Kisakoodarin käsikirja

Antti Laaksonen

4. helmikuuta 2016

Sisältö

Alkusanat v

I Perusasiat 1

1 Johdanto 3

2 Aikavaativuus 9

3 Järjestäminen 17

4 Tietorakenteet 23

5 Peruuttava haku 31

6 Ahneet algoritmit 35

7 Dynaaminen ohjelmointi 41

8 Taulukon käsittely 49

9 Segmenttipuu 53

10 Bittien käsittely 59

II Verkkoalgoritmit 65

11 Verkkojen perusteet 67

12 Verkon läpikäynti 75

13 Lyhimmät polut 81

14 Puiden käsittely 89

15 Virittävät puut 95

16 Syklittömät verkot 101

17 Vahvasti yhtenäisyys 105

18 Puukyselyt **111**

19 Polut ja kierrokset **115**

20 Virtauslaskenta **121**

Alkusanat

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

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

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

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

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

Osa I

Perusasiat

Luku 1

Johdanto

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

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

1.1 Ohjelmointikieli

Tällä hetkellä yleisimmät kisaohjelmoinnissa käytetyt kielet ovat C++, Python ja Java. Esimerkiksi vuoden 2015 Google Code Jamissa 3000 parhaan osallistujan joukossa 75 % käytti C++:aa, 15 % käytti Pythonia ja 11 % käytti Javaa. Jotkut käyttivät myös useita näistä kielistä.

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

Paras ratkaisu on silti hallita useita kieliä ja tuntea niiden edut. Esimerkiksi jos tehtävässä täytyy käsitellä suuria kokonaislukuja, Python voi sopia ratkaisuun paremmin kuin C++, koska Python tukee suoraan suuria kokonaislukuja. Toisaalta tehtävien suunnittelijat yrittävät yleensä laatia sellaisia tehtäviä, ettei tietyn kielen käyttämisestä ole merkittävää hyötyä.

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

1.2 Syöte ja tuloste

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

1.2.1 `cin` ja `cout`

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

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

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

Vastaavasti tulostaminen tapahtuu `cout`-virran kautta:

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

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

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

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

1.2.2 `scanf` ja `printf`

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

Seuraava koodi lukee kokonaislukuja syötteestä:

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

Seuraava koodi taas tulostaa kokonaisluvun:

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

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

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

1.2.3 Rivin lukeminen

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

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

1.2.4 Tiedostot

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

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

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

1.3 Lukujen käsittely

Yleisin kisaohjelmoinnissa tarvittava lukutyyppi on `int`, joka on 32-bittinen kokonaislukutyyppi. Jos muuttujan tyyppi on `int`, sen pienin ja suurin mahdollinen arvo ovat -2^{31} ja $2^{31}-1$ eli noin $-2\cdot10^9$ ja $2\cdot10^9$. Joskus kuitenkin tehtävissä esiintyy lukuja, jotka ovat `int`-tyypin arvoalueen ulkopuolella.

1.3.1 Suuret kokonaisluvut

Tyyppi `long long` on 64-bittinen kokonaislukutyyppi, jonka pienin ja suurin mahdollinen arvo ovat -2^{63} ja $2^{63}-1$ eli noin $-9\cdot10^{18}$ ja $9\cdot10^{18}$. Jos tehtävässä tarvitsee suuria kokonaislukuja, `long long` riittää yleensä.

Esimerkiksi seuraava koodi määrittelee `long long` -muuttujan:

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

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

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

```
typedef long long ll;
```

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

```
ll x;
```

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

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

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

1.3.2 Vastaus modulona

Joskus tehtävän vastaus on hyvin suuri kokonaisluku, mutta vastaus riittää tulostaa "modulo M" eli vastauksen jakojäännös luvulla M (esimerkiksi "modulo 10^9+7"). Ideana on, että vaikka todellinen vastaus voi olla suuri luku, tehtävässä riittää käyttää tyyppejä `int` ja `long long`.

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

Tärkeä modulon ominaisuus on, että yhteen- ja kertolaskussa modulon voi laskea ennen laskutoimitusta. Toisin sanoen seuraavat kaavat pätevät:

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

Tämän ansiosta jos vastaus muodostuu yhteen- ja kertolaskuista, jokaisen laskun vaiheen jälkeen voi ottaa modulon eivätkä luvut kasva liian suuriksi.

Esimerkiksi seuraava koodi laskee luvun 1000 kertoman modulo M:

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

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

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

Tämä on tarpeen kuitenkin vain silloin, kun koodissa on vähennyslaskuja tai modulo voi olla negatiivinen jostain muusta syystä.

1.3.3 Liukuluvut

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

Esimerkiksi seuraava koodi tuottaa yllättävän tuloksen:

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

Koodin tulostus on seuraava:

```
0.99999999999999988898
```

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

Liukulukuja on vaarallista vertailla ==-merkinnällä, koska vaikka luvut olisivat todellisuudessa samat, niissä voi olla pientä eroa pyöristysvirheiden 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 seuraavan tyylisesti ($\varepsilon = 10^{-9}$):

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

1.4 Matematiikka

Kisakoodarin käsikirja on yritetty kirjoittaa niin, että sen lukemiseen ei tarvitse laajoja matemaattisia esitietoja. Seuraavassa on joitakin matemaattisia käsitteitä ja kaavoja, joista on hyötyä kirjan lukemisessa.

1.4.1 Joukko-oppi

Joukko on kokoelma alkioita. Esimerkiksi joukko $S = \{2,4,7\}$ sisältää alkiot 2, 4 ja 7. Sama alkio ei voi esiintyä joukossa monta kertaa. Merkintä $\emptyset$ tarkoittaa tyhjää joukkoa. Joukon S koko eli alkoiden määrä on $|S|$.

Merkintä $x \in S$ tarkoittaa, että alkio x on joukossa S, ja merkintä $x \notin S$ tarkoittaa, että alkio x ei ole joukossa S. Yhdiste $A \cup B$ sisältää alkiot, jotka ovat ainakin toisessa joukoista A ja B. Leikkaus $A \cap B$ sisältää alkiot, jotka ovat molemmissa joukoista A ja B.

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

1.4.2 Summakaavat

Hyödyllisiä summakaavoja ovat:

$$\begin{aligned}
1+2+3+\cdots+n &= \frac{n(n+1)}{2} \\
1^2+2^2+3^2+\cdots+n^2 &= \frac{n(n+1)(2n+1)}{6} \\
2^0+2^1+2^2+\cdots+2^n &= 2^{n+1}-1 \\
x^0+x^1+x^2+\cdots+x^n &= \frac{x^{n+1}-1}{x-1} \\
x^1+2x^2+3x^3+\cdots+nx^n &= \frac{nx^{n+2}-(n+1)x^{n+1}+x}{(x-1)^2}
\end{aligned}$$

Joskus summan yhteydessä käytetään lyhennysmerkintää $\sum_{i=a}^{b}\ldots$, missä i saa arvot $a, a+1, \ldots, b$. Esimerkiksi $\sum_{i=1}^{n} i$ tarkoittaa $1+2+3+\cdots+n$.

1.4.3 Logaritmi

Logaritmi $\log_k(x)$ tarkoittaa, montako kertaa x täytyy jakaa k:lla, ennen kuin tuloksena on 1. Toisin sanoen jos $\log_k(x) = y$, niin $k^y = x$. Esimerkiksi $\log_2(32) = 5$, koska $2^5 = 32$. Algoritmien yhteydessä kantaluku k on yleensä 2, minkä vuoksi pelkkä $\log(x)$ voi tarkoittaa samaa kuin $\log_2(x)$.

Tärkeä logaritmin ominaisuus on, että $\log(x)$ on käytännössä aina pieni luku, vaikka x olisi suuri. Siksi $\log(x)$ askelta suorittava algoritmi on tehokas. On hyödyllistä muistaa, että $\log_2(1000) \approx 10$, $\log_2(10^6) \approx 20$ ja $\log_2(10^9) \approx 30$.

Luku 2

Aikavaativuus

Yleensä on helppoa suunnitella algoritmi, joka ratkaisee tehtävän hitaasti, mutta vaikeus piilee siinä, kuinka saada algoritmi toimimaan nopeasti. Aikavaativuus (*time complexity*) on kätevä tapa arvioida, kuinka nopeasti algoritmi toimii. Tässä luvussa tutustumme aikavaativuuden laskemisen perusteisiin.

2.1 Laskusäännöt

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

Silmukat

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

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

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

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

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

Suuruusluokka

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

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

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

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

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

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

Peräkkäisyys

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

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

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

Monta muuttujaa

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

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

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

Rekursio

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

Tarkastellaan esimerkiksi seuraavaa funktiota:

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

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

Tarkastellaan sitten seuraavaa funktiota:

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

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

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

Aikavaativuus voidaan laskea seuraavasti:

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

2.2 Vaativuusluokat

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

$O(1)$ (vakio)

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

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

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

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

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

$O(\log n)$ (logaritminen)

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

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

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

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

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

$O(n)$ (lineaarinen)

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

Seuraava lineaarinen algoritmi etsii pienimmän luvun taulukosta `t`:

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

Koodi käy läpi taulukon luvut vasemmalta oikealle ja tallentaa pienimmän luvun muuttujaan `p`.

$O(n \log n)$ (järjestäminen)

Aikavaativuus $O(n \log n)$ johtuu usein siitä, että algoritmin osana on syötteen järjestäminen. Tehokkaat järjestämisalgoritmit, kuten C++:n funktio `sort`, toimivat ajassa $O(n \log n)$. Järjestämisen jälkeen on usein helpompaa saada selville haluttu asia syötteestä.

Seuraava algoritmi tarkistaa, onko taulukossa `t` kahta samaa lukua:

```
sort(t, t+n);
bool s = false;
for (int i = 0; i < n-1; i++) {
    if (t[i] == t[i+1]) s = true;
}
```

Algoritmi järjestää ensin taulukon ajassa $O(n \log n)$. Tämän jälkeen algoritmi tarkistaa ajassa $O(n)$ kaikki taulukon vierekkäiset luvut. Jos taulukossa on kaksi samaa lukua, ne ovat vierekkäin järjestetyssä taulukossa. Lopuksi muuttuja `s` on `true`, jos taulukossa on kaksi samaa lukua, ja muuten `false`.

$O(n^2)$ (neliöllinen)

Neliöllinen aikavaativuus $O(n^2)$ näkyy usein siinä, että algoritmissa on kaksi sisäkkäistä silmukkaa. Neliöllinen algoritmi voi esimerkiksi käydä läpi kaikki tavat valita syötteestä kahden alkion pari:

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

Yksi usein esiintyvä tilanne on, että tehtävän ratkaisuun on olemassa suoraviivainen $O(n^2)$-algoritmi, mutta syötteen koko on niin suuri, että tämä ei riitä, vaan haasteena on keksiä $O(n)$- tai $O(n \log n)$-algoritmi.

$O(n^3)$ (kuutiollinen)

Kuutiollinen aikavaativuus $O(n^3)$ tarkoittaa, että algoritmissa voi olla kolme sisäkkäistä silmukkaa. Kuutiollinen algoritmi voi käydä läpi kaikki tavat valita syötteestä kolmen alkion joukko:

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

$O(2^n)$ (osajoukot)

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

$O(n!)$ (permutaatiot)

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

2.3 Nopeuden arviointi

Aikavaativuuden avulla voi arvioida *ennen* algoritmin koodaamista, onko algoritmi riittävän nopea tehtävän ratkaisuun, koska tehtävänanto kertoo, kuinka suuria syötteet voivat olla ja paljonko aikaa algoritmi saa käyttää.

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

- jos $n \approx 10$, algoritmi voi olla $O(n!)$
- jos $n \approx 20$, algoritmi voi olla $O(2^n)$
- jos $n \approx 250$, algoritmi voi olla $O(n^3)$
- jos $n \approx 2500$, algoritmi voi olla $O(n^2)$
- jos $n \approx 10^5$, algoritmi voi olla $O(n \log n)$
- jos $n \approx 10^6$, algoritmi voi olla $O(n)$

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

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

2.4 Esimerkki

Usein tehtävään on olemassa monia ratkaisuja, joilla on eri aikavaativuus. Näin on esimerkiksi seuraavassa klassisessa tehtävässä:

Tehtävä: Annettuna on taulukko, jossa on n lukua. Tehtäväsi on etsiä taulukosta yhtenäinen väli, jossa lukujen summa on mahdollisimman suuri.

Esimerkiksi taulukossa

−1	3	−4	8	5	−1	4	−2

suurin summa on 16, joka syntyy valitsemalla seuraava väli:

−1	3	−4	**8**	**5**	**−1**	**4**	−2

Ratkaisu 1

Yksi lähestymistapa tehtävään on käydä läpi kaikki taulukon välit, laskea jokaisen välin lukujen summa ja valita suurin summa. Algoritmissa on kolme sisäkkäistä silmukkaa, ja sen aikavaativuus on $O(n^3)$.

Seuraava koodi toteuttaa kuvatun algoritmin. Koodi olettaa, että taulukon sisältö on $\texttt{t}[0], \texttt{t}[1], \ldots, \texttt{t}[n-1]$. Koodi valitsee muuttujaan a välin aloituskohdan, muuttujaan b välin lopetuskohdan ja laskee kunkin välin lukujen summan muuttujaan u. Muuttuja s sisältää lopuksi suurimman summan taulukossa.

```
int s = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int u = 0;
        for (int c = a; c <= b; c++) {
            u += t[c];
        }
        s = max(s, u);
    }
}
```

Ratkaisu 2

Tehokkaampi ratkaisu on käydä kaikki taulukon välit läpi, mutta laskea summaa sitä mukaa kuin välin oikea reuna liikkuu eteenpäin. Tällöin algoritmissa on vain kaksi sisäkkäistä silmukkaa ja sen aikavaativuus on $O(n^2)$.

Seuraava koodi toteuttaa algoritmin:

```
int s = 0;
for (int a = 0; a < n; a++) {
    int u = 0;
    for (int b = a; b < n; b++) {
        u += t[b];
        s = max(s, u);
    }
}
```

Ratkaisu 3

Tehtävä on mahdollista ratkaista myös ajassa $O(n)$ käyttämällä yllättävän yksinkertaista algoritmia. Ideana on laskea jokaiseen taulukon kohtaan k, mikä on suurin mahdollinen summa tähän kohtaan päättyvässä välissä.

Mahdollisuuksia on kaksi: joko summa on pelkkä `t`$[k]$, tai sitten se on suurin kohtaan $k-1$ päättyvä summa, johon on lisätty `t`$[k]$.

Seuraava koodi toteuttaa algoritmin:

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

Luku 3

Järjestäminen

Järjestäminen (*sorting*) on keskeinen algoritmiikan ongelma. Esimerkiksi taulukosta $[4,1,5,6,2,5]$, tulee järjestämisen jälkeen $[1,2,4,5,5,6]$. Järjestäminen on usein tarvittava tekniikka sekä itsenäisenä algoritmina että monimutkaisemman algoritmin osana.

Tämän luvun alkuosa käsittelee järjestämisalgoritmien teoriaa, minkä jälkeen tutustumme lähemmin C++:n `sort`-funktioon. Luvun viimeisenä aiheena on binäärihaku, joka on tehokas algoritmi järjestetystä aineistosta etsimiseen.

3.1 Järjestämisalgoritmit

Järjestämiseen on kehitetty monia algoritmeja vuosien aikana. On helppoa toteuttaa järjestäminen ajassa $O(n^2)$, mutta tällaiset algoritmit eivät sovellu suuren tietomäärän järjestämiseen. Tehokkaat yleiset järjestämisalgoritmit toimivat ajassa $O(n \log n)$, ja niiden avulla suurikin taulukko järjestyy nopeasti.

3.1.1 Perusalgoritmit

Yksinkertaiset järjestämisalgoritmit toimivat ajassa $O(n^2)$. Esimerkki tällaisesta algoritmista on kuplajärjestäminen (*bubble sort*), jonka voi toteuttaa näin:

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

Kuplajärjestäminen muodostuu peräkkäisistä taulukon läpikäynneistä. Aina kun läpikäynnin aikana löytyy vierekkäinen alkiopari, jonka järjestys on väärä, algoritmi vaihtaa alkiot keskenään. Jokainen alkioparin vaihto vie taulukkoa lähemmäs järjestystä, kunnes lopulta koko taulukko on järjestyksessä.

3.1.2 Inversiot

Inversio (*inversion*) on taulukossa oleva lukupari, joka on väärässä järjestyksessä. Inversioiden määrä kuvaa, miten lähellä järjestystä taulukko on. Esimerkiksi taulukossa $[4,1,3,2,5]$ on 4 inversiota: $(4,1)$, $(4,3)$, $(4,2)$ ja $(3,2)$. Jos taulukko on järjestyksessä, inversioiden määrä on 0, ja jos järjestys on käänteinen, inversioiden määrä on $1+2+\cdots+(n-1) = n(n-1)/2$.

Inversioiden avulla voi laskea, montako vierekkäisen alkioparin vaihtoa tarvitaan taulukon järjestämiseen. Jokainen vaihto vähentää inversioiden määrää yhdellä, eli inversioiden määrä on sama kuin tarvittava vaihtojen määrä. Inversioiden suurin määrä $n(n-1)/2$ on luokkaa $O(n^2)$, minkä vuoksi minkä tahansa vierekkäisiä alkioita vaihtavan algoritmin aikavaativuus on ainakin $O(n^2)$.

3.1.3 Tehokkaat algoritmit

Tunnetuimmat tehokkaat järjestämisalgoritmit ovat pikajärjestäminen (*quicksort*) ja lomitusjärjestäminen (*mergesort*). Näiden algoritmien aikavaativuus on vain $O(n \log n)$. Molemmat algoritmit perustuvat rekursiiviseen ideaan, jossa järjestettävä taulukko jaetaan kahteen osaan, pienemmät taulukot järjestetään erikseen ja lopullinen taulukko saadaan yhdistämällä järjestetyt osat.

Voidaan osoittaa, että jos järjestämisalgoritmi perustuu alkioiden vertailuun, paras mahdollinen aikavaativuus on $O(n \log n)$. Joskus kuitenkin myös aikavaativuus $O(n)$ on mahdollinen. Esimerkiksi jos kaikki alkiot ovat pieniä kokonaislukuja, voidaan käyttää laskemisjärjestämistä (*counting sort*), joka laskee jokaisesta mahdollisesta alkiosta, montako kertaa se esiintyy taulukossa.

3.2 `sort`-funktio

Käytännössä järjestämistä ei kannata yleensä toteuttaa itse, vaan parempi ratkaisu on käyttää ohjelmointikielen standardikirjastoon kuuluvaa järjestämisalgoritmia. Esimerkiksi C++:ssa on tehokas `sort`-funktio, jolle annetaan parametreiksi järjestettävän välin alku- ja loppukohta.

3.2.1 Peruskäyttö

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

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

Vastaavasti seuraava koodi järjestää vektorin `v`:

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

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

Seuraava koodi järjestää merkkijonon s:

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

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

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

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

Toinen mahdollisuus on kääntää järjestys järjestämisen jälkeen:

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

3.2.2 Parien järjestäminen

Jos järjestettävät alkiot ovat pareja (pair), sort-funktio järjestää ne ensisijaisesti ensimmäisen kentän (first) mukaan ja toissijaisesti toisen kentän (second) mukaan. Seuraava koodi esittelee asiaa:

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

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

3.2.3 Tietueiden järjestäminen

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

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

```
struct P {
    int x, y;

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

3.2.4 Vertailufunktio

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

```
bool vrt(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(), vrt);
```

3.3 Binäärihaku

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

Esimerkiksi taulukossa

2	2	3	5	7	7	7	8
			↑				

luku 5 esiintyy nuolella merkityssä kohdassa.

Perinteinen tapa toteuttaa binäärihaku on jäljitellä sanan etsimistä sanakirjasta. Taulukon järjestyksen ansiosta taulukon keskikohdasta selviää, kummalla puolella taulukkoa etsittävä alkio on jne. Tämä menetelmä on kuitenkin yllättävän hankalaa toteuttaa toimivasti.

Seuraavaksi käsiteltävä binäärihaun toteutus käy taulukkoa läpi vasemmalta oikealle hyppien. Hyppyjen ansiosta suurinta osaa taulukon alkioista ei tarvitse tutkia ja aikavaativuudeksi tulee $O(\log n)$.

3.3.1 Toteutus

Seuraava koodi etsii järjestetystä taulukosta t alkiota k:

```
int x = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (x+b < n && t[x+b] <= k) x += b;
}
// nyt t[x] = k, jos k esiintyy taulukossa
```

Muuttuja x on taulukon käsittelykohta, ja muuttuja b ilmaisee hypyn pituuden. Aluksi hypyn pituus on $n/2$, sitten $n/4$, sitten $n/8$ jne., kunnes hypyn pituus

on lopuksi 1. Kullakin hypyn pituudella x kasvaa b:llä niin kauan, kuin $x+b$ on taulukon sisällä ja ehto $\texttt{t}[x+b] \le k$ pätee.

Algoritmi etsii suurimman x:n arvon, jolle pätee $\texttt{t}[x] \le k$, joten jos taulukossa esiintyy luku k, niin algoritmin päätteeksi $\texttt{t}[x] = k$. Haun aikavaativuus on $O(\log n)$, koska for-silmukka suoritetaan $O(\log n)$ kertaa ja while-silmukka suoritetaan korkeintaan kaksi kertaa kullakin b:n arvolla.

Samalla idealla voi myös laskea, montako kertaa alkio k esiintyy taulukossa:

```
int x1 = -1, x2 = -1;
for (int b = n/2; b >= 1; b /= 2) {
    while (x1+b < n && t[x1+b] < k) x1 += b;
    while (x2+b < n && t[x2+b] <= k) x2 += b;
}
// k esiintyy taulukossa x2-x1 kertaa
```

Nyt algoritmi etsii kaksi kohtaa taulukosta: x_1 on viimeinen kohta, jossa $\texttt{t}[x_1] < k$, ja x_2 on viimeinen kohta, jossa $\texttt{t}[x_2] \le k$. Erotus $x_2 - x_1$ kertoo, montako kertaa alkio k esiintyy taulukossa.

3.3.2 Muutoskohta

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

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

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

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

Nyt muutoskohta on mahdollista etsiä käyttämällä binäärihakua:

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

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

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

3.3.3 Huippukohta

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

Esimerkiksi taulukossa

2	4	5	6	7	4	2	1
				↑			

suurin alkio 7 on nuolella merkityssä kohdassa.

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

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

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

Luku 4

Tietorakenteet

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

Tämä luku esittelee joukon keskeisiä tietorakenteita, jotka ovat valmiina käytettävissä C++:ssa. Käsiteltävät tietorakenteet ovat taulukko, binääripuu, hajautustaulu ja keko. Myöhemmin kirjassa tutustumme pikkuhiljaa monimutkaisempiin ja harvemmin tarvittaviin tietorakenteisiin.

4.1 Taulukko

Taulukko (*array*) on yksinkertaisin ja useimmiten tarvittava tietorakenne. C++:n tavallinen taulukko on staattinen, mikä tarkoittaa, että taulukon koko täytyy päättää sen määrittelyssä. Lisäksi C++:n standardikirjastossa on dynaamisia taulukoita, joiden kokoa pystyy muuttamaan jälkeenpäin.

4.1.1 Staattinen taulukko

C++:n tavallinen taulukko on staattinen taulukko, jonka koko täytyy ilmoittaa määrittelyssä, eikä kokoa voi muuttaa myöhemmin.

Taulukon määrittely tapahtuu seuraavasti:

```
int t[100];
```

Jos taulukon määrittely on globaali, jokainen taulukon alkio on aluksi 0. Jos taas määrittely on funktion sisällä, taulukko täytyy tarvittaessa alustaa itse:

```
int t[100] = {0};
```

4.1.2 Vektori

Tavallisin C++:n dynaaminen taulukko on vektori (*vector*). Se on kuin tavallinen taulukko, mutta taulukon kokoa voi muuttaa.

Seuraava koodi määrittelee 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
```

Vastaavasti vektorista pystyy poistamaan viimeisen alkion näin:

```
v.pop_back();
```

Komennot `push_back` ja `pop_back` on toteutettu niin, että niiden aikavaativuus on keskimäärin $O(1)$, joten vektorin käyttäminen on tehokasta.

Seuraava koodi tulostaa kaikki vektorin alkiot:

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

Saman koodin voi myös kirjoittaa lyhyemmin näin:

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

Toinen tapa määritellä vektori on ilmoittaa sen koko määrittelyssä:

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

Myöhemmin vektorin kokoa voi muuttaa funktiolla `resize`:

```
v.resize(200);
```

4.1.3 Pakka

Vektorin rajoituksena on, että alkion lisäys ja poisto onnistuu tehokkaasti vain taulukon lopussa. Pakka (*deque*) on monimutkaisempi tietorakenne, jossa alkion lisäys ja poisto onnistuu tehokkaasti sekä taulukon alussa että lopussa.

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.push_front(4); // [4,3,5]
d.pop_front(); // [3,5]
```

Vektorin tavoin pakan operaatioiden aikavaativuus on keskimäärin $O(1)$. Pakka on kuitenkin jonkin verran vektoria raskaampi tietorakenne.

Pakan erikoitapauksia ovat tietorakenteet pino (*stack*) ja jono (*queue*). Pinossa lisäys ja poisto tapahtuvat aina rakenteen loppuun. Jonossa taas lisäys tapahtuu loppuun ja poisto tapahtuu alkuun.

4.1.4 Merkkijono

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

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

Seuraava koodi esittelee merkkijonon käyttämistä:

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

4.1.5 Välien käsittely

C++:n standardikirjasto sisältää monia valmiita funktioita taulukon välien käsittelyyn. Funktiot on toteutettu niin, että niille annetaan halutun välin alku- ja loppukohta. Yleensä halutaan, että funktiot kohdistuvat koko taulukkoon, jolloin väli alkaa kohdasta `begin` ja päättyy kohtaan `end`.

Esimerkiksi seuraava koodi ensin järjestää vektorin, sitten kääntää sen toisinpäin ja lopuksi sekoittaa järjestyksen:

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

Komentoja voi käyttää myös tavallisen taulukon yhteydessä seuraavasti:

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

4.2 Binäärihakupuu

Binäärihakupuu (*binary search tree*) on puurakenne, jonka keskeiset operaatiot ovat alkion lisäys, haku ja poisto. Tasapainoinen binäärihakupuu mahdollistaa kunkin operaation toteutuksen ajassa $O(\log n)$. Tällaisia puurakenteita ovat esimerkiksi AVL-puu, punamusta puu, treap-puu sekä splay-puu.

Tutustumme seuraavaksi C++:n tietorakenteisiin `set` ja `map`, jotka perustuvat binäärihakupuuhun. Rakenne `set` eli joukko sisältää kokoelman alkioita, joita pystyy käsittelemään tehokkaasti. Rakenne `map` eli hakemisto on taulukon yleistys, jossa alkioita voi etsiä avainten perusteella.

4.2.1 Joukko

C++:n `set`-rakenne on joukko, jonka tärkeimmät operaatiot ovat alkion lisäys, haku ja poisto. Joukko on toteutettu binäärihakupuun avulla niin, että kunkin operaation aikavaativuus on $O(\log n)$. Seuraava koodi esittelee rakennetta:

```
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
```

Komento `insert` lisää alkion joukkoon, ja funktio `erase` poistaa alkion joukosta. Komento `count` kertoo, montako kertaa alkio esiintyy joukossa. Komento palauttaa aina arvon 0 tai 1, koska sama alkio voi esiintyä vain kerran joukossa.

Jos alkio on valmiiksi joukossa, funktio `insert` ei tee mitään. Seuraava koodi havainnollistaa asiaa:

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

Rakenne `multiset` on kuin `set`, mutta sama alkio voi esiintyä monta kertaa:

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

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

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

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

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

4.2.2 Iteraattorit

Iteraattori (*iterator*) on tietorakenteen alkioon osoittava muuttuja, jota tarvitsee erityisesti `set`-rakenteen yhteydessä. Esimerkiksi iteraattori `s.begin()` osoittaa joukon `s` ensimmäiseen alkioon ja iteraattori `s.end()` osoittaa joukon `s` 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 joukon alkioon, kun taas `s.end()` osoittaa joukon ulkopuolelle. Iteraattoreiden rajaama joukon väli on siis puoliavoin.

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

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

Koodin voi kirjoittaa myös lyhyemmin:

```
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";
```

Iteraattoria täytyi liikuttaa askel taaksepäin, koska se osoitti aluksi joukon viimeisen alkion jälkeiseen kohtaan.

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

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

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

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

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

Iteraattori a osoittaa pienimpään alkioon, joka on ainakin yhtä suuri kuin x. Jos tämä on joukon ensimmäinen alkio, tämä on x:ää lähin alkio. Jos tällaista alkiota ei ole, x:ää lähin alkio on joukon viimeinen alkio. Muussa tapauksessa x:ää lähin alkio on joko a:n osoittama alkio tai tätä edellinen alkio.

4.2.3 Hakemisto

C++:n `map`-rakenne toteuttaa hakemiston, joka sisältää avain-alkio-pareja. Hakemisto muistuttaa taulukkoa, mutta taulukossa avaimet ovat aina peräkkäisiä kokonaislukuja, kun taas hakemistossa ne voivat olla mitä tahansa arvoja.

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

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

Hakemisto on toteutettu joukon tavoin binäärihakupuuna, minkä vuoksi alkion käsittely vie aikaa $O(\log n)$.

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

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

Komennolla `count` voi myös tutkia, esiintyykö avain hakemistossa:

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

Seuraava koodi listaa hakemiston kaikki avaimet ja alkiot:

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

4.3 Hajautustaulu

Hajautustaulu (*hash table*) on vaihtoehtoinen tekniikka joukon ja hakemiston toteuttamiseen. Hajautustauluun liittyy hajautusfunktio, joka määrää alkion sijainnin hajautustaulussa. Tavoitteena on, että hajautusfunktio sijoittaa alkioita tasaisesti hajautustaulun eri osiin.

Hajautustaulun tehokkuus riippuu siitä, miten hyvin alkioiden sijoitus onnistuu. Jos hajautusfunktio toimii hyvin, lisäyksen, etsimisen ja poistamisen aikavaativuus on vain $O(1)$. Käytännössä hajautustaulu toimii yleensä nopeammin kuin binääripuu. C++:n rakenteet `unordered_set` ja `unordered_map` toteuttavat joukon ja hakemiston hajautustaululla.

Hajautustaulun heikkoutena binääripuuhun verrattuna on, että binääripuu pitää yllä siinä olevien alkioiden järjestystä, kun taas hajautustaulussa alkiot ovat sekaisin. Tämän vuoksi hajautustaulusta ei voi kysyä esimerkiksi, mikä on seuraava x:ää suurempi alkio. Tämä näkyy myös siinä, että `unordered_set` ei sisällä järjestykseen liittyviä funktioita, kuten `lower_bound` ja `upper_bound`.

4.4 Keko

Keko (*heap*) on tietorakenne, josta on kaksi versiota: minimikeko ja maksimikeko. Minimikeon operaatiot ovat alkion lisäys, pienimmän alkion haku ja pienimmän alkion poisto. Maksimikeon operaatiot taas ovat alkion lisäys, suurimman alkion haku ja suurimman alkion poisto.

Tavallisin keon toteutus on binäärikeko, jossa lisäyksen ja poiston aikavaativuus on $O(\log n)$ ja haun aikavaativuus on $O(1)$.

Keko on yksinkertaisempi tietorakenne kuin binäärihakupuu, minkä vuoksi sen operaatiot ovat tehokkaammat. Huonona puolena on kuitenkin, että keosta pystyy hakemaan ja poistamaan vain pienimmän tai suurimman alkion, kun taas binäärihakupuusta pystyy hakemaan ja poistamaan minkä tahansa alkion.

C++:ssa tietorakenne `priority_queue` toteuttaa keon. Seuraava koodi esittelee keon 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();
```

C++:n keko on oletuksena maksimikeko. Funktio `push` lisää alkion kekoon, Funktio `top` hakee suurimman alkion ja funktio `pop` poistaa suurimman alkion. Tarvittaessa minimikeon pystyy luomaan seuraavasti:

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

Luku 5

Peruuttava haku

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

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

5.1 Osajoukot

Tehtävä: Annettuna on taulukko, jossa on n lukua. Montako erilaista summaa voit muodostaa taulukon luvuista?

Esimerkiksi jos taulukko on $[1,3,4]$, mahdolliset summat ovat:

luvut	summa
–	0
1	1
3	3
4	4
$1+3$	4
$1+4$	5
$3+4$	7
$1+3+4$	8

Summa 4 esiintyy kahdesti, joten erilaisia summia on 7.

Jokainen taulukon lukujen osajoukko tuottaa yhden mahdollisen summan, joten tehtävän voi ratkaista käymällä läpi kaikki lukujen osajoukot. Tämä onnistuu peruuttavalla haulla niin, että haku käy läpi taulukon luvut ja haarautuu joka askeleella sen mukaan, valitaanko luku mukaan summaan vai ei.

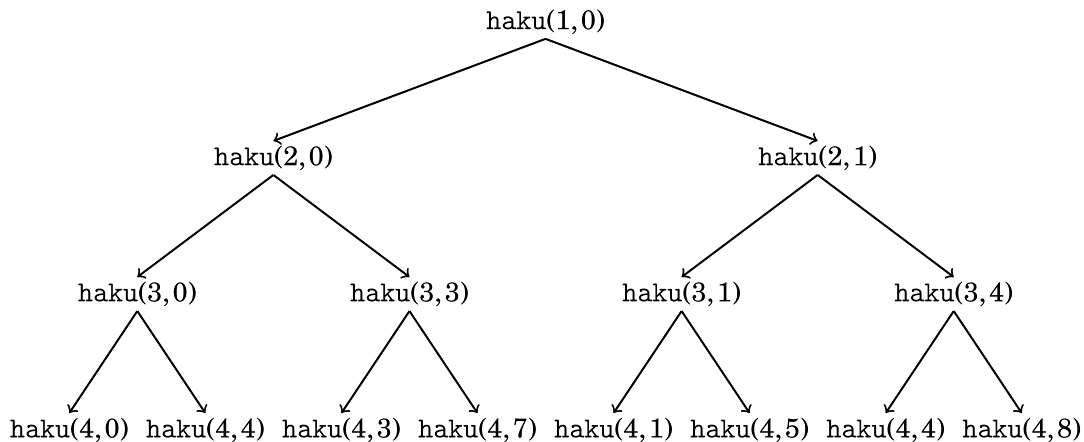
Oletetaan, että taulukon luvut ovat t[1], t[2], ..., t[n]. Seuraava rekursiivinen funktio toteuttaa peruuttavan haun:

```
set<int> z;

void haku(int k, int s) {
    if (k == n+1) {
        z.insert(s);
    } else {
        haku(k+1, s);
        haku(k+1, s+t[k]);
    }
}
```

Funktion parametri k on käsiteltävä taulukon indeksi ja parametri s on tähän mennessä valittujen lukujen summa. Funktio haarautuu kahteen osaan sen mukaan, valitaanko luku t[k] summaan vai ei. Kun $k = n+1$, kaikki luvut on käyty läpi. Silloin funktio lisää muodostetun summan s joukkoon z.

Funktiota kutsutaan haku(1,0), koska aluksi käsiteltävä luku on kohdassa 1 ja summana on 0. Esimerkiksi kun taulukko on [1, 3, 4], funktion suoritus etenee seuraavan kuvan mukaisesti:



Erilaisia osajoukkoja on 2^n, minkä vuoksi funktiota kutsutaan haun aikana $O(2^n)$ kertaa. Niinpä haku on tehokas, jos n on korkeintaan luokkaa 20.

5.2 Permutaatiot

Tehtävä: Monellako tavalla luvut $1, 2, \ldots, n$ voi järjestää niin, että vierekkäin ei ole kahta lukua, joiden erona on 1?

Esimerkiksi jos $n = 4$, ratkaisut ovat [2, 4, 1, 3] ja [3, 1, 4, 2]. Mahdollisia ratkaisuja ovat kaikki lukujen $1, 2, \ldots, n$ permutaatiot, joten tehtävän voi ratkaista käymällä läpi permutaatiot peruuttavalla haulla. Hakua rajoittaa, että vierekkäisten lukujen erona ei saa olla 1.

Seuraava funktio laskee ratkaisujen määrän:

```
void haku(int k, int e) {
    if (k == n+1) {
        c++;
        return;
    }
    for (int x = 1; x <= n; x++) {
        if (v[x] || abs(x-e) == 1) continue;
        v[x] = 1;
        haku(k+1, x);
        v[x] = 0;
    }
}
```

Haku alkaa kutsumalla funktiota `haku(1,-1)`.

Funktion parametri k on käsiteltävä kohta ja e on edelliseen kohtaan valittu luku. Poikkeuksena alussa $e = -1$, koska permutaation ensimmäinen luku voi olla mikä tahansa välillä $1, 2, \ldots, n$.

Taulukko `v` kertoo, mitkä luvut on jo valittu permutaatioon. Jos luku x on valittu permutaatioon, $\texttt{v}[x] = 1$, ja muuten $\texttt{v}[x] = 0$.

Funktio käy läpi kaikki mahdollisuudet valita permutaation seuraava luku. Jos luku x on jo valittu tai $|x - e| = 1$ eli ero viimeksi valittuun lukuun on 1, lukua ei voi valita permutaatioon. Muuttuja c laskee ratkaisujen määrän.

Permutaatioiden määrä on $n!$, joten funktiota kutsutaan $O(n!)$ kertaa ja haku on tehokas, jos n on korkeintaan luokkaa 10.

Vaihtoehtoinen tapa käydä läpi permutaatioita on käyttää C++:n standardikirjastoon kuuluvaa funktiota `next_permutation`. Se muodostaa taulukon sisällöstä seuraavaksi järjestyksessä olevan permutaation.

5.3 Ratsutehtävä

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

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

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

Tehtävä ratkeaa simuloimalla ratsun liikkeitä peruuttavan haun avulla. Ratsu aloittaa reittinsä ruudukon vasemmasta yläkulmasta ja joka siirrolla on eri-

laisia vaihtoehtoja, minne ratsu voi siirtyä seuraavaksi. Jos ruudukko on täynnä, yksi ratsun reitti on löytynyt.

Seuraava koodi laskee reittien määrän:

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

Haku alkaa kutsumalla funktiota `haku(1,1,1)`. Taulukko s on kaksiulotteinen taulukko, jonka vasen yläkulma on s[1][1] ja oikea alakulma on s[n][n].

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

Funktion tehokkuutta on hankalaa arvioida, koska haarautuminen riippuu siitä, missä kohtaa ruudukossa ratsu on ja mitkä ruudut ovat vielä vapaina. Käytännössä tapaus $n = 5$ on nopeaa laskea, mutta sitä suuremmissa tapauksissa haussa alkaa kestää kauan.

Luku 6

Ahneet algoritmit

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

Vaikeutena ahneissa algoritmeissa on keksiä toimiva ahne strategia. Usein tavoitteena on muodostaa paras mahdollinen ratkaisu tehtävään, jolloin ahneen algoritmin tulee olla sellainen, että kulloinkin parhaalta näyttävät valinnat tuottavat myös parhaan kokonaisuuden. Tämän perusteleminen voi olla vaikeaa, vaikka ahne algoritmi olisi toiminnaltaan yksinkertainen.

6.1 Vaihtoraha

Tehtävä: Annettuna on käytössä olevat kolikkojen arvot sekä rahamäärä. Mikä on pienin määrä kolikoita, joilla rahamäärän voi muodostaa?

Esimerkiksi jos käytössä on eurokolikot ja muodostettava rahamäärä on 5,20 euroa, tarvitaan vähintään neljä kolikkoa: kaksi 2 euron kolikkoa, yksi euron kolikko sekä yksi 20 sentin kolikko.

Aloitamme tehtävän käsittelyn eurokolikoista, ja tämän jälkeen siirrymme yleiseen tapaukseen, jossa kolikkokanta voi olla mikä tahansa.

6.1.1 Eurokolikot

Kun käytössä ovat eurokolikot, kolikkojen arvot ovat sentteinä

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

Luonteva ahne algoritmi tehtävään on: poista rahamäärästä aina mahdollisimman suuri kolikko, kunnes rahamäärä on 0. Algoritmi toimii esimerkissä, koska algoritmi poistaa rahamäärästä 520 ensin kahdesti 200, sitten 100 ja lopuksi 20. Mutta toimiiko ahne algoritmi aina oikein?

Osoittautuu, että eurokolikoiden tapauksessa ahne algoritmi toimii aina oikein, eli se tuottaa aina ratkaisun, jossa on pienin määrä kolikoita.

Algoritmin toimivuuden voi perustella seuraavasti:

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

Vastaavasti kutakin kolikkoa 2 ja 20 on optimiratkaisussa enintään kaksi. Jos ratkaisussa olisi kolme tällaista kolikkoa, ne voisi korvata kahdella muulla kolikolla. Jos ratkaisussa olisi kolikot $2+2+2$, ne voisi korvata kolikoilla $1+5$, ja vastaavasti kolikot $20+20+20$ voisi korvata kolikoilla $10+50$.

Edelleen kolikot $1+2+2$ voisi korvata kolikolla 5 ja kolikot $10+20+20$ voisi korvata kolikolla 50.

Näiden havaintojen perusteella jos kolikon arvo on x ja valitaan joukko x:ää pienempiä kolikoita, joiden summa on x tai enemmän, tämän joukon voi korvata pienemmällä joukolla, jossa x on mukana. Niinpä ahne algoritmi, joka valitsee aina suurimman kolikon, tuottaa optimiratkaisun.

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

6.1.2 Yleinen tapaus

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

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

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

6.2 Aikataulutus

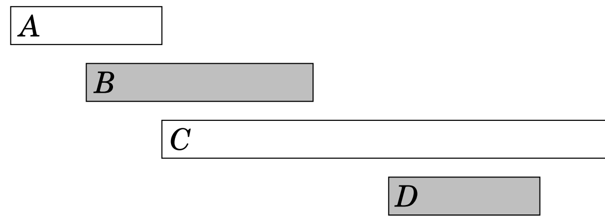
Monet aikataulutukseen liittyvät ongelmat on mahdollista ratkaista ahneilla algoritmeilla. Tutustumme seuraavaksi klassiseen tehtävään, johon on olemassa monta luontevaa ahnetta algoritmia. Kuitenkin vain yksi ahneista algoritmeista tuottaa aina optimaalisen ratkaisun.

Tehtävä: Annettuna on n tapahtumaa, joista tiedetään alku- ja loppuaika. Tehtäväsi on suunnitella aikataulu, jota seuraamalla pystyt osallistumaan mahdollisimman moneen tapahtumaan. Et voi osallistua tapahtumaan osittain.

Tarkastellaan esimerkkiä, jossa tapahtumat ovat:

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

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

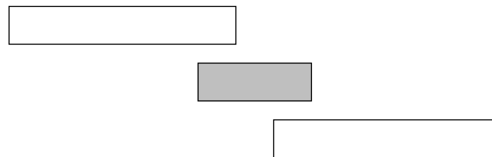


Tarkastellaan seuraavaksi kolmea ahnetta algoritmia tehtävän ratkaisuun.

Algoritmi 1

Ensimmäinen idea on järjestää tapahtumat niiden keston mukaan ja valita mukaan mahdollisimman lyhyitä tapahtumia. Tällä algoritmilla esimerkissä valittaisiin tapahtumat A ja D, jotka ovat lyhimmät tapahtumat.

Tämä algoritmi ei kuitenkaan toimi esimerkiksi seuraavassa tilanteessa:

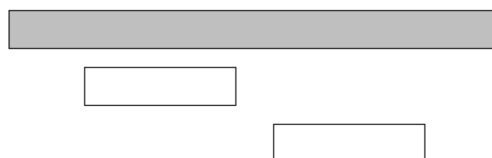


Kun lyhyt tapahtuma valitaan mukaan, on mahdollista osallistua vain yhteen tapahtumaan. Kuitenkin valitsemalla pitkät tapahtumat olisi mahdollista osallistua kahteen tapahtumaan.

Algoritmi 2

Toinen idea on valita mukaan aina tapahtuma, joka alkaa mahdollisimman aikaisin. Tällä algoritmilla esimerkissä valittaisiin tapahtumat A ja C.

Tämäkään algoritmi ei kuitenkaan toimi:



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 mukaan aina tapahtuma, joka päättyy mahdollisimman aikaisin. Tällä algoritmilla esimerkissä valittaisiin tapahtumat A ja C.

Osoittautuu, että tämä ahne algoritmi tuottaa aina optimiratkaisun. Tämän voi perustella tarkastelemalla ensimmäisen tapahtuman valintaa.

Jos ensimmäinen tapahtuma päättyy mahdollisimman aikaisin, sen jälkeen pystyy valitsemaan vielä mahdollisimman paljon muita tapahtumia. Jos sen sijasta valittaisiin toinen myöhemmin päättyvä tapahtuma, tämä ei ainakaan lisäisi mahdollisuuksia valita myöhempiä tapahtumia. Niinpä on aina hyvä ratkaisu valita ensimmäiseksi tapahtuma, joka päättyy mahdollisimman aikaisin.

6.3 Konstruktio

Joskus tehtävänä on laatia konstruktiivinen algoritmi, joka tuottaa jonkin ratkaisun tehtävään tai ilmoittaa, ettei ratkaisua ole olemassa. Konstruktiiviset algoritmit ovat usein ahneita.

Tehtävä: Sinulle on annettu lukujoukko $\{1,2,\ldots,n\}$. Tehtäväsi on etsiä jokin tapa jakaa luvut kahteen joukkoon A ja B niin, että kummankin joukon summa on sama, tai todeta, että tämä ei ole mahdollista.

Esimerkiksi jos $n=7$, yksi ratkaisu on valita joukot $A=\{1,6,7\}$ ja $B=\{2,3,4,5\}$, joissa molemmissa summa on 14.

Lukujen $1,2,\ldots,n$ summa on $s=n(n+1)/2$. Jos s on pariton, ei ole mahdollista jakaa lukuja kahteen joukkoon, joiden summa olisi sama. Jos taas s on parillinen, kummankin joukon lukujen summan tulee olla $s/2$.

Osoittautuu, että jos s on parillinen, niin lukujoukot on aina mahdollista muodostaa ja tämä onnistuu seuraavalla ahneella algoritmilla.

Ideana on käydä läpi luvut järjestyksessä $x=n,n-1,\ldots,1$ ja lisätä luku x joukkoon A, jos lisäyksen jälkeen A:n summa on enintään $s/2$, ja muuten joukkoon B. Algoritmin päätteeksi jokainen luku on jommassakummassa joukossa ja kummankin joukon summa on $s/2$.

Esimerkiksi kun $n=7$, algoritmi sijoittaa joukkoon A luvut $\{1,6,7\}$ ja joukkoon B luvut $\{2,3,4,5\}$. Algoritmi toimii, koska joka askeleella A:sta puuttuva summa on korkeintaan $1+2+\cdots+x$, missä x on käsiteltävä luku.

6.4 Keskiluvut

Tarkastellaan lopuksi kahta tehtävää, joissa tehtävänä on etsiä keskiluku, joka on mahdollisimman lähellä kaikkia taulukon alkioita. Ensin tavoite on minimoida lukujen erotusten itseisarvo ja sen jälkeen lukujen erotusten neliö.

6.4.1 Itseisarvosumma

Ensimmäinen tehtävä on etsiä luku x, joka minimoi summan

$$|\texttt{t}[0]-x|+|\texttt{t}[1]-x|+\cdots+|\texttt{t}[n-1]-x|.$$

Esimerkiksi jos taulukko on $[1,2,9,2,6]$, paras ratkaisu on $x = 2$, koska silloin summaksi tulee

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

joka on pienin mahdollinen.

Osoittautuu, että yleisessä tapauksessa paras valinta x:n arvoksi on taulukon *mediaani* eli keskimmäinen luku järjestetyssä taulukossa. Esimerkiksi taulukko $[1,2,9,2,6]$ on järjestyksessä $[1,2,2,6,9]$, joten mediaani on 2.

Mediaanin valinta on paras ratkaisu, koska jos x on mediaania pienempi tai suurempi, x:n siirtäminen lähemmäs mediaania pienentää summaa.

6.4.2 Neliösumma

Olkoon sitten tilanne muuten sama, mutta minimoitavana on summa

$$(\texttt{t}[0]-x)^2 + (\texttt{t}[1]-x)^2 + \cdots + (\texttt{t}[n-1]-x)^2.$$

Taulukon $[1,2,9,2,6]$ paras ratkaisu on nyt $x = 4$, josta tulee

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

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

Tämän voi johtaa järjestämällä summan uudestaan muotoon

$$(\texttt{t}[0]^2 + \texttt{t}[1]^2 + \cdots + \texttt{t}[n-1]^2) - 2x(\texttt{t}[0] + \texttt{t}[1] + \cdots + \texttt{t}[n-1]) + nx^2.$$

Ensimmäinen osa ei riipu x:stä, joten sen voi unohtaa. Jäljelle jäävistä osista muodostuu funktio $nx^2 - 2xs$, missä $s = \texttt{t}[0]+\texttt{t}[1]+\cdots+\texttt{t}[n-1]$. Tämä on ylöspäin aukeava paraabeli, jonka nollakohdat ovat $x = 0$ ja $x = 2s/n$ ja pienin arvo on näiden keskikohta $x = s/n$ eli taulukon lukujen keskiarvo.

Luku 7

Dynaaminen ohjelmointi

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

Tämä luku johdattaa dynaamiseen ohjelmointiin perusesimerkkien kautta. Dynaamisen ohjelmoinnin ymmärtäminen on yksi merkkipaalu jokaisen kisakoodarin uralla. Dynaaminen ohjelmointi on monipuolinen tekniikka, ja siihen palataan monta kertaa myöhemmin kirjassa erilaisissa tilanteissa.

7.1 Vaihtoraha

Tutustumme dynaamiseen ohjelmointiin tutun tehtävän kautta:

Tehtävä: Annettuna on käytössä olevat kolikkojen arvot sekä rahamäärä. Mikä on pienin määrä kolikoita, joilla rahamäärän voi muodostaa?

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

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

7.1.1 Rekursiivinen esitys

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

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

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

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

Oleellinen ominaisuus funktiossa on, että arvon $f(x)$ pystyy laskemaan rekursiivisesti käyttäen pienempiä funktion arvoja. Esimerkiksi jos kolikot ovat $\{1,3,4\}$, on kolme tapaa alkaa muodostaa rahamäärää x: valitaan kolikko 1, 3 tai 4. Jos valitaan kolikko 1, täytyy muodostaa vielä rahamäärä $x-1$. Vastaavasti jos valitaan kolikot 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 laskee minimin parametreistaan. Yleisemmin jos kolikot ovat $\{k_1, k_2, \ldots, k_n\}$, rekursiivinen kaava on

$$f(x) = \min(f(x-k_1), f(x-k_2), \ldots, f(x-k_n)) + 1.$$

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

C++:lla funktion määrittely näyttää seuraavalta:

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

Koodi olettaa, että käytössä olevat kolikot ovat $\texttt{k}[0], \texttt{k}[1], \ldots, \texttt{k}[n-1]$. Arvo 10^9 kuvastaa ääretöntä. Tämä on toimiva funktio, mutta se ei ole vielä tehokas. Seuraavaksi esiteltävä muistitaulukko tekee funktiosta tehokkaan.

7.1.2 Muistitaulukko

Dynaaminen ohjelmointi tehostaa rekursiivisen funktion laskentaa tallentamalla funktion arvoja muistitaulukkoon. Taulukon avulla funktion arvo tietyllä pa-

rametrilla tarvitsee laskea vain kerran, minkä jälkeen sen voi hakea suoraan taulukosta. Tämä muutos nopeuttaa algoritmia huomattavasti.

Tässä tehtävässä muistitaulukon voi toteuttaa näin:

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

Ideana on, että d[x] tulee sisältämään arvon $f(x)$. Jos d[x] on laskettu, funktio palauttaa sen suoraan. Muuten funktio laskee arvon rekursiivisesti ja tallentaa sen kohtaan d[x]. Taulukon d koko N on valittu niin suureksi, että kaikki x:n arvot mahtuvat taulukkoon.

Tämän muutoksen ansiosta funktio toimii nopeasti, koska sen tarvitsee laskea vastaus kullekin x:n arvolle vain kerran rekursiivisesti. Funktion aikavaativuus on vain $O(mn)$, kun rahamäärä on m ja kolikoita on n.

Huomaa kiinnostava piirre aikavaativuudessa: rahamäärän m suuruus vaikuttaa siihen, kuinka tehokas algoritmi on.

7.1.3 Silmukkatoteutus

Dynaamisen ohjelmoinnin ratkaisu on mahdollista toteuttaa rekursion sijasta myös silmukalla. Ideana on täyttää taulukko d silmukalla käänteisessä järjestyksessä rekursioon verrattuna.

Tässä tehtävässä silmukasta tulee:

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

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

7.1.4 Ratkaisun tulostus

Tyypillinen laajennus tehtävään on, että optimiratkaisun arvon lisäksi pitää tulostaa yksi mahdollinen ratkaisu. Tässä tehtävässä tämä tarkoittaa, että ohjelman täytyy antaa esimerkki tavasta valita kolikot.

Tämä onnistuu laajentamalla ratkaisua toisella taulukolla, joka kertoo kullekin x:n arvolle, mikä kolikko siitä kannattaa poistaa. Seuraavassa koodissa taulukko e huolehtii asiasta:

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

Tämän jälkeen ratkaisun voi tulostaa näin:

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

Nyt kasassa ovat kaikki dynaamisessa ohjelmoinnissa tarvittavat ainekset, ja voimme alkaa tutustua muihin tehtäviin, joissa voi hyödyntää dynaamista ohjelmointia. Usein vaikeutena on huomata, miten tehtävän voi mallintaa rekursiivisesti, jotta siinä voi käyttää dynaamista ohjelmointia.

7.2 Pisin nouseva alijono

Tehtävä: Annettuna on taulukko, jossa on n kokonaislukua. Kuinka pitkä on taulukon pisin nouseva alijono?

Taulukon t alijono on $\texttt{t}[a_1], \texttt{t}[a_2], \ldots, \texttt{t}[a_m]$, missä m on alijonon pituus ja $0 \le a_1 < a_2 < \cdots < a_m < n$. Alijono on nouseva, jos $\texttt{t}[a_1] < \texttt{t}[a_2] < \cdots < \texttt{t}[a_m]$. Esimerkiksi taulukon $[6,2,5,1,7,4,8,3]$ pisin nouseva alijono on $[2,5,7,8]$.

Ongelman rekursiivinen esitys on laskea, kuinka pitkä on pisin taulukon kohtaan k päättyvä nouseva alijono. Olkoon $f(k)$ pisimmän kohtaan k päättyvän nousevan alijonon pituus. Tätä funktiota käyttäen pisimmän nousevan alijonon pituus taulukossa on suurin arvoista $f(0), f(1), \ldots, f(n-1)$.

Esimerkkitaulukossa funktion arvot ovat:

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

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

Osoittautuu, että arvo $f(k)$ voidaan laskea arvoista $f(0), f(1), \ldots, f(k-1)$. Tapauksia on kaksi: Jos alijonossa on vain yksi luku, sen pituus on 1. Muuten alijonon pituus on $f(e)+1$, jossa $e < k$ ja $\texttt{t}[e] < \texttt{t}[k]$. Jos e on mahdollista valita monella tavalla, se tulee valita niin, että $f(e)$ on suurin mahdollinen.

Esimerkiksi alijonoon $[2,5,7,8]$ kuuluvat luvut ovat kohdissa 1, 2, 4 ja 6. Funktion arvot näissä kohdissa ovat $f(1) = 1$, $f(2) = 2$, $f(4) = 3$ ja $f(6) = 4$.

Seuraava koodi laskee pisimmän nousevan alijonon pituuden dynaamisella ohjelmoinnilla. Koodi laskee funktion arvon $f(k)$ taulukkoon $\texttt{d}[k]$ ja pisimmän alijonon pituuden muuttujaan p.

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

Koodin aikavaativuus on $O(n^2)$. Yllättävää kyllä, pisimmän nousevan alijonon voi etsiä myös ajassa $O(n \log n)$. Keksitkö, miten se tapahtuu?

7.3 Bittijonot

Tehtävä: Bittijono on *hauska*, jos siinä ei ole koskaan kahta ykkösbittiä peräkkäin. Montako hauskaa n merkin pituista bittijonoa on olemassa?

Esimerkiksi jos $n = 4$, vastaus on 8, koska bittijonot ovat 0000, 0001, 0010, 0100, 0101, 1000, 1001 sekä 1010.

Tämän tehtävän voi esittää monella tavalla rekursiivisena funktiona. Yksi esitystapa on, että $f(n,c)$ tarkoittaa, montako n-merkkistä bittiin c päättyvää hauskaa bittijonoa on olemassa. Tällöin n-merkkisten hauskojen bittijono-

jen määrä saadaan laskettua kaavalla $f(n,0)+f(n,1)$. Esimerkiksi kun $n=4$, $f(4,0)=5$ ja $f(4,1)=3$, joten hauskoja bittijonoja on $5+3=8$.

Kun $n=1$, bittijonoja on yksi:

$$\begin{aligned} f(1,0) &= 1 \\ f(1,1) &= 1 \end{aligned}$$

Kun $n>1$, rekursioksi tulee:

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

Ensimmäinen tapaus vastaa sitä, että bittijono päättyy nollabittiin. Tällöin edellinen bitti voi olla joko nollabitti tai ykkösbitti. Toisessa tapauksessa bittijono päättyy ykkösbittiin, jolloin edellisen bitin on pakko olla nollabitti.

7.4 Reitti ruudukossa

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

Seuraavassa ruudukossa paras reitti on merkitty harmaalla taustalla:

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

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

Hyvä lähestymistapa tehtävään on laskea kuhunkin ruudukon 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.

Äskeisessä ruudukossa esimerkiksi $f(1,1)=3$, $f(3,1)=13$ ja $f(5,5)=67$.

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

			↓	
		→		

Kun $\texttt{r}[y][x]$ on ruudukon luku kohdassa (y,x), rekursion perustapaukset ovat seuraavat:

$$\begin{array}{lcl} f(1,1) & = & \texttt{r}[1][1] \\ f(1,x) & = & f(1,x-1)+\texttt{r}[1][x] \\ f(y,1) & = & f(y-1,1)+\texttt{r}[y][1] \end{array}$$

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

$$f(y,x) = \max(f(y,x-1), f(y-1,x)) + \texttt{r}[y][x]$$

7.5 Editointietäisyys

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

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

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

Editointietäisyyden pystyy laskemaan tehokkaasti dynaamisen ohjelmoinnin avulla. Oletetaan että merkkijonot ovat $\texttt{x}[1\dots n]$ ja $\texttt{y}[1\dots m]$. Ideana on laskea kaikki mahdolliset arvot funktiolle $e(a,b)$, joka kertoo, mikä on merkkijonojen alkuosien $\texttt{x}[1\dots a]$ ja $\texttt{y}[1\dots b]$ editointietäisyys. Tällöin arvo $e(n,m)$ on sama kuin merkkijonojen editointietäisyys.

Seuraava taulukko sisältää funktion e arvot esimerkin tapauksessa, jossa x on PALLO ja y on TALO:

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

Esimerkiksi taulukossa harmaalla merkitty ruutu sisältää arvon $e(4,2)$: alkuosien PALL ja TA editointietäisyys.

Ensimmäinen havainto on, että jos toinen merkkijono on tyhjä, editointietäisyys on sama kuin toisen merkkijonon pituus:

$$\begin{array}{lcl} e(0,b) & = & b \\ e(a,0) & = & a \end{array}$$

Muussa tapauksessa editointietäisyyden voi laskea seuraavalla kaavalla:

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

Editointietäisyydet $e(a,b-1)+1$ ja $e(a-1,b)+1$ syntyvät siitä, että toisen merkkijonon lopusta poistetaan viimeinen merkki. Tämän kustannuksena on yksi uusi operaatio editointiin.

Editointietäisyys $e(a-1,b-1)+c(a,b)$ kuvaa sitä, että viimeiset merkit tulkitaan samoiksi. Funktio c on määritelty näin:

$$c(a,b) = \begin{cases} 0 & \mathrm{x}[a] = \mathrm{y}[b] \\ 1 & \mathrm{x}[a] \neq \mathrm{y}[b] \end{cases}$$

Funktio c kertoo kustannuksen sille, että viimeiset merkit tulkitaan samoiksi. Jos merkit ovat samat, kustannus on 0, koska editointia ei vaadita. Jos taas merkit eivät ole samat, kustannus on 1, koska tarvitaan yksi operaatio lisää.

Funktion e muodostamasta taulukosta pystyy myös lukemaan, mitä operaatioita tarvitaan pienimmän editointietäisyyden saavuttamiseksi. Esimerkkitapauksessa mahdollinen polku on seuraava:

		P	A	L	L	O
	0	1	2	3	4	5
T	1	1	2	3	4	5
A	2	2	1	2	3	4
L	3	3	2	1	2	3
O	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 merkkijonojen PAL ja TAL, jne.

Luku 8

Taulukon käsittely

Tämä luku esittelee tekniikoita taulukon käsittelyyn. Summataulukon avulla pystyy laskemaan tehokkaasti taulukon välin summan, ja kahden osoittimen tekniikka pitää yllä liikkuvaa väliä taulukossa. Lopuksi tutustumme algoritmeihin, jotka käyttävät taulukon käsittelyssä apurakenteena pinoa ja jonoa.

8.1 Summataulukko

Summataulukko (*prefix array*) kertoo jokaiselle taulukon kohdalle, mikä on taulukon alkuosan lukujen summa kyseiseen kohtaan mennessä. Summataulukon avulla voi laskea tehokkaasti minkä tahansa taulukon välin summan.

Esimerkiksi taulukon

1	3	4	8	6	1	4	2

summataulukko on:

1	4	8	16	22	23	27	29

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

Esimerkiksi välin

1	3	4	8	6	1	4	2

summa on $8+6+1+4=19$. Tämän saa summataulukosta laskemalla $27-8=19$:

1	4	8	16	22	23	27	29

Yleisemmin taulukon T summan välillä $a, a+1, \ldots, b$ eli $T[a]+T[a+1]+\cdots+T[b]$ saa laskettua summataulukosta S kaavalla $S[b]-S[a-1]$.

Summataulukko on mahdollista yleistää myös korkeampiin ulottuvuuksiin. Esimerkiksi kaksiulotteisen summataulukon jokaisessa kohdassa on sellaisen

suorakulmion summa, joka alkaa taulukon vasemmasta yläkulmasta ja päättyy kyseiseen kohtaan. Nyt minkä tahansa suorakulmion summan saa laskettua neljän vasemmasta yläkulmasta alkavan suorakulmion avulla.

Seuraava ruudukko havainnollistaa asiaa:

		D				C			
		B				A			

Harmaan suorakulmion summan saa laskettua kaavalla

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

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

8.2 Kaksi osoitinta

Kahden osoittimen (*two pointers*) tekniikassa algoritmissa on kaksi muuttujaa, jotka kertovat välin alku- ja loppukohdan taulukossa. Algoritmin aikana osoittimet liikkuvat eteenpäin vaihtelevaa tahtia. Yhteensä kumpikin osoitin liikkuu $O(n)$ askelta, joten algoritmin aikavaativuus on $O(n)$.

Tarkastellaan esimerkiksi seuraavaa tehtävää:

Tehtävä: Annettuna on taulukko, jossa on n positiivista lukua. Tehtäväsi on laskea, monessako taulukon yhtenäisessä välissä lukujen summa on x.

Esimerkiksi taulukossa

1	3	4	8	6	1	4	2

on kolme väliä, joissa summana on 7:

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

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

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

Kahden osoittimen ratkaisu pitää yllä osoittimia a ja b niin, että a osoittaa välin vasemman reunan ja b osoittaa välin oikean reunan. Joka kierroksella a liikkuu askeleen eteenpäin ja b liikkuu eteenpäin, kunnes välin summa on x tai enemmän. Jos välin $[a,b]$ summa on x, kyseinen väli on yksi ratkaisuista.

Kumpikin osoitin liikkuu yhteensä $O(n)$ askelta, joten algoritmin kokonaisaikavaativuus on $O(n)$.

Vaihtoehtoinen tapa ratkaista tehtävä on käyttää summataulukkoa ja binäärihakua. Tämän ratkaisun aikavaativuus on $O(n \log n)$, eli se on hieman hitaampi, mutta kuitenkin edelleen tehokas ratkaisu.

8.3 Apurakenteet

Joskus taulukon käsittelyssä tarvitsee apurakenteena pinoa, jonoa tai niiden yhdistelmää. Usein tällöin taulukon läpikäynnin yhteydessä pidetään yllä rakennetta, joka sisältää osan taulukon alkioista järjestyksessä.

Seuraavaksi tutustumme muutamaan algoritmiin, jotka pitävät yllä tällaista apurakennetta.

8.3.1 Seuraava suurempi

Tehtävä: Annettuna on taulukko, jossa on n lukua. Selvitä jokaiseen taulukon kohtaan, mikä on seuraava suurempi luku oikealla.

Esimerkiksi taulukossa

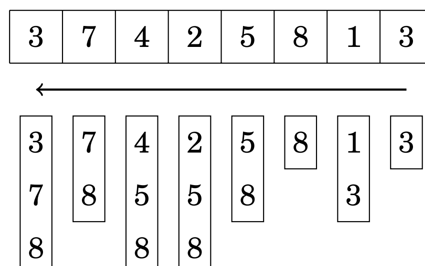
3	7	4	2	5	8	1	3

suuremmat luvut ovat:

7	8	5	5	8	–	3	–

Ideana on käydä läpi taulukko oikealta vasemmalle ja pitää yllä pinoa taulukon luvuista. Joka vaiheessa pinosta poistetaan lukuja, kunnes vastaan tulee luku, joka on käsiteltävää taulukon lukua suurempi. Tämä luku on seuraava suurempi luku taulukossa. Jos pino tyhjenee, niin mitään suurempaa lukua ei ole oikealla. Lopuksi käsiteltävä luku lisätään pinoon.

Esimerkin taulukossa pinon sisältö muuttuu näin:



Algoritmin aikavaativuus on $O(n)$, koska jokainen taulukon luku lisätään kerran pinoon ja poistetaan kerran pinosta. Niinpä jokaista lukua kohden tehdään $O(1)$ työtä. Tällaisesta tavasta laskea aikavaativuus käytetään joskus nimeä tasoitettu analyysi (*amortized analysis*).

8.3.2 Liukuvan ikkunan minimi

Tehtävä: Annettuna on taulukko, jossa on n lukua, sekä ikkunan koko k. Tehtävänä on laskea liukuvan ikkunan minimi (*sliding window minimum*) eli jokaisen k-pituisen välin pienin luku.

Esimerkiksi taulukossa

3	1	5	3	4	7	2	5

4-kokoisen liukuvan ikkunan minimi on lukujono 1, 1, 3, 2, 2.

Tehokas ratkaisu tehtävään on käydä taulukkoa läpi vasemmalta oikealle ja ylläpitää listassa nousevaa lukujonoa välissä olevista luvuista.

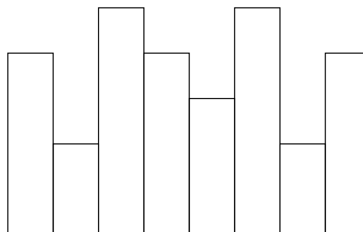
Ideana on, että listan ensimmäinen luku on aina välin pienin luku. Kun ikkuna liikkuu eteenpäin, listan lopusta poistetaan lukuja niin kauan kuin kuin ikkunan uusi luku on pienempi tai yhtä suuri kuin listan viimeinen luku. Tämän jälkeen ikkunan uusi luku lisätään listaan. Lisäksi jos listan ensimmäinen luku jää ikkunan ulkopuolelle, se poistetaan listasta.

Algoritmin aikavaativuus on $O(n)$, koska jokaista lukua kohden listaan tapahtuu yksi lisäys ja yksi poisto ja listaa käsitellään vain sen päistä.

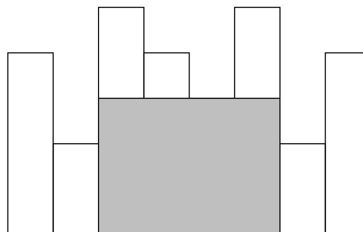
8.3.3 Suurin suorakulmio

Tehtävä: Annettuna on lauta-aita, jonka jokaisen laudan leveys on 1 ja korkeudet vaihtelevat. Mikä on pinta-alaltaan suurin suorakulmion muotoinen mainos, jonka aitaan voi kiinnittää?

Esimerkiksi aidassa



suurin mainos on



Tämä tehtävä ratkeaa $O(n)$-ajassa käymällä läpi aitojen pituudet sisältävä taulukko vasemmalta oikealle ja pitämällä yllä pinoa, joka sisältää mahdolliset aloituskohdat kyseiseen kohtaan päättyvälle mainokselle.

Luku 9

Segmenttipuu

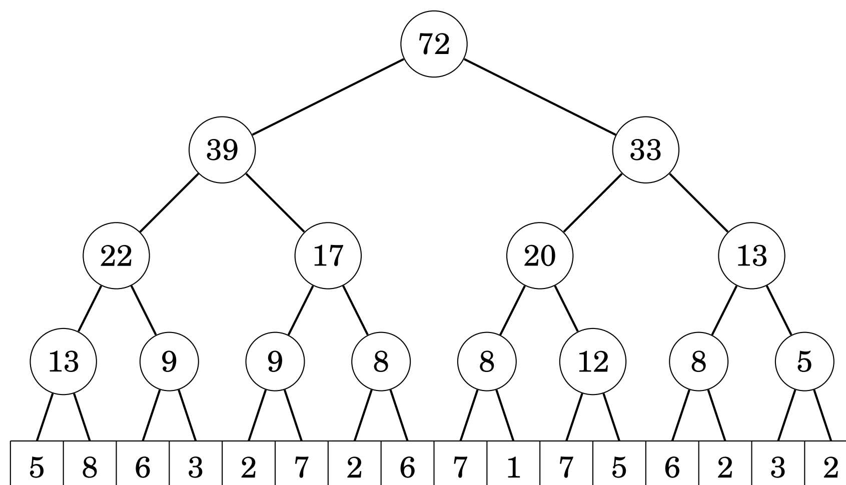
Segmenttipuu (*segment tree*) on tietorakenne, jonka avulla taulukkoon voi toteuttaa tehokkaasti monenlaisia välikyselyitä. Tavallisia välikyselyitä ovat taulukon välin summan, minimin ja maksimin laskeminen. Segmenttipuu mahdollistaa sekä välikyselyn että taulukon muuttamisen ajassa $O(\log n)$.

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

9.1 Rakenne

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

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



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

9.2 Operaatiot

Segmenttipuun operaatiot ovat välikyselyn suoritus sekä taulukon muuttaminen. Puurakenteen ansiosta kumpikin operaatio on mahdollista toteuttaa niin, että operaation aikavaativuus on $O(\log n)$. Tutustumme nyt operaatioihin käyttäen esimerkkinä edelleen summan laskemista taulukossa.

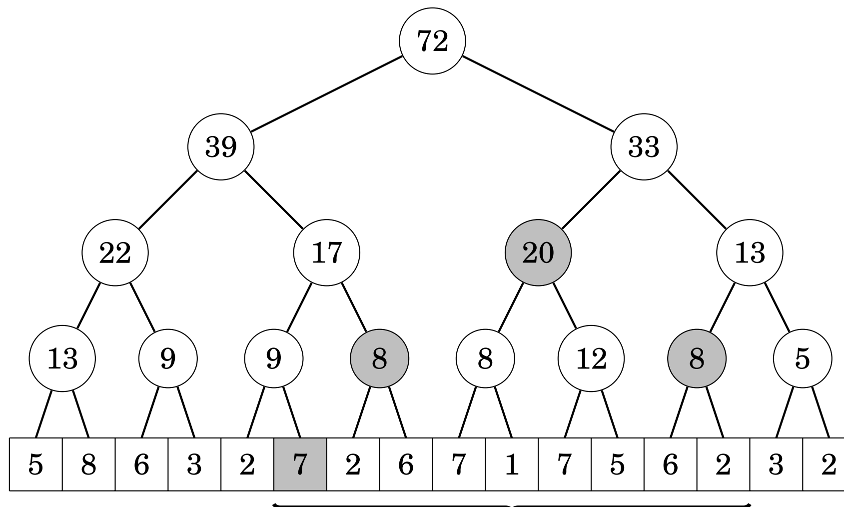
9.2.1 Välikysely

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

Esimerkiksi taulukon välin

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

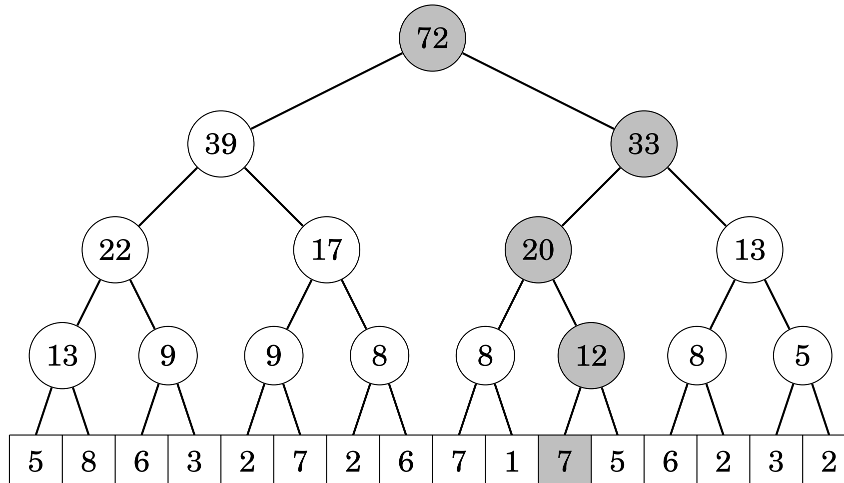
summa muodostuu seuraavista osista:



Taulukon välin lukujen summa saadaan laskemalla yhteen osien summat, eli tässä tapauksessa summaksi tulee $7+8+20+8=43$. Osoittautuu, että kun summa lasketaan mahdollisimman korkealla segmenttipuussa olevista osasummista, summan saa laskettua aina $O(\log n)$ osasta.

9.2.2 Muuttaminen

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



9.2.3 Toteutus

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

```
int p[2*N];
```

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

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

```
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;
}
```

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

distämällä puussa olevia osasummia. Osasumma lisätään summaan silloin, kun ylemmälle tasolle siirtyminen veisi välin ulkopuolelle.

Funktio `muuta` muuttaa kohdan k arvoksi x:

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

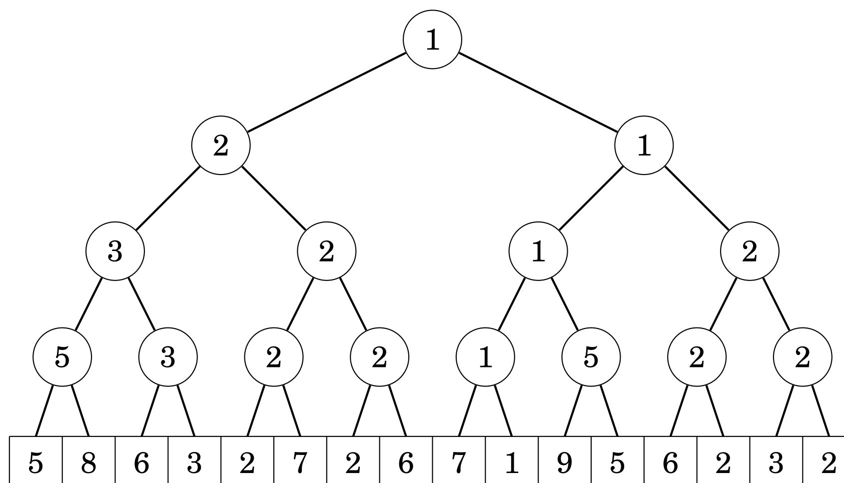
Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasummat puun huipulle asti. Taulukon `p` indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k+1$.

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

9.2.4 Muut kyselyt

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

Esimerkiksi tässä on aiemmasta taulukosta tehty minimipuu:



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

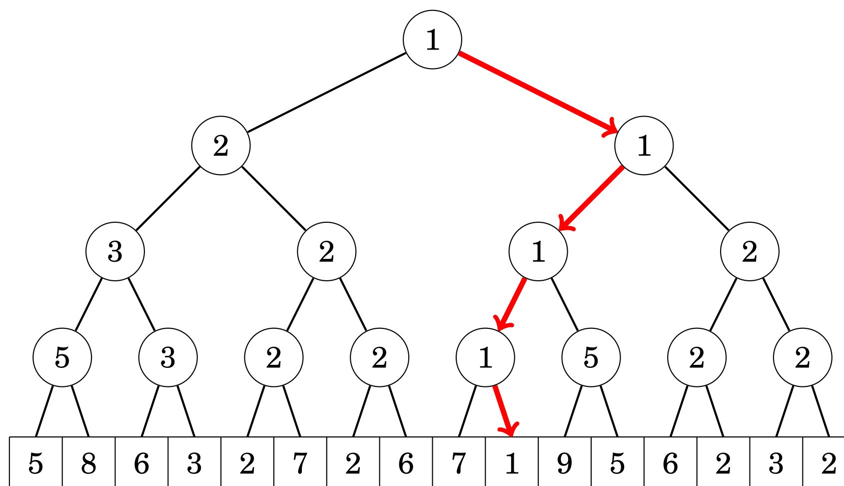
9.3 Lisätekniikoita

Segmenttipuu on joustava tietorakenne, jolle on monenlaista käyttöä kisakoodauksessa. Seuraavassa on joitakin usein esiintyviä tekniikoita, jotka laajentavat segmenttipuun käyttömahdollisuuksia.

9.3.1 Binäärihaku puussa

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

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



9.3.2 Koordinaattien pakkaus

Segmenttipuun rajoituksena on, että se on rakennettu taulukon päälle ja alkiot on indeksoitu kokonaisluvuin $0, 1, 2, \ldots, N-1$. Niinpä segmenttipuu vie muistia $O(N)$ eivätkä indeksit voi olla kovin suuria.

Tätä rajoitusta on kuitenkin mahdollista kiertää usein käyttämällä koordinaattien pakkausta. Se tarkoittaa, että indeksit jaetaan uudestaan niin, että ne ovatkin välillä $0, 1, 2, \ldots, N-1$. Tämä onnistuu silloin, jos kaikki algoritmin aikana tarvittavat indeksit ovat tiedossa koodin 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 $p(a) < p(b)$. Esimerkiksi jos alkuperäiset indeksit ovat 10^9, -555 ja 8, ne muuttuvat näin:

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

9.3.3 Välin muuttaminen

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

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

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

tulee seuraava:

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

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

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

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

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

Luku 10

Bittien käsittely

Tässä luvussa tutustumme C++:n bittioperaatioihin, jotka käsittelevät kokonaislukuja bittimuodossa. Niiden avulla syntyy tietorakenne bittitaulukko, joka mahdollistaa tehokkaan osajoukkojen käsittelyn. Lopuksi näemme, kuinka bittitaulukkoa voi käyttää dynaamisessa ohjelmoinnissa.

10.1 Luku bitteinä

Binäärijärjestelmässä luvut esitetään käyttäen kahta numeroa eli bittiä. Ideana on muodostaa luvut samalla tavalla kuin tutussa kymmenjärjestelmässä, mutta käytettävissä ovat vain numerot 0 ja 1.

Seuraavassa taulukossa on lukujen 0–31 bittiesitykset:

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

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

- $22/2 = 11$, jää $\underline{0}$
- $11/2 = 5$, jää $\underline{1}$
- $5/2 = 2$, jää $\underline{1}$
- $2/2 = 1$, jää $\underline{0}$
- $1/2 = 0$, jää $\underline{1}$

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

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

10.2 Bittioperaatiot

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

10.2.1 And-operaatio

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

	22	10110
	26	11010
&	18	10010

10.2.2 Or-operaatio

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

	22	10110
	26	11010
|	30	11110

10.2.3 Xor-operaatio

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

	22	10110
	26	11010
^	12	01100

10.2.4 Negaatio

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

Esimerkiksi ~22 tuottaa tuloksen -23, jos luvun tyyppinä on `int`. Tämä johtuu siitä, että `int`-luvut tallennetaan sisäisesti kahden komplementtina, jolloin

$\sim a = -a - 1$. Bittimuodossa tilanne näyttää seuraavalta:

22	00000000000000000000000000010110
−23	11111111111111111111111111101001

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

10.2.5 Bittisiirto

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

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

10.2.6 Sovelluksia

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

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

Seuraava koodi muuttaa luvun a bitin k ykkösbitiksi:

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

Seuraava koodi muuttaa luvun a bitin k nollabitiksi:

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

Seuraava koodi muuttaa luvun a bitin k päinvastaiseksi:

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

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

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

10.3 Bittitaulukko

Bittitaulukko (*bit array*) on tietorakenne, joka perustuu luvun bittiesitykseen. Ideana on, että kukin luvun bitti on yksi taulukon alkio. Bittitaulukon koko riippuu siitä, montako bittiä lukuun voi tallentaa. Esimerkiksi 32-bittistä `int`-tyyppiä käyttäessä bittitaulukon koko on 32 alkiota.

10.3.1 Rakenne

Bittitaulukko muodostuu yleensä niin, että luvun viimeinen bitti on taulukon ensimmäinen alkio, toiseksi viimeinen bitti on taulukon toinen alkio jne. Esimerkiksi luku 1234 on bitteinä 10011010010, joten vastaava bittitaulukko on:

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

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

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

Bittitaulukon käyttämisessä on kaksi hyötyä. Ensinnäkin alkioiden käsittely bittioperaatioiden avulla on tehokasta. Lisäksi bittitaulukko vie vähän muistia, koska jokainen alkio vie vain yhden bitin muistia.

Tarvittaessa bittitaulukko voi myös muodostua useamman luvun biteistä. Esimerkiksi jos käytössä on `int`-tyyppi, alkiot 0–31 kuuluvat ensimmäiseen lukuun, alkiot 32–63 kuuluvat toiseen lukuun jne.

10.3.2 Osajoukon esitys

Kätevä bittitaulukon sovellus on joukon osajoukon esittäminen bittitaulukkona. Tällöin joukossa on yhtä monta alkiota kuin luvussa on bittejä, ja ykkösbitti tarkoittaa, että alkio kuuluu osajoukkoon.

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

Bittioperaatioilla voi toteuttaa tehokkaasti joukko-operaatioita:

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

10.3.3 Osajoukon läpikäynti

Bittitaulukkoa käyttäen kaikki joukon osajoukot voi käydä läpi for-silmukalla. Seuraava koodi käy läpi kaikki n alkion joukon osajoukot:

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

Näin voi ratkaista esimerkiksi seuraavan tehtävän:

Tehtävä: Annettuna on joukko, jossa on n lukua. Tehtävänä on laskea, monessako osajoukossa lukujen summa on x.

Seuraava koodi olettaa, että luvut ovat t[0], t[1], ..., t[$n-1$]. Koodi käy läpi osajoukot ja laskee muuttujaan c, monessako osajoukossa lukujen summa on x.

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

10.4 Dynaaminen ohjelmointi

Bittitaulukkoa voi hyödyntää dynaamisessa ohjelmoinnissa niin, että dynaamisen ohjelmoinnin tilana on bittitaulukkona esitetty osajoukko. Yksi usein esiintyvä tekniikka on muuttaa permutaatioiden läpikäynti osajoukkojen läpikäynniksi dynaamisen ohjelmoinnin avulla. Näin on seuraavassa tehtävässä:

Tehtävä: Talossa on hissi, jossa suurin sallittu paino on x. Pohjakerroksessa on n henkilöä, jotka haluavat matkustaa ylimpään kerrokseen. Kun tiedossasi on kunkin henkilön paino, mikä on pienin mahdollinen hissimatkojen määrä?

Tarkastellaan esimerkkiä, jossa hissin painoraja on 10 ja henkilöt ovat:

henkilö	paino
A	3
B	4
C	5
D	7

Tässä tilanteessa tarvitaan kaksi hissimatkaa. Ensin hissillä menevät henkilöt A ja D (yhteispaino 10) ja sen jälkeen B ja C (yhteispaino 9).

Yksi ratkaisu tehtävään on käydä läpi kaikki henkilöiden permutaatiot ja laskea joka permutaatiosta, montako hissimatkaa tarvitaan, jos henkilöt menevät hissiin permutaation järjestyksessä. Tässä on kuitenkin ongelmana, että permutaatioiden määrä $n!$ on suuri luku.

Dynaamisen ohjelmoinnin ratkaisu laskee jokaiselle henkilöiden osajoukolle, mikä on paras tapa viedä henkilöt ylös. Paras tapa on sellainen, että hissimatkoja on mahdollisimman vähän. Jos vaihtoehtoja on useita, niistä paras on sellainen, jossa viimeisen hissin lastin paino on mahdollisimman pieni.

Esimerkiksi osajoukolle $\{A, B, D\}$ paras ratkaisu on 2 hissimatkaa, jossa viimeisen hissimatkan lastin paino on 4. Tässä ratkaisussa ensin menevät henkilöt A ja D (yhteispaino 10) ja sen jälkeen henkilö B (yhteispaino 4).

Dynaamisen ohjelmoinnin rekursio muodostuu käymällä läpi, kuka henkilöistä menee hissiin viimeisenä. Esimerkiksi osajoukossa $\{A, B, D\}$ vaihtoehdot ovat A, B ja D. Jos henkilö A menee viimeisenä, ongelmaksi tulee viedä osajoukko $\{B, D\}$ ylös optimaalisesti, ja vastaavasti muissa valinnoissa.

Kätevä ominaisuus bittitaulukoissa on, että jos A on B:n osajoukko, niin A:n bittitaulukko on pienempi kuin B:n bittitaulukko. Tämän ansiosta osajoukot voi käydä läpi bittitaulukoiden järjestyksessä.

Esimerkin tapauksessa dynaamisen ohjelmoinnin osaongelmat ovat:

bittimuoto	henkilöt	hissien määrä	viimeinen lasti
0000	–	1	0
0001	A	1	3
0010	B	1	4
0011	A,B	1	7
0100	C	1	5
0101	A,C	1	8
0110	B,C	1	9
0111	A,B,C	2	3
1000	D	1	7
1001	A,D	1	10
1010	B,D	2	4
1011	A,B,D	2	4
1100	C,D	2	5
1101	A,C,D	2	5
1110	B,C,D	2	7
1111	A,B,C,D	2	9

Osa II

Verkkoalgoritmit

Luku 11

Verkkojen perusteet

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

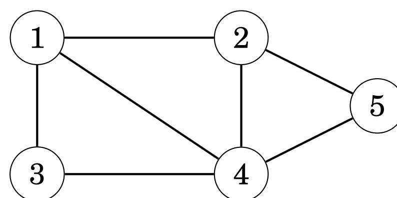
Tutustumme kirjan tässä osassa moniin verkko-ongelmiin sekä tehokkaisiin algoritmeihin niiden ratkaisemiseen. Ensin on kuitenkin paikallaan luoda katsaus siihen, mitä verkot ovat ja miten niitä voi käsitellä algoritmeissa.

11.1 Määritelmiä

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

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

Esimerkiksi seuraavassa verkossa on 5 solmua ja 7 kaarta:

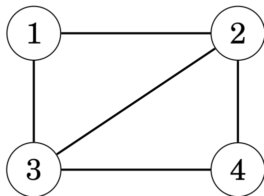


Polku (*path*) on kaaria pitkin kulkeva reitti kahden solmun välillä. Yllä olevassa verkossa solmusta 1 solmuun 5 voi kulkea esimerkiksi polkua $1 \rightarrow 4 \rightarrow 5$ tai $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5$. Solmun naapuri (*neighbor*) on toinen solmu, johon solmusta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

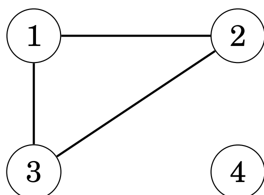
11.1.1 Yhtenäisyys

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

Tässä on esimerkki yhtenäisestä verkosta:

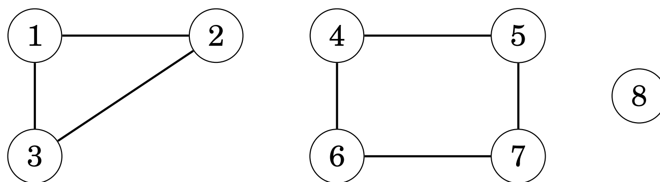


Seuraava verkko taas ei ole yhtenäinen, koska solmu 4 on erillään muista.



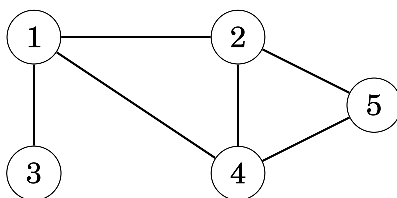
11.1.2 Komponentit

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

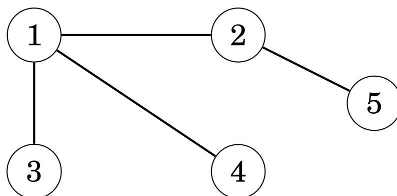


11.1.3 Syklit

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

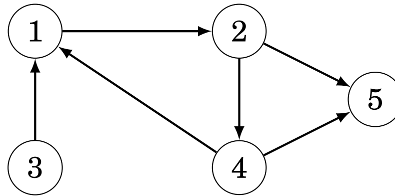


Seuraava verkko taas on syklitön (*acyclic*), eli siinä ei ole sykliä:



11.1.4 Kaarten suunnat

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

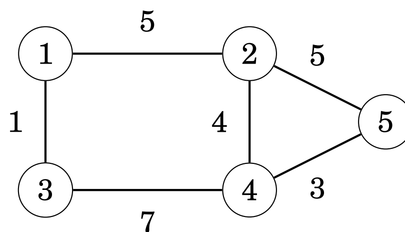


Tässä verkossa on esimerkiksi polku $3 \to 1 \to 2 \to 5$ sekä sykli $2 \to 4 \to 1 \to 2$. Solmuun 3 ei pääse mistään muusta solmusta, ja vastaavasti solmusta 5 ei pääse mihinkään muuhun solmuun.

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

11.1.5 Kaarten painot

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



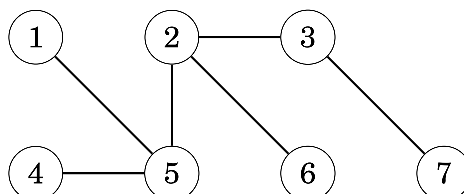
Polun pituus (*path length*) on sen kaarten painojen summa. Esimerkiksi polun $1 \to 2 \to 4$ pituus on $5+4=9$ ja polun $1 \to 3 \to 4$ pituus on $1+7=8$. Jälkimmäinen polku on lyhin polku (*shortest path*) solmusta 1 solmuun 4.

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

11.1.6 Puu

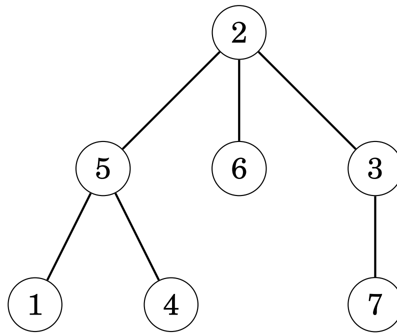
Puu (*tree*) on yhtenäinen, syklitön ja suuntaamaton verkko. Jos verkko on puu, niin sen jokaisen kahden solmun välillä on yksikäsitteinen polku.

Esimerkiksi seuraava verkko on puu:



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

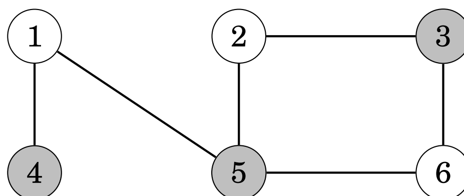
Puu esitetään usein niin, että yksi solmuista nostetaan puun juureksi (*root*) ja muut sijoittuvat tasoittain sen alapuolelle. Esimerkiksi jos äskeisessä verkossa solmusta 2 tehdään juuri, tulos on tämä:



11.1.7 Kaksijakoisuus

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

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

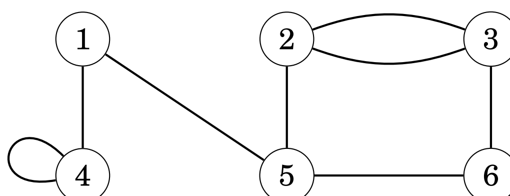


Vaihtoehtoinen määritelmä on, että verkko on kaksijakoinen tarkalleen silloin, kun siinä ei ole parittoman pituista sykliä.

11.1.8 Yksinkertaisuus

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

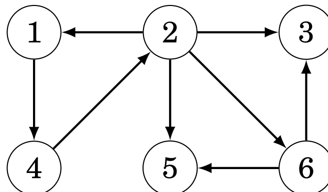
Esimerkiksi seuraava verkko ei ole yksinkertainen, koska solmusta 4 on kaari itseensä ja solmujen 2 ja 3 välillä on kaksi kaarta:



11.2 Verkon esitys

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

Esimerkkinä on seuraava verkko:



11.2.1 Vieruslistat

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

Kätevä tapa tallentaa vieruslistat on luoda taulukko, jossa jokaiselle solmulle on oma vektori:

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

Tämän jälkeen verkon kaaret lisätään näin:

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

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

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

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

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

11.2.2 Vierusmatriisi

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

Vierusmatriisi on yleensä järkevää tallentaa kaksiulotteisena taulukkona:

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

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

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

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

11.2.3 Kaarilista

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

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

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

Kaaret lisätään listalle näin:

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

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

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

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

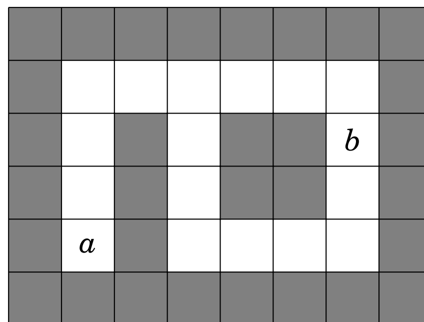
Tämän jälkeen kaaret lisätään näin:

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

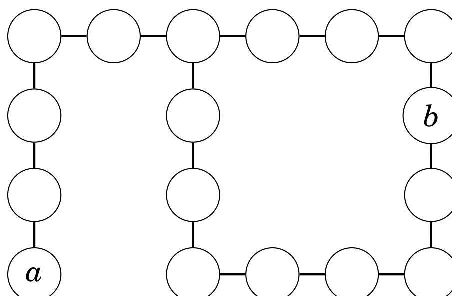
11.3 Epäsuora esitys

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

Tyypillinen esimerkki epäsuorasta verkosta on labyrintti:



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



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

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

Luku 12

Verkon läpikäynti

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

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

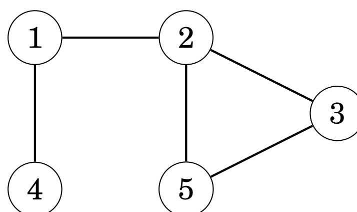
12.1 Syvyyshaku

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

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

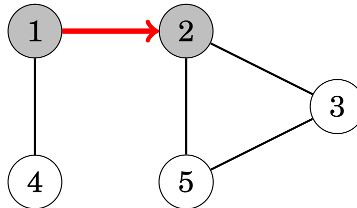
12.1.1 Toiminta

Tarkastellaan syvyyshaun toimintaa seuraavassa verkossa:

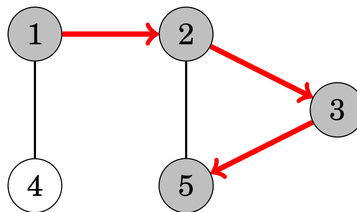


Syvyyshaku voi lähteä liikkeelle mistä tahansa solmusta, mutta oletetaan nyt, että haku lähtee liikkeelle solmusta 1.

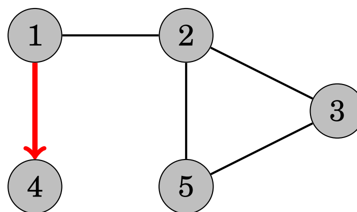
Solmun 1 naapurit ovat solmut 2 ja 4, joista haku etenee ensin solmuun 2:



Tämän jälkeen haku etenee vastaavasti solmuihin 3 ja 5:



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



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

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

12.1.2 Toteutus

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

```
int z[n+1];

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

Haku alkaa kutsumalla funktiota `haku` parametrina aloitussolmu.

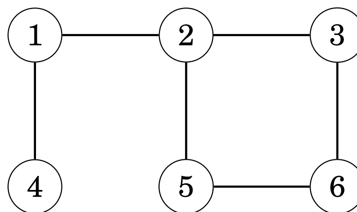
12.2 Leveyshaku

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

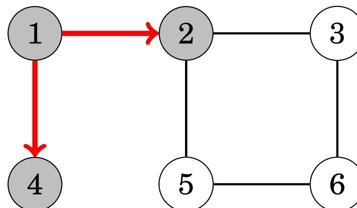
Leveyshakua voi ajatella niin, että se käy solmuja läpi kerros kerrallaan. Ensin haku käy läpi solmut, joihin pääsee yhdellä kaarella alkusolmusta. Tämän jälkeen vuorossa ovat solmut, joihin pääsee kahdella kaarella alkusolmusta jne. Sama jatkuu, kunnes uusia käsiteltäviä solmuja ei enää ole.

12.2.1 Toiminta

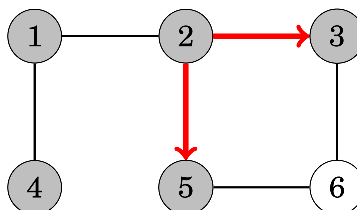
Tarkastellaan leveyshaun toimintaa seuraavassa verkossa:



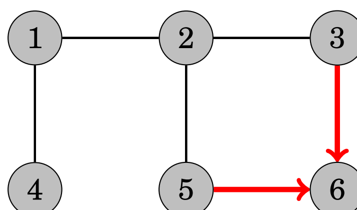
Oletetaan jälleen, että haku alkaa solmusta 1. Haku etenee ensin kaikkiin solmuihin, joihin pääsee alkusolmusta:



Seuraavaksi haku etenee solmuihin 3 ja 5:



Viimeisenä haku etenee solmuun 6:



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

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

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

12.2.2 Toteutus

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

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

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

12.3 Sovelluksia

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

12.3.1 Yhtenäisyys

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

12.3.2 Komponentit

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

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

12.3.3 Kaksijakoisuus

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

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

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

Luku 13

Lyhimmät polut

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

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

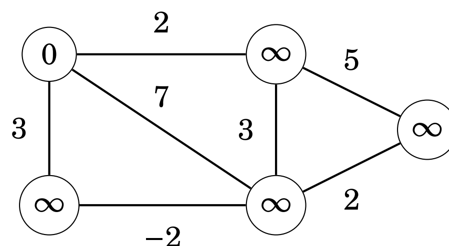
13.1 Bellman-Fordin algoritmi

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

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

13.1.1 Toiminta

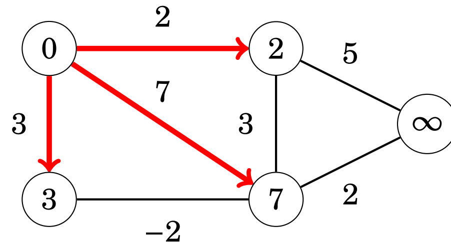
Tarkastellaan Bellman-Fordin algoritmin toimintaa seuraavassa verkossa:



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

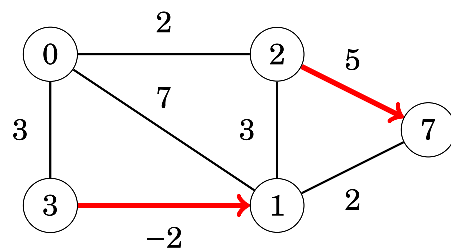
Algoritmin toiminta muodostuu peräkkäisistä kierroksista. Jokaisella kierroksella algoritmi käy kaikki verkon kaaret läpi. Jos kaaren avulla pystyy parantamaan jotain etäisyysarviota, algoritmi tekee näin.

Tässä verkossa ensimmäinen kierros parantaa arvioita näin:

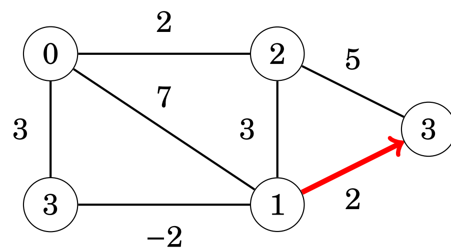


Esimerkiksi alkusolmusta lähtevä 2:n pituinen kaari parantaa sen kohdesolmun etäisyysarviota, koska vanha arvio oli ääretön, mutta kulkemalla tätä kaarta alkusolmusta etäisyys on vain 2.

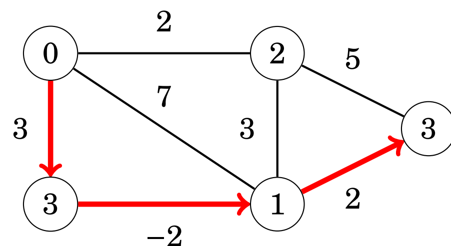
Seuraavalla kierroksella arviot paranevat näin:



Sitten tulee vielä yksi parannus:



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



13.1.2 Toteutus

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

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

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

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

13.1.3 Negatiivinen sykli

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

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

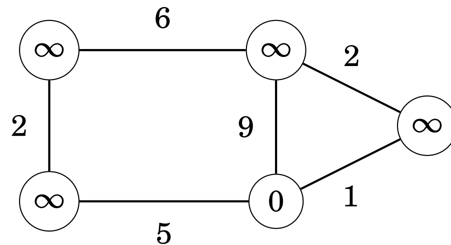
13.2 Dijkstran algoritmi

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

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

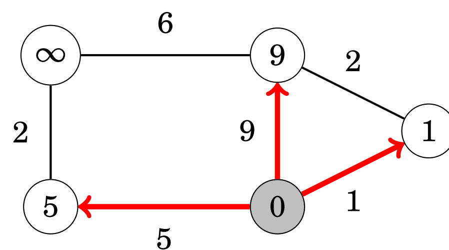
13.2.1 Toiminta

Tarkastellaan Dijkstran algoritmin toimintaa seuraavassa verkossa:



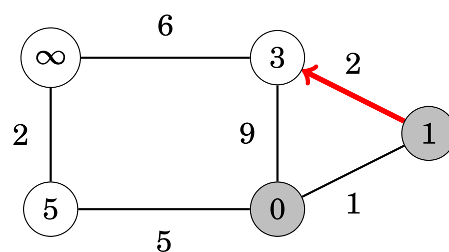
Dijkstran algoritmi ottaa joka askeleella käsittelyyn sellaisen solmun, jota ei ole vielä käsitelty ja jonka etäisyysarvio on mahdollisimman pieni. Alussa tällainen solmu on alkusolmu, jonka etäisyysarvio on 0.

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

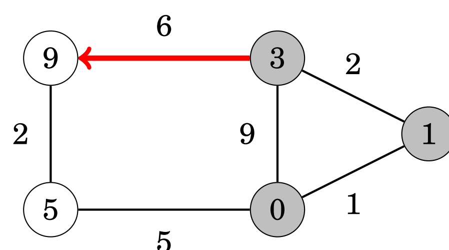


Alkusolmun käsittely paransi etäisyyksiä kolmen muuhun solmuun, joiden uudet etäisyydet ovat nyt 1, 5 ja 9.

Seuraavaksi käsittelyyn tulee solmu, jonka etäisyys on 1:

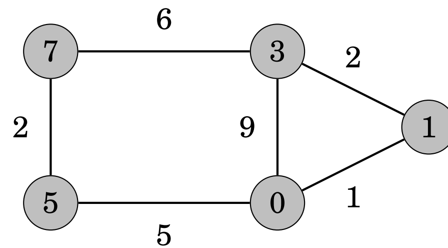


Tämän jälkeen vuorossa on solmu 3:



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

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



13.2.2 Toteutus

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

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

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

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

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

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

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

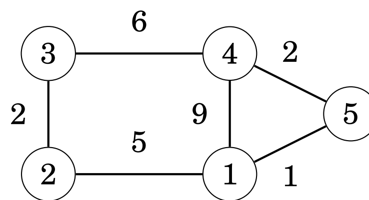
13.3 Floyd-Warshallin algoritmi

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

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

13.3.1 Toiminta

Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



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

Tässä verkossa taulukosta tulee:

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

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

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

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

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

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

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

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

13.3.2 Toteutus

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

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) d[i][j] = 0;
        else if (v[i][j]) d[i][j] = v[i][j];
        else d[i][j] = 1e9;
    }
}
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
        }
    }
}
```

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

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

13.4 Yhteenveto

Olemme nyt käsitelleet neljä algoritmia lyhimpien polkujen etsimiseen:

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

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

Luku 14

Puiden käsittely

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

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

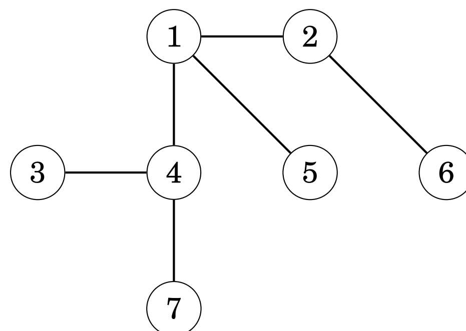
14.1 Perusteet

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

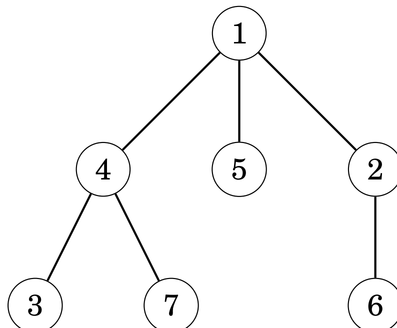
14.1.1 Juuren valinta

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

Tarkastellaan esimerkkinä seuraavaa puuta:



Kun solmu 1 valitaan juureksi, puu asettuu seuraavaan muotoon:



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

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

14.1.2 Läpikäynti

Seuraava rekursiivinen koodi käy läpi vieruslistoina tallennetun puun:

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

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

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

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

14.1.3 Dynaaminen ohjelmointi

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

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

Tarvittava koodi on tässä:

```
int d[n+1];

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

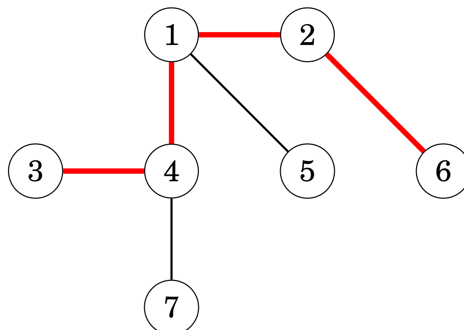
Funktion kutsun jälkeen d[s] kertoo solmun s alipuun solmujen määrän.

14.2 Etäisyydet

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

14.2.1 Läpimitta

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



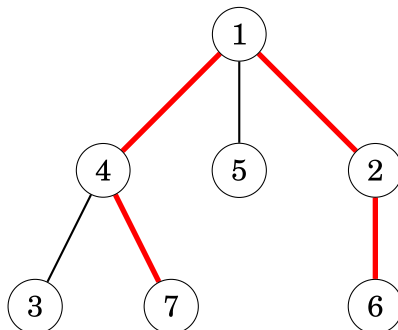
Läpimitta on 4, koska polku solmusta 3 solmuun 6 on pituudeltaan 4. Läpimittaa vastaava polku ei ole välttämättä yksikäsitteinen. Esimerkiksi tässä puussa myös polku solmusta 7 solmuun 6 on pituudeltaan 4.

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

Algoritmi 1

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

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



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

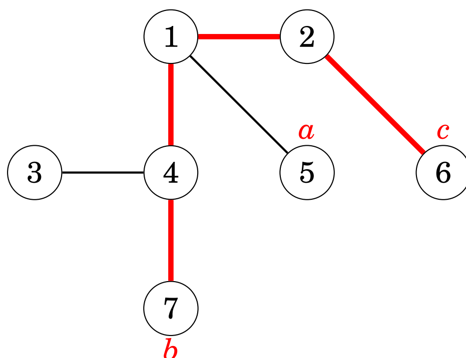
Esimerkiksi solmusta 1 lähtee kolme polkua alaspäin. Niistä kahden pituus on 2 ($1 \rightarrow 4 \rightarrow 7$ sekä $1 \rightarrow 2 \rightarrow 6$) ja yhden pituus on 1 ($1 \rightarrow 5$). Pisin polku syntyy yhdistämällä molemmat 2-pituiset polut.

Algoritmi 2

Toinen tehokas tapa laskea puun läpimitta on seuraava:

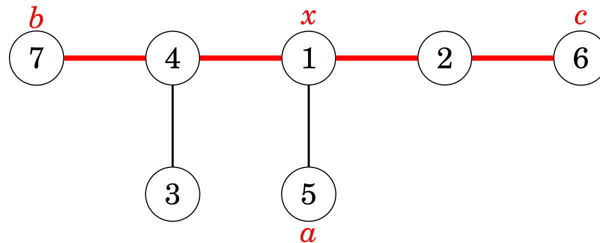
1. Valitse mikä tahansa solmu a.
2. Etsi a:sta kaukaisin solmu b (syvyyshaku).
3. Etsi b:stä kaukaisin solmu c (syvyyshaku).
4. Läpimitta on etäisyys b:n ja c:n välillä.

Esimerkissä a, b ja c voisivat olla:



Menetelmä on tyylikäs, mutta miksi se toimii?

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

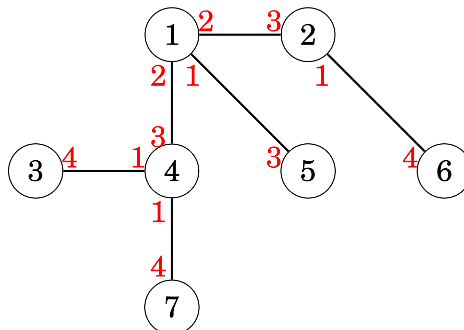


Solmu x on kohta, jossa polku solmusta a liittyy läpimittaa vastaavaan polkuun. Kaukaisin solmu a:sta on solmu b, solmu c tai jokin muu solmu, joka on ainakin yhtä kaukana solmusta x. Niinpä tämä solmu on aina sopiva valinta läpimittaa vastaavan polun toiseksi päätesolmuksi.

14.2.2 Kaikki etäisyydet

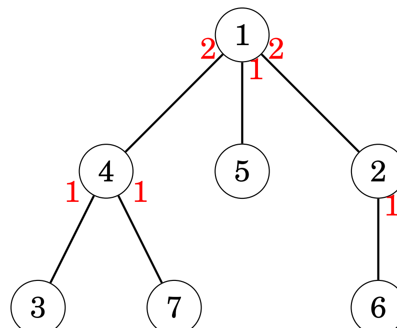
Tarkastellaan sitten vaikeampaa tehtävää, jossa tehtävänä on laskea jokaiselle puun solmulle, kuinka kaukana on kaukaisin solmu kussakin suunnassa.

Esimerkkipuussa etäisyydet ovat:



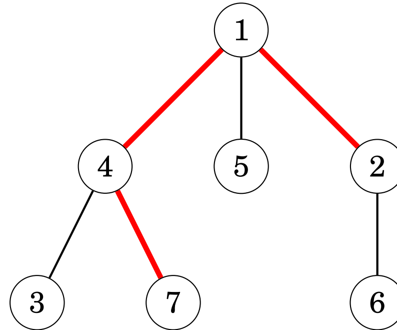
Esimerkiksi solmussa 4 kaukaisin solmu ylöspäin mentäessä on solmu 6, johon etäisyys on 3 polkua $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$.

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

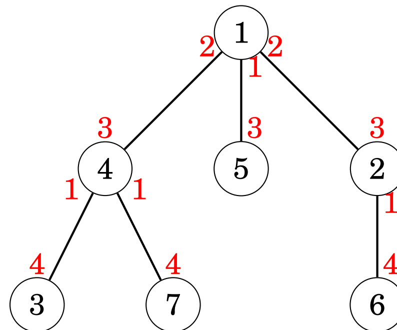


Jäljelle jäävä tehtävä on laskea etäisyydet ylöspäin. Tämä onnistuu tekemällä puuhun toinen läpikäynti, joka pitää mukana tietoa, mikä on suurin etäisyys solmun vanhemmasta johonkin toisessa suunnassa olevaan solmuun.

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



Lopputuloksena on etäisyydet kaikista solmuista kaikkiin suuntiin:



Luku 15

Virittävät puut

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

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

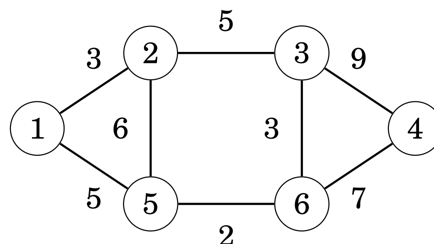
15.1 Kruskalin algoritmi

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

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

15.1.1 Toiminta

Tarkastellaan Kruskalin algoritmin toimintaa seuraavassa verkossa:

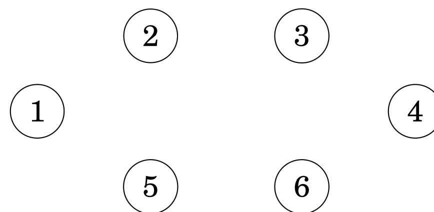


Algoritmin ensimmäinen vaihe on järjestää verkon kaaret niiden painon mukaan. Tuloksena on seuraava lista:

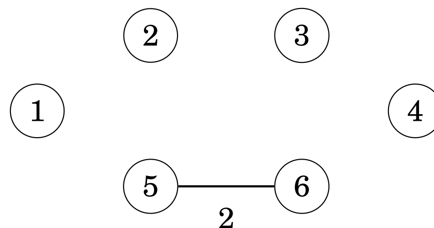
kaari	paino
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

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

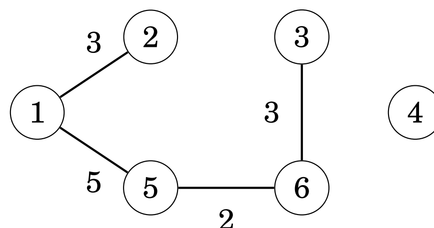
Aluksi jokainen solmu on omassa komponentissaan:



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



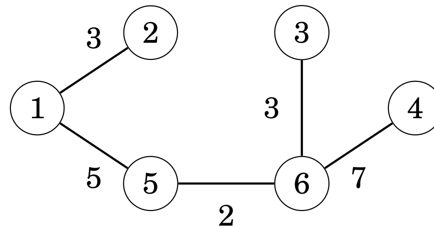
Tämän jälkeen algoritmi lisää puuhun vastaavasti kaaret 1–2, 3–6 ja 1–5:



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

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

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



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

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

15.1.2 Toteutus

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

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

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

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

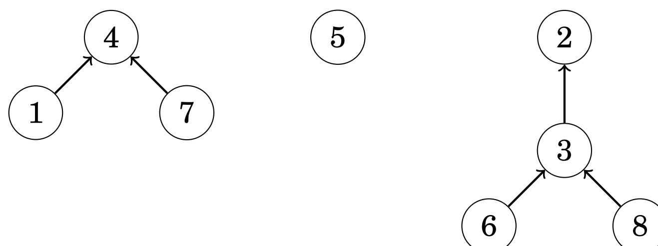
15.2 Union-find-rakenne

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

15.2.1 Rakenne

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

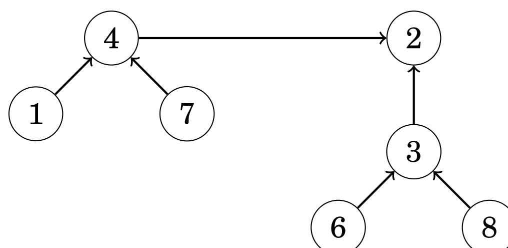
Esimerkiksi jos joukot ovat $\{1,4,7\}$, $\{5\}$ ja $\{2,3,6,8\}$, tilanne voisi olla:



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

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

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



Tästä lähtien alkio 2 edustaa kaikkia joukon alkioita.

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

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

15.2.2 Toteutus

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

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

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

Funktio `sama` kertoo, ovatko alkiot samassa joukossa:

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

Funktio `liita` taas yhdistää kaksi joukkoa toisiinsa:

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

Luku 16

Syklittömät verkot

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

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

16.1 Syklin etsiminen

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

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

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

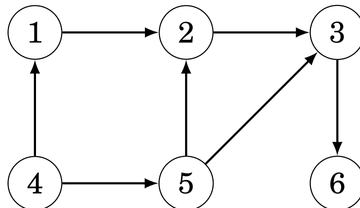
Tässä on algoritmin toteutus:

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

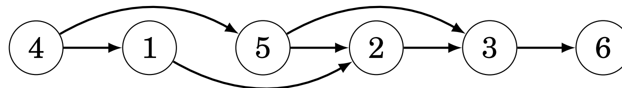
16.2 Topologinen järjestäminen

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

Tarkastellaan esimerkiksi seuraavaa verkkoa:



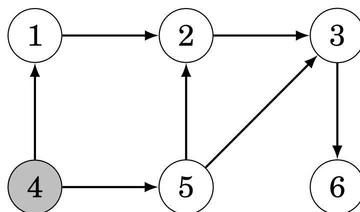
Yksi tämän verkon topologinen järjestys on 4, 1, 5, 2, 3, 6:



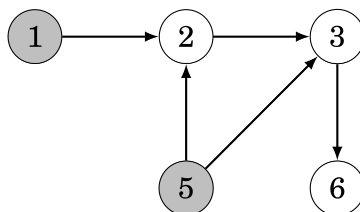
Topologinen järjestys on olemassa silloin, kun verkossa ei ole suunnattua sykliä. Jos verkossa on suunnattu sykli, topologista järjestystä ei voi muodostaa, koska mitään syklin solmuista ei voi valita järjestykseen ensimmäisenä.

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

Esimerkkiverkossa ensimmäinen valittava solmu on 4:



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



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

16.3 Dynaaminen ohjelmointi

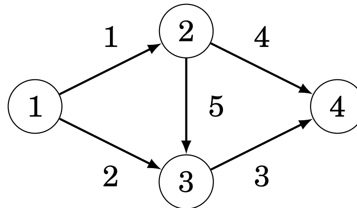
Solmujen käsittely topologisessa järjestyksessä mahdollistaa dynaamisen ohjelmoinnin käyttämisen verkossa. Seuraavaksi käymme läpi kaksi esimerkkiä dynaamisesta ohjelmoinnista polkujen etsimisessä.

16.3.1 Pisin polku

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

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

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

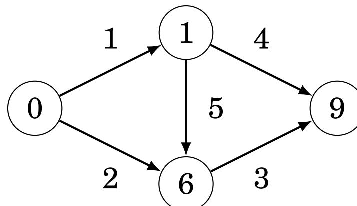


Dynaamisessa ohjelmoinnissa on ideana laskea jokaiseen solmuun, kuinka pitkä on pisin polku alkusolmusta siihen solmuun.

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

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

Tuloksena ovat seuraavat pituudet:



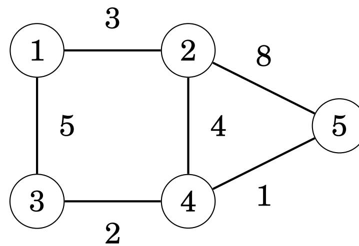
16.3.2 Lyhimmät polut

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

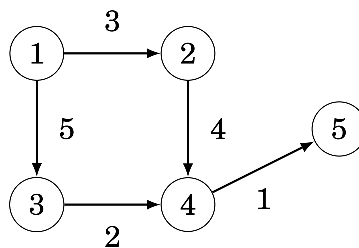
Dijkstran algoritmin tuottama kuvaus verkon lyhimmistä poluista on suunnattu, syklitön verkko, joten siihen voi käyttää dynaamista ohjelmointia.

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

Esimerkiksi verkosta

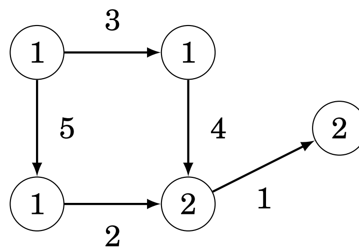


syntyy seuraava kuvaus:



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

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



Luku 17

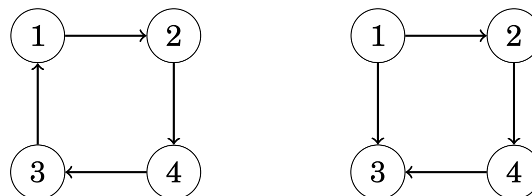
Vahvasti yhtenäisyys

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

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

17.1 Käsitteitä

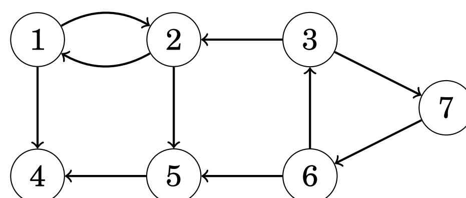
Verkko on vahvasti yhtenäinen (*strongly connected*), jos mistä tahansa solmusta on olemassa polku kaikkiin muihin solmuihin. Esimerkiksi seuraavassa kuvassa vasen verkko on vahvasti yhtenäinen, kun taas oikea verkko ei ole.



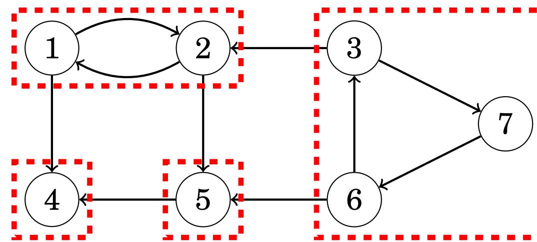
Oikeanpuoleinen verkko ei ole vahvasti yhtenäinen, koska esimerkiksi solmusta 2 ei ole polkua solmuun 1.

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

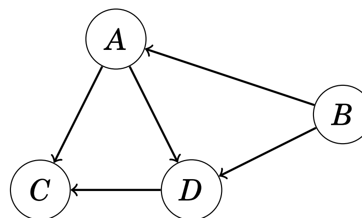
Esimerkiksi verkon



vahvasti yhtenäiset komponentit ovat



ja ne muodostavat seuraavan komponenttiverkon:



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

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

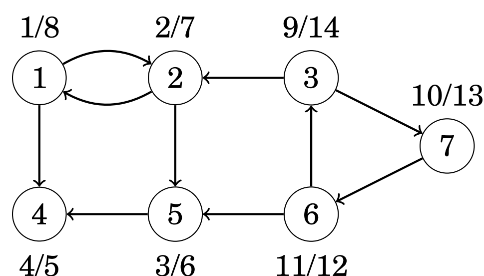
17.2 Kosarajun algoritmi

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

Syvyyshaku 1

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

Seuraava kuva näyttää solmujen käsittelyjärjestyksen esimerkkiverkossa:



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

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

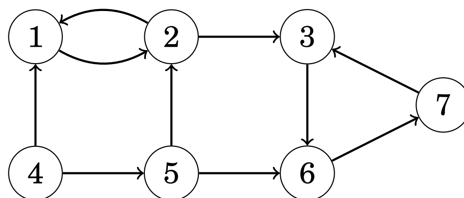
Solmujen käsittelyjärjestys algoritmin seuraavassa vaiheessa tulee olemaan tämä järjestys käänteisenä eli $[3,7,6,1,2,5,4]$.

Syvyyshaku 2

Toinen syvyyshaku muodostaa verkon vahvasti yhtenäiset komponentit.

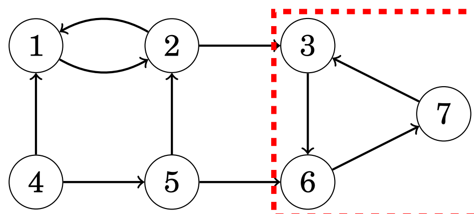
Ennen toista syvyyshakua algoritmi muuttaa jokaisen kaaren suunnan käänteiseksi. Tämä varmistaa, että toisen syvyyshaun aikana löydetään joka kerta vahvasti yhtenäinen komponentti, johon ei kuulu ylimääräisiä solmuja.

Esimerkkiverkko on käännettynä seuraava:



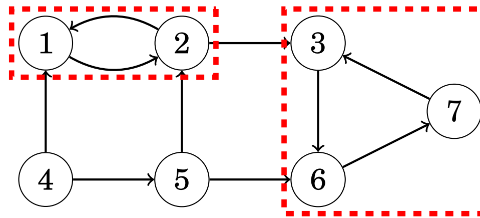
Tämän jälkeen algoritmi käy läpi solmut ensimmäisen syvyyshaun tuottamassa käsittelyjärjestyksessä. Jos solmu ei kuulu vielä komponenttiin, siitä alkaa uusi syvyyshaku käänteisessä verkossa. Solmun komponenttiin kuuluvat kaikki aiemmin käsittelemättömät solmut, joihin syvyyshaku pääsee solmusta.

Esimerkkiverkossa muodostuu ensin komponentti solmusta 3 alkaen:

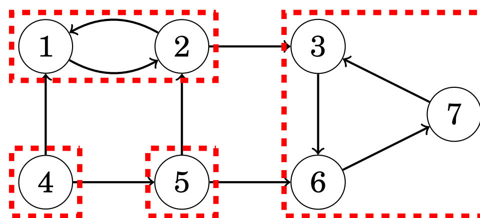


Huomaa, että kaarten kääntämisen ansiosta komponentti ei pääse ”vuotamaan” muihin verkon osiin.

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



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



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

17.3 2SAT-ongelma

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

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

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

Esimerkiksi lauseke

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

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

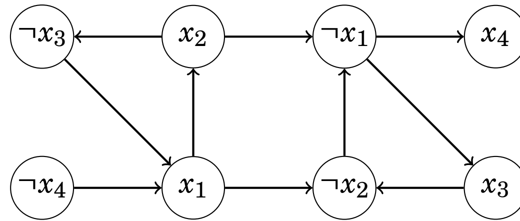
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

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

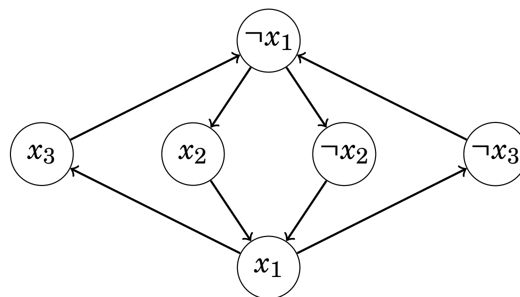
2SAT-ongelman saa muutettua verkoksi niin, että jokainen muuttuja x_i ja negaatio $\neg x_i$ on yksi verkon solmuista ja muuttujien riippuvuudet ovat kaaria.

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

Lausekkeen L_1 verkosta tulee nyt:



Lausekkeen L_2 verkosta taas tulee:



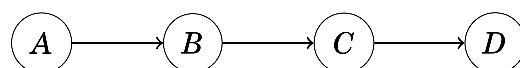
Verkon rakenne kertoo, onko 2SAT-ongelmalla ratkaisua. Jos on jokin muuttuja x_i niin, että x_i ja $\neg x_i$ ovat samassa vahvasti yhtenäisessä komponentissa, niin ratkaisua ei ole olemassa. Tällöin verkossa on polku sekä x_i:stä $\neg x_i$:ään että $\neg x_i$:stä x_i:ään, eli kumman tahansa arvon valitseminen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

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

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

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

Lausekkeen L_1 verkon komponenttiverkko on seuraava:



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

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

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

Luku 18

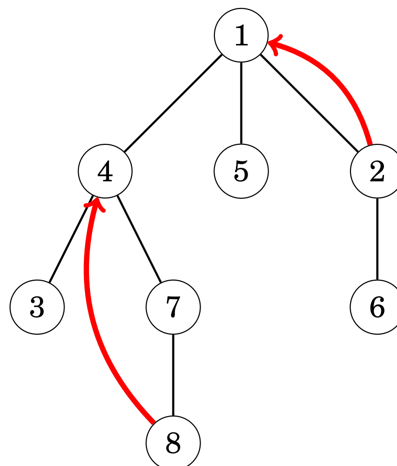
Puukyselyt

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

18.1 Tehokas nouseminen

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

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



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

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

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

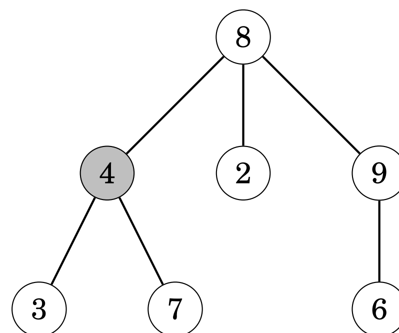
nousun pystyy jakamaan kahdeksi $k/2$ askeleen nousuksi. Esimerkiksi äskeisessä puussa $f(8,1) = 7$ ja $f(7,1) = 4$, joten $f(8,2) = f(f(8,1),1) = 4$.

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

18.2 Alipuut taulukossa

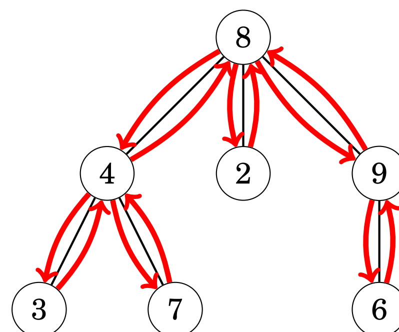
Tehtävä: Annettuna on puu, jolle on valittu juurisolmu. Jokaisella solmulla on tietty paino. Tehtävänä on käsitellä kyselyt muotoa ”muuta solmun s painoa” sekä ”mikä on pienin solmun paino solmun s alipuussa”.

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



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

Esimerkin puussa syvyyshaku etenee näin:



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

a	7	3	1	1	1	2	1

p	8	4	3	7	2	9	6

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

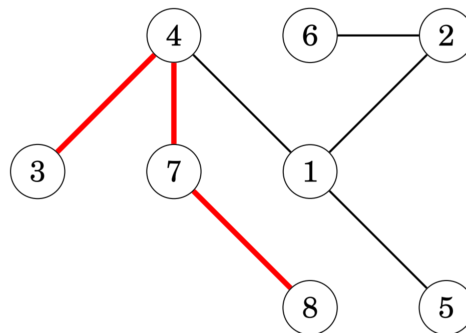
a	7	3	1	1	1	2	1
p	8	4	3	7	2	9	6

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

18.3 Alin yhteinen vanhempi

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

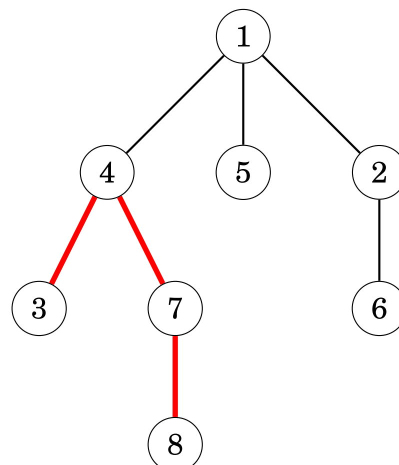
Esimerkiksi seuraavassa puussa polun pituus solmusta 3 solmuun 8 on 3, koska solmu muodostuu kaarista $3 \to 4 \to 7 \to 8$.



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

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

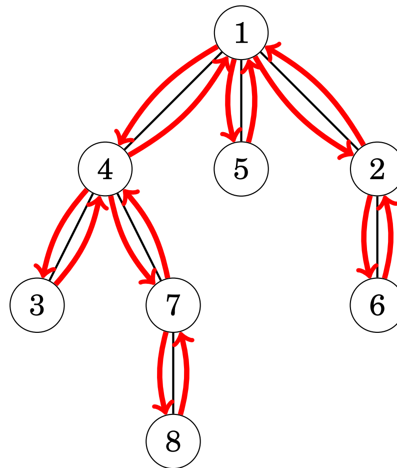
Seuraava kuva havainnollistaa asiaa:



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

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

Esimerkin puussa syvyyshaku etenee näin:



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

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

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

Esimerkiksi solmujen 3 ja 8 alin yhteinen vanhempi löytyy näin:

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

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

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

Luku 19

Polut ja kierrokset

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

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

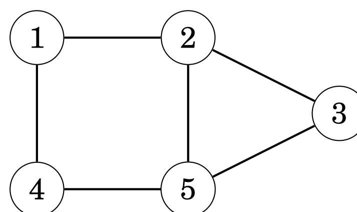
19.1 Hamiltonin polku

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

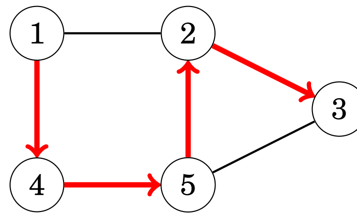
Hamiltonin polkuun liittyvät ongelmat ovat laskennallisesti vaikeita, eikä tällä hetkellä tunneta mitään tehokasta algoritmia, jolla voisi etsiä Hamiltonin polkua tai kierrosta verkosta. Kyseessä on NP-vaikea ongelma, eli luultavasti tehokasta algoritmia ei myöskään ole olemassa.

19.1.1 Esimerkkejä

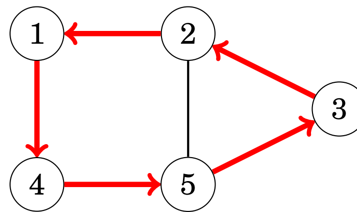
Seuraavassa verkossa on sekä Hamiltonin polku että Hamiltonin kierros:



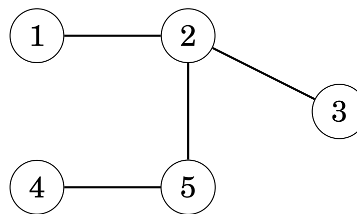
Tässä on esimerkki Hamiltonin polusta solmusta 1 solmuun 3:



Tässä taas on yksi mahdollinen Hamiltonin kierros:



Seuraavassa verkossa ei ole Hamiltonin polkua:



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

19.1.2 Peruuttava haku

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

19.1.3 Dynaaminen ohjelmointi

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

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

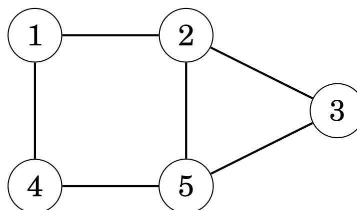
19.2 Eulerin polku

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

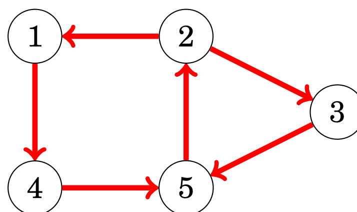
Eulerin polkuun liittyvät ongelmat ovat tehokkaasti ratkaistavissa. Eulerin polun olemassaolon voi päätellä helposti verkon rakenteesta, ja polun muodostamiseen on myös olemassa tehokkaita algoritmeja.

19.2.1 Esimerkkejä

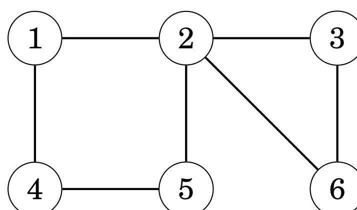
Seuraavassa verkossa on Eulerin polku:



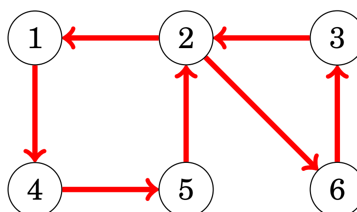
Polun voi muodostaa esimerkiksi näin solmusta 2 solmuun 5:



Seuraavassa verkossa taas on Eulerin kierros:



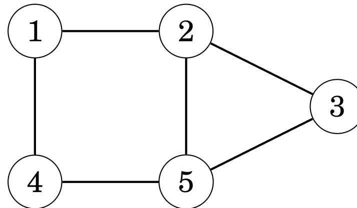
Eulerin kierros näyttää esimerkiksi seuraavalta:



19.2.2 Olemassaolo

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

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



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

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

19.2.3 Muodostus

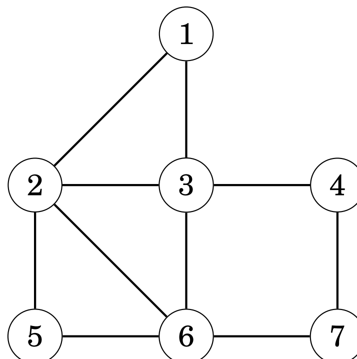
Seuraava algoritmi muodostaa Eulerin kierroksen verkossa, jossa jokaisen solmun aste on parillinen. Algoritmilla voi myös muodostaa Eulerin polun lisäämällä uuden kaaren paritonasteisten solmujen väliin.

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

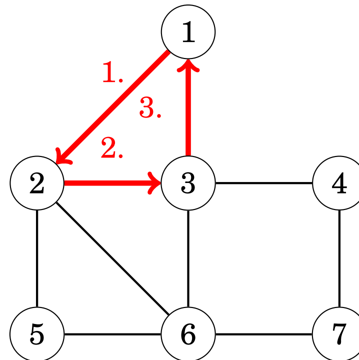
Algoritmi muodostaa ensin pohjan kierrokselle lähtemällä liikkeelle mistä tahansa alkusolmusta ja kulkemalla uusia kaaria eteenpäin, kunnes alkusolmu tulee vastaan uudestaan.

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

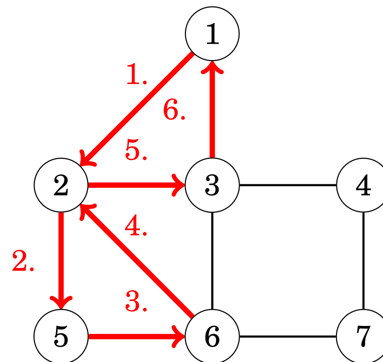
Tarkastellaan algoritmin toimintaa seuraavassa verkossa:



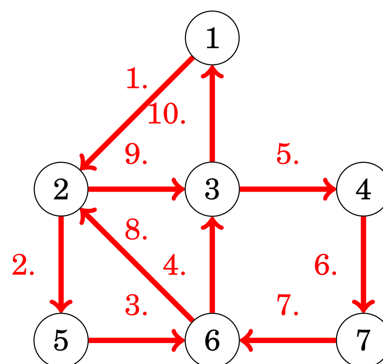
Oletetaan, että algoritmi muodostaa ensimmäisen kierroksen solmusta 1. Siitä syntyy kierros $1 \to 2 \to 3 \to 1$:



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



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



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

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

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

19.3 De Bruijnin jono

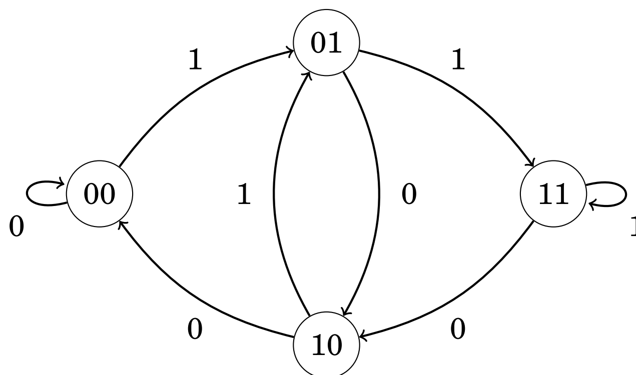
Tarkastellaan lopuksi seuraavaa tehtävää:

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

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

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

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



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

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

Luku 20

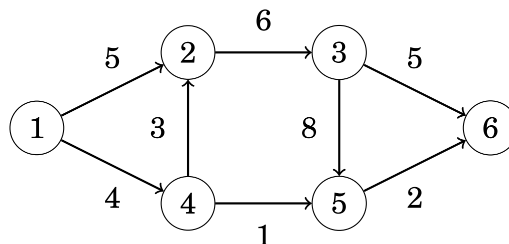
Virtauslaskenta

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

20.1 Maksimivirtaus

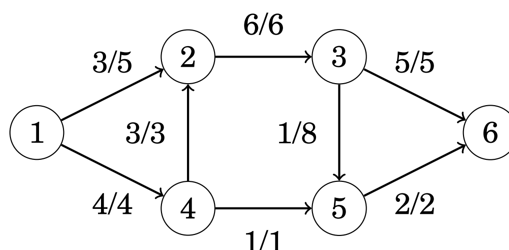
Maksimivirtaus (*maximum flow*) on alkusolmusta loppusolmuun kulkeva virtaus verkossa. Jokaisella verkon kaarella on kapasiteetti, jota kaarta pitkin kulkeva virtaus ei saa ylittää. Lisäksi jokaiseen välisolmuun saapuva virtaus tulee olla yhtä suuri kuin solmusta lähtevä virtaus.

Tarkastellaan esimerkiksi seuraavaa verkkoa:



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

Verkon maksimivirtaus on 7, joka saadaan aikaan seuraavasti:



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

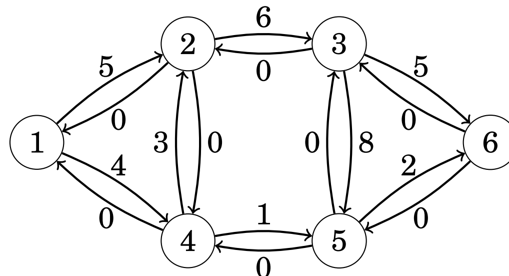
Alkusolmusta lähtevä virtaus ja loppusolmuun saapuva virtaus on verkon maksimivirtaus. Tässä tapauksessa alkusolmusta lähtee virtausta $3+4=7$ ja loppusolmuun saapuu virtausta $5+2=7$.

20.2 Ford-Fulkersonin algoritmi

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

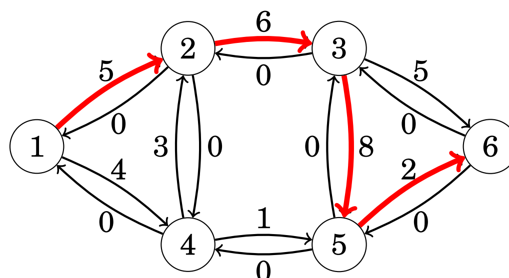
Algoritmi käsittelee verkkoa muodossa, jossa jokaiselle kaarelle on vastakkaiseen suuntaan kulkeva pari. Kaaren paino kuvastaa, miten paljon lisää virtausta sen kautta pystyisi vielä kulkemaan. Aluksi alkuperäisen verkon kaarilla on painona niiden kapasiteetti ja lisätyillä kaarilla on painona 0.

Esimerkkiverkosta syntyy seuraava verkko:



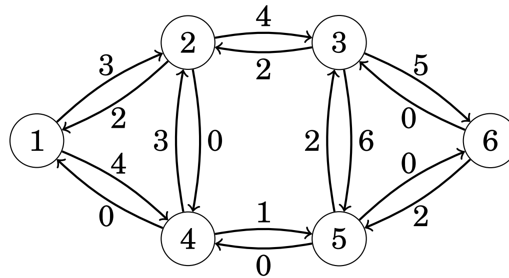
Ford-Fulkersonin algoritmi etsii verkosta joka vaiheessa polun, joka alkaa alkusolmusta, päättyy loppusolmuun ja jossa jokaisen kaaren paino on positiivinen. Jos vaihtoehtoja on useita, mikä tahansa valinta kelpaa.

Esimerkkiverkossa voimme valita vaikkapa seuraavan polun:



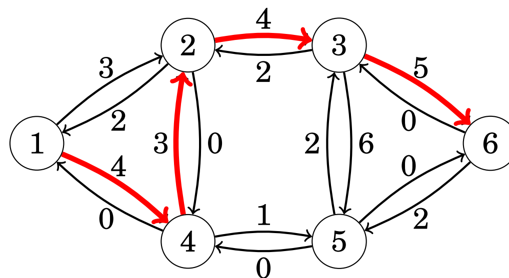
Polun valinnan jälkeen virtaus lisääntyy v yksikköä, jossa v on pienin kaaren paino polulla. Tällöin jokaisen polun osana olevan kaaren paino laskee v:llä ja jokaisen vastakkaiseen suuntaan menevän kaaren paino nousee v:llä.

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

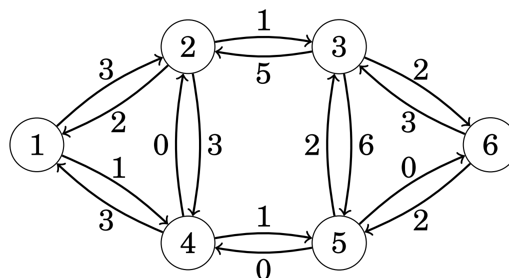


Muutoksessa on ideana, että virtauksen lisääminen vähentää polkuun kuuluvien kaarten kykyä välittää virtausta. Toisaalta virtausta on mahdollista peruuttaa myöhemmin käyttämällä vastakkaiseen suuntaan meneviä kaaria.

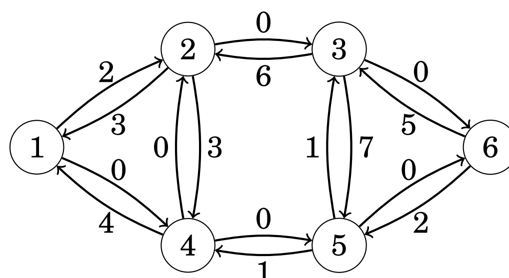
Algoritmi lisää virtausta pikkuhiljaa niin kauan, kuin polkuja alkusolmusta loppusolmuun on olemassa. Esimerkissä seuraava polku voisi olla vaikkapa:



Tämä polku kasvattaa virtausta 3:lla, joten kokonaisvirtaus olisi polun käsittelyn jälkeen 5. Nyt verkko muuttuu seuraavasti:



Maksimivirtaus tulee valmiiksi lisäämällä virtausta vielä polkujen $1 \to 2 \to 3 \to 6$ ja $1 \to 4 \to 5 \to 3 \to 6$ avulla. Molemmat polut tuottavat 1 yksikön lisää virtausta, ja lopullinen verkko on seuraava:



Nyt virtausta ei pysty enää kasvattamaan, koska verkossa ei ole mitään polkua alkusolmusta loppusolmuun, jossa jokaisen kaaren paino olisi positiivinen. Verkon maksimivirtaus on siis 7.

Ford-Fulkersonin algoritmi pysähtyy aina, koska jokainen polku kasvattaa virtausta verkossa. Polun valintatapa kuitenkin vaikuttaa siihen, kuinka tehokas algoritmi on. Pahimmassa tapauksessa jokainen polku lisää virtausta vain 1:llä, jolloin algoritmi voi toimia hitaasti.

Käytännössä hyvä tapa valita polku on käyttää leveyshakua, jolloin polkuun tulee mahdollisimman vähän kaaria. Voidaan osoittaa, että algoritmin aikavaativuus on tällöin $O(nm^2)$, missä n on solmujen määrä ja m on kaarten määrä. Tämä versio Ford-Fulkersonin algoritmista on Edmonds-Karpin algoritmi.

20.3 Sovelluksia

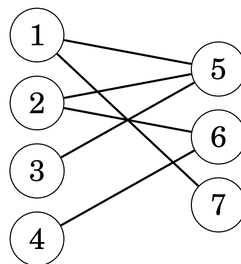
Tavallinen tapa hyödyntää virtauslaskentaa on muuttaa verkko-ongelma sopivalla tavalla maksimivirtauksen laskemiseksi. Seuraavassa on joukko esimerkkejä tällaisista ongelmista.

20.3.1 Maksimiparitus

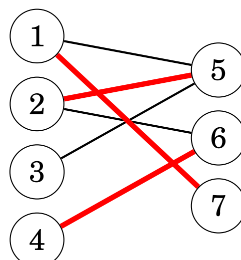
Maksimiparitus (*maximum matching*) on verkko-ongelma, jossa tehtävänä on valita verkosta mahdollisimman suuri joukko kaaria niin, että mikään solmu ei esiinny monen kaaren päätesolmuna.

Maksimiparituksen etsiminen yleisessä verkossa on vaikea ongelma, mutta kaksijakoisessa verkossa se onnistuu maksimivirtauksen avulla. Kaksijakoisessa verkossa solmut jakautuvat kahteen ryhmään, joiden välillä kulkee kaaria.

Tilanne voi näyttää esimerkiksi seuraavalta:

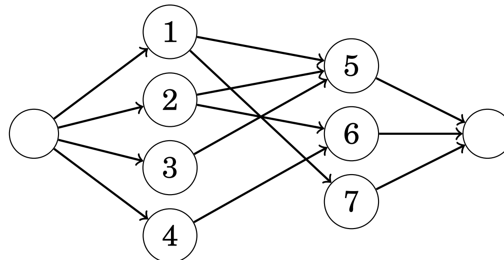


Tämän verkon maksimiparituksessa on 3 paria:



Maksimiparituksen saa muutettua maksimivirtaukseksi lisäämällä verkkoon alkusolmun ja loppusolmun. Alkusolmusta pääsee ensimmäisen ryhmän solmuihin ja loppusolmuun pääsee toisen ryhmän solmuista. Lisäksi ryhmien väliset kaaret muutetaan johtamaan ensimmäisestä ryhmästä toiseen.

Esimerkissä tuloksena on seuraava verkko:

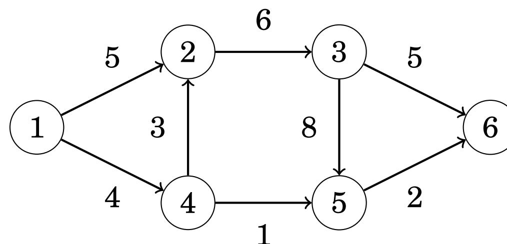


Tämän verkon maksimivirtaus on alkuperäisen verkon maksimiparitus.

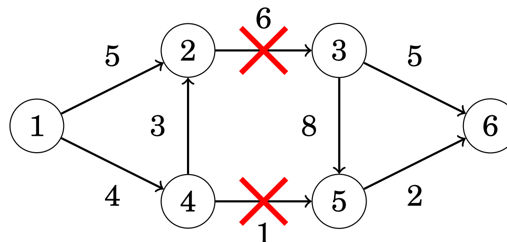
20.3.2 Minimileikkaus

Minimileikkaus (*minimum cut*) on yhteispainoltaan pienin joukko verkon kaaria, joiden poistamisen jälkeen verkossa on kaksi erillistä osaa. Osoittautuu, että maksimivirtaus ja minimileikkaus ovat saman asian kaksi eri puolta.

Esimerkiksi verkon



minimileikkaus on seuraava:

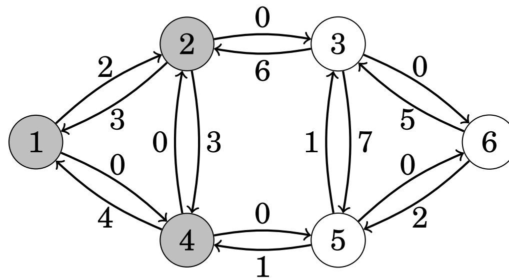


Tässä minimileikkauksessa kaarten yhteispaino on 7.

Yllättävää kyllä, verkon minimileikkaus on aina yhtä suuri kuin maksimivirtaus. Tämä johtuu siitä, että toisaalta minimileikkaus rajoittaa sitä, miten paljon virtausta verkon läpi pystyy kulkemaan, ja toisaalta minimileikkauksen täytyy estää virtauksen kulkeminen verkossa kokonaan.

Maksimivirtauksen ja minimileikkauksen yhteys näkyy Ford-Fulkersonin algoritmin tuottamassa verkossa. Olkoon X niiden solmujen joukko, joihin verkossa pääsee alkusolmusta positiivisia kaaria pitkin.

Esimerkkiverkossa X sisältää solmut 1, 2 ja 4:



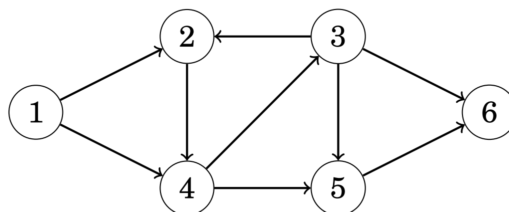
Minimileikkauksen muodostavat ne alkuperäisen verkon kaaret, jotka kulkevat joukosta X joukon X ulkopuolelle ja joiden kapasiteetti on täysin käytetty maksimivirtauksessa. Tässä verkossa kyseiset kaaret ovat $2 \rightarrow 3$ ja $4 \rightarrow 5$.

20.3.3 Polkujoukko

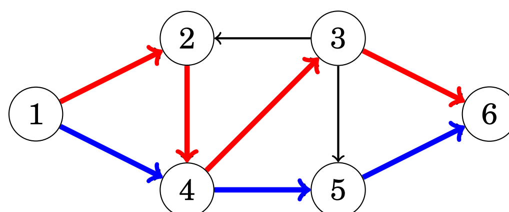
Maksimivirtauksen avulla voi myös etsiä suurimman polkujoukon, jossa jokainen polku alkaa alkusolmusta ja päättyy loppusolmuun ja polut eivät mene päällekkäin missään vaiheessa.

Tarkastellaan ensin tilannetta, jossa samaa kaarta saa käyttää vain yhdessä polussa mutta samaa solmua saa käyttää monessa polussa.

Nyt esimerkiksi verkossa



solmusta 1 solmuun 6 pystyy muodostamaan 2 polkua. Tämä toteutuu valitsemalla polut $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ ja $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$:

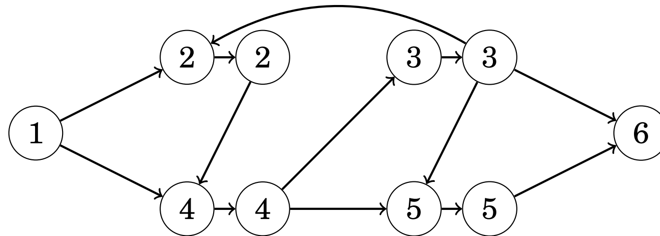


Suurin polkujoukko on sama kuin verkon maksimivirtaus, kun jokaisen kaaren paino on 1. Tämä johtuu siitä, että painon 1 ansiosta jokaista kaarta pystyy käyttämään vain kerran poluissa.

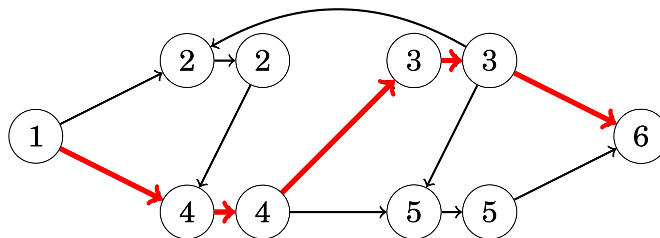
Tarkastellaan sitten tilannetta, jossa myös jokaista solmua (alku- ja loppusolmua lukuun ottamatta) saa käyttää vain kerran.

Tavallinen tapa rajoittaa solmujen kautta kulkemista virtauslaskennassa on korvata jokainen solmu kahdella solmulla, joiden välillä oleva kaari kertoo solmun kapasiteetin. Ensimmäinen solmuista on tulosolmu, johon vain tulee kaaria, ja toinen solmuista on lähtösolmu, josta vain lähtee kaaria.

Esimerkkiverkosta tulee nyt:



Maksimivirtaus tässä verkossa on



mikä vastaa alkuperäisen verkon suurinta polkujoukkoa, jossa jokaista kaarta ja solmua saa käyttää vain kerran: