Osa I

Perusasiat

Luku 1

Ohjelmointitekniikka

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. Niistä voi oppia paljon kisakoodauksesta.

1.1 Kielen valinta

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 komennot 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 $-2 \cdot 10^9$ ja $2 \cdot 10^9$. 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 $-9 \cdot 10^{18}$ ja $9 \cdot 10^{18}$. Jos tehtävässä tarvitsee suuria kokonaislukuja, `long long` riittää yleensä.

Esimerkiksi seuraava koodi määrittelee `long long` -muuttujan:

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

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

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

```
typedef long long ll;
```

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

```
ll x;
```

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

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

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

1.3.2 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. 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 ($\epsilon = 10^{-9}$):

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

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, aikavaativuus on $O(n^k)$.

Suuruusluokka

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

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

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

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

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

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

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

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

Peräkkäisyys

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

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

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

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

Monta muuttujaa

Joskus syötteessä on monta muuttujaa, jotka vaikuttavat aikavaativuuteen. Tällöin 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
...	...
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:

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

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

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

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

$O(n)$ (lineaarinen)

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

Seuraava lineaarinen algoritmi 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 komento `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 $O(n^2)$ tarkoittaa, että algoritmi voi käydä läpi kaikki tavat valita syötteestä kaksi alkiota:

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

$O(n^3)$ (kuutiollinen)

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

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

$O(2^n)$ (osajoukot)

Aikavaativuus $O(2^n)$ voi syntyä siitä, että algoritmi käy läpi syötteen osajoukot. Esimerkiksi taulukon [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 taulukon [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 $n \approx 2000$, algoritmi voi olla $O(n^2)$
- jos $n \approx 10^5$, algoritmi voi olla $O(n \log n)$
- jos $n \approx 10^6$, algoritmi voi olla $O(n)$

Nämä arviot olettavat, että käytössä on tavallinen nykyaikainen tietokone ja koodi saa käyttää sekunnin aikaa syötteen käsittelyyn. Aikavaativuus ei kerro kuitenkaan kaikkea tehokkuudesta, vaan koodin yksityiskohtien optimointi vaikuttaa myös asiaan. 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 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 tulee sisältämään 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 edelleen 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. Annettuna on taulukko, jossa on n alkiota, ja tehtävänä on järjestää taulukon sisältö. Esimerkiksi jos taulukko on $[4,1,5,6,2,5]$, järjestämisen jälkeen sen sisältö on $[1,2,4,5,5,6]$. 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`-komentoon. 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`-komento

Käytännössä järjestämistä ei kannata yleensä toteuttaa itse, vaan parempi ratkaisu on käyttää ohjelmointikielen standardikirjastoon kuuluvaa järjestämisalgoritmia. Esimerkiksi C++:ssa on tehokas komento `sort`, 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`.

3.2.2 Parien järjestäminen

Jos järjestettävät alkiot ovat pareja (`pair`), `sort`-komento 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 $(1,2),(1,5),(2,3)$.

Järjestäminen toimii vastaavasti, jos järjestettävät alkiot ovat `tuple`-tyyppisiä. Silloin järjestys määräytyy ensin ensimmäisen kentän, sitten toisen kentän, sitten kolmannen kentän jne. mukaan.

3.2.3 Tietueiden järjestäminen

Jos järjestettävät alkiot ovat tietueita (`struct`), niiden järjestyksen määrää operaattori <. Operaattorin tulee palauttaa `true`, jos oma alkio on pienempi kuin parametrialkio, ja muuten `false`. Järjestämisen aikana `sort`-komento 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`-komennolle 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 selvittää, missä kohdassa alkio sijaitsee järjestetyssä taulukossa, tai toteaa, että alkiota ei ole taulukossa. Esimerkiksi alkio 5 sijaitsee taulukossa $[2,2,4,5,7,7,7,8]$ kohdassa 3. Binäärihaun aikavaativuus on vain $O(\log n)$, koska se pystyy hyödyntämään taulukon järjestystä.

3.3.1 Toteutus

Seuraavassa on kaksi menetelmää binäärihaun toteuttamiseen. Ensimmäisen menetelmä on tehostettu taulukon läpikäynti vasemmalta oikealle. Toinen menetelmä on perinteinen binäärihaun toteutus, joka puolittaa joka askeleella käsiteltävän taulukon alueen keskikohdan alkion perusteella.

Menetelmä 1

Seuraava binäärihaun toteutus käy taulukon läpi tehokkaasti vasemmalta oikealle käyttäen hyväksi taulukon järjestystä. Ideana on käydä taulukkoa läpi hyppien: aluksi hypyn pituus on $n/2$ askelta, sitten $n/4$ askelta, sitten $n/8$ askelta jne. Mitä lähemmäs etsittävää alkiota päästään, sitä pienempiä hypyt ovat.

Tässä on tämän menetelmän toteutus:

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

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

Menetelmä 2

Perinteinen binäärihaun toteutus muistuttaa sanan etsimistä sanakirjasta. Haku alkaa taulukon keskeltä ja siirtyy siitä vasempaan tai oikeaan osaan sen perusteella, onko etsittävä alkio pienempi vai suurempi kuin keskellä oleva alkio. Tämä jatkuu, kunnes joko alkio löytyy tai etsittävästä välistä tulee tyhjä.

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

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

3.3.2 Muutoskohta

Käytännössä binäärihakua tarvitsee harvoin alkion etsimiseen taulukosta, koska sen sijasta voi käyttää ohjelmointikielen standardikirjaston tietorakenteita. Esimerkiksi C++:n tietorakenne `set` ylläpitää joukkoa, josta voi tarkistaa tehokkaasti ajassa $O(\log n)$, kuuluuko tietty alkio siihen. Sitäkin tärkeämpi binäärihaun käyttökohde on funktion muutoskohdan etsiminen.

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

Tilanne on siis seuraava:

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

Nyt muutoskohta on mahdollista etsiä käyttämällä binäärihakua:

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

Muuttuja x on lopuksi suurin arvo, jolla `ok`(x) on `false`, joten tästä seuraava arvo $x+1$ on pienin arvo, jolla `ok`(x) on `true`. Ylärajan n tulee olla sellainen, että muutoskohta on varmasti sitä ennen.

3.3.3 Suurin arvo

Binäärihaulla voi myös etsiä suurimman alkion taulukossa, jonka alkuosassa jokainen alkio on edellistä suurempi ja loppuosassa jokainen alkio on edellistä pienempi. Toisin sanoen 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 $[2,3,5,8,9,7,4,2]$ suurin arvo on 9 kohdassa 4.

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 etsii kohdan muuttujaan k:

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

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

Luku 4

Tietorakenteet

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

Tämä luku esittelee 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 komennolla `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 tapahtuu aina rakenteen loppuun, ja jonossa 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 komentoja, joita ei ole muissa tietorakenteissa.

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 komentoja taulukon välien käsittelyyn. Komennot on toteutettu niin, että niille annetaan halutun välin alku- ja loppukohta. Yleensä halutaan, että komennot kohdistuvat koko taulukkoon, jolloin väli alkaa kohdasta `begin` ja päättyy kohtaan `end`.

Esimerkiksi seuraava koodi järjestää vektorin ja kääntää sen toisinpäin:

```
vector<int> v;
// ...
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
```

Komentoja voi käyttää myös tavallisen taulukon yhteydessä seuraavasti:

```
int t[n];
// ...
sort(t, t+n);
reverse(t, t+n);
```

4.2 Binäärihakupuu

Binäärihakupuu (*binary search tree*) on puurakenne, jonka keskeiset operaatiot ovat alkion lisäys, haku ja poisto. Binäärihakupuu on mahdollista toteuttaa niin, että kunkin operaation aikavaativuus on vain $O(\log n)$. Tällaisia binäärihakupuita ovat esimerkiksi AVL-puu ja punamusta puu.

Binäärihakupuun avulla voi toteuttaa tietorakenteet joukko ja hakemisto. Joukko sisältää kokoelman alkioita, jota pystyy käsittelemään tehokkaasti. 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 komento `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, komento `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 usein joukon käsittelyssä. Esimerkiksi iteraattori `s.begin()` osoittaa joukon `s` ensimmäiseen alkioon ja iteraattori `s.end()` osoittaa viimeisen alkion jälkeiseen kohtaan. Huomaa epäsymmetria: väli on puoliavoin eli `s.begin()` osoittaa joukkoon mutta `s.end()` joukon ulkopuolelle.

Tilanne on siis tällainen:

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

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

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

Koodin voi kirjoittaa myös lyhyemmin:

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

Iteraattoria vastaavaan joukon alkioon pääsee käsiksi `*`-merkinnällä, eli seuraava koodi tulostaa joukon ensimmäisen alkion:

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

Iteraattoria pystyy liikuttamaan eteen- ja taaksepäin operaatioilla `++` ja `--`. 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 ensimmäiseen alkioon, joka on ainakin yhtä suuri kuin x. Vastaavasti `upper_bound`(x) palauttaa iteraattorin ensimmäiseen alkioon, joka on suurempi kuin x. Jos tällaista alkiota ei ole joukossa, funktiot palauttavat arvon `s.end()`.

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

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

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

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

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 pienim-

män 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. Komento `push` lisää alkion kekoon, komento `top` hakee suurimman alkion ja komento `pop` poistaa suurimman alkion.

Seuraava määrittely luo minimikeon:

```
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 joukko, jossa on n lukua. Tehtävänä on laskea, monessako joukon osajoukossa lukujen summa on x.

Mahdolliset ratkaisut ovat kaikki joukon osajoukot, joiden yhteismäärä on 2^n. Ratkaisu kelpaa, jos osajoukon lukujen summa on x. Esimerkiksi jos joukko on $\{1,3,6,8,9\}$ ja $k = 12$, ratkaisut ovat $\{1,3,8\}$ ja $\{3,9\}$.

Oletetaan, että joukon luvut ovat `t[0]`, `t[1]`, ..., `t[`$n-1$`]`. Seuraava koodi käy kaikki osajoukot läpi ja laskee, monessako osajoukossa lukujen summa on x.

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

// funktion kutsu:
haku(0, 0);
```

Funktion parametri k on käsiteltävän luvun indeksi taulukossa `t`, ja parametri s on muodosteilla olevan osajoukon lukujen summa. Funktio haarautuu joka askeleella kahteen osaan sen mukaan, valitaanko luku `t[`k`]` mukaan osajoukkoon vai ei. Kun $k = n$, osajoukon muodostus on valmis. Funktio laskee ratkaisujen määrän globaaliin muuttujaan c.

Osajoukkojen määrä kasvaa seuraavasti:

joukon koko (n)	osajoukkoja (2^n)
10	$2^{10} \approx 10^3$
20	$2^{20} \approx 10^6$
30	$2^{30} \approx 10^9$
40	$2^{40} \approx 10^{12}$

Tämän vuoksi jos tehtävässä on tarkoitus käydä kaikki joukon osajoukot läpi, n on yleensä 20 tai vähemmän.

Osajoukot on mahdollista käydä läpi myös ilman rekursiota käyttämällä bittioperaatioita. Tästä menetelmästä on lisätietoa luvussa 10.3.

5.2 Permutaatiot

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

Mahdolliset ratkaisut ovat lukujen $1, 2, \ldots, n$ permutaatiot, joita on yhteensä $n!$. Ratkaisu kelpaa, jos $|a - b| \neq 1$, kun a ja b ovat vierekkäin permutaatiossa. Esimerkiksi jos $n = 4$, ratkaisut ovat $(2, 4, 1, 3)$ ja $(3, 1, 4, 2)$.

Seuraava koodi laskee ratkaisujen määrän:

```
int v[n];

void haku(int k, int e) {
    if (k == n) {
        c++;
    } else {
        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;
        }
    }
}

// funktion kutsu:
haku(0, -1);
```

Funktion parametri k on permutaatioon valittujen lukujen määrä, ja parametri e on edellinen permutaatioon valittu luku. Erikoistapauksena alussa $e = -1$, kun permutaatioon ei ole valittu mitään lukua. Joka askeleella funktio käy läpi mahdollisuudet valita seuraava permutaation luku x. Globaali taulukko `v` ker-

too, mitkä luvut on kulloinkin valittu permutaatioon.

Permutaatioiden määrä kasvaa seuraavasti:

joukon koko (n)	permutaatioita ($n!$)
8	$8! \approx 40000$
10	$10! \approx 3 \cdot 10^6$
12	$12! \approx 5 \cdot 10^8$
14	$14! \approx 9 \cdot 10^{10}$

Tämän vuoksi jos tehtävässä on tarkoitus käydä kaikki joukon permutaatiot läpi, n on yleensä 10 tai vähemmän.

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 yksi ratkaisu on:

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

Seuraava koodi laskee reittien määrän:

```
int s[n][n];

void haku(int y, int x, int k) {
    if (y < 0 || x < 0 || 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;
}

// funktion kutsu:
haku(0, 0, 1);
```

Esimerkiksi kun $n = 5$, reittejä on yhteensä 304.

Funktiossa on ideana simuloida ratsun liikettä laudalla. 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 ajankäyttöä on vaikea arvioida, koska haarautumisen määrä riippuu siitä, missä ruudussa ratsu on ja miten pitkälle reitti on edennyt. Käytännössä osoittautuu, että rajakohtana on $n = 5$: kun $n \leq 5$, haku on hyvin nopea, mutta tapauksen $n = 6$ laskeminen vie jo kauan aikaa.

5.4 Puolivälihaku

Puolivälihaku (*meet in the middle*) on tekniikka, jossa hakualue puolitetaan kahteen osaan ja kummallekin osalle suoritetaan erillinen peruuttava haku. Tämän jälkeen hakujen tuottamat osittaiset ratkaisut yhdistetään kokonaisiksi ratkaisuiksi. Menetelmän käyttäminen vaatii, että ongelman voi jakaa kahdeksi erilliseksi osaksi, joiden ratkaisut pystyy yhdistämään tehokkaasti.

Puolivälihaun etuna on, että kummankin peruuttavan haun täytyy käsitellä vain puolet aineistosta. Esimerkiksi jos $n = 40$ ja muodostettavana on osajoukkoja, yksittäinen haku kävisi läpi 2^{40} ratkaisua ja olisi liian hidas. Sen sijaan jakamalla aineisto kahteen osaan kummankin haun riittää käydä läpi 2^{20} ratkaisua, mikä onnistuu tehokkaasti.

Puolivälihakua voi käyttää esimerkiksi aiemmassa tehtävässä (5.1):

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

Ideana on jakaa joukon luvut kahdeksi joukoksi A ja B, joista kumpikin sisältää noin $n/2$ lukua. Kummallekin joukolle suoritetaan peruuttava haku, joka muodostaa kaikki mahdolliset summat joukossa olevista luvuista. Oletetaan, että S_A sisältää A:n summat ja S_B sisältää B:n summat. Tehtävän ratkaisuja ovat parit $a \in S_A$ ja $b \in S_B$, joille pätee $a + b = x$.

Esimerkiksi jos joukko on $\{2,5,6,8\}$ ja $x = 8$, uudet joukot ovat $A = \{2,5\}$ ja $B = \{6,8\}$. Näistä muodostuvat summat $S_A = [0,2,5,7]$ ja $S_B = [0,6,8,14]$. Ratkaisuja voidaan muodostaa kaksi: $a = 0$ ja $b = 8$ sekä $a = 2$ ja $b = 6$. Nämä vastaavat osajoukkoja $\{8\}$ ja $\{2,6\}$.

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 tehtävään. 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

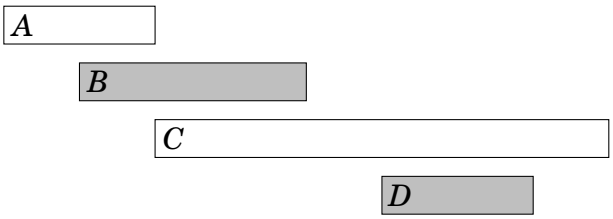
Monet aikataulutukseen liittyvät ongelmat on mahdollista ratkaista ahneilla algoritmeilla. Tutustumme seuraavaksi klassiseen tehtävään, johon on olemassa monta luontevaa ahnetta algoritmia. Kuitenkin vain yksi ahneista algoritmeista tuottaa aina optimaalisen ratkaisun.

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:

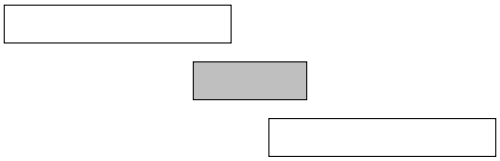


Tarkastellaan seuraavaksi kolmea ahnetta algoritmia tehtävän ratkaisuun.

Algoritmi 1

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

Tämä algoritmi ei kuitenkaan toimi esimerkiksi seuraavassa tilanteessa:

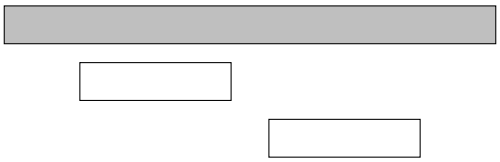


Kun lyhyt tapahtuma valitaan mukaan, on mahdollista osallistua vain yhteen tapahtumaan. Kuitenkin valitsemalla pitkät tapahtumat olisi mahdollista osallistua kahteen tapahtumaan.

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.

6.4.1 Itseisarvosumma

Ensimmäinen tehtävä on etsiä luku x, joka minimoi summan

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

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

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

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 $(1+2+9+2+6)/5 = 5$.

Tämän voi johtaa järjestämällä summan uudestaan muotoon

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

Ensimmäinen osa ei riipu x:stä, joten sen voi unohtaa. Jäljelle jäävistä osista muodostuu funktio $nx^2 - 2xs$, missä $s = \mathtt{t}[0]+\mathtt{t}[1]+\cdots+\mathtt{t}[n-1]$. 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 `k[0]`, `k[1]`, ..., `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 $O(xn)$, kun lasketaan vastaus rahamäärälle x.

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 kaikille 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 ongelmaan 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];
}
```

7.2 Pisin nouseva alijono

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

Taulukon t alijono on $t[a_1], t[a_2], \ldots, t[a_m]$, missä m on alijonon pituus ja $0 \leq a_1 < a_2 < \cdots < a_m < n$. Alijono on nouseva, jos $t[a_1] < t[a_2] < \cdots < t[a_m]$. 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 Bittijono

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.

Tästä syntyy dynaamisen ohjelmoinnin ratkaisu:

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

Koodi laskee hauskojen n-pituisten bittijonojen määrän muuttujaan c, ja koodin aikavaativuus on $O(n)$.

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:

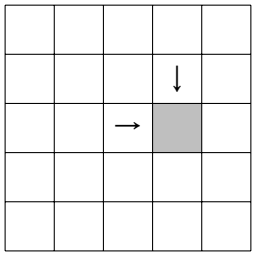
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

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 $f(n-1,n-1)$ on suurin summa reitillä vasemmasta yläkulmasta oikeaan alakulmaan.

Äskeisessä ruudukossa esimerkiksi $f(0,0) = 3$, $f(2,0) = 13$ ja $f(4,4) = 67$.

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



Kun r[y][x] on ruudukon luku kohdassa (y,x), rekursion perustapaukset ovat:

$$\begin{aligned} f(0,0) &= \texttt{r}[0][0] \\ f(0,x) &= f(0,x-1) + \texttt{r}[0][x] \\ f(y,0) &= f(y-1,0) + \texttt{r}[y][0] \end{aligned}$$

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

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

Dynaamisen ohjelmoinnin ratkaisuksi tulee:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == 0 && j == 0) {
            d[i][j] = r[i][j];
        } else if (i == 0) {
            d[i][j] = d[i][j-1]+r[i][j];
        } else if (j == 0) {
            d[i][j] = d[i-1][j]+r[i][j];
        } else {
            d[i][j] = max(d[i][j-1],d[i-1][j])+r[i][j];
        }
    }
}
```

Koodin aikavaativuus on $O(n^2)$ eli paras mahdollinen.

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 $27-8 = 19$:

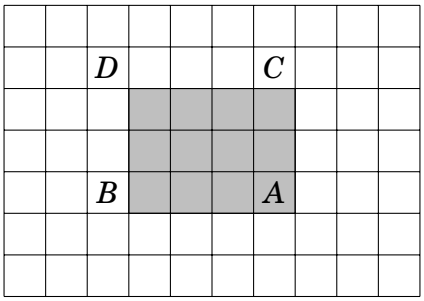
1	4	8	16	22	23	27	29

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:



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. Yleensä 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

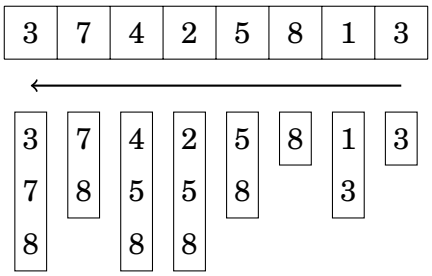
3	7	4	2	5	8	1	3

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.

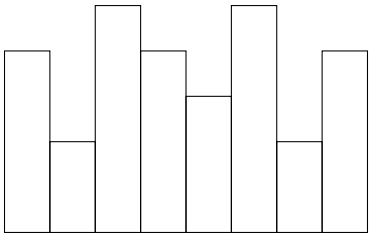
Ideana on, että listan ensimmäinen luku on aina välin pienin luku. Kun ikkuna liikkuu eteenpäin, listan lopusta poistetaan lukuja niin kauan kuin kuin ikkunan uusi luku on pienempi tai yhtä suuri kuin listan viimeinen luku. Tämän jälkeen ikkunan uusi luku lisätään listaan. Lisäksi jos listan ensimmäinen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

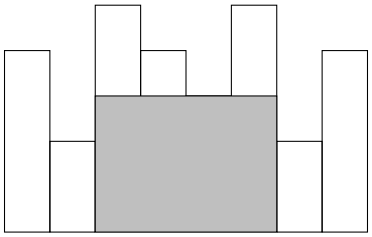
8.3.3 Suurin suorakulmio

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

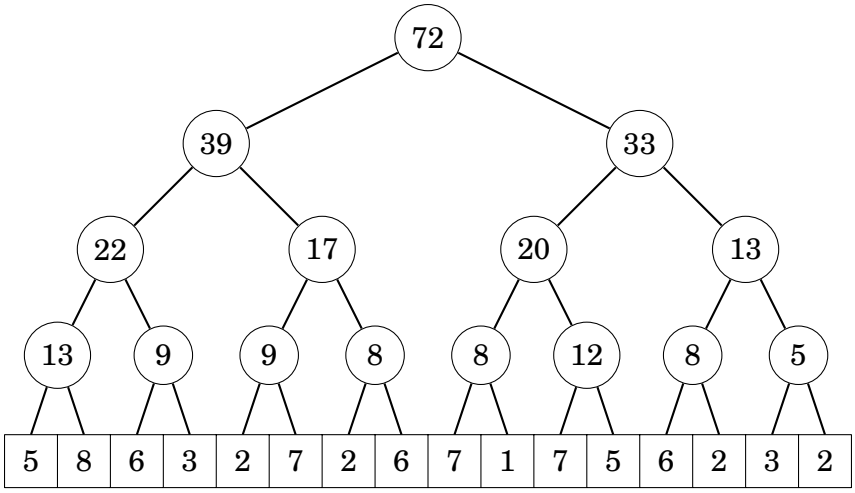
Segmenttipuu (*segment tree*) on tietorakenne, jonka avulla taulukkoon voi toteuttaa tehokkaasti monenlaisia välikyselyitä. Tavallisia välikyselyitä ovat taulukon välin summan, minimin ja maksimin laskeminen. Segmenttipuu mahdollistaa sekä välikyselyn että taulukon muuttamisen ajassa $O(\log n)$.

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 monia 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.

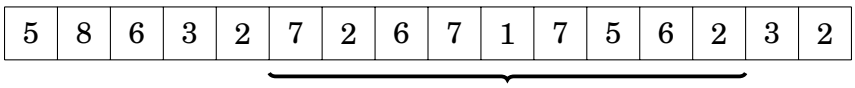
9.2 Operaatiot

Segmenttipuun operaatiot ovat välikyselyn suoritus sekä taulukon muuttaminen. Puurakenteen ansiosta kumpikin operaatio on mahdollista toteuttaa niin, että operaation aikavaativuus on $O(\log n)$

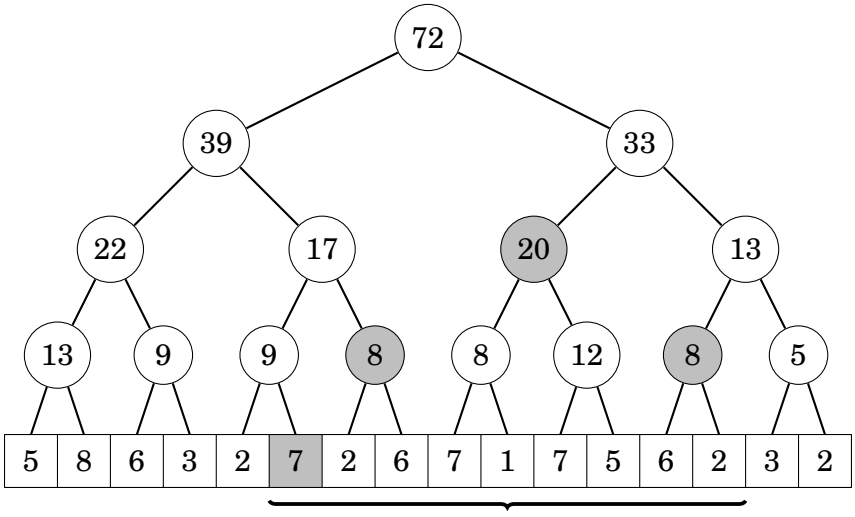
9.2.1 Välikysely

Oletamme edelleen, että välikyselynä on summan laskeminen. Ideana on laskea summa käyttäen mahdollisimman korkealla puussa olevia summia.

Esimerkiksi taulukon välin

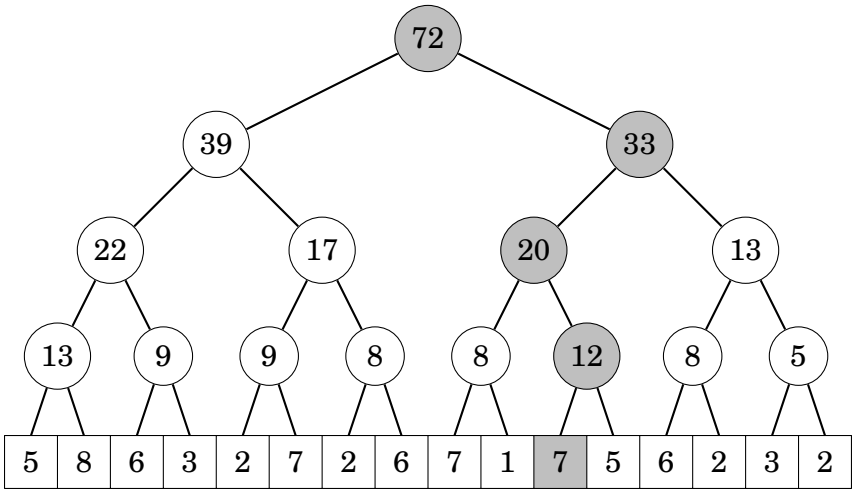


summa muodostuu seuraavista osista:



9.2.2 Muuttaminen

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



9.2.3 Toteutus

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

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

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

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

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

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

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];
    }
}
```

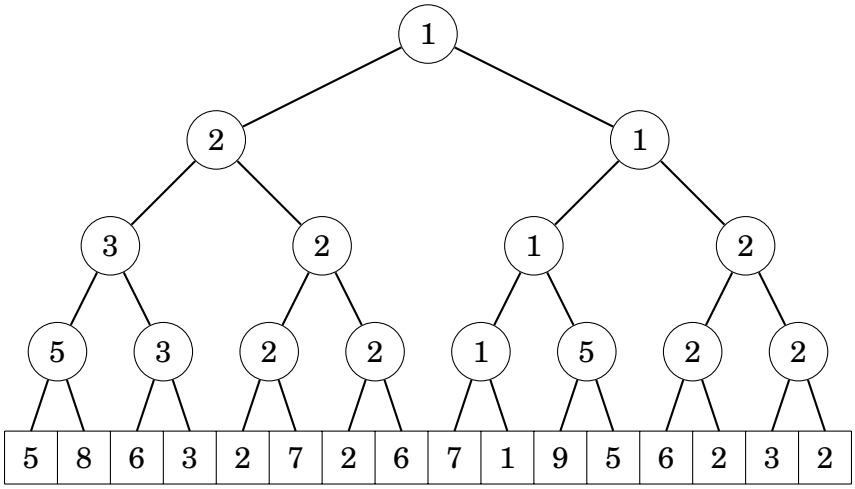
Ensin funktio tekee muutoksen puun alimmalle tasolle taulukkoon. Tämän jälkeen se päivittää kaikki osasummat puun huipulle asti. Taulukon p indeksoinnin ansiosta kohdasta k alemmalla tasolla ovat kohdat $2k$ ja $2k+1$.

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 $a \ldots b$ ja $c \ldots d$ tuloksista pystyy laskemaan tehokkaasti välin $a \ldots d$ 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 Välin muuttaminen

Tähän mennessä binääri-indeksipuun ja 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 samoja puurakenteita 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 binääri-indeksipuun ja 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 0
- 11/2 = 5, jää 1
- 5/2 = 2, jää 1
- 2/2 = 1, jää 0
- 1/2 = 0, jää 1

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

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

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:

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

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 $t[0], t[1], \ldots, t[n-1]$. 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 valinta 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.