

Kisakoodarin käsikirja

Antti Laaksonen

13. joulukuuta 2016

Sisältö

Alkusanat	ix
I Perusasiat	1
1 Johdanto	3
1.1 Ohjelmointikielet	3
1.2 Syöte ja tuloste	4
1.3 Lukujen käsittely	6
1.4 Koodin lyhentäminen	8
1.5 Matematiikka	9
2 Aikavaativuus	15
2.1 Laskusäännöt	15
2.2 Vaativuusluokkia	18
2.3 Tehokkuuden arviointi	19
2.4 Suurin alitaulukon summa	19
3 Järjestäminen	23
3.1 Järjestämisen teoriaa	23
3.2 Järjestäminen C++:ssa	28
3.3 Binäärihaku	30
4 Tietorakenteet	33
4.1 Dynaaminen taulukko	33
4.2 Joukkorakenne	35
4.3 Hakemisto	36
4.4 Iteraattorit ja välit	37
4.5 Muita tietorakenteita	39
4.6 Vertailu järjestämiseen	42
5 Täydellinen haku	45
5.1 Osajoukkojen läpikäynti	45
5.2 Permutaatioiden läpikäynti	47
5.3 Peruuttava haku	48
5.4 Haun optimointi	49
5.5 Puolivälihaku	52

6	Ahneet algoritmit	55
6.1	Kolikkotehtävä	55
6.2	Aikataulutus	56
6.3	Tehtävät ja deadlinet	58
6.4	Keskiluvut	59
6.5	Tiedonpakkaus	60
7	Dynaaminen ohjelmointi	63
7.1	Kolikkotehtävä	63
7.2	Pisin nouseva alijono	68
7.3	Reitinhaku ruudukossa	69
7.4	Repunpakkaus	70
7.5	Editointietäisyys	71
7.6	Laatoitukset	73
8	Tasoitettu analyysi	75
8.1	Kahden osoittimen tekniikka	75
8.2	Lähin pienempi edeltäjä	78
8.3	Liukuvan ikkunan minimi	79
9	Välikyselyt	81
9.1	Staattisen taulukon kyselyt	81
9.2	Binääri-indeksipuu	84
9.3	Segmenttipuu	86
9.4	Lisätekniikoita	91
10	Bittien käsittely	93
10.1	Luvun bittiesitys	93
10.2	Bittioperaatiot	94
10.3	Joukon bittiesitys	96
10.4	Dynaaminen ohjelmointi	98
II	Verkkoalgoritmit	101
11	Verkkojen perusteet	103
11.1	Käsitteitä	103
11.2	Verkko muistissa	106
12	Verkon läpikäynti	111
12.1	Syvyyshaku	111
12.2	Leveyshaku	113
12.3	Sovelluksia	115
13	Lyhimmät polut	117
13.1	Bellman–Fordin algoritmi	117
13.2	Dijkstran algoritmi	120
13.3	Floyd–Warshallin algoritmi	123

14 Puiden käsittely	127
14.1 Puun läpikäynti	128
14.2 Lämpimä	129
14.3 Solmujen etäisyydet	130
14.4 Binaäripuut	131
15 Virittävät puut	133
15.1 Kruskalin algoritmi	134
15.2 Union-find-rakenne	137
15.3 Primin algoritmi	139
16 Suunnatut verkot	141
16.1 Topologinen järjestys	141
16.2 Dynaaminen ohjelmointi	143
16.3 Tehokas eteneminen	146
16.4 Syklin tunnistaminen	147
17 Vahvasti yhtenäisyys	149
17.1 Kosarajun algoritmi	150
17.2 2SAT-ongelma	152
18 Puukyselyt	155
18.1 Tehokas nouseminen	155
18.2 Solmutaulukko	156
18.3 Alin yhteinen esivanhempi	159
19 Polut ja kierrokset	163
19.1 Eulerin polku	163
19.2 Hamiltonin polku	167
19.3 De Bruijnin jono	169
19.4 Ratsun kierros	170
20 Virtauslaskenta	171
20.1 Ford–Fulkersonin algoritmi	172
20.2 Rinnakkaiset polut	176
20.3 Maksimiparitus	177
20.4 Polkupeitteet	180
III Lisäaiheita	183
21 Lukuteoria	185
21.1 Alkuluvut ja tekijät	185
21.2 Modulolaskenta	189
21.3 Yhtälönratkaisu	192
21.4 Muita tuloksia	193

22 Kombinatoriikka	195
22.1 Binomikerroin	196
22.2 Catalanin luvut	198
22.3 Inkluisio-ekskluisio	200
22.4 Burnsiden lemma	202
22.5 Cayleyn kaava	203
23 Matriisit	205
23.1 Laskutoimitukset	205
23.2 Lineaariset rekursioyhtälöt	208
23.3 Verkot ja matriisit	210
24 Todennäköisyys	213
24.1 Laskutavat	213
24.2 Tapahtumat	214
24.3 Satunnaismuuttuja	216
24.4 Markovin ketju	218
24.5 Satunnaisalgoritmit	219
25 Peliteoria	223
25.1 Pelin tilat	223
25.2 Nim-peli	225
25.3 Sprague–Grundyn lause	226
26 Merkkijonoalgoritmit	231
26.1 Trie-rakenne	231
26.2 Merkkijonohajautus	232
26.3 Z-algoritmi	235
27 Neliöjuorialgoritmit	239
27.1 Eräkäsittely	240
27.2 Tapauskäsittely	241
27.3 Mo’n algoritmi	241
28 Lisää segmenttipuusta	243
28.1 Laiska eteneminen	244
28.2 Dynaaminen toteutus	247
28.3 Tietorakenteet	249
28.4 Kaksiulotteisuus	250
29 Geometria	253
29.1 Kompleksiluvut	254
29.2 Pisteet ja suorat	256
29.3 Monikulmion pinta-ala	259
29.4 Etäisyysmitat	260

30 Pyyhkäisyviiva	263
30.1 Janojen leikkauspisteet	264
30.2 Lähin pistepari	265
30.3 Konvekssi peite	266
Kirjallisuutta	269

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 algoritmikkaa ja mahdollisesti osallistua kansainvälisiin tietotekniikan olympialaisiin (IOI). Kirja sopii myös mainiosti 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 algoritmikan perusteista kirjaa lukemalla ja tehtäviä ratkomalla.

Kisakoodarin käsikirja on 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. Tyyppillisesti 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 ylläpitää kilpailun jälkeen.

1.1 Ohjelmointikielet

Tällä hetkellä yleisimmät koodauskisoissa käytetyt ohjelmointikielet 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 Javaa¹. Jotkut osallistujat 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 etua 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.

C++-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 ottaa erikseen mukaan kirjastoja `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`).

1.2 Syöte ja tuloste

Useimmissa kisoissa käytetään standardivirtoja syötteen lukemiseen ja tulosteen kirjoittamiseen. C++:ssa standardivirrat 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 puolestaan 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 syötteestä:

```
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 syötteestä 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".

1.3 Lukujen käsittely

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;
cout << b << "\n"; // -1757895751
```

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 tyyppin `long long` käyttäminen riittää. On kuitenkin hyvä tietää, että g++-kääntäjä tarjoaa myös 128-bittisen tyyppin `__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.

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 tyyppiä `int` ja `long long`.

Luvun x jakojäännöstä m :llä merkitään $x \bmod m$. Esimerkiksi $17 \bmod 5 = 2$, koska $17 = 3 \cdot 5 + 2$.

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 monissa 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.

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`-funktioilla, 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); // 0.99999999999999988898
```

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

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

Huomaa, että 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} .

1.4 Koodin lyhentäminen

Kisakoodauksessa ihanteena on lyhyt koodi, koska algoritmi täytyy pystyä toteuttamaan mahdollisimman nopeasti. Monet kisakoodarit käyttävätkin lyhennysmerkintöjä tyypeille ja muille koodin osille.

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 koodin

```
long long a = 123456789;  
long long b = 987654321;  
cout << a*b << "\n";
```

voi lyhentää seuraavasti:

```
ll a = 123456789;  
ll b = 987654321;  
cout << a*b << "\n";
```

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;
```

Makrot

Toinen tapa lyhentää koodia on määritellä **makroja**. Makro ilmaisee, että tietyt koodissa olevat merkkijonot korvataan toisilla ennen koodin kääntämistä. C++:ssa makro määritellään esikäntäjän komennolla `#define`.

Määritellään esimerkiksi seuraavat makrot:

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

Tämän jälkeen koodin

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

voi kirjoittaa lyhyemmin seuraavasti:

```
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 koodin

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

voi lyhentää seuraavasti:

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

1.5 Matematiikka

Matematiikka on tärkeässä asemassa kisakoodauksessa, ja menestyvän kisakoodarin täytyy osata myös hyvin matematiikkaa. Käymme seuraavaksi läpi joukon keskeisiä matematiikan käsitteitä ja kaavoja. Palaamme moniin aiheisiin myöhemmin tarkemmin kirjan aikana.

Summakaavat

Jokaiselle summalle muotoa

$$\sum_{x=1}^n x^k = 1^k + 2^k + 3^k + \dots + n^k$$

on olemassa laskukaava, kun k on jokin positiivinen kokonaisluku. Tällainen laskukaava on aina astetta $k + 1$ oleva polynomi. Esimerkiksi

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

ja

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

Aritmeettinen summa on summa, jossa jokaisen vierekkäisen luvun erotus on vakio. Esimerkiksi

$$3 + 7 + 11 + 15$$

on aritmeettinen summa, jossa vakio on 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 $(a+b)/2$.

Geometrinen summa on summa, jossa jokaisen vierekkäisen luvun suhde on vakio. Esimerkiksi

$$3 + 6 + 12 + 24$$

on geometrinen summa, jossa vakio on 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

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

Geometrisen summan sukulainen on

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

Harmoninen summa on summa muotoa

$$\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}.$$

Yläraja harmonisen summan suuruudelle on $\log_2(n) + 1$, koska summaa voi 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 $\log_2(n) + 1$ osaan ($1, 2 \cdot 1/2, 4 \cdot 1/4$, jne.), joista jokaisen summa on enintään 1 .

Joukko-oppi

Joukko on kokoelma alkioita. Esimerkiksi joukko

$$X = \{2, 4, 7\}$$

sisältää alkiot $2, 4$ ja 7 . Merkintä \emptyset tarkoittaa tyhjää joukkoa. Joukon S koko eli alkoiden määrä on $|S|$. Esimerkiksi äskeisessä joukossa $|X| = 3$.

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 äskeisessä joukossa

$$4 \in X \quad \text{ja} \quad 5 \notin X.$$

Uusia joukkoja voidaan muodostaa joukko-operaatioilla seuraavasti:

- **Leikkaus** $A \cap B$ sisältää alkiot, jotka ovat molemmissa joukoista A ja B . Esimerkiksi jos $A = \{1, 2, 5\}$ ja $B = \{2, 4\}$, niin $A \cap B = \{2\}$.
- **Yhdiste** $A \cup B$ sisältää alkiot, jotka ovat ainakin toisessa joukoista A ja B . Esimerkiksi jos $A = \{3, 7\}$ ja $B = \{2, 3, 8\}$, niin $A \cup B = \{2, 3, 7, 8\}$.
- **Komplementti** \bar{A} sisältää alkiot, jotka eivät ole joukossa A . Komplementin tulkinta riippuu siitä, mikä on **perusjoukko** eli joukko, jossa on kaikki mahdolliset alkiot. Esimerkiksi jos $A = \{1, 2, 5, 7\}$ ja perusjoukko on $P = \{1, 2, \dots, 10\}$, niin $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- **Erotus** $A \setminus B = A \cap \bar{B}$ sisältää alkiot, jotka ovat joukossa A mutta eivät joukossa B . Huomaa, että B :ssä voi olla alkioita, joita ei ole A :ssa. Esimerkiksi jos $A = \{2, 3, 7, 8\}$ ja $B = \{3, 5, 8\}$, niin $A \setminus B = \{2, 7\}$.

Merkintä $A \subset S$ tarkoittaa, että A on S :n **osajoukko** eli jokainen A :n alkio

esiintyy S :ssä. Joukon S osajoukkojen yhteismäärä on $2^{|S|}$. Esimerkiksi joukon $\{2, 4, 7\}$ osajoukot ovat

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\} \text{ ja } \{2, 4, 7\}.$$

Usein esiintyviä joukkoja ovat

- \mathbb{N} (luonnolliset luvut),
- \mathbb{Z} (kokonaisluvut),
- \mathbb{Q} (rationaaliluvut) ja
- \mathbb{R} (reaaliluvut).

Luonnollisten lukujen joukko \mathbb{N} voidaan määritellä tilanteesta riippuen kahdella tavalla: joko $\mathbb{N} = \{0, 1, 2, \dots\}$ tai $\mathbb{N} = \{1, 2, 3, \dots\}$.

Joukon voi muodostaa myös säännöllä muotoa

$$\{f(n) : n \in S\},$$

missä $f(n)$ on jokin funktio. Tällainen joukko sisältää kaikki alkiot $f(n)$, jossa n on valittu joukosta S . Esimerkiksi joukko

$$X = \{2n : n \in \mathbb{Z}\}$$

sisältää kaikki parilliset kokonaisluvut.

Logiikka

Loogisen lausekkeen arvo on joko **tosi** (1) tai **epätosi** (0). 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 on tosi. 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.

Kvanttori ilmaisee, että looginen lauseke liittyy jollakin tavalla joukon alkioihin. Tavalliset kvanttorit ovat \forall (kaikille) ja \exists (on olemassa). Esimerkiksi

$$\forall x(\exists y(y < x))$$

tarkoittaa, että jokaiselle joukon alkioille x on olemassa jokin joukon alkio y niin, että y on x :ää pienempi. Tämä lauseke on tosi kokonaislukujen joukossa, mutta lauseke on epätosi luonnollisten lukujen joukossa.

Tässä esitettyjen 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 kokonaislukujen joukossa, 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. Tämän lausekkeen arvo on tosi.

Funktioita

Funktio $\lfloor x \rfloor$ pyöristää luvun x alaspäin kokonaisluvuksi ja funktio $\lceil x \rceil$ pyöristää luvun x ylöspäin kokonaisluvuksi. Esimerkiksi

$$\lfloor 3/2 \rfloor = 1 \quad \text{ja} \quad \lceil 3/2 \rceil = 2.$$

Funktiot $\min(x_1, x_2, \dots, x_n)$ ja $\max(x_1, x_2, \dots, x_n)$ palauttavat pienimmän ja suurimman arvoista x_1, x_2, \dots, x_n . Esimerkiksi

$$\min(1, 2, 3) = 1 \quad \text{ja} \quad \max(1, 2, 3) = 3.$$

Kertoma $n!$ määritellään

$$\prod_{x=1}^n x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

eli toisin sanoen rekursiivisesti

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

Fibonacciin luvut esiintyvät monissa erilaisissa yhteyksissä. Ne määritellään seuraavasti rekursiivisesti:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Ensimmäiset Fibonacciin luvut ovat

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Fibonacciin lukujen laskemiseen on olemassa myös suljetun muodon kaava

$$f(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}.$$

Logaritmi

Luvun x **logaritmi** merkitään $\log_k(x)$, missä k on logaritmin kantaluku. Logaritmin määritelmän mukaan $\log_k(x) = a$ 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 lukua 32 täytyy jakaa 2:lla 5 kertaa:

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

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

Logaritmi muuttaa kertolaskun yhteenlaskuksi:

$$\log_k(ab) = \log_k(a) + \log_k(b)$$

Samoin logaritmi muuttaa jakolaskun vähennyslaskuksi:

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b)$$

Lisäksi on voimassa kaava

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

minkä ansiosta logaritmeja voi laskea mille tahansa kantaluvulle, jos on keino laskea logaritmeja jollekin kantaluvulle.

Luvun x **luonnollinen logaritmi** $\ln(x)$ on logaritmi, jonka kantaluku on **Neperin luku** $e \approx 2,71828$.

Vielä yksi logaritmin ominaisuus on, että luvun x numeroiden määrä b -järjestelmässä on $\lfloor \log_b(x) + 1 \rfloor$. Esimerkiksi luvun 123 esitys 2-järjestelmässä on 1111011 ja $\lfloor \log_2(123) + 1 \rfloor = 7$.

Luku 2

Aikavaativuus

Kisakoodauksessa oleellinen asia on algoritmien tehokkuus. Yleensä on helpoa 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 on kätevä tapa arvioida, kuinka nopeasti algoritmi toimii. Se esittää algoritmin tehokkuuden 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 $\lceil n/2 \rceil$ 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 puolestaan $O(n^2)$:

```
for (int i = 1; i <= n; i++) {  
    for (int j = i+1; j <= n; 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 aikaa $O(1)$, 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
...	...
$g(1)$	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

Usein esiintyviä vaativuusluokkia ovat seuraavat:

$O(1)$ **Vakioaikainen** algoritmi käyttää saman verran aikaa minkä tahansa syötteen käsittelyyn, eli algoritmin nopeus ei riipu syötteen koosta. Tyyppillinen vakioaikainen algoritmi on suora kaava vastauksen laskemiseen.

$O(\log n)$ **Logaritminen** aikavaativuus 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.

$O(\sqrt{n})$ Tällainen algoritmi sijoittuu aikavaativuuksien $O(\log n)$ ja $O(n)$ väli-maastoon. Neliöjuuren erityinen ominaisuus on, että $\sqrt{n} = n/\sqrt{n}$, joten neliöjuuri osuu tietyllä tavalla syötteen puoliväliin.

$O(n)$ **Lineaarinen** algoritmi käy syötteen läpi kiinteään määrän kertoja. Tämä on usein paras mahdollinen aikavaativuus, koska yleensä syöte täytyy käydä läpi ainakin kerran, ennen kuin algoritmi voi ilmoittaa vastauksen.

$O(n \log n)$ Tämä aikavaativuus 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** aikavaativuus voi syntyä siitä, että algoritmissa on kaksi sisäkkäistä silmukkaa. Neliöllinen algoritmi voi käydä läpi kaikki tavat valita joukosta kaksi alkia.

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

$O(2^n)$ Tämä aikavaativuus 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.

$O(n!)$ Tämä aikavaativuus 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 ongelmaa ei osata ratkaista tehokkaasti. **NP-vaikeat** ongelmat ovat tärkeä joukko 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 joitakin satoja miljoonia koodissa olevia komentoja.

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 arviolta kymmeniä sekunteja, 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 Suurin alitaulukon summa

Usein 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 $O(n)$.

Annettuna on taulukko, jossa on n kokonaislukua x_1, x_2, \dots, x_n , ja tehtävänä on etsiä taulukon **suurin alitaulukon summa** eli mahdollisimman suuri summa taulukon yhtenäisellä välillä. Tehtävän kiinnostavuus on siinä, että taulukossa saattaa olla negatiivisia lukuja. Esimerkiksi taulukossa

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

suurimman summan 10 tuottaa seuraava alitaulukko:

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

Ratkaisu 1

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 += x[c];
        }
        p = max(p,s);
    }
}
cout << p << "\n";
```

Koodi olettaa, että luvut on tallennettu taulukkoon x , jota indeksoidaan $1 \dots n$. Muuttujat a ja b valitsevat alitaulukon ensimmäisen ja viimeisen luvun, ja alitaulukon summa lasketaan muuttujaan s . Muuttujassa p on puolestaan 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

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 += x[b];
        p = max(p,s);
    }
}
cout << p << "\n";
```

Tämän muutoksen jälkeen koodin aikavaativuus on $O(n^2)$.

Ratkaisu 3

Yllättävää kyllä, tehtävään on olemassa myös $O(n)$ -aikainen ratkaisu eli koodista pystyy karsimaan vielä yhden silmukan. Ideana on laskea taulukon jokaiseen kohtaan, mikä on suurin alitaulukon summa, jos alitaulukko päättyy kyseiseen kohtaan. Tämän jälkeen ratkaisu tehtävään on suurin näistä summista.

Tarkastellaan alitaulukon etsimistä, kun valittuna on alitaulukon loppukohta k . Vaihtoehtoja on kaksi:

1. Alitaulukossa on vain kohdassa k oleva luku.
2. Alitaulukossa on ensin jokin kohtaan $k - 1$ päättyvä alitaulukko ja sen jälkeen kohdassa k oleva luku.

Koska tavoitteena on löytää alitaulukko, jonka lukujen summa on suurin, tapauksessa 2 myös kohtaan $k - 1$ päättyvän alitaulukon tulee olla sellainen, että sen summa on suurin. Niinpä tehokas ratkaisu syntyy käymällä läpi kaikki alitaulukon loppukohdat järjestyksessä ja laskemalla jokaiseen kohtaan suurin mahdollinen kyseiseen kohtaan päättyvän alitaulukon summa.

Seuraava koodi toteuttaa ratkaisun:

```
int p = 0, s = 0;
for (int k = 1; k <= n; k++) {
    s = max(x[k], s+x[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 nykyaikaisella tietokoneella.

Jokaisessa testissä syöte on muodostettu satunnaisesti. 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 merkittäviä eroja algoritmien suoritussajassa. $O(n^3)$ -aikainen ratkaisu 1 alkaa hidastua, kun $n = 10^3$, ja $O(n^2)$ -aikainen ratkaisu 2 alkaa hidastua, kun $n = 10^4$. Vain $O(n)$ -aikainen ratkaisu 3 selvittää suurimmatkin syötteen salamannopeasti.

Luku 3

Järjestäminen

Järjestäminen on keskeinen algoritmiikan ongelma. Moni tehokas algoritmi perustuu järjestämiseen, koska järjestetyn tiedon käsittely on 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:

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

$O(n^2)$ -algoritmit

Yksinkertaiset algoritmit taulukon järjestämiseen vievät aikaa $O(n^2)$. Ehkä tunnetuin tällainen algoritmi on **kuplajärjestäminen**, joka muodostuu peräkkäisistä taulukon läpikäynneistä.

Algoritmissa jokainen taulukon läpikäynti etsii 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 alkioista $t[1], t[2], \dots, t[n]$:

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

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 $n - 1$ läpikäynnin jälkeen. Koska jokainen läpikäynti vie aikaa $O(n)$, algoritmin aikavaativuus on $O(n^2)$.

Inversiot

Kuplajärjestäminen on esimerkki algoritmista, joka perustuu taulukon vierekkäisten alkioden vaihtamiseen keskenään. Osoittautuu, että tällaisen algoritmin aikavaativuus on aina vähintään $O(n^2)$, koska pahimmassa tapauksessa taulukon järjestäminen vaatii $O(n^2)$ alkioparin vaihtamista.

Hyödyllinen käsite järjestämisalgoritmien analyysissä on **inversio**. Se on taulukossa oleva alkio pari $(t[a], t[b])$, missä $a < b$ ja $t[a] > t[b]$ eli alkiot ovat väärässä järjestyksessä taulukossa. Esimerkiksi taulukon

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

inversiot ovat $(6, 3)$, $(6, 5)$ ja $(9, 8)$. Inversioiden määrä kuvaa, miten lähellä järjestystä taulukko on. Taulukko on järjestyksessä tarkalleen silloin, kun siinä ei ole yhtään inversiota. Inversioiden määrä on suurin, kun taulukon järjestys on käänteinen, jolloin inversioita on

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

Jos vierekkäiset taulukon alkiot ovat väärässä järjestyksessä, niiden järjestyksen korjaaminen poistaa taulukosta tarkalleen yhden inversion. Niinpä jos järjestämisalgoritmi pystyy vaihtamaan keskenään vain taulukon vierekkäisiä alkioita, jokainen vaihto voi poistaa enintään yhden inversion ja algoritmin aikavaativuus on varmasti ainakin $O(n^2)$.

$O(n \log n)$ -algoritmit

Taulukon järjestäminen on mahdollista tehokkaasti ajassa $O(n \log n)$ algoritmilla, joka ei rajoitu vierekkäisten alkioden vaihtamiseen. Yksi tällainen algoritmi on **lomitusjärjestäminen**, joka järjestää taulukon rekursiivisesti jakamalla sen pienemmiksi osataulukoiksi.

Lomitusjärjestäminen jakaa taulukon kahdeksi osataulukoksi, 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 $a \dots b$ alkiot. 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 $c = (a + b)/2$. Funktio järjestää osat rekursiivisesti kutsumalla itseään.

Rekursiivisen järjestämisen jälkeen funktio merge lomittaa välin vasemman ja oikean osan alkiot eli kerää alkiot yhteen niin, että koko väli on järjestyksessä. Lomitus on mahdollista toteuttaa ajassa $O(n)$ valitsemalla alkiot 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

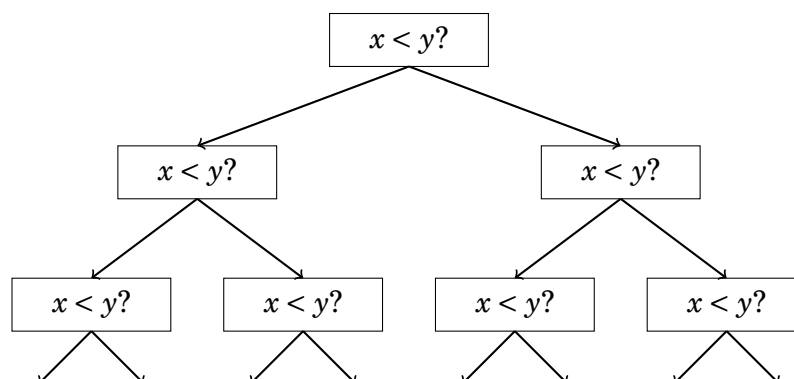
1	2	3	4	5	6	7	8
1	2	2	3	5	6	8	9

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.

Järjestämisen alaraja

Onko sitten mahdollista järjestää taulukkoa nopeammin kuin ajassa $O(n \log n)$? Osoittautuu, että tämä *ei* ole mahdollista, kun rajoitumme järjestämisalgoritmeihin, jotka perustuvat taulukon alkioiden vertailemiseen.

Aikavaativuuden alaraja on mahdollista todistaa tarkastelemalla 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

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

Voimme arvioida tätä summaa alaspäin valitsemalla summasta $n/2$ viimeistä termiä ja muuttamalla kunkin termin arvoksi $\log_2(n/2)$. Tästä saadaan arvio

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

eli puun korkeus ja sen myötä pienin mahdollinen järjestämisalgoritmin askelten määrä on ainakin luokkaa $n \log n$.

Laskemisjärjestäminen

Järjestämisen alaraja $n \log n$ ei koske algoritmeja, jotka eivät perustu alkioiden vertailemiseen vaan hyödyntävät jotain muuta tietoa alkioista. Esimerkki tällaisesta algoritmista on **laskemisjärjestäminen**, jonka avulla on mahdollista järjestää taulukko ajassa $O(n)$ olettaen, että jokainen taulukon alkio on kokonaisluku välillä $0 \dots c$, missä c on pieni vakio.

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 vakio c on niin pieni, että taulukon alkioita voi käyttää kirjanpidon taulukon indeksöinnissä.

3.2 Järjestäminen C++:ssa

Järjestämisalgoritmia ei yleensä koskaan kannata koodata itse, vaan on parempi ratkaisu käyttää ohjelmointikielen valmista toteutusta. Esimerkiksi C++:n standardikirjastossa on funktio `sort`, jonka avulla voi järjestää helposti taulukoita ja muita tietorakenteita.

Valmiin toteutuksen käyttämisessä on monia etuja. Ensinnäkin se säästää aikaa, koska järjestämisalgoritmia ei tarvitse kirjoittaa itse. Lisäksi valmis toteutus on varmasti tehokas ja toimiva: vaikka järjestämisalgoritmin toteuttaisi itse, siitä tulisi tuskin valmista toteutusta parempi.

Tutustumme seuraavaksi C++:n `sort`-funktion käyttämiseen. Seuraava koodi järjestää vektorin `v` luvut pienimmästä suurimpaan:

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

Järjestämisen seurauksena vektorin sisällöksi tulee {2,3,3,4,5,5,8}. Oletuksena `sort`-funktio järjestää siis alkiot 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 taas 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”.

Vertailuoperaattori

Funktion `sort` käyttäminen vaatii, että järjestettävien alkioden tietotyypille 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 järjestys on (1,2), (1,5), (2,3).

Vastaavasti tuple-rakenteet järjestyvät ensisijaisesti ensimmäisen kentän, toissijaisesti toisen kentän, jne., mukaan. Seuraava koodi esittelee asiaa:

```
vector<tuple<int,int,int>> v;  
v.push_back(make_tuple(2,1,4));  
v.push_back(make_tuple(1,5,3));  
v.push_back(make_tuple(2,1,3));  
sort(v.begin(), v.end());
```

Koodin suorituksen jälkeen järjestys on (1,5,3), (2,1,3), (2,1,4).

Omat tietueet

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;  
    }  
};
```

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 alkia taulukosta on käyttää for-silmukkaa, joka käy läpi taulukon sisällön alusta loppuun. Esimerkiksi seuraava koodi etsii alkia x taulukosta t .

```
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 mikä tahansa, tämä on kuitenkin tehokkain mahdollinen menetelmä, koska saatavilla ei ole lisätietoa siitä, mistä päin taulukkoa alkia x kannattaa etsiä.

Tilanne on toinen silloin, kun taulukko on järjestyksessä. Tässä tapauksessa haku on mahdollista toteuttaa paljon nopeammin, koska taulukon järjestys ohjaa etsimään alkia oikeasta suunnasta. Seuraavaksi käsiteltävä **binäärihaku** löytää alkion järjestyksestä taulukosta tehokkaasti ajassa $O(\log n)$.

Toteutus 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 keskimmäinen alkio on etsittävä alkio, haku päättyy. Muuten haku jatkuu taulukon vasempaan tai oikeaan osaan sen mukaan, onko keskimmäinen 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 $O(\log n)$.

Toteutus 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.

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

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

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 kokonaisaikavaativuus riippuu siitä, kauanko funktion ok suoritus kestää. Esimerkiksi jos ratkaisun tarkastus vie aikaa $O(n)$, niin kokonaisaikavaativuus on $O(n \log z)$.

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 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

Dynaaminen taulukko on taulukko, jonka kokoa voi muuttaa ohjelman suorituksen aikana. C++:n tavallisin dynaaminen taulukko on **vektori** (vector). Sitä voi käyttää hyvin samalla tavalla kuin tavallista taulukkoa.

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> v(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 $O(1)$.

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 Joukkorakenne

Joukko on tietorakenne, joka sisältää kokoelman alkioita. 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
```

Joukkoa voi käsitellä muuten suunnilleen samoin kuin vektoria, mutta joukkoa ei voi indeksoida `[]`-merkinnällä. Seuraava koodi luo joukon, tulostaa sen alkioden määrän ja käy sitten läpi kaikki alkiot.

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

Tärkeä joukon ominaisuus on, että tietty alkio voi esiintyä siinä enintään kerran. Niinpä funktio `count` palauttaa aina arvon 0 (alkiota ei ole joukossa) tai 1 (alkio on joukossa) ja funktio `insert` ei lisää alkia uudelleen 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`-rakenteesta:

```
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 on taulukon yleistys, joka sisältää kokoelman avain-arvo-pareja. Taulukon avaimet ovat aina peräkkäiset kokonaisluvut $0, 1, \dots, n-1$, missä n on taulukon koko, mutta hakemiston avaimet voivat olla mitä tahansa tyyppiä eikä niiden tarvitse olla peräkkäin.

C++ sisältää kaksi toteutusta hakemistolle samaan tapaan kuin joukolle. Rakenne `map` perustuu tasapainoiseen binääripuuhun ja sen alkioden käsittely vie aikaa $O(\log n)$, kun taas rakenne `unordered_map` perustuu hajautustauluun ja sen alkioden käsittely vie keskimäärin aikaa $O(1)$.

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
```

Funktiolla `count` voi tutkia, esiintyykö avain hakemistossa:

```
if (m.count("aybabbu")) {  
    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** 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.

Välien käsittely

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);
```

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(x)` palauttaa iteraattorin joukon alkioon, jonka arvo on `x`. Poikkeuksena jos alkioita `x` 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`. Näitä funktioita ei voi käyttää `unordered_set`-rakenteessa, joka ei pidä yllä alkioiden järjestystä.

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


```

auto a = s.lower_bound(x);
if (a == s.begin() && a == s.end()) {
    cout << "joukko on tyhjä\n";
} else 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";
}

```

Koodi käy läpi mahdolliset tapaukset iteraattorin `a` avulla. Iteraattori osoittaa aluksi pienimpään alkioon, joka on ainakin yhtä suuri kuin x . Jos `a` on samaan aikaan `begin` ja `end`, joukko on tyhjä. Muuten jos `a` on `begin`, sen osoittama alkio on x :ää lähin alkio. Jos taas `a` on `end`, x :ää lähin alkio on joukon viimeinen alkio. Jos mikään edellisistä tapauksista ei päde, niin x :ää lähin alkio on joko `a`:n osoittama alkio tai sitä edellinen alkio.

4.5 Muita tietorakenteita

Bittijoukko

Bittijoukko (bitset) on taulukko, jonka jokaisen alkion arvo on 0 tai 1. Esimerkiksi seuraava koodi luo bittijoukon, jossa on 10 alkiota.

```

bitset<10> s;
s[2] = 1;
s[5] = 1;
s[6] = 1;
s[8] = 1;
cout << s[4] << "\n"; // 0
cout << s[5] << "\n"; // 1

```

Bittijoukon etuna on, että se vie tavallista taulukkoa vähemmän muistia, koska jokainen alkio vie vain yhden bitin muistia. Esimerkiksi n bitin tallentaminen int-taulukukkona vie $32n$ bittiä tilaa, mutta bittijoukkona vain n bittiä tilaa. Lisäksi bittijoukon sisältöä voi käsitellä tehokkaasti bittiopeeraatioilla, minkä ansiosta sillä voi tehostaa algoritmeja.

Seuraava koodi näyttää toisen tavan bittijoukon luomiseen:

```

bitset<10> s(string("0010011010"));
cout << s[4] << "\n"; // 0
cout << s[5] << "\n"; // 1

```

Funktio `count` palauttaa bittijoukon ykkösbittien määrän:

```
bitset<10> s(string("0010011010"));  
cout << s.count() << "\n"; // 4
```

Seuraava koodi näyttää esimerkkejä bittioperaatioiden käyttämisestä:

```
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
```

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.

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 alkia.

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
```

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 kekkoon perustuva sisäinen toteutus on yksinkertaisempi kuin set-rakenteen tasapainoinen 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 Vertailu järjestämiseen

Monen tehtävän voi ratkaista tehokkaasti joko käyttäen sopivia tietorakenteita tai taulukon järjestämistä. Vaikka erilaiset ratkaisutavat olisivat kaikki periaatteessa tehokkaita, niissä voi olla käytännössä merkittäviä eroja.

Tarkastellaan ongelmaa, jossa annettuna on kaksi listaa A ja B , joista kummassakin on n kokonaislukua. Tehtävänä on selvittää, moniko luku esiintyy kummassakin listassa. Esimerkiksi jos listat ovat

$$A = [5, 2, 8, 9, 4] \quad \text{ja} \quad B = [3, 2, 9, 5],$$

niin vastaus on 3, koska luvut 2, 5 ja 9 esiintyvät kummassakin listassa. Suora-
viivainen ratkaisu tehtävään on käydä läpi kaikki lukuparit ajassa $O(n^2)$, mutta seuraavaksi keskitymme tehokkaampiin ratkaisuihin.

Ratkaisu 1

Tallennetaan listan A luvut joukkoon ja käydään sitten läpi listan B luvut ja tarkistetaan jokaisesta, esiintyykö se myös listassa A . Joukon ansiosta on tehokasta tarkastaa, esiintyykö listan B luku listassa 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 helppo tapa parantaa algoritmin tehokkuutta, koska algoritmin toteutus säilyy samana ja vain tietorakenne vaihtuu. Uuden algoritmin aikavaativuus on $O(n)$.

Ratkaisu 3

Tietorakenteiden sijasta voimme käyttää 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 $O(n \log n)$.

Tehokkuusvertailu

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

Ratkaisut 1 ja 2 ovat muuten samanlaisia, mutta ratkaisu 1 käyttää set-rakennetta, kun taas ratkaisu 2 käyttää unordered_set-rakennetta. Tässä tapauksessa tällä valinnalla on merkittävä vaikutus suoritusaikaan, koska ratkaisu 2 on 4–5 kertaa nopeampi kuin ratkaisu 1.

Tehokkain ratkaisu on kuitenkin järjestämistä käyttävä ratkaisu 3, joka on vielä puolet nopeampi kuin ratkaisu 2. Kiinnostavaa on, että sekä ratkaisun 1 että ratkaisun 3 aikavaativuus on $O(n \log n)$, mutta siitä huolimatta ratkaisu 3 vie aikaa vain kymmenesosan. Tämän voi selittää sillä, että järjestäminen on kevyt operaatio ja se täytyy tehdä vain kerran ratkaisussa 3 algoritmin alussa, minkä jälkeen algoritmin loppuosa on lineaarinen. Ratkaisu 1 taas pitää yllä monimutkaista tasapainoista binääripuuta koko algoritmin ajan.

Luku 5

Täydellinen haku

Täydellinen haku 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

Aloitamme tapauksesta, jossa tehtävän mahdollisia ratkaisuja ovat n -alkioisen joukon osajoukot. Tällöin täydellisen haun tulee käydä läpi kaikki joukon osajoukot, joita on yhteensä 2^n kappaletta. Käymme seuraavaksi läpi kaksi menetelmää tällaisen haun toteuttamiseen.

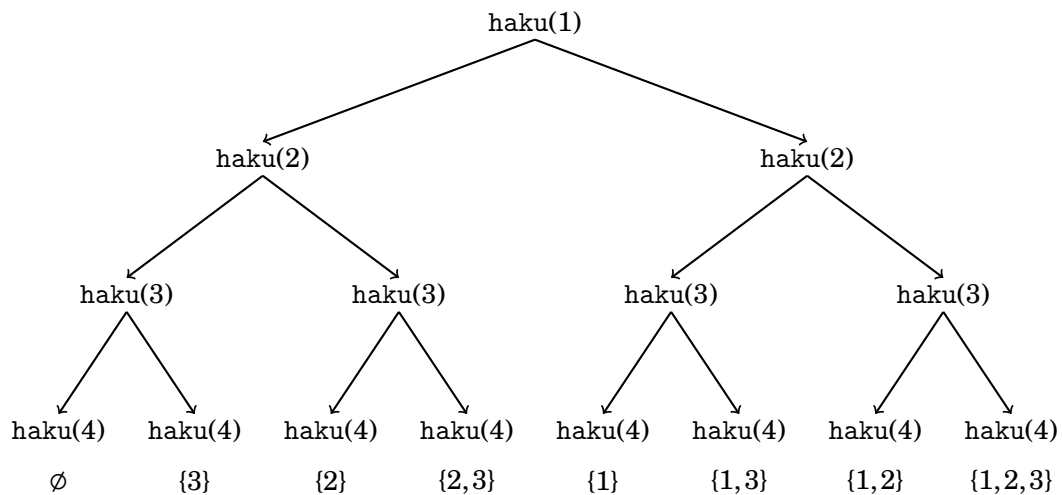
Menetelmä 1

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 muodostaminen alkaa tekemällä funktiokutsu $\text{haku}(1)$.

```
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 $n = 3$, funktiokutsut muodostavat seuraavan kuvan mukaisen puun. 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.

Tavallinen käytäntö on tulkita kokonaisluvun bittiesitys osajoukkona niin, että alkio k kuuluu osajoukkoon, jos lopusta lukien k . bitti on 1. Esimerkiksi luvun 25 bittiesitys on 11001, mikä vastaa osajoukkoa $\{1, 4, 5\}$.

Seuraava koodi käy läpi n alkion joukon osajoukkojen bittiesitykset:

```

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

```

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

Toinen usein esiintyvä tilanne on, että tehtävän ratkaisut ovat n -alkioisen joukon permutaatioita, jolloin täydellisen haun tulee käydä läpi $n!$ mahdollista permutaatiota. Myös tässä tapauksessa on kaksi luontevaa menetelmää täydellisen haun toteuttamiseen.

Menetelmä 1

Osajoukkojen tavoin permutaatioita voi muodostaa rekursiivisesti. Seuraava funktio 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

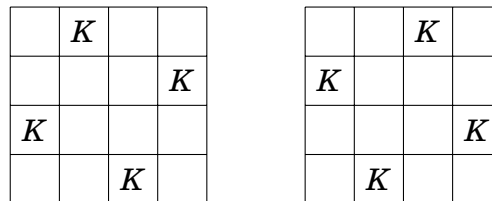
Toinen ratkaisu on aloittaa permutaatiosta $\{1, 2, \dots, n\}$ ja muodostaa joka askeleella järjestyksessä seuraava permutaatio. C++:n standardikirjastossa on funktio `next_permutation`, joka tekee tämän muunnoksen. Seuraava koodi käy läpi joukon $\{1, 2, \dots, n\}$ permutaatiot funktion avulla:

```
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

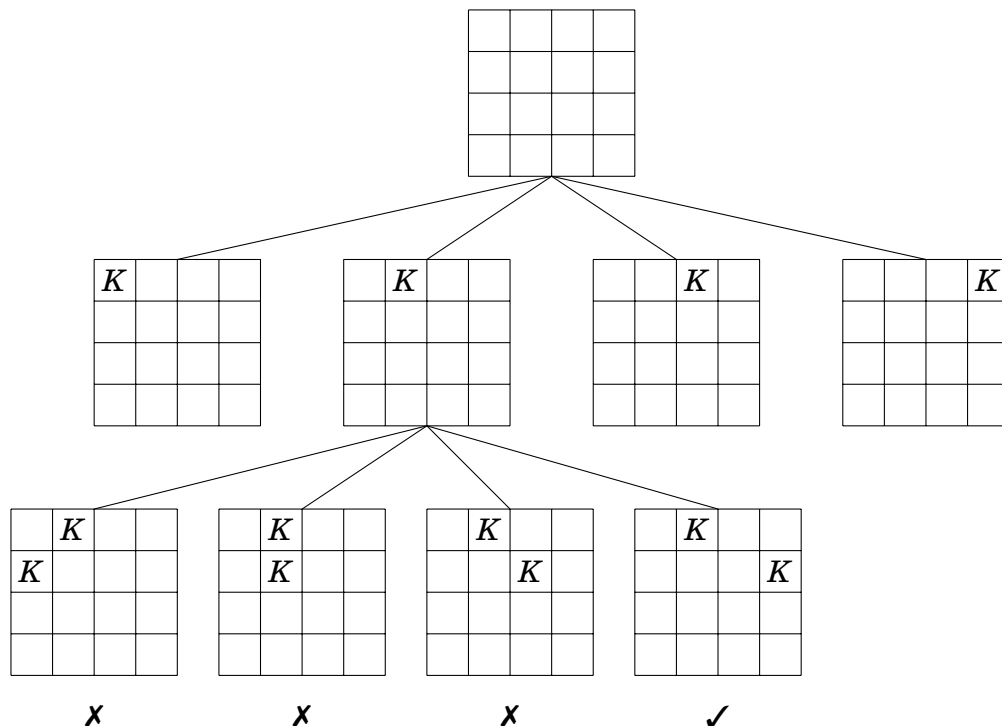
Peruuttava haku aloittaa ratkaisun etsimisen tyhjästä ja laajentaa ratkaisua askel kerrallaan. Joka askeleella haku haarautuu kaikkiin mahdollisiin suuntiin, joihin ratkaisua voi laajentaa. Haaran tutkimisen jälkeen haku peruuttaa takaisin ja jatkaa muihin mahdollisiin suuntiin.

Tarkastellaan esimerkkinä **kuningatarongelmaa**, jossa laskettavana on, monellako tavalla $n \times n$ -shakkilaudalle voidaan asettaa n kuningatarta 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 asettamalla vielä kaksi kuningatarta laudalle.

Seuraava koodi toteuttaa peruuttavan haun:

```
void haku(int y) {
    if (y == n) {
        c++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (r1[x] || r2[x+y] || r3[x-y+n-1]) continue;
        r1[x] = r2[x+y] = r3[x-y+n-1] = 1;
        haku(y+1);
        r1[x] = r2[x+y] = r3[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 `r1` pitää kirjaa, millä laudan pystyriveillä on jo kuningatar. Vastaavasti taulukot `r2` ja `r3` pitävät kirjaa vinoriveistä. Tällaisille riveille ei voi laittaa enää toista kuningatarta. Esimerkiksi 4×4 -laudan tapauksessa rivit on numeroitu seuraavasti:

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

`r1`

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

`r2`

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

`r3`

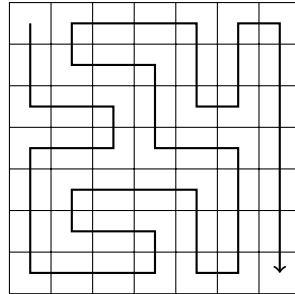
Koodin avulla selviää esimerkiksi, että tapauksessa $n = 8$ on 92 tapaa sijoittaa 8 kuningatarta 8×8 -laudalle. Kun n kasvaa, koodi hidastuu nopeasti, koska ratkaisujen määrä kasvaa räjähdysmäisesti. Tapauksen $n = 16$ laskeminen vie jo noin minuutin nykyaikaisella tietokoneella (14772512 ratkaisua).

5.4 Haun optimointi

Peruuttavaa hakua on usein mahdollista tehostaa erilaisten optimointien avulla. Ideana on lisätä hakuun ”älykkyyttä” niin, että haku pystyy havaitsemaan mahdollisimman aikaisin, jos muodosteilla oleva ratkaisu ei voi johtaa kokonaiseen ratkaisuun. Tällaiset optimoinnit karsivat haaroja hakupuusta, millä voi olla suuri vaikutus peruuttavan haun tehokkuuteen.

Tarkastellaan esimerkkinä tehtävää, jossa laskettavana on reittien määrä $n \times n$ -ruudukon vasemmasta yläkulmasta oikeaan alakulmaan, kun reitin ai-

kana tulee käydä tarkalleen kerran jokaisessa ruudussa. 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.

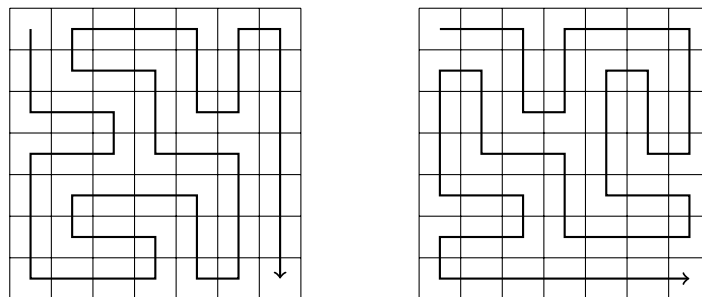
Perusalgoritmi

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. Esimerkiksi seuraavat ratkaisut ovat symmetrisiä keskenään:

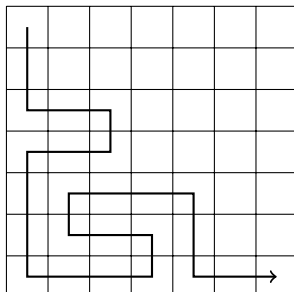


Tämän ansiosta voimme tehdä päätöksen, että reitin ensimmäinen askel on alaspäin, ja kertoa lopuksi reittien määrän 2:lla.

- 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. Näin on esimerkiksi seuraavassa tilanteessa:

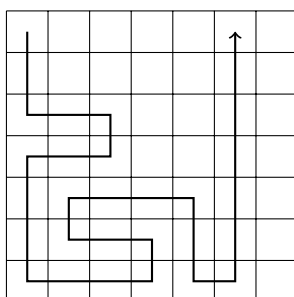


Niinpä voimme keskeyttää hakuhaaran heti, jos tulemme oikean alakulman ruutuun liian aikaisin.

- 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. Esimerkiksi seuraavassa tilanteessa sekä vasemmalla että oikealla puolella on tyhjä ruutu:



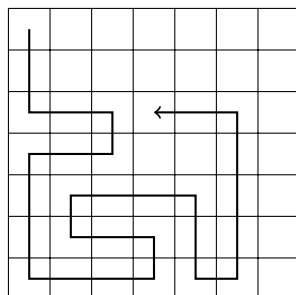
Nyt ei ole enää mahdollista käydä kaikissa ruuduissa, joten voimme keskeyttää hakuhaaran. Tämä optimointi on hyvin hyödyllinen:

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

Optimointi 4

Äskeisen optimoinnin ideaa voi yleistää: ruudukko jakaantuu kahteen osaan, jos nykyisen ruudun ylä- ja alapuolella on tyhjä ruutu sekä vasemmalla ja oikealla puolella on seinä tai aiemmin käyty ruutu (tai päinvastoin).

Esimerkiksi seuraavassa tilanteessa nykyisen ruudun ylä- ja alapuolella on tyhjä ruutu eikä reitti voi enää edetä molempiin ruutuihin:



Haku tehostuu entisestään, kun keskeytämme hakuhaaran kaikissa tällaisissa tapauksissa:

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

Nyt on hyvä hetki lopettaa optimointi ja muistella, mistä lähdimme liikkeelle. Alkuperäinen algoritmi vei aikaa 483 sekuntia, ja nyt optimointien jälkeen algoritmi vie aikaa vain 0,6 sekuntia. Optimointien ansiosta algoritmi nopeutui siis lähes 1000-kertaisesti.

Tämä on yleinen ilmiö peruuttavassa haussa, koska hakupuun on yleensä valtava ja yksinkertainenkin optimointi voi karsia suuren määrän haaroja hakupuusta. Erityisen hyödyllisiä ovat optimoinnit, jotka kohdistuvat hakupuun yläosaan, koska ne karsivat eniten haaroja.

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 käyttäminen edellyttää, että erillisten hakujen tulokset pysyy yhdistämään tehokkaasti. Tällöin puolivälihaku on tehokkaampi kuin yksi haku, joka käy läpi koko hakualueen. Tyypillisesti puolivälihaku tehostaa algoritmia niin, että aikavaativuuden kertoimesta 2^n tulee kerroin $2^{n/2}$.

Tarkastellaan ongelmaa, jossa annettuna on n lukua sisältävä lista sekä kokonaisluku x . Tehtävänä on selvittää, voiko listalta valita joukon lukuja niin, että niiden summa on x . Esimerkiksi jos lista on $[2, 4, 5, 9]$ ja $x = 15$, voimme valita listalta luvut $[2, 4, 9]$, jolloin $2 + 4 + 9 = 15$. Jos taas lista säilyy ennallaan ja $x = 14$, mikään valinta ei täytä vaatimusta.

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 . Seuraavaksi näemme, miten puolivälihaun avulla on mahdollista luoda tehokkaampi $O(2^{n/2})$ -aikainen ratkaisu. Huomaa, että aikavaativuuksissa $O(2^n)$ ja $O(2^{n/2})$ on merkittävä ero, koska $2^{n/2}$ tarkoittaa samaa kuin $\sqrt{2^n}$.

Ideana on jakaa syötteenä oleva lista kahteen listaan A ja B , joista kumpikin sisältää noin puolet luvuista. Ensimmäinen haku muodostaa kaikki osajoukot listan A luvuista ja laittaa muistiin niiden summat listaan S_A . Toinen haku muodostaa vastaavasti listan B perusteella listan 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 . Tämä on mahdollista tarkalleen silloin, kun alkuperäisen listan luvuista saa summan x .

Tarkastellaan esimerkkiä, jossa lista on $[2, 4, 5, 9]$ ja $x = 15$. Puolivälihaku jakaa luvut kahteen listaan niin, että $A = [2, 4]$ ja $B = [5, 9]$. Näistä saadaan edelleen 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 valinta vastaa ratkaisua $[2, 4, 9]$.

Ratkaisun aikavaativuus on $O(2^{n/2})$, koska kummassakin listassa A ja B on $n/2$ lukua ja niiden osajoukkojen summien laskeminen listoihin S_A ja S_B vie aikaa $O(2^{n/2})$. Tämän jälkeen on mahdollista tarkastaa ajassa $O(2^{n/2})$, voiko summaa x muodostaa listojen S_A ja S_B luvuista.

Luku 6

Ahneet algoritmit

Ahne algoritmi muodostaa ongelman ratkaisun tekemällä joka askeleella sil- lä hetkellä parhaalta näyttävän valinnan. Ahne algoritmi ei koskaan peruuta tekemiään valintoja vaan muodostaa ratkaisun suoraan valmiiksi. Tämän an- siosta 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 perusteleminen on usein hankalaa.

6.1 Kolikkotehtävä

Aloitamme ahneisiin algoritmeihin tutustumisen tehtävästä, jossa muodostet- tavana on rahamäärä x kolikoista. Kolikoiden arvot ovat $\{c_1, c_2, \dots, c_k\}$, ja jokais- ta kolikkoa on saatavilla rajattomasti. Tehtävänä on selvittää, mikä on pienin määrä kolikoita, joilla rahamäärän voi muodostaa.

Esimerkiksi jos muodostettava rahamäärä on 520 ja kolikot ovat eurokolikot eli sentteinä

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

niin kolikoita tarvitaan vähintään 4. Tämä on mahdollista valitsemalla kolikot $\{20, 100, 200, 200\}$, joiden summa on 520.

Ahne algoritmi

Luonteva ahne algoritmi tehtävään on poistaa rahamäärästä aina mahdollisim- man 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* oi- kein, 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. Tä- mä johtuu siitä, että jos ratkaisussa olisi kaksi tällaista kolikkoa, saman ratkai-

sun 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 työlästä, 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 ahneesti. Klassinen ongelma on seuraava: 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

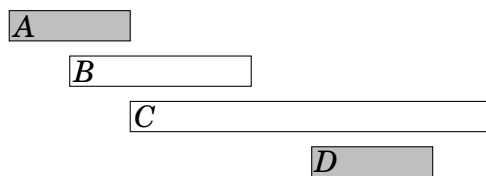
niin on mahdollista osallistua korkeintaan kahteen tapahtumaan. Yksi mahdollisuus on osallistua tapahtumiin *B* ja *D* seuraavasti:



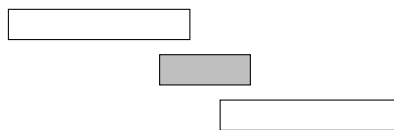
Tehtävän ratkaisuun on mahdollista keksiä useita ahneita algoritmeja, mutta mikä niistä toimii kaikissa tapauksissa?

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, vaan algoritmi epäonnistuu esimerkiksi seuraavassa tilanteessa:



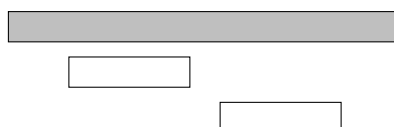
Kun lyhyt tapahtuma valitaan mukaan, on mahdollista osallistua vain yhteen tapahtumaan. Kuitenkin valitsemalla pitkät tapahtumat olisi mahdollista osallistua kahteen tapahtumaan.

Algoritmi 2

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



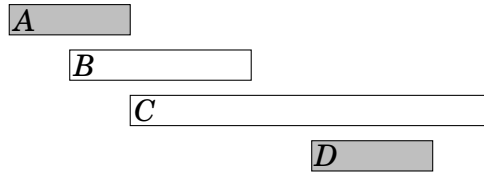
ja tuottaa optimiratkaisun. Tämä algoritmi ei kuitenkaan toimi esimerkiksi seuraavassa tilanteessa:



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

Algoritmi 3

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



ja tuottaa optimiratkaisun. Osoittautuu, että tämä ahne algoritmi tuottaa *aina* optimiratkaisun. 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 kuin aiemmin päättyvän tapahtuman jälkeen.

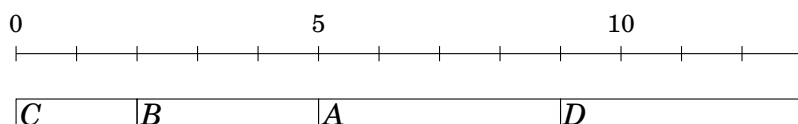
6.3 Tehtävät ja deadlinet

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 yhteispistemäärä, jonka voit saada tehtävistä?

Esimerkiksi jos tehtävät ovat

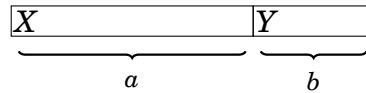
tehtävä	kesto	deadline
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

niin optimaalinen ratkaisu on suorittaa tehtävät seuraavasti:

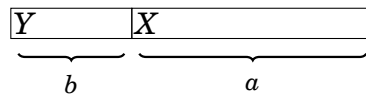


Tässä ratkaisussa C tuottaa 5 pistettä, B tuottaa 0 pistettä, A tuottaa -7 pistettä ja D tuottaa -8 pistettä, joten yhteispistemäärä on -10 .

Yllättävää kyllä, tehtävän optimaalinen ratkaisu ei riipu lainkaan deadlineista, vaan toimiva ahne strategia on yksinkertaisesti 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.

6.4 Keskiluvut

Tarkastelemme seuraavaksi ongelmaa, jossa annettuna on n lukua a_1, a_2, \dots, a_n ja tehtävänä on etsiä luku x niin, että summa

$$|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c$$

on mahdollisimman pieni. Keskitymme tapauksiin, joissa $c = 1$ tai $c = 2$.

Tapaus $c = 1$

Tässä tapauksessa minimoitavana on summa

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

Esimerkiksi jos luvut ovat $[1, 2, 9, 2, 6]$, niin paras ratkaisu on valita $x = 2$, 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, ja 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.

Tapaus $c = 2$

Tässä tapauksessa minimoitavana on summa

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

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

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

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

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

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

6.5 Tiedonpakkaus

Annettuna on merkkijono ja tehtävänä on *pakata* se niin, että tilaa kuluu vähemmän. Käytämme tähän **binäärikoodia**, joka määrittää kullekin merkillä biteistä muodostuvan **koodisanan**. Tällöin merkkijonon voi pakata korvaamalla jokaisen merkin vastaavalla koodisanalla. Esimerkiksi seuraava binäärikoodi määrittää koodisanat merkeille A–D:

merkki	koodisana
A	00
B	01
C	10
D	11

Tämä koodi on **vakiopituinen** eli jokainen koodisana on yhtä pitkä. Esimerkiksi merkkijono AABACDACA on pakattuna

000001001011001000,

eli se vie tilaa 18 bittiä. Pakkausta on kuitenkin mahdollista parantaa ottamalla käyttöön **muuttuvan pituinen** koodi, jossa koodisanojen pituus voi vaihdella. Tällöin voimme antaa usein esiintyville merkeille lyhyen koodisanan ja harvoin esiintyville merkeille pitkän koodisanan. Osoittautuu, että yllä olevalle merkkijonolle **optimaalinen** koodi on seuraava:

merkki	koodisana
A	0
B	110
C	10
D	111

Tätä koodia käyttäen merkkijono on pakattuna

001100101110100,

ja tilaa kuluu vain 15 bittiä. Paremman koodin ansiosta onnistuimme siis säästämään 3 bittiä tilaa pakkauksessa.

Huomaa, että koodin tulee olla aina sellainen, että mikään koodisana ei ole toisen koodisanan alkuosa. Esimerkiksi ei ole sallittua, että koodissa olisi molemmat koodisanat 10 ja 1011. Tämä rajoitus johtuu siitä, että haluamme myös pystyä palauttamaan alkuperäisen merkkijonon pakkauksen jälkeen. Jos koodisana voisi olla toisen alkuosa, tämä ei välttämättä olisi mahdollista. Esimerkiksi seuraava koodi *ei* ole kelvollinen:

merkki	koodisana
A	10
B	11
C	1011
D	111

Tätä koodia käyttäen ei olisi mahdollista tietää, tarkoittaako pakattu merkkijono 1011 merkkijonoa AB vai merkkijonoa C.

Huffmanin koodaus

Huffmanin koodaus on ahne algoritmi, joka muodostaa optimaalisen koodin merkkijonon pakkaamista varten. Se muodostaa merkkien esiintymiskertojen perustella binääripuun, josta voidaan lukea merkkejä vastaavat koodisanat liikkumalla huipulta merkkiä vastaavaan solmuun. Liikkuminen vasemmalle vastaa bittiä 0 ja liikkuminen oikealle vastaa bittiä 1.

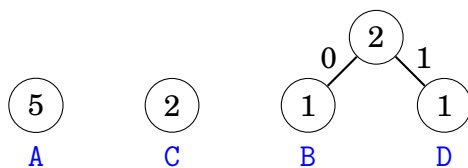
Aluksi jokaista merkkijonon merkkiä vastaa solmu, jonka painona on merkin esiintymiskertojen määrä merkkijonossa. Sitten joka vaiheessa puusta valitaan kaksi painoltaan pienintä solmua ja ne yhdistetään luomalla niiden yläpuolelle uusi solmu, jonka paino on solmujen yhteispaino. Näin jatketaan, kunnes kaikki solmut on yhdistetty ja koodi on valmis.

Tarkastellaan nyt, miten Huffmanin koodaus muodostaa optimaalisen koodin merkkijonolle AABACDACA. Alkutilanteessa on neljä solmua, jotka vastaavat merkkijonossa olevia merkkejä:

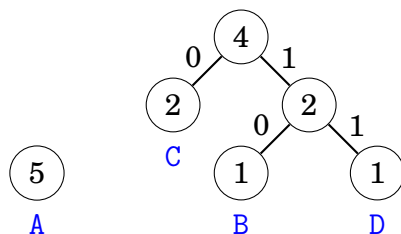


Merkkiä A vastaavan solmun paino on 5, koska merkki A esiintyy 5 kertaa merkkijonossa. Muiden solmujen painot on laskettu vastaavalla tavalla.

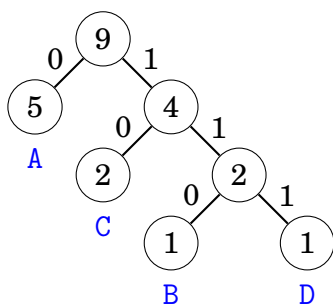
Ensimmäinen askel on yhdistää merkkejä B ja D vastaavat solmut, joiden kummankin paino on 1. Tuloksena on:



Tämän jälkeen yhdistetään solmut, joiden paino on 2:



Lopuksi yhdistetään kaksi viimeistä solmua:



Nyt kaikki solmut ovat puussa, joten koodi on valmis. Puusta voidaan lukea seuraavat koodisanat:

merkki	koodisana
A	0
B	110
C	10
D	111

Luku 7

Dynaaminen ohjelmointi

Dynaaminen ohjelmointi 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:

- **Optimiratkaisun etsiminen:** Haluamme etsiä ratkaisun, joka on jollakin tavalla suurin mahdollinen tai pienin mahdollinen.
- **Ratkaisuiden määrän laskeminen:** Haluamme laskea, kuinka monta mahdollista ratkaisua on olemassa.

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 Kolikkotehtävä

Aloitamme dynaamisen ohjelmoinnin tutun tehtävän kautta: Muodostettavana on rahamäärä x käyttäen mahdollisimman vähän kolikoita. Kolikoiden arvot ovat $\{c_1, c_2, \dots, c_k\}$ ja jokaista kolikko on saatavilla rajattomasti.

Luvussa 6.1 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. Algoritmi perustuu rekursiiviseen funktioon, joka käy läpi kaikki vaihtoehdot rahamäärän muodostamiseen täydellisen haun kaltaisesti. Algoritmi toimii kuitenkin tehokkaasti, koska se talentaa välituloksia muistitaulukkoon, minkä ansiosta sen ei tarvitse laskea samoja asioita moneen kertaan.

Rekursiivinen esitys

Dynaamisessa ohjelmoinnissa on ideana esittää ongelma rekursiivisesti niin, että ongelman ratkaisun voi laskea saman ongelman pienempien tapausten ratkaisuksista. 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:

$$\begin{aligned}f(0) &= 0 \\f(1) &= 1 \\f(2) &= 2 \\f(3) &= 1 \\f(4) &= 1 \\f(5) &= 2 \\f(6) &= 2 \\f(7) &= 2 \\f(8) &= 2 \\f(9) &= 3 \\f(10) &= 3\end{aligned}$$

Nyt $f(0) = 0$, 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

$$f(0) = 0,$$

koska rahamäärän 0 muodostamiseen ei tarvita yhtään kolikkoa. Lisäksi on hyvä määritellä

$$f(x) = \infty, \text{ jos } x < 0.$$

Tämä tarkoittaa, että negatiivisen rahamäärän muodostaminen vaatii äärettömästi kolikoita, mikä estää sen, että rekursio muodostaisi ratkaisun, johon kuuluu negatiivinen rahamäärä.

Nyt voimme toteuttaa funktion C++:lla suoraan rekursiivisen määritelmän perusteella:

```
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 $c[1], c[2], \dots, c[k]$, 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.

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 . Käytännössä ratkaisu on mahdollista toteuttaa, jos x on niin pieni, että on mahdollista varata riittävän suuri muistitaulukko.

Huomaa, että muistitaulukon voi muodostaa myös suoraan silmukalla ilman rekursiota laskemalla arvot pienimmästä suurimpaan:

```
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;
}
```

Silmukkatoteutus on lyhyempi ja hieman tehokkaampi kuin rekursiototeutus, minkä vuoksi kokeneet kisakoodarit toteuttavat dynaamisen ohjelmoinnin usein silmukan avulla. Kuitenkin silmukkatoteutuksen taustalla on sama rekursiivinen idea kuin ennenkin.

Ratkaisun muodostaminen

Joskus optimiratkaisun arvon selvittämisen lisäksi täytyy muodostaa näytteeksi yksi mahdollinen optimiratkaisu. Tässä tehtävässä tämä tarkoittaa, että ohjelman täytyy antaa esimerkki tavasta valita kolikot, joista muodostuu rahamäärä x käyttäen mahdollisimman vähän kolikoita.

Ratkaisun muodostaminen 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];  
}
```

Ratkaisuiden määrän laskeminen

Tarkastellaan sitten kolikkotehtävän muunnelmaa, joka on muuten samanlainen kuin ennenkin, mutta laskettavana on mahdollisten ratkaisuiden yhteismäärä optimaalisen ratkaisun sijasta. Esimerkiksi jos kolikot ovat $\{1, 3, 4\}$ ja rahamäärä on 5, niin ratkaisuja on kaikkiaan 6:

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

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

Tässä tapauksessa voimme määritellä funktion $f(x)$, joka kertoo, monella 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 riviä

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

jälkeen rivi

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

Nyt olemme käyneet läpi kaikki dynaamisen ohjelmoinnin perusasiat. Dynaamista ohjelmointia voi soveltaa monilla tavoilla erilaisissa tehtävissä, minä vuoksi tutustumme seuraavaksi joukkoon tehtäviä, jotka esittelevät dynaamisen ohjelmoinnin mahdollisuuksia.


7.2 Pisin nouseva alijono

Annettuna on taulukko, jossa on n kokonaislukua x_1, x_2, \dots, x_n . Tehtävänä on selvittää, kuinka pitkä on taulukon **pin nouseva alijono** eli vasemmalta oikealle kulkeva ketju taulukon alkioita, jotka on valittu niin, että jokainen alkio on edellistä suurempi. 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



Merkitään $f(k)$ kohtaan k päättyvän pisin nousevan alijonon pituutta, jolloin ratkaisu tehtävään on suurin arvoista $f(1), f(2), \dots, f(n)$. Esimerkiksi yllä

olevassa taulukossa funktion arvot ovat seuraavat:

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

Arvon $f(k)$ laskemisessa on kaksi vaihtoehtoa, millainen kohtaan k päättyvä pisin nouseva alijono on:

1. Pisin nouseva alijono sisältää vain luvun x_k , jolloin $f(k) = 1$.
2. Valitaan jokin kohta i , jolle pätee $i < k$ ja $x_i < x_k$. Pisin nouseva alijono saadaan liittämällä kohtaan i päättyvän pisimmän nousevan alijonon perään luku x_k . Tällöin $f(k) = f(i) + 1$.

Tarkastellaan esimerkkinä arvon $f(7)$ laskemista. Paras ratkaisu on ottaa pohjaksi kohtaan 5 päättyvä pisin nouseva alijono $[2, 5, 7]$ ja lisätä sen perään luku $x_7 = 8$. Tuloksena on alijono $[2, 5, 7, 8]$ ja $f(7) = f(5) + 1 = 4$.

Suoraviivainen tapa toteuttaa algoritmi on käydä kussakin kohdassa k läpi kaikki kohdat $i = 1, 2, \dots, k - 1$, joissa voi olla alijonon edellinen luku. Tällaisen algoritmin aikavaativuus on $O(n^2)$. Yllättävää kyllä, algoritmin voi toteuttaa myös ajassa $O(n \log n)$, mutta tämä on vaikeampaa.

7.3 Reitinhaku ruudukossa

Seuraava tehtävämme on etsiä reitti $n \times n$ -ruudukon vasemmasta yläkulmasta oikeaan alakulmaan. Jokaisessa ruudussa on luku, ja reitti tulee muodostaa niin, että reittiin kuuluvien lukujen summa on mahdollisimman suuri. Rajoituksena ruudukossa on mahdollista liikkua vain oikealla ja alaspäin.

Seuraavassa ruudukossa paras reitti on merkitty harmaalla taustalla:

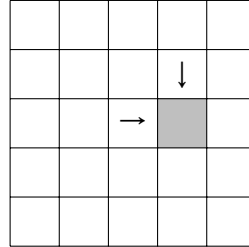
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

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:

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

Ratkaisun aikavaativuus on $O(n^2)$, koska jokaisessa ruudussa $f(y, x)$ saadaan laskettua vakioajassa viereisten ruutujen arvoista.

7.4 Repunpakkaus

Repunpakkaus on klassinen ongelma, jossa annettuna on n tavaraa, joiden painot ovat p_1, p_2, \dots, p_n ja arvot ovat a_1, a_2, \dots, a_n . Tehtävänä on valita reppuun pakattavat tavarat niin, että tavaroiden painojen summa on enintään x ja tavaroiden 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: Merkitään $f(k, u)$ suurinta mahdollista tavaroiden yhteisarvoa, kun reppuun pakataan jokin osajoukko tavaroista $1 \dots k$, jossa tavaroiden yhteispaino on u . Ratkaisu tehtävään on suurin arvo $f(n, u)$, 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 kohdassa k oleva tavara joko otetaan tai ei oteta mukaan ratkaisuun. Pohjatapauksina on $f(0, 0) = 0$ ja $f(0, u) = -\infty$, kun $u \neq 0$. Tämän ratkaisun aikavaativuus on $O(nx)$.

Esimerkin tilanteessa optimiratkaisu on $f(4, 11) = 6$, joka muodostuu seuraavan ketjun kautta:

$$f(4, 11) = f(3, 6) + 3 = f(2, 6) + 3 = f(1, 0) + 3 + 3 = f(0, 0) + 3 + 3 = 6.$$

Ratkaisu 2

Minimointi: Merkitään $f(k, u)$ pienintä mahdollista tavaroiden yhteispainoa, kun reppuun pakataan jokin osajoukko tavaroista $1 \dots k$, jossa tavaroiden yhteisarvo on u . Ratkaisu tehtävään on suurin arvo u , jolle pätee $0 \leq u \leq s$ ja $f(n, u) \leq x$, missä $s = \sum_{i=1}^n a_i$. Rekursiivinen kaava funktion laskemiseksi on

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

ratkaisua 1 vastaavasti. Pohjatapauksina on $f(0, 0) = 0$ ja $f(0, u) = \infty$, kun $u \neq 0$. Tämän ratkaisun aikavaativuus on $O(ns)$.

Esimerkin tilanteessa optimiratkaisu on $f(4, 6) = 11$, joka muodostuu seuraavan ketjun kautta:

$$f(4, 6) = f(3, 3) + 5 = f(2, 3) + 5 = f(1, 0) + 6 + 5 = f(0, 0) + 6 + 5 = 11.$$

Kiinnostava seikka on, että eri asiat syötteessä vaikuttavat ratkaisuiden tehokkuuteen. Ratkaisussa 1 tavaroiden painot vaikuttavat tehokkuuteen mutta arvoilla ei ole merkitystä. Ratkaisussa 2 puolestaan tavaroiden arvot vaikuttavat tehokkuuteen mutta painoilla ei ole merkitystä.

7.5 Editointietäisyys

Editointietäisyys eli **Levenšteinin etäisyys** 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.

Oletetaan, että annettuna on merkkijonot x (pituus n merkkiä) ja y (pituus m merkkiä), ja haluamme laskea niiden editointietäisyyden. Tämä onnistuu tehokkaasti dynaamisella ohjelmoinnilla ajassa $O(nm)$. Merkitään funktiolla $f(a, b)$ editointietäisyyttä x :n a ensimmäisen merkin sekä y :n b :n ensimmäisen merkin välillä. Tätä funktiota käyttäen merkkijonojen x ja y editointietäisyys on $f(n, m)$, ja funktio kertoo myös 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 merkiksi ($c = 1$)

Seuraava taulukko sisältää funktion f arvot esimerkin tapauksessa:

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

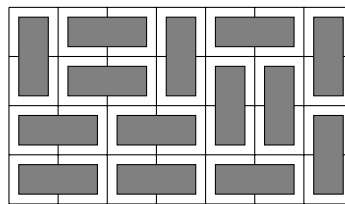
Taulukon oikean alanurkan ruutu kertoo, että merkkijonojen TALO ja PALLO editointietäisyys on 2. Taulukosta pystyy myös lukemaan, miten pienimmän editointietäisyyden voi saavuttaa. Tässä tapauksessa polku on seuraava:

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

7.6 Laatoitukset

Joskus dynaamisen ohjelmoinnin tila on monimutkaisempi kuin kiinteä yhdistelmä lukuja. Tarkastelemme lopuksi tehtävää, jossa laskettavana on, monellako tavalla kokoa 1×2 ja 2×1 olevilla laatoilla voi täyttää $n \times m$ -kokoisen ruudukon. Esimerkiksi ruudukolle kokoa 4×7 yksi mahdollinen ratkaisu on



ja ratkaisujen yhteismäärä on 781.

Tehtävän voi ratkaista dynaamisella ohjelmoinnilla käymällä ruudukkoa läpi rivi riviltä. Jokainen ratkaisun rivi pelkistyy merkkijonoksi, jossa on m merkkiä joukosta $\{\sqcup, \sqcup, \sqcup, \sqcup\}$. Esimerkiksi yllä olevassa ratkaisussa on 4 riviä, jotka vastaavat merkkijonoja

- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$,
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$,
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ ja
- $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$.

Tehtävään sopiva rekursiivinen funktio on $f(k, x)$, joka laskee, montako tapaa on muodostaa ratkaisu ruudukon riveille $1 \dots k$ niin, että riviä k vastaa merkkijono x . Dynaaminen ohjelmointi on mahdollista, koska jokaisen rivin sisältöä rajoittaa vain edellisen rivin sisältö.

Riveistä muodostuva ratkaisu on kelvollinen, jos rivillä 1 ei ole merkkiä \sqcup , rivillä n ei ole merkkiä \sqcup ja kaikki peräkkäiset rivit ovat *yhteensopivat*. Esimerkiksi rivit $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ ja $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ ovat yhteensopivat, kun taas rivit $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ ja $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$ eivät ole yhteensopivat.

Koska rivillä on m merkkiä ja jokaiselle merkille on 4 vaihtoehtoa, erilaisia rivejä on korkeintaan 4^m . Niinpä ratkaisun aikavaativuus on $O(n4^{2m})$, koska joka rivillä käydään läpi $O(4^m)$ vaihtoehtoa rivin sisällölle ja jokaista vaihtoehtoa kohden on $O(4^m)$ vaihtoehtoa edellisen rivin sisällölle. Käytännössä ruudukko kannattaa kääntää niin päin, että pienempi sivun pituus on $m:n$ roolissa, koska $m:n$ suuruus on ratkaiseva ajankäytön kannalta.

Ratkaisua on mahdollista tehostaa parantamalla rivien esitystapaa merkkijonoina. Osoittautuu, että ainoa seuraavalla rivillä tarvittava tieto on, missä

kohdissa riviltä lähtee laattoja alaspäin. Niinpä rivin voikin tallentaa käyttämällä vain merkkejä \sqcap ja \sqcup , missä \square kokoaa yhteen vanhat merkit \sqcup , \square ja \sqcap . Tällöin erilaisia rivejä on vain 2^m ja aikavaativuudeksi tulee $O(n2^{2m})$.

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

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

Tämä kaava on sinänsä hyvin tehokas, koska se laskee laatoitusten määrän ajassa $O(nm)$, mutta käytännön ongelma kaavan käyttämisessä on, kuinka tallentaa välitulokset 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 soveltuu sellaisten algoritmien analyysiin, 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 Kahden osoittimen tekniikka

Kahden osoittimen tekniikka on taulukon käsittelyssä käytettävä menetelmä, 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.

Alitaulukon summa

Annettuna on taulukko, jossa on n positiivista kokonaislukua t_1, t_2, \dots, t_n . Tehtävänä on selvittää, onko taulukossa alitaulukkoa, jossa lukujen summa on x . Esimerkiksi taulukossa

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

tällainen alitaulukko on seuraava:

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

Osoittautuu, että tämän tehtävän voi ratkaista ajassa $O(n)$ kahden osoittimen tekniikalla. 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 $O(n)$.

Kahden luvun summa

Annettuna on taulukko, jossa on n kokonaislukua, sekä kokonaisluku x . Tehtävänä on etsiä taulukosta kaksi lukua, joiden summa on x , tai todeta, että tämä ei ole mahdollista. Tämä ongelma tunnetaan nimellä **2SUM** ja se ratkeaa tehokkaasti kahden osoittimen tekniikalla.

Taulukon luvut järjestetään ensin pienimmästä suurimpaan, minkä jälkeen taulukkoa aletaan käydä läpi kahdella osoittimella, jotka lähtevät liikkelle taulukon molemmista päistä. Vasen osoitin aloittaa taulukon alusta ja liikkuu joka vaiheessa askeleen eteenpäin. Oikea osoitin taas aloittaa taulukon lopusta ja peruuttaa vuorollaan taaksepäin, kunnes osoittimen määrittämän välin lukujen summa on enintään x . Jos summa on tarkalleen x , ratkaisu on löytynyt.

Tarkastellaan 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 $O(\log n)$.

Hieman vaikeampi ongelma on **3SUM**, jossa taulukosta tulee etsiä kolme lukua, joiden summa on x . Tämä ongelma on mahdollista ratkaista ajassa $O(n^2)$. Keksitkö, miten se tapahtuu?

8.2 Lähin pienempi edeltäjä

Tasoitettun analyysin avulla arvioidaan usein tietorakenteeseen kohdistuvien operaatioiden määrää. Algoritmin operaatiot voivat jakautua epätasaisesti niin, että useimmat operaatiot tehdään tietyssä algoritmin vaiheessa, mutta operaatioiden yhteismäärä on kuitenkin rajoitettu.

Tarkastellaan esimerkkinä ongelmaa, jossa tehtävänä on etsiä kullekin taulukon alkioille **lähin pienempi edeltäjä** eli lähinnä oleva pienempi alkio taulukon alkuosassa. On mahdollista, ettei tällaista alkioita ole olemassa, jolloin algoritmin tulee huomata asia. Osoittautuu, että tehtävä on mahdollista ratkaista tehokkaasti ajassa $O(n)$ sopivan tietorakenteen avulla.

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 algoritmin kokonaisaikaavaativuus on $O(n)$.

8.3 Liukuvan ikkunan minimi

Liukuva ikkuna on taulukon halki kulkeva aktiivinen alitaulukko, jonka pituus on vakio. Jokaisessa liukuvan ikkunan sijainnissa halutaan tyypillisesti laskea jotain tietoa ikkunan alueelle osuvista alkioista. Kiinnostava tehtävä on pitää yllä **liukuvan ikkunan minimiä**. Tämä tarkoittaa, että jokaisessa liukuvan ikkunan sijainnissa tulee ilmoittaa pienin alkio ikkunan alueella.


Liukuvan ikkunan minimiä voi laskea lähes samalla tavalla kuin lähimmät pienimmät edeltäjät. 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 minimiä 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



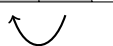
Kun ikkuna siirtyy eteenpäin, mukaan tulee luku 3, joka on pienempi kuin luvut 5 ja 4 ketjun alussa. Niinpä luvut 5 ja 4 poistuvat ketjusta ja luku 3 siirtyy sen alkuun. Pienin luku on edelleen 1.

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



Ikkuna siirtyy taas eteenpäin, minkä 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



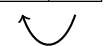
Seuraavaksi ikkunaan tuleva luku 1 on pienempi kuin kaikki ketjussa olevat luvut. Tämän seurauksena koko ketju tyhjäytyy ja siihen jää vain luku 1:

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 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.

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 vähentämällä 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 $27 - 8 = 19$.

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.

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-pituisen 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.

9.2 Binääri-indeksipuu

Binääri-indeksipuu eli **Fenwick-puu** 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.

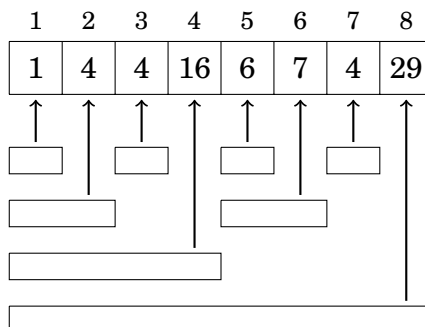
Rakenne

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.

Esimerkiksi taulukkoa

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

vastaava binääri-indeksipuu on seuraava:

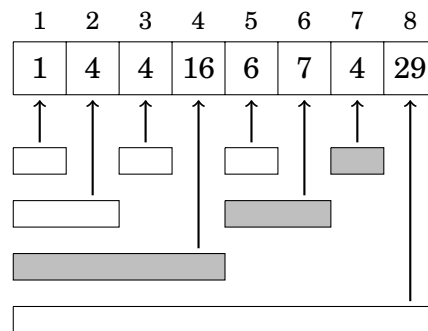


Esimerkiksi binääri-indeksipuun kohdassa 6 on luku 7, koska välin $[5,6]$ lukujen summa on $6 + 1 = 7$.

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:

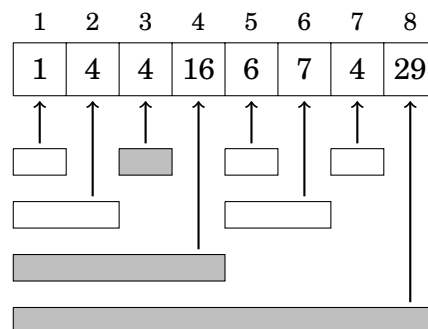


Välin $[1, 7]$ summa on siis $16 + 7 + 4 = 27$. Binääri-indeksipuun rakenteen ansiosta jokainen summaan kuuluva väli on eripituinen. Niinpä summa muodostuu aina $O(\log n)$ välin summasta.

Summataulukon tavoin binääri-indeksipuusta voi laskea tehokkaasti minikä 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 kaikki välit, joihin muutos vaikuttaa, ovat eripituisia, joten muutos kohdistuu $O(\log n)$ kohtaan binääri-indeksipuussa.

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 lisää kasvattaa taulukon kohtaa k arvolla x :

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

Kummankin yllä olevan funktion aikavaativuus on $O(\log n)$, koska funktiot muuttavat $O(\log n)$ kohtaa binääri-indeksipuussa ja uuteen kohtaan siirtyminen vie aikaa $O(1)$ bittioperaation avulla.

9.3 Segmenttipuu

Segmenttipuu on tietorakenne, jonka operaatiot ovat taulukon välin $[a, b]$ välilyksely 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.

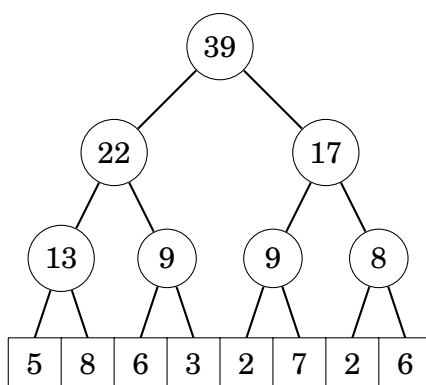
Rakenne

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.

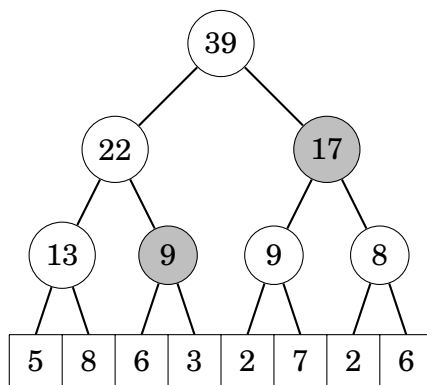
Välikysely

Segmenttipuussa vastaus välikyselyyn lasketaan väliin kuuluvista solmuista, jotka ovat mahdollisimman korkealla puussa. Jokainen solmu antaa vastauksen väliin kuuluvalla osavälillä, ja vastaus kyselyyn selviää yhdistämällä segmenttipuusta saadut osavälit koskeva 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:



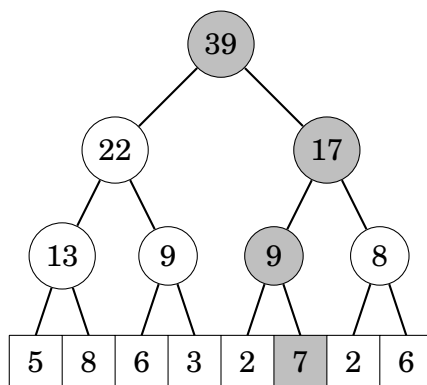
Taulukon välin summaksi tulee osasummista $9 + 17 = 26$.

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.

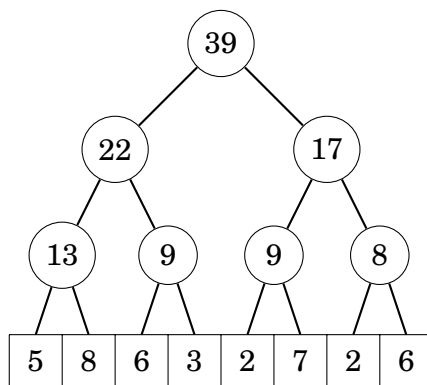


Polku segmenttipuun pohjalta huipulle muodostuu aina $O(\log n)$ solmusta, joten taulukon arvon muuttuminen vaikuttaa $O(\log n)$ solmuun puussa.

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 kohdassa k olevalle solmulle pätee, että

- ylempi solmu on kohdassa $\lfloor k/2 \rfloor$,
- vasen alempi solmu on kohdassa $2k$ ja
- oikea alempi solmu on kohdassa $2k + 1$.

Huomaa, että tämän seurauksena solmun kohta on parillinen, jos se on vasemmalla ylemmästä solmusta katsoen, ja pariton, jos se on 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];
    }
}

```

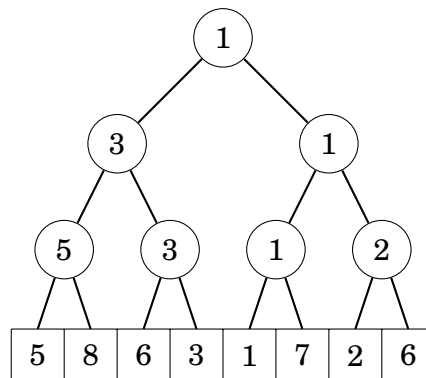
Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasummat puun huipulle asti. Taulukon p indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k + 1$.

Molemmat segmenttipuun operaatiot toimivat ajassa $O(\log n)$, koska n 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 $[b + 1, c]$ tuloksista pystyy laskemaan tehokkaasti välin $[a, c]$ tuloksen. Tällaisia kyselyitä ovat esimerkiksi minimi ja maksimi, suurin yhteinen tekijä sekä bittiopeeraatiot and, or ja xor.

Esimerkiksi seuraavan segmenttipuun avulla voi laskea taulukon välien minimejä:

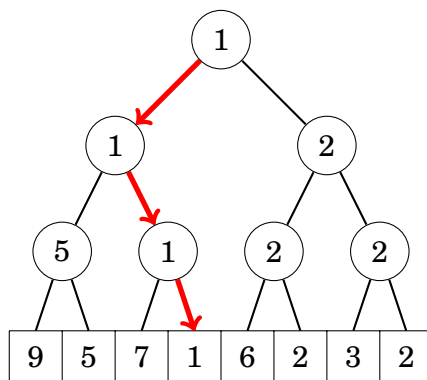


Tässä segmenttipuussa jokainen puun solmu kertoo, mikä on pienin luku sen alapuolella olevassa taulukon osassa. Segmenttipuun ylin luku on pienin luku koko taulukon alueella. Puun toteutus on samanlainen kuin summan laskemisessa, mutta joka kohdassa pitää laskea summan sijasta lukujen minimi.

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 kulkeamalla puussa huipulta alaspäin:



9.4 Lisäteknikoita

Indeksien pakkaus

Taulukon päälle rakennettujen tietorakenteiden rajoituksena on, että 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 alkia, 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:

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

Välin muuttaminen

Tähän asti olemme toteuttaneet tietorakenteita, joissa voi tehdä tehokkaasti välilykselyitä ja muuttaa yksittäisiä taulukon arvoja. Tarkastellaan lopuksi käänteistä tilannetta, jossa pitääkin muuttaa välejä ja kysellä yksittäisiä arvoja. Keskitymme operaatioon, 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.


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

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

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

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

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

10.2 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

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

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 bittiopeeraatioilla.

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
```

Kun joukko on tallennettu bittiesityksenä, niin joukko-operaatiot voi toteuttaa tehokkaasti bittiopeeraatioiden avulla:

- $a \& b$ 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
```

Osajoukkojen läpikäynti

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

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

Seuraava koodi käy läpi osajoukot, joissa on k alkia:

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

Seuraava koodi käy läpi joukon x osajoukot:

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

Esimerkiksi jos x esittää joukkoa $\{2, 5, 7\}$, niin koodi käy läpi osajoukot \emptyset , $\{2\}$, $\{5\}$, $\{7\}$, $\{2, 5\}$, $\{2, 7\}$, $\{5, 7\}$ ja $\{2, 5, 7\}$.

10.4 Dynaaminen ohjelmointi

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 tietyillä n :n arvoilla permutaatioita ei ehdi käydä läpi mutta osajoukot ehtii käydä läpi.

Lasketaan esimerkkinä, monessako joukon $\{0, 1, \dots, n-1\}$ permutaatiossa ei ole missään kohdassa kahta peräkkäistä lukua. Esimerkiksi tapauksessa $n = 4$ ratkaisuja on kaksi:

- (1, 3, 0, 2)
- (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];
}
```

Osajoukkojen summat

Tarkastellaan sitten joukon $\{0, 1, \dots, n-1\}$ osajoukkoja, kun jokaista osajoukkoa x vastaa arvo $c(x)$. Tehtävänä 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 $s(110) = c(000) + c(010) + c(100) + c(110) = 5$.

Tehtävä on mahdollista ratkaista ajassa $O(2^n n)$ laskemalla arvoja funktiolle $f(x, k)$: mikä on lukujen $c(y)$ summa, missä x :stä saa y :n muuttamalla mil-lä tahansa 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 nolaksi.

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

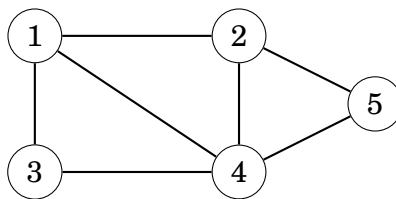
Monen ohjelmointitehtävän voi ratkaista tulkitsemalla tehtävän verkko-ongelmana ja käyttämällä sopivaa verkkoalgoritmia. Tyypillinen esimerkki verkosta on tieverkosto, jonka rakenne muistuttaa luonnostaan verkkoa. Joskus taas verkko kätkeytyy syvemmälle ongelmaan ja sitä voi olla vaikeaa huomata.

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 muodostuu **solmuista** ja niiden välisistä **kaarista**. 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:

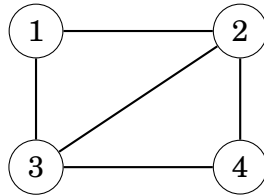


Polku on solmusta a solmuun b johtava reitti, joka kulkee verkon kaaria pitkin. Polun **pituus** on kaarten määrä polulla. Esimerkiksi yllä olevassa verkossa polkuja solmusta 1 solmuun 5 ovat:

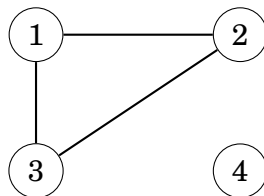
- $1 \rightarrow 2 \rightarrow 5$ (pituus 2)
- $1 \rightarrow 4 \rightarrow 5$ (pituus 2)
- $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ (pituus 3)
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ (pituus 3)
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5$ (pituus 4)

Yhtenäisyys

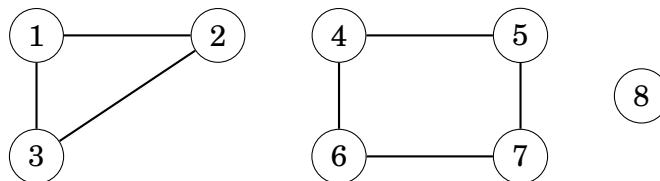
Verkko on **yhtenäinen**, jos siinä on polku mistä tahansa solmusta mihin tahansa solmuun. Esimerkiksi seuraava verkko on yhtenäinen:



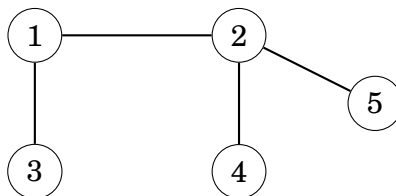
Seuraava verkko taas ei ole yhtenäinen, koska solmusta 4 ei pääse muihin verkon solmuihin.



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



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



Kaarten suunnat

Verkko on **suunnattu**, jos verkon kaaria pystyy kulkemaan vain niiden merkittyyn suuntaan. Esimerkiksi seuraava verkko on suunnattu:

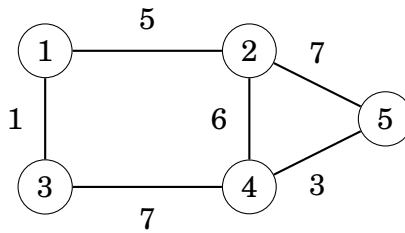


Yllä olevassa verkossa on polku solmusta 3 solmuun 5, joka kulkee kaaria $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$. Sen sijaan verkossa ei ole polkua solmusta 5 solmuun 3.

Sykli 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 **sykkitön**.

Kaarten painot

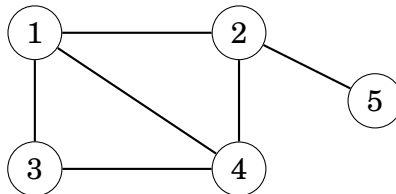
Painotetussa verkossa jokaiseen kaareen liittyy paino. Usein painot kuvaavat kaarien pituuksia. Esimerkiksi seuraava verkko on painotettu:



Nyt polun pituus on polulla olevien kaarten painojen summa. Esimerkiksi yllä olevassa verkossa 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 on lyhin polku solmusta 1 solmuun 5.

Naapurit ja asteet

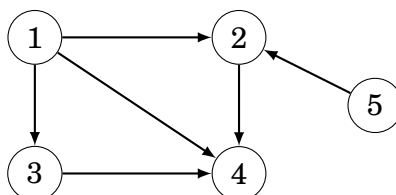
Kaksi solmua ovat **naapureita**, jos ne ovat verkossa **vierekkäin** eli niiden välillä on kaari. Solmun **aste** on sen naapurien määrä. Esimerkiksi seuraavassa verkossa solmun 2 naapurit ovat 1, 4 ja 5, joten sen aste on 3.



Verkon solmujen asteiden summa on aina $2m$, 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**, jos jokaisen solmun aste on vakio d . Verkko on **täydellinen**, jos jokaisen solmun aste on $n-1$ eli verkossa on kaikki mahdolliset kaaret solmujen välillä.

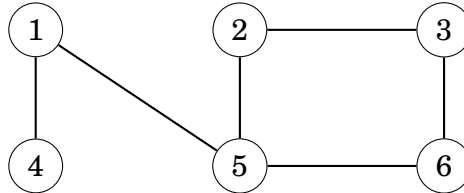
Suunnatussa verkossa **lähtöaste** on solmusta lähtevien kaarten määrä ja **tuloaste** on solmuun tulevien kaarten määrä. Esimerkiksi seuraavassa verkossa solmun 2 lähtöaste on 1 ja tuloaste on 2.



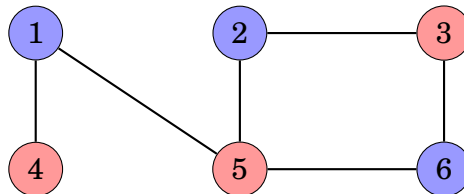
Väritykset

Verkon **värityksessä** jokaiselle solmulle valitaan tietty väri niin, että millään kahdella vierekkäisellä solmulla ei ole samaa väriä.

Verkko on **kaksijakoinen**, jos on mahdollista värittää se kahdella värillä. Osoittautuu, että verkko on kaksijakoinen tarkalleen silloin, kun siinä ei ole sykliä, johon kuuluu pariton määrä solmuja. Esimerkiksi verkko

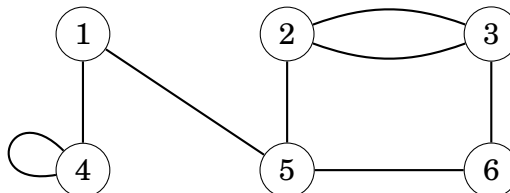


on kaksijakoinen, koska sen voi värittää seuraavasti:



Yksinkertaisuus

Verkko on **yksinkertainen**, 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.

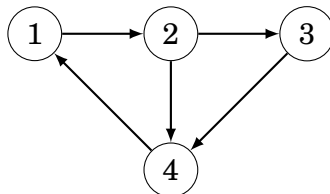
Vieruslistaesitys

Tavallisin tapa pitää verkkoa muistissa on luoda jokaisesta solmusta **vieruslista**, 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];
```

Taulukossa solmun s vieruslista on kohdassa $v[s]$. Vakio N on valittu niin suureksi, että kaikki vieruslistat mahtuvat taulukkoon. Esimerkiksi verkon



voi tallentaa seuraavasti:

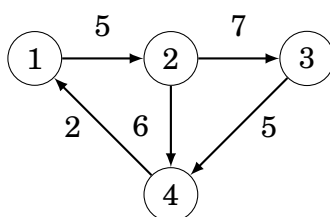
```
v[1].push_back(2);  
v[2].push_back(3);  
v[2].push_back(4);  
v[3].push_back(4);  
v[4].push_back(1);
```

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 kenttä on kaaren kohdesolmu ja toinen kenttä 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 :

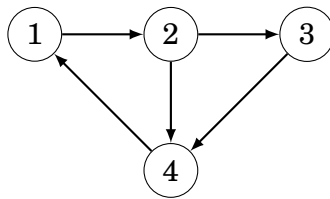
```
for (auto u : v[s]) {
    // käsittele solmu u
}
```

Vierusmatriisiesitys

Vierusmatriisi 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];
```

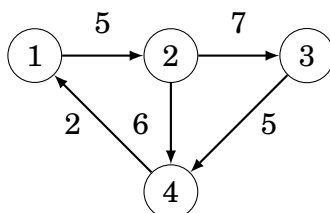
jossa arvo $v[a][b]$ ilmaisee, onko kaari solmusta a solmuun b mukana verkossa. Jos kaari on mukana verkossa, niin $v[a][b] = 1$, ja muussa tapauksessa $v[a][b] = 0$. Nyt esimerkiksi verkkoa



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

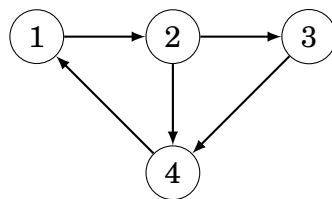
Kaarilistaesitys

Kaarilista 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



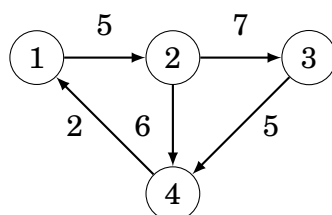
voi tallentaa seuraavasti:

```
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;
```

Nyt listalla on pareja, joiden ensimmäinen jäsen sisältää parina kaaren alku- ja loppusolmun, ja toinen jäsen on kaaren paino. Esimerkiksi 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 alkusolmista ja käyvät läpi kaikki solmut, joihin alkusolmista pääsee. Algoritmien erona on, missä järjestyksessä ne kulkevat verkossa.

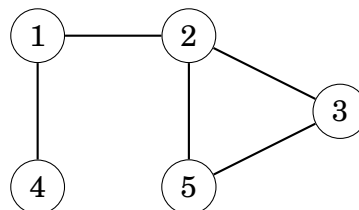
12.1 Syvyyshaku

Syvyyshaku on suoraviivainen menetelmä verkon läpikäyntiin. Algoritmi lähtee liikkeelle tietyistä verkon solmista ja etenee siitä kaikkiin solmuihin, jotka ovat saavutettavissa kaaria kulkemalla.

Syvyyshaku etenee verkossa syvyys-suuntaisesti 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.

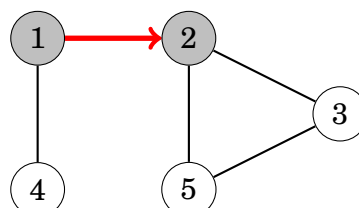
Esimerkki

Tarkastellaan syvyysshaun toimintaa seuraavassa verkossa:

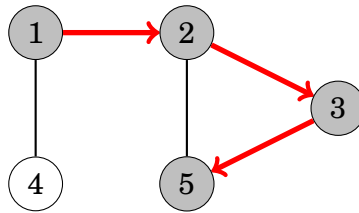


Syvyyshaku voi lähteä liikkeelle mistä tahansa solmista, mutta oletetaan nyt, että haku lähtee liikkeelle solmista 1.

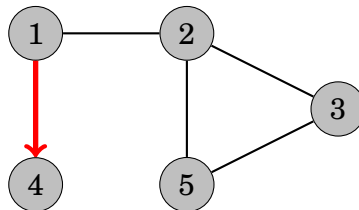
Solmun 1 naapurit ovat solmut 2 ja 4, joista haku etenee ensin solmuun 2:



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.

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

Toteutus

Syvyysshaku on yleensä mukavinta toteuttaa rekursiolla. Seuraava funktio haku suorittaa syvyysshaun 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 , kohtaan $z[s]$ merkitään 1. Funktion toteutus on seuraavanlainen:

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

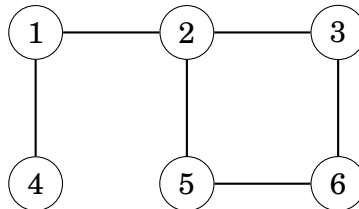
12.2 Leveyshaku

Leveyshaku käy solmut läpi järjestyksessä sen mukaan, kuinka kaukana ne ovat alkusolmusta. Niinpä leveyshaun avulla pystyy laskemaan etäisyyden aloitussolmusta kaikkiin muihin solmuihin. Leveyshaku on kuitenkin vaikeampi toteuttaa kuin syvyyshaku.

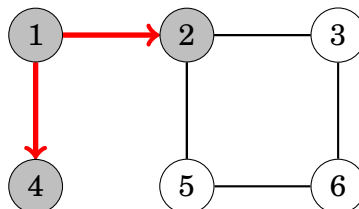
Leveyshakua voi ajatella niin, että se käy solmuja läpi kerros kerrallaan. Ensin haku käy läpi solmut, joihin pääsee yhdellä kaarella alkusolmusta. Tämän jälkeen vuorossa ovat solmut, joihin pääsee kahdella kaarella alkusolmusta jne. Sama jatkuu, kunnes uusia käsiteltäviä solmuja ei enää ole.

Esimerkki

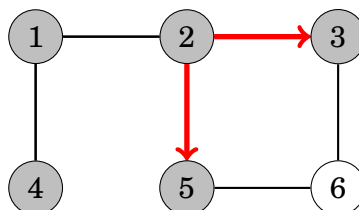
Tarkastellaan leveyshaun toimintaa seuraavassa verkossa:



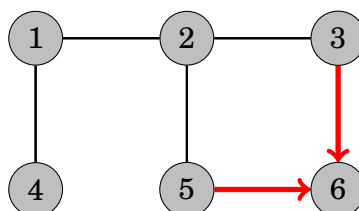
Oletetaan jälleen, että haku alkaa solmusta 1. Haku etenee ensin kaikkiin solmuihin, joihin pääsee alkusolmusta:



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

Leveyshaku on syvyyshakua hankalampi toteuttaa, 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);
    }
}
```

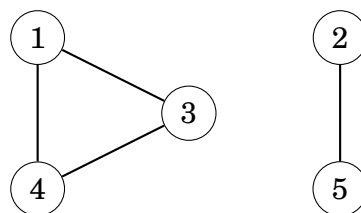
12.3 Sovelluksia

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

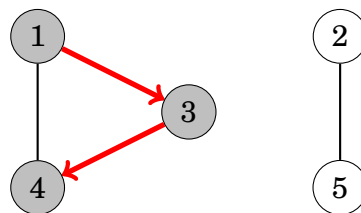
Yhtenäisyyden tarkastaminen

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



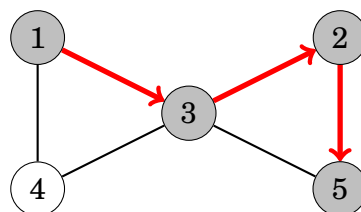
solmusta 1 alkava syvyysshaku löytää seuraavat solmut:



Koska syvyysshaku 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 $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

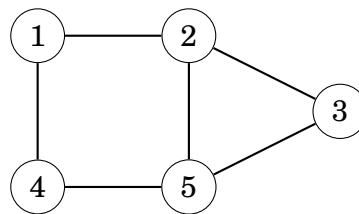
Syklin olemassaolon voi myös päätellä laskemalla, montako solmua ja kaarta komponentissa on. Jos komponentissa on c solmua ja siinä ei ole sykliä, niin siinä on oltava tarkalleen $c - 1$ kaarta. Jos kaaria on c tai enemmän, niin komponentissa on varmasti sykli.

Kaksijakoisuuden tarkastaminen

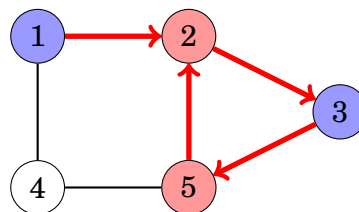
Verkko on kaksijakoinen, jos sen solmut voi värittää kahdella värillä niin, että kahta samanväristä solmua ei ole vierekkäin. On yllättävän helppoa selvittää verkon läpikäynnin avulla, onko verkko kaksijakoinen.

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-vaikea ongelma.

Luku 13

Lyhimmät polut

Lyhimmän polun etsiminen alkusolmusta loppusolmuun on keskeinen verkko-ongelma, joka esiintyy usein käytännön tilanteissa. Esimerkiksi tieverkostossa luonteva 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.

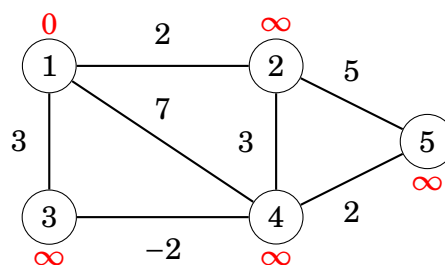
13.1 Bellman–Fordin algoritmi

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.

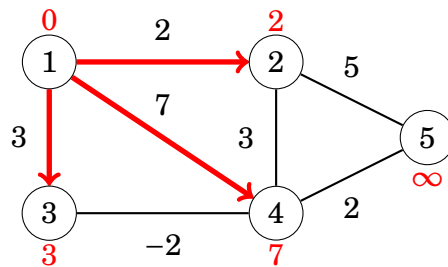
Esimerkki

Tarkastellaan Bellman–Fordin algoritmin toimintaa seuraavassa verkossa:

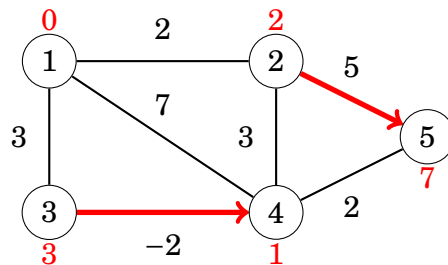


Verkon jokaiseen solmun viereen on merkitty etäisyysarvio. Alussa alkusolmun etäisyysarvio on 0 ja muiden solmujen etäisyysarvio on ääretön.

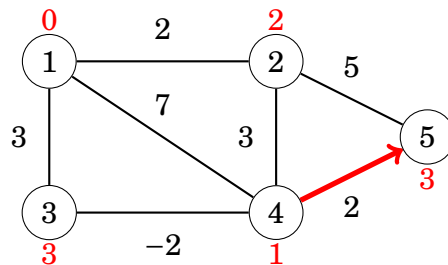
Algoritmi etsii verkosta kaaria, jotka parantavat etäisyysarvioita. Aluksi kaikki solmusta 0 lähtevät kaaret parantavat arvioita:



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

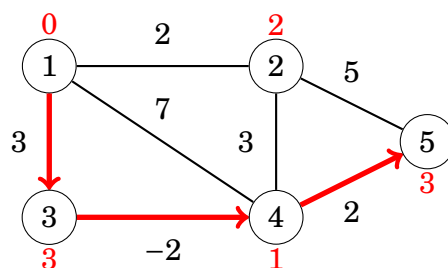


Lopuksi tulee vielä yksi parannus:



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.

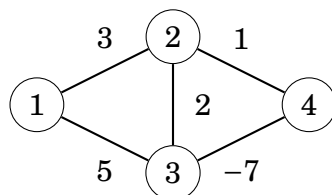
```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (int a = 1; a <= n; a++) {
        for (auto b : v[a]) {
            e[b.first] = min(e[b.first], e[a] + b.second);
        }
    }
}
```

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 etäisyysarvot ovat lopulliset $n - 1$ kierroksen jälkeen, koska jokaisessa lyhimässä polussa on enintään $n - 1$ kaarta.

Käytännössä kaikki lopulliset etäisyysarvot 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 alkusolmun 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.

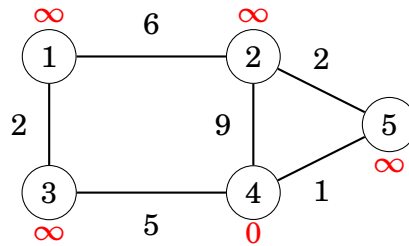
13.2 Dijkstran algoritmi

Dijkstran algoritmi etsii Bellman–Fordin algoritmin tavoin lyhimmat 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.

Esimerkki

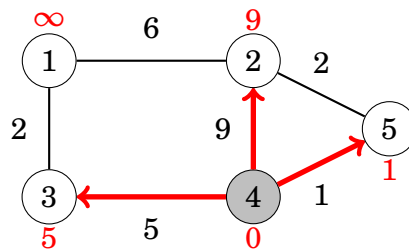
Tarkastellaan Dijkstran algoritmin toimintaa seuraavassa verkossa:



Bellman–Fordin algoritmin tavoin alkusolmun etäisyysarvio on 0 ja kaikissa muissa solmuissa etäisyysarvio on aluksi ääretön.

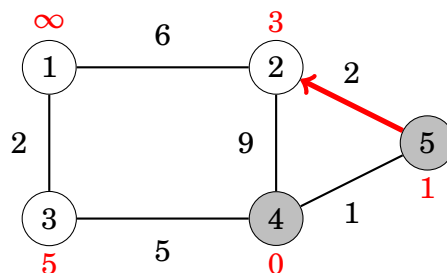
Dijkstran algoritmi ottaa joka askeleella käsittelyyn sellaisen solmun, jota ei ole vielä käsitelty ja jonka etäisyysarvio on mahdollisimman pieni. Alussa tällainen solmu on solmu 4, jonka etäisyysarvio on 0.

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

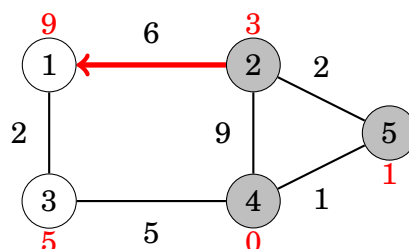


Solmun 4 käsittely paransi etäisyysarvioita solmuihin 2, 3 ja 5, joiden uudet etäisyydet ovat nyt 9, 5 ja 1.

Seuraavaksi käsittelyyn tulee solmu 5, jonka etäisyys on 1:

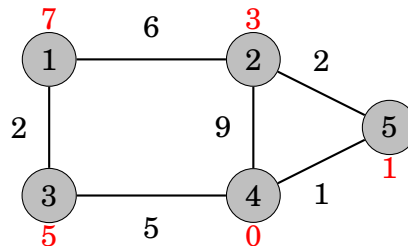


Tämän jälkeen vuorossa on solmu 2:



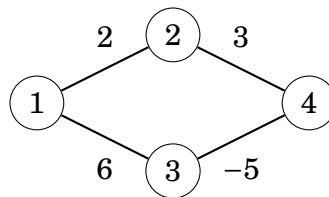
Dijkstran algoritmissa on hienoutena, että aina kun solmu tulee käsittelyyn, sen etäisyysarvio on siitä lähtien lopullinen. Esimerkiksi tässä vaiheessa etäisyydet 0, 1 ja 3 ovat lopulliset etäisyydet solmuihin 4, 5 ja 2.

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 algoritmissa 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 etäisyyssarvioita 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 $O(n+m \log m)$, koska algoritmi käy läpi kaikki verkon solmut ja lisää jokaista kaarta kohden korkeintaan yhden etäisyysarvion prioriteettijonoon.

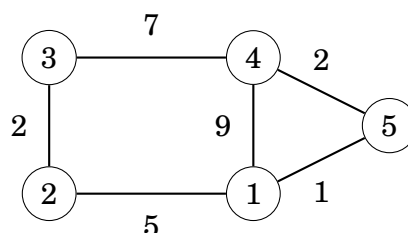
13.3 Floyd–Warshallin algoritmi

Floyd–Warshallin algoritmi on toisenlainen lähestymistapa lyhimpien polkujen etsintään. Toisin kuin muut tämän luvun algoritmit, se etsii yhdellä kertaa lyhimmät polut kaikkien verkon solmujen 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.

Esimerkki

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

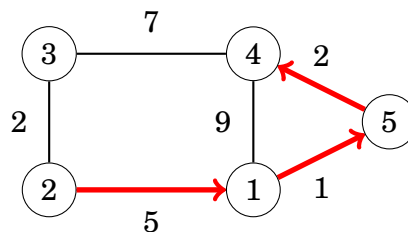
Kolmannella kierroksella solmu 3 saa toimia välisolmuna, jolloin syntyy uusi polku solmuparin 2 ja 4 välille:

	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 vierusmatriisin 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.

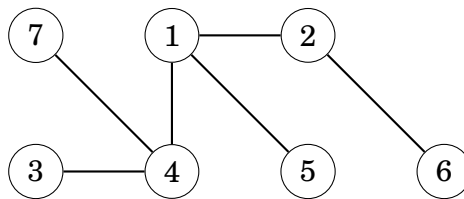
Koska algoritmin toteutus on yksinkertainen, se voi olla hyvä valinta jopa silloin, kun haettavana on yksittäinen lyhin polku verkossa. Tämä vaatii kuitenkin, että verkko on pieni ja kuutiollinen aikavaativuus on riittävä.

Luku 14

Puiden käsittely

Puu on yhtenäinen, syklitön verkko, jossa on n solmua ja $n - 1$ kaarta. Jos puusta poistaa yhden kaaren, se ei ole enää yhtenäinen, ja jos puuhun lisää yhden kaaren, se ei ole enää syklitön. Puussa pätee myös aina, että jokaisen kahden puun solmun välillä on yksikäsitteinen polku.

Esimerkiksi seuraavassa puussa on 7 solmua ja 6 kaarta:



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

Jos puu on **juurellinen**, yksi solmuista on puun **juuri**, jonka alapuolelle muut solmut asettuvat. Esimerkiksi jos yllä olevassa puussa valitaan juureksi solmu 1, solmut asettuvat seuraavaan järjestykseen:



Juurellisessa puussa solmun **lapset** ovat sen alemman tason naapurit ja solmun **vanhempi** 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**, 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 ja solmun 1 alipuuhun kuuluvat kaikki puun solmut.

14.1 Puun 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 päätyisi tiettyyn solmuun monesta eri suunnasta.

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

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

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

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

Dynaaminen ohjelmointi

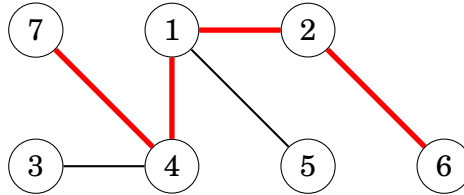
Puun läpikäyntiin voi myös yhdistää dynaamista ohjelmointia ja laskea sen avulla jotakin tietoa puusta. Dynaamisen ohjelmoinnin avulla voi esimerkiksi laskea ajassa $O(n)$ jokaiselle solmulle, montako solmua sen alipuussa on tai kuinka pitkä on pisin solmusta alaspäin jatkuva polku puussa.

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

14.2 Lämpimitta

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



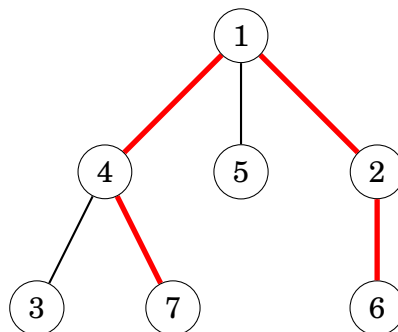
läpimitta on 4, koska polku solmusta 3 solmuun 6 on pituudeltaan 4. Lämpimittava vastaava polku ei ole välttämättä yksikäsitteinen. Esimerkiksi tässä puussa myös polku solmusta 7 solmuun 6 on pituudeltaan 4.

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 jostakin 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:



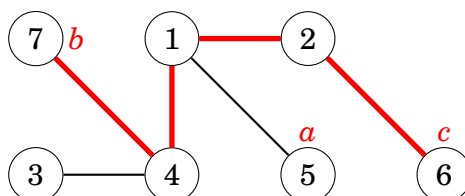
Algoritmi laskee ensin dynaamisella ohjelmoinnilla jokaiselle solmulle, kuinka pitkä on pisin solmu, joka lähtee solmusta alaspäin. Esimerkiksi yllä olevassa puussa pisin polku solmusta 1 alaspäin on pituudeltaan 2 (vaihtoehdot $1 \rightarrow 4 \rightarrow 3$, $1 \rightarrow 4 \rightarrow 7$ ja $1 \rightarrow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

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

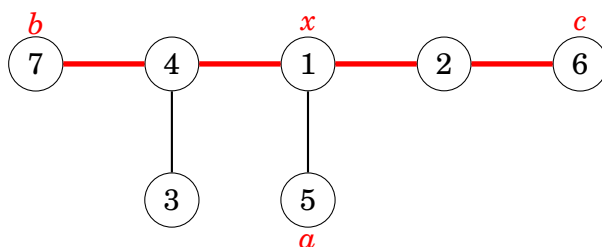
Toinen tehokas tapa laskea puun läpimitta perustuu kahteen syvyyshakuun. Ensin valitaan mikä tahansa solmu a puusta ja etsitään siitä kaukaisin solmu b syvyyshaulla. Tämän jälkeen etsitään b :stä kaukaisin solmu c syvyyshaulla. Puun läpimitta on etäisyys b :n ja c :n välillä.

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:

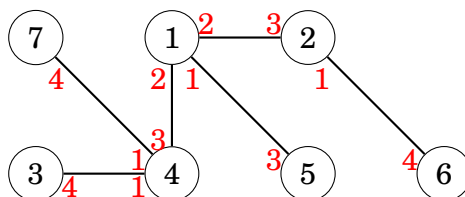


Solmu x on kohta, jossa polku solmusta a liittyy läpimittaa vastaavaan polkuun. Kaukaisin solmu a :sta on solmu b , solmu c tai jokin muu solmu, joka on ainakin yhtä kaukana solmusta x . Niinpä tämä solmu on aina sopiva valinta läpimittaa vastaavan polun toiseksi päätesolmuksi.

14.3 Solmujen etäisyydet

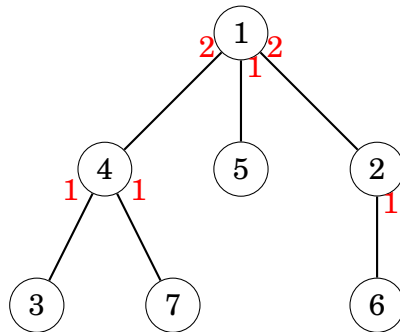
Vaikeampi tehtävä on laskea jokaiselle puun solmulle jokaiseen suuntaan, mikä on suurin etäisyys johonkin kyseisessä suunnassa olevaan solmuun. Osoittautuu, että tämäkin tehtävä ratkeaa ajassa $O(n)$ dynaamisella ohjelmoinnilla.

Esimerkkipuussa etäisyydet ovat:



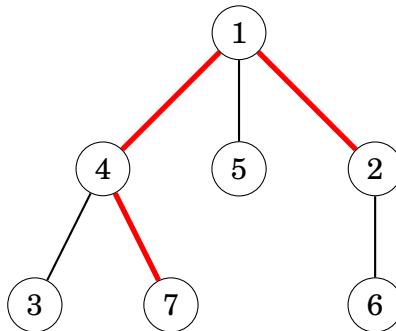
Esimerkiksi solmussa 4 kaukaisin solmu ylöspäin mentäessä on solmu 6, johon etäisyys on 3 käyttäen polkua $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$.

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

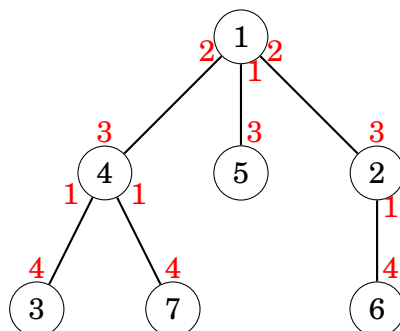


Jäljelle jäävä tehtävä on laskea etäisyydet ylöspäin. Tämä onnistuu tekemällä puuhun toinen läpikäynti, joka pitää mukana tietoa, mikä on suurin etäisyys solmun vanhemmasta johonkin toisessa suunnassa olevaan solmuun.

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



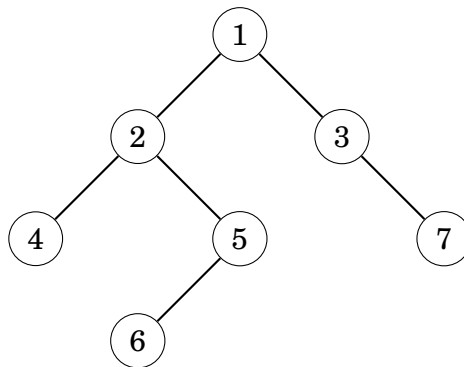
Lopputuloksena on etäisyydet kaikista solmuista kaikkiin suuntiin:



14.4 Binääripuut

Binääripuu 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:



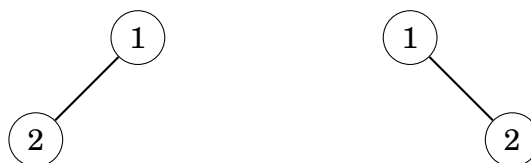
Binääripuun solmuilla on kolme luontevaa järjestystä, jotka syntyvät rekursiivisesta läpikäynnistä:

- **esijärjestys:** juuri, vasen alipuu, oikea alipuu
- **sisäjärjestys:** vasen alipuu, juuri, oikea alipuu
- **jälkijärjestys:** 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



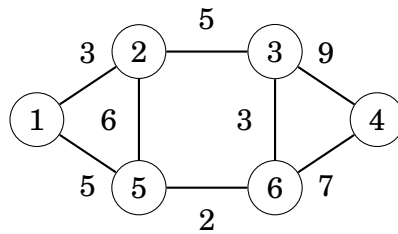
esijärjestys on (1,2) ja jälkijärjestys on (2,1), mutta siitä huolimatta puiden rakenteet eivät ole samat.

Luku 15

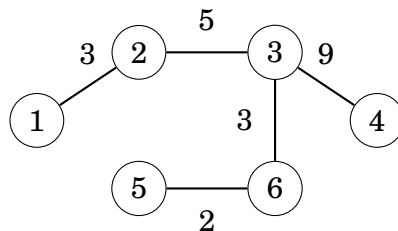
Virittävät puut

Virittävä puu on kokoelma verkon kaaria, joka kytkee kaikki verkon solmut toisiinsa. Kuten puut yleensäkin, virittävä puu on yhtenäinen ja syklitön. Virittävän puun muodostamiseen on yleensä monia tapoja.

Esimerkiksi verkossa

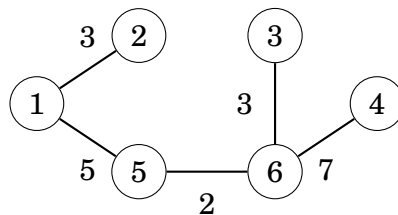


yksi mahdollinen virittävä puu on seuraava:

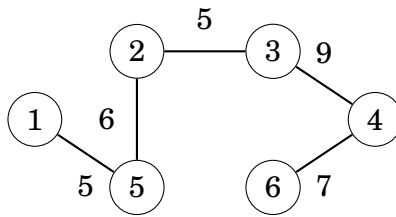


Virittävän puun paino on siihen kuuluvien kaarten painojen summa. Esimerkiksi yllä olevan puun paino on $3 + 5 + 9 + 3 + 2 = 22$.

Pienin virittävä puu 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** on virittävä puu, jonka paino on mahdollisimman suuri. Yllä olevan verkon suurin virittävä puu on painoltaan 32:



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

Tässä luvussa tutustumme algoritmeihin, jotka muodostavat verkon pienimmän tai suurimman virittävän puun. 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 painojärjestyksessä. Keskitymme pienimmän virittävän puun etsimiseen, mutta samoilla algoritmeilla voi muodostaa myös suurimman virittävän puun käsittelemällä kaaret käänteisessä järjestyksessä.

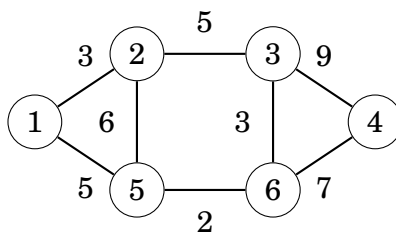
15.1 Kruskalin algoritmi

Kruskalin algoritmi aloittaa pienimmän virittävän puun muodostamisen 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.

Esimerkki

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}:



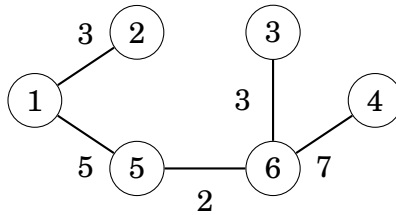
Tämän jälkeen algoritmi lisää puuhun vastaavasti kaaret 1–2, 3–6 ja 1–5:



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:

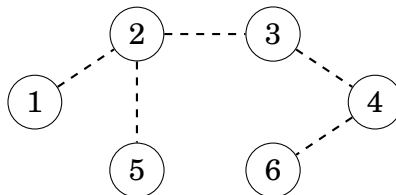


Tämän lisäyksen jälkeen algoritmi päättyy, koska kaikki solmut on kytketty toisiinsa kaarilla ja verkko on yhtenäinen. Tuloksena on verkon pienin virittävä puu, jonka paino on $2 + 3 + 3 + 5 + 7 = 20$.

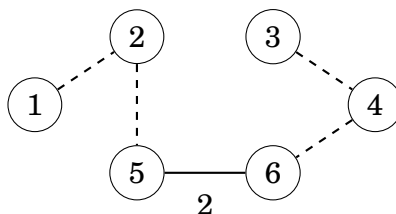
Miksi algoritmi toimii?

On hyvä kysymys, miksi Kruskalin algoritmi toimii aina eli miksi ahne strategia tuottaa varmasti pienimmän mahdollisen virittävän puun.

Voimme perustella algoritmin toimivuuden tekemällä vastaoletuksen, että pienimmässä virittävässä puussa ei olisi verkon keveintä kaarta. Oletetaan esimerkiksi, että äskeisen verkon pienimmässä virittävässä puussa ei olisi 2:n painoista kaarta solmujen 5 ja 6 välillä. Emme tiedä tarkalleen, millainen uusi pienin virittävä puu olisi, mutta siinä täytyy olla kuitenkin joukko kaaria. Oletetaan, että virittävä puu olisi vaikkapa seuraavanlainen:



Ei ole kuitenkaan mahdollista, että yllä oleva virittävä puu olisi todellisuudessa verkon pienin virittävä puu. Tämä johtuu siitä, että voimme poistaa siitä jonkin kaaren ja korvata sen 2:n painoisella kaarella. Tuloksena on virittävä puu, jonka paino on *pienempi*:



Niinpä on aina optimaalinen ratkaisu valita pienimpään virittävään puuhun verkon kevein kaari. Vastaavalla tavalla on mahdollista perustella seuraavaksi keveimmän kaaren valinta jne. Niinpä Kruskalin algoritmi toimii oikein ja tuottaa aina pienimmän virittävän puun.

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-rakenne ratkaisee asian. Se toteuttaa molemmat funktiot ajassa $O(\log n)$, jolloin Kruskalin algoritmin aikavaativuus on vain $O(m \log n)$ kaarilistan järjestämisen jälkeen.

15.2 Union-find-rakenne

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

Rakenne

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

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

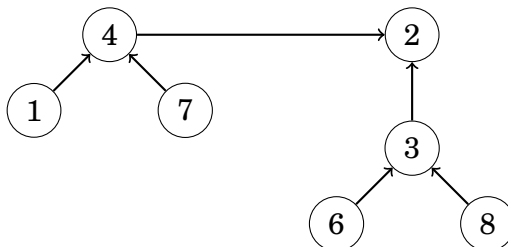


Tässä tapauksessa alkiot 4, 5 ja 2 ovat joukkojen edustajat.

Minkä tahansa alkion edustaja löytyy kulkemalla polku loppuun 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, alkiot ovat samassa joukossa, ja muuten eri joukoissa.

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



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 $O(\log n)$, koska jokainen askel eteenpäin ketjussa kaksinkertaistaa vastaavan joukon koon.

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. Alkion edustaja löytyy käymällä ketju läpi alkioista x alkaen.

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

Funktio `sama` kertoo, ovatko alkiot a ja b samassa joukossa. Tämä onnistuu helposti funktion `id` avulla.

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

Funktio `liita` yhdistää puolestaan alkuioiden a ja b osoittamat joukot yhdeksi joukoksi. Funktio etsii ensin joukkojen edustajat ja yhdistää sitten pienemmän joukon suurempaan.

```
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.

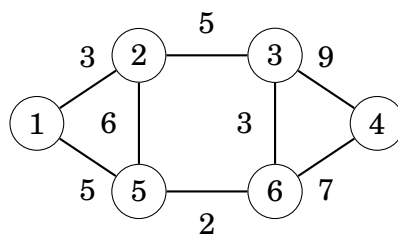
15.3 Primin algoritmi

Primin algoritmi on vaihtoehtoinen menetelmä verkon pienimmän virittävän puun muodostamiseen. Algoritmi aloittaa puun muodostamisen valitusta verkon solmusta ja lisää puuhun aina kaaren, joka on mahdollisimman kevyt ja joka liittyy puuhun uuden solmun. Lopulta kaikki solmut on lisätty puuhun ja pienin virittävä puu on valmis.

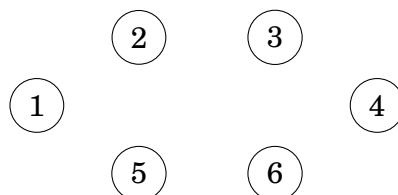
Primin algoritmin toiminta on lähellä Dijkstran algoritmia. Erona on, että Dijkstran algoritmista valitaan kaari, jonka kautta syntyy lyhin polku alkusolmusta uuteen solmuun, mutta Primin algoritmista valitaan vain kevein kaari, joka johtaa uuteen solmuun.

Esimerkki

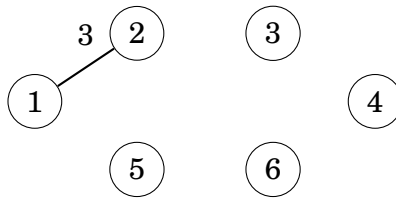
Tarkastellaan Primin algoritmin toimintaa seuraavassa verkossa:



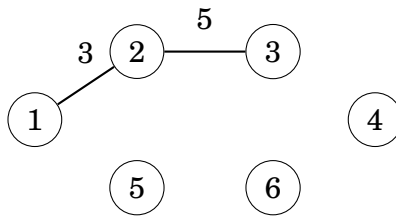
Aluksi solmujen välillä ei ole mitään kaaria:



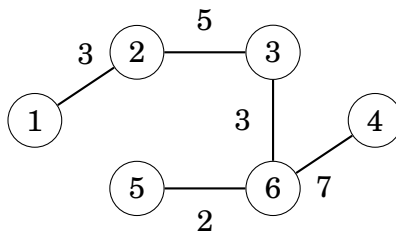
Puun muodostuksen voi aloittaa mistä tahansa solmusta, ja aloitetaan se nyt solmusta 1. Kevein kaari on painoltaan 3 ja se johtaa solmuun 2:



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



Sama jatkuu, kunnes kaikki solmut ovat mukana puussa:



Algoritmin 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 $O(n + m \log m)$ eli sama kuin Dijkstran algoritmilla. 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

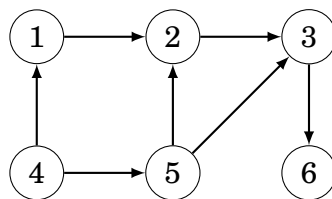
Tässä luvussa tutustumme kahteen suunnattujen verkkojen alaluokkaan:

- **Syklitön verkko:** Tällaisessa verkossa ei ole yhtään sykliä, eli mistään solmusta ei ole polkua takaisin solmuun itseensä.
- **Seuraajaverkko:** Tällaisessa verkossa jokaisen solmun lähtöaste on 1 eli kullakin solmulla on yksikäsitteinen seuraaja.

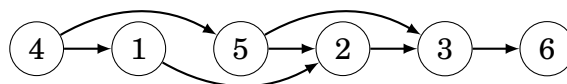
Osoittautuu, että kummassakin tapauksessa verkon käsittely on yleistä verkkoa helpompaa ja voimme käyttää tehokkaita algoritmeja, jotka hyödyntävät oletuksia verkon rakenteesta.

16.1 Topologinen järjestys

Topologinen järjestys on tapa järjestää suunnatun verkon solmut niin, että jos solmusta a pääsee solmuun b , niin a on ennen b :tä järjestyksessä. Esimerkiksi verkon



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



Topologinen järjestys on olemassa aina silloin, kun verkko on syklitön. Jos taas verkossa on sykli, topologista järjestystä ei voi muodostaa, koska mitään syklissä olevaa solmua ei voi valita ennen toista topologiseen järjestykseen.

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 syvyyshauulla, jossa on kolme tilaa:

- tila 0: solmua ei ole käsitelty (valkoinen)
- tila 1: solmun käsittely on alkanut (vaaleanharmaa)
- tila 2: solmu on käsitelty (tummanharmaa)

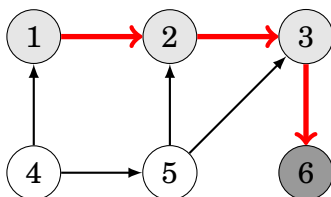
Aluksi jokaisen verkon solmun tila on 0. Kun syvyyshaku saapuu solmuun, sen tilaksi tulee 1. Lopuksi kun syvyyshaku on käsitellyt kaikki solmun naapurit, solmun tilaksi tulee 2.

Jos verkossa on sykli, tämä selviää syvyyshauun aikana siitä, että jossain vaiheessa haku saapuu solmuun, jonka tila on 1. Tässä tapauksessa topologista järjestystä ei voi muodostaa.

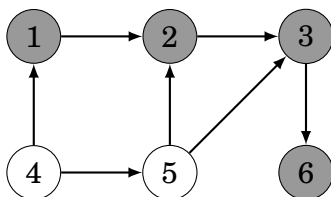
Jos verkossa ei ole sykliä, topologinen järjestys saadaan muodostamalla lista, johon kukin solmu lisätään silloin, kun sen tilaksi tulee 2. Tämä lista käänteisenä on yksi verkon topologinen järjestys.

Esimerkki 1

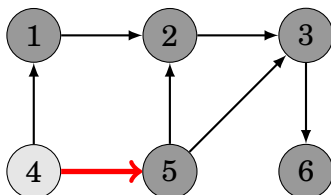
Esimerkkiverkossa syvyyshaku etenee ensin solmusta 1 solmuun 6 asti:



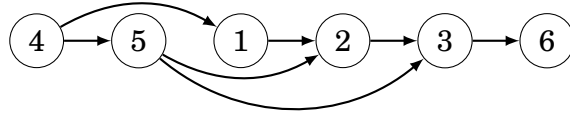
Tässä vaiheessa solmu 6 on käsitelty, joten se lisätään listalle. Sen jälkeen haku palaa takaisinpäin:



Tämän jälkeen listan sisältönä on (6,3,2,1). Seuraavaksi syvyyshauun toinen vaihe alkaa solmusta 4:



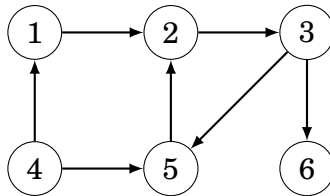
Tämän seurauksena listaksi tulee (6, 3, 2, 1, 5, 4). Kaikki solmut on käyty läpi, joten topologinen järjestys on valmis. Topologinen järjestys on lista käänteisenä eli (4, 5, 1, 2, 3, 6):



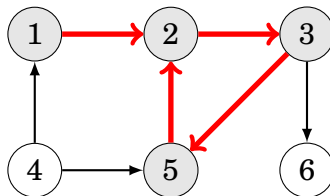
Huomaa, että topologinen järjestys ei ole yksikäsitteinen, vaan verkolla voi olla useita topologisia järjestyksiä.

Esimerkki 2

Tarkastellaan sitten tilannetta, jossa topologista järjestystä ei voi muodostaa syklin takia. Näin on seuraavassa verkossa:



Nyt syvyysshaun aikana tapahtuu näin:



Syvyysshaku saapuu tilassa 1 olevaan solmuun 2, mikä tarkoittaa, että verkossa on sykli. Tässä tapauksessa sykli on $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

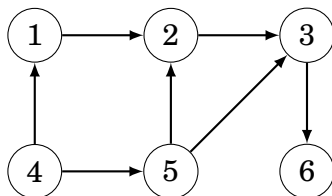
16.2 Dynaaminen ohjelmointi

Jos suunnattu verkko on sykliton, siihen voi soveltaa dynaamista ohjelmointia. Tämän avulla voi ratkaista tehokkaasti ajassa $O(n + m)$ esimerkiksi seuraavat ongelmat koskien verkossa olevia polkuja alkusolmusta loppusolmuun:

- montako erilaista polkua on olemassa?
- mikä on lyhin/pisin polku?
- mikä on pienin/suurin määrä kaaria polulla?
- mitkä solmut esiintyvät varmasti polulla?

Polkujen määrän laskeminen

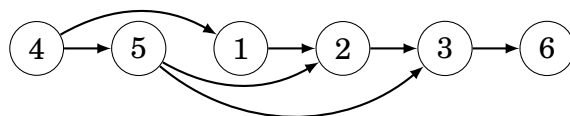
Lasketaan esimerkkinä polkujen määrä alkusolmusta loppusolmuun suunnatussa, syklittömässä verkossa. Esimerkiksi verkossa



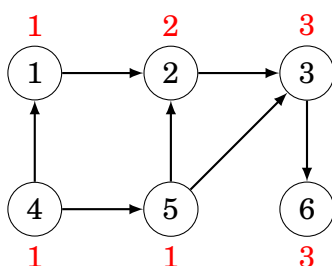
on 3 polkua solmusta 4 solmuun 6:

- $4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

Ideana on käydä läpi verkon solmut topologisessa järjestyksessä ja laskea kunkin solmun kohdalla yhteen eri suunnista tulevien polkujen määrät. Verkon topologinen järjestys on seuraava:



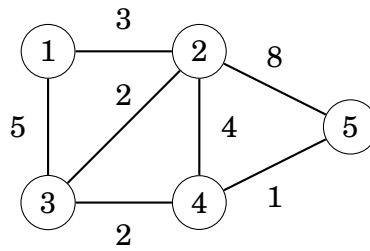
Tuloksena ovat seuraavat lukumäärät:



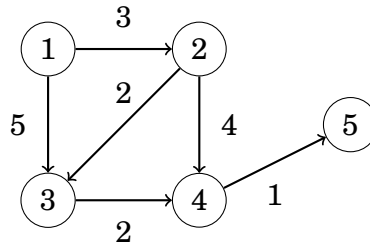
Esimerkiksi solmuun 2 pääsee solmuista 1 ja 5. Kumpaankin solmuun päättyy yksi polku solmusta 4 alkaen, joten solmuun 2 päättyy kaksi polkua solmusta 4 alkaen. Vastaavasti solmuun 3 pääsee solmuista 2 ja 5, joiden kautta tulee kaksi ja yksi polkua solmusta 4 alkaen.

Dijkstran algoritmin sovellukset

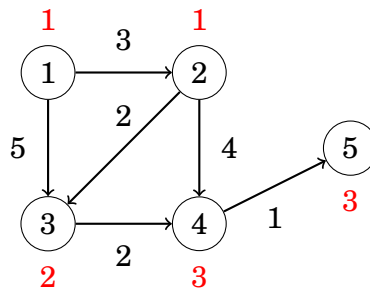
Dijkstran algoritmin sivutuotteena syntyy suunnattu, sykliton verkko, joka kertoo jokaiselle alkuperäisen verkon solmulle, mitä tapoja alkusolmusta on päästä kyseiseen solmuun lyhintä polkua käyttäen. Tähän verkkoon voi soveltaa edelleen dynaamista ohjelmointia. Esimerkiksi verkossa



solmusta 1 lähteviin lyhimpiin polkuihin kuuluvat seuraavat kaaret:



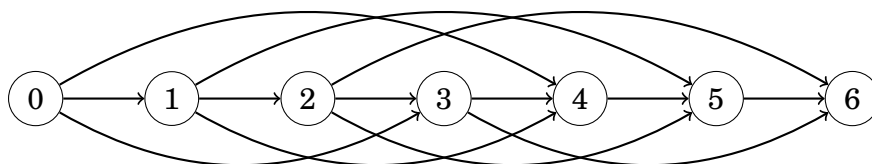
Koska kyseessä on suunnaton, syklittön verkko, siihen voi soveltaa dynaamista ohjelmointia. Niinpä voi esimerkiksi laskea, montako lyhintä polkua on olemassa solmusta 1 solmuun 5:



Ongelman esitys verkkona

Itse asiassa mikä tahansa dynaamisen ohjelmoinnin ongelma voidaan esittää suunnattuna, syklittömänä verkkona. Tällaisessa verkossa solmuja ovat dynaamisen ohjelmoinnin tilat ja kaaret kuvaavat, miten tilat riippuvat toisistaan.

Tarkastellaan esimerkkinä tuttua tehtävää, jossa annettuna on rahamäärä x ja tehtävänä on muodostaa se kolikoista $\{c_1, c_2, \dots, c_k\}$. Tässä tapauksessa voimme muodostaa verkon, jonka solmut vastaavat rahamääriä ja kaaret kuvaavat kolikkojen valintoja. Esimerkiksi jos kolikot ovat $\{1, 3, 4\}$ ja $x = 6$, niin verkosta tulee seuraava:



Tätä verkkoesitystä käyttäen lyhin polku solmusta 0 solmuun x vastaa ratkaisua, jossa kolikoita on mahdollisimman vähän, ja polkujen yhteismäärä solmusta 0 solmuun x on sama kuin ratkaisujen yhteismäärä.

16.3 Tehokas eteneminen

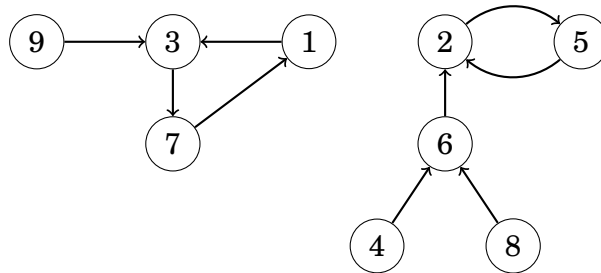
Seuraavaksi oletamme, että suunnaton verkko on **seuraajaverkko**, jolloin jokaisen solmun lähtöaste on 1 eli siitä lähtee tasan yksi kaari ulospäin. Niinpä verkko muodostuu yhdestä tai useammasta komponentista, joista jokaisessa on yksi sykli ja joukko siihen johtavia polkuja.

Seuraajaverkosta käytetään joskus nimeä **funktionaalinen verkko**. Tämä johtuu siitä, että jokaista seuraajaverkkoa vastaa funktio f , joka määrittelee verkon kaaret. Funktion parametrina on verkon solmu ja se palauttaa solmusta lähtevän kaaren kohdesolmun.

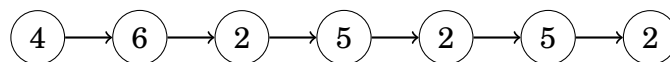
Esimerkiksi funktio

x	1	2	3	4	5	6	7	8	9
$f(x)$	3	5	7	6	2	2	1	6	3

määrittelee seuraavan verkon:



Koska seuraajaverkon 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 arvo $f(x, k)$ on käydä läpi polku askel askeleelta, mihin kuluu aikaa $O(k)$. Sopivan esikäsittelyn avulla voimme laskea kuitenkin minkä tahansa arvon $f(x, k)$ ajassa $O(\log u)$, missä u on suurin mahdollinen k :n arvo, jonka laskeminen tulee olla mahdollista.

Ideana on laskea etukäteen kaikki arvot $f(x, k)$, kun k on 2:n potenssi ja enintään u . Tämä onnistuu tehokkaasti, koska voimme käyttää rekursiota

$$f(x, k) = \begin{cases} f(x) & k = 1 \\ f(f(x, k/2), k/2) & k > 1 \end{cases}$$

Arvojen $f(x, k)$ esilaskenta vie aikaa $O(n \log u)$, koska jokaisesta solmusta lasketaan $O(\log u)$ arvoa. Esimerkin tapauksessa taulukko alkaa muodostua seuraavasti:

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 jos haluamme laskea arvon $f(x, 11)$, muodostamme ensin esityksen $11 = 8 + 2 + 1$. Tämän ansiosta

$$f(x, 11) = f(f(f(x, 8), 2), 1).$$

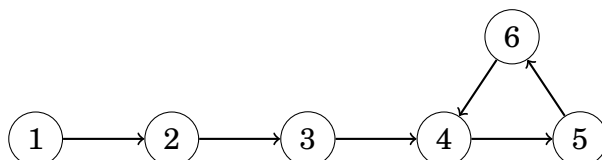
Esimerkiksi yllä olevassa verkossa

$$f(4, 11) = f(f(f(4, 8), 2), 1) = 5.$$

Tällaisessa esityksessä on aina $O(\log k)$ osaa, joten arvon $f(x, k)$ laskemiseen kuluu aikaa $O(\log k)$.

16.4 Syklin tunnistaminen

Kiinnostavia kysymyksiä seuraajaverkossa ovat, minkä solmun kohdalla saavutaan sykliin solmusta x lähdettäessä ja montako solmua kyseiseen sykliin kuuluu. Esimerkiksi verkossa



solmusta 1 lähdettäessä ensimmäinen sykliin kuuluva solmu on solmu 4 ja syklissä on kolme solmua (solmut 4, 5 ja 6).

Helppo tapa tunnistaa 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 syklin tunnistamiseen 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 Floydin algoritmiin, joka saavuttaa nämä ominaisuudet.

Floydin algoritmi

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:

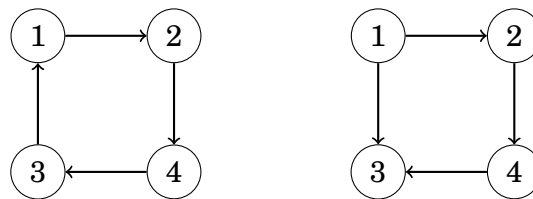
```
b = f(a);
c = 1;
while (a != b) {
    b = f(b);
    c++;
}
```

Luku 17

Vahvasti yhtenäisyys

Suunnatussa verkossa yhtenäisyys ei takaa sitä, että kahden solmun välillä olisi olemassa polkua, koska kaarten suunnat rajoittavat liikkumista. Niinpä suunnattuja verkkoja varten on mielekästä määritellä uusi käsite, joka vaatii enemmän verkon yhtenäisyydeltä.

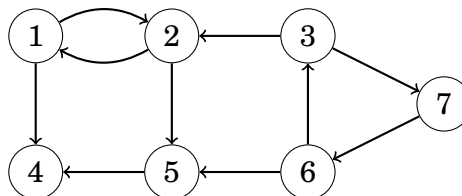
Verkko on **vahvasti yhtenäinen**, jos mistä tahansa solmusta on olemassa polku kaikkiin muihin solmuihin. Esimerkiksi seuraavassa kuvassa vasen verkko on vahvasti yhtenäinen, kun taas oikea verkko ei ole.



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

Verkon **vahvasti yhtenäiset komponentit** jakavat verkon solmut mahdollisimman suuriin vahvasti yhtenäisiin osiin. Verkon vahvasti yhtenäiset komponentit muodostavat syklittömän **komponenttiverkon**, joka kuvaa alkupe-
räisen verkon syvärakennetta.

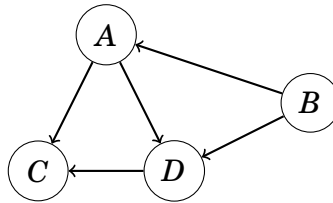
Esimerkiksi verkon



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 alkuperäistä verkkoa helpompaa, koska siinä ei ole syklejä. Niinpä komponenttiverkolle voi muodostaa luvun 16 tapaan topologisen järjestyksen ja soveltaa sen jälkeen dynaamista ohjelmintia verkon käsittelyyn.

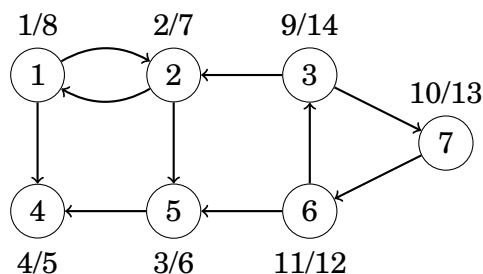
17.1 Kosarajun algoritmi

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 syvyysshaussa alkoi hetkellä x ja päättyi hetkellä y . Kun solmut järjestetään käsittelyn päättymisajan mukaan, tuloksena on seuraava järjestys:

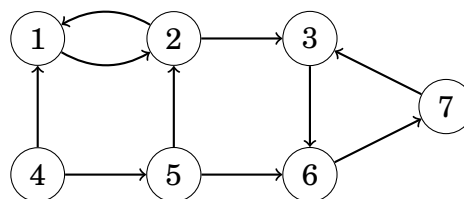
solmu	päätymisaika
4	5
5	6
2	7
1	8
6	12
7	13
3	14

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

Syvyyshaku 2

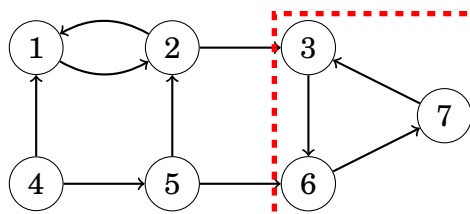
Algoritmin toinen vaihe muodostaa verkon vahvasti yhtenäiset komponentit. Ennen toista syvyyshakua algoritmi muuttaa jokaisen kaaren suunnan käänteiseksi. Tämä varmistaa, että toisen syvyyshaun aikana löydetään joka kerta vahvasti yhtenäinen komponentti, johon ei kuulu ylimääräisiä solmuja.

Esimerkkiverkko on käännettynä seuraava:



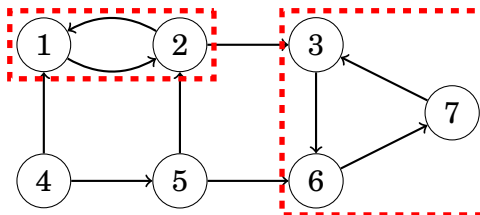
Tämän jälkeen algoritmi käy läpi solmut käänteisessä ensimmäisen syvyyshaun tuottamassa järjestyksessä. Jos solmu ei kuulu vielä komponenttiin, siitä alkaa uusi syvyyshaku. Solmun komponenttiin liitetään kaikki aiemmin käsittelemättömät solmut, joihin syvyyshaku pääsee solmusta.

Esimerkkiverkossa muodostuu ensin komponentti solmusta 3 alkaen:

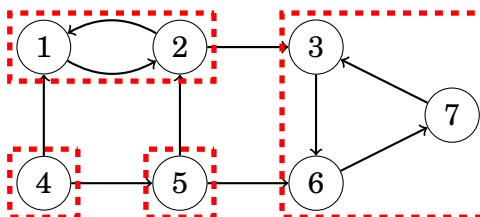


Huomaa, että kaarten kääntämisen ansiosta komponentti ei pääse ”vuotamaan” muihin verkon osiin.

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 $O(n + m)$.

17.2 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

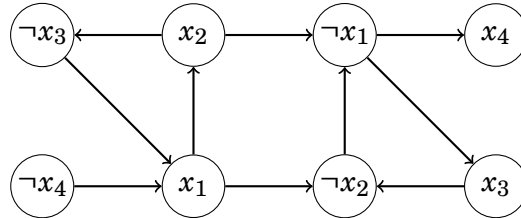
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

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.

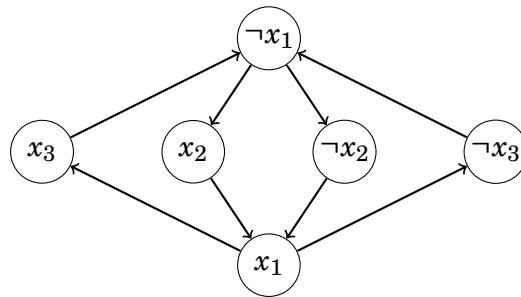
2SAT-ongelman saa muutettua verkoksi niin, että jokainen muuttuja x_i ja negaatio $\neg x_i$ on yksi verkon solmuista ja muuttujien riippuvuudet ovat kaaria.

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:

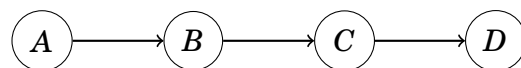


Verkon rakenne kertoo, onko 2SAT-ongelmalla ratkaisua. Jos on jokin muuttuja x_i niin, että x_i ja $\neg x_i$ ovat samassa vahvasti yhtenäisessä komponentissa, niin ratkaisua ei ole olemassa. Tällöin verkossa on polku sekä x_i :stä $\neg x_i$:ään että $\neg x_i$:stä x_i :ään, eli kumman tahansa arvon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

Lausekkeen L_1 verkon komponenttiverkko on seuraava:



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

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

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

Luku 18

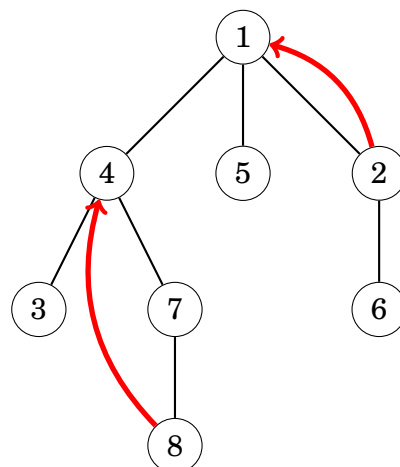
Puukyselyt

Tässä luvussa tutustumme algoritmeihin, joiden avulla voi toteuttaa tehokkaasti kyselyitä juurelliseen puuhun. Kyselyt liittyvät puussa oleviin polkuihin ja alipuihin. Esimerkkejä kyselyistä ovat:

- mikä solmu on k askelta ylempänä solmua x ?
- mikä on solmun x alipuun arvojen summa?
- mikä on solmujen a ja b välisen polun arvojen summa?
- mikä on solmujen a ja b alin yhteinen esivanhempi?

18.1 Tehokas nouseminen

Tehokas nouseminen puussa tarkoittaa, että voimme selvittää nopeasti, mihin solmuun päätyy kulkemalla k askelta ylöspäin solmusta x alkaen. 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, 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:

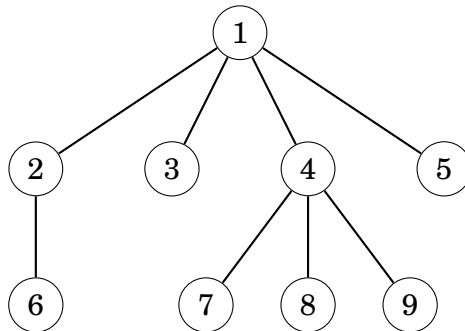
x	1	2	3	4	5	6	7	8
$f(x, 1)$	0	1	4	1	1	2	4	7
$f(x, 2)$	0	0	1	0	0	1	1	4
$f(x, 4)$	0	0	0	0	0	0	0	0
...								

Taulukossa arvo 0 tarkoittaa, että nousemalla k askelta päätyy puun ulkopuolelle juurisolmun yläpuolelle.

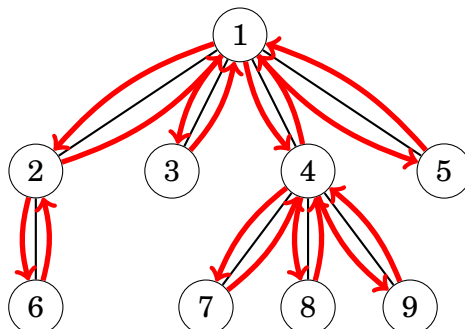
Esilaskenta vie aikaa $O(n \log n)$, koska jokaisesta solmusta voi nousta korkeintaan n askelta ylöspäin. Tämän jälkeen minkä tahansa funktion $f(x, k)$ arvon saa laskettua ajassa $O(\log k)$ jakamalla nousun 2:n potenssin osiin.

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

Alipuiden käsittely

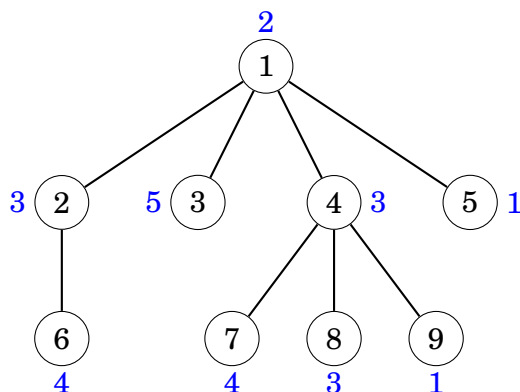
Solmutaulukossa jokaisen alipuun kaikki solmut ovat peräkkäin niin, että ensin on alipuun juurisolmu ja sitten kaikki muut alipuun solmut. Esimerkiksi äskeisessä taulukossa solmun 4 alipuuta vastaa seuraava taulukon osa:

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. Ratkaistaan esimerkkinä tehtävä, jossa kuhunkin puun solmuun liittyy arvo ja toteutettavana on seuraavat kyselyt:

- muuta solmun x arvoa
- 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

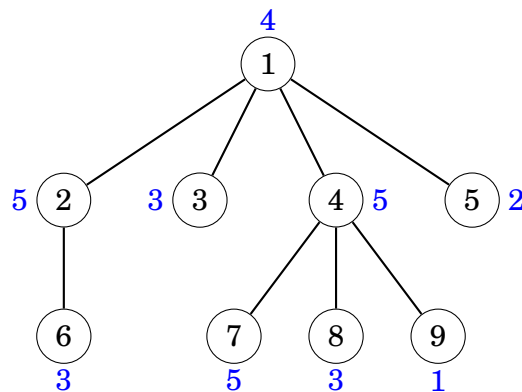
Viimeinen tarvittava askel on tallentaa solmujen arvot binääri-indeksipuuhun tai segmenttipuuhun. Tällöin sekä alipuun arvojen summan laskeminen että solmun arvon muuttaminen onnistuvat ajassa $O(\log n)$, eli pystymme vastaamaan kyselyihin tehokkaasti.

Polkujen käsittely

Solmutaulukon avulla voi myös käsitellä tehokkaasti polkuja, jotka kulkevat juuresta tiettyyn solmuun puussa. Ratkaistaan seuraavaksi tehtävä, jossa toteutettavana on seuraavat kyselyt:

- muuta solmun x arvoa
- laske arvojen summa juuresta solmuun x

Esimerkiksi seuraavassa puussa polulla solmusta 1 solmuun 8 arvojen summa on $4 + 5 + 3 = 12$.



Ideana on muodostaa samanlainen taulukko kuin alipuiden käsittelyssä mutta tallentaa solmujen arvot erikoisella tavalla: kun taulukon kohdassa k olevan solmun arvo on a , kohdan k arvoon lisätään a ja kohdan $k + c$ arvosta vähennetään 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	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

Summaksi tulee

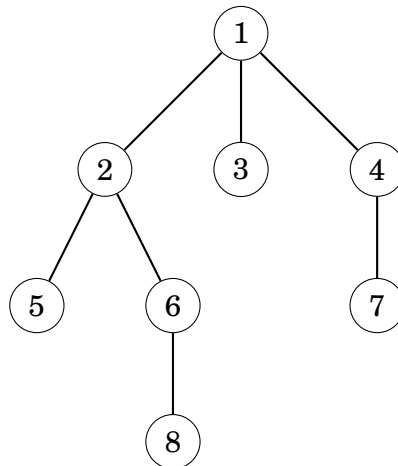
$$4 + 5 + 3 - 5 + 2 + 5 - 2 = 12,$$

mikä vastaa polun summaa $4 + 5 + 3 = 12$. Tämä laskentatapa toimii, koska jokaisen solmun arvo lisätään summaan, kun se tulee vastaan syvyyshaussa, 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

Kahden puun solmun **alin yhteinen esivanhempi** on mahdollisimman alhaalla puussa oleva solmu, jonka alipuuhun kumpikin solmu kuuluu. Tyypillinen tehtävä on vastata tehokkaasti joukkoon kyselyitä, jossa selvittävänä on kahden solmun alin yhteinen esivanhempi. Esimerkiksi puussa



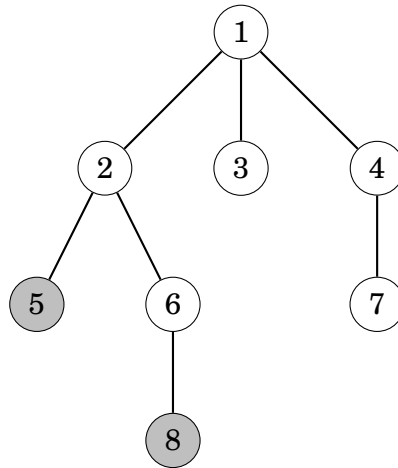
solmujen 5 ja 8 alin yhteinen esivanhempi on solmu 2 ja solmujen 3 ja 4 alin yhteinen esivanhempi on solmu 1.

Tutustumme seuraavaksi kahteen menetelmään alimman yhteisen esivanhemman selvittämiseen.

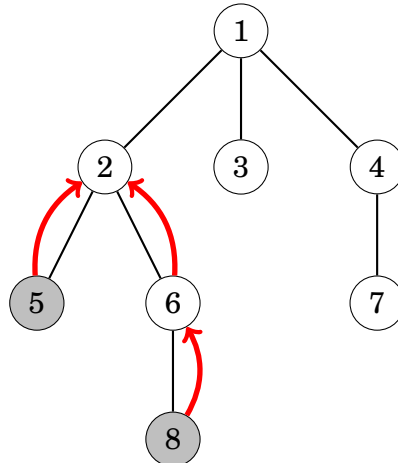
Menetelmä 1

Yksi tapa ratkaista tehtävä on hyödyntää tehokasta nousemista puussa. Tällöin alimman yhteisen esivanhemman etsiminen muodostuu kahdesta vaiheesta. Ensin nouseaan alemmasta solmusta samalle tasolle ylemmän solmun kanssa, sitten nouseaan rinnakkain kohti alinta yhteistä esivanhempaa.

Tarkastellaan esimerkkinä solmujen 5 ja 8 alimman yhteisen esivanheman etsimistä:



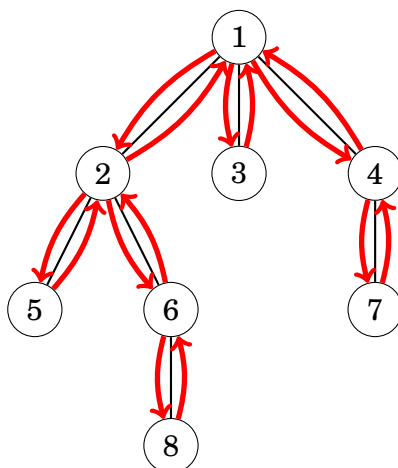
Solmu 5 on tasolla 3, kun taas solmu 8 on tasolla 4. Niinpä nousemme ensin solmusta 8 yhden tason ylemmäs solmuun 6. Tämän jälkeen nousemme rinnakkain solmuista 5 ja 6 lähtien yhden tason, jolloin päädymme solmuun 2:



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. Esimerkkiä 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
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

↑

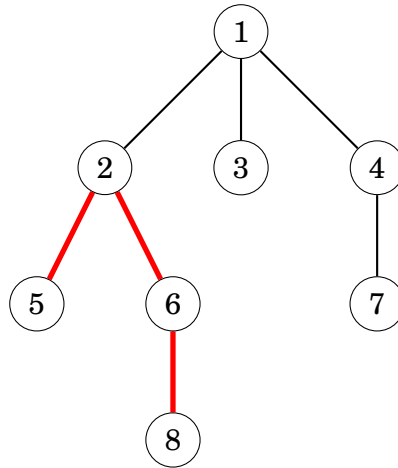
Solmu 5 on taulukossa kohdassa 3, solmu 8 on taulukossa kohdassa 6 ja alimman tason solmu välillä 3...6 on kohdassa 4 oleva solmu 2, jonka taso on 2. Niinpä solmujen 5 ja 8 alin yhteinen esivanhempi on solmu 2.

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

Tarkastellaan lopuksi tehtävää, jossa kyselyissä tulee laskea tehokkaasti kahden solmun etäisyys eli solmujen välisen polun pituus puussa. Osoittautuu, että tämä tehtävä palautuu alimman yhteisen esivanhemman etsimiseen.

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

Tämä luku käsittelee kahdenlaisia polkuja verkossa:

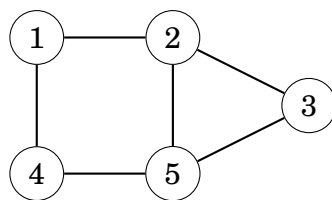
- **Eulerin polku** on verkossa oleva polku, joka kulkee tasan kerran jokaista verkon kaarta pitkin.
- **Hamiltonin polku** on verkossa oleva polku, joka käy tasan kerran jokaisessa verkon solmussa.

Vaikka Eulerin ja Hamiltonin polut näyttävät päältä päin samantapaisilta käsitteiltä, niihin liittyy hyvin erilaisia laskennallisia ongelmia.

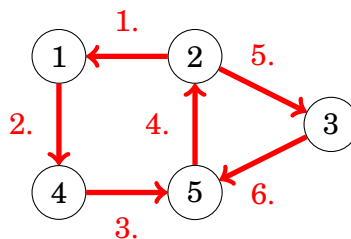
Osoittautuu, että yksinkertainen verkon solmujen asteisiin liittyvä sääntö ratkaisee, onko verkossa Eulerin polkua, ja polun voi myös muodostaa tehokkaasti. Sen sijaan Hamiltonin polun etsimiseen ei tunneta mitään tehokasta algoritmia, vaan kyseessä on NP-vaikea ongelma.

19.1 Eulerin polku

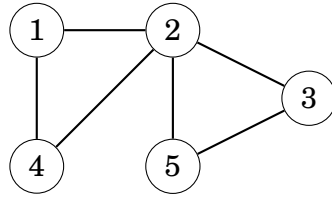
Eulerin polku on verkossa oleva polku, joka kulkee tarkalleen kerran jokaista kaarta pitkin. Esimerkiksi verkossa



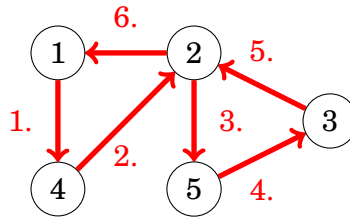
on Eulerin polku solmusta 2 solmuun 5:



Eulerin kierros on puolestaan Eulerin polku, jonka alku- ja loppusolmu ovat samat. Esimerkiksi verkossa



on Eulerin kierros, jonka alku- ja loppusolmu on 1:



Olemassaolo

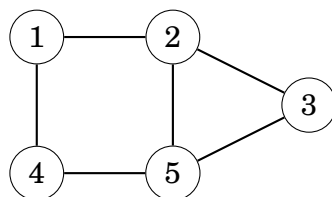
Osoittautuu, että Eulerin polun ja kierroksen olemassaolo riippuu verkon solmujen asteista. Solmun aste on sen naapurien määrä eli niiden solmujen määrä, jotka ovat yhteydessä solmuun kaarella.

Suuntaamattomassa verkossa on Eulerin polku, jos kaikki kaaret ovat samassa yhtenäisessä komponentissa ja

- jokaisen solmun aste on parillinen *tai*
- tarkalleen kahden solmun aste on pariton ja kaikkien muiden solmujen aste on parillinen.

Ensimmäisessä tapauksessa Eulerin polku on samalla myös Eulerin kierros. Jälkimmäisessä tapauksessa Eulerin polun alku- ja loppusolmu ovat paritonasteiset solmut ja se ei ole Eulerin kierros.

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.

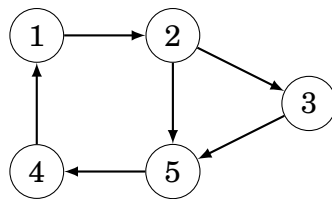
Jos verkko on suunnattu, tilanne on hieman hankalampi. Silloin Eulerin polun ja kierroksen olemassaoloon vaikuttavat solmujen lähtö- ja tuloasteet. Solmun lähtöaste on solmusta lähtevien kaarten määrä, ja vastaavasti solmun tuloaste on solmuun tulevien kaarten määrä.

Suunnatussa verkossa on Eulerin polku, jos kaikki kaaret ovat samassa vahvasti yhtenäisessä komponentissa ja

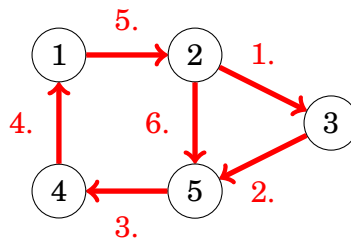
- jokaisen solmun lähtö- ja tuloaste on sama *tai*
- yhdessä solmussa lähtöaste on yhden suurempi kuin tuloaste, toisessa solmussa tuloaste on yhden suurempi kuin lähtöaste ja kaikissa muissa solmuissa lähtö- ja tuloaste on sama.

Tilanne on vastaava kuin suuntaamattomassa verkossa: ensimmäisessä tapauksessa Eulerin polku on myös Eulerin kierros, ja toisessa tapauksessa verkossa on vain Eulerin polku, jonka lähtösolmussa lähtöaste on suurempi ja päätesolmussa tuloaste on suurempi.

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:



Hierholzerin algoritmi

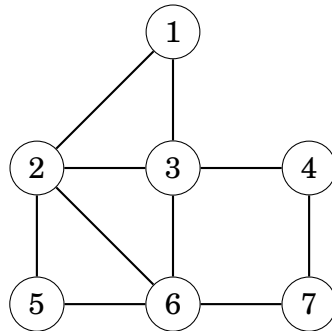
Hierholzerin algoritmi muodostaa Eulerin kierroksen suuntaamattomassa verkossa. Algoritmi olettaa, että kaikki kaaret ovat samassa komponentissa ja jokaisen solmun aste on parillinen.

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.

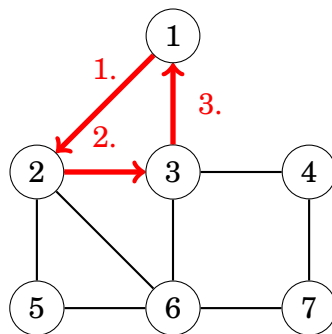
Algoritmi muodostaa ensin verkkoon jonkin kierroksen, johon kuuluu osa verkon kaarista. Sen jälkeen algoritmi alkaa laajentaa kierrosta lisäämällä sen osaksi uusia alikierroksia. Tämä jatkuu niin kauan, kunnes kaikki kaaret kuuluvat kierrokseen ja siitä on tullut Eulerin kierros.

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 lähtösolmuun x .

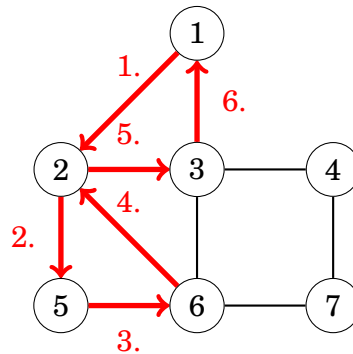
Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



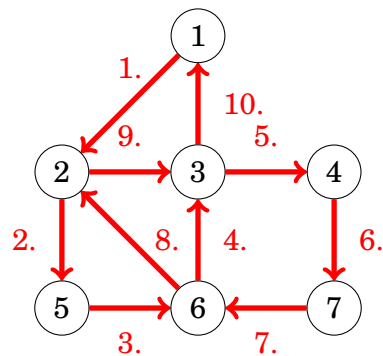
Oletetaan, että algoritmi aloittaa ensimmäisen kierroksen solmusta 1. Siitä syntyy kierros $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



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

```
stack<int> s;
```

jossa on aluksi vain kierroksen alkusolmu. Jos pinon ylimmän solmun u aste on 0, niin algoritmi lisää solmun Eulerin kierrokseen. Muuten algoritmi rakentaa pinon päälle uuden alikierroksen solmusta u alkaen ja poistaa kaikki alikierrokseen kuuluvat kaaret verkosta.

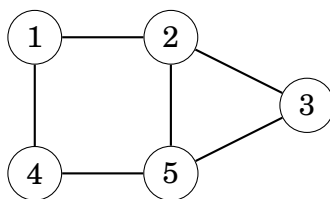
```
s.push(x);
while (!s.empty()) {
    int u = s.top(); s.pop();
    if (v[u].size() == 0) {
        // lisää solmu u Eulerin kierrokseen
    } else {
        int a = u;
        s.push(a);
        do {
            int b = *v[a].begin();
            v[a].erase(b);
            v[b].erase(a);
            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 $O(\log n)$.

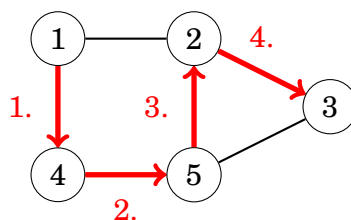
Myös toteutus ajassa $O(n + m)$ on mahdollista mutta vaikeampaa. Tämä vaatii verkon esittämistä niin, että kaaria pystyy poistamaan ajassa $O(1)$, mikä on mahdollista kahteen suuntaan linkitettyjen vieruslistojen avulla.

19.2 Hamiltonin polku

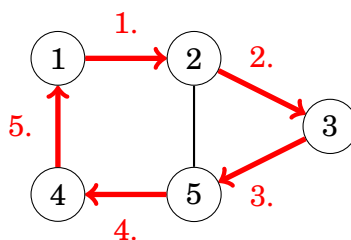
Hamiltonin polku on verkossa oleva polku, joka kulkee tarkalleen kerran jokaisen solmun kautta. Esimerkiksi verkossa



on Hamiltonin polku solmusta 1 solmuun 3:



Jos Hamiltonin polun alku- ja loppusolmu on sama, kyseessä on **Hamiltonin kierros**. Äskeisessä verkossa on myös Hamiltonin kierros, jonka alku- ja loppusolmu on solmu 1:



Olemassaolo

Hamiltonin polun olemassaoloon ei tiedetä mitään verkon rakenteeseen liittyvää ehtoa, jonka voisi tarkistaa tehokkaasti. Joissakin erikoistapauksissa voidaan silti sanoa varmasti, että verkossa on Hamiltonin polku.

Yksinkertainen havainto on, että jos verkko on täydellinen eli jokaisen solmun välillä on kaari, niin siinä on Hamiltonin polku. Myös vahvempia tuloksia on saatu aikaan:

- **Diracin lause:** Jos jokaisen verkon solmun aste on $n/2$ tai suurempi, niin verkossa on Hamiltonin polku.
- **Oren lause:** Jos jokaisen ei-vierekkäisen solmuparin asteiden summa on n tai suurempi, niin verkossa on Hamiltonin polku.

Yhteistä näissä ja muissa tuloksissa on, että ne takaavat Hamiltonin polun olemassaolon, jos verkossa on *paljon* kaaria. Tämä on ymmärrettävää, koska mitä enemmän kaaria verkossa on, sitä enemmän mahdollisuuksia Hamiltonin polun muodostamiseen on olemassa.

Muodostaminen

Koska Hamiltonin polun olemassaoloa ei voi tarkastaa tehokkaasti, on selvää, että polkua ei voi myöskään muodostaa tehokkaasti, koska muuten polun olemassaolon voisi selvittää yrittämällä muodostaa sen.

Yksinkertaisin tapa etsiä Hamiltonin polkua on käyttää peruuttavaa hakua, joka käy läpi kaikki vaihtoehdot polun muodostamiseen. Tällaisen algoritmin aikavaativuus on ainakin luokkaa $O(n!)$, koska n solmusta voidaan muodostaa $n!$ järjestystä, jossa ne voivat esiintyä polulla.

Tehokkaampi tapa perustuu dynaamiseen ohjelmointiin luvun 10.4 tapaan. Ideana on määritellä funktio $f(s, x)$, jossa s on verkon solmujen osajoukko ja x on yksi osajoukon solmuista. Funktio kertoo, onko olemassa Hamiltonin polkua, joka käy läpi joukon s solmut päätyen solmuun x . Tällainen ratkaisu on mahdollista toteuttaa ajassa $O(2^n n^2)$.

19.3 De Bruijin jono

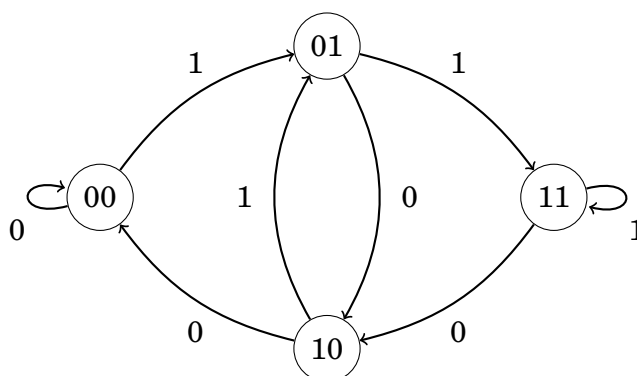
De Bruijin jono on lyhin k -merkkisen aakkoston merkkijono, jonka osajonoina ovat kaikki mahdolliset n merkin yhdistelmät. Esimerkiksi kun $k = 2$ ja $n = 3$, niin eräs de Bruijin jono on

0001011100.

Tämän merkkijonon osajonoina ovat kaikki 3 merkin yhdistelmät 000, 001, 010, 011, 100, 101, 110 ja 111.

Osoittautuu, että de Bruijin jonon pituus on aina $k^n + n - 1$, jolloin jokainen n merkin osajono esiintyy tarkalleen kerran merkkijonossa. Lisäksi de Bruijin jono vastaa Eulerin kierrosta sopivasti muodostetussa verkossa.

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 kaaressa on k^n merkkiä, joten tuloksena on lyhin mahdollinen merkkijono.

19.4 Ratsun kierros

Ratsun kierros on tapa liikuttaa ratsua shakin sääntöjen mukaisesti $n \times n$ - kokoisella shakkilaudalla niin, että ratsu käy tarkalleen kerran jokaisessa ruudussa. Ratsun kierros on **suljettu**, jos ratsu palaa lopuksi alkuruutuun, ja muussa tapauksessa kierros on **avoin**.

Esimerkiksi tapauksessa 5×5 yksi ratsun kierros on seuraava:

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

Ratsun kierros shakkilaudalla vastaa Hamiltonin polkua verkossa, jonka solmut ovat ruutuja ja kahden solmun välillä on kaari, jos ratsu pystyy siirtymään solmusta toiseen shakin sääntöjen mukaisesti.

Peruuttava haku on luonteva menetelmä ratsun kierroksen muodostamiseen. Hakua voi tehostaa erilaisilla **heuristiikoilla**, jotka pyrkivät ohjaamaan ratsua niin, että kokonainen kierros tulee valmiiksi nopeasti.

Warnsdorffin sääntö

Warnsdorffin sääntö on yksinkertainen mutta käytännössä hyvä heuristiikka ratsun kierroksen etsimiseen. Sen avulla on mahdollista löytää nopeasti ratsun kierros suurestakin ruudukosta.

Heuristiikassa on ideana siirtää ratsua aina niin, että se päättyy ruutuun, josta on mahdollisimman *vähän* mahdollisuuksia jatkaa kierrosta. Esimerkiksi seuraavassa tilanteessa on valittavana viisi ruutua, joihin ratsu voi siirtyä:

1				<i>a</i>
		2		
<i>b</i>				<i>e</i>
	<i>c</i>		<i>d</i>	

Tässä tapauksessa Warnsdorffin sääntö valitsee ruudun *a*, koska tämän valinnan jälkeen on vain yksi mahdollisuus jatkaa kierrosta. Muissa valinnoissa mahdollisuuksia olisi kolme.

Luku 20

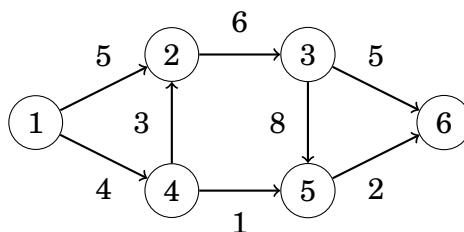
Virtauslaskenta

Virtauslaskennan keskeiset ongelmat ovat:

- **Maksimivirtauksen etsiminen:** Kuinka paljon virtausta on mahdollista kuljettaa verkon alkusolmusta loppusolmuun kaaria pitkin?
- **Minimileikkauksen etsiminen:** Mikä on yhteispainoltaan pienin joukko kaaria, joiden poistaminen erottaa alkusolmun loppusolmusta?

Osoittautuu, että nämä ongelmat vastaavat toisiaan ja ne on mahdollista ratkaista samanaikaisesti toistensa avulla.

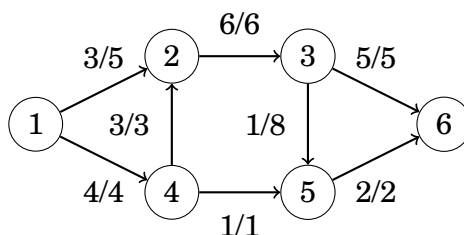
Oletamme, että annettuna on suunnattu, painotettu verkko, jossa on valittu tietty alkusolmu ja loppusolmu. Tarkastelemme esimerkkinä seuraavaa verkkoa, jossa solmu 1 on alkusolmu ja solmu 6 on loppusolmu:



Maksimivirtaus

Virtaus lähtee liikkeelle alkusolmusta ja päättyy loppusolmuun. Kunkin kaaren paino on kapasiteetti, joka ilmaisee, kuinka paljon virtausta kaaren kautta voi kulkea. Kaikissa solmuissa alkusolmu ja loppusolmu lukuun ottamatta tulevan ja lähtevän virtauksen on oltava yhtä suuri.

Maksimivirtaus 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 alkusolmista lähtevä virtaus on $3 + 4 = 7$ ja loppusolmuun saapuva virtaus on $5 + 2 = 7$.

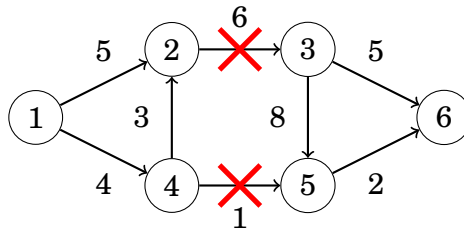
Huomaa, että jokaisessa välisolmussa tulevan ja lähtevän virtauksen määrä on sama. Esimerkiksi solmuun 2 tulee virtausta $3 + 3 = 6$ yksikköä solmuista 1 ja 4 ja siitä lähtee virtausta 6 yksikköä solmuun 3.

Virtaus 7 on verkon maksimivirtaus, koska verkon rakenteesta johtuen ei ole tapaa kuljettaa enempää virtausta verkossa.

Minimileikkaus

Leikkaus jakaa verkon solmut kahteen osaan niin, että alkusolmu ja loppusolmu ovat eri osissa. Leikkauksen paino on niiden kaarten yhteispaino, jotka kulkevat alkuosasta loppuosaan.

Minimileikkaus 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.

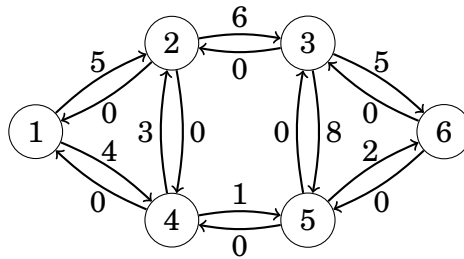
Seuraavaksi tutustumme Ford–Fulkersonin algoritmiin, jolla voi etsiä verkon maksimivirtauksen ja minimileikkauksen. Algoritmi auttaa myös ymmärtämään, *miksi* maksimivirtaus ja minimileikkaus ovat yhtä suuret.

20.1 Ford–Fulkersonin algoritmi

Ford–Fulkersonin algoritmi etsii verkon maksimivirtauksen. Algoritmi aloittaa tilanteesta, jossa virtaus on 0, ja alkaa sitten etsiä 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.

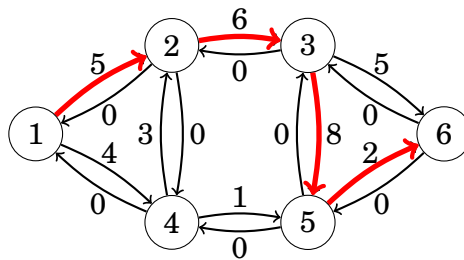
Esimerkkiverkosta syntyy seuraava verkko:



Algoritmin toiminta

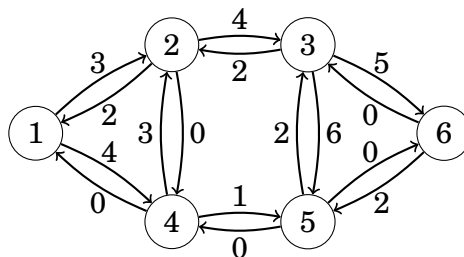
Ford–Fulkersonin algoritmi etsii verkosta joka vaiheessa polun, joka alkaa alkusolmusta, päättyy loppusolmuun ja jossa jokaisen kaaren paino on positiivinen. Jos vaihtoehtoja on useita, mikä tahansa valinta kelpaa.

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. Onneksi tämän ilmiön pystyy estämään käyttämällä seuraavia menetelmiä.

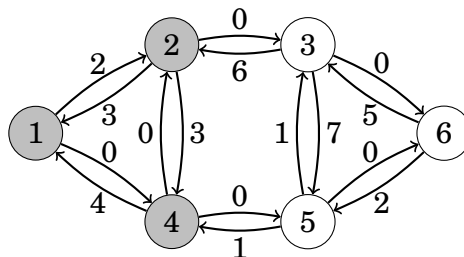
Edmonds–Karpin algoritmi on Ford–Fulkersonin algoritmin toteutus, jossa virtausta kasvattava polku valitaan aina niin, että siinä on mahdollisimman vähän kaaria. Tämä onnistuu etsimällä polku syvyysshaun sijasta leveyshaulla. Osoittautuu, että tämä varmistaa virtauksen kasvamisen nopeasti ja maksimivirtauksen 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 kelpollista 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

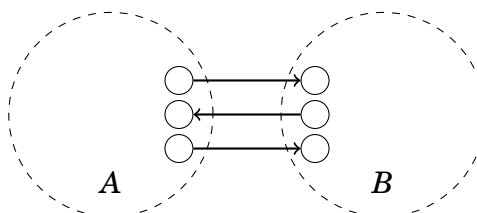
Osoittautuu, että kun Ford–Fulkersonin algoritmi on saanut valmiiksi maksimivirtauksen, se on tuottanut samalla minimileikkauksen. Olkoon A niiden solmujen joukko, joihin verkossa pääsee alkusolmusta positiivisia kaaria pitkin. Esimerkkiverkossa A sisältää solmut 1, 2 ja 4:



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:



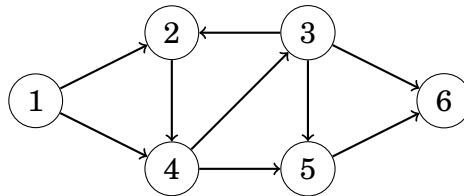
Leikkauksen paino on niiden kaarten painojen summa, jotka kulkevat osasta A osaan B . Tämä on yläraja sille, kuinka suuri verkossa oleva virtaus voi olla, koska virtauksen täytyy edetä osasta A osaan B . Niinpä maksimivirtaus on pienempi tai yhtä suuri kuin mikä tahansa verkon leikkaus.

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.

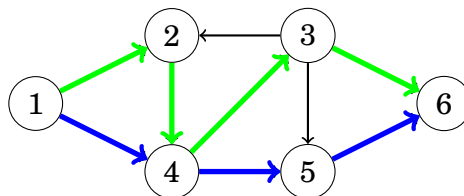
20.2 Rinnakkaiset polut

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 $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$:



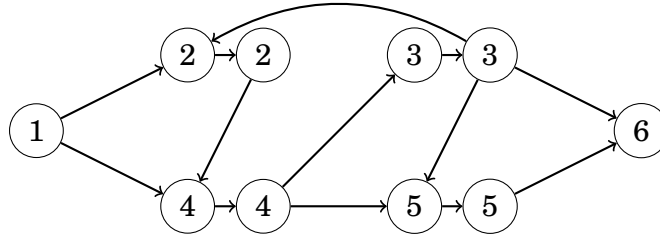
Osoittautuu, että suurin rinnakkaisten polkujen määrä on yhtä suuri kuin maksimivirtaus verkossa, jossa jokaisen kaaren kapasiteetti on 1. Kun maksimivirtaus on muodostettu, rinnakkaiset polut voi löytää ahneesti etsimällä alkusolmusta loppusolmuun kulkevia polkuja.

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:

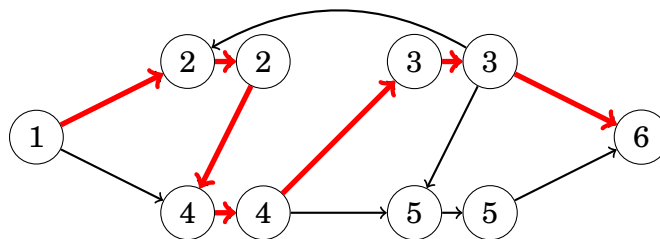


Tavallinen keino rajoittaa solmun kautta kulkevaa virtausta on jakaa solmu tulosolmuksi ja lähtösolmuksi. Kaikki solmuun tulevat kaaret saapuvat tulosolmuun ja kaikki solmusta lähtevät kaaret poistuvat lähtösolmusta. Lisäksi tulosolmusta lähtösolmuun on kaari, jossa on haluttu kapasiteetti.

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.

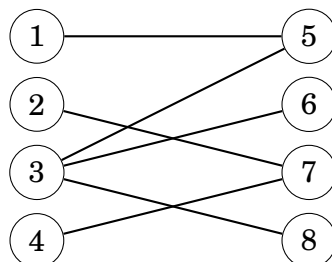
20.3 Maksimiparitus

Paritus on kokoelma verkon kaaria, jotka on valittu niin, että jokainen verkon solmu esiintyy enintään yhden kaaren päätesolmuna. Paritus muodostaa siis verkon solmuista joukon pareja. **Maksimiparitus** on puolestaan paritus, jossa parien määrä on mahdollisimman suuri.

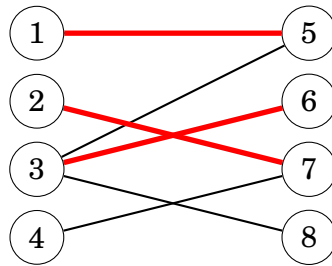
Maksimiparituksen etsimiseen yleisessä verkossa on olemassa polynominen algoritmi, mutta se on hyvin monimutkainen. Tässä luvussa keskitymmekin tilanteeseen, jossa verkko on kaksijakoinen. Tällöin maksimiparituksen pystyy etsimään helposti virtauslaskennan avulla.

Maksimiparituksen etsiminen

Kaksijakoinen verkko voidaan esittää aina niin, että se muodostuu vasemman ja oikean puolen solmuista ja kaikki verkon kaaret kulkevat puolten välillä. Tarkastellaan esimerkkinä seuraavaa verkkoa:

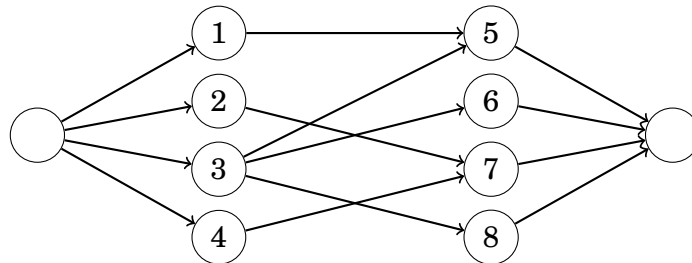


Tässä verkossa maksimiparituksen koko on 3:



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:



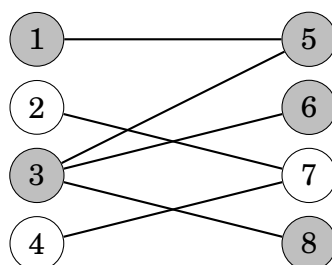
Tämän verkon maksimivirtaus on yhtä suuri kuin alkuperäisen verkon maksimiparitus, koska virtaus muodostuu joukosta polkuja alkusolmusta loppusolmuun ja jokainen polku ottaa mukaan uuden kaaren paritukseen. Tässä tapauksessa maksimivirtaus on 3, joten maksimiparitus on myös 3.

Hallin lause

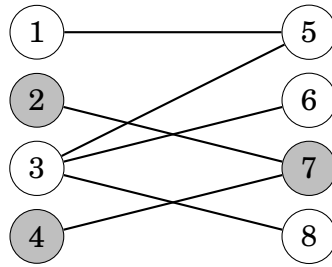
Hallin lause antaa ehdon, milloin kaksijakoiseen verkkoon voidaan muodostaa paritus, joka sisältää kaikki toisen puolen solmut. Jos kummallakin puolella on yhtä monta solmua, Hallin lause kertoo, voidaanko muodostaa **täydellinen paritus**, jossa kaikki solmut paritetaan keskenään.

Oletetaan, että haluamme muodostaa parituksen, johon kuuluvat kaikki vasemman puolen solmut. Merkitään $f(X)$ joukkoa, jonka jokainen solmu on jonkin joukon X solmun naapuri. Hallin lauseen mukaan paritus on mahdollinen, kun jokaiselle vasemman puolen 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 ole voimassa. Tämä tarkoittaa, että ei ole mahdollista muodostaa paritusta, jossa ovat mukana kaikki vasemman puolen solmut (eli täydellistä paritusta). Tämä on odotettu tulos, koska verkon maksimiparitutus on 3 eikä 4.

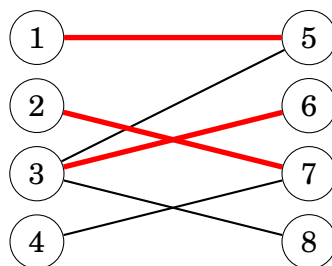
Jos Hallin lauseen ehto ei päde, osajoukko X kertoo syyn sille, miksi paritusta ei voi muodostaa. Koska X sisältää enemmän solmuja kuin $f(X)$, kaikille X :n solmuille ei riitä paria oikealta. Esimerkiksi yllä molemmat solmut 2 ja 4 tulisi yhdistää solmuun 7, mutta tämä ei ole mahdollista.

Königin lause

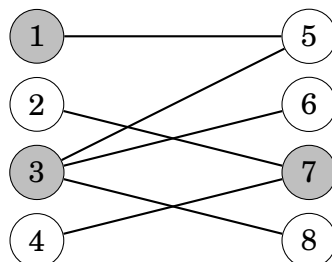
Solmupeite on sellainen joukko verkon solmuja, että jokaisesta verkon kaaresta ainakin toinen kaaren päätesolmuista kuuluu joukkoon. **Pienin solmupeite** on puolestaan solmupeite, jossa on mahdollisimman vähän solmuja.

Yleisessä verkossa pienimmän solmupeitteen etsiminen on NP-vaikea ongelma. Kaksijakoisessa verkossa tilanne on kuitenkin toinen, koska **Königin lause** tuo yhteyden maksimiparituksen ja pienimmän solmupeitteen välille. Lauseen mukaan maksimiparitutus ja pienin solmupeite ovat aina yhtä suuria.

Esimerkiksi seuraavan verkon maksimiparituksen koko on 3:



Niinpä myös pienimmän solmupeitteen koko on 3. Solmupeite voidaan muodostaa valitsemalla siihen solmut $\{1, 3, 7\}$:

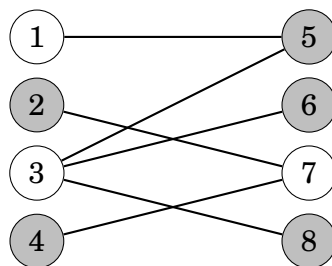


Pienin solmupeite muodostuu aina niin, että jokaisesta maksimiparituksen kaaresta toinen kaaren päätesolmuista kuuluu peitteeseen.

Riippumaton joukko on joukko verkon solmuja, jossa minkään kahden solmun välillä ei ole kaarta. **Suurin riippumaton joukko** on taas riippumaton joukko, jossa on mahdollisimman monta solmua.

Solmupeite ja riippumaton joukko liittyvät toisiinsa, koska solmupeitteen ulkopuolelle jäävät solmut muodostavat riippumattoman joukon. Niinpä jos verkossa on n solmua ja pienin solmupeite muodostuu k solmusta, niin suurin riippumaton joukko muodostuu $n - k$ solmusta.

Tämän ansiosta Königin lause lauseen avulla saa selville myös kaksijakoisen verkon suurimman riippumattoman joukon. Esimerkkiverkossa suurin riippumaton joukko on seuraava:



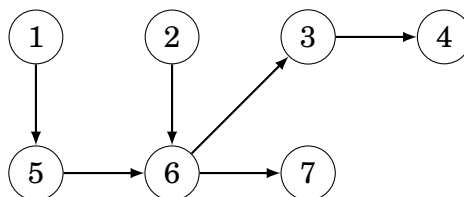
20.4 Polkupeitteet

Polkupeite on joukko verkon polkuja, joka on muodostettu 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.

Solmuerillinen peite

Tarkastellaan esimerkkinä seuraavaa verkkoa:



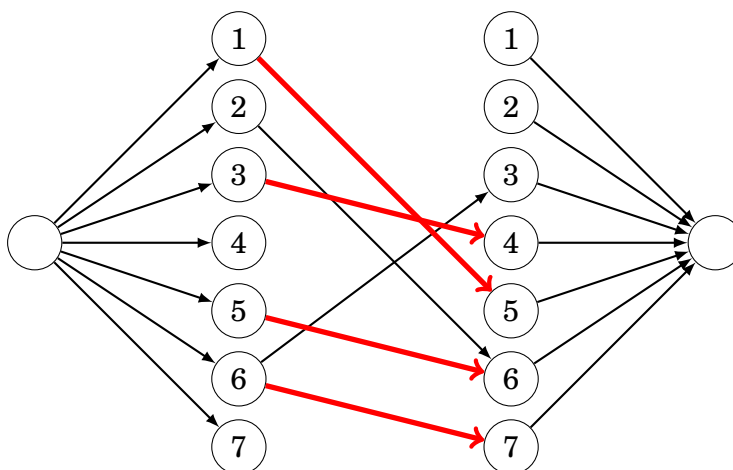
Tässä tapauksessa pienin solmuerillinen polkupeite muodostuu kolmesta polusta. Voimme valita polut esimerkiksi seuraavasti:



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:

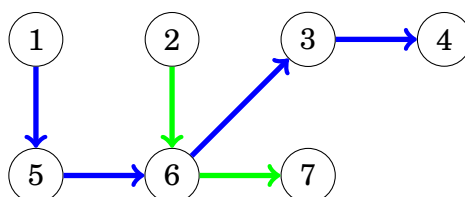


Tässä tapauksessa maksimiparitukseen kuuluu neljä kaarta, jotka vastaavat alkuperäisen verkon kaaria $1 \rightarrow 5$, $3 \rightarrow 4$, $5 \rightarrow 6$ ja $6 \rightarrow 7$. Niinpä pienin solmuerillinen polkupeite syntyy muodostamalla polut kyseisten kaarten avulla.

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 $7 - 4 = 3$.

Yleinen peite

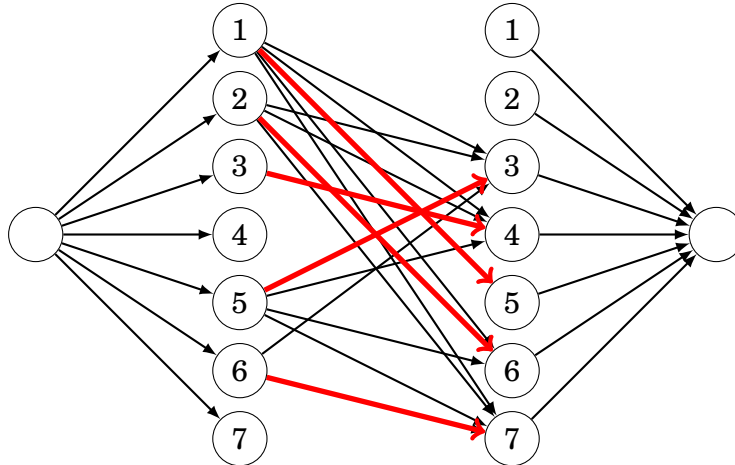
Yleisessä polkupeitteessä sama solmu voi kuulua moneen polkuun, minkä ansiosta tarvittava polkujen määrä saattaa olla pienempi. Esimerkkiverkossa pienin yleinen polkupeite muodostuu kahdesta polusta seuraavasti:



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).

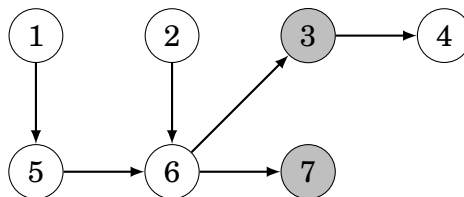
Nyt esimerkkiverkossa on seuraava tilanne:



Dilworthin lause

Dilworthin lauseen mukaan suunnatun, syklittömän verkon pienin yleinen polkupeite on yhtä suuri kuin suurin verkossa oleva **antiketju** eli kokoelma solmuja, jossa minkään kahden solmun välillä ei ole polkua.

Esimerkiksi äskeisessä verkossa pienin yleinen polkupeite sisältää kaksi polkua, joten verkon suurimmassa antiketjussa on kaksi solmua. Tällainen antiketju muodostuu esimerkiksi valitsemalla solmut 3 ja 7:



Verkossa ei ole polkua solmusta 3 solmuun 7 eikä polkua solmusta 7 solmuun 3, joten valinta on kelvallinen. Toisaalta jos verkosta valitaan mitkä tahansa kolme solmua, jostain solmusta toiseen on polku.

Osa III

Lisäaiheita

Luku 21

Lukuteoria

Lukuteoria on kokonaislukuja tutkiva matematiikan ala, jonka keskeinen käsite on lukujen jaollisuus. Lukuteoriassa on kiehtovaa, että monet kokonaislukuihin liittyvät kysymykset ovat hyvin vaikeita ratkaista, vaikka ne saattavat näyttää päältä päin yksinkertaisilta.

Tarkastellaan esimerkkinä seuraavaa yhtälöä:

$$x^3 + y^3 + z^3 = 33$$

On helppoa löytää kolme reaalilukua x , y ja z , jotka toteuttavat yhtälön. Voimme valita esimerkiksi

$$\begin{aligned}x &= 0, \\y &= 0, \\z &= \sqrt[3]{33}.\end{aligned}$$

Sen sijaan kukaan ei tiedä, onko olemassa kolmea kokonaislukua x , y ja z , jotka toteuttaisivat yhtälön, vaan kyseessä on avoin ongelma.

Tässä luvussa tutustumme lukuteorian peruskäsitteisiin ja -algoritmeihin. Lähdemme liikkeelle lukujen jaollisuudesta, johon liittyvät keskeiset algoritmit ovat alkuluvun tarkastaminen sekä luvun jakaminen tekijöihin.

21.1 Alkuluvut ja tekijät

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 positiiviset jakajat ovat 1 ja n . Esimerkiksi luvut 7, 19 ja 41 ovat alkulukuja. Luku 35 taas ei ole alkuluku, koska se voidaan jakaa tekijöihin $5 \cdot 7 = 35$. Jokaiselle luvulle $n > 1$ on olemassa yksikäsitteinen **alkutekijähajotelma**

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

missä p_1, p_2, \dots, p_k ovat alkulukuja ja $\alpha_1, \alpha_2, \dots, \alpha_k$ ovat positiivisia lukuja. Esimerkiksi luvun 84 alkutekijähajotelma on

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Luvun n **jakajien määrä** on

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

koska alkutekijän p_i kohdalla on $\alpha_i + 1$ tapaa valita, montako kertaa alkutekijä esiintyy jakajassa. Esimerkiksi luvun 84 jakajien määrä on $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. Jakajat ovat 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 ja 84.

Luvun n **jakajien summa** on

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

missä jälkimmäinen muoto perustuu geometriseen summaan. Esimerkiksi luvun 84 jakajien summa on

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

Luvun n **jakajien tulo** on

$$\mu(n) = n^{\tau(n)/2},$$

koska jakajista voi muodostaa $\tau(n)/2$ paria, joiden jokaisen tulona on n . Esimerkiksi luvun 84 jakajista muodostuu parit $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$, jne., ja jakajien tulo on $\mu(84) = 84^6 = 351298031616$.

Luku n on **täydellinen**, jos $n = \sigma(n) - n$ eli luku on yhtä suuri kuin summa sen jakajista välillä $1 \dots n - 1$. Esimerkiksi luku 28 on täydellinen, koska se muodostuu summasta $1 + 2 + 4 + 7 + 14$.

Alkulukujen määrä

On helppoa osoittaa, että alkulukuja on äärettömästi. Jos nimittäin alkulukuja olisi äärellinen määrä, voisimme muodostaa joukon $P = \{p_1, p_2, \dots, p_n\}$, joka sisältää kaikki alkuluvut. Esimerkiksi $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, jne.

Nyt kuitenkin voisimme muodostaa uuden alkuluvun

$$p_1 p_2 \cdots p_n + 1,$$

joka on kaikkia P :n lukuja suurempi. Koska tätä lukua ei ole joukossa P , syntyy ristiriita ja alkulukujen määrän on pakko olla ääretön.

Alkulukujen tiheys

Alkulukujen tiheys tarkoittaa, kuinka usein alkulukuja esiintyy muiden lukujen joukossa. Merkitään funktiolla $\pi(n)$, montako alkulukua on välillä $1 \dots n$. Esimerkiksi $\pi(10) = 4$, koska välillä $1 \dots 10$ on alkuluvut 2, 3, 5 ja 7.

On mahdollista osoittaa, että

$$\pi(n) \approx \frac{n}{\ln n},$$

mikä tarkoittaa, että alkulukuja esiintyy varsin usein. Esimerkiksi alkulukujen määrä välillä $1 \dots 10^6$ on $\pi(10^6) = 78498$ ja $10^6 / \ln 10^6 \approx 72382$.

Konjektuureja

Alkulukuihin liittyy useita *konjektuureja* eli lauseita, joiden uskotaan olevan tosia mutta joita kukaan ei ole onnistunut todistamaan tähän mennessä. Kuuluisia konjektuureja ovat seuraavat:

- **Goldbachin konjektuuri:** Jokainen parillinen kokonaisluku $n > 2$ voidaan esittää muodossa $n = a + b$ niin, että a ja b ovat alkulukuja.
- **Alkulukuparit:** On olemassa äärettömästi pareja muotoa $\{p, p + 2\}$, joissa sekä p että $p + 2$ on alkuluku.
- **Legendren konjektuuri:** Lukujen n^2 ja $(n + 1)^2$ välillä on aina alkuluku, kun n on mikä tahansa positiivinen kokonaisluku.

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, sekä myös selvittää ajassa $O(\sqrt{n})$ luvun alkutekijät.

Seuraava funktio alkuluku tutkii, 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 alkuluku(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 tekijat muodostaa vektorin, joka sisältää luvun n alkutekijät. 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> tekijat(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 vektoriin niin monta kertaa, kuin kyseinen alkutekijä jakaa luvun. Esimerkiksi $24 = 2^3 \cdot 3$, joten funktio muodostaa vektorin $[2, 2, 2, 3]$.

Eratostheneen seula

Eratostheneen seula on esilaskenta-algoritmi, jonka suorituksen jälkeen mistä tahansa välin $2 \dots n$ luvusta pystyy tarkastamaan nopeasti, onko se alkuluku, sekä etsimään yhden luvun alkutekijän, jos luku ei ole alkuluku.

Algoritmi luo taulukon a , jossa on käytössä indeksit $2, 3, \dots, n$. Taulukossa $a[k] = 0$ 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.

Algoritmi käy läpi välin $2 \dots n$ luvut yksi kerrallaan. Aina kun vastaan tulee uusi alkuluku x , niin algoritmi merkitsee taulukkoon, että x :n moninkerrat $2x, 3x, 4x, \dots$ eivät ole alkulukuja, koska niillä on alkutekijä x .

Esimerkiksi jos $n = 20$, 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 muodostaa 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/x kertaa tietyllä x :n arvolla, joten yläraja algoritmin ajankäytölle on harmoninen summa

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

Todellisuudessa algoritmi on vielä nopeampi, koska sisäsilmukka suoritetaan vain, jos luku x on alkuluku. Voidaan osoittaa, että algoritmin aikavaativuus on vain $O(n \log \log n)$ eli hyvin lähellä vaativuutta $O(n)$.

Eukleideen algoritmi

Lukujen a ja b **suurin yhteinen tekijä** eli $\text{syt}(a, b)$ on suurin luku, jolla sekä a että b on jaollinen. Lukujen a ja b **pienin yhteinen moninkerta** eli $\text{pym}(a, b)$ on puolestaan pienin luku, joka on jaollinen sekä a :lla että b :llä. Esimerkiksi $\text{syt}(24, 36) = 12$ ja $\text{pym}(24, 36) = 72$.

Suurimman yhteisen tekijän ja pienimmän yhteisen moninkerran välillä on yhteys

$$\text{pym}(a, b) = \frac{ab}{\text{syt}(a, b)}.$$

Eukleideen algoritmi on tehokas tapa etsiä suurin yhteinen tekijä. 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 algoritmin aikavaativuus on $O(\log n)$, kun $n = \min(a, b)$. Pahin tapaus algoritmille on, jos luvut ovat peräkkäiset Fibonaccin luvut. Silloin algoritmi käy läpi kaikki pienemmät peräkkäiset Fibonaccin luvut. Esimerkiksi

$$\text{syt}(13, 8) = \text{syt}(8, 5) = \text{syt}(5, 3) = \text{syt}(3, 2) = \text{syt}(2, 1) = \text{syt}(1, 0) = 1.$$

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 luvut 1, 5, 7 ja 11 ovat suhteellisia alkulukuja luvun 12:n kanssa.

Totienttifunktion arvon $\varphi(n)$ pystyy laskemaan luvun n alkutekijähajotelmasta kaavalla

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1).$$

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

Modulolaskennassa lukualuetta rajoitetaan niin, että käytössä ovat vain kokonaisluvut $0, 1, 2, \dots, m-1$, missä m on vakio. Ideana on, että lukua x kuvaa luku $x \bmod m$ eli luvun x jakojäännös luvulla m . Esimerkiksi jos $m = 17$, niin lukua 75 kuvaa luku $75 \bmod 17 = 7$.

Useissa laskutoimituksissa jakojäännöksen voi laskea ennen laskutoimitusta, minkä ansiosta saadaan seuraavat kaavat:

$$\begin{aligned} (x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\ (x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\ (x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\ (x^k) \bmod m &= (x \bmod m)^k \bmod m \end{aligned}$$

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.

Seuraava funktio laskee luvun $x^n \bmod m$:

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

Fermat'n pieni lause ja Eulerin lause

Fermat'n pienen lauseen mukaan

$$x^{m-1} \bmod m = 1,$$

kun m on alkuluku ja x ja m ovat suhteelliset alkuluvut. Tällöin myös

$$x^k \bmod m = (x^{k \bmod (m-1)}) \bmod m.$$

Yleisemmin **Eulerin lauseen** mukaan

$$x^{\varphi(m)} \bmod m = 1,$$

kun x ja m ovat suhteelliset alkuluvut. Fermat'n pieni lause seuraa Eulerin lauseesta, koska jos m on alkuluku, niin $\varphi(m) = m - 1$.

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, koska jakolasku luvulla x vastaa kertolaskua luvulla x^{-1} . Esimerkiksi jos haluamme laskea jakolaskun $36/6 \bmod 17$, voimme muuttaa sen muotoon $2 \cdot 3 \bmod 17$, koska $36 \bmod 17 = 2$ ja $6^{-1} \bmod 17 = 3$.

Modulon käänteislukua ei kuitenkaan ole aina olemassa. Esimerkiksi jos $x = 2$ ja $m = 4$, yhtälölle

$$xx^{-1} \bmod m = 1.$$

ei ole ratkaisua, koska kaikki luvun 2 moninkerrat ovat parillisia eikä jakojäännös 4:llä voi koskaan olla 1. Osoittautuu, että $x^{-1} \bmod m$ on olemassa tarkalleen silloin, kun x ja m ovat suhteelliset alkuluvut.

Jos modulon käänteisluku on olemassa, sen saa laskettua kaavalla

$$x^{-1} = x^{\varphi(m)-1}.$$

Erityisesti jos m on alkuluku, kaavasta tulee

$$x^{-1} = x^{m-2}.$$

Esimerkiksi jos $x = 6$ ja $m = 17$, niin

$$x^{-1} = 6^{17-2} \bmod 17 = 3.$$

Tämän kaavan ansiosta modulon käänteisluvun pystyy laskemaan nopeasti tehokkaan potenssilaskun avulla.

Modulon käänteisluvun kaavan voi perustella Eulerin lauseen avulla. Ensinnäkin käänteisluvulle täytyy päteä

$$xx^{-1} \bmod m = 1.$$

Toisaalta Eulerin lauseen mukaan

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

joten lukujen x^{-1} ja $x^{\varphi(m)-1}$ on oltava samat.

Modulot tietokoneessa

Tietokone käsittelee etumerkittömiä kokonaislukuja modulo 2^k , missä k on luvun bittien määrä. Usein näkyvä seuraus tästä on luvun arvon pyörähtäminen ympäri, jos luku kasvaa liian suureksi.

Esimerkiksi C++:ssa `unsigned int` -tyyppinen arvo lasketaan modulo 2^{32} . Seuraava koodi määrittelee muuttujan tyyppiä `unsigned int`, joka saa arvon 123456789. Sitten muuttujan arvo kerrotaan itsellään, jolloin tuloksena on luku $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3 Yhtälönratkaisu

Diofantoksen yhtälö on muotoa

$$ax + by = c,$$

missä a , b ja c ovat vakioita ja tehtävänä on ratkaista muuttujat x ja y . Jokaisen yhtälössä esiintyvän luvun tulee olla kokonaisluku. 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

$$ax + by = \text{syt}(a, b).$$

Diofantoksen 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

$$\left(x + \frac{kb}{\text{syt}(a, b)}, y - \frac{ka}{\text{syt}(a, b)}\right)$$

on ratkaisu, missä k on mikä tahansa kokonaisluku.

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

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

Tässä ratkaisussa jokaiselle luvulle $k = 1, 2, \dots, n$ pätee, että

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

sillä

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Koska kaikki muut summan osat ovat jaollisia luvulla m_k , ne eivät vaikuta jakojäännökseen ja koko summan jakojäännös m_k :lla on a_k .

Esimerkiksi yhtälöryhmän

$$\begin{aligned}x &= 3 \bmod 5 \\x &= 4 \bmod 7 \\x &= 2 \bmod 3\end{aligned}$$

ratkaisu on

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Kun luku x on yhtälöryhmän ratkaisu, niin myös kaikki luvut muotoa

$$x + m_1 m_2 \cdots m_n$$

ovat yhtälöryhmän ratkaisuja.

21.4 Muita tuloksia

Lagrangen lause

Lagrangen lauseen mukaan jokainen positiivinen kokonaisluku voidaan esittää neljän neliöluvun summana eli muodossa $a^2 + b^2 + c^2 + d^2$. Esimerkiksi luku 123 voidaan esittää muodossa $8^2 + 5^2 + 5^2 + 3^2$.

Huomaa, että kaikkia positiivisia lukuja ei voi esittää muodossa $a^2 + b^2 + c^2$: esimerkiksi luvun 7 esittäminen vaatii neljä neliölukua.

Zeckendorfin lause

Zeckendorfin lauseen mukaan jokainen positiivinen kokonaisluku voidaan esittää Fibonaccin lukujen summana niin, että mitkään kaksi lukua eivät ole samat eivätkä peräkkäiset Fibonaccin luvut. Esimerkiksi luku 74 voidaan esittää muodossa $55 + 13 + 5 + 1$.

Pythagoraan kolmikot

Pythagoraan kolmikko on lukukolmikko (a, b, c) , joka toteuttaa Pythagoraan lauseen $a^2 + b^2 = c^2$ eli a , b ja c voivat olla suorakulmaisen kolmion sivujen pituudet. Esimerkiksi $(3, 4, 5)$ on Pythagoraan kolmikko.

Jos (a, b, c) on Pythagoraan kolmikko, niin myös kaikki kolmikot muotoa (ka, kb, kc) ovat Pythagoraan kolmikoita, missä $k > 1$. Pythagoraan kolmikko on **primitiivinen**, jos a , b ja c ovat suhteellisia alkulukuja, ja primitiivisistä kolmikoista voi muodostaa kaikki muut kolmikot kertoimen k avulla.

Eukleideen kaava on tehokas menetelmä muodostaa Pythagoraan kolmikoita. Sen mukaan jokainen primitiivinen Pythagoraan kolmikko on muotoa

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

missä $0 < m < n$, n ja m ovat suhteelliset alkuluvut ja ainakin toinen luvuista n ja m on parillinen. Esimerkiksi valitsemalla $m = 1$ ja $n = 2$ syntyy pienin mahdollinen Pythagoraan kolmikko

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

Wilsonin lause

Wilsonin lauseen mukaan luku n on alkuluku tarkalleen silloin, kun

$$(n - 1)! \bmod n = n - 1.$$

Esimerkiksi luku 11 on alkuluku, koska

$$10! \bmod 11 = 10$$

ja luku 12 ei ole alkuluku, koska

$$11! \bmod 12 = 0 \neq 11.$$

Wilsonin lauseen avulla voi siis tutkia, onko luku alkuluku. Tämä ei ole kuitenkaan käytännössä hyvä tapa, koska luvun $(n - 1)!$ laskeminen on työlästä, jos n on suuri luku.

Luku 22

Kombinatoriikka

Kombinatoriikka tarkoittaa yhdistelmien määrän laskemista. Yleensä tavoitteena on toteuttaa laskenta tehokkaasti niin, että jokaista yhdistelmää ei tarvitse muodostaa erikseen, koska yhdistelmien määrä on usein suuri.

Tarkastellaan esimerkkinä tehtävää, jossa laskettavana on, monellako tavalla luvun n voi esittää positiivisten kokonaislukujen summana. Esimerkiksi luvun 4 voi esittää 8 tavalla seuraavasti:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- 4

Kombinatorisen tehtävän ratkaisun voi usein laskea rekursiivisen funktion avulla. Tässä tapauksessa voimme määritellä funktion $f(n)$, joka laskee luvun n esitystapojen määrän. Esimerkiksi $f(4) = 8$ yllä olevan esimerkin mukaisesti. Funktion voi laskea rekursiivisesti seuraavasti:

$$f(n) = \begin{cases} 1 & n = 1 \\ f(1) + f(2) + \dots + f(n-1) + 1 & n > 1 \end{cases}$$

Pohjatapauksena on $f(1) = 1$, koska luvun 1 voi esittää vain yhdellä tavalla. Rekursiivinen tapaus käy läpi kaikki vaihtoehdot, mikä on summan viimeinen luku. Esimerkiksi tapauksessa $n = 4$ summa voi päättyä $+1$, $+2$ tai $+3$. Tämän lisäksi lasketaan mukaan esitystapa, jossa on pelkkä luku n .

Funktion ensimmäiset arvot ovat:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \\ f(5) &= 16 \end{aligned}$$

Osoittautuu, että 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.

22.1 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 alkiota x . Jos alkio x valitaan osajoukkoon, täytyy vielä valita $n - 1$ alkiosta $k - 1$ alkiota. Jos taas alkiota x ei valita osajoukkoon, täytyy vielä valita $n - 1$ alkiosta k alkiota.

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 alkiot 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 alkiota osajoukkoon. Koska ei ole merkitystä, missä järjestyksessä osajoukon alkiot ja ulkopuoliset alkiot 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

$$(a + b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \dots + \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^0 b^n.$$

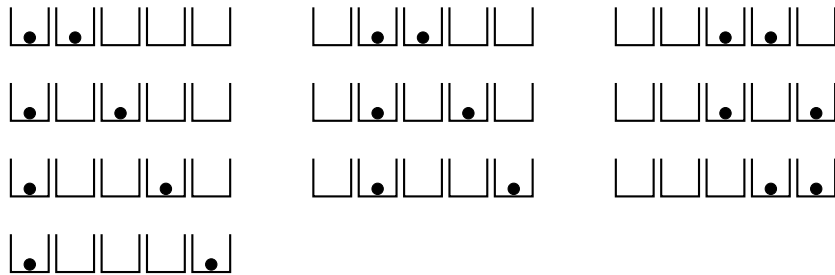
Binomikertoimet esiintyvät myös Pascalin kolmiossa, jonka reunoilla on lukua 1 ja jokainen luku saadaan kahden yllä olevan luvun summana:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & & \dots & & \dots & & \dots & & \dots \end{array}$$

Laatikot ja pallot

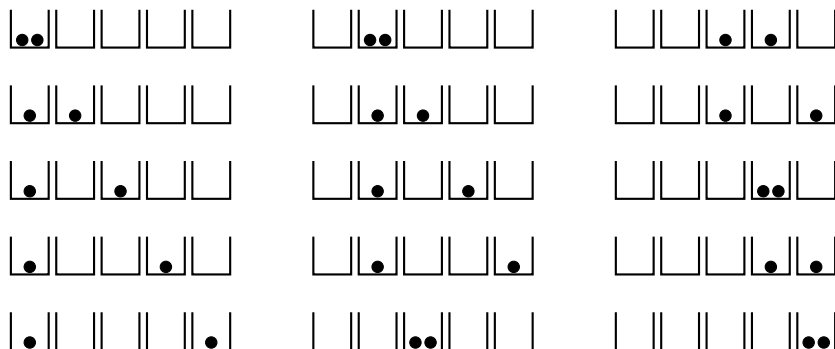
”Laatikot ja pallot” on usein hyödyllinen malli, jossa n laatikkoon sijoitetaan k palloa. Tarkastellaan seuraavaksi kolmea tapausta:

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 $\binom{n}{k}$.

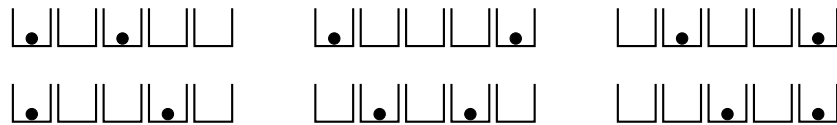
Tapaus 2: Samaan laatikkoon saa sijoittaa monta palloa. Esimerkiksi kun $n = 5$ ja $k = 2$, sijoitustapoja on 15:



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**

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

missä $k_1 + k_2 + \cdots + k_m = n$. Multinomikerroin ilmaisee, monellako tavalla n alkiota voidaan jakaa osajoukkoihin, joiden koot ovat k_1, k_2, \dots, k_m . Jos $m = 2$, multinomikertoimen kaava vastaa binomikertoimen kaavaa.

22.2 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 on mahdollista muodostaa seuraavat kelvolliset sulkulausekkeet:

- $()()()$
- $((()))$
- $()(())$
- $((()))$
- $((())())$

Sulkulausekkeet

Mikä sitten tarkkaan ottaen on *kelvollinen sulkulauseke*? Seuraavat säännöt kuvailevat täsmällisesti kaikki kelvolliset sulkulausekkeet:

- Sulkulauseke $()$ on kelvollinen.
- Jos sulkulauseke A on kelvollinen, niin myös sulkulauseke (A) on kelvollinen.
- Jos sulkulausekkeet A ja B ovat kelvollisia, niin myös sulkulauseke AB on kelvollinen.

Toinen tapa luonnehtia kelvollista sulkulauseketta on, että jos valitaan mikä tahansa lausekkeen alkuosa, niin alkusulkuja on ainakin yhtä monta kuin loppusulkuja. Lisäksi koko lausekkeessa tulee olla tarkalleen yhtä monta alkusulkua ja loppusulkua.

Laskutapa 1

Catalanin lukuja voi laskea rekursiivisesti kaavalla

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

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 kelvollinen.

Jos sulkulauseke ei ole kelvollinen, 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 $n + 1$ alkusulkuja ja $n - 1$ loppusulkuja. Tällaisia lausekkeita on kaikkiaan $\binom{2n}{n+1}$, joka on sama kuin ei-kelvollisten sulkulausekkeiden määrä. Niinpä kelvollisten sulkulausekkeiden määrä voidaan laskea kaavalla

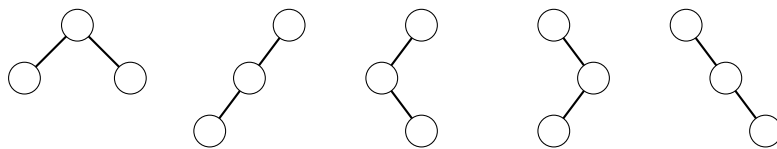
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Puiden laskeminen

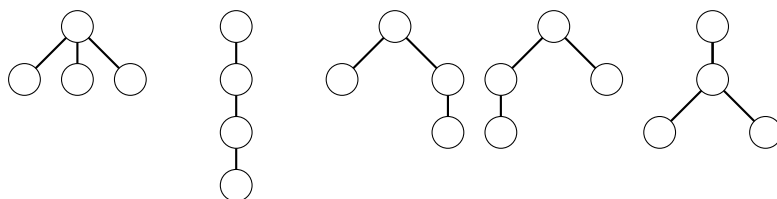
Catalanin luvut kertovat myös juurellisten puiden lukumääriä:

- 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

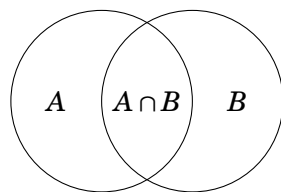


22.3 Inklusio-ekskluusio

Inklusio-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:

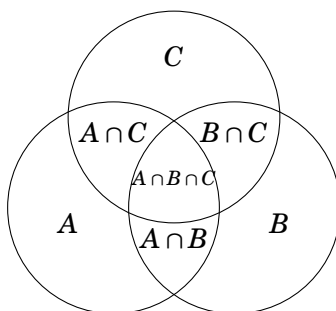


Tavoitteena on laskea, kuinka suuri on yhdiste $A \cup B$ eli alue, joka on toisen tai kummankin ympyrän sisällä. Kuvan mukaisesti yhdisteen $A \cup B$ koko saadaan laskemalla ensin yhteen ympyröiden A ja B koot ja vähentämällä siitä sitten leikkauksen $A \cap B$ koko.

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

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Epäjärjestykset

Lasketaan esimerkkinä, montako tapaa on muodostaa luvuista $(1, 2, \dots, n)$ **epäjärjestys** eli permutaatio, jossa mikään luku ei ole alkuperäisellä paikallaan. Esimerkiksi jos $n = 3$, niin epäjärjestyksiä on kaksi: $(2, 3, 1)$ ja $(3, 1, 2)$.

Yksi tapa lähestyä tehtävää on käyttää inklusio-ekskluusiota. 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 epäjärjestyksen määrä on

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

eli riittää laskea joukkojen yhdisteen koko. Tämä palautuu inklusio-ekskluusion 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-ekskluusiota. Merkitään $f(n)$:llä jonon $(1, 2, \dots, n)$ epäjärjestyksen 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.

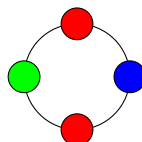
22.4 Burnsiden lemma

Burnsiden lemma laskee yhdistelmien määrän niin, että symmetrisistä yhdistelmistä lasketaan mukaan vain yksi edustaja. Burnsiden lemmän mukaan yhdistelmien määrä on

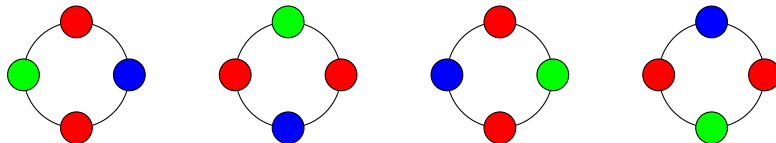
$$\sum_{k=1}^n \frac{c(k)}{n},$$

missä yhdistelmän asentoa voi muuttaa n tavalla ja $c(k)$ on niiden yhdistelmien määrä, jotka pysyvät ennallaan, kun asentoa muutetaan tavalla k .

Lasketaan esimerkkinä, montako erilaista tapaa on muodostaa n helmen helminauha, kun kunkin helmen värin tulee olla väliltä $1, 2, \dots, m$. 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

$$\sum_{i=0}^{n-1} \frac{m^{\text{sy}(i,n)}}{n}.$$

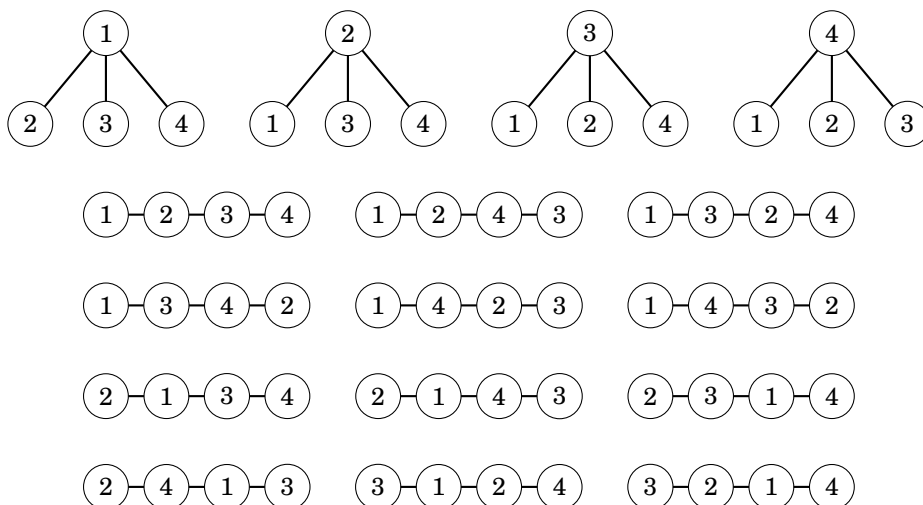
Esimerkiksi kun helminauhan pituus on 4 ja värejä on 3, helminauhoja on

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 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 $4^{4-2} = 16$:

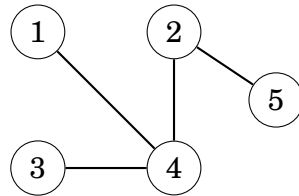


Seuraavaksi näemme, miten Cayleyn kaavan voi perustella samastamalla numeroidut puut Prüfer-koodeihin.

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 koodiin.

Esimerkiksi 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-koodeja eli n^{n-2} .

Luku 23

Matriisit

Matriisi on kaksiulotteista taulukkoa vastaava matemaattinen käsite, jolle on määritelty laskutoimituksia. Esimerkiksi

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

on matriisi, jossa on 3 riviä ja 4 saraketta eli se on kokoa 3×4 . Matriisin alkioihin on mahdollista viitata merkinnällä $[i, j]$, jossa i on rivi ja j on sarake. Esimerkiksi yllä olevassa matriisissa $A[2, 3] = 8$ ja $A[3, 1] = 9$.

Matriisin erikoistapaus on **vektori**, joka on kokoa $n \times 1$ oleva yksiulotteinen matriisi. Esimerkiksi

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

on vektori, jossa on 3 alkia.

Matriisin A **transpoosi** A^T syntyy, kun matriisin rivit ja sarakkeet vaihdetaan keskenään eli $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Matriisi on **neliömatriisi**, jos sen korkeus ja leveys ovat samat. Esimerkiksi seuraava matriisi on neliömatriisi:

$$A = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Laskutoimitukset

Matriisien A ja B summa $A + B$ on määritelty, jos matriisit ovat yhtä suuret. Tuloksena oleva matriisi on samaa kokoa kuin matriisit A ja B ja sen jokainen alkio on vastaavissa kohdissa olevien alkioiden summa.

Esimerkiksi

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Matriisin A kertominen luvulla x tarkoittaa, että jokainen matriisin alkio kerrotaan luvulla x .

Esimerkiksi

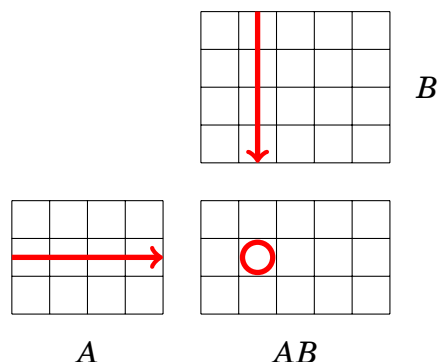
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Matriisitulo

Matriisien A ja B tulo AB on määritelty, jos matriisi A on kokoa $a \times n$ ja matriisi B on kokoa $n \times b$ eli matriisin A leveys on sama kuin matriisin B korkeus. Tuloksena oleva matriisi on kokoa $a \times b$ ja sen alkiot lasketaan kaavalla

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j].$$

Kaavan tulkintana on, että kukin AB :n alkio saadaan summana, joka muodostuu A :n ja B :n alkioparien tuloista seuraavan kuvan mukaisesti:



Esimerkiksi

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Matriisitulo ei ole vaihdannainen, eli ei ole voimassa $A \cdot B = B \cdot A$. Kuitenkin matriisitulo on liitännäinen, eli on voimassa $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.

Ykkösmatriisi on neliömatriisi, jonka lävistäjän jokainen alkio on 1 ja jokainen muu alkio on 0. Esimerkiksi 3×3 -yksikkömatriisi on seuraavanlainen:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Ykkösmatriisilla kertominen säilyttää matriisin ennallaan. Esimerkiksi

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{ja} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Kahden $n \times n$ kokoisen matriisin tulon laskeminen vie aikaa $O(n^3)$ käyttäen suoraviivaista algoritmia. Myös nopeampia algoritmeja on olemassa: tällä hetkellä nopein tunnettu algoritmi vie aikaa $O(n^{2,37})$. Tällaiset algoritmit eivät kuitenkaan ole tarpeen kisakoodauksessa.

Matriisipotenssi

Matriisin A potenssi A^k on määritelty, jos A on neliömatriisi. Määritelmä nojautuu kertolaskuun:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ kertaa}}$$

Esimerkiksi

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Lisäksi A^0 tuottaa ykkösmatriisin. Esimerkiksi

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Matriisin A^k voi laskea tehokkaasti ajassa $O(n^3 \log k)$ soveltamalla luvun 21.2 tehokasta potenssilaskua. Esimerkiksi

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

Determinantti

Matriisin A **determinantti** $\det(A)$ on määritelty, jos A on neliömatriisi. Jos A on kokoa 1×1 , niin $\det(A) = A[1, 1]$. Suuremmalle matriisille determinantaatti lasketaan rekursiivisesti kaavalla

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

missä $C[i, j]$ on matriisin A **kofaktori** kohdassa $[i, j]$. Kofaktori lasketaan puolestaan kaavalla

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

missä $M[i, j]$ on matriisi A , josta on poistettu rivi i ja sarake j . Kofaktorissa olevan kertoimen $(-1)^{i+j}$ ansiosta joka toinen determinantti lisätään summaan positiivisena ja joka toinen negatiivisena.

Esimerkiksi

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

ja

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

Determinantti kertoo, onko matriisille A olemassa **käänteismatriisia** A^{-1} , jolle pätee $A \cdot A^{-1} = I$, missä I on ykkösmatriisi. Osoittautuu, että A^{-1} on olemassa tarkalleen silloin, kun $\det(A) \neq 0$, ja sen voi laskea kaavalla

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Esimerkiksi

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 Lineaariset rekursioyhtälöt

Lineaarinen rekursioyhtälö voidaan esittää funktiona $f(n)$, jolle on annettu alkuarvot $f(0), f(1), \dots, f(k-1)$ ja jonka suuremmat arvot parametrissa k lähtien lasketaan rekursiivisesti kaavalla

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

missä c_1, c_2, \dots, c_k ovat vakiokertoimia.

Funktion arvon $f(n)$ voi laskea dynaamisella ohjelmoinnilla ajassa $O(kn)$ laskemalla kaikki arvot $f(0), f(1), \dots, f(n)$ järjestyksessä. Tätä ratkaisua voi kuitenkin tehostaa merkittävästi matriisien avulla, kun k on pieni. Seuraavaksi näemme, miten arvon $f(n)$ voi laskea ajassa $O(k^3 \log n)$.

Fibonaccin luvut

Yksinkertainen esimerkki lineaarisesta rekursioyhtälöstä on Fibonaccin luvut määrittelevä funktio:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Tässä tapauksessa $k = 2$ ja $c_1 = c_2 = 1$.

Ideana on esittää Fibonaccin lukujen laskukaava 2×2 -kokoisena neliömatriisina X , jolle pätee

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

eli X :lle annetaan ”syötteenä” arvot $f(i)$ ja $f(i+1)$, ja X muodostaa niistä arvot $f(i+1)$ ja $f(i+2)$. Osoittautuu, että tällainen matriisi on

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Esimerkiksi

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Tämän ansiosta arvon $f(n)$ sisältävän matriisin saa laskettua kaavalla

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Potenssilasku X^n on mahdollista laskea ajassa $O(k^3 \log n)$, joten myös funktion arvon $f(n)$ saa laskettua ajassa $O(k^3 \log n)$.

Yleinen tapaus

Tarkastellaan sitten yleistä tapausta, missä $f(n)$ on mikä tahansa lineaarinen rekursioyhtälö. Nyt tavoitteena on etsiä matriisi X , jolle pätee

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Tällainen matriisi on

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

Matriisin $k-1$ ensimmäisen rivin jokainen alkio on 0, paitsi yksi alkio on 1. Nämä rivit kopioivat arvon $f(i+1)$ arvon $f(i)$ tilalle, arvon $f(i+2)$ arvon $f(i+1)$ tilalle jne. Viimeinen rivi sisältää rekursiokaavan kertoimet, joiden avulla muodostuu uusi arvo $f(i+k)$.

Nyt arvon $f(n)$ pystyy laskemaan ajassa $O(k^3 \log n)$ kaavalla

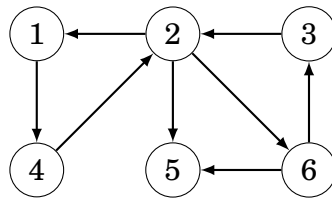
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3 Verkot ja matriisit

Polkujen määrä

Matriisipotenssilla on mielenkiintoinen vaikutus verkon vierusmatriisiin sisältöön. Kun V on painottoman verkon vierusmatriisi, niin V^n kertoo, montako n kaaren pituista polkua eri solmuista on toisiinsa.

Esimerkiksi verkon



vierusmatriisi on

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Esimerkiksi matriisi

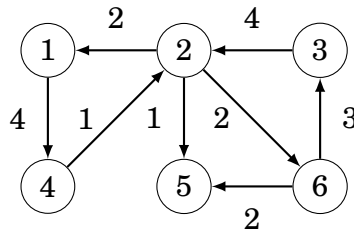
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

kertoo, montako 4 kaaren pituista polkua solmuista on toisiinsa. Esimerkiksi $V^4[2,5] = 2$, koska solmusta 2 solmuun 5 on olemassa 4 kaaren pituiset polut $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ ja $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Lyhimmät polut

Samantapaisella idealla voi laskea painotetussa verkossa kullekin solmuparille, mikä on lyhin n kaaren pituinen polku solmujen välillä. Tämä vaatii matriisitulon määritelmän muuttamista niin, että siinä ei lasketa polkujen yhteismäärää vaan minimoidaan polun pituutta.

Tarkastellaan esimerkkinä seuraavaa verkkoa:



Muodostetaan verkosta vierusmatriisi, jossa arvo ∞ tarkoittaa, että kaarta ei ole, ja muut arvot ovat kaarten pituuksia. Matriisista tulee

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Nyt ideana on laskea matriisitulo kaavan

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

sijasta kaavalla

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j],$$

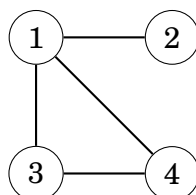
eli summa muuttuu minimiksi ja tulo summaksi. Tämän seurauksena matriisipotenssi selvittää lyhimät polkujen pituudet solmujen välillä. Esimerkiksi

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix}$$

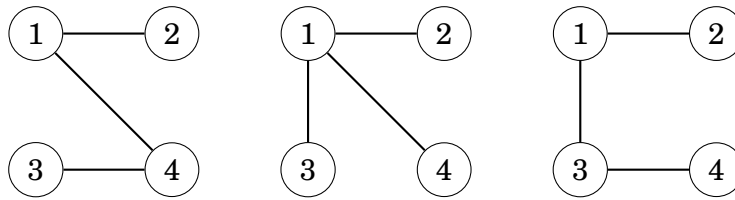
eli esimerkiksi lyhin 4 kaaren pituinen polku solmusta 2 solmuun 5 on pituudeltaan 8. Tämä polku on $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

Kirchhoffin lause

Kirchhoffin lause laskee verkon virittävän puiden määrän verkosta muodostetun matriisin determinantin avulla. Esimerkiksi verkolla



on kolme virittävää puuta:



Muodostetaan verkosta **Laplacen matriisi** L , jossa $L[i, i]$ on solmun i aste ja $L[i, j] = -1$, jos solmujen i ja j välillä on kaari, ja muuten $L[i, j] = 0$. Tässä tapauksessa matriisista tulee

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}.$$

Nyt virittävien puiden määrä on determinantti matriisista, joka saadaan poistamasta matriisista L jokin rivi ja jokin sarake. Esimerkiksi jos poistamme ylimmän rivin ja vasemman sarakkeen, tuloksena on

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

Determinantista tulee aina sama riippumatta siitä, mikä rivi ja sarake matriisista L poistetaan.

Huomaa, että Kirchhoffin lauseen erikoistapauksena on luvun 22.5 Cayleyn kaava, koska voidaan osoittaa, että täydellisessä n solmun verkossa

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

Luku 24

Todennäköisyys

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 Laskutavat

Todennäköisyyden laskemiseen on kaksi tavallista menetelmää. Tarkastellaan esimerkkinä tavallista 52 kortin korttipakkaa. Kortit jakaantuvat neljään maahan (hertta, risti, ruutu, pata) ja kussakin maassa korttien arvot ovat $1, 2, \dots, 13$. Lasketaan todennäköisyys sille, että kun sekoitetusta pakasta nostetaan kolme päällimmäistä korttia, niin 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. Vaatimuksena on, 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

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 Tapahtumat

Todennäköisyyden tapahtuma voidaan esittää joukkona

$$A \subset X,$$

missä X sisältää kaikki mahdolliset alkeistapaukset ja A on jokin alkeistapausten osajoukko. Esimerkiksi nopanheitossa alkeistapaukset ovat

$$X = \{x_1, x_2, x_3, x_4, x_5, x_6\},$$

missä x_k tarkoittaa silmälukua k . Nyt esimerkiksi tapahtumaa ”silmäluku on parillinen” vastaa joukko

$$A = \{x_2, x_4, x_6\}.$$

Jokaista alkeistapausta x vastaa todennäköisyys $p(x)$. Tämän ansiosta joukkoa A vastaavan tapahtuman todennäköisyys $P(A)$ voidaan laskea alkeistapausten todennäköisyyksien summana kaavalla

$$P(A) = \sum_{x \in A} p(x).$$

Esimerkiksi nopanheitossa $p(x) = 1/6$ jokaiselle alkeistapaukselle x , joten tapahtuman ”silmäluku on parillinen” todennäköisyys on

$$p(x_2) + p(x_4) + p(x_6) = 1/2.$$

Alkeistapahtumat tulee aina valita niin, että kaikkien alkeistapausten todennäköisyyksien summa on 1 eli $P(X) = 1$.

Koska todennäköisyyden tapahtumat ovat joukkoja, niihin voi soveltaa joukko-opin operaatioita:

- **Komplementti** \bar{A} tarkoittaa tapahtumaa ” A ei tapahdu”. Esimerkiksi nopanheitossa tapahtuman $A = \{x_2, x_4, x_6\}$ komplementti on $\bar{A} = \{x_1, x_3, x_5\}$.
- **Yhdiste** $A \cup B$ tarkoittaa tapahtumaa ” A tai B tapahtuu”. Esimerkiksi tapahtumien $A = \{x_2, x_5\}$ ja $B = \{x_4, x_5, x_6\}$ yhdiste on $A \cup B = \{x_2, x_4, x_5, x_6\}$.
- **Leikkaus** $A \cap B$ tarkoittaa tapahtumaa ” A ja B tapahtuvat”. Esimerkiksi tapahtumien $A = \{x_2, x_5\}$ ja $B = \{x_4, x_5, x_6\}$ leikkaus on $A \cap B = \{x_5\}$.

Komplementti

Komplementin \bar{A} todennäköisyys lasketaan kaavalla

$$P(\bar{A}) = 1 - P(A).$$

Joskus tehtävän ratkaisu on kätevää laskea komplementin kautta miettimällä tilannetta käänteisesti. Esimerkiksi todennäköisyys saada silmäluku 6 ainakin kerran, kun noppaa heitetään kymmenen kertaa, on

$$1 - (5/6)^{10}.$$

Tässä $5/6$ on todennäköisyys, että yksittäisen heiton silmäluku ei ole 6, ja $(5/6)^{10}$ on todennäköisyys, että yksikään silmäluku ei ole 6 kymmenessä heitossa. Tämän komplementti tuottaa halutun tuloksen.

Yhdiste

Yhdisteen $A \cup B$ todennäköisyys lasketaan kaavalla

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Esimerkiksi nopanheitossa tapahtumien

$$A = \text{”silmäluku on parillinen”}$$

ja

$$B = \text{”silmäluku on alle 4”}$$

yhdisteen

$$A \cup B = \text{”silmäluku on parillinen tai alle 4”}$$

todennäköisyys on

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 2/3 - 1/3 = 5/6.$$

Jos tapahtumat A ja B ovat erilliset eli $A \cap B$ on tyhjä, yhdisteen $A \cup B$ todennäköisyys on yksinkertaisesti

$$P(A \cup B) = P(A) + P(B).$$

Ehdollinen todennäköisyys

Ehdollinen todennäköisyys

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

on tapahtuman A todennäköisyys olettaen, että tapahtuma B tapahtuu. Tällöin todennäköisyyden laskennassa otetaan huomioon vain ne alkeistapaukset, jotka kuuluvat joukkoon B .

Äskeisen esimerkin joukkoja käyttäen

$$P(A|B) = 1/3,$$

koska joukon B alkeistapaukset ovat $\{x_1, x_2, x_3\}$ ja niistä yhdessä silmäluku on parillinen. Tämä on todennäköisyys saada parillinen silmäluku, jos tiedetään, että silmäluku on välillä $1 \dots 3$.

Leikkaus

Ehdollisen todennäköisyyden avulla leikkauksen $A \cap B$ todennäköisyys voidaan laskea kaavalla

$$P(A \cap B) = P(A)P(B|A).$$

Tapahtumat A ja B ovat **riippumattomat**, jos

$$P(A|B) = P(A),$$

jolloin B :n tapahtuminen ei vaikuta A :n todennäköisyyteen ja päinvastoin. Tässä tapauksessa leikkauksen todennäköisyys on

$$P(A \cap B) = P(A)P(B).$$

Esimerkiksi pelikortin nostamisessa tapahtumat

$$A = \text{"kortin maa on risti"}$$

ja

$$B = \text{"kortin arvo on 4"}$$

ovat riippumattomat. Niinpä tapahtuman

$$A \cap B = \text{"kortti on ristinelonen"}$$

todennäköisyys on

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3 Satunnaismuuttuja

Satunnaismuuttuja on arvo, joka syntyy satunnaisen prosessin tuloksena. Satunnaismuuttujaa merkitään yleensä suurella kirjaimella. Esimerkiksi kahden nopan heitossa yksi mahdollinen satunnaismuuttuja on

$$X = \text{"silmälukujen summa"}.$$

Esimerkiksi jos heitot ovat $(4, 6)$, niin X saa arvon 10.

Merkintä $P(X = x)$ tarkoittaa todennäköisyyttä, että satunnaismuuttujan X arvo on x . Edellisessä esimerkissä $P(X = 10) = 3/36$, koska erilaisia heittotapoja on 36 ja niistä summan 10 tuottavat heitot $(4, 6)$, $(5, 5)$ ja $(6, 4)$.

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 nopan heitossa silmäluvun odotusarvo on

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

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

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Tarkastellaan sitten tehtävää, jossa n laatikkoon sijoitetaan satunnaisesti n palloa ja laskettavana on odotusarvo, montako laatikkoa jää tyhjäksi. Kulakin pallolla on yhtä suuri todennäköisyys päätyä mihin tahansa laatikkoon. Esimerkiksi jos $n = 2$, niin vaihtoehdot ovat seuraavat:



Tässä tapauksessa odotusarvo tyhjien laatikoiden määrälle on

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

Yleisessä tapauksessa todennäköisyys, että yksittäinen hattu on tyhjä, on

$$\left(\frac{n-1}{n}\right)^n,$$

koska mikään pallo ei saa mennä sinne. Niinpä odotusarvon lineaarisuuden ansiosta tyhjien hattujen määrän odotusarvo on

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Jakaumat

Satunnaismuuttujan **jakauma** kertoo, millä todennäköisyydellä satunnaismuuttuja saa minkäkin arvon. Jakauma muodostuu arvoista $P(X = x)$. Esimerkiksi kahden nopan heitossa silmälukujen summan jakauma on:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Tutustumme seuraavaksi muutamaan usein esiintyvään jakaumaan.

Tasajakauman satunnaismuuttuja saa arvoja väliltä $a \dots b$ ja jokaisen arvon todennäköisyys on sama. Esimerkiksi yhden nopan heitto tuottaa tasajakau-
man, jossa $P(X = x) = 1/6$, kun $x = 1, 2, \dots, 6$.

Tasajakaumassa X :n odotusarvo on

$$E[X] = \frac{a+b}{2}.$$

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 $(1/6)^3(5/6)^7 \binom{10}{3}$.

Binomijakaumassa X :n odotusarvo on

$$E[X] = pn.$$

Geometrisen 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 $(5/6)^3 1/6$.

Geometrisessa jakaumassa X :n odotusarvo on

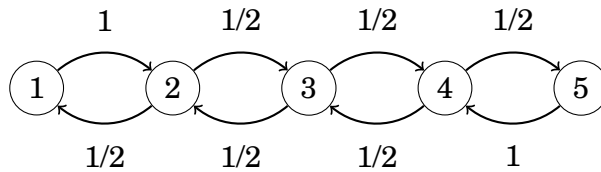
$$E[X] = \frac{1}{p}.$$

24.4 Markovin ketju

Markovin ketju on satunnaisprosessi, joka muodostuu tiloista ja niiden välisistä siirtymistä. Jokaisesta tilasta tiedetään, millä todennäköisyydellä siitä siirrytään toisiin tiloihin. Markovin ketju voidaan esittää verkkona, jonka solmut ovat tiloja ja kaaret niiden välisiä siirtymiä.

Tarkastellaan esimerkkinä tehtävää, jossa olet alussa n -kerroksisen rakennuksen kerroksessa 1. Joka askeleella liikut satunnaisesti kerroksen ylöspäin tai alaspäin, paitsi kerroksesta 1 liikut aina ylöspäin ja kerroksesta n aina alaspäin. Mikä on todennäköisyys, että olet m askeleen jälkeen kerroksessa k ?

Tehtävässä kukin rakennuksen kerros on yksi tiloista, ja kerrosten välillä liikutaan satunnaisesti. Esimerkiksi jos $n = 5$, verkosta tulee:



Markovin ketjun tilajakauma on vektori $[p_1, p_2, \dots, p_n]$, missä p_k tarkoittaa todennäköisyyttä olla tällä hetkellä tilassa k . Todennäköisyyksille pätee aina $p_1 + p_2 + \dots + p_n = 1$.

Esimerkissä jakauma on ensin $[1, 0, 0, 0, 0]$, koska on varmaa, että kulku alkaa kerroksesta 1. Seuraava jakauma on $[0, 1, 0, 0, 0]$, koska kerroksesta 1 pääsee vain kerrokseen 2. Tämän jälkeen on mahdollisuus mennä joko ylöspäin tai alaspäin, joten seuraava jakauma on $[1/2, 0, 1/2, 0, 0]$ jne.

Tehokas tapa simuloida kulkua Markovin ketjussa on käyttää dynaamista ohjelmointia. Ideana on pitää yllä tilajakaumaa ja käydä joka vuorolla läpi kaikki tilat ja jokaisesta tilasta kaikki mahdollisuudet jatkaa eteenpäin. Tätä menetelmää käyttäen m askeleen simulointi vie aikaa $O(n^2 m)$.

Markovin ketjun tilasiirtymät voi esittää myös matriisina, jonka avulla voi päivittää tilajakaumaa askeleen eteenpäin. Tässä tapauksessa matriisi on

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

Tällaisella matriisilla voi kertoa tilajakaumaa esittävän vektorin, jolloin saadaan seuraava tilajakauma. Esimerkiksi jakaumasta $[1, 0, 0, 0, 0]$ pääsee jakaumaan $[0, 1, 0, 0, 0]$ seuraavasti:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Matriisiin voi soveltaa edelleen tehokasta matriisipotensssia, jonka avulla voi laskea ajassa $O(n^3 \log m)$, mikä on jakauma m askeleen jälkeen.

24.5 Satunnaisalgoritmit

Joskus tehtävässä voi hyödyntää satunnaisuutta, vaikka tehtävä ei itsessään liittyisi todennäköisyyteen. Satunnaisalgoritmit ovat algoritmeja, joiden toiminta perustuu satunnaisuuteen.

Monte Carlo -algoritmi on satunnaisalgoritmi, joka saattaa tuottaa joskus väärän tuloksen. Jotta algoritmi olisi käyttökelpoinen, väärän vastauksen todennäköisyyden tulee olla pieni.

Las Vegas -algoritmi on satunnaisalgoritmi, joka tuottaa aina oikean tuloksen mutta jonka suoritus aika vaihtelee satunnaisesti. Tavoitteena on, että algoritmi toimisi nopeasti suurella todennäköisyydellä.

Tutustumme seuraavaksi kolmeen esimerkkitehtävään, jotka voi ratkaista satunnaisuuden avulla.

Järjestystunnusluku

Taulukon k . **järjestystunnusluku** on kohdassa k oleva alkio, kun alkiojärjestetään pienimmästä suurimpaan. On helppoa laskea mikä tahansa järjestystunnusluku ajassa $O(n \log n)$ järjestämällä taulukko, mutta onko tarpeen järjestää koko taulukkoa yhden alkion selvittämiseksi?

Osoittautuu, että tehtävän voi ratkaista myös satunnaisalgoritmeilla. Algoritmi on Las Vegas -tyyppinen: sen aikavaativuus on yleensä $O(n)$, mutta pahimmassa tapauksessa $O(n^2)$.

Algoritmi valitsee taulukosta satunnaisen alkion x ja siirtää x :ää pienemmät alkio taulukon vasempaan osaan ja loput alkio taulukon oikeaan osaan. Tämä vie aikaa $O(n)$, kun taulukossa on n alkioita. Oletetaan, että vasemmassa osassa on a alkioita ja oikeassa osassa on b alkioita. Nyt jos $a = k - 1$, alkio x on haluttu alkio. Jos $a > k - 1$, etsitään rekursiivisesti vasemmasta osasta, mikä on kohdassa k oleva alkio. Jos taas $a < k - 1$, etsitään rekursiivisesti oikeasta osasta, mikä on kohdassa $k - a - 1$ oleva alkio. Haku jatkuu vastaavalla tavalla rekursiivisesti, kunnes haluttu alkio on löytynyt.

Kun alkio x valitaan satunnaisesti, taulukon koko suunnilleen puolittuu joka vaiheessa, joten kohdassa k olevan alkion etsiminen vie aikaa

$$n + n/2 + n/4 + n/8 + \dots = O(n).$$

Algoritmin pahin tapaus on silti $O(n^2)$, koska on mahdollista, että x valitaan sattumalta aina niin, että se on taulukon pienin alkio. Silloin taulukko pienenee joka vaiheessa vain yhden alkion verran. Tämän todennäköisyys on kuitenkin erittäin pieni, eikä näin tapahdu käytännössä.

Matriisitulon tarkastaminen

Seuraava tehtävämme on *tarkastaa*, päteekö matriisitulo $AB = C$, kun A , B ja C ovat $n \times n$ -kokoisia matriiseja. Tehtävän voi ratkaista laskemalla matriisitulon AB (perusalgoritmeilla ajassa $O(n^3)$), mutta voisi toivoa, että ratkaisun tarkastaminen olisi helpompaa kuin sen laskeminen alusta alkaen uudestaan.

Osoittautuu, että tehtävän voi ratkaista Monte Carlo -algoritmeilla, jonka aikavaativuus on vain $O(n^2)$. Idea on yksinkertainen: valitaan satunnainen $n \times 1$ -matriisi X ja lasketaan matriisit ABX ja CX . Jos $ABX = CX$, ilmoitetaan, että $AB = C$, ja muuten ilmoitetaan, että $AB \neq C$.

Algoritmin aikavaativuus on $O(n^2)$, koska matriisien ABX ja CX laskeminen vie aikaa $O(n^2)$. Matriisin ABX tapauksessa laskennan voi suorittaa osissa $A(BX)$, jolloin riittää kertoa kahdesti $n \times n$ - ja $n \times 1$ -kokoiset matriisit.

Algoritmin heikkoutena on, että on pieni mahdollisuus, että algoritmi erehtyy, kun se ilmoittaa, että $AB = C$. Esimerkiksi

$$\begin{bmatrix} 2 & 4 \\ 1 & 6 \end{bmatrix} \neq \begin{bmatrix} 0 & 5 \\ 7 & 4 \end{bmatrix},$$

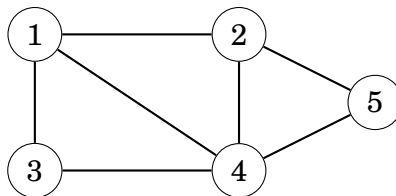
mutta

$$\begin{bmatrix} 2 & 4 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 5 \\ 7 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$

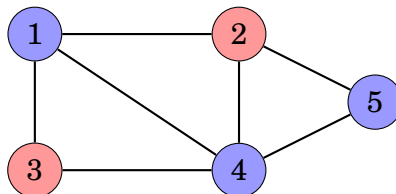
Käytännössä erehtymisen todennäköisyys on kuitenkin hyvin pieni ja todennäköisyyttä voi pienentää lisää tekemällä tarkastuksen usealla satunnaisella matriisilla X ennen vastauksen $AB = C$ ilmoittamista.

Verkon värittäminen

Annettuna on verkko, jossa on n solmua ja m kaarta. Tehtävänä on etsiä tapa värittää verkon solmut kahdella värillä niin, että ainakin $m/2$ kaarella päätesolmut ovat eri väriset. Esimerkiksi verkossa



yksi kelvollinen väritys on seuraava:



Yllä olevassa verkossa on 7 kaarta ja niistä 5:ssä päätesolmut ovat eri väriset, joten väritys on kelvollinen.

Tehtävä on mahdollista ratkaista Las Vegas -algoritmillä muodostamalla satunnaisia värityksiä niin kauan, kunnes syntyy kelvollinen väritys. Satunnaisessa värityksessä jokaisen solmun väri on valittu toisistaan riippumatta niin, että kummankin värin todennäköisyys on $1/2$.

Satunnaisessa värityksessä todennäköisyys, että yksittäisen kaaren päätesolmut ovat eri väriset on $1/2$. Niinpä odotusarvo, monessako kaarella päätesolmut ovat eri väriset, on $1/2 \cdot m = m/2$. Koska satunnainen väritys on odotusarvoisesti kelvollinen, jokin kelvollinen väritys löytyy käytännössä nopeasti.

Luku 25

Peliteoria

Tässä luvussa keskitymme kahden pelaajan peleihin, joissa molemmat pelaajat tekevät samanlaisia siirtoja eikä pelissä ole satunnaisuutta. Tavoitteemme on etsiä strategia, jota käyttäen pelaaja pystyy voittamaan pelin toisen pelaajan toimista riippumatta, jos tämä on mahdollista.

Osoittautuu, että kaikki tällaiset pelit ovat pohjimmiltaan samanlaisia ja niiden analyysi on mahdollista **nim-teorian** avulla. Perehdymme aluksi yksinkertaisiin tikkupeleihin, joissa pelaajat poistavat tikkuja kasoista, ja yleistämme sitten näiden pelien teorian kaikkiin peleihin.

25.1 Pelin tilat

Tarkastellaan peliä, jossa kasassa on n tikkua. Pelaajat A ja B siirtävät vuorotellen ja pelaaja A aloittaa. Jokaisella siirrolla pelaajan tulee poistaa 1, 2 tai 3 tikkua kasasta. Pelin voittaa se pelaaja, joka poistaa viimeisen tikun.

Esimerkiksi jos $n = 10$, peli saattaa edetä seuraavasti:

1. Pelaaja A poistaa 2 tikkua (jäljellä 8 tikkua).
2. Pelaaja B poistaa 3 tikkua (jäljellä 5 tikkua).
3. Pelaaja A poistaa 1 tikun (jäljellä 4 tikkua).
4. Pelaaja B poistaa 2 tikkua (jäljellä 2 tikkua).
5. Pelaaja A poistaa 2 tikkua ja voittaa.

Tämä peli muodostuu tiloista $0, 1, 2, \dots, n$, missä tilan numero vastaa sitä, montako tikkua kasassa on jäljellä. Tietyssä tilassa olevan pelaajan valittavana on, montako tikkua hän poistaa kasasta.

Voittotila ja häviötila

Voittotila on tila, jossa oleva pelaaja voittaa pelin varmasti, jos hän pelaa optimaalisesti. Vastaavasti **häviötila** on tila, jossa oleva pelaaja häviää varmasti, jos vastustaja pelaa optimaalisesti. Osoittautuu, että pelin tilat on mahdollista luokitella niin, että jokainen tila on joko voittotila tai häviötila.

Yllä olevassa pelissä tila 0 on selkeästi häviötila, koska siinä oleva pelaaja häviää pelin suoraan. Tilat 1, 2 ja 3 taas ovat voittotiloja, koska niissä oleva pelaaja voi poistaa 1, 2 tai 3 tikkua ja voittaa pelin. Vastaavasti tila 4 on häviötila, koska mikä tahansa siirto johtaa toisen pelaajan voittoon.

Yleisemmin voidaan havaita, että jos tilasta on jokin häviötilaan johtava siirto, niin tila on voittotila, ja muussa tapauksessa tila on häviötila. Tämän ansios- ta voidaan luokitella kaikki pelin tilat alkaen varmoista häviötiloista, joista ei ole siirtoja mihinkään muuhun tilaan.

Seuraavassa on pelin tilojen 0...15 luokittelu (V tarkoittaa voittotilaa ja 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

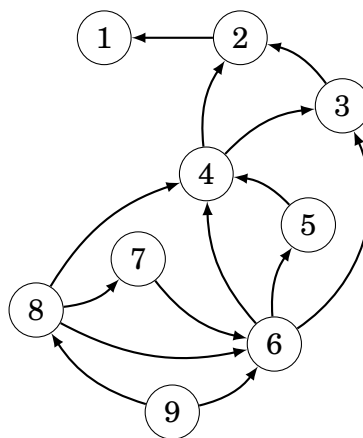
Tämän pelin analyysi on yksinkertainen: tila k on häviötila, jos k on jaollinen 4:llä, ja muuten tila k on voittotila. Optimaalinen tapa pelata peliä on valita aina sellainen siirto, että vastustajalle jää 4:llä jaollinen määrä tikkuja, kunnes lopulta tikut loppuvat ja vastustaja on hävinnyt.

Tämä pelitapa edellyttää luonnollisesti sitä, että tikkujen määrä omalla siir- rolla ei ole 4:llä jaollinen. Jos näin kuitenkin on, mitään ei ole tehtävissä vaan vastustaja voittaa pelin varmasti, jos hän pelaa optimaalisesti.

Tilaverkko

Tarkastellaan sitten toisenlaista tikkupeliä, jossa tilassa k saa poistaa minkä tahansa määrän tikkuja x , kunhan k on jaollinen x :llä ja x on pienempi kuin k . Esimerkiksi tilassa 8 on sallittua poistaa 1, 2 tai 4 tikkua, mutta tilassa 7 ainoa mahdollinen siirto on poistaa 1 tikku.

Esitetään pelin tilat 1...9 **tilaverkkona**, jossa solmut ovat pelin tiloja ja kaaret kuvaavat mahdollisia siirtoja tilojen välillä:



Tämä peli päättyy aina tilaan 1, joka on häviötila, koska siinä ei voi tehdä mitään siirtoja. Pelin tilojen 1...9 luokittelu on seuraava:

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

Yllättävää kyllä, tässä pelissä kaikki parilliset tilat ovat voittotiloja ja kaikki parittomat tilat ovat häviötiloja.

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-pelissä on n kasaa tikkuja, joista kussakin on tietty määrä tikkuja. Pelaajat poistavat kasoista tikkuja vuorotellen. Joka vuorolla pelaaja valitsee yhden kasan, jossa on vielä tikkuja, 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 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 tähän tilaan.

Analyysi

Osoittautuu, että nim-pelin tilan luonteen kertoo **nim-summa** $x_1 \oplus x_2 \oplus \dots \oplus x_n$, missä \oplus tarkoittaa xor-operaatiota. Jos nim-summa on 0, tila on häviötila, ja muussa tapauksessa tila on voittotila. Esimerkiksi tilan $(10, 12, 5)$ nim-summa on $10 \oplus 12 \oplus 5 = 3$, joten tila on voittotila.

Mutta miten nim-summa liittyy nim-peliin? Tämä selviää tutkimalla, miten nim-summa muuttuu, kun nim-pelin tila muuttuu.

Häviötilat: Pelin päätöstilä $[0, 0, \dots, 0]$ on häviötila, ja sen nim-summa on 0, kuten kuuluukin. Muissa häviötiloissa mikä tahansa siirto johtaa voittotilaan, koska yksi luvuista x_k muuttuu ja samalla pelin nim-summa muuttuu eli siirron jälkeen nim-summasta tulee jokin muu kuin 0.

Voittotilat: Voittotilasta pääsee häviötilaan muuttamalla jonkin kasan k tikkujen määräksi $x_k \oplus s$, missä s on pelin nim-summa. Vaatimuksena on, että $x_k \oplus s < x_k$, koska kasasta voi vain poistaa tikkuja. Sopiva kasa x_k on sellainen, jossa on ykkösbitti samassa kohdassa kuin s :n vasemmanpuoleisin ykkösbitti.

Tarkastellaan esimerkkinä tilaa $(10, 2, 5)$. Tämä tila on voittotila, koska sen nim-summa on 3. Täytyy siis olla olemassa siirto, jolla tilasta pääsee häviötilaan. Selvitetään se seuraavaksi.

Pelin nim-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 nim-summan vasemmanpuoleisin ykkösbitti:

10	10 1 0
12	1100
5	0101
3	00 1 1

Kasan uudeksi sisällöksi täytyy saada $10 \oplus 3 = 9$ tikkua, mikä onnistuu poistamalla 1 tikku 10 tikun kasasta. Tämän seurauksena tilaksi tulee (9,12,5), joka on häviötila, kuten pitääkin:

9	1001
12	1100
5	0101
0	0000

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. Käännepähtyy 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 nim-summa ei ole 0.

25.3 Sprague–Grundyn lause

Sprague–Grundyn lause yleistää nim-pelin strategian kaikkiin peleihin, jotka täyttävät seuraavat vaatimukset:

- Pelissä on kaksi pelaajaa, jotka tekevät vuorotellen siirtoja.
- Peli muodostuu tiloista ja mahdolliset siirrot tilasta eivät riipu siitä, kumpi pelaaja on vuorossa.

- Peli päättyy, kun toinen pelaaja ei voi tehdä siirtoa.
- Peli päättyy varmasti ennemmin tai myöhemmin.
- Pelaajien saatavilla on kaikki tieto tiloista ja siirroista, eikä pelissä ole satunnaisuutta.

Ideana on laskea kullekin pelin tilalle Grundy-luku, joka vastaa tikkujen määrää nim-pelin kasassa. Kun kaikkien tilojen Grundy-luvut ovat tiedossa, peliä voi pelata aivan kuin se olisi nim-peli.

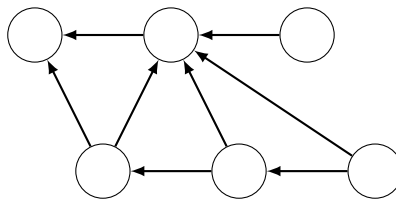
Grundy-luku

Pelin tilan **Grundy-luku** määritellään rekursiivisesti kaavalla

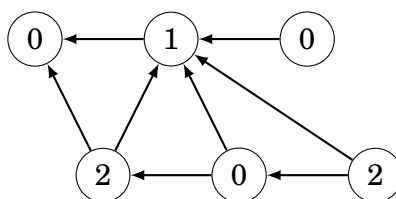
$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

jossa g_1, g_2, \dots, g_n ovat niiden tilojen Grundy-luvut, joihin tilasta pääsee yhdellä siirrolla, ja funktio mex antaa pienimmän ei-negatiivisen luvun, jota ei esiinny joukossa. Esimerkiksi $\text{mex}(\{0, 1, 3\}) = 2$. Jos tilasta ei voi tehdä mitään siirtoa, sen Grundy-luku on 0, koska $\text{mex}(\emptyset) = 0$.

Esimerkiksi tilaverkossa



Grundy-luvut ovat seuraavat:



Jos tila on häviötila, sen Grundy-luku on 0. Jos taas tila on voittotila, sen Grundy-luku on jokin positiivinen luku.

Grundy-luvun hyötynä on, että se vastaa tikkujen määrää nim-kasassa. Jos Grundy-luku on 0, niin tilasta pääsee vain tiloihin, joiden Grundy-luku ei ole 0. Jos taas Grundy-luku on $x > 0$, niin tilasta pääsee tiloihin, joiden Grundy-luvut kattavat välin $0, 1, \dots, x - 1$.

Tarkastellaan esimerkkinä peliä, jossa pelaajat siirtävät vuorotellen pelihahmoa sokkelossa. Jokainen sokkelon ruutu on lattiaa tai seinää. Kullakin siirrolla hahmon tulee liikkua jokin määrä askeleita vasemmalle tai jokin määrä askeleita ylöspäin. Pelin voittaja on se, joka tekee viimeisen siirron.

Esimerkiksi seuraavassa on pelin mahdollinen aloitustilanne, jossa @ on pelihahmo ja * merkitsee ruutua, johon hahmo voi siirtyä.

				*
				*
*	*	*	*	@

Sokkelopelin tiloja ovat kaikki sokkelon lattiaruudut. Tässä tapauksessa 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 seurauksena sokkelopelin tila käyttäytyy samalla tavalla kuin nimipelin kasa. Esimerkiksi oikean alakulman ruudun Grundy-luku on 2, joten kyseessä on voittotila. Voittoon johtava siirto on joko liikkua neljä askelta vasemmalle tai kaksi askelta ylöspäin.

Huomaa, että toisin kuin nimipelissä, tilasta saattaa päästä toiseen tilaan, jonka Grundy-luku on suurempi. Tällaisen siirron voi kuitenkin aina peruuttaa niin, että Grundy-luku palautuu samaksi.

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 nim-summa. Peliä pystyy pelaamaan nimipelin tapaan selvittämällä kaikkien alipelien Grundy-luvut ja laskemalla niiden nim-summa.

Tarkastellaan esimerkkinä kolmen sokkelon peliä. Tässä pelissä pelaaja valitsee joka siirrolla 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 nim-summa on $2 \oplus 3 \oplus 3 = 2$, joten aloittaja pystyy voittamaan pelin. Voittoon johtava siirto on liikkua vasemmassa sokkelossa 2 askelta ylöspäin, jolloin nim-summaksi tulee $0 \oplus 3 \oplus 3 = 0$.

Jakautuminen

Joskus siirto pelissä jakaa pelin alipeleihin, jotka ovat toisistaan riippumattomia. Tällöin pelin Grundy-luku on

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

missä n on siirtojen määrä ja

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

missä siirron k tuottamien alipelien Grundy-luvut ovat $a_{k,1}, a_{k,2}, \dots, a_{k,m}$.

Esimerkki tällaisesta pelistä on **Grundyn peli**. Pelin alkutilanteessa on kasa, jossa on n tikkua. Joka vuorolla pelaaja valitsee jonkin kasan ja jakaa sen kahdeksi epätyhjäksi kasaksi niin, että kasoissa on eri määrä tikkuja. Pelin voittaja on se, joka tekee viimeisen jaon.

Merkitään $f(n)$ Grundy-lukua kasalle, jossa on n tikkua. Grundy-luku muodostuu käymällä läpi tavat jakaa kasa kahdeksi kasaksi. Esimerkiksi tapauksessa $n = 8$ mahdolliset jakotavat ovat $1 + 7$, $2 + 6$ ja $3 + 5$, joten

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

Tässä pelissä luvun $f(n)$ laskeminen vaatii lukujen $f(1), \dots, f(n-1)$ laskemista. Pohjatapauksina $f(1) = f(2) = 0$, koska 1 ja 2 tikun kasaa ei ole mahdollista jakaa mitenkään. Ensimmäiset Grundy-luvut ovat:

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

Tapauksen $n = 8$ Grundy-luku on 2, joten peli on mahdollista voittaa. Voittosiirto on muodostaa kasat $1 + 7$, koska $f(1) \oplus f(7) = 0$.

Luku 26

Merkkijonoalgoritmit

Merkkijonon s merkit ovat $s[1], s[2], \dots, s[n]$, missä n on merkkijonon pituus.

Aakkosto sisältää ne merkit, joita merkkijonossa voi esiintyä. Esimerkiksi aakkosto $\{A, B, \dots, Z\}$ sisältää englannin kielen suuret kirjaimet.

Osajono sisältää merkkijonon merkit yhtenäiseltä väliltä. Merkkijonon osajonojen määrä on $n(n+1)/2$. Esimerkiksi merkkijonon ALGORITMI yksi osajono on ORITM, joka muodostuu valitsemalla välin ALGORITMI.

Alijono on osajoukko merkkijonon merkeistä. Merkkijonon alijonojen määrä on $2^n - 1$. Esimerkiksi merkkijonon ALGORITMI yksi alijono on LGRMI, joka muodostuu valitsemalla merkit ALGORITMI.

Alkuosa on merkkijonon alusta alkava osajono, ja **loppuosa** on merkkijonon loppuun päättyvä osajono. Esimerkiksi merkkijonon KISSA alkuosat ovat K, KI, KIS, KISS ja KISSA ja loppuosat ovat A, SA, SSA, ISSA ja KISSA. Alkuosa tai loppuosa on **aito**, jos se ei ole koko merkkijono.

Kierto syntyy siirtämällä jokin alkuosa merkkijonon loppuun tai jokin loppuosa merkkijonon alkuun. Esimerkiksi merkkijonon APILA kierrot ovat APILA, PILAA, ILAAP, LAAPI ja AAPIL.

Jakso on alkuosa, jota toistamalla merkkijono muodostuu. Jakson viimeinen toistokerta voi olla osittainen niin, että siinä on vain jakson alkuosa. Usein on kiinnostavaa selvittää, mikä on merkkijonon **lyhin jakso**. Esimerkiksi merkkijonon ABCABCA lyhin jakso on ABC. Tässä tapauksessa merkkijono syntyy toistamalla jaksoa ensin kahdesti kokonaan ja sitten kerran osittain.

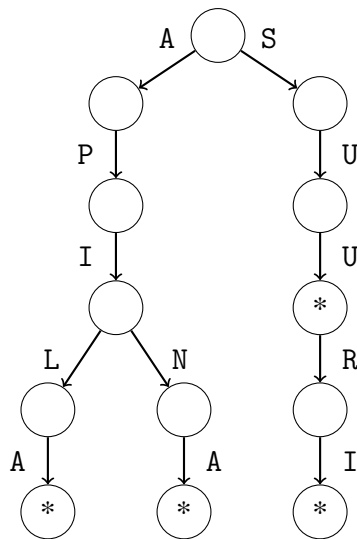
Reuna on merkkijono, joka on sekä alkuosa että loppuosa. Esimerkiksi merkkijonon ABADABA reunat ovat A, ABA ja ABADABA. Usein halutaan etsiä **pisin reuna**, joka ei ole koko merkkijono.

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 $x[k] < y[k]$.

26.1 Trie-rakenne

Trie on puurakenne, joka pitää yllä joukkoa merkkijonoja. 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 merkkijono lisääminen ja hakeminen vievät aikaa $O(n)$, kun n on merkkijonon pituus. Molemmat operaatiot voi toteuttaa lähtemällä liikkeelle juuresta ja kulkemalla alaspäin ketjua merkkien mukaisesti. Tarvittaessa puuhun lisätään 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.

Trie on kätevää tallentaa taulukkona

```
int t[N][A];
```

missä N on solmujen suurin mahdollinen määrä (eli tallennettavien merkkijonojen yhteispituus) ja A on aakkoston koko. Ideana on numeroida solmut $1, 2, 3, \dots$ niin, että juuren numero on 1. Nyt $t[s][c]$ kertoo, mihin solmuun solmusta s pääsee merkillä c .

26.2 Merkkijonohajautus

Merkkijonohajautus on tekniikka, jonka avulla voi esikäsittelyn jälkeen tarkastaa tehokkaasti, ovatko kaksi merkkijonon osajonoa samat. Ideana on verrata toisiinsa osajonojen hajautusarvoja, mikä on tehokkaampaa kuin osajonojen vertaaminen merkki kerrallaan.

Hajautusarvon laskeminen

Merkkijonon **hajautusarvo** on luku, joka lasketaan merkkijonon merkeistä etukäteen valitulla tavalla. Jos kaksi merkkijonoa ovat samat, myös niiden ha-

jautusarvot ovat samat, minkä ansiosta merkkijonoja voi vertailla niiden hajautusarvojen kautta.

Tavallinen tapa toteuttaa merkkijonohajautus on käyttää polynomista hajautusta. Siinä hajautusarvo lasketaan kaavalla

$$(c[1]A^{n-1} + c[2]A^{n-2} + \dots + c[n]A^0) \bmod B,$$

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

Merkkijonohajautuksen esikäsittely muodostaa tietoa, jonka avulla voi laskea tehokkaasti merkkijonon osajonojen hajautusarvoja. Osoittautuu, että polynomisessa hajautuksessa $O(n)$ -aikaisen esikäsittelyn jälkeen voi laskea minkä tahansa osajonon hajautusarvon ajassa $O(1)$.

Ideana on muodostaa taulukko h , jossa $h[k]$ on hajautusarvo merkkijonon alkuosalle kohtaan k asti. Taulukon voi muodostaa rekursiolla seuraavasti:

$$\begin{aligned} h[0] &= 0 \\ h[k] &= (h[k-1]A + c[k]) \bmod B \end{aligned}$$

Lisäksi muodostetaan taulukko p , jossa $p[k] = A^k \bmod B$:

$$\begin{aligned} p[0] &= 1 \\ p[k] &= (p[k-1]A) \bmod B. \end{aligned}$$

Näiden taulukoiden muodostaminen vie aikaa $O(n)$. Tämän jälkeen hajautusarvo merkkijonon osajonolle, joka alkaa kohdasta a ja päättyy kohtaan b , voidaan laskea $O(1)$ -ajassa kaavalla

$$(h[b] - h[a-1]p[b-a+1]) \bmod B.$$

Hajautuksen käyttö

Hajautusarvot tarjoavat nopean tavan merkkijonojen vertailemiseen. Ideana on vertailla merkkijonojen koko sisällön sijasta niiden hajautusarvoja. Jos hajautusarvot ovat samat, myös merkkijonot ovat *todennäköisesti* samat, ja jos taas hajautusarvot eivät ole samat, merkkijonot eivät ole samat.

Hajautuksen avulla voi usein tehostaa raa'an voiman algoritmia niin, että siitä tulee tehokas. Tarkastellaan esimerkkinä raa'an voiman algoritmia, joka laskee, montako kertaa merkkijono p esiintyy osajonona merkkijonossa s . Algoritmi käy läpi kaikki kohdat, joissa p voi esiintyä, ja vertailee merkkijonoja merkki merkiltä. Tällaisen algoritmin aikavaativuus on $O(n^2)$.

Voimme kuitenkin tehostaa algoritmia hajautuksen avulla, koska algoritmista vertaillaan merkkijonojen osajonoja. Hajautusta käyttäen kukin vertailu vie aikaa vain $O(1)$, koska vertailua ei tehdä merkki merkiltä vaan suoraan hajautusarvon perusteella. Tuloksena on algoritmi, jonka aikavaativuus on $O(n)$, joka on paras mahdollinen aikavaativuus tehtävään.

Yhdistämällä hajautus ja *binäärihaku* on mahdollista myös selvittää logaritmisessa ajassa, kumpi kahdesta osajonosta on suurempi aakkosjärjestyksessä. Tämä onnistuu tutkimalla ensin binäärihaulla, kuinka pitkä on merkkijonojen yhteinen alkuosa, minkä jälkeen yhteisen alkuosan jälkeinen merkki kertoo, kumpi merkkijono on suurempi.

Törmäykset ja parametrit

Ilmeinen riski hajautusarvojen vertailussa on **törmäys**, joka 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. Riskin saa kuitenkin pieneksi valitsemalla hajautuksen 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 suuria satunnaislukuja. Tavallinen tapa on valita vakiot läheltä lukua 10^9 , esimerkiksi

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Tällainen valinta takaa 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 mahtuvat `long long`-tyyppiin. Mutta onko 10^9 riittävä määrä hajautusarvoja?

Tarkastellaan nyt kolmea hajautuksen käyttötapaa:

Tapaus 1: Merkkijonoja x ja y verrataan toisiinsa. Törmäyksen todennäköisyys on $1/B$ olettaen, että kaikki hajautusarvot esiintyvät yhtä usein.

Tapaus 2: Merkkijonoa x verrataan merkkijonoihin y_1, y_2, \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 tapauksessa 1 törmäyksen riski on olematon valinnalla $B \approx 10^9$. Tapauksessa 2 riski on olemassa, mutta se on silti edelleen vähäinen. Tapauksessa 3 tilanne on kuitenkin täysin toinen: törmäys tapahtuu käytännössä varmasti vielä valinnalla $B \approx 10^9$.

Tapauksen 3 ilmiö tunnetaan nimellä **syntymäpäiväparadoksi**: jos huoneessa on n henkilöä, on suuri todennäköisyys, että jollain kahdella henkilöllä on sama syntymäpäivä, vaikka n olisi melko pieni. Vastaavasti hajautuksessa kun kaikkia hajautusarvoja verrataan keskenään, käy helposti niin, että jotkin kaksi ovat sattumalta samoja.

Hyvä tapa pienentää törmäyksen riskiä on laskea *useita* hajautusarvoja eri parametreilla ja verrata niitä kaikkia. On hyvin pieni todennäköisyys, että törmäys tapahtuisi samaan aikaan kaikissa hajautusarvoissa. Esimerkiksi kaksi hajautusarvoa parametrilla $B \approx 10^9$ vastaa yhtä hajautusarvoa parametrilla $B \approx 10^{18}$, mikä takaa hyvän suojan törmäyksiltä.

Jotkut käyttävät hajautuksessa vakioita $B = 2^{32}$ tai $B = 2^{64}$, jolloin modulo B tulee laskettua automaattisesti, kun muuttujan arvo pyörähtää ympäri. Tämä ei ole kuitenkaan hyvä valinta, koska muotoa 2^x olevaa moduloa vastaan pystyy tekemään testisyötteen, joka aiheuttaa varmasti törmäyksen¹.

26.3 Z-algoritmi

Z-algoritmi muodostaa merkkijonosta **Z-aulukon**, joka kertoo kullekin merkkijonon kohdalle, mikä on pisin kyseisestä kohdasta alkava osajono, joka on myös merkkijonon alkuosa. Z-algoritmin avulla voi ratkaista tehokkaasti monia merkkijonotehtäviä.

Z-algoritmi ja merkkijonohajautus ovat usein vaihtoehtoisia tekniikoita, ja on makuasia, kumpaa algoritmia käyttää. Toisin kuin hajautus, Z-algoritmi toimii varmasti oikein eikä siinä ole törmäysten riskiä. Toisaalta Z-algoritmi on vaikeampi toteuttaa eikä se sovellu kaikkeen samaan kuin hajautus.

Algoritmin toiminta

Z-algoritmi muodostaa merkkijonolle Z-aulukon, jonka jokaisessa kohdassa lukee, kuinka pitkälle kohdasta alkava osajono vastaa merkkijonon alkuosaa. Esi-

¹ J. Pachocki ja Jakub Radoszweski: "Where to use and how not to use polynomial string hashing". *Olympiads in Informatics*, 2013.

merkiksi Z-taulukko merkkijonolle ACBACDACBACBACDA on seuraava:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Esimerkiksi kohdassa 7 on arvo 5, koska siitä alkava 5-merkkinen osajono ACBAC on merkkijonon alkuosa, mutta 6-merkkinen osajono ACBACB ei ole enää merkkijonon alkuosa.

Z-algoritmi käy läpi merkkijonon vasemmalta oikealle ja laskee jokaisessa kohdassa, kuinka pitkälle kyseisestä kohdasta alkava osajono täsmää merkkijonon alkuun. Algoritmi laskee yhteisen alkuosan pituuden vertaamalla merkkijonon alkua ja osajonon alkua toisiinsa.

Suoraviivaisesti toteutettuna tällaisen algoritmin aikavaativuus olisi $O(n^2)$, koska yhteiset alkuosat voivat olla pitkiä. Z-algoritmissa on kuitenkin yksi tärkeä optimointi, jonka ansiosta algoritmin aikavaativuus on vain $O(n)$.

Ideana on pitää muistissa väliä $[x, y]$, joka on aiemmin laskettu merkkijonon alkuun täsmäävä väli, jossa y on mahdollisimman suuri. Tällä välillä olevia merkkejä ei tarvitse koskaan verrata uudestaan merkkijonon alkuun, vaan niitä koskevan tiedon saa suoraan Z-taulukon lasketusta osasta.

Z-algoritmin aikavaativuus on $O(n)$, koska algoritmi aloittaa merkki kerrallaan vertailun aina kohdasta $y + 1$. Jos merkit täsmäävät, kohta y siirtyy eteenpäin eikä algoritmin tarvitse enää koskaan vertailla tätä kohtaa, vaan se pystyy hyödyntämään Z-taulukon alussa olevaa tietoa.

Esimerkki

Katsotaan nyt, miten Z-algoritmi muodostaa seuraavan Z-taulukon:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Ensimmäinen mielenkiintoinen kohta tulee, kun yhteisen alkuosan pituus on 5. Silloin algoritmi laittaa muistiin välin $[7, 11]$ seuraavasti:

						x		y							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?

Välin $[7, 11]$ hyötynä on, että algoritmi voi sen avulla laskea seuraavat Z-taulukon arvot nopeammin. Koska välin $[7, 11]$ merkit ovat samat kuin merkkijonon alussa, myös Z-taulukon arvoissa on vastaavuutta.

Ensinnäkin kohdissa 8 ja 9 tulee olla samat arvot kuin kohdissa 2 ja 3, koska väli $[7, 11]$ vastaa väliä $[1, 5]$:

						x		y							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

Seuraavaksi kohdasta 4 saa tietoa kohdan 10 arvon laskemiseksi. Koska kohdassa 4 on arvo 2, tämä tarkoittaa, että osajono täsmää kohtaan $y = 11$ asti, mutta sen jälkeen on tutkimatonta aluetta merkkijonossa.

						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 $y + 1 = 12$ alkaen merkki kerrallaan. Algoritmi ei voi hyödyntää valmiina Z-taulukossa olevaa tietoa, 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-taulukon arvojen laskemisessa pystyy hyödyntämään jälleen välin $[x, y]$ antamaa tietoa ja algoritmi saa Z-taulukon loppuun tulevat arvot suoraan Z-taulukon 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-taulukon käyttö

Ratkaistaan esimerkkinä tehtävä, jossa laskettavana on, montako kertaa merkkijono p esiintyy osajonona merkkijonossa s . Ratkaisimme tehtävän aiemmin

tehokkaasti merkkijonohajautuksen avulla, ja nyt Z-algoritmi tarjoaa siihen vaihtoehtoisen lähestymistavan.

Usein esiintyvä idea Z-algoritmin yhteydessä on muodostaa merkkijono, jonka osana on useita välimerkeillä erotettuja merkkijonoja. Tässä tehtävässä sopiva merkkijono on $p\#s$, jossa merkkijonojen p ja s välissä on erikoismerkki $\#$, jota ei esiinny merkkijonoissa. Nyt merkkijonoa $p\#s$ vastaava Z-taulukko kertoo, missä kohdissa merkkijonoa p esiintyy merkkijono s . Tällaiset kohdat ovat tarkalleen ne Z-taulukon kohdat, joissa on merkkijonon p pituus.

Esimerkiksi jos $s = \text{HATTIVATTI}$ ja $p = \text{ATT}$, niin Z-taulukosta tulee:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	T	T	#	H	A	T	T	I	V	A	T	T	I
–	0	0	0	0	3	0	0	0	0	3	0	0	0

Taulukon kohdissa 6 ja 11 on luku 3, mikä tarkoittaa, että ATT esiintyy vastavissa kohdissa merkkijonossa HATTIVATTI.

Tuloksena olevan algoritmin aikavaativuus on $O(n)$, koska riittää muodostaa Z-taulukko ja käydä se läpi.

Algoritmin toteutus

Seuraavassa on lyhyt Z-algoritmin toteutus, joka palauttaa Z-taulukon vektorina. Huomaa, että tässä toteutuksessa merkkijonossa ja taulukossa on 0-indeksointi toisin kuin aiemmin tässä luvussa.

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0, min(z[i-x], y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```


Luku 27

Neliöjuurialgoritmit

Neliöjuurialgoritmi on algoritmi, jonka aikavaativuudessa esiintyy neliöjuuri. Neliöjuurta voi ajatella ”köyhän miehen logaritmina”: aikavaativuus $O(\sqrt{n})$ on parempi kuin $O(n)$ mutta huonompi kuin $O(\log n)$. Toisaalta neliöjuurialgoritmit toimivat käytännössä hyvin ja niiden vakiokertoimet ovat pieniä.

Tarkastellaan esimerkkinä tuttua ongelmaa, jossa toteutettavana on summakysely taulukkoon. Halutut operaatiot ovat:

- muuta kohdassa x olevaa lukua
- laske välin $[a, b]$ lukujen summa

Olemme aiemmin ratkaisseet tehtävän binääri-indeksipuun ja segmenttipuun avulla, jolloin kummankin operaation aikavaativuus on $O(\log n)$. Nyt ratkaisemme tehtävän toisella tavalla neliöjuurirakennetta käyttäen, jolloin summan laskenta vie aikaa $O(\sqrt{n})$ ja luvun muuttaminen vie aikaa $O(1)$.

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 summan laskemisen aikavaativuus on yhteensä $O(\sqrt{n})$.

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.

27.1 Eräkäsittely

Eräkäsittelyssä algoritmin suorittamat operaatiot jaetaan eriin, jotka käsitellään omina kokonaisuuksina. Erien välissä tehdään yksittäinen työläs toimenpide, joka auttaa tulevien operaatioiden käsittelyä.

Neliöjuurialgoritmi syntyy, kun n operaatiota jaetaan $O(\sqrt{n})$ -kokoisiin eriin, jolloin sekä eriä että operaatioita kunkin erän sisällä on $O(\sqrt{n})$. Tämä tasapainottaa sitä, miten usein erien välinen työläs toimenpide tapahtuu sekä miten paljon työtä erän sisällä täytyy tehdä.

Tarkastellaan esimerkkinä tehtävää, jossa ruudukossa on $k \times k$ ruutua, jotka ovat aluksi valkoisia. Tehtävänä on suorittaa n operaatiota ruudukkoon, joista jokainen on jompikumpi seuraavista:

- väritä ruutu (y, x) mustaksi
- etsi ruudusta (y, x) lähin musta ruutu, kun 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} .

27.2 Tapauskäsittely

Tapauskäsittelyssä algoritmissa on useampia toimintatapoja, jotka aktivoidaan 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.

Tarkastellaan esimerkkinä tehtävää, jossa puussa on n solmua, joista jokaisella on tietty väri. Tavoitteena on etsiä puusta kaksi solmua, jotka ovat samavärisiä ja mahdollisimman kaukana toisistaan.

Tehtävän voi ratkaista käymällä läpi värit yksi kerrallaan ja etsimällä kullekin värille kaksi solmua, jotka ovat mahdollisimman kaukana toisistaan. Tietysti värillä c algoritmin toiminta riippuu siitä, montako kyseisen väristä solmua puussa on. Tapaukset ovat seuraavat:

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 $O(n\sqrt{n})$.

27.3 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

Tarkastellaan esimerkkinä tehtävää, jossa annettuna on joukko välejä taulukossa ja 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 syvemältä segmenttipuun mahdollisuuksiin.

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 (b < c || a > d) return 0;
    if (a == c && b == d) return p[k];
    int w = (d-c+1)/2;
    return summa(a, min(c+w-1,b), 2*k, c, c+w-1) +
           summa(max(c+w,a), b, 2*k+1, c+w, d);
}
```

Nyt välin $[a, b]$ summan saa laskettua kutsumalla funktiota näin:

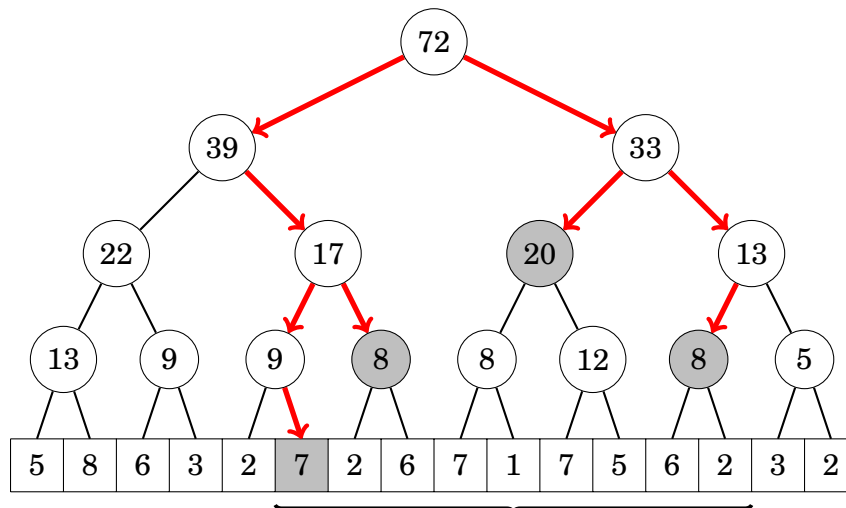
```
int s = summa(a, b, 1, 0, N-1);
```

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 kokonaan tai osittain välin $[c, d]$ sisällä, haku jatkuu rekursiivisesti välin $[c, d]$ vasemmasta ja oikeasta puoliskosta. Kummankin puoliskon koko on $w = \frac{1}{2}(d - c + 1)$, joten vasen puolisko kattaa välin $[c, c + w - 1]$ ja oikea puolisko kattaa välin $[c + w, d]$.

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 $O(\log n)$.

28.1 Laiska eteneminen

Laiska eteneminen 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.

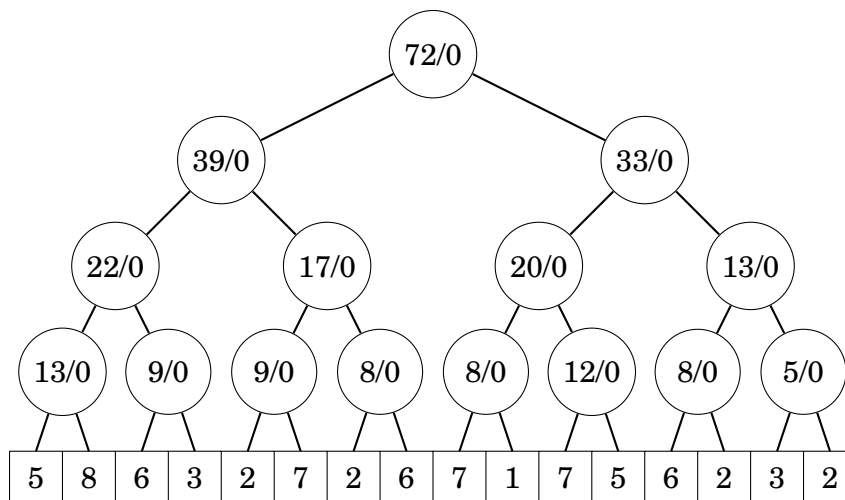
Välin muutostapa voi olla joko *lisäys* tai *asetus*. Lisäyksessä välin jokaiseen alkioon lisätään tietty arvo, ja asetuksessa välin jokainen alkio saa tietyn arvon. Kummankin operaation toteutus on melko samanlainen, ja puu voi myös sallia samaan aikaan molemmat muutostavat.

Laiska segmenttipuu

Tarkastellaan esimerkkinä tilannetta, jossa segmenttipuun tulee toteuttaa seuraavat operaatiot:

- lisää jokaisen välin $[a, b]$ alkioon arvo x
- laske välin $[a, b]$ alkioden summa

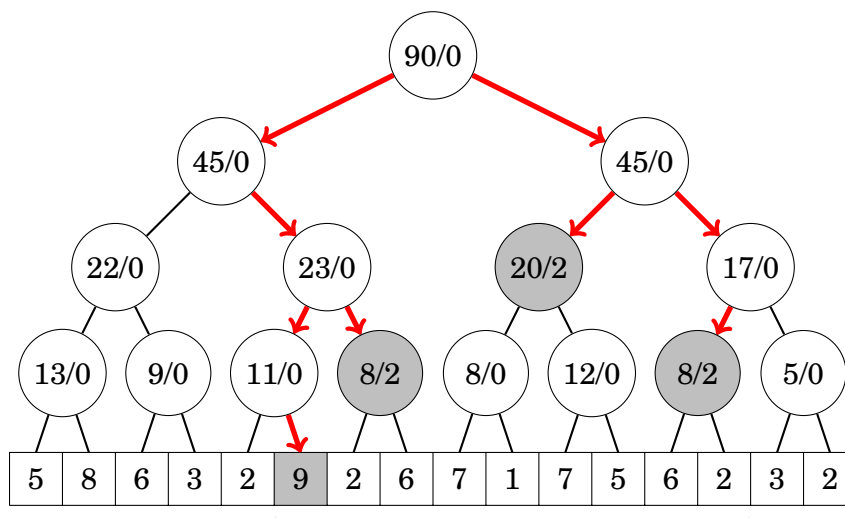
Toteutamme puun, jonka jokaisessa solmussa on kaksi arvoa s/z : välin lukujen summa s , kuten tavallisessa segmenttipuussa, sekä laiska muutos z , joka tarkoittaa, että kaikkiin välin lukuihin tulee lisätä z . Seuraavassa puussa jokaisessa solmussa $z = 0$ eli mitään muutoksia ei ole kesken.



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 z -arvo kasvaa x :llä ja kulku pysähtyy. Jos taas väli $[c, d]$ kuuluu osittain välille $[a, b]$, solmun s -arvo kasvaa hx :llä, missä h on välien $[a, b]$ ja $[c, d]$ yhteisen osan pituus, ja kulku jatkuu rekursiivisesti alaspäin.

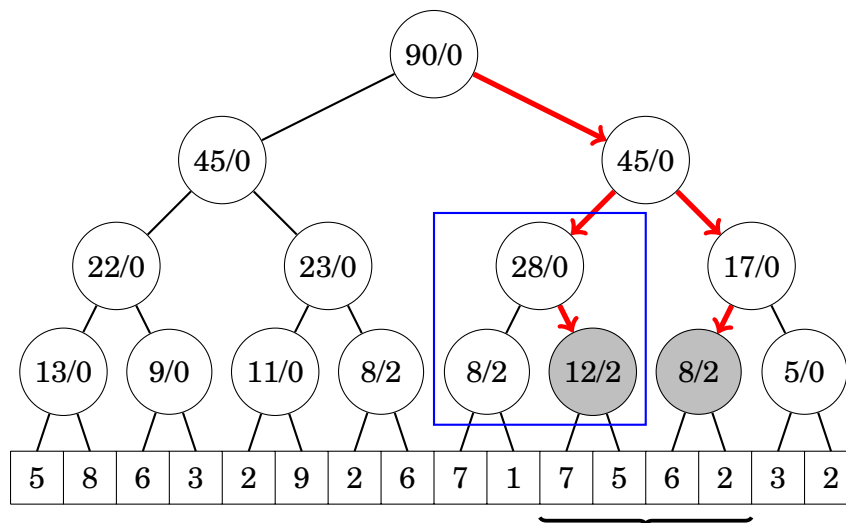
Kasvatetaan esimerkiksi puun alle merkittyä väliä 2:lla:



Välin $[a, b]$ summan laskenta tapahtuu myös kulkuna puun juuresta lehtiä kohti. Jos solmun väli $[c, d]$ kuuluu kokonaan väliin $[a, b]$, kyselyn summaan lisätään solmun s -arvo sekä mahdollinen z -arvon tuottama lisäys. Muussa tapauksessa kulku jatkuu rekursiivisesti alaspäin solmun lapsiin.

Aina ennen solmun käsittelyä siinä mahdollisesti oleva laiska muutos välitetään tasoa alemmas. Tämä tapahtuu sekä välin muutoskyselyssä että summakyselyssä. Ideana on, että laiska muutos etenee alaspäin vain silloin, kun tämä on välttämätöntä, jotta puun käsittely on tehokasta.

Seuraava kuva näyttää, kuinka äskeinen puu muuttuu, kun siitä lasketaan puun alapuolelle merkityn välin summa:



Tämän kyselyn seurauksena laiska muutos eteni alaspäin laatikolla ympäröidyssä puun osassa. Laiskaa muutosta täytyi viedä alaspäin, koska kyselyn kohteena oleva väli osui osittain laiskan muutoksen välille.

Huomaa, että joskus puussa olevia laiskoja muutoksia täytyy yhdistää. Näin tapahtuu silloin, kun solmussa on valmiina laiska muutos ja siihen tulee ylhäältä toinen laiska muutos. Tässä tapauksessa yhdistäminen on helppoa, koska muutokset z_1 ja z_2 aiheuttavat yhdessä muutoksen $z_1 + z_2$.

Polynomimuutos

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 välin $[a, b]$ kohdan 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

$$s + \sum_{x=0}^{d-c} z_k x^k + z_{k-1} x^{k-1} + \dots + z_0,$$

jonka saa laskettua tehokkaasti osissa summakaavoilla. 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 $p'(x) = p(x + h)$ on aina samanasteinen polynomi kuin $p(x)$. Esimerkiksi jos $p(x) = t_2 x^2 + t_1 x - t_0$, niin

$$p'(x) = t_2(x + h)^2 + t_1(x + h) - t_0 = t_2 x^2 + (2ht_2 + t_1)x + t_2 h^2 + t_1 h - t_0.$$

28.2 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.

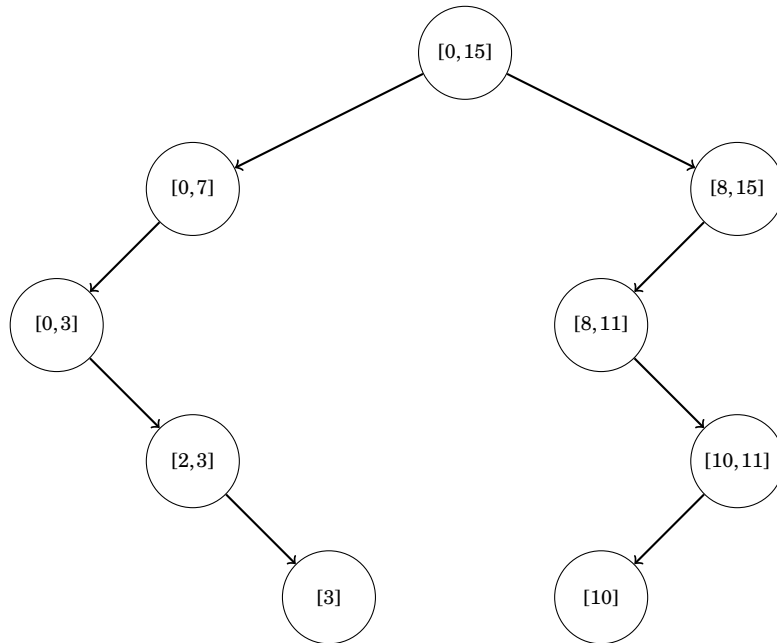
Tämän jälkeen solmuja voi käsitellä seuraavasti:

```
// uuden solmun luonti
node *s = new node(0, 0, 15);
// kentän muuttaminen
s->x = 5;
```

Harva segmenttipuu

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.

Harva segmenttipuu 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.

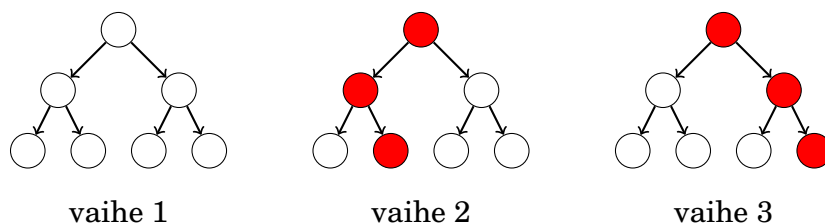
Huomaa, että jos kaikki tarvittavat indeksit ovat tiedossa algoritmin alussa, dynaamisen segmenttipuun sijasta voi käyttää tavallista segmenttipuuta ja indeksien pakkausta (luku 9.4). Tämä ei ole kuitenkaan mahdollista, jos indeksit syntyvät vasta algoritmin aikana.

Persistentti segmenttipuu

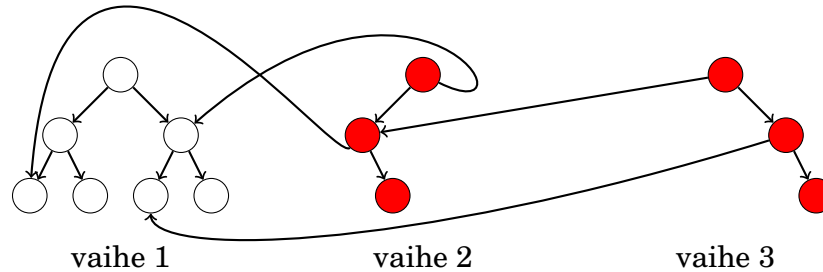
Dynaamisen toteutuksen avulla on myös mahdollista luoda **persistentti segmenttipuu**, joka säilyttää puun muutoshistorian. Tällöin muistissa on jokainen segmenttipuun vaihe, joka on esiintynyt algoritmin suorituksen aikana.

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, joten muistia säästävä tapa tallentaa muutoshistoria on käyttää mahdollisimman paljon hyväksi puun vanhoja osia muutoksissa. Tässä tapauksessa muutoshistorian voi 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.

28.3 Tietorakenteet

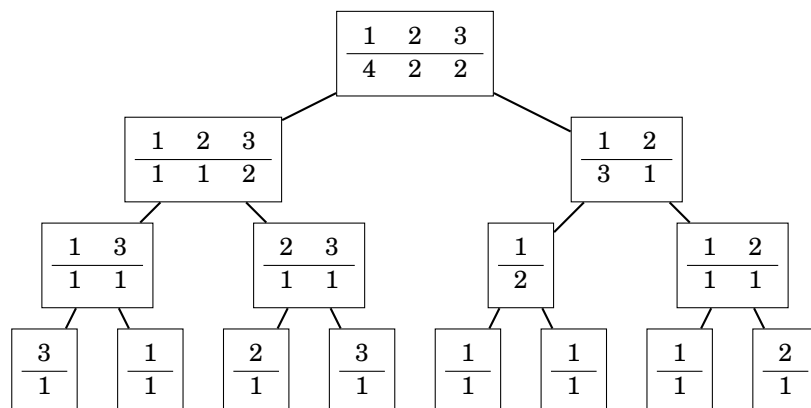
Segmenttipuun solmussa voi olla yksittäisen arvon sijasta myös jokin tietorakenne, joka pitää yllä tietoa solmua vastaavasta välistä. Tällöin segmenttipuun operaatiot vievät aikaa $O(f(n) \log n)$, missä $f(n)$ on yksittäisen solmun tietorakenteen käsittelyyn kuluva aika.

Tarkastellaan esimerkkinä segmenttipuuta, jonka avulla voi laskea, montako kertaa luku x esiintyy taulukon välillä $[a, b]$. Esimerkiksi seuravassa taulukossa luku 1 esiintyy kolme kertaa merkityllä välillä:

3	1	2	3	1	1	1	2
---	---	---	---	---	---	---	---

Ideana on toteuttaa segmenttipuu, jonka jokaisessa solmussa on tietorakenne, josta voi kysyä, montako kertaa luku x esiintyy solmun välillä. Tällaisen segmenttipuun avulla vastaus kyselyyn syntyy laskemalla yhteen esiintymismäärät väleiltä, joista $[a, b]$ muodostuu.

Esimerkiksi yllä olevasta taulukosta syntyy seuraava segmenttipuu:



Sopiva tietorakenne segmenttipuun toteuttamiseen on hakemistorakenne, joka pitää kirjaa välillä esiintyvien lukujen määrästä. Esimerkiksi map-rakennetta käyttäen yhden solmun käsittely vie aikaa $O(\log n)$, minkä seurauksena 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 hakemistorakenteet sisältävät $O(n)$ lukua. Tässä oletuksena on, että k luvun tallentaminen hakemistoon vie tilaa $O(k)$.

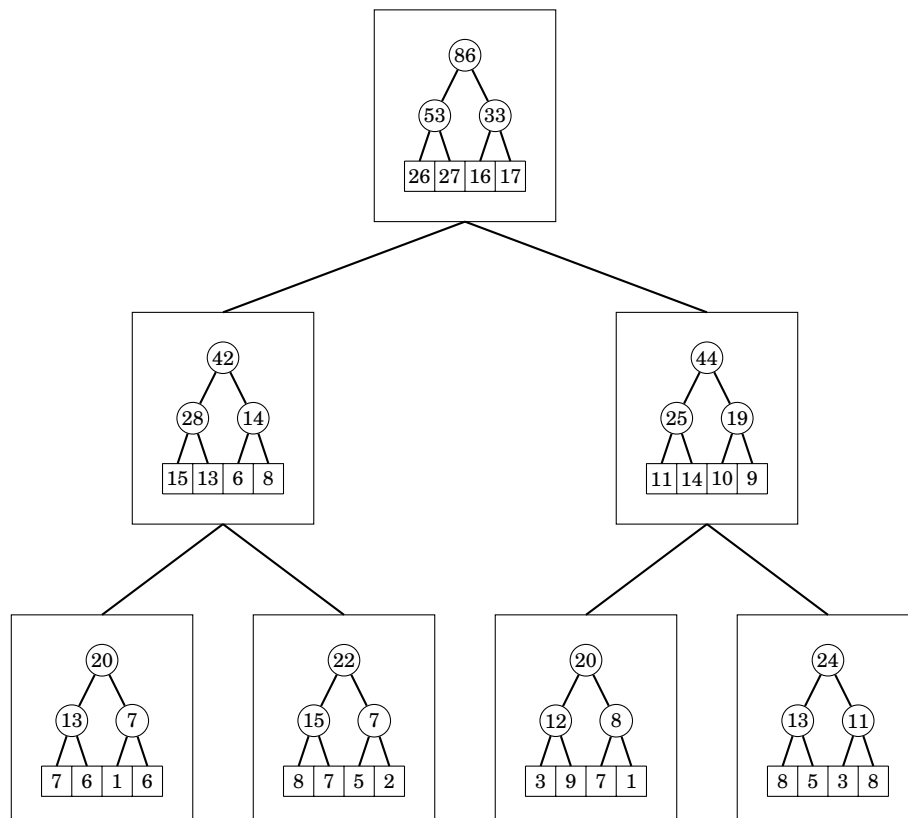
28.4 Kaksiulotteisuus

Kaksiulotteinen segmenttipuu mahdollistaa kaksiulotteisen taulukon suorakulmaisia alueita koskevat kyselyt. Tällaisen segmenttipuun voi toteuttaa sisäkkäisinä segmenttipuina: suuri puu vastaa taulukon rivejä ja sen kussakin solmussa on pieni puu, joka vastaa taulukon 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:



Kaksiulotteisen 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

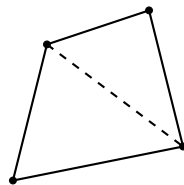
Geometria

Geometrian tehtävissä on usein haasteena keksiä, mistä suunnasta ongelmaa kannattaa lähestyä, jotta ratkaisun saa koodattua mukavasti ja erikoistapauksia tulee mahdollisimman vähän.

Tarkastellaan esimerkkinä tehtävää, jossa annettuna on nelikulmion kulmapisteet ja tehtävänä on laskea sen pinta-ala. Esimerkiksi syötteenä voi olla seuraavanlainen nelikulmio:



Yksi tapa lähestyä tehtävää on jakaa nelikulmio kahdeksi kolmioksi vetämällä jakoviiva kahden vastakkaisen kulmapisteen välille:



Tämän jälkeen riittää laskea yhteen kolmioiden pinta-alat. Kolmion pinta-alan voi laskea esimerkiksi **Heronin kaavalla**

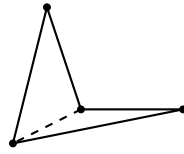
$$\sqrt{s(s-a)(s-b)(s-c)},$$

kun kolmion sivujen pituudet ovat a , b ja c ja $s = (a + b + c)/2$.

Tämä on mahdollinen tapa ratkaista tehtävä, mutta siinä on ongelma: miten löytää kelvollinen tapa vetää jakoviiva? Osoittautuu, että mitkä tahansa vastakkaiset pisteet eivät kelpaa. Esimerkiksi seuraavassa nelikulmiossa jakoviiva menee nelikulmion ulkopuolelle:



Toinen tapa vetää jakoviiva on kuitenkin toimiva:



Ihmiselle on selvää, kumpi jakoviiva jakaa nelikulmion kahdeksi kolmioksi, mutta tietokoneen kannalta tilanne on hankala.

Osoittautuu, että tehtävän ratkaisuun on olemassa paljon helpommin toteutettava tapa, jossa ei tarvitse miettiä erikoistapauksia. Nelikulmion pinta-alan laskemiseen on nimittäin yleinen kaava

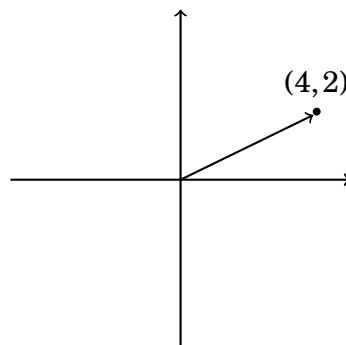
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

kun kulmapisteet ovat (x_1, y_1) , (x_2, y_2) , (x_3, y_3) ja (x_4, y_4) . Tämä kaava on helppo laskea, siinä ei ole erikoistapauksia ja osoittautuu, että kaava on mahdollista yleistää *kaikille* monikulmioille.

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. Luokan 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 koordinaatin tyyppi C on tilanteesta riippuen `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 koordinaatin tyyppinä on `long double`.

Funktio `abs(v)` laskee vektorin $v = (x, y)$ pituuden $|v|$ kaavalla $\sqrt{x^2 + y^2}$. Silä voi laskea myös pisteiden (x_1, y_1) ja (x_2, y_2) etäisyyden, koska pisteiden etäisyys on sama kuin vektorin $(x_2 - x_1, y_2 - y_1)$ pituus. Joskus hyödyllinen on myös funktio `norm(v)`, joka laskee vektorin $v = (x, y)$ pituuden neliön $|v|^2$.

Seuraava koodi laskee pisteiden $(4, 2)$ ja $(3, -1)$ etäisyyden:

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.60555
```

Funktio `arg(v)` laskee vektorin $v = (x, y)$ kulman radiaaneina suhteessa x-akseliin. Radiaaneina ilmoitettu kulma r vastaa asteina kulmaa $180r/\pi$ astetta. Jos vektori osoittaa suoraan oikealle, sen kulma on 0. Kulma kasvaa vastapäivään ja vähenee myötäpäivään liikuttaessa.

Funktio `polar(s, a)` muodostaa vektorin, jonka pituus on s ja joka osoittaa kulmaan a . Lisäksi vektoria pystyy kääntämään kulman a verran kertomalla se vektorilla, jonka pituus on 1 ja kulma on a .

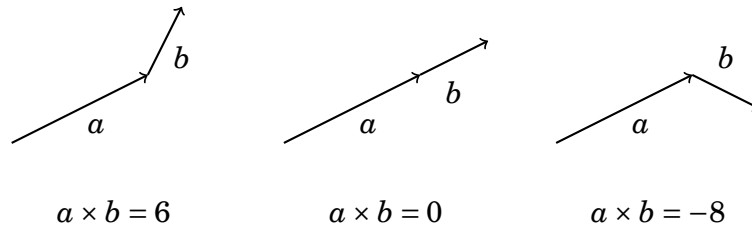
Seuraava koodi laskee vektorin $(4, 2)$ kulman, kääntää sitä sitten $1/2$ radiaania vastapäivään ja laskee uuden kulman:

```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0,0.5);  
cout << arg(v) << "\n"; // 0.963648
```

29.2 Pisteet ja suorat

Vektorien $a = (x_1, y_1)$ ja $b = (x_2, y_2)$ **ristitulo** $a \times b$ lasketaan kaavalla $x_1y_2 - x_2y_1$. 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:



Esimerkiksi vasemmassa kuvassa $a = (4, 2)$ ja $b = (1, 2)$. Seuraava koodi laskee vastaavan ristitulon luokkaa complex käyttäen:

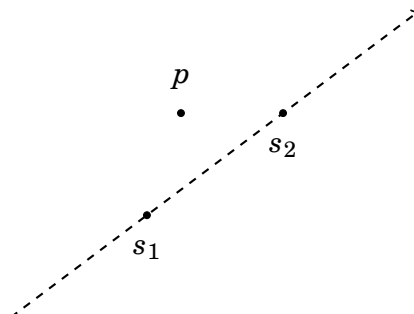
```
P a = {4,2};  
P b = {1,2};  
C r = (conj(a)*b).Y; // 6
```

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 $x_1y_2 - x_2y_1$.

Pisteen sijainti suoraan nähden

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:

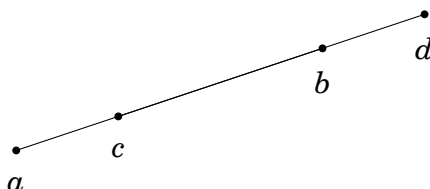


Nyt ristitulo $(p - s_1) \times (p - s_2)$ kertoo, kummalla puolella suoraa piste p sijaitsee. Jos ristitulo on positiivinen, piste p on suoran vasemmalla puolella, ja jos ristitulo on negatiivinen, piste p on suoran oikealla puolella. Jos taas ristitulo on nolla, piste p on pisteiden s_1 ja s_2 kanssa suoralla.

Janojen leikkaaminen

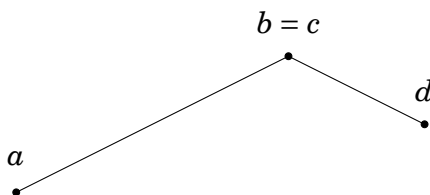
Tarkastellaan tilannetta, jossa tasossa on kaksi janaa ab ja cd ja tehtävänä on selvittää, leikkaavatko janat. Mahdolliset tapaukset ovat seuraavat:

Tapaus 1: Janat ovat samalla suoralla ja ne sivuavat toisiaan. Tällöin janoilla on ääretön määrä leikkauspisteitä. Esimerkiksi seuraavassa kuvassa janat leikkaavat pisteestä c pisteeseen b :



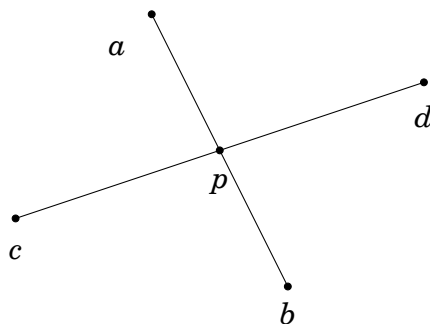
Tässä tapauksessa ristitulon avulla voi tarkastaa, ovatko kaikki pisteet samalla suoralla. Tämän jälkeen riittää järjestää pisteet ja tarkastaa, menevätkö janat toistensa päälle.

Tapaus 2: Janoilla on yhteinen päätepiste, joka on ainoa leikkauspiste. Esimerkiksi seuraavassa kuvassa janat leikkaavat pisteessä $b = c$:



Tämä tapaus on helppoa tarkastaa, koska mahdolliset vaihtoehdot yhteiselle päätepisteelle ovat $a = c$, $a = d$, $b = c$ ja $b = d$.

Tapaus 3: Janoilla on yksi leikkauspiste, joka ei ole mikään janojen päätepisteistä. Seuraavassa kuvassa leikkauspiste on p :



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. Niinpä janojen leikkaamisen voi tarkastaa ristitulon avulla.

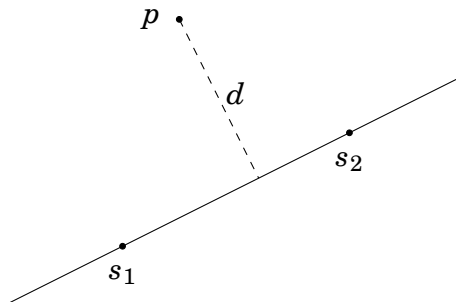
Pisteen etäisyys suorasta

Kolmion pinta-ala voidaan lausua ristitulon avulla

$$\frac{|(a - c) \times (b - c)|}{2},$$

missä a , b ja c ovat kolmion kärkipisteet.

Tämän kaavan avulla on mahdollista laskea, kuinka kaukana annettu piste on suorasta. Esimerkiksi seuraavassa kuvassa d on lyhin etäisyys pisteestä p suoralle, jonka määrittävät pisteet s_1 ja s_2 :

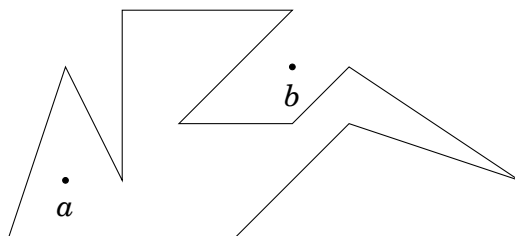


Pisteiden s_1 , s_2 ja p muodostaman kolmion pinta-ala on toisaalta $\frac{1}{2}|s_2 - s_1|d$ ja toisaalta $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Niinpä haluttu etäisyys on

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

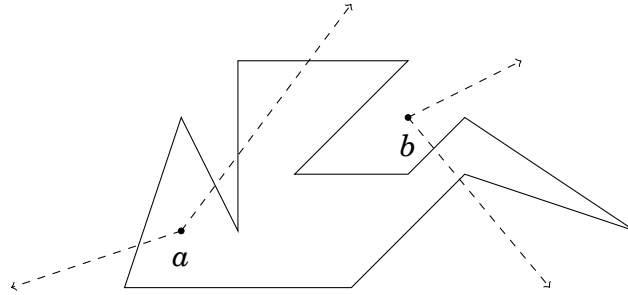
Piste monikulmiossa

Tarkastellaan sitten tehtävää, jossa tulee selvittää, onko annettu piste monikulmion sisäpuolella vai ulkopuolella. Esimerkiksi seuraavassa kuvassa piste a on sisäpuolella ja piste b on ulkopuolella.



Kätevä ratkaisu tehtävään on lähettää pisteestä säde satunnaiseen suuntaan ja laskea, montako kertaa se osuu monikulmion reunaan. Jos kertoja on pariton määrä, niin piste on sisäpuolella, ja jos taas kertoja on parillinen määrä, niin piste on ulkopuolella.

Äskeisessä tilanteessa säteitä voisi lähteä seuraavasti:



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.

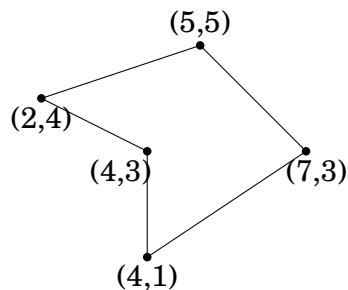
29.3 Monikulmion pinta-ala

Yleinen kaava monikulmion pinta-alan laskemiseen on

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

kun kärkipisteet ovat $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ järjestettynä niin, että p_i ja p_{i+1} ovat vierekkäiset kärkipisteet monikulmion reunalla ja ensimmäinen ja viimeinen kärkipiste ovat samat eli $p_1 = p_n$.

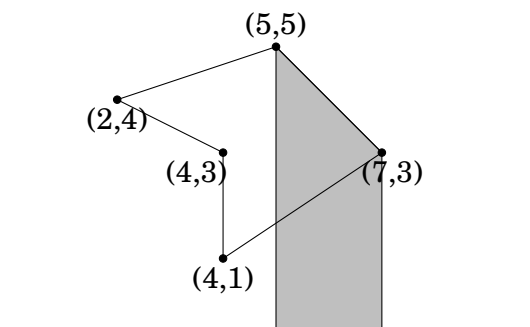
Esimerkiksi monikulmion



pinta-ala on

$$\frac{|(2 \cdot 5 - 4 \cdot 5) + (5 \cdot 3 - 5 \cdot 7) + (7 \cdot 1 - 3 \cdot 4) + (4 \cdot 3 - 1 \cdot 4) + (4 \cdot 4 - 3 \cdot 2)|}{2} = 17/2.$$

Kaavassa on ideana käydä läpi puolisuunnikkaita, joiden yläreuna on yksi monikulmion sivuista ja alareuna on vaakataso. Esimerkiksi:



Tällaisen puolisuunnikkaan pinta-ala on

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

kun kärkipisteet ovat p_i ja p_{i+1} . Jos $x_{i+1} > x_i$, niin pinta-ala on positiivinen, ja jos $x_{i+1} < x_i$, niin pinta-ala on negatiivinen.

Monikulmion pinta-ala saadaan laskemalla yhteen kaikkien tällaisten puolisuunnikkaiden pinta-alat, mistä tulee:

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

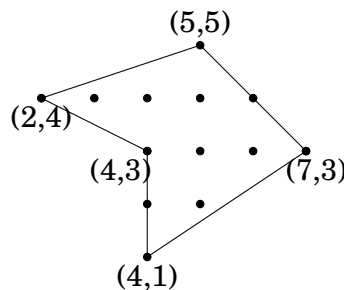
Huomaa, että pinta-alan kaavassa on itseisarvo, koska monikulmion kiertosuunnasta (myötä- tai vastapäivään) riippuen tuloksena oleva pinta-ala on joko positiivinen tai negatiivinen.

Pickin lause on vaihtoehtoinen tapa laskea monikulmion pinta-ala, kun kaikki monikulmion 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 $6 + 7/2 - 1 = 17/2$.

29.4 Etäisyysmitat

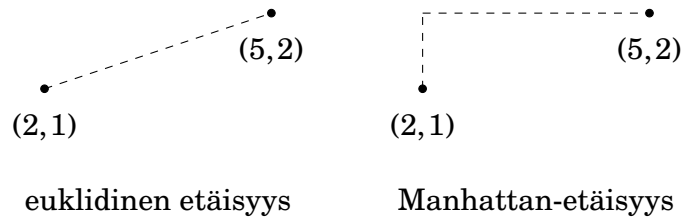
Etäisyysmitta määrittää tavan laskea kahden pisteen etäisyys. Tavallisimmin geometriassa käytetty etäisyysmitta on **euklidinen etäisyys**, jolloin pisteiden (x_1, y_1) ja (x_2, y_2) etäisyys on

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Vaihtoehtoinen etäisyysmitta on **Manhattan-etäisyys**, jota käyttäen pisteiden (x_1, y_1) ja (x_2, y_2) etäisyys on

$$|x_1 - x_2| + |y_1 - y_2|.$$

Esimerkiksi kuvaparissa



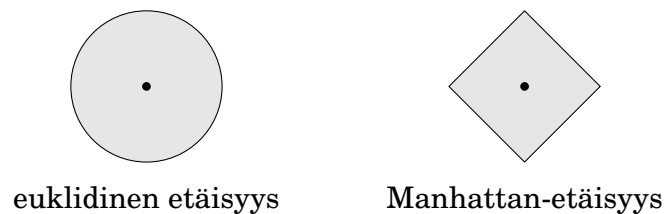
pisteiden Euklidinen etäisyys on

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

ja pisteiden Manhattan-etäisyys on

$$|5-2| + |2-1| = 4.$$

Seuraava kuvapari näyttää alueen, joka on pisteestä etäisyyden 1 sisällä käyttäen euklidista ja Manhattan-etäisyyttä:



Joidenkin ongelmien ratkaiseminen on helpompaa, jos käytössä on Manhattan-etäisyys euklidisen etäisyyden sijasta. Tarkastellaan esimerkkinä tehtävää, jossa annettuna on n pistettä $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ ja tehtävänä on laskea, mikä on suurin mahdollinen etäisyys kahden pisteen välillä.

Tämän tehtävän ratkaiseminen tehokkaasti on vaikeaa, jos laskettavana on euklidinen etäisyys. Sen sijaan suurin Manhattan-etäisyys on helppoa selvittää, koska se on suurempi etäisyyksistä

$$\max A - \min A \quad \text{ja} \quad \max B - \min B,$$

missä

$$A = \{x_i + y_i : i = 1, 2, \dots, n\}$$

ja

$$B = \{x_i - y_i : i = 1, 2, \dots, n\}.$$

Tämä johtuu siitä, että Manhattan-etäisyys

$$|x_a - x_b| + |y_a - y_b|$$

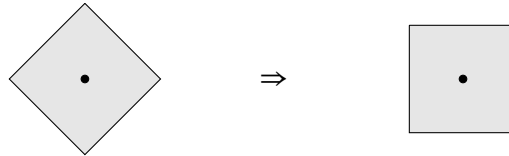
voidaan ilmaista muodossa

$$\begin{aligned} & \max(x_a - x_b + y_a - y_b, x_a - x_b - y_a + y_b) \\ &= \max(x_a + y_a - (x_b + y_b), x_a - y_a - (x_b - y_b)) \end{aligned}$$

olettaen, että $x_a \geq x_b$.

Kätevä tekniikka Manhattan-etäisyyden yhteydessä on myös kääntää koordinaatistoa 45 astetta niin, että pisteestä (x, y) tulee piste $(a(x+y), a(y-x))$, missä $a = 1/\sqrt{2}$. Kerroin a on valittu niin, että pisteiden etäisyydet säilyvät samana käännöksen jälkeen.

Tämän seurauksena pisteestä etäisyydellä d oleva alue on neliö, jonka sivut ovat vaaka- ja pystysuuntaisia, mikä helpottaa alueen käsittelyä.



Luku 30

Pyyhkäisyviiva

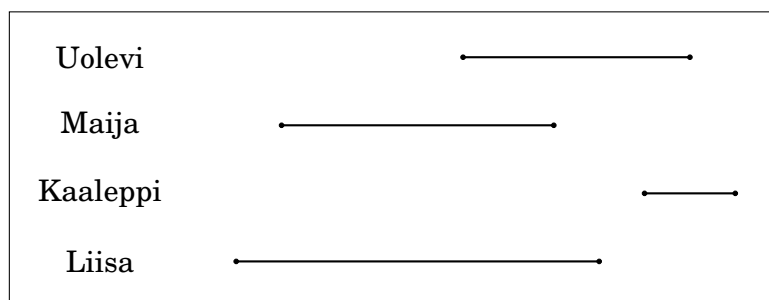
Pyyhkäisyviiva on tason halki kulkeva viiva, jonka avulla voi ratkaista useita geometrisia tehtäviä. Ideana on esittää tehtävä joukkona tapahtumia, jotka vastaavat tason pisteitä. Kun pyyhkäisyviiva törmää pisteeseen, tapahtuma käsitellään ja tehtävän ratkaisu edistyy.

Tarkastellaan esimerkkinä tekniikan käyttämisestä tehtävää, jossa yrityksessä on töissä n henkilöä ja jokaisesta henkilöstä tiedetään, milloin hän tuli töihin ja lähti töistä tietyinä päivinä. Tehtävänä on laskea, mikä on suurin määrä henkilöitä, jotka olivat samaan aikaan töissä.

Tehtävän voi ratkaista mallintamalla tilanteen niin, että jokaista henkilöä vastaa kaksi tapahtumaa: tuloaika töihin ja lähtöaika töistä. Pyyhkäisyviiva käy läpi tapahtumat aikajärjestyksessä ja pitää kirjaa, montako henkilöä on töissä milloinkin. Esimerkiksi tilannetta

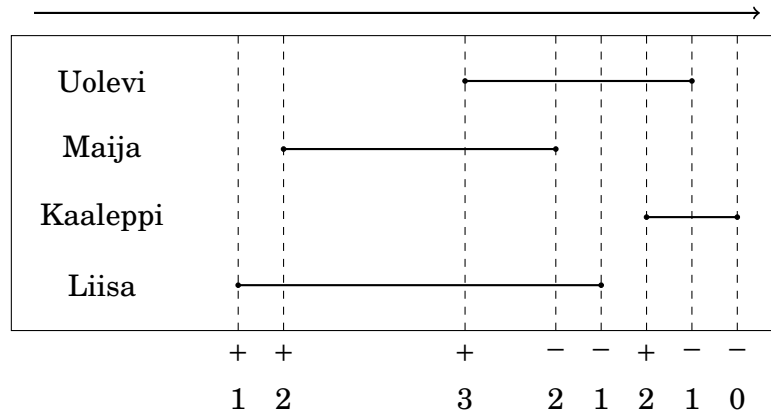
henkilö	tuloaika	lähtöaika
Uolevi	10	15
Maija	6	12
Kaaleppi	14	16
Liisa	5	13

vastaavat seuraavat tapahtumat:



Pyyhkäisyviiva käy läpi tapahtumat vasemmalta oikealle ja pitää yllä laskuria. Aina kun henkilö tulee töihin, laskurin arvo kasvaa yhdellä, ja kun henkilö lähtee töistä, laskurin arvo vähenee yhdellä. Tehtävän ratkaisu on suurin laskuri arvo pyyhkäisyviivan kulun aikana.

Pyyhkäisyviiva kulkee seuraavasti tason halki:



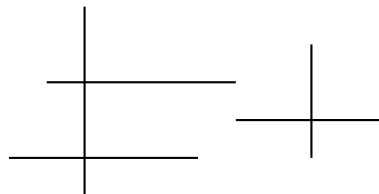
Kuvan alareunan merkinnät + ja - tarkoittavat, että laskurin arvo kasvaa ja vähenee yhdellä. Niiden alapuolella on laskurin uusi arvo. Laskurin suurin arvo 3 on voimassa Uolevi tulohetken ja Maijan lähtöhetken välillä.

Ratkaisun aikavaativuus on $O(n \log n)$, koska tapahtumien järjestäminen vie aikaa $O(n \log n)$ ja pyyhkäisyviivan läpikäynti vie aikaa $O(n)$.

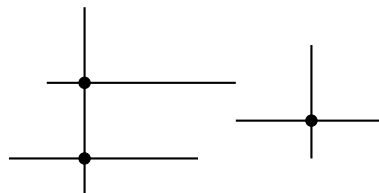
Tässä luvussa tutustumme kolmeen klassiseen tehtävään, jotka ratkeavat tehokkaasti pyyhkäisyviivan avulla.

30.1 Janojen leikkauspisteet

Annettuna on n janaa, joista jokainen on vaaka- tai pystysuuntainen. Tehtävänä on laskea tehokkaasti, monessako pisteessä kaksi janaa leikkaavat toisensa. Esimerkiksi tilanteessa



leikkauspisteitä on kolme:

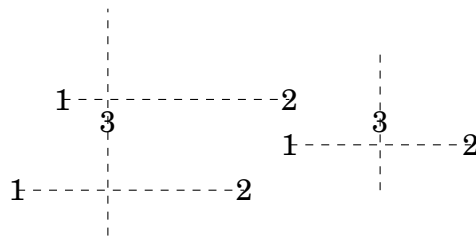


Tehtävä on helppoa ratkaista ajassa $O(n^2)$, koska riittää käydä läpi kaikki mahdolliset janaparit ja tarkistaa, moniko leikkaa toisiaan. Seuraavaksi ratkaisemme tehtävän ajassa $O(n \log n)$ pyyhkäisyviivan avulla.

Ideana on luoda janoista kolmenlaisia tapahtumia:

- (1) vaakajana alkaa
- (2) vaakajana päättyy
- (3) pystyjana

Äskeistä esimerkkiä vastaava pistejoukko on seuraava:



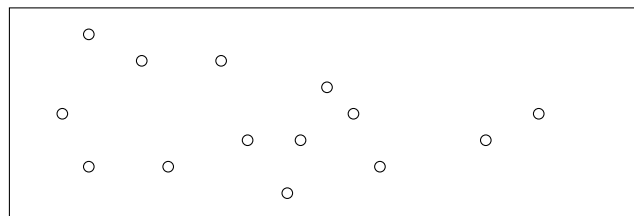
Algoritmi käy läpi pisteet vasemmalta oikealle ja pitää yllä tietorakennetta y-koordinaateista, joissa on tällä hetkellä aktiivinen vaakajana. Tapahtuman 1 kohdalla vaakajanan y-koordinaatti lisätään joukkoon ja tapahtuman 2 kohdalla vaakajanan y-koordinaatti poistetaan joukosta.

Algoritmi laskee janojen leikkauspisteet tapahtumien 3 kohdalla. Kun pysyjana 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.

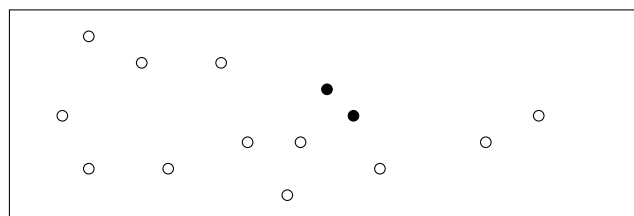
Sopiva tietorakenne vaakajanojen y-koordinaattien tallentamiseen on binääri-indeksipuu tai segmenttipuu, johon on tarvittaessa yhdistetty indeksien pakkaus. Tällöin jokaisen pisteen käsittely vie aikaa $O(\log n)$, joten algoritmin kokonaisaikavaativuus on $O(n \log n)$.

30.2 Lähin pistepari

Seuraava tehtävämme on etsiä n pisteen joukosta kaksi pistettä, jotka ovat mahdollisimman lähellä toisiaan. Esimerkiksi tilanteessa



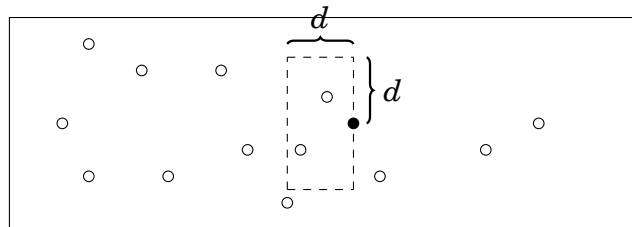
lähin pistepari on seuraava:



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.



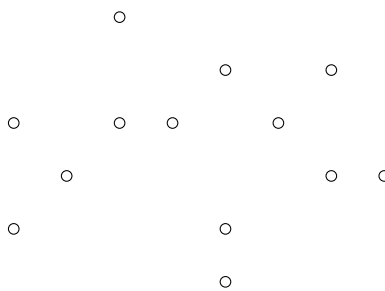
Algoritmin tehokkuus perustuu siihen, että d :n rajoittamalla alueella on aina vain $O(1)$ pistettä. Nämä pisteet pystyy käymään läpi $O(\log n)$ -aikaisesti pitämällä algoritmin aikana yllä joukkoa pisteistä, joiden x-koordinaatti on välillä $[x-d, x]$ ja jotka on järjestetty y-koordinaatin mukaan.

Algoritmin aikavaativuus on $O(n \log n)$, koska se käy läpi n pistettä ja etsii jokaiselle lähimmän edeltävän pisteen ajassa $O(\log n)$.

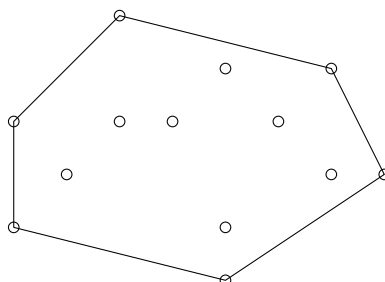
30.3 Konvekksi peite

Konvekksi peite 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



konvekksi peite on seuraava:

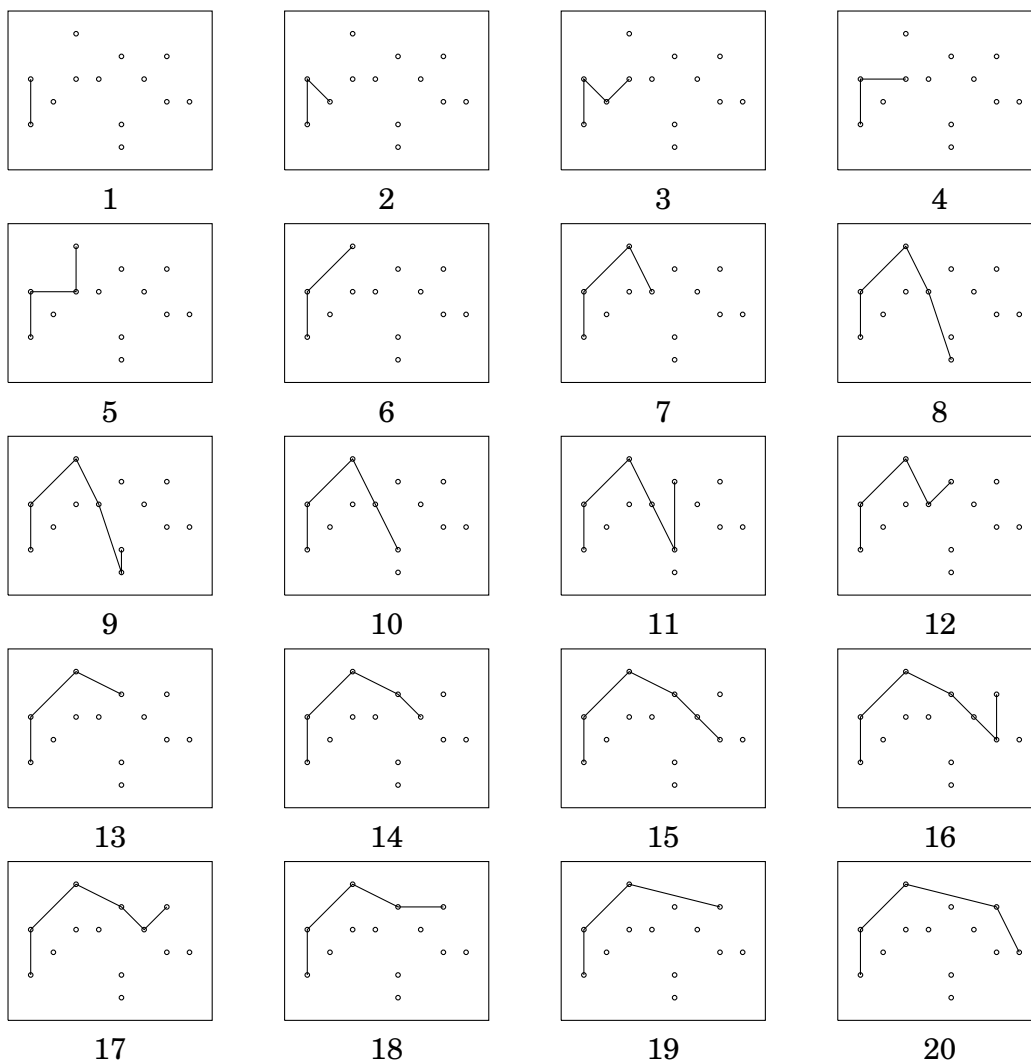


Tehokas ja helposti toteutettava menetelmä muodostaa konvekksi peite on **Andrew'n algoritmi**, jonka aikavaativuus on $O(n \log n)$.

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:



Algoritmi tarkastaa ristitulon avulla, muodostavatko kolme pistettä vasemmalle kääntyvän osan. Algoritmin aikavaativuus on $O(n \log n)$, koska pisteiden järjestäminen vie aikaa $O(n \log n)$ ja sen jälkeen konveksin peitteen muodostaminen vie aikaa $O(n)$.

Kirjallisuutta

Algoritmiikan oppikirjoja

- T. H. Cormen, C. E. Leiserson, R. L. Rivest ja C. Stein: *Introduction to Algorithms*, MIT Press, 2009 (3. painos)
- J. Kleinberg ja É. Tardos: *Algorithm Design*, Pearson, 2005
- S. S. Skiena: *The Algorithm Design Manual*, Springer, 2008 (2. painos)

Kisakoodauksen oppikirjoja

- K. Diks, T. Idziaszek, J. Łacki ja J. Radoszewski: *Looking for a Challenge?*, Varsovan yliopisto, 2012
- S. Halim ja F. Halim: *Competitive Programming*, 2013 (3. painos)
- S. S. Skiena ja M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003

Muita kirjoja

- J. Bentley: *Programming Pearls*, Addison-Wesley, 1999 (2. painos)

Hakemisto

- 2SAT-ongelma, 152
- 2SUM-ongelma, 77
- 3SAT-ongelma, 154
- 3SUM-ongelma, 78

- aakkosto, 231
- ahne algoritmi, 55
- aikavaativuus, 15
- alijono, 231
- alin yhteinen esivanhempi, 159
- alipuu, 127
- alkuluku, 185
- alkulukupari, 187
- alkuosa, 231
- alkutekijähajotelma, 185
- and-operaatio, 94
- Andrew'n algoritmi, 267
- antiketju, 182
- aritmeettinen summa, 10
- aste, 105

- Bellman–Fordin algoritmi, 117
- binomijakauma, 218
- binomikerroin, 196
- binääri-indeksipuu, 84
- binäärihaku, 30
- binäärikoodi, 60
- binääripuu, 131
- bitset, 39
- bittiesitys, 93
- bittijoukko, 39
- bittisiirto, 95
- Burnsiden lemma, 202

- Catalanin luku, 198
- Cayleyn kaava, 203
- complex, 254

- de Bruijnin jono, 169
- deque, 40

- determinantti, 207
- Dijkstran algoritmi, 120, 144
- Dilworthin lause, 182
- Diofantoksen yhtälö, 192
- Diracin lause, 168
- disjunktio, 12
- dynaaminen ohjelmointi, 63
- dynaaminen segmenttipuu, 247

- editointietäisyys, 71
- Edmonds–Karpin algoritmi, 174
- ehdollinen todennäköisyys, 215
- ekvivalenssi, 12
- epäjärjestys, 201
- Eratostheneen seula, 188
- erotus, 11
- eräkäsittely, 240
- esijärjestys, 132
- etäisyysmitta, 260
- Eukleideen algoritmi, 188, 192
- Eukleideen kaava, 194
- Euklidinen etäisyys, 260
- Eulerin kierros, 164
- Eulerin lause, 190
- Eulerin polku, 163
- Eulerin totienttifunktio, 189

- Fenwick-puu, 84
- Fermat'n pieni lause, 190
- Fibonaccin luku, 13, 194, 208
- Floyd–Warshallin algoritmi, 123
- Floydin algoritmi, 147
- Ford–Fulkersonin algoritmi, 172
- funktionaalinen verkko, 146

- geometria, 253
- geometrinen jakauma, 218
- geometrinen summa, 10
- Goldbachin konjektuuri, 187
- Grundy-luku, 227

Grundyn peli, 229
 hajautus, 232
 hajautusarvo, 232
 hakemisto, 36
 Hallin lause, 178
 Hamiltonin kierros, 168
 Hamiltonin polku, 167
 harmoninen summa, 11, 188
 harva segmenttipuu, 247
 Heronin kaava, 253
 heuristiikka, 170
 Hierholzerin algoritmi, 165
 Huffmanin koodaus, 61
 häviötila, 223

 implikaatio, 12
 indeksien pakkaus, 91
 inkluusio-ekskluusio, 200
 inversio, 25
 iteraattori, 37

 järjestystunnusluku, 220
 jakaja, 185
 jakauma, 217
 jakso, 231
 jaollisuus, 185
 jono, 41
 joukko, 11, 35
 joukko-oppi, 11
 juurellinen puu, 127
 juuri, 127
 jälkijärjestys, 132
 järjestäminen, 23

 Königin lause, 179
 kaari, 103
 kaarilista, 109
 kahden osoittimen tekniikka, 75
 kaksijakoinen verkko, 106
 kaksijakoisuus, 116
 kaksiulotteinen segmenttipuu, 250
 keko, 41
 kertoma, 13
 kierto, 231
 kiinalainen jäännöslause, 193
 Kirchhoffin lause, 211
 kofaktori, 207
 kokonaisluku, 6
 kombinatoriikka, 195
 kompleksiluku, 254
 komponentti, 104
 komponenttiverkko, 149
 konjunktio, 12
 koodisana, 60
 Kosarajun algoritmi, 150
 Kruskalin algoritmi, 134
 kuningatarongelma, 48
 kuplajärjestäminen, 23
 kuutiollinen algoritmi, 18
 kvanttori, 12
 käänteismatriisi, 208

 laajennettu Eukleideen algoritmi, 192
 Lagrangen lause, 193
 laiska eteneminen, 244
 laiska segmenttipuu, 244
 Laplacen matriisi, 212
 lapsi, 127
 Las Vegas -algoritmi, 219
 laskemisjärjestäminen, 27
 Legendren konjektuuri, 187
 lehti, 127
 leikkaus, 11, 172
 leikkauspiste, 257, 264
 leksikografinen järjestys, 231
 Levenšteinin etäisyys, 71
 leveyshaku, 113
 lineaarinen algoritmi, 18
 lineaarinen rekursioyhtälö, 208
 liukuluku, 7
 liukuva ikkuna, 79
 liukuvan ikkunan minimi, 79
 logaritmi, 14
 logaritminen algoritmi, 18
 logiikka, 12
 lomituserjestäminen, 25
 loppuosa, 231
 lukuteoria, 185
 luonnollinen logaritmi, 14
 lyhin polku, 117
 lähin pienempi edeltäjä, 78
 lähin pistepari, 265
 lähtöaste, 105
 läpimitta, 129

makro, 9
 maksimikysely, 81
 maksimiparitus, 177
 maksimivirtaus, 171
 Manhattan-etäisyys, 260
 map, 36
 Markovin ketju, 218
 matriisi, 205
 matriisipotenssi, 207
 matriisitulo, 206, 220
 merkkijono, 34, 231
 merkkijonohajautus, 232
 mex-funktio, 227
 minimikysely, 81
 minimileikkaus, 172, 175
 misääripeli, 226
 Mo'n algoritmi, 241
 modulolaskenta, 6, 189
 modulon käänteisluku, 190
 Monte Carlo -algoritmi, 219
 muistitaulukko, 65
 multinomikerroin, 198
 multiset, 35
 muutoshistoria, 248

 naapuri, 105
 negaatio, 12
 negatiivinen sykli, 119
 neliöjuorialgoritmi, 239
 neliöllinen algoritmi, 18
 neliömatriisi, 205
 Neperin luku, 14
 next_permutation, 47
 nim-peli, 225
 nim-summa, 225
 not-operaatio, 95
 NP-vaikea ongelma, 18

 odotusarvo, 216
 ohjelmointikieli, 3
 or-operaatio, 94
 Oren lause, 168
 osajono, 231
 osajoukko, 11, 45

 painotettu verkko, 105
 pair, 28

pakka, 40
 paritus, 177
 permutaatio, 47
 persistentti segmenttipuu, 248
 perusjoukko, 11
 peruuttava haku, 48
 Pickin lause, 260
 pienin solmupeite, 179
 pienin yhteinen moninkerta, 188
 pino, 40
 pisin nouseva alijono, 68
 piste, 254
 polku, 103
 polkupeite, 180
 polynominen algoritmi, 18
 polynominen hajautus, 232
 Prüfer-koodi, 204
 predikaatti, 12
 prefiksi, 231
 Primin algoritmi, 139
 prioriteettijono, 41
 priority_queue, 41
 puolivälihaku, 52
 puu, 104, 127
 puukysely, 155
 Pythagoraan kolmikko, 194
 pyyhkäisyviiva, 263

 queue, 41

 random_shuffle, 37
 ratsun kierros, 170
 rekursioyhtälö, 64, 208
 repunpakkaus, 70
 reverse, 37
 riippumaton joukko, 180
 riippumattomuus, 216
 ristitulo, 256

 satunnaisalgoritmi, 219
 satunnaismuuttuja, 216
 segmenttipuu, 86, 243
 set, 35
 seuraajaverkko, 146
 sisäjärjestys, 132
 skaalaava algoritmi, 175
 solmu, 103

solmupeite, 179
 solmutaulukko, 156
 sort, 28, 37
 SPFA-algoritmi, 120
 Sprague–Grundyn lause, 226
 stack, 40
 string, 34
 suffiksi, 231
 suhteellinen alkuluku, 189
 sulkulauseke, 199
 summakysely, 81
 summataulukko, 82
 suunnattu verkko, 104
 suurin alitaulukon summa, 19
 suurin riippumaton joukko, 180
 suurin yhteinen tekijä, 188
 sykli, 105, 115, 141, 147
 syklin tunnistaminen, 147
 syklitön verkko, 105
 syntymäpäiväparadoksi, 235
 syvyyshaku, 111
 syöte ja tuloste, 4
 säännöllinen verkko, 105

 tapauskäsittely, 241
 tasajakauma, 217
 tasoitettu analyysi, 75
 tekijä, 185
 tiedonpakkaus, 60
 tietorakenne, 33
 todennäköisyys, 213
 topologinen järjestys, 141
 transpoosi, 205
 trie, 231
 tuloaste, 105
 tuple, 29
 typedef, 8
 täydellinen luku, 186
 täydellinen paritus, 178
 täydellinen verkko, 105
 törmäys, 234

 union-find-rakenne, 137
 unordered_map, 36
 unordered_multiset, 35
 unordered_set, 35

 vaativuusluokka, 18

 vahvasti yhtenäinen komponentti, 149
 vahvasti yhtenäinen verkko, 149
 vakioaikainen algoritmi, 18
 vakiokerroin, 19
 vanhempi, 127
 vector, 33
 vektori, 33, 205, 254
 verkko, 103
 vertailufunktio, 29
 vertailuoperaattori, 28
 vieruslista, 106
 vierusmatriisi, 108
 virittävä puu, 133, 211
 virtaus, 171
 voittotila, 223
 välikysely, 81
 väritys, 106, 221

 Warnsdorffin sääntö, 170
 Wilsonin lause, 194

 xor-operaatio, 94

 yhdiste, 11
 yhtenäinen verkko, 104
 yhtenäisyys, 115
 ykkösmatriisi, 206
 yksinkertainen verkko, 106

 Z-algoritmi, 235
 Z-taulukko, 235
 Zeckendorfin lause, 194