulla voi ratkaista myös seuraavan tehtävän:

Tehtävä: Tarkastellaan n -alkioisen joukon osajoukkoja. Jokaista osajoukkoa x vastaa arvo $c(x)$. Tehtäväsi on jokaiselle osajoukolle x summa

$$s(x) = \sum_{y \subset x} c(y)$$

eli bittimuodossa ilmaistuna

$$s(x) = \sum_{y \& x = y} c(y).$$

Seuraavassa on esimerkki funktioiden arvoista, kun $n = 3$:

x	$c(x)$	$s(x)$
000	2	2
001	0	2
010	1	3
011	3	6
100	0	2
101	4	6
110	2	5
111	0	12

Esimerkiksi $s(110) = c(000) + c(010) + c(100) + c(110) = 7$.

Tehtävä on mahdollista ratkaista ajassa $O(2^n n)$ laskemalla arvoja funktiolle $f(x, k)$: mikä on lukujen $c(y)$ summa, missä y :n saa x :stä muuttamalla halutulla tavalla bittien $0, 1, \dots, k$ joukossa ykkösbittejä nollabiteiksi. Tämän funktion avulla ilmaistuna $s(x) = f(x, n - 1)$.

Funktion f voi laskea rekursiivisesti seuraavasti:

$$f(x, k) = \begin{cases} c(x) & \text{jos } k = -1 \\ f(x, k - 1) & \text{jos } x\text{:n bitti } k \text{ on } 0 \\ f(x, k - 1) + f(x \wedge (1 \ll k), k - 1) & \text{jos } x\text{:n bitti } k \text{ on } 1 \end{cases}$$

Pohjatapauksena $f(x, -1) = c(x)$, koska mitään bittejä ei saa muokata. Muuten jos kohdan k bitti on nolla, se säilyy nollana, ja jos kohdan k bitti on ykkönen, se joko säilyy ykkösenä tai muuttuu nollaksi.

Seuraava koodi laskee kaikki funktion s arvot taulukkoon s olettaen, että funktion c arvot ovat taulukossa c .

```
for (int b = 0; b < (1<<n); b++) s[b] = c[b];
for (int k = 0; k < n; k++) {
    for (int b = 0; b < (1<<n); b++) {
        if (b&(1<<k)) s[b] += s[b^(1<<k)];
    }
}
```

Koodi laskee ensin kaikki arvot funktiolle $f(x, 0)$, sitten kaikki arvot funktiolle $f(x, 1)$, jne.

Osa II

Verkkoalgoritmit

Luku 11

Verkkojen perusteet

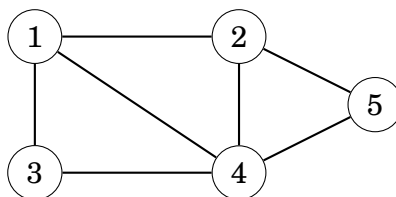
Monen ohjelmointitehtävän voi ratkaista tulkitsemalla tehtävän verkko-ongelmana ja käyttämällä sopivaa verkkoalgoritmia. Tyypillinen esimerkki verkosta on tieverkosto, jonka rakenne muistuttaa luonnostaan verkkoa. Joskus taas verkko kätkeytyy syvemmälle ongelmaan ja sitä voi olla vaikeaa huomata.

Tässä kirjan osassa tutustumme verkkojen käsittelyyn liittyviin tekniikoihin ja kisakoodauksessa keskeisiin verkkoalgoritmeihin. Aloitamme aiheeseen perehtymisen käymällä läpi verkkoihin liittyviä käsitteitä sekä erilaisia tapoja pitää verkkoa muistissa algoritmeissa.

11.1 Käsitteitä

Verkko (*graph*) muodostuu solmuista (*node* tai *vertex*) ja niiden välisistä kaarista (*edge*). Merkitsemme tässä kirjassa verkon solmujen määrää muuttujalla n ja kaarten määrää muuttujalla m . Lisäksi numeroimme verkon solmut kokonaisluvuin $1, 2, \dots, n$.

Esimerkiksi seuraavassa verkossa on 5 solmua ja 7 kaarta:

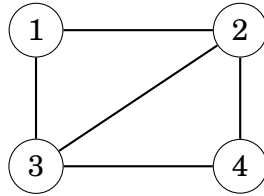


Polku (*path*) on solmusta a solmuun b johtava reitti, joka kulkee verkon kaaria pitkin. Polun pituus (*length*) on kaarten määrä polulla. Esimerkiksi yllä olevassa verkossa polkuja solmusta 1 solmuun 5 ovat:

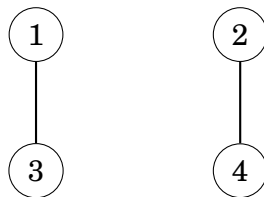
- $1 \rightarrow 2 \rightarrow 5$ (pituus 2)
- $1 \rightarrow 4 \rightarrow 5$ (pituus 2)
- $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ (pituus 3)
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ (pituus 3)
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5$ (pituus 4)

Yhtenäisyys

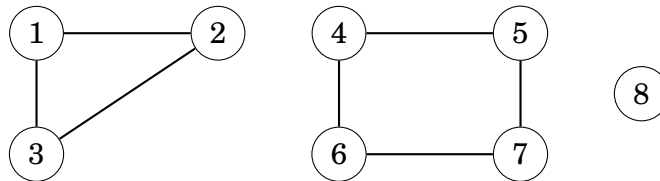
Verkko on yhtenäinen (*connected*), jos siinä on polku mistä tahansa solmusta mihin tahansa solmuun. Esimerkiksi seuraava verkko on yhtenäinen:



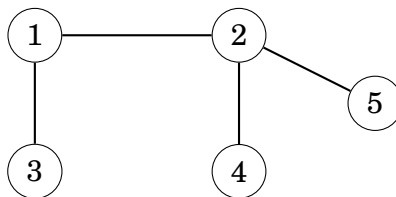
Seuraava verkko taas ei ole yhtenäinen, koska esimerkiksi solmusta 1 ei ole polkua solmuun 2.



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

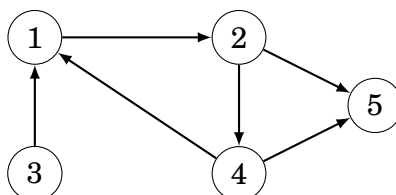


Puu (*tree*) on yhtenäinen verkko, jossa on n solmua ja $n - 1$ kaarta. Siinä minkä tahansa kahden solmun välillä on yksikäsitteinen polku. Esimerkiksi seuraava verkko on puu:



Kaarten suunnat

Verkko on suunnattu (*directed*), jos verkon kaaria pystyy kulkemaan vain niiden merkittyy suuntaan. Esimerkiksi seuraava verkko on suunnattu:



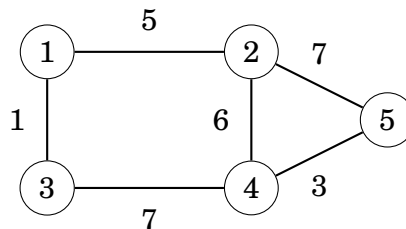
Yllä olevassa verkossa solmusta 3 on polku kaikkiin muihin verkon solmuihin. Esimerkiksi solmusta 3 pääsee solmuun 5 pääsee polkua $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$. Sen sijaan solmusta 5 ei lähde polkua mihinkään muuhun solmuun.

Suunnattu verkko on vahvasti yhtenäinen (*strongly connected*), jos mistä tahansa solmusta on polku mihin tahansa toiseen solmuun.

Sykli (*cycle*) on polku, jonka ensimmäinen ja viimeinen solmu on sama. Esimerkiksi yllä olevassa verkossa on sykli $1 \rightarrow 2 \rightarrow 4 \rightarrow 1$. Jos verkossa ei ole yhtään sykliä, se on sykliton (*acyclic*).

Kaarten painot

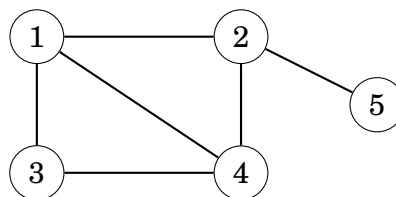
Painotetussa (*weighted*) verkossa jokaiseen kaareen liittyy paino. Tavallinen tulkinta on, että painot kuvaavat kaarien pituuksia. Seuraavassa on esimerkki painotetusta verkosta:



Painotetussa verkossa polun pituus on sen kaarten painojen summa. Esimerkiksi polun $1 \rightarrow 2 \rightarrow 5$ pituus on $5 + 7 = 12$ ja polun $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ pituus on $1 + 7 + 3 = 11$. Jälkimmäinen polku on lyhin polku solmusta 1 solmuun 5.

Naapurit ja asteet

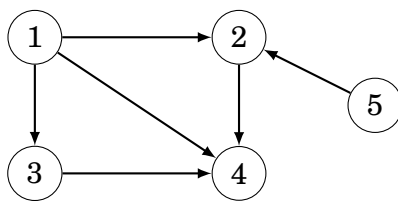
Kaksi solmua ovat naapureita (*neighbor*), jos ne ovat vierekkäin eli niiden välillä on kaari. Solmun aste (*degree*) on sen naapurien määrä. Esimerkiksi seuraavassa verkossa solmun 2 naapurit ovat 1, 4 ja 5, joten sen aste on 3.



Verkon solmujen asteiden summa on $2m$, missä m on kaarten määrä. Tämä johtuu siitä, että jokainen kaari lisää kahden solmun astetta yhdellä. Niinpä solmujen asteiden summa on aina parillinen.

Verkko on säännöllinen (*regular*), jos jokaisen solmun aste on vakio d . Verkko on täydellinen (*complete*), jos jokaisen solmun aste on $n - 1$ eli verkossa on kaikki mahdolliset kaaret solmujen välillä.

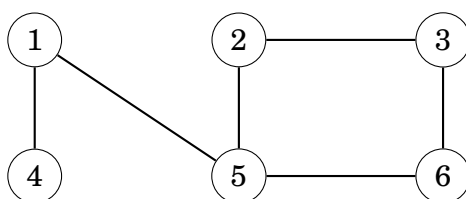
Suunnatussa verkossa lähtöaste (*outdegree*) on solmusta lähtevien kaarten määrä ja tuloaste (*indegree*) on solmuun tulevien kaarten määrä. Esimerkiksi seuraavassa verkossa solmun 2 lähtöaste on 1 ja tuloaste on 2.



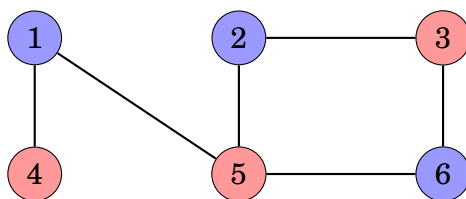
Väritykset

Verkon värityksessä (*coloring*) jokaiselle solmulle valitaan tietty väri niin, että millään kahdella vierekkäisellä solmulla ei ole samaa väriä. Verkko on kaksijakoinen (*bipartite*), jos kaksi väriä riittää sen värittämiseen.

Esimerkiksi verkko



on kaksijakoinen, koska sen voi värittää näin:

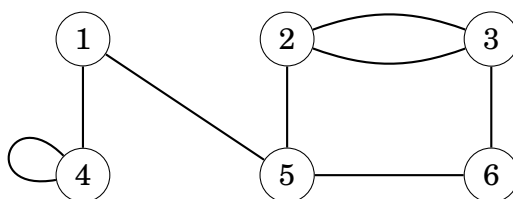


Osoittautuu, että verkko on kaksijakoinen tarkalleen silloin, kun siinä ei ole syklä, johon kuuluu pariton määrä solmuja.

Yksinkertaisuus

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

Esimerkiksi verkko



ei ole yksinkertainen, koska solmusta 4 on kaari itseensä ja solmujen 2 ja 3 välillä on kaksi kaarta.

11.2 Verkko muistissa

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

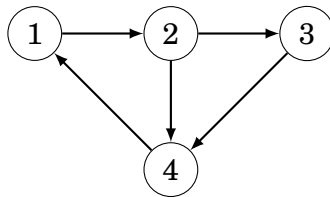
11.2.1 Vieruslistat

Tavallisin tapa pitää verkkoa muistissa on luoda jokaisesta solmusta vieruslista (*adjacency list*), joka sisältää kaikki solmut, joihin solmusta pystyy siirtymään kaarta pitkin. Vieruslistaesitys on tavallisin verkon esitysmuoto ja useimmat algoritmit pystyy toteuttamaan tehokkaasti sitä käyttäen.

Kätevä tapa tallentaa verkon vieruslistaesitys on luoda taulukko, jossa jokainen alkio on vektori:

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

Ideana on, että solmun s vieruslista löytyy kohdasta $v[s]$. Vakio N on valittu niin suureksi, että kaikki solmut mahtuvat taulukkoon. Esimerkiksi verkon



voi tallentaa seuraavasti:

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

Jos verkko on suuntaamaton, sen voi tallentaa samalla tavalla, mutta silloin jokainen kaari lisätään kumpaankin suuntaan.

Painotetun verkon tapauksessa rakennetta voi laajentaa näin:

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

Nyt vieruslistalla on pareja, joiden ensimmäinen jäsen on kaaren kohdesolmu ja toinen jäsen on kaaren paino. Esimerkiksi verkon



voi tallentaa seuraavasti:

```
v[1].push_back({2,5});  
v[2].push_back({3,7});  
v[2].push_back({4,6});  
v[3].push_back({4,5});  
v[4].push_back({1,2});
```

Vieruslistaesityksen etuna on, että sen avulla on nopeaa selvittää, mihin solmuihin tietystä solmusta pääsee kulkemaan. Esimerkiksi seuraava silmukka käy läpi kaikki solmut, joihin pääsee solmusta s :

```
for (auto u : v[s]) {  
    // käsittele solmu u  
}
```

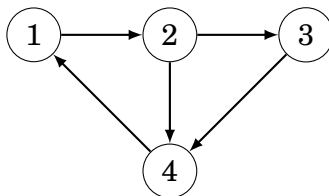
11.2.2 Vierusmatriisi

Vierusmatriisi (*adjacency matrix*) on kaksiulotteinen taulukko, joka kertoo jokaisesta mahdollisesta kaaresta, onko se mukana verkossa. Vierusmatriisista on nopeaa tarkistaa, onko kahden solmun välillä kaari. Toisaalta matriisi vie paljon tilaa, jos verkko on suuri.

Vierusmatriisi tallennetaan taulukkona

```
int v[N][N];
```

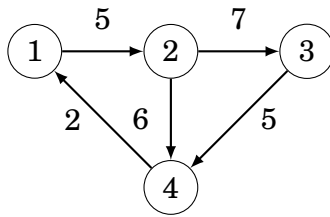
jossa arvo $v[a][b]$ ilmaisee, onko kaari solmusta a solmuun b mukana verkossa. Yksinkertaisin tapa luoda vierusmatriisi on käyttää arvoja 1 (mukana) ja 0 (ei mukana). Tällöin esimerkiksi verkkoa



vastaa seuraava vierusmatriisi:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Jos verkko on painotettu, vierusmatriisiesitystä voi laajentaa luontevasti niin, että matriisissa kerrotaan kaaren paino, jos kaari on olemassa. Tätä esitystapaa käyttäen esimerkiksi verkkoa



vastaa seuraava vierusmatriisi:

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

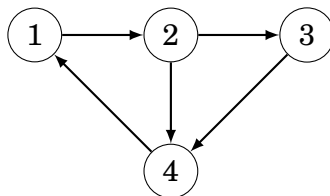
11.2.3 Kaarilista

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.

Kaarilistan voi tallentaa vektoriin

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

jossa jokaisessa solmussa on parina kaaren alku- ja loppusolmu. Tällöin esimerkiksi verkon



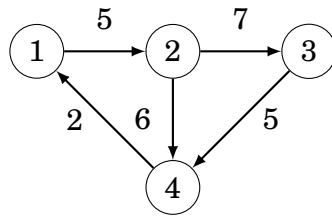
voi tallentaa seuraavasti:

```
v.push_back({1,2});
v.push_back({2,3});
v.push_back({2,4});
v.push_back({3,4});
v.push_back({4,1});
```

Painotetun verkon tapauksessa rakennetta voi laajentaa esimerkiksi näin:

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

Ideana on, että listalla on pareja, joiden ensimmäinen jäsen sisältää parina kaaren alku- ja loppusolmun, ja toinen jäsen on kaaren paino. Nyt verkon



voi tallentaa seuraavasti:

```
v.push_back({{1,2},5});  
v.push_back({{2,3},7});  
v.push_back({{2,4},6});  
v.push_back({{3,4},5});  
v.push_back({{4,1},2});
```

Luku 12

Verkon läpikäynti

Tässä luvussa tutustumme syvyyshakuun ja leveyshakuun, jotka ovat keskeisiä menetelmiä verkon läpikäyntiin. Molemmat algoritmit lähtevät liikkeelle tietyistä verkon solmista ja käyvät läpi kaikki solmut, joihin aloitussolmista pääsee. Algoritmien erona on, missä järjestyksessä ne kulkevat verkossa.

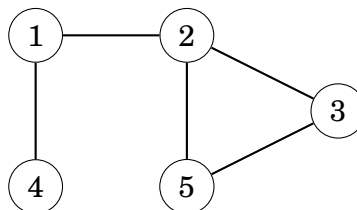
12.1 Syvyyshaku

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

Syvyyshaku etenee verkossa syvyysuuntaisesti eli kulkee eteenpäin verkossa niin kauan kuin vastaan tulee uusia solmuja. Tämän jälkeen haku perääntyy kokeilemaan muita suuntia. Algoritmi pitää kirjaa vierailemistaan solmuista, jotta se käsittelee kunkin solmun vain kerran.

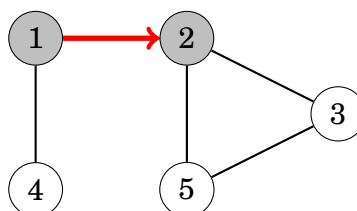
Toiminta

Tarkastellaan syvyysshaun toimintaa seuraavassa verkossa:

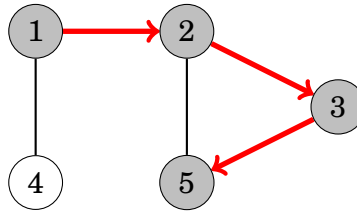


Syvyyshaku voi lähteä liikkeelle mistä tahansa solmista, mutta oletetaan nyt, että haku lähtee liikkeelle solmista 1.

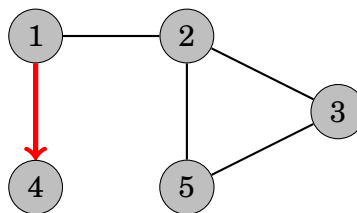
Solmun 1 naapurit ovat solmut 2 ja 4, joista haku etenee ensin solmuun 2:



Tämän jälkeen haku etenee vastaavasti solmuihin 3 ja 5:



Solmun 5 naapurit ovat 2 ja 3, mutta haku on käynyt jo molemmissa, joten on aika 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.

Toteutus

Syvyyshaku on yleensä mukavinta toteuttaa rekursiolla. Seuraava funktio `dfs` suorittaa syvyyshaun sille parametrina annetusta solmusta lähtien. Funktio olettaa, että verkko on tallennettu vieruslistoina taulukkoon

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

ja pitää lisäksi yllä taulukkoa

```
int z[N];
```

joka kertoo, missä solmuissa haku on käynyt. Alussa taulukon jokainen arvo on 0, ja kun haku saapuu solmuun s , sen kohdalle merkitään luku 1. Funktion toteutus on seuraavanlainen:

```
void dfs(int s) {
    if (z[s]) return;
    z[s] = 1;
    // solmun s käsittely tähän
    for (auto u: v[s]) {
        dfs(u);
    }
}
```

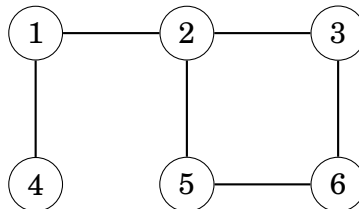

12.2 Leveyshaku

Leveyshaku (*breadth-first search* eli *BFS*) 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.

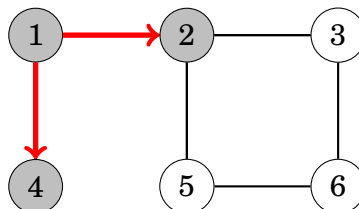
Leveyshakua voi ajatella niin, että se käy solmuja läpi kerros kerrallaan. Ensin haku käy läpi solmut, joihin pääsee yhdellä kaarella alkusolmusta. Tämän jälkeen vuorossa ovat solmut, joihin pääsee kahdella kaarella alkusolmusta jne. Sama jatkuu, kunnes uusia käsiteltäviä solmuja ei enää ole.

Toiminta

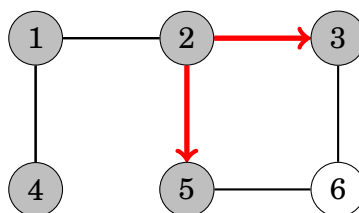
Tarkastellaan leveyshaun toimintaa seuraavassa verkossa:



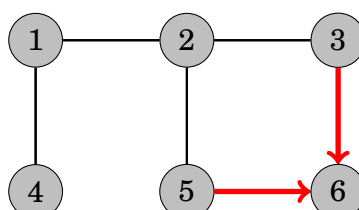
Oletetaan jälleen, että haku alkaa solmusta 1. Haku etenee ensin kaikkiin solmuihin, joihin pääsee alkusolmusta:



Seuraavaksi haku etenee solmuihin 3 ja 5:



Viimeisenä haku etenee solmuun 6:



Leveyshaun tuloksena selviää 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 syvyysshaun tavoin $O(n + m)$, missä n on solmujen määrä ja m on kaarten määrä.

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 solmusta s lähtien. Koodi olettaa, että verkko on tallennettu vieruslistoina, ja pitää yllä jonoa

```
queue<int> q;
```

joka sisältää solmut käsittelyjärjestyksessä. Koodi lisää aina uudet vastaan tulevat solmut jonon perään ja ottaa seuraavaksi käsiteltävän solmun jonon alusta, minkä ansiosta solmut käsitellään tasoittain alkusolmusta lähtien.

Lisäksi koodi käyttää taulukoita

```
int z[N], e[N];
```

niin, että taulukko z sisältää tiedon, missä solmuissa haku on käynyt, ja taulukkoon e lasketaan lyhin etäisyys alkusolmusta kaikkiin verkon solmuihin. Toteutuksesta tulee seuraavanlainen:

```
z[s] = 1; e[s] = 0;
q.push(s);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // solmun s käsittely tähän
    for (auto u : v[s]) {
        if (z[u]) continue;
        z[u] = 1; e[u] = e[s]+1;
        q.push(u);
    }
}
```

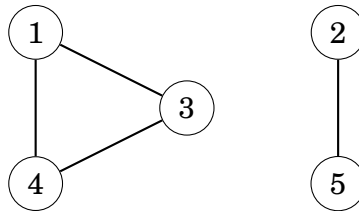
12.3 Sovelluksia

Verkon läpikäynnin avulla saa selville monia asioita verkon rakenteesta. Läpikäynnin voi yleensä aina toteuttaa joko syvyyshaulla tai leveyshaulla, mutta käytännössä syvyyshaku on parempi valinta, koska sen toteutus on helpompi. Oletamme seuraavaksi, että käsiteltävänä on suuntaamaton verkko.

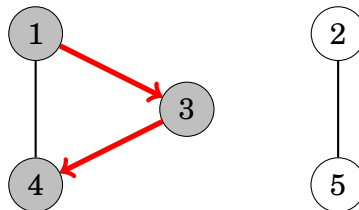
Yhtenäisyys

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.

Esimerkiksi verkossa



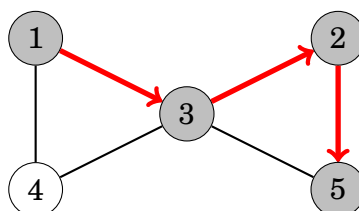
solmusta 1 alkava syvyyshaku löytää seuraavat solmut:



Koska syvyyshaku ei pääse kaikkiin solmuihin, tämä tarkoittaa, että verkko ei ole yhtenäinen. Vastaavalla tavalla voi etsiä myös verkon komponentit käymällä solmut läpi ja aloittamalla uuden syvyysshaun aina, jos käsiteltävä solmu ei kuulu vielä mihinkään komponenttiin.

Syklin etsiminen

Verkossa on sykli, jos jonkin komponentin läpikäynnin aikana tulee vastaan solmu, jonka naapuri on jo käsitelty ja solmuun ei ole saavuttu kyseisen naapurin kautta. Esimerkiksi verkossa



on sykli, koska tultaessa solmusta 2 solmuun 5 havaitaan, että naapurina oleva solmu 3 on jo käsitelty. Niinpä verkossa täytyy olla solmun 3 kautta kulkeva sykli. Tällainen sykli on esimerkiksi $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

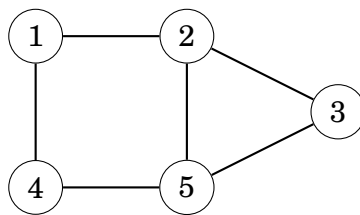
Syklin olemassaolon voi myös päätellä laskemalla, montako solmua ja kaarta komponentissa on. Jos komponentissa on c solmua ja siinä ei ole sykliä, niin siinä on oltava tarkalleen $c - 1$ kaarta. Jos kaaria on c tai enemmän, niin komponentissa on varmasti sykli.

Kaksijakoisuus

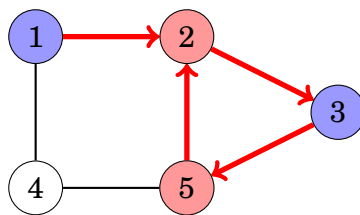
Verkko on kaksijakoinen, jos sen solmut voi värittää kahdella värillä niin, että kahta samanväristä solmua ei ole vierekkäin. On yllättävän helppoa selvittää verkon läpikäynnin avulla, onko verkko kaksijakoinen.

Ideana on värittää alkusolmu siniseksi, sen kaikki naapurit punaiseksi, niiden kaikki naapurit siniseksi, jne. Jos jossain vaiheessa ilmenee ristiriita (saman solmun tulisi olla sekä sininen että punainen), verkko ei ole kaksijakoinen. Muuten verkko on kaksijakoinen ja yksi väritys on muodostunut.

Esimerkiksi verkko



ei ole kaksijakoinen, koska läpikäynti solmusta 1 alkaen aiheuttaa seuraavan ristiriidan:



Tässä vaiheessa havaitaan, että sekä solmun 2 että solmun 5 väri on punainen, vaikka solmut ovat vierekkäin verkossa, joten verkko ei ole kaksijakoinen.

Tämä algoritmi on luotettava tapa selvittää verkon kaksijakoisuus, 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.

Huomaa, että yleensä ottaen on vaikeaa selvittää, voiko verkon solmut värittää k värillä niin, ettei missään kohtaa ole vierekkäin kahta samanväristä solmua. Edes tapaukseen $k = 3$ ei tunneta mitään tehokasta algoritmia, vaan kyseessä on NP-täydellinen ongelma.

Luku 13

Lyhimmät polut

Lyhimmän polun etsiminen alkusolmusta loppusolmuun on keskeinen verkko-ongelma, joka esiintyy usein käytännön tilanteissa. Esimerkiksi tieverkostossa tyypillinen ongelma on selvittää, mikä on lyhin reitti kahden kaupungin välillä, kun tiedossa ovat kaupunkien väliset tiet ja niiden pituudet.

Jos verkon kaarilla 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.

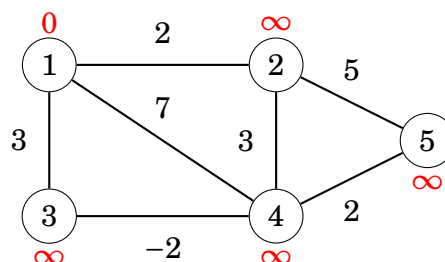
13.1 Bellman-Fordin algoritmi

Bellman-Fordin algoritmi etsii lyhimmän polun alkusolmusta kaikkiin muihin verkon solmuihin. Algoritmi toimii kaikenlaisissa verkoissa, kunhan verkossa ei ole sykliä, jonka kaarten yhteispaino on negatiivinen. Jos verkossa on negatiivinen sykli, algoritmi huomaa tilanteen.

Algoritmi pitää yllä etäisyysarvioita alkusolmusta kaikkiin muihin verkon solmuihin. Alussa alkusolmun etäisyysarvio on 0 ja muiden solmujen etäisyysarvio on ääretön. Algoritmi parantaa arvioita etsimällä verkosta kaaria, jotka lyhentävät polkuja, kunnes mitään arviota ei voi enää parantaa.

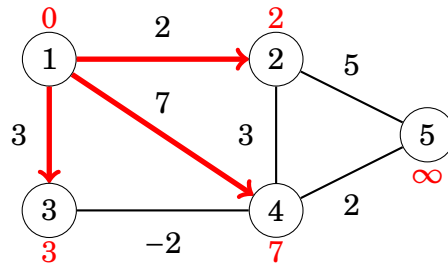
Toiminta

Tarkastellaan Bellman-Fordin algoritmin toimintaa seuraavassa verkossa:

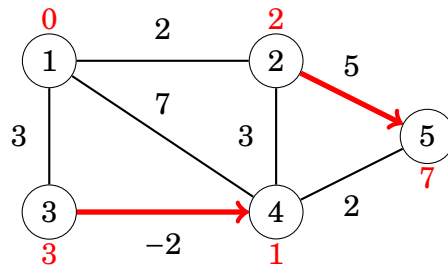


Verkon jokaiseen solmun viereen on merkitty etäisyysarvio. Alussa alkusolmun etäisyysarvio on 0 ja muiden solmujen etäisyysarvio on ääretön (∞).

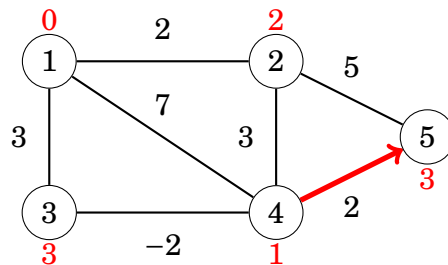
Algoritmin etsii verkosta kaaria, jotka parantavat etäisyysarvioita. Aluksi kaikki solmusta 0 lähtevät kaaret parantavat arvioita:



Sitten kaaret $2 \rightarrow 5$ ja $3 \rightarrow 4$ parantavat arvioita:

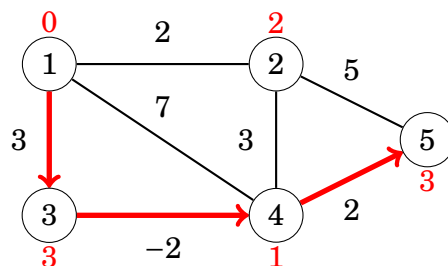


Lopuksi tulee vielä yksi parannus:



Tämän jälkeen mikään kaari ei paranna etäisyysarvioita. Tämä tarkoittaa, että etäisyydet ovat lopulliset, eli joka solmussa on nyt pienin etäisyys alkusolmusta kyseiseen solmuun.

Esimerkiksi pienin etäisyys 3 solmusta 1 solmuun 5 toteutuu käyttämällä seuraavaa reittiä:



Toteutus

Seuraava Bellman-Fordin algoritmin toteutus etsii lyhimät polut solmusta x kaikkiin muihin verkon solmuihin. Koodi olettaa, että verkko on tallennettuna vieruslistoina taulukossa

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

niin, että parissa on ensin kaaren kohdesolmu ja sitten kaaren paino.

Algoritmi muodostuu $n - 1$ kierroksesta, joista jokaisella algoritmi käy läpi kaikki verkon kaaret ja koettaa parantaa etäisyysarvioita. Algoritmi laskee taulukkoon e etäisyyden solmusta x kuhunkin verkon solmuun. Koodissa oleva alkuarvo 10^9 kuvastaa ääretöntä.

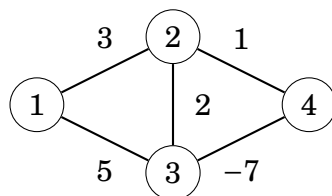
```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (int a = 1; a <= n; a++) {
        for (auto b : v[a]) {
            e[b.first] = min(e[b.first], e[a]+b.second);
        }
    }
}
```

Algoritmin aikavaativuus on $O(nm)$, koska se muodostuu $n - 1$ kierroksesta ja käy läpi jokaisen kierroksen aikana kaikki m kaarta. Jos verkossa ei ole negatiivista sykliä, kaikki etäisyysarviot ovat lopulliset $n - 1$ kierroksen jälkeen, koska jokaisessa lyhimässä polussa on enintään $n - 1$ kaarta.

Käytännössä kaikki lopulliset etäisyysarviot saadaan usein laskettua selvästi alle $n - 1$ kierroksessa, joten mahdollinen tehostus algoritmiin on lopettaa heti, kun mikään etäisyysarvio ei parane kierroksen aikana.

Negatiivinen sykli

Bellman-Fordin algoritmin avulla voi myös tarkastaa, onko verkossa sykliä, jonka pituus on negatiivinen. Esimerkiksi verkossa



on negatiivinen sykli $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$, jonka pituus on -4 .

Jos verkossa on negatiivinen sykli, sen kautta kulkevaa polkua voi lyhentää äärettömästi toistamalla negatiivista sykliä uudestaan ja uudestaan, minkä vuoksi lyhimmän polun käsite ei ole mielekäs.

Negatiivisen syklin voi tunnistaa Bellman-Fordin algoritmilla suorittamalla algoritmia n kierrosta. Jos viimeinen kierros parantaa jotain etäisyysarviota,

verkossa on negatiivinen sykli. Huomaa, että tässä mikään solmuista ei ole alkusolmu ja algoritmi etsii negatiivista sykliä koko verkon alueelta.

SPFA-algoritmi

SPFA-algoritmi (*Shortest Path Faster Algorithm*) on Bellman-Fordin algoritmin muunnos, joka on usein alkuperäistä algoritmia tehokkaampi. Se ei tutki joka kierroksella koko verkkoa läpi parantaakseen etäisyysarvioita, vaan valitsee tutkittavat kaaret älykkäämmin.

Algoritmi pitää yllä jonoa solmuista, joiden kautta saattaa pystyä parantamaan etäisyysarvioita. Algoritmi lisää jonoon aluksi aloitussolmun x ja valitsee aina seuraavan tutkittavan solmun a jonon alusta. Aina kun kaari $a \rightarrow b$ parantaa etäisyysarviota, algoritmi lisää jonoon solmun b .

Seuraavassa toteutuksessa jonona on queue-rakenne q . Lisäksi taulukko z kertoo, onko solmu valmiina jonossa, jolloin algoritmi ei lisää solmua jonoon uudestaan.

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push(x);
while (!q.empty()) {
    int a = q.front(); q.pop();
    z[a] = 0;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b.first]) {
            e[b.first] = e[a]+b.second;
            if (!z[b]) {q.push(b); z[b] = 1;}
        }
    }
}
```

SPFA-algoritmin tehokkuus riippuu verkon rakenteesta: algoritmi on keskimäärin hyvin tehokas mutta sen pahimman tapauksen aikavaativuus on edelleen $O(nm)$ ja on mahdollista laatia syötteitä, jotka saavat algoritmin yhtä hitaaksi kuin tavallisen Bellman-Fordin algoritmin.

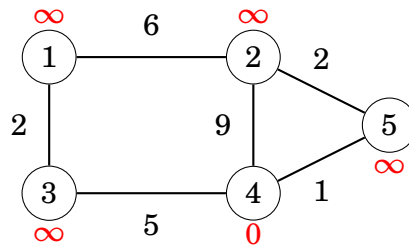
13.2 Dijkstran algoritmi

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 suurten verkkojen käsittelyyn. Algoritmi vaatii kuitenkin, ettei verkossa ole negatiivisia kaaria.

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

Toiminta

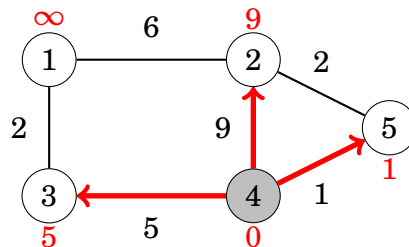
Tarkastellaan Dijkstran algoritmin toimintaa seuraavassa verkossa:



Bellman-Fordin algoritmin tavoin alkusolmun etäisyysarvio on 0 ja kaikissa muissa solmuissa etäisyysarvio on aluksi ∞ .

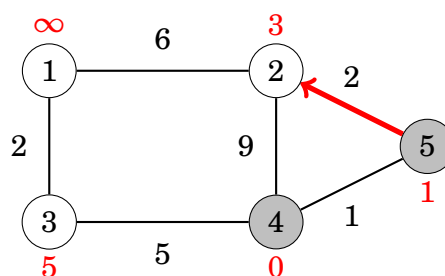
Dijkstran algoritmi ottaa joka askeleella käsittelyyn sellaisen solmun, jota ei ole vielä käsitelty ja jonka etäisyysarvio on mahdollisimman pieni. Alussa tällainen solmu on solmu 4, jonka etäisyysarvio on 0.

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

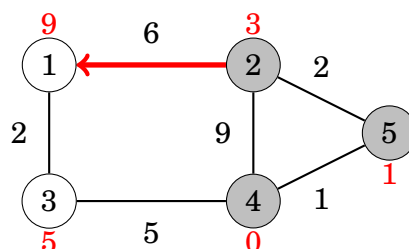


Solmun 4 käsittely paransi etäisyysarvioita solmuihin 2, 3 ja 5, joiden uudet etäisyydet ovat nyt 9, 5 ja 1.

Seuraavaksi käsittelyyn tulee solmu 5, jonka etäisyys on 1:

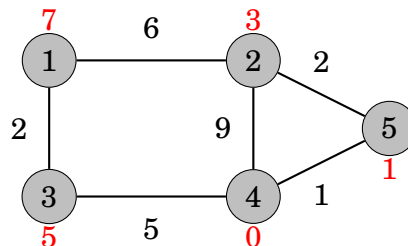


Tämän jälkeen vuorossa on solmu 2:



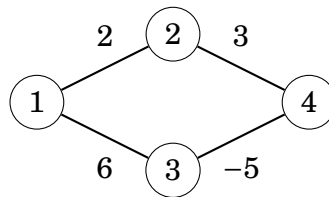
Dijkstran algoritmissa on hienoutena, että aina kun solmu tulee käsittelyyn, sen etäisyysarvio on siitä lähtien lopullinen. Esimerkiksi tässä vaiheessa etäisyydet 0, 1 ja 3 ovat lopulliset etäisyydet solmuihin 4, 5 ja 2.

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



Negatiiviset kaaret

Dijkstran algoritmin tehokkuus perustuu siihen, että verkossa ei ole negatiivisia kaaria. Jos verkossa on negatiivinen kaari, algoritmi ei välttämättä toimi oikein. Tarkastellaan esimerkkinä seuraavaa verkkoa:



Lyhin polku solmusta 1 solmuun 4 kulkee $1 \rightarrow 3 \rightarrow 4$, ja sen pituus on 1. Dijkstran algoritmi löytää kuitenkin keveimpiä kaaria seuraten polun $1 \rightarrow 2 \rightarrow 4$. Algoritmi ei pysty ottamaan huomioon, että alemmalla polulla kaaren paino -5 kumoaa aiemman suuren kaaren painon 6.

Toteutus

Seuraava Dijkstran algoritmin toteutus laskee pienimmän etäisyyden solmusta x kaikkiin muihin solmuihin. Verkko on tallennettu taulukkoon v vieruslistoina, joissa on pareina kohdesolmu ja kaaren pituus.

Dijkstran algoritmin tehokas toteutus vaatii, että verkosta pystyy löytämään nopeasti vielä käsittelemättömän solmun, jonka etäisyysarvio on pienin. Sopiva tietorakenne tähän on prioriteettijono, jossa solmut ovat järjestyksessä etäisyysarvioiden mukaan. Prioriteettijonon avulla seuraavaksi käsiteltävän solmun saa selville logaritmisessa ajassa.

Seuraavassa toteutuksessa prioriteettijono sisältää pareja, joiden ensimmäinen kenttä on etäisyysarvio ja toinen kenttä on solmun tunniste:

```
priority_queue<pair<int,int>> q;
```

Pieni hankaluus on, että Dijkstran algoritmista täytyy saada selville pienimmän etäisyysarvion solmu, kun taas C++:n prioriteettijono antaa oletuksena

suurimman alkion. Helppo ratkaisu on tallentaa etäisyysarviot *negatiivisina*, jolloin C++:n prioriteettijonoa voi käyttää suoraan.

Koodi merkitsee taulukkoon z , onko solmu käsitelty, ja pitää yllä etäisyysarvioita taulukossa e . Alussa alkusolmun etäisyysarvio on 0 ja jokaisen muun solmun etäisyysarviona on ääretöntä vastaava 10^9 .

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (z[a]) continue;
    z[a] = 1;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b]) {
            e[b] = e[a]+b.second;
            q.push({-e[b],b});
        }
    }
}
```

Yllä olevan toteutuksen aikavaativuus on $O(n + m \log n)$, koska algoritmi käy läpi kaikki verkon solmut ja lisää jokaista kaarta kohden korkeintaan yhden etäisyysarvion prioriteettijonoon. Huomaa, että $O(\log m) = O(\log n)$, koska n solmun verkossa on enintään $O(n^2)$ eri kaarta ja $\log(n^2) = 2 \log n$.

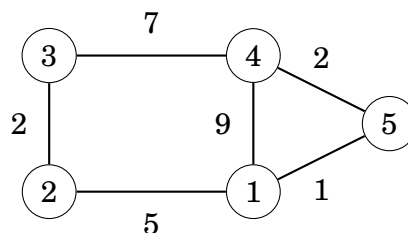
13.3 Floyd-Warshallin algoritmi

Floyd-Warshallin algoritmi lähestyy toisella tavalla lyhimpien polkujen etsimistä kuin Bellman-Fordin ja Dijkstran algoritmit. Siinä missä muut algoritmit etsivät lyhimpiä polkuja tietystä solmusta alkaen, Floyd-Warshallin algoritmi etsii samalla kertaa 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 verkon solmut saavat yksi kerrallaan toimia välisolmuina poluilla.

Toiminta

Tarkastellaan Floyd-Warshallin 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	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Algoritmin toiminta muodostuu peräkkäisistä kierroksista. Jokaisella kierroksella valitaan yksi uusi solmu, joka saa toimia välisolmuna poluilla, ja algoritmi parantaa taulukon etäisyyksiä muodostaen polkuja tämän solmun avulla.

Ensimmäisellä kierroksella solmu 1 on välisolmu. Tämän ansiosta solmujen 2 ja 4 välille muodostuu polku, jonka pituus on 14, koska solmu 1 yhdistää ne toisiinsa. Vastaavasti solmut 2 ja 5 yhdistyvät polulla, jonka pituus on 6.

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

Toisella kierroksella 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	7	8
4	9	14	7	0	2
5	1	6	8	2	0

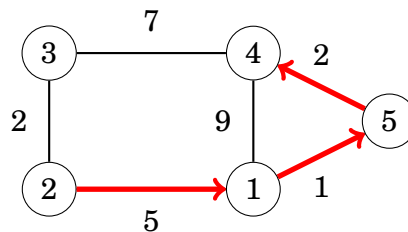
Kolmannella kierroksella solmu 3 saa toimia välisolmuna, jolloin syntyy uusi polku solmuparin 2 ja 4 välille:

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

Esimerkiksi taulukosta selviää, että lyhin polku solmusta 2 solmuun 4 on pituudeltaan 8. Tämä vastaa seuraavaa polkua:



Toteutus

Floyd-Warshallin algoritmin etuna on, että se on helppoa toteuttaa. Seuraava toteutus muodostaa etäisyysmatriisin d , jossa $d[a][b]$ on pienin etäisyys polulla solmusta a solmuun b . Aluksi algoritmi alustaa matriisin d verkon vierusmatriisiin v perusteella (arvo 10^9 kuvastaa ääretöntä):

```
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;
    }
}
```

Tämän jälkeen lyhimmat polut löytyvät seuraavasti:

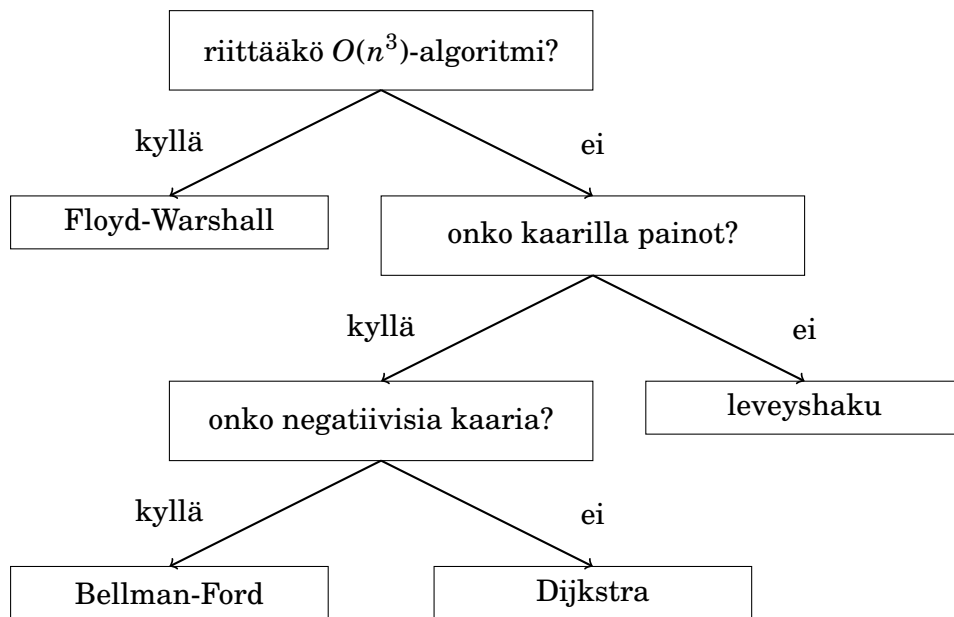
```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

Algoritmin aikavaativuus on $O(n^3)$, koska siinä on kolme sisäkkäistä silmukkaa, jotka käyvät läpi verkon solmut.

Koska algoritmin toteutus on yksinkertainen, se voi olla hyvä valinta jopa silloin, kun haettavana on yksittäinen lyhin polku verkossa. Tämä vaatii kuitenkin, että verkko on pieni ja kuutiollinen aikavaativuus on riittävä.

13.4 Yhteenveto

Olemme käyneet läpi useita algoritmeja, joilla voi etsiä lyhimmän polun verkossa solmusta a solmuun b . Jokaisella algoritmilla on jokin etu verrattuna muihin algoritmeihin. Seuraava kaavio auttaa sopivan algoritmin valinnassa:



Luku 14

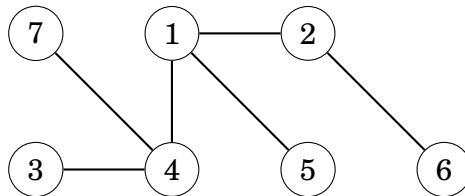
Puiden käsittely

Jos verkko on rakenteeltaan puu, sen käsittely on yleistä verkkoa helpompaa. Esimerkiksi lyhin polku kahden puun solmun välillä löytyy aina syvyyshaulla, koska puussa ei ole silmukoita. Tässä luvussa opimme lisää puiden käsittelystä ja näemme, miten puihin voi soveltaa dynaamista ohjelmointia.

14.1 Käsitteitä

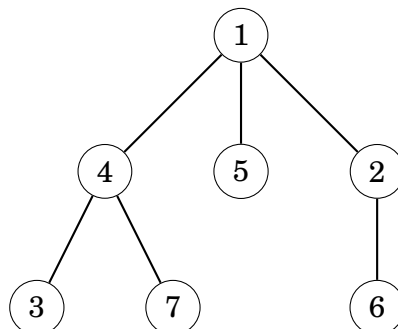
Puu (*tree*) on yhtenäinen, syklitön ja suuntaamaton verkko, jossa on n solmua ja $n - 1$ kaarta. Jokaisen kahden puun solmun välillä on yksikäsitteinen polku. Jos puusta poistaa yhden kaaren, se ei ole enää yhtenäinen, ja jos puuhun lisää yhden kaaren, se ei ole enää syklitön.

Esimerkiksi seuraavassa puussa on 7 solmua ja 6 kaarta:



Puun lehdet (*leaves*) ovat solmut, joiden aste on 1 eli joista lähtee vain yksi kaari. Esimerkiksi yllä olevan puun lehdet ovat solmut 3, 5, 6 ja 7.

Usein yksi solmuista valitaan puun juureksi (*root*) ja muut solmut asettuvat sen alapuolelle. Esimerkiksi jos yllä olevassa puussa valitaan juureksi solmu 1, solmut asettuvat seuraavaan järjestykseen:



Solmun lapset (*children*) ovat sen alemman tason naapurit ja solmun vanhempi (*parent*) on sen ylemmän tason naapuri. Jokaisella solmulla on tasan yksi vanhempi, paitsi juurella ei ole vanhempaa. Esimerkiksi yllä olevassa puussa solmun 4 lapset ovat solmut 3 ja 7 ja solmun 4 vanhempi on solmu 1.

Juurellisen puun rakenne on rekursiivinen: jokaisesta puun solmusta alkaa alipuu (*subtree*), jonka juurena on solmu itse ja johon kuuluvat kaikki solmut, joihin solmusta pääsee kulkemalla alaspäin puussa. Esimerkiksi solmun 4 alipuussa ovat solmut 4, 3 ja 7.

14.2 Perustekniikat

14.2.1 Läpikäynti

Puun läpikäyntiin voi käyttää syvyyshakua ja leveyshakua samaan tapaan kuin yleisen verkon läpikäyntiin. Erona on kuitenkin, että puussa ei ole silmukoita, minkä ansiosta ei tarvitse huolehtia siitä, että läpikäynti päättyisi tiettyyn solmuun monesta eri suunnasta.

Tavallisin menetelmä puun läpikäyntiin on valita tietty solmu juureksi ja aloittaa siitä syvyyshaku. Seuraava rekursiivinen funktio toteuttaa sen:

```
void dfs(int s, int e) {  
    // solmun s käsittely tähän  
    for (auto u : v[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Funktion parametrit ovat käsiteltävä solmu s sekä edellinen käsitelty solmu e . Parametrin e ideana on varmistaa, että läpikäynti etenee vain alaspäin puussa sellaisiin solmuihin, joita ei ole vielä käsitelty.

Seuraava kutsu käy läpi puun aloittaen juuresta x :

```
dfs(x, 0);
```

Ensimmäisessä kutsussa $e = 0$, koska läpikäynti saa edetä juuresta kaikkiin suuntiin alaspäin.

14.2.2 Dynaaminen ohjelmointi

Puun läpikäyntiin voi yhdistää dynaamista ohjelmointia ja laskea sen avulla jotain tietoa puusta. Dynaamisen ohjelmoinnin avulla voi esimerkiksi laskea ajassa $O(n)$ jokaiselle solmulle, montako solmua sen alipuussa on tai kuinka pitkä on pisin solmusta alaspäin jatkuva polku puussa.

Lasketaan esimerkiksi jokaiselle solmulle s sen alipuun solmujen määrä $c[s]$. Solmun alipuuhun kuuluvat solmu itse sekä kaikki sen lasten alipuut. Niinpä solmun alipuun solmujen määrä on yhden suurempi kuin summa lasten alipuiden solmujen määristä. Laskennan voi toteuttaa seuraavasti:


```

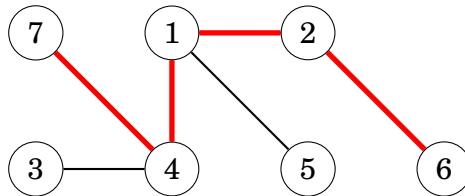
void dfs(int s, int e) {
    c[s] = 1;
    for (auto u : v[s]) {
        if (u == e) continue;
        dfs(u, s);
        c[s] += c[u];
    }
}

```

14.3 Etäisyydet

14.3.1 Läpimitta

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



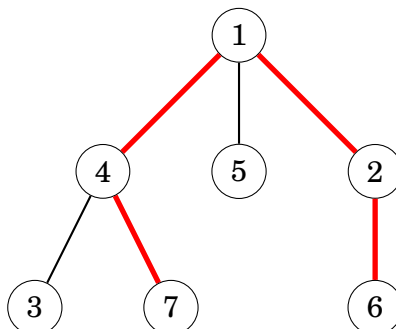
läpimitta on 4, koska polku solmusta 3 solmuun 6 on pituudeltaan 4. Läpimittaa vastaava polku ei ole välttämättä yksikäsitteinen. Esimerkiksi tässä puussa myös polku solmusta 7 solmuun 6 on pituudeltaan 4.

Käymme seuraavaksi läpi kaksi tehokasta algoritmia puun läpimitan laskemiseen. Molemmat algoritmit laskevat läpimitan ajassa $O(n)$. 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 josta-kin lehdestä, nousee kyseiseen solmuun asti ja laskeutuu toiseen lehteen. Pisin tällaisista poluista vastaa puun läpimittaa.

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



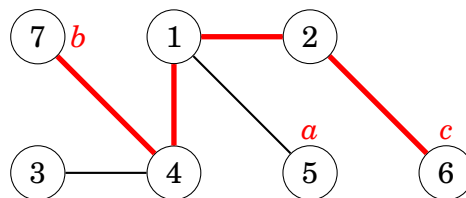
Algoritmi laskee ensin dynaamisella ohjelmoinnilla jokaiselle solmulle, kuinka pitkä on pisin solmu, joka lähtee solmusta alaspäin. Esimerkiksi yllä olevassa puussa pisin polku solmusta 1 alaspäin on pituudeltaan 2 (vaihtoehdot $1 \rightarrow 4 \rightarrow 3$, $1 \rightarrow 4 \rightarrow 7$ ja $1 \rightarrow 2 \rightarrow 6$).

Tämän jälkeen algoritmi laskee kullekin solmulle, kuinka pitkä on pisin polku, jossa solmu on käännekohtana. Pisin tällainen polku syntyy valitsemalla kaksi lasta, joista lähtee alaspäin mahdollisimman pitkä polku. Esimerkiksi yllä olevassa puussa solmun 1 lapsista valitaan solmut 2 ja 4.

Algoritmi 2

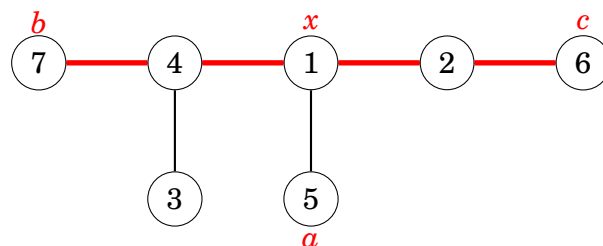
Toinen tehokas tapa laskea puun läpimitta perustuu kahteen syvyyshakuun. Ensimmäin valitaan mikä tahansa solmu a puusta ja etsitään siitä kaukaisin solmu b syvyyshaulla. Tämän jälkeen etsitään b :stä kaukaisin solmu c syvyyshaulla. Puun läpimitta on etäisyys b :n ja c :n välillä.

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:

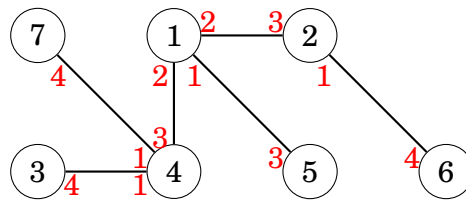


Solmu x on kohta, jossa polku solmusta a liittyy läpimittaa vastaavaan polkuun. Kaukaisin solmu a :sta on solmu b , solmu c tai jokin muu solmu, joka on ainakin yhtä kaukana solmusta x . Niinpä tämä solmu on aina sopiva valinta läpimittaa vastaavan polun toiseksi päätesolmuksi.

14.3.2 Kaikki etäisyydet

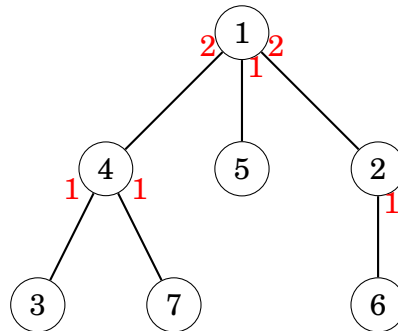
Vaikeampi tehtävä on laskea jokaiselle puun solmulle jokaiseen suuntaan, mikä on suurin etäisyys johonkin kyseisessä suunnassa olevaan solmuun. Osoittautuu, että tämäkin tehtävä ratkeaa ajassa $O(n)$ dynaamisella ohjelmoinnilla.

Esimerkkipuussa etäisyydet ovat:



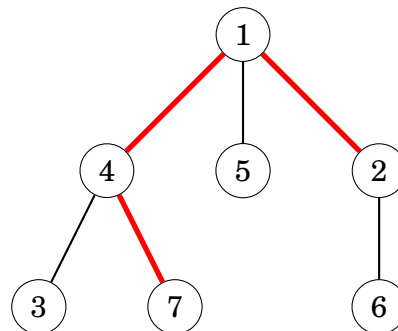
Esimerkiksi solmussa 4 kaukaisin solmu ylöspäin mentäessä on solmu 6, johon etäisyys on 3 käyttäen polkua $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$.

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

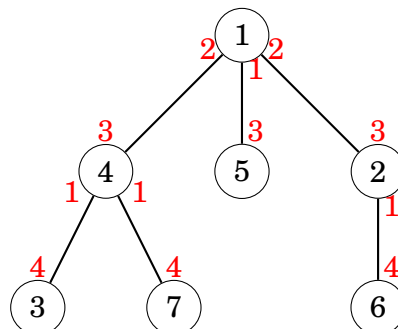


Jäljelle jäävä tehtävä on laskea etäisyydet ylöspäin. Tämä onnistuu teke-
mällä puuhun toinen läpikäynti, joka pitää mukana tietoa, mikä on suurin etäi-
syys solmun vanhemmasta johonkin toisessa suunnassa olevaan solmuun.

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

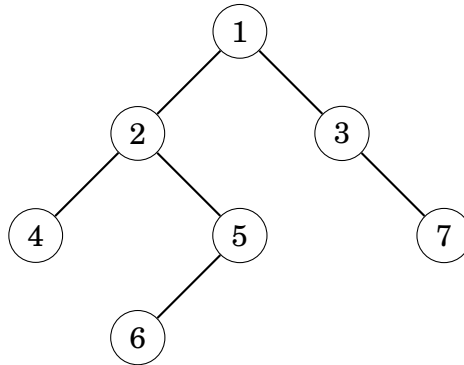


Lopputuloksena on etäisyydet kaikista solmuista kaikkiin suuntiin:



14.4 Binääripuut

Binääripuu (*binary tree*) on juurellinen puu, jonka jokaisella solmulla on vasen ja oikea alipuu. On mahdollista, että alipuu on tyhjä, jolloin puu ei jatku siitä pidemmälle alaspäin. Niinpä jokaisella solmulla on 0, 1 tai 2 lasta. Esimerkiksi seuraava puu on binääripuu:



Binääripuun solmuilla on kolme luontevaa järjestystä, jotka syntyvät rekursiivisesta läpikäynnistä:

- esijärjestys (*pre-order*): juuri, vasen alipuu, oikea alipuu
- sisäjärjestys (*in-order*): vasen alipuu, juuri, oikea alipuu
- jälkijärjestys (*post-order*): vasen alipuu, oikea alipuu, juuri

Esimerkissä kuvatun puun esijärjestys on (1, 2, 4, 5, 6, 3, 7), sisäjärjestys on (4, 2, 6, 5, 1, 3, 7) ja jälkijärjestys on (4, 6, 5, 2, 7, 3, 1).

Osoittautuu, että tietämällä puun esijärjestyksen ja sisäjärjestyksen voi päätellä puun koko rakenteen. Esimerkiksi yllä oleva puu on ainoa mahdollinen puu, jossa esijärjestys on (1, 2, 4, 5, 6, 3, 7) ja sisäjärjestys on (4, 2, 6, 5, 1, 3, 7). Vastaavasti myös jälkijärjestys ja sisäjärjestys määrittävät puun rakenteen.

Tilanne on toinen, jos tiedossa on vain esijärjestys ja jälkijärjestys. Nämä järjestykset eivät kuvaa välttämättä puuta yksikäsitteisesti. Esimerkiksi molemmissa puissa



esijärjestys on (1, 2) ja jälkijärjestys on (2, 1), mutta siitä huolimatta puiden rakenteet eivät ole samat.

Luku 15

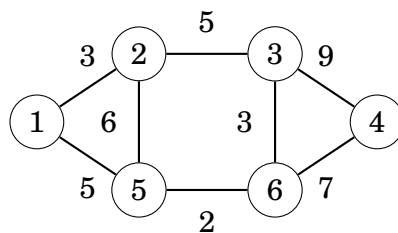
Virittävät puut

Tässä luvussa tutustumme algoritmeihin, jotka muodostavat verkon pienimmän tai suurimman virittävän puun. Tällaisessa puussa on joukko verkon kaaria niin, että kaikki verkon solmut ovat yhteydessä toisiinsa kaarten kautta ja lisäksi kaarten painojen summa on pienin tai suurin mahdollinen.

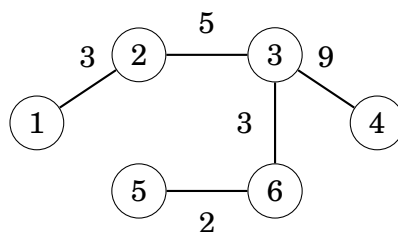
Osoittautuu, että virittävien puiden etsiminen on siinä mielessä helppo ongelma, että monenlaiset ahneet menetelmät tuottavat optimaalisen ratkaisun. Käymme läpi kaksi algoritmia, jotka molemmat valitsevat puuhun mukaan kaaria ahneesti niiden painojen perusteella.

15.1 Käsitteitä

Virittävä puu (*spanning tree*) on kokoelma verkon kaaria, joka kytkee kaikki verkon solmut toisiinsa. Kuten puut yleensä, virittävä puu on yhtenäinen ja syklitön. Esimerkiksi verkossa

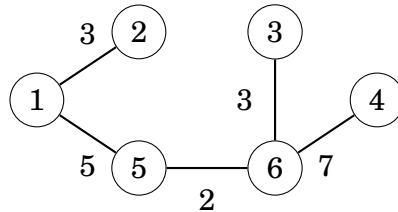


yksi mahdollinen virittävä puu on seuraava:

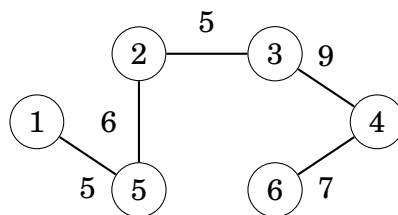


Virittävän puun paino on siihen kuuluvien kaarten painojen summa. Esimerkiksi yllä olevan puun paino on $3 + 5 + 9 + 3 + 2 = 22$.

Pienin virittävä puu (*minimum spanning tree*) on virittävä puu, jonka paino on mahdollisimman pieni. Yllä olevan verkon pienin virittävä puu on painoltaan 20, ja sen voi muodostaa seuraavasti:



Vastaavasti suurin virittävä puu (*maximum spanning tree*) on virittävä puu, jonka paino on mahdollisimman suuri. Yllä olevan verkon suurin virittävä puu on painoltaan 32:



Huomaa, että virittävät puut eivät ole yksikäsitteisiä vaan voi olla monta tapaa muodostaa pienin tai suurin virittävä puu.

Sekä pienimmän että suurimman virittävän puun voi etsiä tehokkaasti ahneilla algoritmeilla. Keskitymme seuraavaksi pienimmän virittävän puun etsimiseen, mutta samoja algoritmeja voi käyttää myös suurimman virittävän puun etsimiseen käsittelemällä kaaret käänteisessä järjestyksessä.

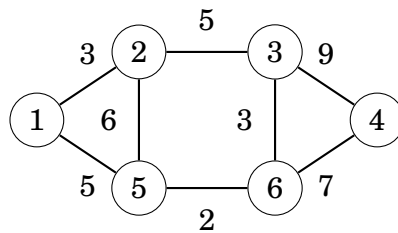
15.2 Kruskalin algoritmi

Kruskalin algoritmi aloittaa pienimmän virittävän puun rakentamisen tilanteesta, jossa puussa ei ole yhtään kaaria. Sitten algoritmi alkaa lisätä puuhun kaaria järjestyksessä kevyimmästä raskaimpaan. Algoritmi lisää kaaren mukaan puuhun, jos sen lisääminen ei muodosta sykliä.

Kruskalin algoritmi pitää yllä tietoa verkon komponenteista. Aluksi jokainen solmu on omassa komponentissaan, ja komponentit yhdistyvät pikkuhiljaa algoritmin aikana puuhun tulevista kaarista. Lopulta kaikki solmut ovat samassa komponentissa, jolloin pienin virittävä puu on valmis.

Toiminta

Tarkastellaan Kruskalin algoritmin toimintaa seuraavassa verkossa:

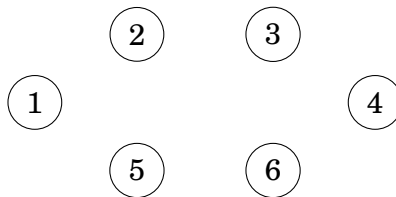


Algoritmin ensimmäinen vaihe on järjestää verkon kaaret niiden painon mukaan. Tuloksena on seuraava lista:

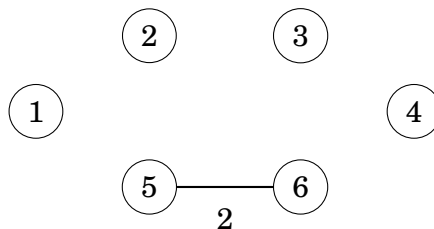
kaari	paino
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

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

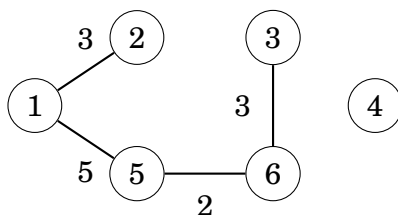
Aluksi jokainen solmu on omassa komponentissaan:



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



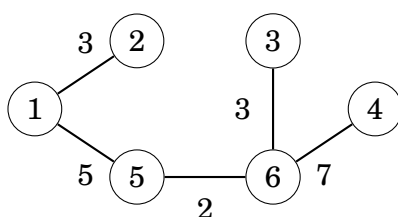
Tämän jälkeen algoritmi lisää puuhun vastaavasti kaaret 1–2, 3–6 ja 1–5:



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:



Tämän lisäyksen jälkeen algoritmi päättyy, koska kaikki solmut on kytketty toisiinsa kaarilla ja verkko on yhtenäinen. Tuloksena on verkon pienin virittävä puu, jonka paino on $2 + 3 + 3 + 5 + 7 = 20$.

Miksi Kruskalin algoritmi tuottaa varmasti pienimmän virittävän puun?

Kruskalin algoritmin tuloksena on aina pienin mahdollinen virittävä puu, koska kahden komponentin yhdistämisessä mahdollisimman kevyt kaari on aina paras valinta. Jos kevyintä kaarta ei valittaisi, komponentit täytyisi yhdistää myöhemmin käyttäen raskaampaa kaarta.

Toteutus

Kruskalin algoritmi on mukavinta toteuttaa kaarilistan avulla. Algoritmin ensimmäinen vaihe on järjestää kaaret painojärjestykseen, missä kuluu aikaa $O(m \log m)$. Tämän jälkeen seuraa algoritmin toinen vaihe, jossa listalta valitaan kaaret mukaan puuhun.

Algoritmin toinen vaihe rakentuu seuraavanlaisen silmukan ympärille:

```
for (...) {
    if (!sama(a,b)) liita(a,b);
}
```

Silmukka käy läpi kaikki listan kaaret niin, että muuttujat a ja b ovat kulloinkin kaaren päissä olevat solmut. Koodi käyttää kahta funktiota: funktio `sama` tutkii, ovatko solmut samassa komponentissa, ja funktio `liita` yhdistää kaksi komponenttia toisiinsa.

Ongelmana on, kuinka toteuttaa tehokkaasti funktiot `sama` ja `liita`. Yksi mahdollisuus on pitää yllä verkkoa tavallisesti ja toteuttaa funktio `sama` verkon

läpikäyntinä. Tällöin kuitenkin funktion sama suoritus veisi aikaa $O(n + m)$, mikä on hidasta, koska funktiota kutsutaan jokaisen kaaren kohdalla.

Seuraavaksi esiteltävä union-find-tietorakenne ratkaisee asian. Se toteuttaa kummankin funktion ajassa $O(\log n)$, jolloin Kruskalin algoritmin aikavaativuus on vain $O(m \log n)$ kaarilistan järjestämisen jälkeen.

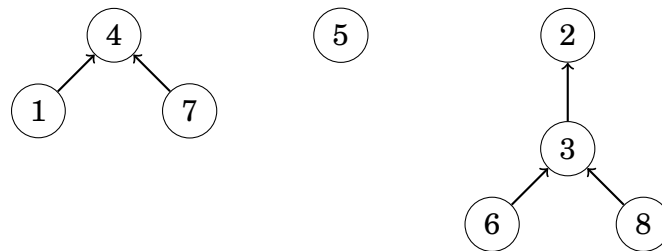
15.3 Union-find-rakenne

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 alkioita samassa joukossa. Toinen operaatio yhdistää kaksi joukkoa toisiinsa.

Rakenne

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

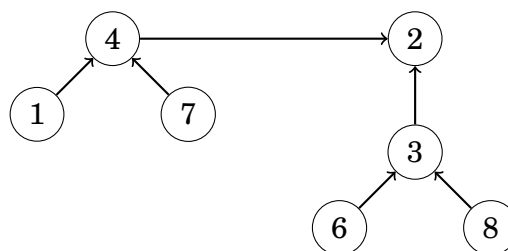
Esimerkiksi jos joukot ovat $\{1, 4, 7\}$, $\{5\}$ ja $\{2, 3, 6, 8\}$, tilanne voisi olla:



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

Minkä tahansa alkion edustaja löytyy kulkemalla polku loppuun alkioista. Esimerkiksi alkion 6 edustaja on 2, koska polku on $6 \rightarrow 3 \rightarrow 2$. Tämän avulla voi selvittää, ovatko kaksi alkioita samassa joukossa: jos kummankin alkion edustaja on sama, alkioit 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 $\{1, 2, 3, 4, 6, 7, 8\}$:



Joukkojen yhteiseksi edustajaksi valitaan alkio 2, minkä vuoksi alkio 4 yhdistetään siihen. Tästä lähtien alkio 2 edustaa kaikkia joukon alkioita.

Tehokkuuden kannalta oleellista on, miten yhdistäminen tapahtuu. Osoitetaan, että ratkaisu on yksinkertainen: riittää yhdistää aina pienempi joukko suurempaan, tai kummin päin tahansa, jos joukot ovat yhtä suuret. Tällöin pisin ketju alkioista edustajaan on aina luokkaa $O(\log n)$.

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 alkioita 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 `id` kertoo alkion x joukon edustajan:

```
int id(int x) {
    while (x != k[x]) x = k[x];
    return x;
}
```

Funktio `sama` kertoo, ovatko alkio a ja b samassa joukossa:

```
bool sama(int a, int b) {
    return id(a) == id(b);
}
```

Funktio `liita` taas yhdistää kaksi joukkoa toisiinsa:

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

Funktion `id` aikavaativuus on $O(\log n)$ olettaen, että ketjun pituus on luokkaa $O(\log n)$. Niinpä myös funktioiden `sama` ja `liita` aikavaativuus on $O(\log n)$. Funktio `liita` varmistaa, että ketjun pituus on luokkaa $O(\log n)$ yhdistämällä pienemmän joukon suurempaan.

Funktiota `id` on mahdollista vielä tehostaa seuraavasti:

```
int id(int x) {
    if (x == k[x]) return x;
    return k[x] = id(x);
}
```

Nyt joukon edustajan etsimisen yhteydessä kaikki ketjun alkiot laitetaan osoittamaan suoraan edustajaan. On mahdollista osoittaa, että tämän avulla funktioiden sama ja liita aikavaativuus on tasoitettu vain $O(\alpha(n))$, missä $\alpha(n)$ on hyvin hitaasti kasvava käänteinen Ackermannin funktio.

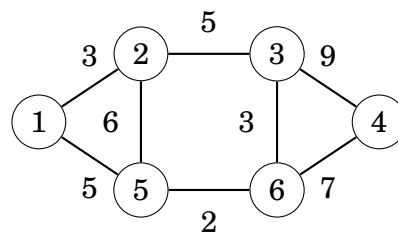
15.4 Primin algoritmi

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.

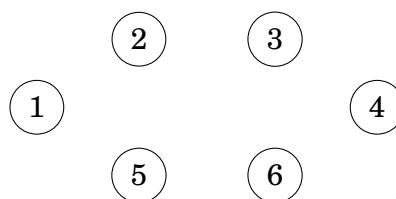
Primin algoritmin toiminta on hyvin lähellä Dijkstran algoritmia. Erona on, että Dijkstran algoritmissa valitaan kaari, jonka kautta syntyy lyhin polku alkusolmusta uuteen solmuun, mutta Primin algoritmissa valitaan vain kevein kaari, joka johtaa uuteen solmuun.

Toiminta

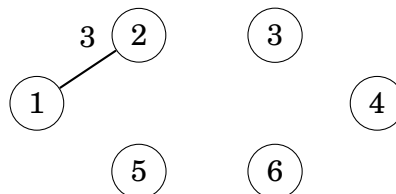
Tarkastellaan Primin algoritmin toimintaa seuraavassa verkossa:



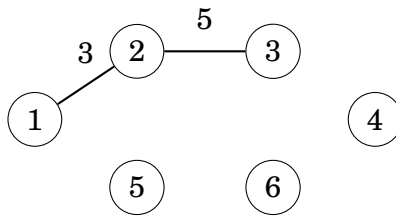
Aluksi solmujen välillä ei ole mitään kaaria:



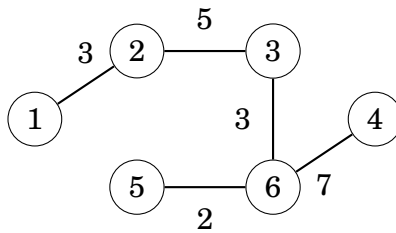
Puun muodostuksen voi aloittaa mistä tahansa solmusta, ja aloitetaan se nyt solmusta 1. Kevyin kaari on painoltaan 3 ja se johtaa solmuun 2:



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:



Toteutus

Dijkstran algoritmin tavoin Primin algoritmin voi toteuttaa tehokkaasti käyttämällä prioriteettijonoa. Primin algoritmin tapauksessa jono sisältää kaikki solmut, jotka voi yhdistää nykyiseen komponenttiin kaarella, järjestyksessä kaaren painon mukaan kevyimmästä raskaimpaan.

Primin algoritmin aikavaativuus on $O(n + m \log n)$ eli sama kuin Dijkstran algoritmista. Käytännössä Primin algoritmi on suunnilleen yhtä nopea kuin Kruskalin algoritmi, ja onkin makuasia, kumpaa algoritmia käyttää. Useimmat kisakoodarit käyttävät Kruskalin algoritmia.

Luku 16

Suunnatut verkot

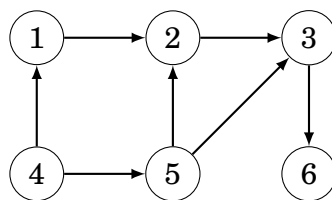
Tämä luku käsittelee kahta suunnattuihin verkkoihin liittyvää aihetta. Ensin keskitymme verkkoihin, joissa ei ole silmukoita. Tämä mahdollistaa topologisen järjestyksen muodostamisen verkon solmuille, minkä ansiosta verkossa voi edelleen käyttää dynaamista ohjelmointia.

Tämän jälkeen tutustumme funktionaalisiin verkkoihin eli verkkoihin, joissa jokaisesta solmusta lähtee tasan yksi kaari. Tällaisessa verkossa jokaisella solmulla on yksikäsitteinen seuraaja ja verkossa etenemiseen ja syklien etsimiseen on olemassa tehokkaita algoritmeja.

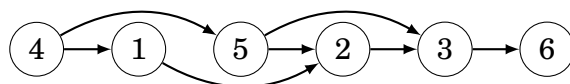
16.1 Topologinen järjestys

Topologinen järjestys (*topological sort*) on tapa järjestää suunnatun verkon solmut niin, että jos solmusta a pääsee solmuun b , niin a on ennen b :tä järjestyksessä. Esimerkiksi jos solmut ovat kurssoja ja kaaret ovat kurssien esitietovaatimukset, topologinen järjestys antaa tavan suorittaa kaikki kurssit.

Esimerkiksi verkon



yksi topologinen järjestys on (4, 1, 5, 2, 3, 6):



Topologinen järjestys on olemassa aina silloin, kun verkossa ei ole sykliä. Jos taas verkossa on sykli, topologista järjestystä ei voi muodostaa. Seuraavaksi näemme, miten syvyyshaun avulla voi muodostaa topologisen järjestyksen tai todeta, että tämä ei ole mahdollista syklin takia.

Algoritmi

Ideana on käydä läpi verkon solmut syvyyshaulla, jossa on kolme tilaa:

- tila 0: solmua ei ole käsitelty (valkoinen)
- tila 1: solmun käsittely on alkanut (vaaleanharmaa)
- tila 2: solmu on käsitelty (tummanharmaa)

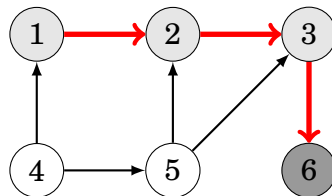
Aluksi jokaisen verkon solmun tila on 0. Kun syvyyshaku saapuu solmuun, sen tilaksi tulee 1. Lopuksi kun syvyyshaku on käsitellyt kaikki solmun naapurit, solmun tilaksi tulee 2.

Jos verkossa on sykli, tämä selviää syvyysshaun aikana siitä, että jossain vaiheessa haku saapuu solmuun, jonka tila on 1. Tässä tapauksessa topologista järjestystä ei voi muodostaa.

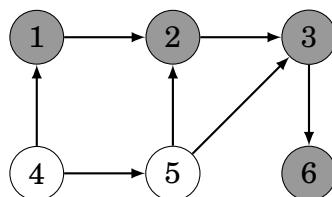
Jos verkossa ei ole sykliä, topologinen järjestys saadaan muodostamalla lista, johon kukin solmu lisätään silloin, kun sen tilaksi tulee 2. Tämä lista käänteisenä on yksi verkon topologinen järjestys.

Esimerkki 1

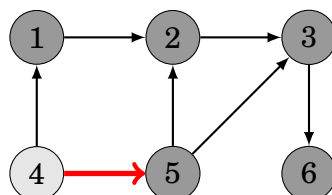
Esimerkkiverkossa syvyyshaku etenee ensin solmusta 1 solmuun 6 asti:



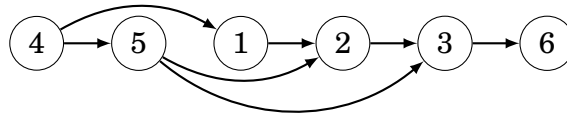
Tässä vaiheessa solmu 6 on käsitelty, joten se lisätään listalle. Sen jälkeen haku palaa takaisinpäin:



Tämän jälkeen listan sisältönä on (6,3,2,1). Seuraavaksi syvyysshaun toinen vaihe alkaa solmusta 4:



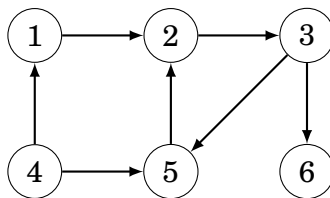
Tämän seurauksena listaksi tulee (6, 3, 2, 1, 5, 4). Kaikki solmut on käyty läpi, joten topologinen järjestys on valmis. Se on lista käänteisenä eli (4, 5, 1, 2, 3, 6):



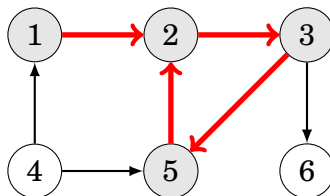
Huomaa, että topologinen järjestys ei ole yksikäsitteinen, vaan verkolla voi olla useita topologisia järjestyksiä.

Esimerkki 2

Tarkastellaan sitten tilannetta, jossa topologista järjestystä ei voi muodostaa syklin takia. Näin on seuraavassa verkossa:



Nyt syvyysshaun aikana tapahtuu näin:



Syvyyshaku saapuu tilassa 1 olevaan solmuun 2, mikä tarkoittaa, että verkossa on sykli. Tässä tapauksessa sykli on $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

16.2 Dynaaminen ohjelmointi

Jos suunnatussa verkossa ei ole syklä, sen solmuihin voi soveltaa dynaamista ohjelmointia topologisessa järjestyksessä. Dynaamisen ohjelmoinnin avulla voi vastata esimerkiksi seuraaviin kysymyksiin koskien verkossa olevia polkuja solmusta a solmuun b :

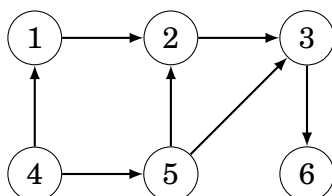
- montako erilaista polkua on olemassa?
- mikä on pienin/suurin mahdollinen määrä kaaria polulla?
- mitkä solmut esiintyvät varmasti polulla?

Tutustumme seuraavaksi tekniikkaan käyttäen esimerkkinä polkujen määrän laskemista verkossa.

16.2.1 Polkujen määrä

Tehtävä: Annettuna on suunnattu verkko, jossa ei ole sykliä. Tehtäväsi on laskea, montako polkua on olemassa solmusta a solmuun b .

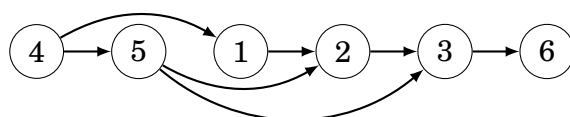
Esimerkiksi verkossa



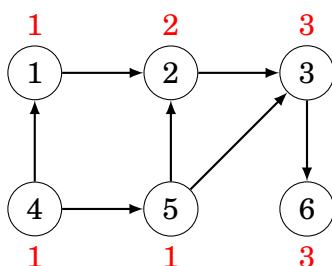
on 3 polkua solmusta 4 solmuun 6:

- $4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

Ideana on käydä läpi verkon solmut topologisessa järjestyksessä ja laskea kunkin solmun kohdalla yhteen eri suunnista tulevien polkujen määrät. Verkon topologinen järjestys on seuraava:



Tuloksena ovat seuraavat lukumäärät:

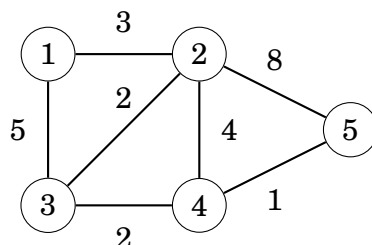


Esimerkiksi solmuun 2 pääsee solmuista 1 ja 5. Kumpaankin solmuun päättyy yksi polku solmusta 4 alkaen, joten solmuun 2 päättyy kaksi polkua solmusta 4 alkaen. Vastaavasti solmuun 3 pääsee solmuista 2 ja 5, joiden kautta tulee kaksi ja yksi polkua solmusta 4 alkaen.

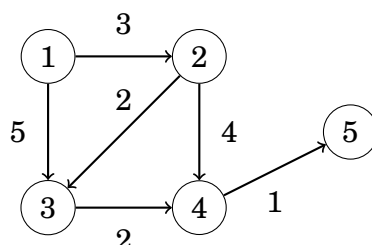
16.2.2 Dijkstran sovellukset

Dijkstran algoritmin sivutuotteena syntyy suunnattu, syklitön verkko, joka kertoo jokaiselle alkuperäisen verkon solmulle, mitä tapoja alkusolmusta on päästä kyseiseen solmuun lyhintä polkua käyttäen. Tähän verkkoon voi soveltaa edelleen dynaamista ohjelmointia.

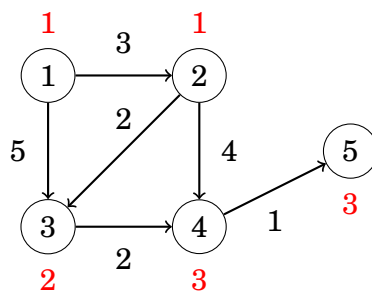
Esimerkiksi verkossa



solmusta 1 lähteviin lyhimpiin polkuihin kuuluvat seuraavat kaaret:



Koska kyseessä on suunnaton, syklitön verkko, siihen voi soveltaa dynaamista ohjelmointia. Niinpä voi esimerkiksi laskea, montako lyhintä polkua on olemassa solmusta 1 solmuun 5:



16.3 Funktionaaliset verkot

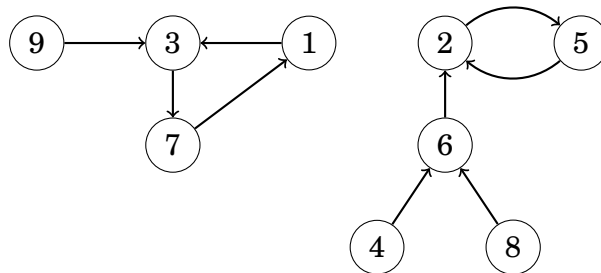
Funktionaalinen verkko (*functional graph*) on suunnattu verkko, jonka jokaisesta solmusta lähtee tasan yksi kaari ulospäin. Funktionaalinen verkko muodostuu yhdestä tai useammasta komponentista, joista jokaisessa on yksi sykli ja joukko siihen johtavia polkuja.

Termi ”funktionaalinen” johtuu siitä, että jokaista funktionaalista verkkoa vastaa funktio f , joka määrittelee verkon kaaret. Funktion parametrina on verkon solmu ja se palauttaa solmusta lähtevän kaaren kohdesolmun. Verkon kaaret ovat siis muotoa $(x, f(x))$, missä $s = 1, 2, \dots, n$.

Esimerkiksi funktio

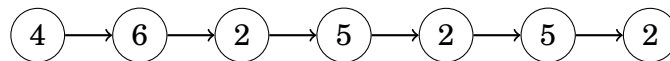
x	1	2	3	4	5	6	7	8	9
$f(x)$	3	5	7	6	2	2	1	6	3

määrittelee seuraavan verkon:



16.3.1 Nopea eteneminen

Koska funktionaalisessa verkossa jokaisella solmulla on yksikäsitteinen seuraaja, voimme määritellä funktion $f(x, k)$, joka kertoo solmun, johon päättyy solmusta x kulkemalla k askelta. Esimerkiksi yllä olevassa verkossa $f(4, 6) = 2$, koska solmusta 4 päättyy solmuun 2 kulkemalla 6 askelta:



Suoraviivainen tapa laskea funktion $f(x, k)$ arvo on käydä läpi polku verkossa askel askeleelta, mihin kuluu aikaa $O(k)$. Sopivan esikäsittelyn avulla kuitenkin minkä tahansa funktion arvon pystyy laskemaan ajassa $O(\log k)$.

Ideana on laskea etukäteen funktion $f(x, k)$ arvot, kun k on $2:n$ potenssi. Tämä onnistuu tehokkaasti, koska $f(x, 1) = f(x)$ ja $f(x, k) = f(f(x, k/2), k/2)$. Esilaskenta vie aikaa $O(n \log k)$, koska jokaisesta solmusta lasketaan $O(\log k)$ arvoa. Esimerkin tapauksessa muodostuu seuraava taulukko:

x	1	2	3	4	5	6	7	8	9
$f(x, 1)$	3	5	7	6	2	2	1	6	3
$f(x, 2)$	7	2	1	2	5	5	3	2	7
$f(x, 4)$	3	2	7	2	5	5	1	2	3
$f(x, 8)$	7	2	1	2	5	5	3	2	7
...									

Tämän jälkeen funktion $f(x, k)$ arvon saa laskettua esittämällä luvun k summana $2:n$ potensseja. Esimerkiksi $11 = 8 + 2 + 1$, joten $f(x, 11) = f(f(f(x, 8), 2), 1)$. Niinpä yllä olevassa verkossa $f(4, 11) = f(f(f(4, 8), 2), 1) = 5$. Polussa on aina $O(\log k)$ osaa, joten laskemiseen kuluu aikaa $O(\log k)$.

16.3.2 Syklin etsiminen

Toinen kiinnostava kysymys funktionaalisessa verkossa on, kuinka monta askelta solmusta x tulee kulkea ennen sykliin pääsemistä ja mikä on syklin pituus. Esimerkiksi yllä olevassa verkossa solmusta 9 pääsee sykliin kulkemalla yhden askeleen ja sykliin kuuluu kolme solmua (1, 3 ja 7).

Helppo tapa etsiä sykli on alkaa kulkea verkossa solmusta x alkaen ja pitää kirjaa kaikista vastaan tulevista solmuista. Kun jokin solmu tulee vastaan toista kertaa, sykli on löytynyt. Tämän menetelmän aikavaativuus on $O(n)$ ja muistia kuluu myös $O(n)$.

Osoittautuu kuitenkin, että ongelman ratkaisuun on olemassa parempia algoritmeja. Niissä aikavaativuus on edelleen $O(n)$, mutta muistia kuluu vain $O(1)$. Tästä on merkittävää hyötyä, jos n on suuri. Tutustumme seuraavaksi kahteen tällaiseen algoritmiin.

Algoritmi 1 (Floyd)

Floydin algoritmi kulkee verkossa eteenpäin rinnakkain kahdesta kohdasta. Osoitin a liikkuu joka vuorolla askeleen eteenpäin, kun taas osoitin b liikkuu kaksi askelta eteenpäin. Haku jatkuu, kunnes osoittimet kohtaavat:

```
a = f(x);
b = f(f(x));
while (a != b) {
    a = f(a);
    b = f(f(b));
}
```

Tässä vaiheessa osoitin a on kulkenut k askelta ja osoitin b on kulkenut $2k$ askelta, missä k on jaollinen syklin pituudella. Niinpä ensimmäinen sykliin kuuluva solmu löytyy siirtämällä osoitin a alkuun ja liikuttamalla osoittimia rinnakkain eteenpäin, kunnes ne kohtaavat.

```
a = x;
while (a != b) {
    a = f(a);
    b = f(b);
}
```

Nyt a ja b osoittavat ensimmäiseen syklin solmuun, joka tulee vastaan solmusta x lähdettäessä. Lopuksi syklin pituus c voidaan laskea näin:

```
b = f(a);
c = 1;
while (a != b) {
    b = f(b);
    c++;
}
```

Algoritmi 2 (Brent)

Brentin algoritmi muodostuu peräkkäisistä vaiheista, joissa osoitin a pysyy paikallaan ja osoitin b liikkuu k askelta. Alussa $k = 1$ ja k :n arvo kaksinkertaistuu joka vaiheen alussa. Lisäksi a siirretään b :n kohdalle vaiheen alussa. Näin jatketaan, kunnes osoittimet kohtaavat.

```
a = x;
b = f(x);
c = k = 1;
while (a != b) {
    if (c == k) {
        a = b;
        c = 0;
        k *= 2;
    }
    b = f(b);
    c++;
}
```

Nyt tiedossa on, että syklin pituus on c . Tämän jälkeen ensimmäinen sykliin kuuluva solmu löytyy palauttamalla ensin osoittimet alkuun, siirtämällä sitten osoitinta b eteenpäin c askelta ja liikuttamalla lopuksi osoittimia rinnakkain, kunnes ne osoittavat samaan solmuun.

```
a = b = x;
for (int i = 0; i < c; i++) b = f(b);
while (a != b) {
    a = f(a);
    b = f(b);
}
```

Nyt a ja b osoittavat ensimmäiseen sykliin kuuluvaan solmuun.

Brentin algoritmin etuna Floydin algoritmiin verrattuna on, että se kutsuu funktiota f harvemmin. Floydin algoritmi kutsuu funktiota f ensimmäisessä silmukassa kolmesti joka kierroksella, kun taas Brentin algoritmi kutsuu funktiota f vain kerran kierrosta kohden.

Luku 17

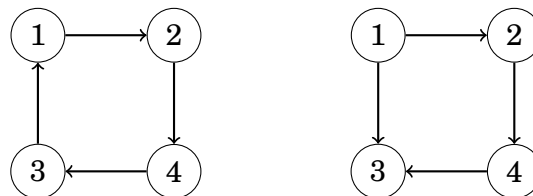
Vahvasti yhtenäisyys

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. Tässä luvussa opimme tehokkaan algoritmin tämän toteuttamiseen.

17.1 Käsitteitä

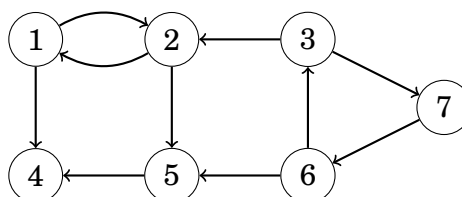
Verkko on vahvasti yhtenäinen (*strongly connected*), jos mistä tahansa solmusta on olemassa polku kaikkiin muihin solmuihin. Esimerkiksi seuraavassa kuvassa vasen verkko on vahvasti yhtenäinen, kun taas oikea verkko ei ole.



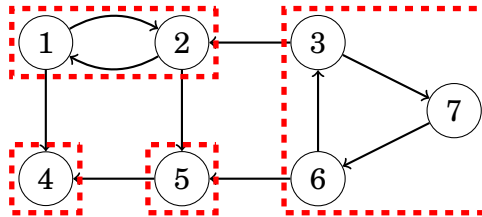
Oikeanpuoleinen verkko ei ole vahvasti yhtenäinen, koska esimerkiksi solmusta 2 ei ole polkua solmuun 1.

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.

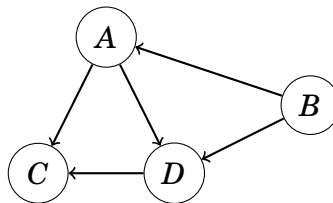
Esimerkiksi verkon



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 komponenttiverkoon voi soveltaa luvun 16.2 tyylistä dynaamista ohjelmointia.

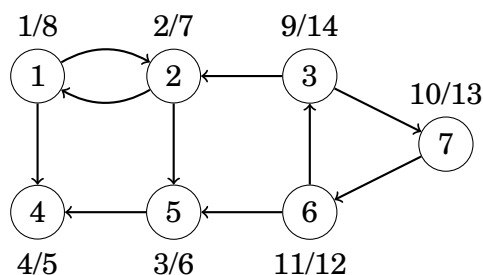
17.2 Kosarajun algoritmi

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

Algoritmin ensimmäinen vaihe muodostaa listan solmuista syvyysshaun käsittelyjärjestyksessä. Algoritmi käy solmut läpi yksi kerrallaan, ja jos solmua ei ole vielä käsitelty, algoritmi suorittaa solmusta alkaen syvyysshaun. Solmu lisätään listalle, kun syvyyshaku on käsitellyt kaikki siitä lähtevät kaaret.

Esimerkkiverkossa solmujen käsittelyjärjestys on:



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:

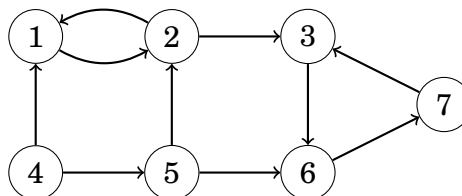
solmu	päättymisaika
4	5
5	6
2	7
1	8
6	12
7	13
3	14

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

Syvyyshaku 2

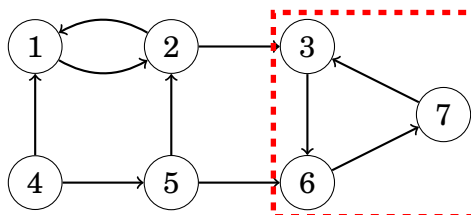
Algoritmin toinen vaihe muodostaa verkon vahvasti yhtenäiset komponentit. Ennen toista syvyyshakua algoritmi muuttaa jokaisen kaaren suunnan käänteiseksi. Tämä varmistaa, että toisen syvyyshaun aikana löydetään joka kerta vahvasti yhtenäinen komponentti, johon ei kuulu ylimääräisiä solmuja.

Esimerkkiverkko on käännettynä seuraava:



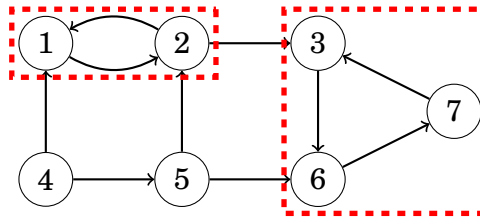
Tämän jälkeen algoritmi käy läpi solmut käänteisessä ensimmäisen syvyyshaun tuottamassa järjestyksessä. Jos solmu ei kuulu vielä komponenttiin, siitä alkaa uusi syvyyshaku. Solmun komponenttiin liitetään kaikki aiemmin käsittelemättömät solmut, joihin syvyyshaku pääsee solmusta.

Esimerkkiverkossa muodostuu ensin komponentti solmusta 3 alkaen:

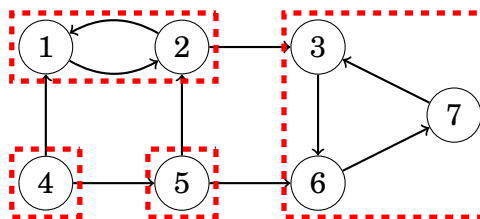


Huomaa, että kaarten kääntämisen ansiosta komponentti ei pääse ”vuotamaan” muihin verkon osiin.

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 ratkaisemiseen. 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, \dots, x_n) tai sen negaatio $(\neg x_1, \neg x_2, \dots, \neg x_n)$.

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.

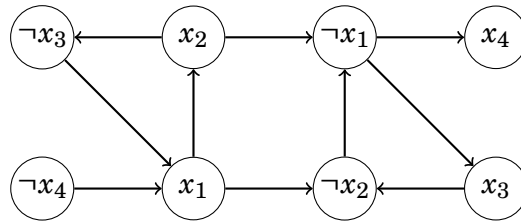
Vastaavasti lauseke

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

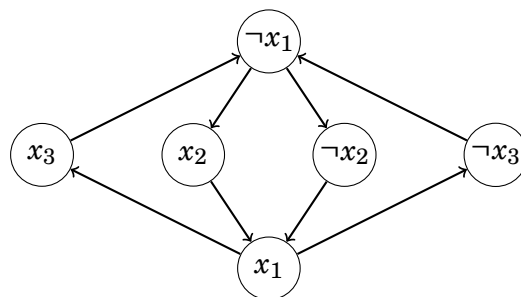
on epätosi riippumatta muuttujien valinnasta. Tämän näkee siitä, että muuttujalle x_1 ei ole mahdollista arvoa, joka ei tuottaisi ristiriitaa. Jos x_1 on tosi, pitäisi päteä sekä x_3 että $\neg x_3$, mikä on mahdotonta. Jos taas x_1 on epätosi, pitäisi päteä sekä x_2 että $\neg x_2$, mikä on myös mahdotonta.

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: $\neg a_i \rightarrow b_i$ sekä $\neg b_i \rightarrow a_i$. 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:



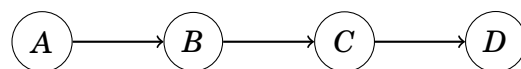
Verkon rakenne kertoo, onko 2SAT-ongelmalla ratkaisua. Jos on jokin muuttuja x_i niin, että x_i ja $\neg x_i$ ovat samassa vahvasti yhtenäisessä komponentissa, niin ratkaisua ei ole olemassa. Tällöin verkossa on polku sekä x_i :stä $\neg x_i$:ään että $\neg x_i$:stä x_i :ään, eli kumman tahansa arvon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

Luku 18

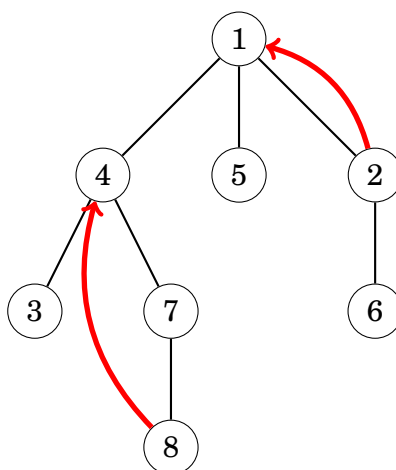
Puukyselyt

Tyypillisiä puukyselyitä ovat puun polkuihin ja alipuihin liittyvät kyselyt. Tämä luku esittelee tekniikoita, joiden avulla on mahdollista toteuttaa puukyselyjä tehokkaasti. Usein esiintyvä idea on muuttaa puu jollakin tavalla taulukoksi ja hyödyntää tehokkaita taulukon välikyselyitä.

18.1 Tehokas nouseminen

Tehtävä: Annettuna on juurellinen puu, jonka solmut on numeroitu $1 \dots n$ ja juurisolmu on 1. Tehtäväsi on vastata tehokkaasti kyselyihin muotoa ”mikä solmu on k askelta ylempänä solmua x ”.

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



Suoraviivainen tapa laskea funktion $f(x, k)$ arvo on kulkea puussa k askelta ylöspäin solmusta x alkaen. Tämän aikavaativuus on kuitenkin $O(n)$, koska on mahdollista, että puussa on ketju, jossa on $O(n)$ solmua.

Kuten luvussa 16.3.1, funktion $f(x, k)$ arvo on mahdollista laskea tehokkaasti ajassa $O(\log k)$ sopivan esikäsittelyn avulla. Ideana on laskea etukäteen kaikki arvot $f(x, k)$, joissa $k = 1, 2, 4, 8, \dots$ eli 2:n potenssi. Esimerkiksi yllä olevassa puussa muodostuu seuraava taulukko:

x	1	2	3	4	5	6	7	8
$f(x, 1)$	0	1	4	1	1	2	4	7
$f(x, 2)$	0	0	1	0	0	1	1	4
$f(x, 4)$	0	0	0	0	0	0	0	0
...								

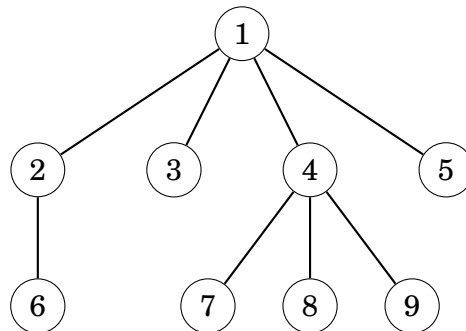
Taulukossa arvo 0 tarkoittaa, että nousemalla k askelta päätyy puun ulkopuolelle juurisolmun yläpuolelle.

Esilaskenta vie aikaa $O(n \log n)$, koska jokaisesta solmusta voi nousta korkeintaan n askelta ylöspäin. Tämän jälkeen minkä tahansa funktion $f(x, k)$ arvon saa laskettua ajassa $O(\log k)$ jakamalla nousun 2:n potenssin osiin.

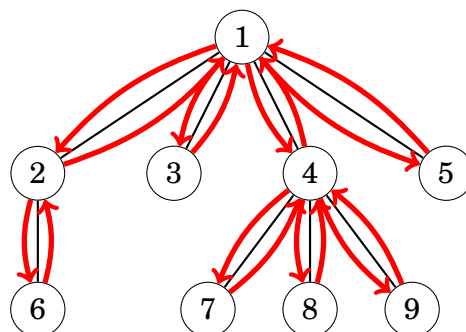
18.2 Solmutaulukko

Solmutaulukko sisältää juurellisen puun solmut siinä järjestyksessä kuin juuresta alkava syvyyshaku vierailee solmuissa.

Esimerkiksi puussa



syvyyshaku etenee



ja solmutaulukoksi tulee:

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

18.2.1 Alipuiden käsittely

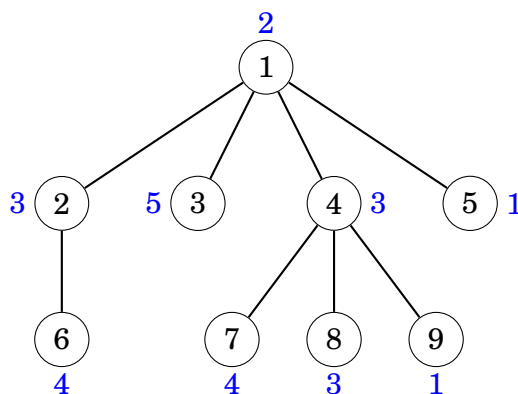
Solmutaulukossa jokaisen alipuun kaikki solmut ovat peräkkäin niin, että ensin on alipuun juurisolmu ja sitten kaikki muut alipuun solmut. Esimerkiksi äskeisessä taulukossa solmun 4 alipuuta vastaa seuraava taulukon osa:

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

Tämän ansiosta solmutaulukon avulla voi käsitellä tehokkaasti puun alipuihin liittyviä kyselyitä. Voimme ratkaista esimerkiksi seuraavan tehtävän:

Tehtävä: Annettuna on juurellinen puu, jossa on n solmua ja jokaisella solmulla on tietty arvo. Tehtäväsi on käsitellä kyselyt muotoa ”muuta solmun x arvoa” sekä ”laske arvojen summa solmun x alipuussa”.

Tarkastellaan seuraavaa puuta, jossa siniset luvut ovat solmujen arvoja. Esimerkiksi solmun 4 alipuun arvojen summa on $3 + 4 + 3 + 1 = 11$.



Ideana on luoda solmutaulukko, joka sisältää jokaisesta solmusta kolme tietoa: (1) solmun tunnus, (2) alipuun koko ja (3) solmun arvo. Esimerkiksi yllä olevasta puusta syntyy seuraava taulukko:

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

Tästä taulukosta alipuun solmujen arvojen summa selviää lukemalla ensin alipuun koko ja sitten sitä vastaavat solmut. Esimerkiksi solmun 4 alipuun arvojen summa selviää näin:

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

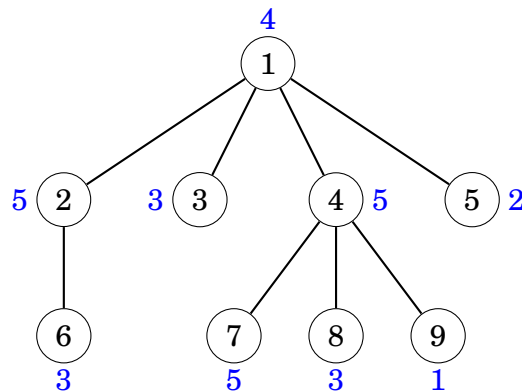
Viimeinen tarvittava askel on tallentaa solmujen arvot binääri-indeksipuu-
hun tai segmenttipuuhun. Tällöin sekä alipuun arvojen summan laskeminen
että solmun arvon muuttaminen onnistuvat ajassa $O(\log n)$, eli pystymme vas-
taamaan kyselyihin tehokkaasti.

18.2.2 Polkujen käsittely

Solmutaulukon avulla voi myös käsitellä tehokkaasti polkuja, jotka kulkevat
juuresta tiettyyn solmuun puussa. Näin on seuraavassa tehtävässä:

Tehtävä: Annettuna on juurellinen puu, jossa on n solmua ja jokaisella sol-
mulla on tietty arvo. Tehtäväsi on käsitellä kyselyt muotoa ”muuta solmun
 x arvoa” sekä ”laske arvojen summa juuresta solmuun x ”.

Esimerkiksi seuraavassa puussa polulla solmusta 1 solmuun 8 arvojen sum-
ma on $4 + 5 + 3 = 12$.



Ideana on muodostaa samanlaiset taulukot kuin alipuiden käsittelyssä mut-
ta tallentaa solmujen arvot erikoisella tavalla: kun taulukon kohdassa k olevan
solmun arvo on a , kohdan k arvoon lisätään a ja kohdan $k + c$ arvosta vähenne-
tään a , missä c on alipuun koko.

Esimerkiksi yllä olevaa puuta vastaa seuraava taulukko:

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

Esimerkiksi solmun 3 arvona on -5 , koska se on solmujen 2 ja 6 alipuiden jälkeinen solmu, mistä tulee arvoa $-5-3$, ja sen oma arvo on 3. Yhteensä solmun 3 arvo on siis $-5-3+3=-5$. Huomaa, että taulukossa on ylimääräinen kohta 10, johon on tallennettu vain juuren arvon vastaluku.

Nyt solmujen arvojen summa polulla juuresta alkaen selviää laskemalla kaikkien taulukon arvojen summa taulukon alusta solmuun asti. Esimerkiksi summa solmusta 1 solmuun 8 selviää näin:

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

Summaksi tulee $4+5+3-5+2+5-2=12$, mikä vastaa polun summaa $4+5+3=12$. Tämä laskentatapa toimii, koska jokaisen solmun arvo lisätään summaan, kun se tulee vastaan syvyysshaussa, ja vähennetään summasta, kun sen käsittely päättyy.

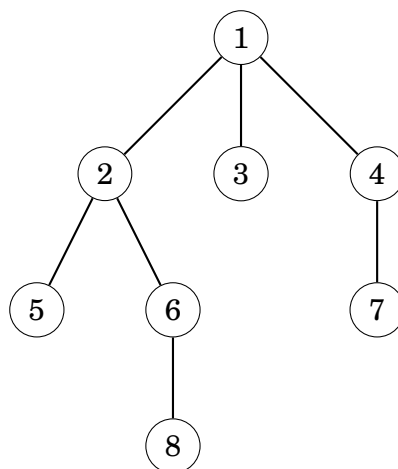
Alipuiden käsittelyä vastaavasti voimme tallentaa arvot binääri-indeksipuuhun tai segmenttipuuhun ja sekä polun summan laskeminen että arvon muuttaminen onnistuvat ajassa $O(\log n)$.

18.3 Alin yhteinen esivanhempi

Solmujen a ja b alin yhteinen esivanhempi (*lowest common ancestor*) on mahdollisimman alhaalla puussa oleva solmu, jonka alipuuhun kuuluvat molemmat solmut a ja b . Luonteva tehtävä on:

Tehtävä: Annettuna on puu, jossa on n solmua. Tehtäväsi on vastata kyselyihin ”mikä on solmujen a ja b alin yhteinen esivanhempi”.

Esimerkiksi puussa



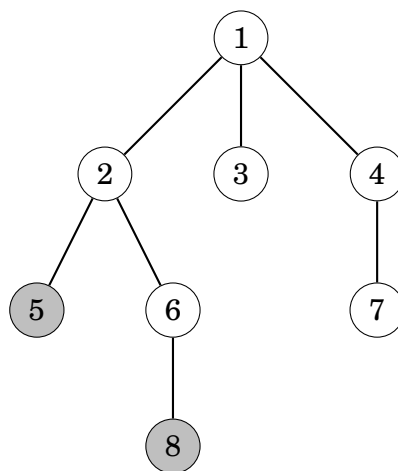
solmujen 5 ja 8 alin yhteinen esivanhempi on solmu 2 ja solmujen 3 ja 4 alin yhteinen esivanhempi on solmu 1.

Tutustumme seuraavaksi kahteen menetelmään alimman yhteisen esivanhemman selvittämiseen.

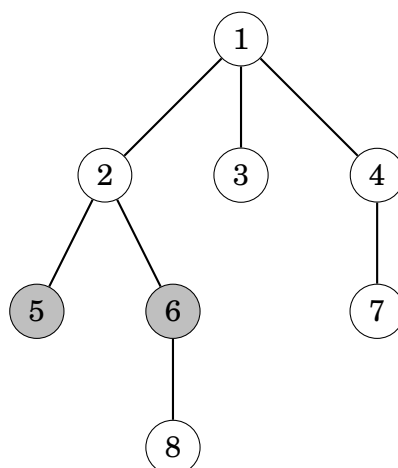
Menetelmä 1

Yksi tapa ratkaista tehtävä on hyödyntää tehokasta nousemista puussa. Tällöin alimman yhteisen esivanhemman etsiminen muodostuu kahdesta vaiheesta. Ensin nousetaan alemmasta solmusta samalle tasolle ylemmän solmun kanssa, sitten nousetaan rinnakkain kohti alinta yhteistä esivanhempaa.

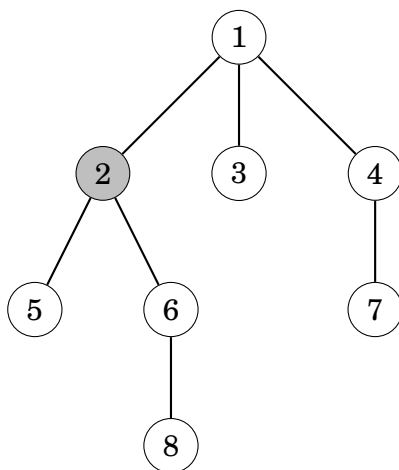
Tarkastellaan esimerkkinä solmujen 5 ja 8 alimman yhteisen esivanhemman etsimistä:



Solmu 5 on tasolla 3, kun taas solmu 8 on tasolla 4. Niinpä nousemme ensin solmusta 8 yhden tason ylemmäs solmuun 6:



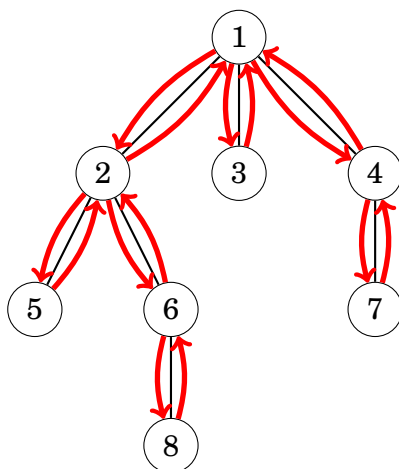
Tämän jälkeen etsitään pienin määrä tasoja, jotka nousemalla ylöspäin päädyimme samaan solmuun. Tässä tapauksessa riittää nousta yksi taso ylöspäin solmuun 2. Niinpä solmujen alin yhteinen esivanhempi on 2:



Menetelmä vaatii $O(n \log n)$ -aikaisen esikäsittelyn, jonka jälkeen minkä tahansa kahden solmun alin yhteinen esivanhempi selviää ajassa $O(\log n)$, koska kumpikin vaihe nousussa vie aikaa $O(\log n)$.

Menetelmä 2

Toinen tapa ratkaista tehtävä perustuu solmutaulukon käyttämiseen. Ideana on jälleen järjestää solmut syvyysshaun mukaan:



Erona aiempaan solmu lisätään kuitenkin solmutaulukkoon mukaan *aina*, kun syvyyshaku käy solmussa, eikä vain ensimmäisellä kerralla. Niinpä solmu esiintyy solmutaulukossa $x + 1$ kertaa, missä x on solmun lasten määrä, ja solmutaulukossa on yhteensä $2n - 1$ solmua.

Tallennamme solmutaulukkoon kaksi tietoa: (1) solmun tunnus ja (2) solmun taso puussa. Esimerkkipuuta vastaavat taulukot ovat:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Tämän taulukon avulla solmujen a ja b alin yhteinen esivanhempi selviää etsimällä taulukosta alimman tason solmu solmujen a ja b välissä. Esimerkiksi solmujen 5 ja 8 alin yhteinen esivanhempi löytyy seuraavasti:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
node	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

Solmu 5 on taulukossa kohdassa 3, solmu 8 on taulukossa kohdassa 6 ja alimman tason solmu välillä 3...6 on kohdassa 4 oleva solmu 2, jonka taso on 2. Niinpä solmujen 5 ja 8 alin yhteinen esivanhempi on solmu 2.

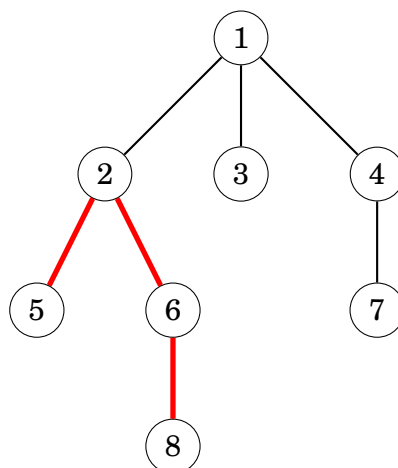
Alimman tason solmu välillä selviää ajassa $O(\log n)$, kun taulukon sisältö on tallennettu segmenttipuuhun. Myös aikavaativuus $O(1)$ on mahdollinen, koska taulukko on staattinen, mutta tälle on harvoin tarvetta. Kummassakin tapauksessa esikäsittely vie aikaa $O(n \log n)$.

Solmujen etäisyydet

Myös seuraava tehtävä palautuu alimman yhteisen esivanhemman hakuun:

Tehtävä: Annettuna on juurellinen puu, jonka solmut on numeroitu $1 \dots n$. Tehtäväsi on vastata kyselyihin ”laske solmujen a ja b etäisyys puussa”.

Valitaan ensin mikä tahansa solmu puun juureksi. Tämän jälkeen solmujen a ja b etäisyys on $d(a) + d(b) - 2 \cdot d(c)$, missä c on solmujen alin yhteinen esivanhempi ja $d(s)$ on etäisyys puun juuresta solmuun s . Esimerkiksi puussa



solmujen 5 ja 8 alin yhteinen esivanhempi on 2. Polku solmusta 5 solmuun 8 kulkee ensin ylöspäin solmusta 5 solmuun 2 ja sitten alaspäin solmusta 2 solmuun 8. Solmujen etäisyydet juuresta ovat $d(5) = 3$, $d(8) = 4$ ja $d(2) = 2$, joten solmujen 5 ja 8 etäisyys on $3 + 4 - 2 \cdot 2 = 3$.

Luku 19

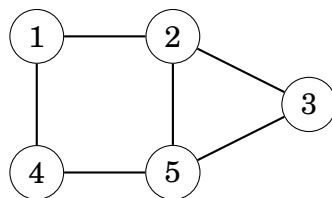
Polut ja kierrokset

Verkkoteorian syntyhetkenä pidetään vuotta 1736, jolloin matemaatikko Leonhard Euler tutki Königsbergin siltaongelmaa. Tehtävänä oli etsiä reitti, joka kulkisi tarkalleen kerran kaupungin seitsemän sillan yli. Euler esitti tilanteen verkkona ja osoitti, ettei halutunlaista reittiä ole olemassa.

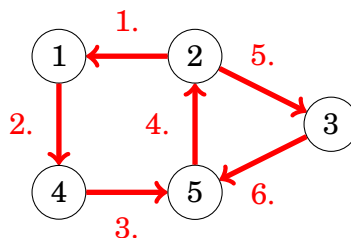
Eulerin analyysi tarjoaa yleisen menetelmän tutkia, onko verkossa polkua, joka kulkee tarkalleen kerran jokaista kaarta pitkin. Osoittautuu, että polun olemassaolon voi päätellä suoraan verkon solmujen asteista, ja jos polku on olemassa, sen pystyy myös muodostamaan tehokkaasti.

19.1 Käsitteitä

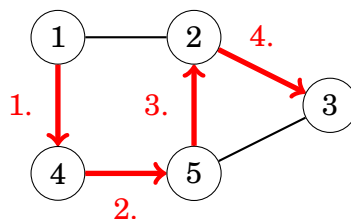
Eulerin polku (*Eulerian path*) on verkossa oleva polku, joka kulkee tarkalleen kerran jokaista kaarta pitkin. Esimerkiksi verkossa



on Eulerin polku solmusta 2 solmuun 5:

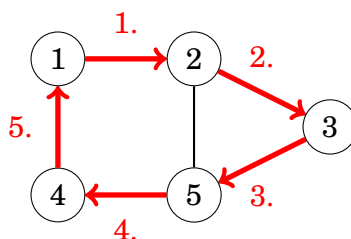


Vastaavasti Hamiltonin polku (*Hamiltonian path*) on verkossa oleva polku, joka kulkee tarkalleen kerran jokaisen solmun kautta. Yllä olevassa verkossa on esimerkiksi Hamiltonin polku solmusta 1 solmuun 3:



Jos Eulerin polun alku- ja loppusolmu on sama, kyseessä on Eulerin kierros (*Eulerian circuit*). Vastaavasti jos Hamiltonin polun alku- ja loppusolmu on sama, kyseessä on Hamiltonin kierros (*Hamiltonian circuit*).

Yllä olevassa verkossa ei ole Eulerin kierrosta, mutta siinä on Hamiltonin kierros, jonka alku- ja loppusolmu on solmu 1:



Vaikka Eulerin ja Hamiltonin polku ovat samantapaisia käsitteitä, niihin liittyy laskennallisesti hyvin erilaisia ongelmia. Näemme seuraavaksi, että yksinkertainen verkon rakenteeseen liittyvä ehto ratkaisee, onko verkossa Eulerin polkua, ja myönteisessä tapauksessa on helppoa etsiä jokin polku verkosta.

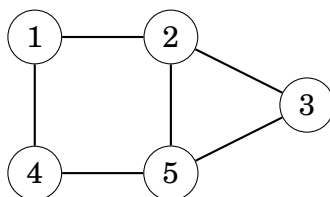
Hamiltonin polun tapauksessa tilanne on kuitenkin täysin toinen: ei tunneta mitään tehokasta menetelmää, jolla voi selvittää, onko verkossa Hamiltonin polkua, vaan kyseessä on NP-täydellinen ongelma. Ainoat tunnetut yleiset menetelmät Hamiltonin polun muodostamiseen perustuvat raakaan voimaan.

19.2 Eulerin polku

Osoittautuu, että Eulerin polun ja kierroksen olemassaolo riippuu verkon solmujen asteista. Solmun aste on sen naapurien määrä eli niiden solmujen määrä, jotka ovat yhteydessä solmuun kaarella.

Jos verkko on suuntaamaton, siinä on Eulerin kierros tarkalleen silloin, kun kaikki kaaret ovat samassa yhtenäisessä komponentissa ja jokaisen solmun aste on parillinen. Lisäksi jos tasan kahden solmun aste on pariton, verkossa ei ole Eulerin kierrosta, mutta siinä on Eulerin polku, jonka päätesolmut ovat paritonasteiset solmut.

Esimerkiksi verkossa



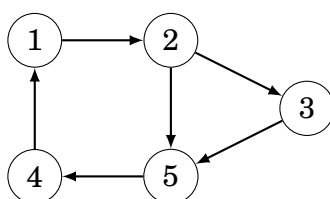
solmujen 1, 3 ja 4 aste on 2 ja solmujen 2 ja 5 aste on 3. Tarkalleen kahden

solmun aste on pariton, joten verkossa on Eulerin polku solmujen 2 ja 5 välillä, mutta verkossa ei ole Eulerin kierrosta.

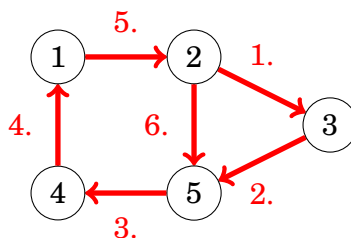
Jos verkko on suunnattu, tilanne on hieman hankalampi. Silloin Eulerin polun ja kierroksen olemassaoloon vaikuttavat solmujen lähtö- ja tuloasteet. Solmun lähtöaste on solmusta lähtevien kaarten määrä, ja vastaavasti solmun tuloaste on solmuun tulevien kaarten määrä.

Suunnatussa verkossa on Eulerin kierros, jos kaikki kaaret ovat samassa vahvasti yhtenäisessä komponentissa ja joka solmun lähtöaste ja tuloaste on sama. Lisäksi verkossa on Eulerin polku, jos alkusolmussa lähtöaste on yhden suurempi kuin tuloaste, loppusolmussa tuloaste on yhden suurempi kuin lähtöaste ja muissa solmuissa lähtö- ja tuloasteet ovat samat.

Esimerkiksi verkossa



solmuissa 1, 3 ja 4 sekä lähtöaste että tuloaste on 1. Solmussa 2 tuloaste on 1 ja lähtöaste on 2, kun taas solmussa 5 tuloaste on 2 ja lähtöaste on 1. Niinpä verkossa on Eulerin polku solmusta 2 solmuun 5:



Algoritmi

Seuraavaksi esitettävä algoritmi muodostaa Eulerin kierroksen suuntaamattomassa verkossa. Algoritmi olettaa, että kaikki kaaret ovat samassa komponentissa ja jokaisen solmun aste on parillinen.

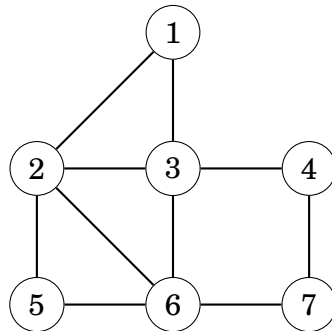
Jos verkossa on kaksi paritonasteista solmua, samalla algoritmilla voi myös muodostaa Eulerin polun lisäämällä kaaren paritonasteisten solmujen välille. Tämän jälkeen verkosta voi etsiä Eulerin kierroksen, ja lopuksi Eulerin kierroksesta saa Eulerin polun poistamalla ylimääräisen kaaren.

Algoritmi muodostaa ensin verkkoon jonkin kierroksen, johon kuuluu osa verkon kaarista. Sen jälkeen algoritmi alkaa laajentaa kierrosta lisäämällä sen osaksi uusia alikierroksia. Tämä jatkuu niin kauan, kunnes kaikki kaaret kuuluvat kierrokseen ja siitä on tullut Eulerin kierros.

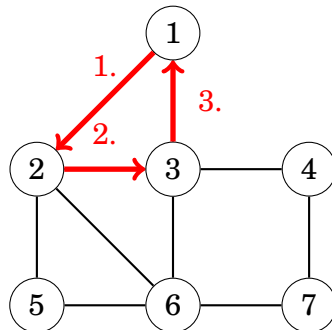
Algoritmi laajentaa kierrosta valitsemalla jonkin kierrokseen kuuluvan solmun x , jonka kaikki kaaret eivät ole vielä mukana kierroksessa. Algoritmi muodostaa solmusta x alkaen uuden polun kulkien vain sellaisia kaaria, jotka eivät

ole mukana kierroksessa. Koska jokaisen solmun aste on parillinen, ennemmin tai myöhemmin polku palaa takaisin solmuun.

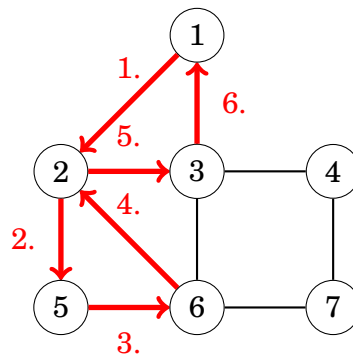
Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



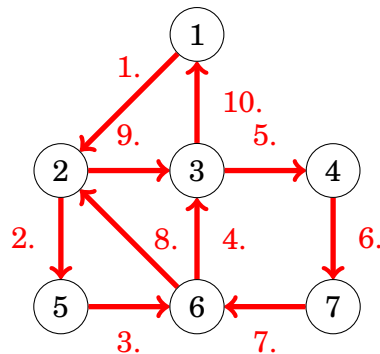
Oletetaan, että algoritmi aloittaa ensimmäisen kierroksen solmusta 1. Siitä syntyy kierros $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



Seuraavaksi algoritmi lisää mukaan kierroksen $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$:



Lopuksi algoritmi lisää mukaan kierroksen $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$:



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

Toteutus

Edellä kuvattu algoritmi on mukavaa toteuttaa niin, että solmujen vieruslistat on tallennettu joukkoina

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

jolloin verkosta on helppoa poistaa kahden solmun välinen kaari, kun se tulee mukaan kierrokseen.

Seuraava koodi muodostaa Eulerin kierroksen solmusta x alkaen. Se käyttää apuna pinoa s , jossa on aluksi vain kierroksen alkusolmu. Jos pinon ylimmän solmun u aste on 0, se lisätään Eulerin kierrokseen. Muuten pinon päälle lisätään uusi alikierros solmusta u alkaen ja kaikki alikierrokseen kuuluvat kaaret poistetaan verkosta.

```
stack<int> s;
s.push(x);
while (!s.empty()) {
    int u = s.top(); s.pop();
    if (v[u].size() == 0) {
        // lisää solmu u Eulerin kierrokseen
    } else {
        int a = u;
        s.push(a);
        do {
            int b = *v[a].begin();
            v[a].erase(b);
            v[b].erase(a);
            s.push(b);
            a = b;
        } while (a != u);
    }
}
```

Toteutuksen aikavaativuus on $O(n + m \log n)$, koska se käy läpi kaikki solmut ja kaaret ja kunkin kaaren poistaminen vie aikaa $O(\log n)$. Myös toteutus ajassa $O(n + m)$ on mahdollista mutta vaikeampaa. Tämä vaatii verkon esittämistä niin, että kaaria pystyy poistamaan ajassa $O(1)$.

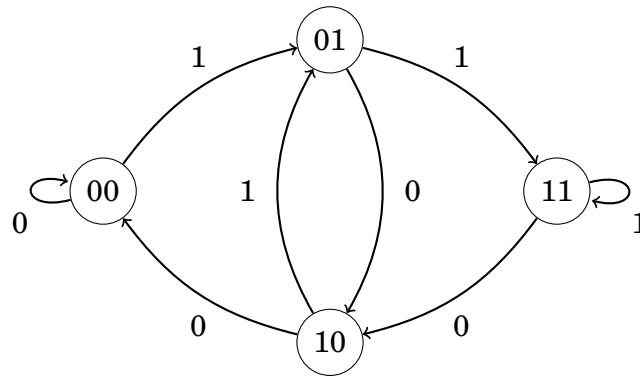
19.3 De Bruijnin jono

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 merkkijono vastaa Eulerin kierrosta sopivasti muodostetussa verkossa. Tällaista merkkijonoa kutsutaan de Bruijin 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 merkkijonon, joka sisältää kaikki n merkin yhdistelmät, kun mukaan otetaan aloitussolmun merkit sekä kussakin kaaressa olevat merkit. Aloitussolmussa on $n - 1$ merkkiä ja kaarissa on k^n merkkiä, joten tuloksena on lyhin mahdollinen merkkijono.

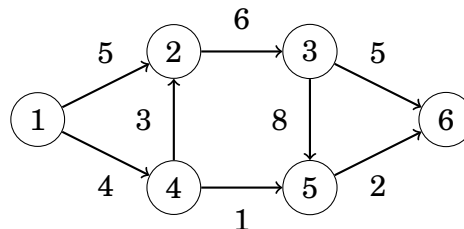
Luku 20

Virtauslaskenta

Virtauslaskenta on verkkoteorian osa-alue, jonka keskeinen ongelma on verkon maksimivirtauksen laskeminen. Tässä luvussa tutustumme ensin algoritmiin, jonka avulla voi laskea verkon maksimivirtauksen, ja sen jälkeen perehdymme erilaisiin virtauslaskennan sovelluksiin.

20.1 Käsitteitä

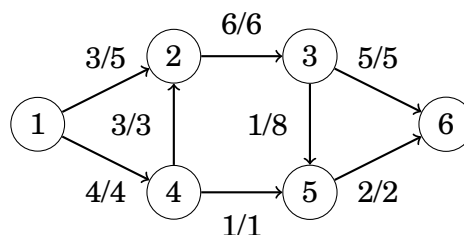
Oletamme, että annettuna on suunnattu, painotettu verkko, jossa on valittu tietty alkusolmu ja loppusolmu. Käytämme esimerkkinä seuraavaa verkkoa, jossa solmu 1 on alkusolmu ja solmu 6 on loppusolmu:



Maksimivirtaus

Virtaus (*flow*) lähtee liikkeelle alkusolmusta ja päättyy loppusolmuun. Kunkin kaaren paino on kapasiteetti, joka ilmaisee, kuinka paljon virtausta kaaren kautta voi kulkea. Kaikissa solmuissa alkusolmuun tulevan ja loppusolmusta lähtevän virtauksen on oltava yhtä suuri.

Maksimivirtaus (*maximum flow*) on suurin mahdollinen virtaus verkossa. Esimerkkiverkossa maksimivirtauksen suuruus on 7:



Merkintä v/k kaareissa tarkoittaa, että kaareissa kulkee virtausta v ja kaaren kapasiteetti on k . Virtauksen suuruus on 7, koska alkusolmusta lähtevä virtaus on $3 + 4 = 7$ ja loppusolmuun saapuva virtaus on $5 + 2 = 7$.

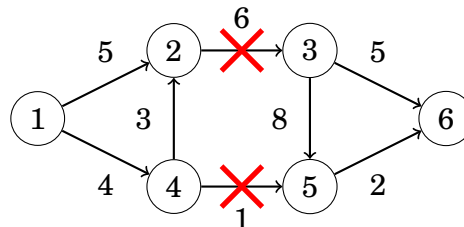
Huomaa, että jokaisessa välisolmussa tulevan ja lähtevän virtauksen määrä on sama. Esimerkiksi solmuun 2 tulee virtausta $3 + 3 = 6$ yksikköä solmuista 1 ja 4 ja siitä lähtee virtausta 6 yksikköä solmuun 3.

Virtaus 7 on verkon maksimivirtaus, koska verkon rakenteesta johtuen ei ole tapaa kuljettaa enempää virtausta verkossa.

Minimileikkaus

Leikkaus (*cut*) jakaa verkon solmut kahteen osaan niin, että alkusolmu ja loppusolmu ovat eri osissa. Leikkauksen paino on niiden kaarten yhteispaino, jotka kulkevat alkuosasta loppuosaan.

Minimileikkaus (*minimum cut*) on leikkaus, jonka paino on pienin mahdollinen. Esimerkkiverkossa minimileikkaus on painoltaan 7:



Tässä leikkauksessa alkuosassa ovat solmut $\{1, 2, 4\}$ ja loppuosassa ovat solmut $\{3, 5, 6\}$. Alkuosasta loppuosaan kulkevat kaaret $2 \rightarrow 3$ ja $4 \rightarrow 5$, joiden yhteispaino on $6 + 1 = 7$.

Ei ole sattumaa, että yllä olevassa verkossa sekä maksimivirtauksen suuruus että minimileikkauksen paino on 7. Virtauslaskennan keskeinen tulos on, että verkon maksimivirtaus ja minimileikkaus ovat *aina* yhtä suuret, eli käsitteet kuvaavat saman asian kahta eri puolta.

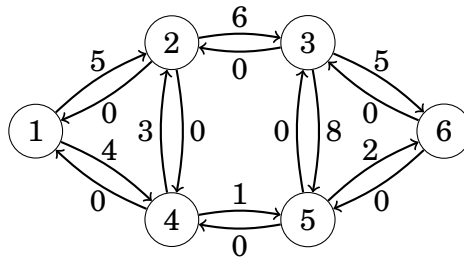
Seuraavaksi tutustumme Ford-Fulkersonin algoritmiin, jolla voi etsiä verkon maksimivirtauksen ja minimileikkauksen. Algoritmi auttaa myös ymmärtämään, *miksi* maksimivirtaus ja minimileikkaus ovat yhtä suuret.

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 pystyy vielä kulkemaan. Aluksi alkuperäisen verkon kaarilla on painona niiden kapasiteetti ja käänteisillä kaarilla on painona 0.

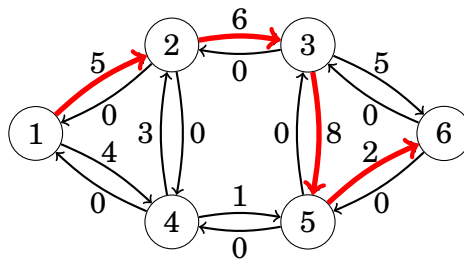
Esimerkkiverkosta syntyy seuraava verkko:



Algoritmin toiminta

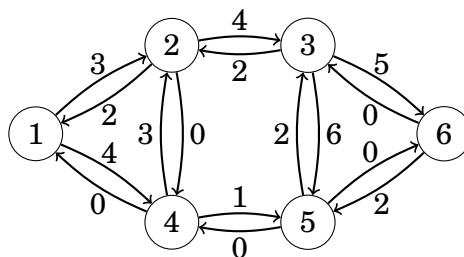
Ford-Fulkersonin algoritmi etsii verkosta joka vaiheessa polun, joka alkaa alkusolmusta, päättyy loppusolmuun ja jossa jokaisen kaaren paino on positiivinen. Jos vaihtoehtoja on useita, mikä tahansa valinta kelpaa.

Esimerkkiverkossa voimme valita vaikkapa seuraavan polun:



Polun valinnan jälkeen virtaus lisääntyy x yksikköä, jossa x on pienin kaaren kapasiteetti polulla. Samalla jokaisen polulla olevan kaaren kapasiteetti vähenee x :llä ja jokaisen käänteisen kaaren kapasiteetti kasvaa x :llä.

Yllä valitussa polussa kaarten kapasiteetit ovat 5, 6, 8 ja 2. Pienin kapasiteetti on 2, joten virtaus kasvaa 2:lla ja verkko muuttuu seuraavasti:



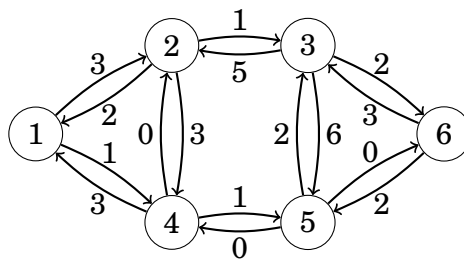
Muutoksessa on ideana, että virtauksen lisääminen vähentää polkuun kuuluvien kaarten kykyä välittää virtausta. Toisaalta virtausta on mahdollista peruuttaa myöhemmin käyttämällä käänteisiä kaaria, jos osoittautuu, että virtausta on järkevää reitittää verkossa toisella tavalla.

Algoritmi kasvattaa virtausta niin kauan, kuin verkossa on olemassa polku alkusolmusta loppusolmuun positiivisia kaaria pitkin. Tässä tapauksessa voimme valita seuraavan polun vaikkapa näin:

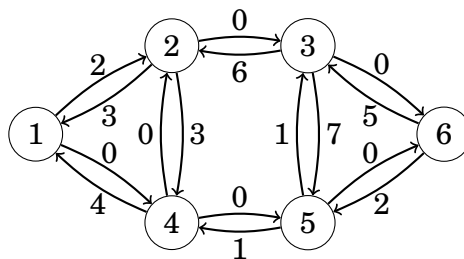


Tämän polun pienin kapasiteetti on 3, joten polku kasvattaa virtausta 3:lla ja kokonaisvirtaus polun käsittelyn jälkeen on 5.

Nyt verkko muuttuu seuraavasti:



Maksimivirtaus tulee valmiiksi lisäämällä virtausta vielä polkujen $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ ja $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ 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. Niinpä algoritmi pysähtyy ja verkon maksimivirtaus on 7.

Polun valinta

Ford-Fulkersonin algoritmi ei ota kantaa siihen, millä tavoin virtausta kasvattava polku valitaan verkossa. Valintatavasta riippumatta algoritmi pysähtyy ja tuottaa maksimivirtauksen ennemmin tai myöhemmin, mutta polun valinnalla on vaikutusta algoritmin tehokkuuteen.

Yksinkertainen tapa on valita virtausta kasvattava polku syvyysshaulla. Tämä toimii usein hyvin, mutta pahin tapaus on, että jokainen polku kasvattaa virtausta vain 1:llä ja algoritmi toimii hitaasti. Seuraavaksi käymme läpi kaksi tapaa polun valintaan, jotka estävät tämän ilmiön.

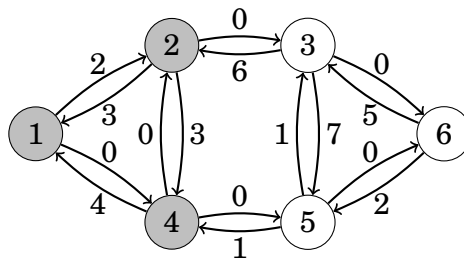
Edmonds-Karpin algoritmi on Ford-Fulkersonin algoritmin toteutus, jossa virtausta kasvattava polku valitaan aina niin, että siinä on mahdollisimman vähän kaaria. Tämä onnistuu etsimällä polku syvyysshaun sijasta leveyshaulla. Osoittautuu, että tämä varmistaa virtauksen kasvamisen nopeasti ja maksimivirtauksen etsiminen vie aikaa $O(m^2n)$.

Skaalaava algoritmi asettaa minimiarvon, joka on ensin alkusolmusta lähtevien kaarten kapasiteettien summa c . Joka vaiheessa verkosta etsitään syvyysshaulla polku, jonka jokaisen kaaren kapasiteetti on vähintään minimiarvo. Aina jos kelpoollista polkua ei löydy, minimiarvo jaetaan 2:lla, kunnes lopuksi minimiarvo on 1. Algoritmin aikavaativuus on $O(m^2 \log c)$.

Käytännössä skaalaava algoritmi on mukavampi koodattava, koska siinä riittää etsiä polku syvyysshaulla. Molemmat algoritmit ovat yleensä aina riittävän nopeita ohjelmointikisoissa esiintyviin tehtäviin.

Minimileikkaus

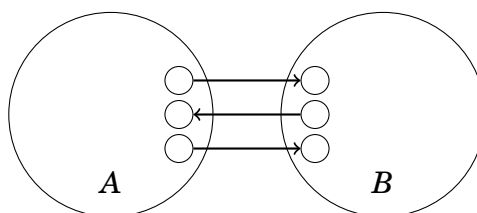
Osoittautuu, että kun Ford-Fulkersonin algoritmi on saanut valmiiksi maksimivirtauksen, se on tuottanut samalla minimileikkauksen. Olkoon A niiden solmujen joukko, joihin verkossa pääsee alkusolmusta positiivisia kaaria pitkin. Esimerkkiverkossa A sisältää solmut 1, 2 ja 4:



Nyt minimileikkauksen muodostavat ne alkuperäisen verkon kaaret, jotka kulkevat joukosta A joukon A ulkopuolelle ja joiden kapasiteetti on täysin käytetty maksimivirtauksessa. Tässä verkossa kyseiset kaaret ovat $2 \rightarrow 3$ ja $4 \rightarrow 5$, jotka tuottavat minimileikkauksen $6 + 1 = 7$.

Miksi sitten algoritmin tuottama virtaus ja leikkaus ovat varmasti maksimivirtaus ja minimileikkaus? Syynä tähän on, että virtauksen suuruus on *aina* enintään yhtä suuri kuin leikkauksen paino. Niinpä kun virtaus ja leikkaus ovat yhtä suuret, ne ovat varmasti maksimivirtaus ja minimileikkaus.

Tarkastellaan mitä tahansa verkon leikkausta, jossa alkusolmu kuuluu osaan A , loppusolmu kuuluu osaan B ja osien välillä kulkee kaaria:



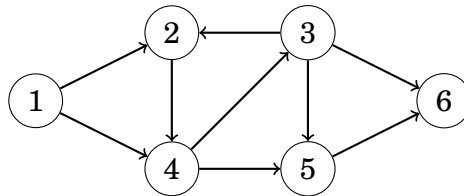
Leikkauksen paino on niiden kaarten painojen summa, jotka kulkevat osasta A osaan B . Tämä on yläraja sille, kuinka suuri verkossa oleva virtaus voi olla, koska virtauksen täytyy edetä osasta A osaan B . Niinpä maksimivirtaus on pienempi tai yhtä suuri kuin mikä tahansa verkon leikkaus.

Toisaalta Ford-Fulkersonin algoritmi tuottaa virtauksen, joka on tarkalleen yhtä suuri kuin verkossa oleva leikkaus. Niinpä tämän virtauksen on oltava maksimivirtaus ja vastaavasti leikkauksen on oltava minimileikkaus.

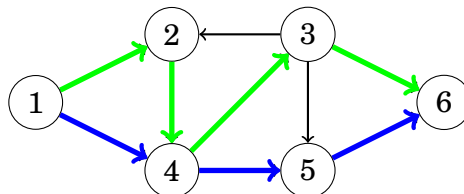
20.3 Rinnakkaiset polut

Ensimmäisenä virtauslaskennan sovelluksena tarkastelemme tehtävää, jossa tavoitteena on muodostaa mahdollisimman monta rinnakkaista polkua verkon alkusolmusta loppusolmuun. Vaatimuksena on, että jokainen verkon kaari esiintyy enintään yhdellä polulla.

Esimerkiksi verkossa

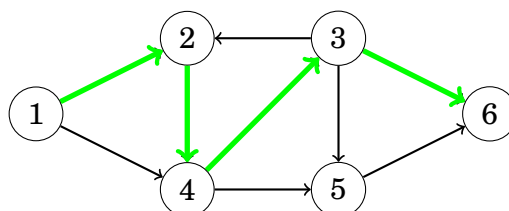


pystyy muodostamaan kaksi rinnakkaista polkua solmusta 1 solmuun 6. Tämä toteutuu valitsemalla polut $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ ja $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$:



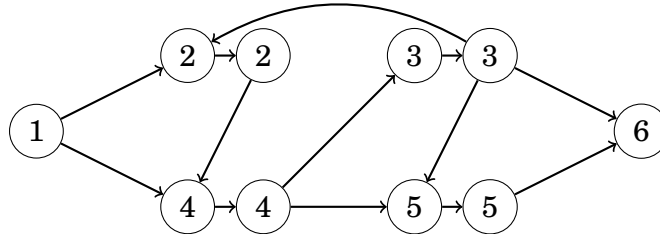
Osoittautuu, että suurin rinnakkaisten polkujen määrä on yhtä suuri kuin maksimivirtaus verkossa, jossa jokaisen kaaren kapasiteetti on 1. Kun maksimivirtaus on muodostettu, rinnakkaiset polut voi löytää ahneesti etsimällä alkusolmusta loppusolmuun kulkevia polkuja.

Tarkastellaan sitten tehtävän muunnelmää, jossa jokainen solmu (alku- ja loppusolmuja lukuun ottamatta) saa esiintyä enintään yhdellä polulla. Tämän rajoituksen seurauksena äskeisessä verkossa voi muodostaa vain yhden polun, koska solmu 4 ei voi esiintyä monella polulla:

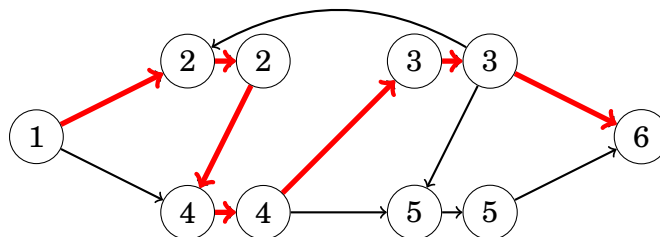


Tavallinen keino rajoittaa solmun kautta kulkevaa virtausta on jakaa solmu tulosolmuksi ja lähtösolmuksi. Kaikki solmuun tulevat kaaret saapuvat tulosolmuun ja kaikki solmusta lähtevät kaaret poistuvat lähtösolmusta. Lisäksi tulosolmusta lähtösolmuun on kaari, jossa on haluttu kapasiteetti.

Tässä tapauksessa verkosta tulee seuraava:



Tämän verkon maksimivirtaus on:



Tämä tarkoittaa, että verkossa on mahdollista muodostaa vain yksi polku alkusolmusta lähtösolmuun, kun sama solmu ei saa esiintyä monessa polussa.

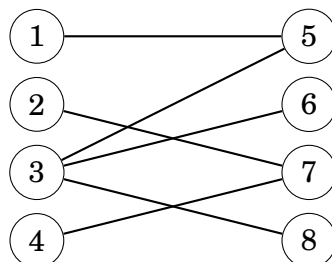
20.4 Maksimiparitus

Verkon paritus (*matching*) on kokoelma kaaria, jotka on valittu niin, että jokainen verkon solmu esiintyy enintään yhden paritukseen kuuluvan kaaren päätesolmuna. Maksimiparitus (*maximum matching*) on puolestaan paritus, jossa parien määrä on mahdollisimman suuri.

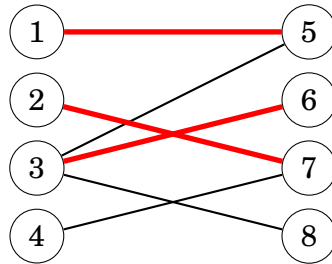
Maksimiparituksen etsimiseen yleisessä verkossa on olemassa polynominen algoritmi, mutta se on hyvin monimutkainen. Tässä luvussa keskitymmekin tilanteeseen, jossa verkko on kaksijakoinen. Tällöin maksimiparituksen pystyy etsimään helposti virtauslaskennan avulla.

20.4.1 Maksimiparituksen etsiminen

Kaksijakoinen verkko voidaan esittää aina niin, että kukin verkon solmu on vasemmalla tai oikealle puolella ja kaikki verkon kaaret kulkevat puolten välillä. Tarkastellaan esimerkkinä seuraavaa verkkoa:

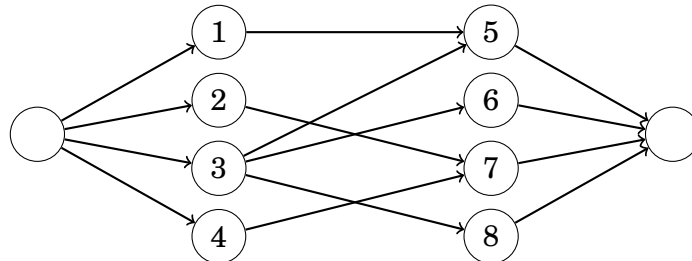


Tässä verkossa maksimiparituksen koko on 3:



Kaksijakoisen verkon maksimiparitus vastaa aina maksimivirtausta verkossa, johon on lisätty alkusolmu ja loppusolmu. Alkusolmusta on kaari jokaiseen vasemman puolen solmuun, ja vastaavasti loppusolmuun on kaari jokaisesta oikean puolen solmusta. Jokaisen kaaren kapasiteettina on 1.

Esimerkissä tuloksena on seuraava verkko:



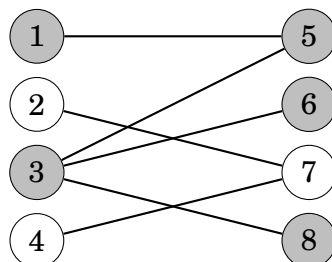
Tämän verkon maksimivirtaus on yhtä suuri kuin alkuperäisen verkon maksimiparitus, koska virtaus muodostuu joukosta polkuja alkusolmusta loppusolmuun ja jokainen polku ottaa mukaan uuden kaaren paritukseen. Tässä tapauksessa maksimivirtaus on 3, joten maksimiparitus on myös 3.

20.4.2 Hallin lause

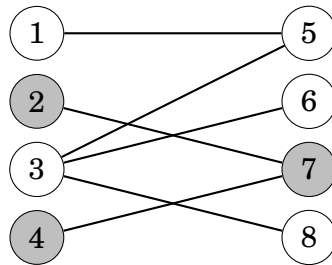
Hallin lause antaa ehdon, milloin kaksijakoiseen verkkoon voidaan muodostaa paritus, joka sisältää kaikki toisen puolen solmut. Jos kummallakin puolella on yhtä monta solmua, Hallin lause kertoo, voidaanko muodostaa täydellinen paritus (*perfect matching*), jossa kaikki solmut paritetaan keskenään.

Oletetaan, että haluamme muodostaa parituksen, johon kuuluvat kaikki vasemman puolen solmut. Olkoon X jokin joukko vasemman puolen solmuja ja joukko $f(X)$ ne oikean puolen solmut, jotka ovat yhteydessä joukon X solmuihin. Hallin lauseen mukaan paritus on mahdollinen tarkalleen silloin, kun mille tahansa joukolle X pätee $|X| \leq |f(X)|$.

Tarkastellaan Hallin lauseen merkitystä esimerkiverkossa. Valitaan ensin $X = \{1, 3\}$, jolloin $f(X) = \{5, 6, 8\}$:



Tämä täyttää Hallin lauseen ehdon, koska $|X| = 2$ ja $|f(X)| = 3$. Valitaan sitten $X = \{2, 4\}$, jolloin $f(X) = \{7\}$:



Tässä tapauksessa $|X| = 2$ ja $|f(X)| = 1$, joten Hallin lauseen ehto ei pidä paikkaansa. Tämä tarkoittaa, että ei ole mahdollista muodostaa paritusta, jossa ovat mukana kaikki vasemman puolen solmut (eli täydellistä paritusta). Tämä on odotettu tulos, koska verkon maksimiparitus on 3 eikä 4.

Jos Hallin lauseen ehto ei päde, osajoukko X kertoo syyn sille, miksi paritusta ei voi muodostaa. Koska X sisältää enemmän solmuja kuin $f(X)$, kaikille X :n solmuille ei riitä paria oikealta. Esimerkiksi yllä molemmat solmut 2 ja 4 tulisi yhdistää solmuun 7, mutta tämä ei ole mahdollista.

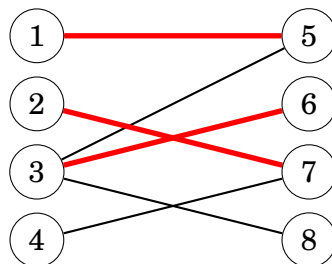
20.4.3 Königin lause

Solmupeite (*vertex cover*) on sellainen joukko verkon solmuja, että jokaisesta verkon kaaresta ainakin toinen kaaren päätesolmuista kuuluu joukkoon. Riippumaton joukko (*independent set*) on puolestaan joukko verkon solmuja, jossa minkään solmuparin välillä ei ole kaarta.

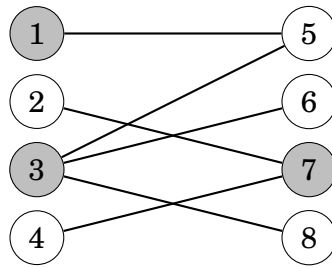
Jokaista verkon solmupeitettä vastaa riippumaton joukko, joka muodostuu niistä solmuista, jotka eivät kuulu solmupeitteeseen. Niinpä jos verkossa on solmupeite, jossa on x solmua, niin siinä on myös riippumaton joukko, jossa on $n - x$ solmua. Vastaava riippuvuus pätee myös toiseen suuntaan.

Yleisessä verkossa pienimmän solmupeitteen ja suurimman riippumattoman joukon etsiminen ovat NP-vaikeita ongelmia. Kuitenkin kaksijakoisessa verkossa ongelmat ratkeavat tehokkaasti, koska Königin lauseen nojalla pienin solmupeite on yhtä suurin kuin maksimiparitus.

Esimerkiksi seuraavan verkon maksimiparituksen koko on 3:

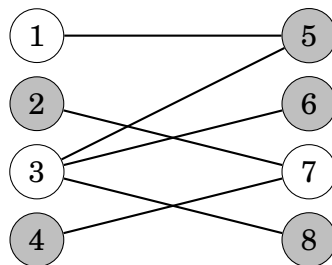


Niinpä myös pienimmän solmupeitteen koko on 3. Solmupeite voidaan muodostaa valitsemalla siihen solmut $\{1, 3, 7\}$:



Pienin solmupeite muodostuu aina niin, että jokaisesta maksimiparituksen kaaresta toinen kaaren päätesolmuista kuuluu peitteeseen.

Pienimmästä solmupeitteestä saadaan suurin riippumaton joukko valitsemalla kaikki solmut, jotka eivät kuulu peitteeseen. Esimerkiksi yllä olevassa verkossa suurin riippumaton joukko on seuraava:



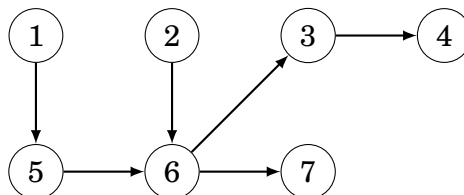
20.5 Polkupeitteet

Polkupeite (*path cover*) on joukko verkon polkuja, jotka on valittu niin, että jokainen verkon solmu kuuluu ainakin yhteen polkuun. Seuraavaksi näemme, miten virtauslaskennan avulla voi etsiä pienimmän polkupeitteen suunnatussa, syklittömässä verkossa.

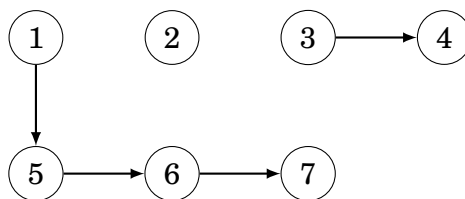
Polkupeitteestä on kaksi muunnelmaa: Solmuerillinen peite on polkupeite, jossa jokainen verkon solmu esiintyy tasan yhdessä polussa. Yleinen peite taas on polkupeite, jossa sama solmu voi esiintyä useammassa polussa. Kummassakin tapauksessa pienin polkupeite löytyy samanlaisella idealla.

20.5.1 Solmuerillinen peite

Tarkastellaan esimerkkinä seuraavaa verkkoa:



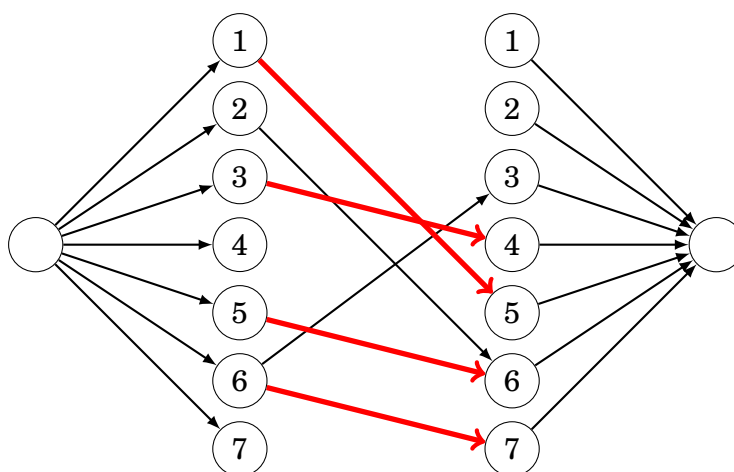
Tässä tapauksessa pienin solmuerillinen polkupeite muodostuu kolmesta polusta. Voimme valita polut esimerkiksi seuraavasti:



Huomaa, että yksi poluista sisältää vain solmun 2, eli on sallittua, että polussa ei ole kaaria.

Polkupeitteen etsiminen voidaan tulkita paritusongelmana verkossa, jossa jokaista alkuperäisen verkon solmua vastaa kaksi solmua: vasen ja oikea solmu. Vasemmasta solmusta oikeaan solmuun on kaari, jos tällainen kaari esiintyy alkuperäisessä verkossa. Ideana on, että paritus määrittää, mitkä solmut ovat yhteydessä toisiinsa poluissa.

Esimerkkiverkossa tilanne on seuraava:

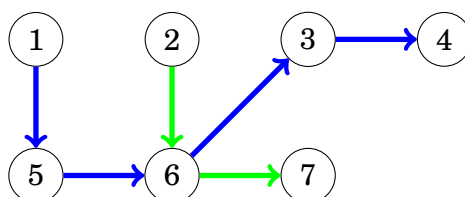


Tässä tapauksessa maksimiparitukseen kuuluu neljä kaarta, jotka vastaavat alkuperäisen verkon kaaria $1 \rightarrow 5$, $3 \rightarrow 4$, $5 \rightarrow 6$ ja $6 \rightarrow 7$. Niinpä pienin solmuerillinen polkupeite syntyy muodostamalla polut kyseisten kaarten avulla.

Pienimmän polkupeitteen koko on $n - c$, jossa n on verkon solmujen määrä ja c on maksimiparituksen kaarten määrä. Esimerkiksi yllä olevassa verkossa pienimmän polkupeitteen koko on $7 - 4 = 3$.

20.5.2 Yleinen peite

Yleisessä polkupeitteessä sama solmu voi kuulua moneen polkuun, minkä ansiosta tarvittava polkujen määrä saattaa olla pienempi. Esimerkkiverkossa pienin yleinen polkupeite muodostuu kahdesta polusta seuraavasti:



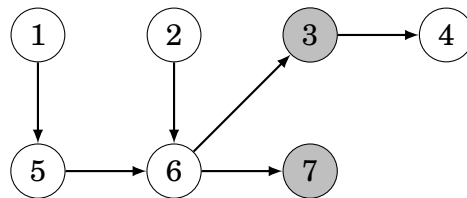
Tässä verkossa yleisessä polkupeitteessä on 2 polkua, kun taas solmueriallisessa polkupeitteessä on 3 polkua. Erona on, että yleisessä polkupeitteessä solmua 6 käytetään kahdessa polussa.

Yleisen polkupeitteen voi löytää lähes samalla tavalla kuin solmueriallisen polkupeitteen. Riittää täydentää maksimiparituksen verkkoa niin, että siinä on kaari $a \rightarrow b$ aina silloin, kun alkuperäisessä verkossa solmusta a pääsee solmuun b (mahdollisesti usean kaaren kautta).

20.5.3 Dilworthin lause

Dilworthin lauseen mukaan suunnatun, syklittömän verkon pienin yleinen polkupeite on yhtä suuri kuin suurin mahdollinen kokoelma solmuja, jossa minäkään kahden solmun välillä ei ole polkua.

Esimerkiksi äskeisessä verkossa pienin yleinen polkupeite sisältää kaksi polkua. Niinpä verkosta voidaan valita enintään kaksi solmua niin, että minäkään solmujen välillä ei ole polkua. Vaihtoehtoja valintaan on monia: voimme valita esimerkiksi solmut 3 ja 7:



Verkossa ei ole polkua solmusta 3 solmuun 7 eikä polkua solmusta 7 solmuun 3, joten valinta on kelvollinen. Toisaalta jos verkosta valitaan mitkä tahansa kolme solmua, jostain solmusta toiseen on polku.

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, \dots, 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 $2 \dots \sqrt{n}$. 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 $2 \dots \sqrt{n}$, 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 $2 \dots \sqrt{n}$. 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 $2 \dots n$ 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 $2 \dots n$ lukuja yksi kerrallaan. Aina kun uusi alkuluku x löytyy, niin algoritmi merkitsee taulukkoon, että x :n moninkerrat $2x, 3x, 4x, \dots$ eivät ole alkulukuja, koska niillä on alkutekijä x .

Esimerkiksi jos väli on 2...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 $\sum_{i=2}^n n/i = n/2 + n/3 + n/4 + \dots + n/n$. 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 $1 \dots n$. 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 $s = \text{syta}(a, b)$ ja k on mikä tahansa kokonaisluku.

21.3.2 Kiinalainen jäännöslause

Kiinalainen jäännöslause ratkaisee yhtälöryhmän muotoa

$$\begin{aligned} x &= a_1 \bmod M_1 \\ x &= a_2 \bmod M_2 \\ &\dots \\ x &= a_n \bmod M_n \end{aligned}$$

missä kaikki parit luvuista M_1, M_2, \dots, 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

$$x = a_1 X_1 X_{1M_1}^{-1} + a_2 X_2 X_{2M_2}^{-1} + \cdots + a_n X_n X_{nM_n}^{-1}.$$

Ideana on, että jokaisessa yhtälön osassa $a_k X_k X_{kM_k}^{-1}$ 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 jakojäännöseen 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 $x + M_1 M_2 \cdots M_n$ 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 yhdistelmien määrän saa selville hyödyntämällä säännöllisyyksiä ja dynaamista ohjelmointia.

22.1 Perustekniikat

Summat ja tulot

Summa kertoo yhdistelmien määrän, kun on olemassa useita vaihtoehtoja ja niistä valitaan yksi. Tulo kertoo puolestaan yhdistelmien määrän, kun tehdään joukko peräkkäisiä valintoja.

Tehtävä: Ravintolassa A on 4 alkuruokaa, 7 pääruokaa ja 3 jälkiruokaa. Ravintolassa B on 6 alkuruokaa, 8 pääruokaa ja 2 jälkiruokaa. Montako menua on olemassa (alkuruoka, pääruoka, jälkiruoka)?

Ravintolassa A menuja on $4 \cdot 7 \cdot 3 = 84$ ja ravintolassa B menuja on $6 \cdot 8 \cdot 2 = 96$. Niinpä menuja on yhteensä $84 + 96 = 180$.

Merkkijonot

On k^n tapaa muodostaa n merkin pituinen merkkijono, kun käytössä on k merkkiä, koska jokaisessa merkkijonon kohdassa merkin valintaan on k vaihtoehtoa. Esimerkiksi on $2^3 = 8$ tapaa muodostaa kolmen merkin pituinen bittijono: 000, 001, 010, 011, 100, 101, 110 ja 111.

Permutaatiot

Kertoma $n!$ ilmaisee, monellako tavalla voidaan järjestää n alkiota eli muodostaa n alkion permutaatio. Tämä perustuu siihen, että ensimmäisen alkion voi valita n tavalla, seuraavan $n - 1$ tavalla, jne. Esimerkiksi merkkijonolla abc on $3! = 6$ permutaatiota: abc, acb, bac, bca, cab ja cba.

Rekursio vs. suljettu muoto

Tarkastellaan seuraavaa tehtävää:

Tehtävä: Monellako tavalla luvun n voi esittää positiivisten kokonaislukujen summana? Esimerkiksi luvun 4 voi esittää 8 tavalla: $1+1+1+1$, $1+1+2$, $1+2+1$, $2+1+1$, $2+2$, $3+1$, $1+3$ ja 4.

Kombinatorisen tehtävän ratkaisun voi usein esittää rekursiivisesti. Tässä tapauksessa voimme merkitä $f(n)$:llä esitystapojen määrää luvulle n . Funktion voi laskea rekursiivisesti näin:

$$f(n) = \begin{cases} 1 & n = 1 \\ f(1) + f(2) + \dots + f(n-1) + 1 & n > 1 \end{cases}$$

Pohjatapauksena $f(1) = 1$. Yleisessä tapauksessa käydään läpi kaikki vaihtoehdot, mikä on summan viimeinen luku. Viimeisenä oleva 1 tulee siitä, että summa voi olla myös pelkkä luku n .

Funktion ensimmäiset arvot ovat:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \\ f(5) &= 16 \end{aligned}$$

Tässä tehtävässä funktiolle on myös *suljettu muoto*

$$f(n) = 2^{n-1},$$

mikä johtuu siitä, että summassa on $n-1$ mahdollista kohtaa $+$ -merkille ja niistä valitaan mikä tahansa osajoukko.

22.2 Binomikerroin

Binomikerroin $\binom{n}{k}$ ilmaisee, monellako tavalla n alkion joukosta voidaan muodostaa k alkion osajoukko. Esimerkiksi $\binom{5}{2} = 10$, koska alkioista $\{A, B, C, D, E\}$ voidaan valita 10 tavalla 2 alkiota:

$$\{A, B\}, \{A, C\}, \{A, D\}, \{A, E\}, \{B, C\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\}, \{D, E\}$$

Laskutapa 1

Binomikertoimen voi laskea rekursiivisesti seuraavasti:

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

Ideana rekursiossa on tarkastella tiettyä joukon alkia x . Jos alkio x valitaan osajoukkoon, täytyy vielä valita $n - 1$ alkia $k - 1$ alkia. Jos taas alkia x ei valita osajoukkoon, täytyy vielä valita $n - 1$ alkia k alkia.

Rekursioiden pohjatapaukset ovat seuraavat:

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

Selityksenä on, että on aina yksi tapa muodostaa tyhjä osajoukko, samoin kuin valita kaikki alkut osajoukkoon.

Laskutapa 2

Toinen tapa laskea binomikerroin on seuraava:

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

Kaavassa $n!$ on n alkion permutaatioiden määrä. Ideana on käydä läpi kaikki permutaatiot ja valita kussakin tapauksessa permutaation k ensimmäistä alkia osajoukkoon. Koska ei ole merkitystä, missä järjestyksessä osajoukon alkut ja ulkopuoliset alkut ovat, tulos jaetaan luvuilla $k!$ ja $(n-k)!$.

Ominaisuuksia

Binomikertoimelle pätee

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

koska k alkion valinta osajoukkoon tarkoittaa samaa kuin että valitaan $n-k$ alkia osajoukon ulkopuolelle.

Binomikerrointen summa on

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Nimi "binomikerroin" tulee siitä, että

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

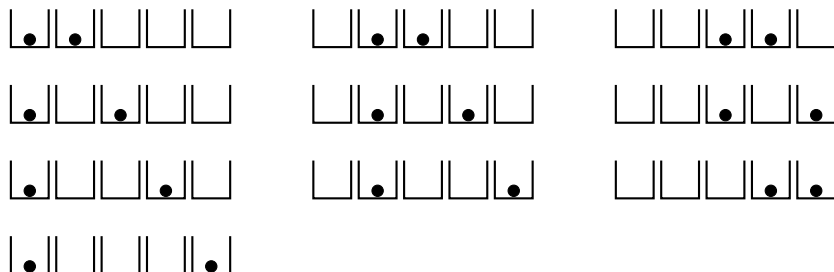
Binomikertoimet esiintyvät myös Pascalin kolmiossa, jonka reunoilla on lukua 1 ja jokainen luku saadaan kahden yllä olevan luvun summana:

$$\begin{array}{ccccccc} & & & & 1 & & & & \\ & & & & 1 & & 1 & & \\ & & & 1 & & 2 & & 1 & \\ & & 1 & & 3 & & 3 & & 1 \\ 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & & \dots & & \dots & & \dots & & \dots \end{array}$$

Laatikot ja pallot

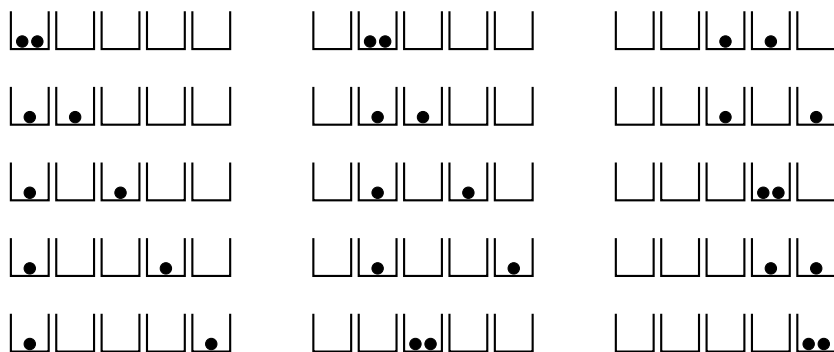
Laatikot ja pallot on usein hyödyllinen malli kombinatoriikan tehtävissä. Siinä n laatikkoon sijoitetaan k palloa. Tarkastellaan seuraavaksi kolmea tapausta:

Tapaus 1: Kuhunkin laatikkoon saa sijoittaa enintään yhden pallon. Esimerkiksi kun $n = 5$ ja $k = 2$, sijoitustapoja on 10:



Tässä tapauksessa vastauksen kertoo suoraan binomikerroin $\binom{n}{k}$.

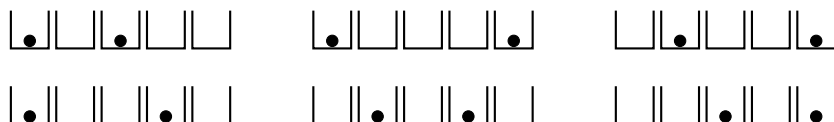
Tapaus 2: Samaan laatikkoon saa sijoittaa monta palloa. Esimerkiksi kun $n = 5$ ja $k = 2$, sijoitustapoja on 15:



Prosessin voi kuvata merkkijonona, joka muodostuu merkeistä "o" ja "→". Pallojen sijoittaminen alkaa vasemmanpuoleisimmasta laatikosta. Merkki "o" tarkoittaa, että pallo sijoitetaan nykyiseen laatikkoon, ja merkki "→" tarkoittaa, että siirrytään seuraavaan laatikkoon.

Nyt jokainen sijoitustapa on merkkijono, jossa on k kertaa merkki "o" ja $n - 1$ kertaa merkki "→". Esimerkiksi sijoitustapaa ylhäällä oikealla vastaa merkkijono "→ → o → o →". Niinpä sijoitustapojen määrä on $\binom{n+k-1}{k}$.

Tapaus 3: Kuhunkin laatikkoon saa sijoittaa enintään yhden pallon ja lisäksi missään kahdessa vierekkäisessä laatikossa ei saa olla palloa. Esimerkiksi kun $n = 5$ ja $k = 2$, sijoitustapoja on 6:



Tässä tapauksessa voi ajatella, että alussa k palloa ovat laatikoissaan ja joka välissä on yksi tyhjä laatikko. Tämän jälkeen jää valittavaksi $n - k - (k - 1) = n - 2k + 1$ tyhjän laatikon paikat. Mahdollisia välejä on $k + 1$, joten tapauksen 2 perusteella sijoitustapoja on $\binom{k+1+n-2k+1-1}{n-2k+1} = \binom{n-k+1}{n-2k+1}$.

Multinomikerroin

Binomikertoimen yleistys on multinomikerroin

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

missä $k_1 + k_2 + \dots + k_m = n$. Multinomikerroin ilmaisee, monellako tavalla n alkiota voidaan jakaa osajoukkoihin, joiden koot ovat k_1, k_2, \dots, k_m . Jos $m = 2$, multinomikertoimen kaava vastaa binomikertoimen kaavaa.

22.3 Catalanin luvut

Catalanin luku C_n ilmaisee, montako tapaa on muodostaa kelvollinen sulkulauseke n alkusulusta ja n loppusulusta. Esimerkiksi $C_3 = 5$, koska 3 alkusulusta ja 3 loppusulusta voidaan muodostaa kelvolliset sulkulausekkeet $()()()$, $((()))$, $()(())$, $((()))$ ja $((()))$.

Sulkulausekkeet

Sulkulauseke on kelvollinen, jos sitä vastaa jokin matemaattinen lauseke, josta on poistettu kaikki merkit sulkuja lukuun ottamatta. Esimerkiksi yllä mainittu sulkulauseke $((()))()$ on kelvollinen, koska sitä vastaa esimerkiksi matemaattinen lauseke $(2 \cdot (4 + 5) + 3) \cdot (1 + 3)$.

Kelvollisen sulkulausekkeen voi tunnistaa käymällä läpi lausekkeen merkit vasemmalta oikealle ja pitämällä yllä laskuria, jonka arvo on aluksi 0. Merkki $($ kasvattaa laskuria ja merkki $)$ vähentää laskuria. Sulkulauseke on kelvollinen, jos laskurin arvo on aina vähintään 0 ja lopuksi tarkalleen 0.

Huomaa myös, että kelvollisen sulkulausekkeen jokaisessa alkuosassa alkusulkujen määrä on sama tai suurempi kuin loppusulkujen määrä.

Laskutapa 1

Catalanin lukuja voi laskea rekursiivisesti seuraavasti:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$$

Summa käy läpi tavat jakaa sulkulauseke kahteen osaan niin, että kumpikin osa on kelvollinen sulkulauseke ja alkuosa on mahdollisimman lyhyt mutta ei tyhjä. Kunkin vaihtoehdon kohdalla alkuosassa on $i + 1$ sulkuparia ja lausekkeiden määrä saadaan kertomalla keskenään:

- C_i : tavat muodostaa sulkulauseke alkuosan sulkupareista ulointa sulkuparia lukuun ottamatta
- C_{n-i-1} : tavat muodostaa sulkulauseke loppuosan sulkupareista

Lisäksi pohjatapauksena on $C_0 = 1$, koska 0 sulkuparista voi muodostaa tyhjän sulkulausekkeen.

Laskutapa 2

Catalanin lukuja voi laskea myös binomikertoimen avulla:

$$C_n = \frac{1}{n+1} \binom{2n}{n}.$$

Kaavan voi perustella seuraavasti:

Kun käytössä on n alkusulkua ja n loppusulkua, niistä voi muodostaa kaikkiaan $\binom{2n}{n}$ sulkulauseketta. Lasketaan seuraavaksi, moniko tällainen sulkulauseke *ei* ole kelvallinen.

Jos sulkulauseke ei ole kelvallinen, siinä on oltava alkuosa, jossa loppusulkuja on alkusulkuja enemmän. Muutetaan jokainen tällaisen alkuosan sulkumerkki käänteiseksi. Esimerkiksi lausekkeessa $()()()()$ alkuosa on $()()$ ja kääntämisen jälkeen lausekkeesta tulee $)((()()$.

Tuloksena olevassa lausekkeessa on $n+1$ alkusulkua ja $n-1$ loppusulkua. Tällaisia lausekkeitä on kaikkiaan $\binom{2n}{n+1}$, joka on sama kuin ei-kelvollisten sulkulausekkeiden määrä. Niinpä kelvollisten sulkulausekkeiden määrä voidaan laskea kaavalla

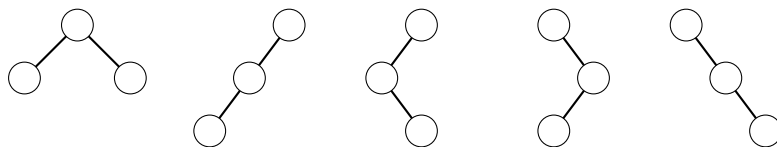
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Puiden laskeminen

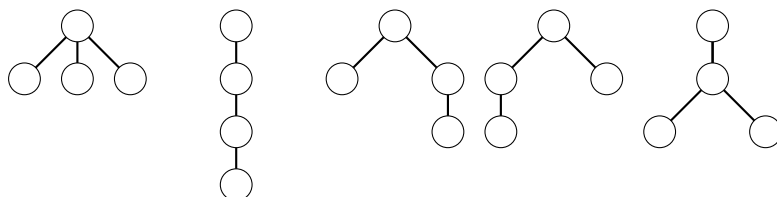
Catalanin luvut kertovat myös juurellisten puiden lukumääriä:

- n solmun binääripuiden määrä on C_n
- n solmun yleisten puiden määrä on C_{n-1}

Esimerkiksi tapauksessa $C_3 = 5$ binääripuut ovat



ja yleiset puut ovat

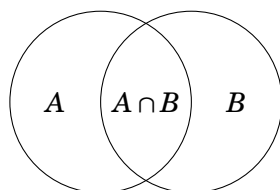


22.4 Inkluusio-ekskluusio

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 $|X|$ tarkoittaa joukon X kokoa. Seuraava kuva havainnollistaa kaavaa, kun joukot ovat tason ympyröitä:

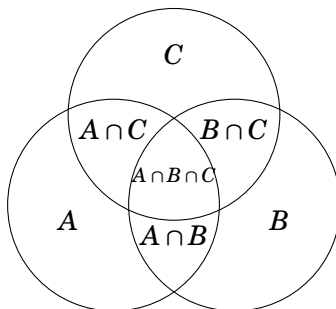


Tavoitteena on laskea, kuinka suuri on yhdiste $A \cup B$ eli alue, joka on toisen tai kummankin ympyrän sisällä. Kuvan mukaisesti yhdisteen $A \cup B$ koko saadaan laskemalla ensin yhteen ympyröiden A ja B koot ja vähentämällä siitä sitten leikkauksen $A \cap B$ koko.

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 $X_1 \cup X_2 \cup \dots \cup X_n$ koon saa laskettua käymällä läpi kaikki tavat muodostaa leikkaus joukoista X_1, X_2, \dots, 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

Tarkastellaan esimerkkinä seuraavaa tehtävää:

Tehtävä: Lukujonon $(1, 2, \dots, n)$ epäjärjestys on permutaatio, jossa mikään luku ei ole alkuperäisellä paikallaan. Tehtäväsi on laskea, montako epäjärjestystä on olemassa. Esimerkiksi jos $n = 3$, niin epäjärjestyksiä on kaksi: $(2, 3, 1)$ ja $(3, 1, 2)$

Yksi tapa lähestyä tehtävää on käyttää inklusio-eksklusiota. 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 epäjärjestysten määrä on

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

eli riittää laskea joukkojen yhdisteen koko. Tämä palautuu inklusio-eksklusion 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 $3! - 4 = 2$.

Osoittautuu, että tehtävän voi ratkaista myös toisella tavalla käyttämättä inklusio-eksklusiota. Merkitään $f(n)$:llä jonon $(1, 2, \dots, n)$ epäjärjestysten määrää, jolloin seuraava rekursio pätee:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Kaavan voi perustella käymällä läpi tapaukset, miten luku 1 muuttuu epäjärjestyksessä. On $n-1$ tapaa valita jokin luku x luvun 1 tilalle. Jokaisessa tällaisessa valinnassa on kaksi vaihtoehtoa:

Vaihtoehto 1: Luvun x tilalle valitaan luku 1. Tällöin jää $n-2$ lukua, joille tulee muodostaa epäjärjestys.

Vaihtoehto 2: Luvun x tilalle ei valita lukua 1. Tällöin jää $n-1$ lukua, joille tulee muodostaa epäjärjestys, koska luvun x tilalle ei saa valita lukua 1 ja kaikki muut luvut tulee saattaa epäjärjestykseen.

22.5 Burnsiden lemma

Burnsiden lemma laskee yhdistelmien määrän niin, että symmetrisistä yhdistelmistä lasketaan mukaan vain yksi edustaja. Burnsiden lemmän mukaan yhdistelmien määrä on

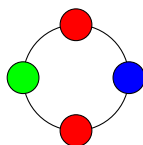
$$\sum_{k=1}^n \frac{c(k)}{n},$$

missä yhdistelmän asentoa voi muuttaa n tavalla ja $c(k)$ on niiden yhdistelmien määrä, jotka pysyvät ennallaan, kun asentoa muutetaan tavalla k .

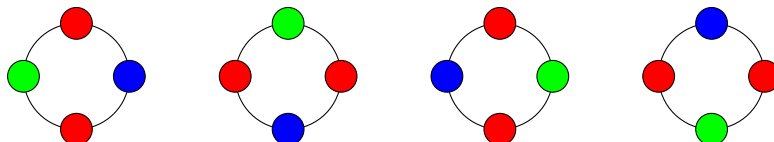
Tutustumme Burnsiden lemmaan seuraavan tehtävän kautta:

Tehtävä: Helminauhassa on n helmeä, joista jokaisen väri on väliltä $1, 2, \dots, m$. 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 $0, 1, \dots, n-1$ 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

$$m^{\text{sy}(k,n)},$$

missä $\text{sy}(k,n)$ on lukujen k ja n suurin yhteinen tekijä. Tämä johtuu siitä, että $\text{sy}(k,n)$ -kokoiset pätkät helmiä siirtyvät toistensa paikoille k askelta eteenpäin. Niinpä helminauhojen määrä on Burnsiden lemmän mukaan

$$\sum_{i=0}^{n-1} \frac{m^{\text{sy}(i,n)}}{n}.$$

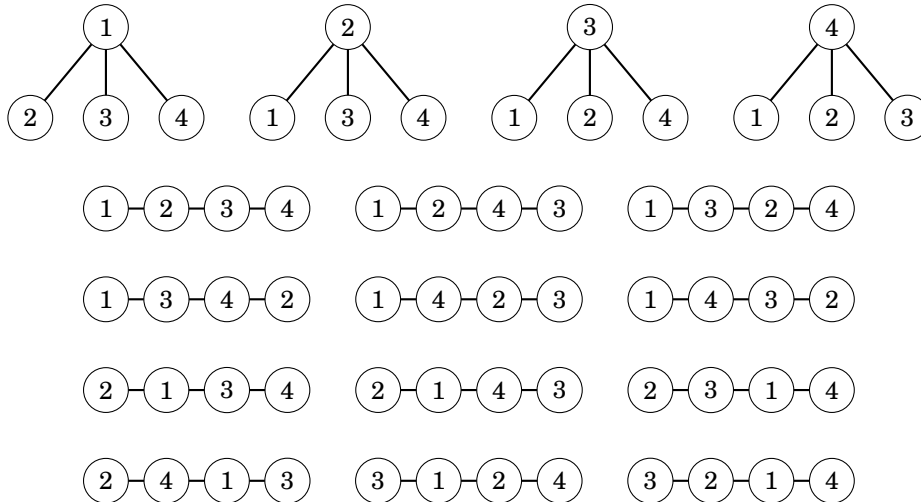
Esimerkiksi kun helminauhan pituus on 4 ja värejä on 3, helminauhoja on

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.6 Cayleyn kaava

Cayleyn kaavan mukaan n solmusta voi muodostaa n^{n-2} numeroitua puuta. Puun solmut on numeroitu $1, 2, \dots, n$, ja kaksi puuta ovat erilaiset, jos niiden rakenne on erilainen tai niissä on eri numerointi.

Esimerkiksi kun $n = 4$, numeroitujen puiden määrä on $4^{4-2} = 16$:

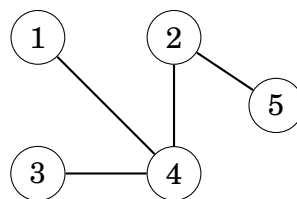


Seuraavaksi näemme, miten Cayleyn kaavan voi perustella samastamalla numeroidut puut Prüfer-kodeihin.

Prüfer-koodi

Prüfer-koodi on $n - 2$ luvun jono, joka kuvaa numeroidun puun rakenteen. Koodi muodostuu poistamalla puusta joka askeleella lehden, jonka numero on pienin, ja lisäämällä lehden vieressä olevan solmun numeron kodiin.

Tarkastellaan esimerkiksi seuraavaa puuta:



Tämän puun Prüfer-koodi on $(4, 4, 2)$, koska puusta poistetaan ensin solmu 1, sitten solmu 3 ja lopuksi solmu 5.

Jokaiselle puulle voidaan laskea Prüfer-koodi, minkä lisäksi Prüfer-koodista pystyy palauttamaan yksikäsitteisesti alkuperäisen puun. Niinpä numeroituja puita on yhtä monta kuin Prüfer-kodeja eli n^{n-2} .

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:

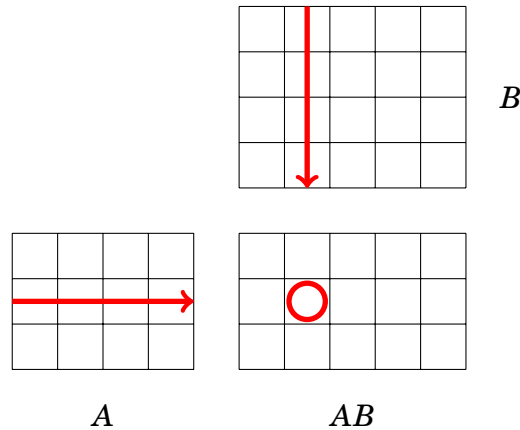
$$A = \begin{pmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{pmatrix} = \begin{pmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{pmatrix}$$

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:

$$A = \begin{pmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 6 \\ 2 & 9 \end{pmatrix}$$

$$AB = \begin{pmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{pmatrix} = \begin{pmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{pmatrix}$$

$$A = \begin{pmatrix} 9 & 6 & 5 \\ 4 & 1 & 8 \end{pmatrix} \quad B = \begin{pmatrix} 3 \\ 5 \\ 8 \end{pmatrix}$$

$$AB = \begin{pmatrix} 9 \cdot 3 + 6 \cdot 5 + 5 \cdot 8 \\ 4 \cdot 3 + 1 \cdot 5 + 8 \cdot 8 \end{pmatrix} = \begin{pmatrix} 97 \\ 81 \end{pmatrix}$$

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 $O(n^3k)$, 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 -matriiseissa 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

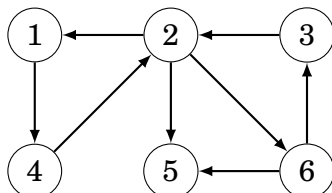
$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

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 $1 \rightarrow 4 \rightarrow 2 \rightarrow 5$. Vastaavasti $(V^3)_{2,2} = 2$, koska solmusta 2 pääsee itseensä kulkemalla polkuja $2 \rightarrow 1 \rightarrow 4 \rightarrow 2$ sekä $2 \rightarrow 6 \rightarrow 3 \rightarrow 2$.

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 $0 \dots 1$, 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 $1, 2, \dots, 6$. Yleensä oletetaan, että kunkin silmäluvun todennäköisyys on $1/6$ eli kaikki tulokset ovat yhtä todennäköisiä.

Tapahtuman todennäköisyyttä merkitään $P(\dots)$, jossa kolmen pisteen tilalla on tapahtuman kuvaus. Esimerkiksi nopan heitossa $P(\text{"silmäluku on 4"}) = 1/6$, $P(\text{"silmäluku ei ole 6"}) = 5/6$ ja $P(\text{"silmäluku on parillinen"}) = 1/2$.

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 $1, 2, \dots, 13$. 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 $\{1, 2, \dots, 6\}$, 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 $A = \text{”silmluku on parillinen”}$ ja $B = \text{”silmluku on alle 5”}$, 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 $A = \text{”silmluku on parillinen”}$ ja $B = \text{”silmluku ei ole 6”}$, 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 $X = \text{”silmlukujen summa”}$. 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 silmlukujen 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 $a, a+1, \dots, b$ ja jokaisen arvon todennäköisyys on sama. Esimerkiksi yhden nopan heitto tuottaa tasajakauman, jossa $P(X = x) = 1/6$, kun $x = 1, 2, \dots, 6$.

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 $E[X_1 + X_2 + \dots + X_n]$ voidaan laskea $E[X_1] + E[X_2] + \dots + E[X_n]$. 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?

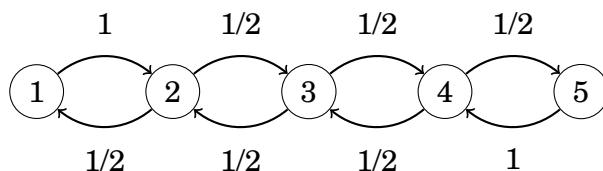
Todennäköisyys, että yksittäinen hattu on tyhjä, on $(\frac{n-1}{n})^n$, koska mikään pallo ei saa mennä sinne. Odotusarvon lineaarisuuden ansiosta tyhjien hattujen määrän odotusarvo on $n \cdot (\frac{n-1}{n})^n$.

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 $[p_1, p_2, \dots, p_n]$, missä p_k tarkoittaa todennäköisyyttä olla tällä hetkellä tilassa k . Todennäköisyyksille pätee aina $p_1 + p_2 + \dots + p_n = 1$.

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 $O(n^2m)$, 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 $0, 1, \dots, 15$. 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 $[x_1, x_2, \dots, x_n]$, 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 $[0, 0, \dots, 0]$ 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 $x_1 \oplus x_2 \oplus \dots \oplus x_n$, 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 $[0, 0, \dots, 0]$ on häviötila, ja sen xor-summa on 0, kuten kuuluukin. Muissa häviötiloissa mikä tahansa siirto johtaa voittotilaan, koska yksi luvusta 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 $10 \oplus 3 = 9$ tikkuja, 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. Käännä 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 minäkään 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 $0, 1, \dots, x - 1$.

Esimerkki

Sokkelopeli: Sokkelossa on pelihahmo, jota pelaajat siirtävät vuorotellen. Jokainen sokkelon ruutu on lattiaa tai seinää. Kullakin siirrolla hahmon tulee liikua 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:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

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.

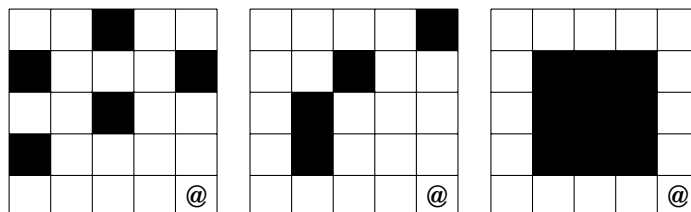
25.3.2 Alipelit

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:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

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

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

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, \dots, s_n , missä n on merkkijonon pituus. Aakkosto (*alphabet*) määrittää, mitä merkkejä merkkijonossa voi esiintyä. Esimerkiksi aakkosto $\{A, B, \dots, Z\}$ 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.

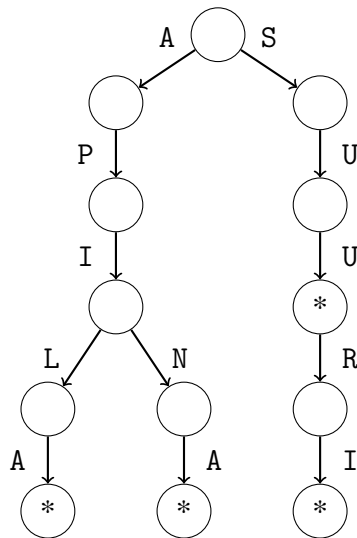
Kierto (*rotation*) syntyy, kun merkkejä siirretään yksi kerrallaan alusta loppuun. Esimerkiksi merkkijonon ABC kierrot ovat ABC, BCA ja CAB. Jakso (*period*) on alkuosa, jota toistamalla merkkijono muodostuu. Esimerkiksi merkkijonon ABCABCA lyhin jakso on ABC. Reuna (*border*) on merkkijono, joka on 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

$$(c_1A^{n-1} + c_2A^{n-2} + \dots + c_nA^0) \bmod B,$$

missä merkkijonon merkkien koodit ovat c_1, c_2, \dots, 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 osajonon 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 $0 \dots B-1$. Suuruusluokan 10^9 etuna on, että long long -tyyppi riittää hajautusarvojen käsittelyyn koodissa, koska tulot AB ja BB mahduttavat 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, \dots, y_n . Yhden tai useamman törmäyksen todennäköisyys on $1 - (1 - 1/B)^n$.

Tapaus 3: Merkkijonoja x_1, x_2, \dots, 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 testisyytteen, 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-tilukon, jonka jokaisessa kohdassa lukee, kuinka pitkälle kohdasta alkava osajono vastaa merkkijonon alkuosaa. Esimerkiksi Z-tilukko 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-aulukon 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-aulukon alussa olevaa tietoa.

Esimerkki

Katsotaan nyt, miten Z-algoritmi muodostaa seuraavan Z-aulukon:

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:

1	2	3	4	5	6	$x \qquad \qquad \qquad y$					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-aulukon arvot nopeammin. Koska välin $[7, 11]$ merkit ovat samat kuin merkkijonon alussa, myös Z-aulukon arvoissa on vastaavuutta.


Ensinnäkin kohdissa 8 ja 9 tulee olla samat arvot kuin kohdissa 2 ja 3, koska väli $[7, 11]$ vastaa väliä $[1, 5]$:

1	2	3	4	5	6	$x \qquad \qquad \qquad y$					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.

							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	?	?	?	?	?	?	?	



Nyt algoritmi alkaa vertailla merkkejä kohdasta $y + 1$ alkaen merkki kerrallaan. Algoritmi ei voi hyödyntää aiempaa tietoa Z-aulukossa, koska se ei ole vielä aiemmin tutkinut merkkijonoa näin pitkälle. Tuloksena osajonon pituudeksi tulee 7 ja väli $[x, y]$ päivittyy vastaavasti:

									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	7	?	?	?	?	?	?	

Tämän jälkeen kaikkien seuraavien Z-aulukon arvojen laskemisessa pystyy hyödyntämään jälleen välin $[x, y]$ antamaa tietoa ja algoritmi saa Z-aulukon loppuun tulevat arvot suoraan Z-aulukon alusta:

									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	7	0	0	2	0	0	1	

Z-aulukon käyttö

Z-aulukossa 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-aulukko 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-aulukon 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 vakio kertoimet ovat pieniä.

27.1 Summakysely

Aloitamme tutusta ongelmasta, jossa toteutettavana on kaksi operaatiota luku-
taulukkoon: alkion muuttaminen ja välin summan laskeminen. Olemme aiem-
min ratkaisseet tämän tehtävän segmenttipuulla, mutta vaihtoehtoinen lähes-
tymistapa 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 syöttestä. 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 $n/k > \sqrt{n}$ alkioita.

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 eria 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 $O(k^2n)$. 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 $1, 2, \dots, m$. 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

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

ja seuraava kysely koskee väliä

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

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 9.3):

```
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:

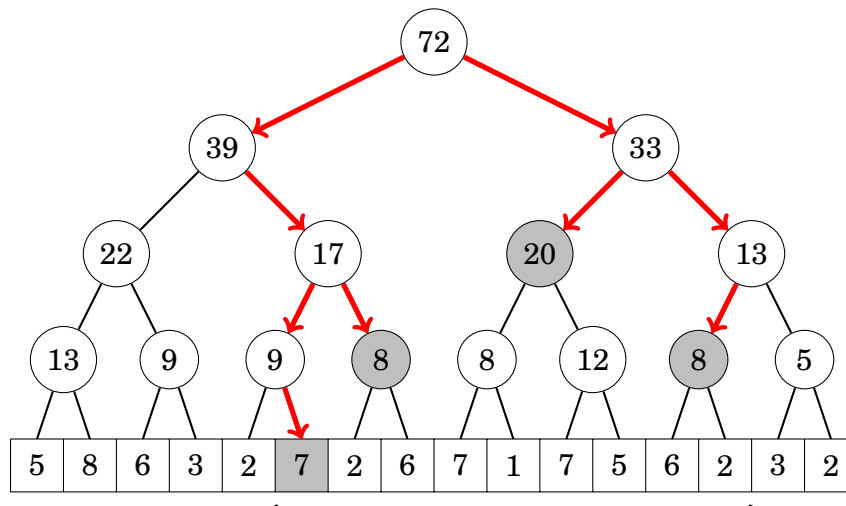
```
int s = summa(a, b, 1, 0, N-1);
```

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.

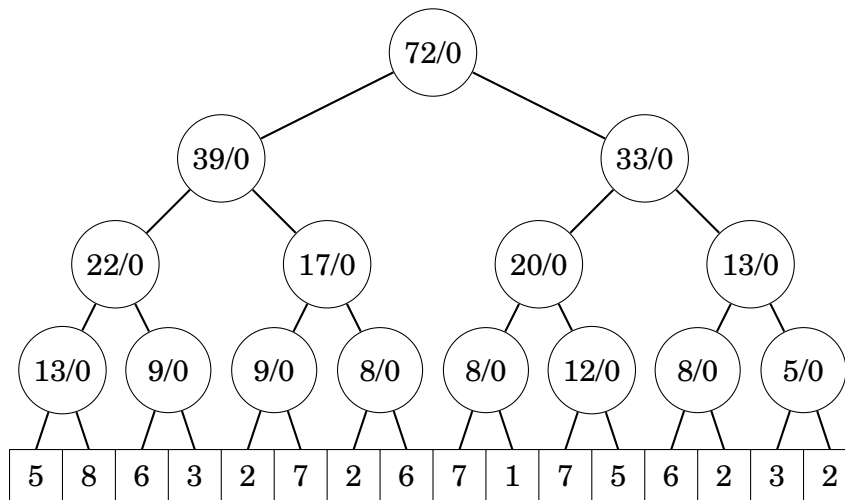
Välin muutostapa voi olla joko *lisäys* tai *asetus*. Lisäyksessä välin jokaiseen alkioon lisätään tietty arvo, ja asetuksessa välin jokainen alkio saa tietyn arvon.

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.

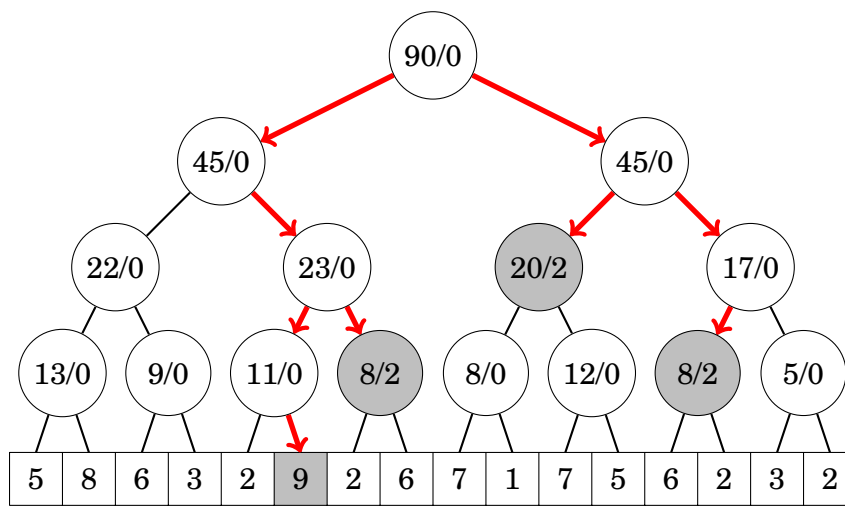
Jokaisessa puun solmussa on kaksi arvoa (s/z): välin lukujen summa s , kuten tavallisessa segmenttipuussa, sekä laiska muutos z , joka tarkoittaa, että kaikkiin välin lukuihin tulee lisätä z . Seuraavassa puussa jokaisessa solmussa $z = 0$ eli mitään muutoksia ei ole kesken.



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

$$p(x) = t_k x^k + t_{k-1} x^{k-1} + \dots + t_0.$$

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, \dots, z_k ovat polynomin kertoimet, joka ilmaisee väliin kohdistuvan laiskan muutoksen.

Nyt välin $[c, d]$ summa on

$$s + \sum_{x=0}^{d-c} z_k x^k + z_{k-1} x^{k-1} + \dots + z_0.$$

Summan saa laskettua tehokkaasti osissa summakaavalla. Esimerkiksi termin z_0 summaksi tulee $(d - c + 1)z_0$ ja termin $z_1 x$ summaksi tulee

$$z_1(0 + 1 + \dots + d - c) = z_1 \frac{(d - c)(d - c + 1)}{2}.$$

Kun muutos etenee alaspäin puussa, polynomin $p(x)$ indeksointi muuttuu, koska jokaisella välillä $[c, d]$ polynomin arvot tulevat kohdista $x = 0, 1, \dots, d - c$. 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.

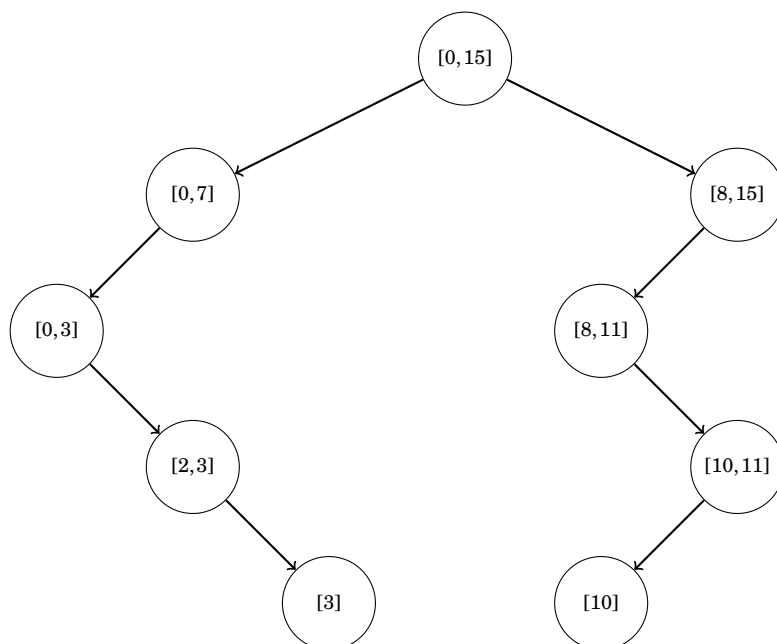
Tämän jälkeen solmuja voi käsitellä seuraavasti:

```
// 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.

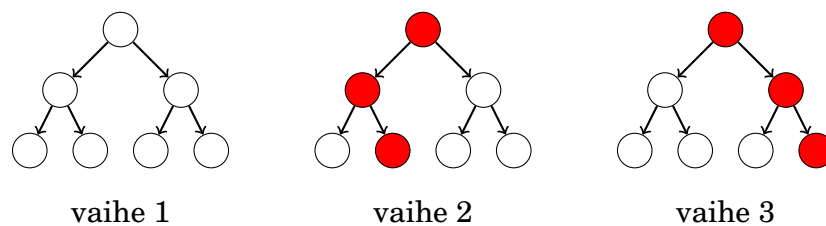
Huomaa, että jos kaikki tarvittavat indeksit ovat tiedossa algoritmin alussa, dynaamisen segmenttipuun sijasta voi käyttää tavallista segmenttipuuta ja indeksien pakkausta (luku 9.4). Tämä ei ole kuitenkaan mahdollista, jos indeksit syntyvät vasta algoritmin aikana.

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.

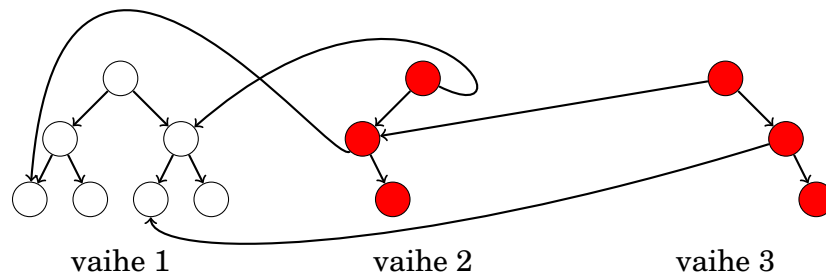
Muutoshistorian hyötynä on, että kaikkia vanhoja puita voi käsitellä segmenttipuun tapaan, koska niiden rakenne on edelleen olemassa. Vanhoista puista voi myös johtaa uusia puita, joita voi muokata edelleen.

Tarkastellaan esimerkiksi seuraavaa muutossarjaa, jossa punaiset solmut muuttuvat päivityksessä ja muut solmut säilyvät ennallaan:



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:



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.

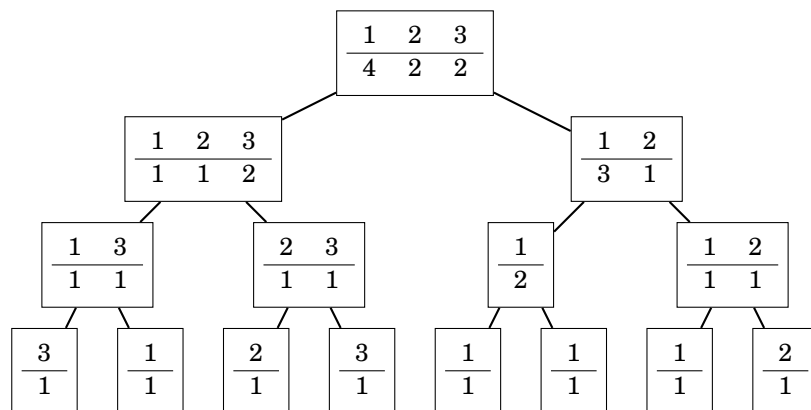
Sopiva tietorakenne tehtävään on map-rakenne, jonka avulla voi pitää kirjaa välillä esiintyvien lukujen määrästä. Yhden solmun käsittely vie aikaa $O(\log n)$, joten kunkin kyselyn aikavaativuus on $O(\log^2 n)$.

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

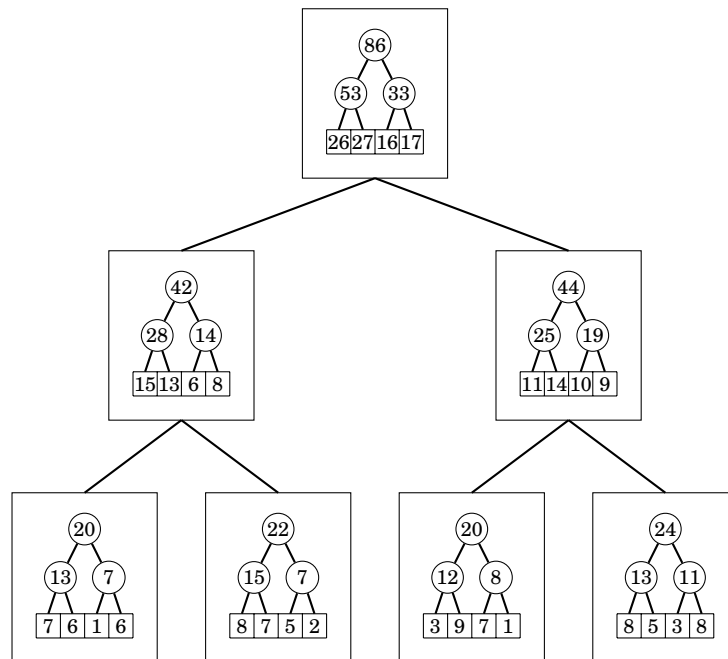
Tehtävä: Toteuta segmenttipuu, jolla pystyy laskemaan $n \times n$ -kokoisen kaksiulotteisen taulukon alitaulukon summan sekä muuttamaan taulukkoa.

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

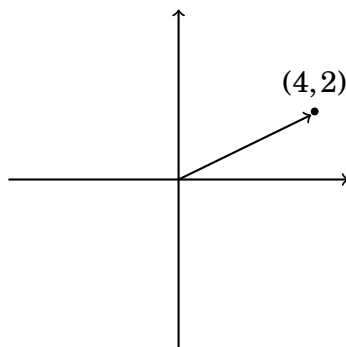
Geometria

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 geometrinen 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 $\sqrt{x^2 + y^2}$. Sil-
lä voi laskea myös pisteiden (x_1, y_1) ja (x_2, y_2) etäisyyden, koska pisteiden etäi-
syys 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 vastapäi-
vään 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 $1/2$ radiaa-
nia vastapäivään ja laskee uuden kulman:

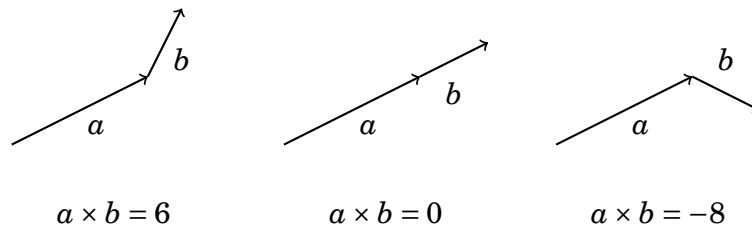
```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0,0.5);  
cout << arg(v) << "\n"; // 0.963648
```

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:



Luokkaa complex käyttäen vektorien a ja b ristitulon voi laskea näin:

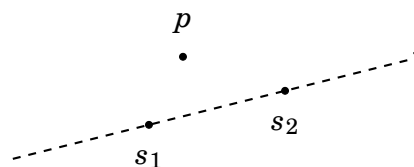
```
P a = {4,2};  
P b = {1,2};  
C r = (conj(a)*b).Y; // 6
```

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 .

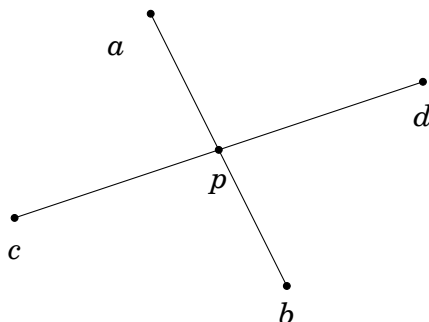
Esimerkiksi seuraavassa kuvassa piste p on suoran vasemmalla puolella:



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.

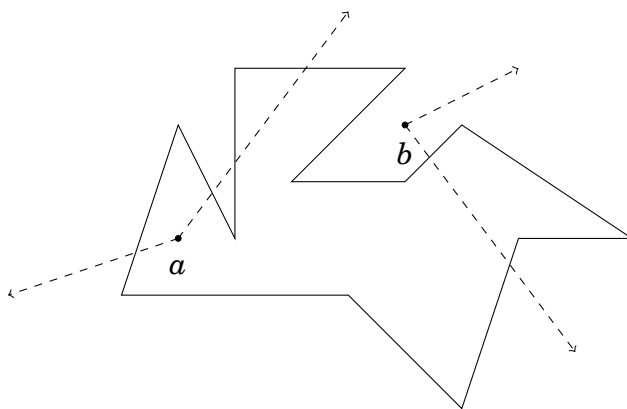
Janojen leikkauspiste p selviää etsimällä parametrit t ja u niin, että

$$p = a + t(b - a) = c + u(d - c).$$

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 $s = (a + b + c)/2$.

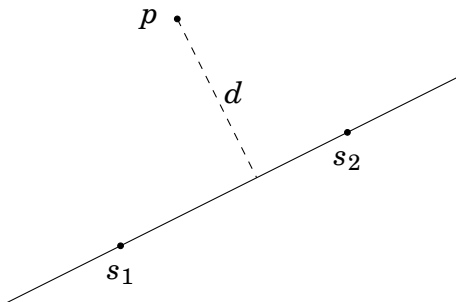
Pinta-alan voi laskea myös ristitulon avulla kaavalla

$$\frac{1}{2}((p_2 - p_1) \times (p_3 - p_1)),$$

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

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

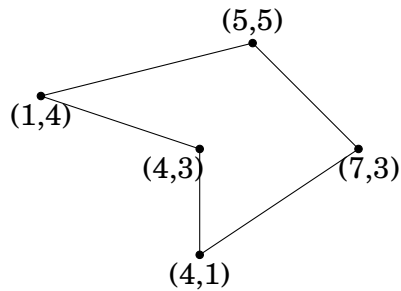
29.3.2 Monikulmion pinta-ala

Yleinen kaava monikulmion pinta-alan laskemiseen on

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right|,$$

missä monikulmion kärkipisteet ovat järjestyksessä p_1, p_2, \dots, p_n ja ensimmäinen ja viimeinen kärkipiste ovat samat eli $p_1 = p_n$.

Esimerkiksi monikulmion



pinta-ala on

$$\frac{|(4 \cdot 3 - 7 \cdot 1) + (7 \cdot 5 - 5 \cdot 3) + (5 \cdot 4 - 1 \cdot 5) + (1 \cdot 3 - 4 \cdot 4) + (4 \cdot 1 - 4 \cdot 3)|}{2} = 19/2.$$

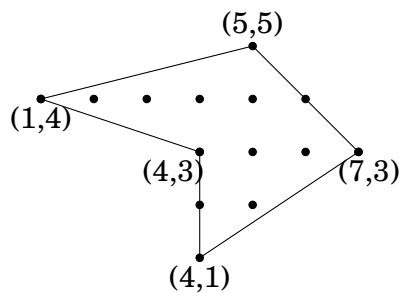
29.3.3 Pickin lause

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

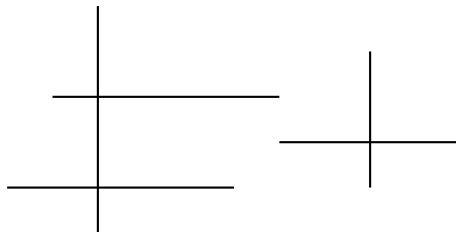
Pyyhkäisyviiva

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?

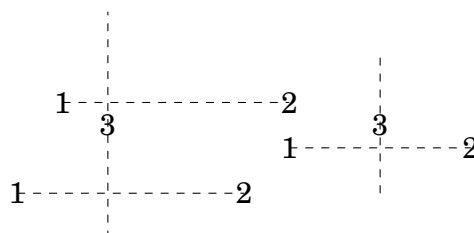
Esimerkiksi seuraavassa tilanteessa leikkauspisteitä on kolme:



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.

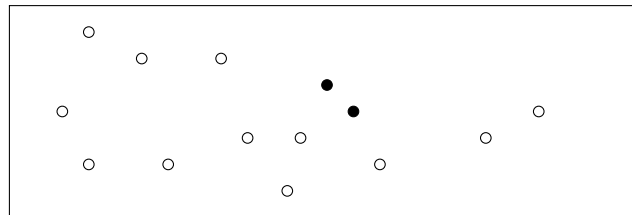
Algoritmi laskee janojen leikkauspisteet tapahtumien 3 kohdalla. Kun pystyjana kulkee y-koordinaattien $y_1 \dots y_2$ välillä, algoritmi laskee tietorakenteesta, monessako vaakajanassa on y-koordinaatti välillä $y_1 \dots y_2$ ja kasvattaa leikkauspisteiden määrää tällä arvolla.

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 kokonaisaikaavaativuus 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.

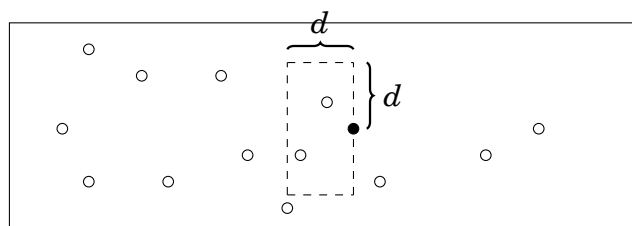
Esimerkiksi seuraavassa kuvassa ratkaisu on tummennettu pistepari:



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.



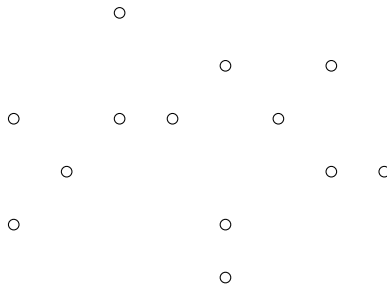
Algoritmin tehokkuus perustuu siihen, että d :n rajoittamalla alueella on aina vain $O(1)$ pistettä. Nämä pisteet pystyy käymään läpi $O(\log n)$ -aikaisesti pitämällä algoritmin aikana yllä joukkoa pisteistä, joiden x-koordinaatti on välillä $[x - d, x]$ ja jotka on järjestetty y-koordinaatin mukaan.

Algoritmin aikavaativuus on $O(n \log n)$, koska se käy läpi n pistettä ja etsii jokaiselle lähimmän edeltävän pisteen ajassa $O(\log n)$.

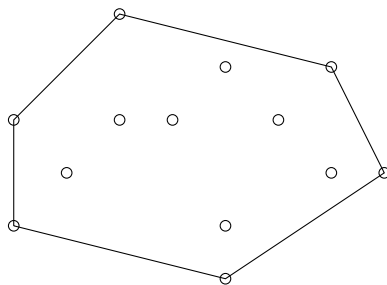
30.3 Konvekksi peite

Konvekssi peite (*convex hull*) on pienin konvekksi 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



konvekssi peite on seuraava:

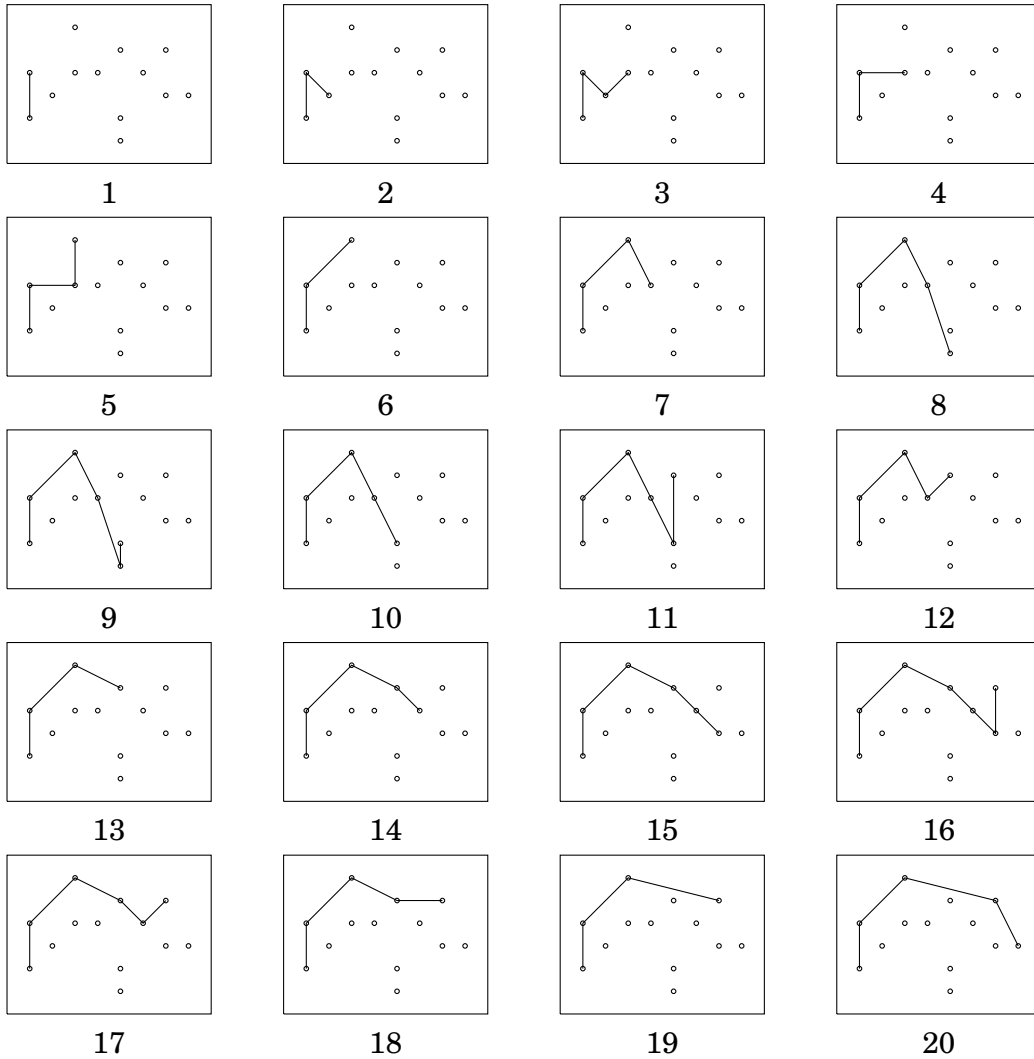


Tehokas ja helposti toteutettava tapa muodostaa konvekssi peite on Andrew'n algoritmi, jonka aikavaativuus on $O(n \log n)$.

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 $O(n)$.