

Tietorakenteet ja algoritmit

Antti Laaksonen

29. tammikuuta 2019

Alkusanat

Ohjelmoinnin oppiminen on pitkä prosessi, josta voi erottaa kaksi vaihetta. Ensimmäinen vaihe on oppia ohjelmoinnin perustaidot, kuten miten käytetään muuttujia, ehtolauseita, silmukoita ja taulukoita. Ohjelmoinnin peruskurssit käsittelevät näitä aiheita. Toinen vaihe, johon keskitymme tällä kursilla, on oppia luomaan *tehokkaita* algoritmeja.

Kun saamme eteemme ohjelmointiongelman, se on portti seikkailuun, jossa voi odottaa monenlaisia haasteita. Kaikki keinot ovat sallittuja, kunhan vain saamme aikaan algoritmin, joka ratkaisee ongelman tehokkaasti. Tämä kurssi opettaa monia tekniikoita ja ideoita, joista on hyötyä algoritmisten ongelmien ratkaisemisessa.

Algoritmien suunnittelu on keskeisessä asemassa tietojenkäsittelytieteen teoreettisessa tutkimuksessa, mutta tehokkaat algoritmit ovat tärkeitä myös monissa käytännön sovelluksissa. Tulemme huomaamaan kurssin aikana jatkuvasti, mikä yhteys teoreettisilla tuloksilla on siihen, miten hyvin algoritmit toimivat käytännössä.

Jos sinulla on palautetta kirjasta, voit lähettää sitä sähköpostitse osoitteeseen ahslaaks@cs.helsinki.fi. Lukijoiden palautteesta on paljon hyötyä kirjan kehittämisessä. Kirjan uusin versio on aina saatavilla GitHubissa osoitteessa <https://github.com/pllk/tirakirja>.

Sisältö

Alkusanat	i
1 Johdanto	1
1.1 Mitä algoritmit ovat?	1
1.2 Ohjelmoinnin peruspalikat	2
1.3 Rekursio	6
1.3.1 Osajoukkojen läpikäynti	6
1.3.2 Permutaatioiden läpikäynti	8
1.3.3 Peruuttava haku	8
2 Tehokkuus	11
2.1 Aikavaativuus	11
2.1.1 Laskusääntöjä	12
2.1.2 Yleisiä aikavaativuuksia	14
2.1.3 Tehokkuuden arviointi	16
2.1.4 Esimerkki: Merkkijonot	17
2.2 Lisää algoritmien analysoinnista	19
2.2.1 Merkinnät O , Ω ja Θ	19
2.2.2 Tilavaativuus	21
2.2.3 Rajojen todistaminen	22
3 Järjestäminen	25
3.1 Järjestäminen ajassa $O(n^2)$	25
3.1.1 Lisäysjärjestäminen	25
3.1.2 Inversiot	27
3.2 Järjestäminen ajassa $O(n \log n)$	28
3.2.1 Lomitusjärjestäminen	28
3.2.2 Pikajärjestäminen	30
3.2.3 Algoritmien vertailua	33
3.3 Järjestämisen alaraja	33
3.3.1 Alarajatodistus	34

3.3.2	Laskemisjärjestäminen	34
3.4	Järjestäminen Javassa	35
3.5	Järjestämisen sovelluksia	37
3.5.1	Taulukkoalgoritmeja	37
3.5.2	Binäärihaku	38
4	Lista	41
4.1	Taulukkolista	41
4.1.1	Muutokset lopussa	41
4.1.2	Muutokset alussa ja lopussa	43
4.2	Linkitetty lista	44
4.2.1	Linkitetyt rakenteet	45
4.2.2	Listan operaatiot	46
4.2.3	Listojen vertailua	47
4.3	Pino ja jono	48
4.4	Javan toteutukset	49
4.4.1	ArrayList-rakenne	49
4.4.2	ArrayDeque-rakenne	50
4.4.3	LinkedList-rakenne	51
4.5	Tehokkuusvertailu	51
5	Hajautustaulu	55
5.1	Hajautustaulun toiminta	55
5.1.1	Hajautusfunktio	56
5.1.2	Hajautuksen tehokkuus	58
5.1.3	Hajautustaulu hakemistona	58
5.2	Javan toteutukset	59
5.2.1	HashSet-rakenne	59
5.2.2	HashMap-rakenne	60
5.2.3	Omat luokat	61
5.3	Hajautustaulun käyttäminen	61
6	Binäärihakupuu	65
6.1	Taustaa binääripuista	65
6.1.1	Binääripuun käsittely	66
6.1.2	Läpikäyntijärjestykset	67
6.2	Binäärihakupuun toiminta	68
6.2.1	Operaatioiden toteutus	68
6.2.2	Operaatioiden tehokkuus	71
6.3	AVL-puu	71
6.3.1	Tasapainoehto	72

6.3.2	Kiertojen toteuttaminen	73
6.4	Javan toteutukset	76
6.4.1	TreeSet-rakenne	76
6.4.2	TreeMap-rakenne	76
6.4.3	Omat luokat	77
6.5	Tehokkuusvertailu	77
7	Keko	81
7.1	Binäärikeko	81
7.1.1	Keon tallentaminen	82
7.1.2	Operaatioiden toteutus	83
7.2	Prioriteettijono	85
7.3	Tehokkuusvertailu	85
7.4	Lisää keosta	86
7.4.1	Taulukosta keoksi	87
7.4.2	Kekojärjestäminen	88
8	Algoritmien suunnittelu	91
8.1	Ratkaisun vaiheet	91
8.2	Algoritmien aineksia	93
8.2.1	Järjestäminen	93
8.2.2	Tietorakenteet	95
8.2.3	Binäärihaku	96
8.2.4	Tasoitettu analyysi	98
9	Dynaaminen ohjelmointi	101
9.1	Perustekniikat	101
9.1.1	Rekursiivinen esitys	102
9.1.2	Tehokas toteutus	103
9.2	Esimerkkejä	104
9.2.1	Pisin nouseva alijono	104
9.2.2	Reitti ruudukossa	106
9.2.3	Repunpakkaus	107
9.2.4	Binomikertoimet	109
10	Verkkojen perusteet	111
10.1	Verkkojen käsitteitä	111
10.2	Verkot ohjelmoinnissa	114
10.2.1	Vieruslistaesitys	114
10.2.2	Kaarilistaesitys	115
10.2.3	Vierusmatriisiesitys	116

10.3	Verkon läpikäynti	116
10.3.1	Syvyyshaku	117
10.3.2	Leveyshaku	119
10.3.3	Esimerkki: Labyrintti	120
11	Lyhimmät polut	123
11.1	Lyhimmät polut lähtösolmusta	123
11.1.1	Bellman–Fordin algoritmi	124
11.1.2	Dijkstran algoritmi	127
11.1.3	Esimerkki: Reittiopas	129
11.2	Kaikki lyhimmät polut	130
11.2.1	Floyd–Warshallin algoritmi	131
11.2.2	Algoritmien vertailua	132
12	Suunnatut syklittömät verkot	135
12.1	Topologinen järjestys	136
12.1.1	Järjestyksen muodostaminen	136
12.1.2	Esimerkki: Kurssivalinnat	138
12.2	Dynaaminen ohjelmointi	139
12.2.1	Polkujen laskeminen	139
12.2.2	Ongelmat verkkoina	140
12.3	Vahvasti yhtenäisyys	141
12.3.1	Kosarajun algoritmi	141
12.3.2	Esimerkki: Luolapeli	143
13	Komponentit ja virittävät puut	145
13.1	Union-find-rakenne	145
13.1.1	Rakenteen toteutus	146
13.1.2	Esimerkki: Kaupungit	148
13.2	Pienin virittävä puu	149
13.2.1	Kruskalin algoritmi	149
13.2.2	Primin algoritmi	151
13.2.3	Miksi algoritmit toimivat?	152
14	Maksimivirtaus	155
14.1	Maksimivirtauksen laskeminen	155
14.1.1	Ford–Fulkersonin algoritmi	156
14.1.2	Yhteys minimileikkaukseen	157
14.1.3	Polkujen valitseminen	159
14.2	Maksimivirtauksen sovelluksia	160
14.2.1	Erilliset polut	160

14.2.2	Maksimiparitus	161
14.2.3	Pienin polkupeite	162
15	NP-ongelmat	165
15.1	Vaativuusluokat	165
15.1.1	Luokka P	165
15.1.2	Luokka NP	166
15.1.3	P vs. NP	166
15.1.4	Muita luokkia	167
15.2	NP-täydellisyys	168
15.2.1	SAT-ongelma	168
15.2.2	Ongelmien palautukset	169
15.2.3	Lisää ongelmia	172
15.2.4	Optimointiongelmat	173
15.3	Ongelmien ratkaiseminen	173
A	Matemaattinen tausta	177

Luku 1

Johdanto

Kurssin *Tietorakenteet ja algoritmit* tarkoituksena on opettaa menetelmiä, joiden avulla voimme ratkaista *tehokkaasti* laskennallisia ongelmia. Ohjelmoinnin peruskursseilla olemme keskittyneet ohjelmointitaidon opetteluun. Nyt on aika siirtyä askel eteenpäin ja alkaa kiinnittää huomiota myös siihen, miten nopeasti algoritmit toimivat.

Algoritmien tehokkuudella on suuri merkitys käytännössä. Esimerkiksi netissä toimiva reittiopas on käyttökelpoinen sen vuoksi, että se antaa ehdotuksen reitistä heti sen jälkeen, kun olemme ilmoittaneet, mistä mihin haluamme matkustaa. Jos reittiehdotusta pitäisi odottaa vaikkapa minuutti tai tunti, tämä rajoittaisi paljon palvelun käyttöä.

Jotta reittiopas toimisi tehokkaasti, sen taustalla on hyvin suunniteltu algoritmi. Tällä kurssilla opimme, kuinka voimme luoda itse vastaavia algoritmeja. Tutustumme kurssilla sekä algoritmien suunnittelun teoriaan että käytäntöön – haluamme ymmärtää syvällisesti, mistä algoritmeissa on kysymys, mutta myös osata toteuttaa niitä käytännössä.

1.1 Mitä algoritmit ovat?

Algoritmi on toimintaohje, jota seuraamalla voimme ratkaista jonkin laskennallisen ongelman. Algoritmille annetaan *syöte* (*input*), joka kuvaa ratkaistavan ongelman tapauksen, ja algoritmin tulee tuottaa *tuloste* (*output*), joka on vastaus sille annettuun syötteeseen.

Tarkastellaan esimerkkinä ongelmaa, jossa syötteenä on n kokonaislukua sisältävä taulukko ja tehtävänä on laskea lukujen summa. Esimerkiksi jos syöte on $[2, 4, 1, 8]$, haluttu tuloste on 15, koska $2 + 4 + 1 + 8 = 15$. Voimme ratkaista tämän ongelman algoritmilla, joka käy luvut läpi silmukalla ja laskee niiden summan muuttujaan.

Algoritmin toiminnan esittämiseen on useita mahdollisuuksia. Yksi tapa on selostaa sanallisesti, kuinka algoritmi toimii, kuten teimme äsken. Toinen tapa taas on antaa koodi, joka toteuttaa algoritmin. Tällöin meidän täytyy valita jokin ohjelmointikieli, jonka avulla esitämme algoritmin. Esimerkiksi seuraava Java-koodi kuvaa algoritmin, joka laskee lukujen summan:

```
int summa = 0;
for (int i = 0; i < n; i++) {
    summa += luvut[i];
}
System.out.println(summa);
```

Voimme myös esittää algoritmin *pseudokoodina* todellisen ohjelmointikielen sijasta. Tämä tarkoittaa, että kirjoitamme koodia, joka on lähellä käytössä olevia ohjelmointikieliä, mutta voimme päättää koodin tarkan kirjoitusasun itse ja ottaa joitakin vapauksia, joiden ansiosta voimme kuvata algoritmin mukavammin. Voisimme esimerkiksi esittää äskeisen algoritmin pseudokoodina seuraavasti:

```
summa = 0
for i = 0 to n-1
    summa += luvut[i]
print(summa)
```

Tässä kirjassa esitämme algoritmeja sekä Java-koodina että pseudokoodina tilanteesta riippuen. Käytämme Java-koodia silloin, kun haluamme kiinnittää huomiota siihen, miten jokin asia toteutetaan tarkalleen Javassa. Pseudokoodia käytämme taas silloin, kun haluamme kuvata algoritmin yleisen idean eikä käytetyllä kielellä ole merkitystä. Taulukko 1.1 näyttää vertailun kirjan pseudokoodin ja Java-koodin merkintätavoista.

Tärkeä työkalu algoritmien suunnittelussa on *matematiikka*, jonka avulla voi perustella täsmällisesti, miksi jokin algoritmi toimii ja käyttää tietyn verran aikaa. Tutustumme tarvittaviin matematiikan asioihin pikkuhiljaa kirjan aikana. Kirjan lopussa olevassa liitteessä on lisäksi yhteenveto matematiikan merkinnöistä ja kaavoista, joita kirjassa käytetään.

1.2 Ohjelmoinnin peruspalikat

Kiehtova seikka ohjelmoinnissa on, että monimutkaisetkin algoritmit syntyvät yksinkertaisista aineksista. Käymme seuraavaksi läpi ohjelmoinnin peruspalikat, jotka muodostavat pohjan algoritmien suunnittelulle.

pseudokoodi	Java-koodi
<pre>x = 5 t = [1,2,3]</pre>	<pre>int x = 5; int[] t = {1,2,3};</pre>
<pre>if a == b // koodia</pre>	<pre>if (a == b) { // koodia }</pre>
<pre>while a <= b // koodia</pre>	<pre>while (a <= b) { // koodia }</pre>
<pre>for i = 1 to n // koodia</pre>	<pre>for (int i = 1; i <= n; i++) { // koodia }</pre>
<pre>for i = n to 1 // koodia</pre>	<pre>for (int i = n; i >= 1; i--) { // koodia }</pre>
<pre>sort(x)</pre>	<pre>Arrays.sort(x);</pre>
<pre>print(x)</pre>	<pre>System.out.println(x);</pre>
<pre>swap(a,b)</pre>	<pre>t = a; a = b; b = t;</pre>
<pre>a = min(x,y) b = max(x,y)</pre>	<pre>a = Math.min(x,y); b = Math.max(x,y);</pre>
<pre>procedure toisto(x) print(x) print(x)</pre>	<pre>void testi(int x) { System.out.println(x); System.out.println(x); }</pre>
<pre>function summa(a,b) return a+b</pre>	<pre>int summa(int a, int b) { return a+b; }</pre>

Taulukko 1.1: Pseudokoodin ja Java-koodin vertailu.

Muuttuja

Muuttuja (*variable*) säilyttää algoritmissa tarvittavia tietoja. Esimerkiksi seuraavassa koodissa on kaksi muuttujaa:

```
a = 3
b = 4
print(a+b)
```

Pseudokoodissa käytäntönä on, ettei muuttujia tarvitse määritellä vaan voimme alkaa käyttää niitä suoraan.

Ehtolause

Ehtolause (*conditional statement*) saa ohjelman toiminnan riippumaan muuttujista. Esimerkiksi seuraava koodi kertoo, onko x parillinen vai pariton:

```
if x%2 == 0
    print("parillinen")
else
    print("pariton")
```

Silmukka

Silmukka (*loop*) toistaa koodia. Esimerkiksi seuraava for-silmukka tulostaa luvut $1, 2, 3, \dots, 10$.

```
for i = 1 to 10
    print(i)
```

Seuraava while-silmukka puolestaan puolittaa lukua x , kunnes se on pienempi kuin 1:

```
while x >= 1
    print(x)
    x /= 2
```

Taulukko

Taulukko (*array*) on ohjelmoinnin tavallisin *tietorakenne* (*data structure*) eli tapa säilyttää kokoelmaa tietoa ohjelmassa. Taulukossa on n alkiota, joiden kohdat ovat $0, 1, \dots, n - 1$.

Esimerkiksi seuraava koodi luo taulukon ja käsittelee sen kohdassa 2 olevaa alkia:

```
x = [4,2,7,5,1]
print(x[2]) // 7
x[2] = 3
print(x[2]) // 3
```

Seuraava koodi puolestaan tulostaa kaikki n -kokoisen taulukon alkiot:

```
for i = 0 to n-1
    print(x[i])
```

Tutustumme kirjan aikana moniin tietorakenteisiin, joista on hyötyä algoritmien suunnittelussa. Pystymme kuitenkin halutessamme toteuttamaan minkä tahansa tietorakenteen taulukon avulla. Tämä johtuu siitä, että tietokoneen muisti on pohjimmiltaan suuri taulukko.

Aliohjelma

Aliohjelma (*subprogram*) on ohjelman osa, jota voi kutsua parametreilla. Javassa aliohjelmasta käytetään nimeä *metodi* (*method*).

Tässä kirjassa käytäntönä on, että *proseduuri* (*procedure*) on aliohjelma, joka ei palauta mitään (Javassa `void`), ja *funktio* (*function*) on aliohjelma, jolla on palautusarvo.

Esimerkiksi seuraava proseduuri tulostaa kahdesti parametrinsa:

```
procedure toisto(x)
    print(x)
    print(x)
```

Seuraava funktio puolestaan palauttaa parametriensa summan:

```
function summa(a,b)
    return a+b
```

Olemme nyt käyneet läpi ainekset, joiden avulla voimme toteuttaa *minkä tahansa* algoritmin. On huojentava tieto, että näinkin pieni määrä tekniikoita riittää algoritmien suunnittelussa. Nyt kaikki on vain kiinni siitä, miten osaamme *soveltaa* näitä tekniikoita eri tilanteissa.

1.3 Rekursio

Rekursio (*recursion*) on hyödyllinen ohjelmointitekniikka, joka jää kuitenkin usein sivurooliin ohjelmoinnin peruskursseilla. Nyt on aika perehtyä kunnolla siihen, mitä hyötyä rekursiosta on. Osoittautuu, että voimme toteuttaa monia algoritmeja kätevästi rekursion avulla.

Rekursiossa on ideana tehdä aliohjelma, joka kutsuu itseään. Yksi tapa ajatella rekursiota on, että voimme muuttaa sen avulla silmukan sarjaksi aliohjelman kutsuja. Tarkastellaan esimerkkinä seuraavaa silmukkaa:

```
for i = 1 to 10
  print(i)
```

Tämä silmukka tulostaa luvut $1, 2, 3, \dots, 10$. Voimme toteuttaa samalla tavalla toimivan ohjelman seuraavasti rekursion avulla:

```
procedure testi(x)
  if x > 10
    return
  print(x)
  testi(x+1)

testi(1)
```

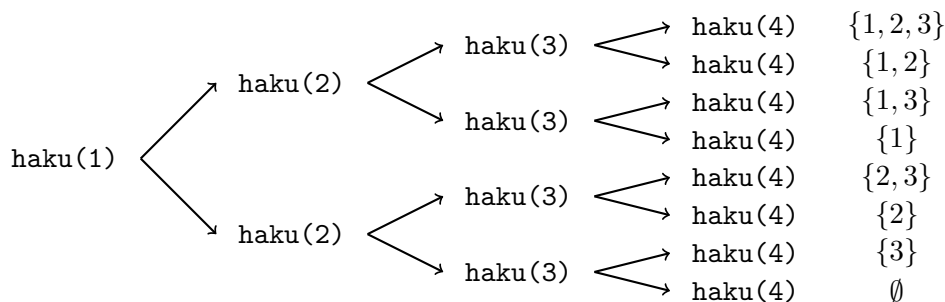
Tässä `testi` on rekursiivinen proseduuri, jolle annetaan parametrina luku x . Jos x on yli 10, proseduuri ei tee mitään (rekursion loppuehto). Muuten proseduuri tulostaa luvun x ja kutsuu sitten itseään parametrilla $x + 1$ (rekursiivinen kutsu). Kun proseduuria kutsutaan parametrilla 1, se tulostaa luvut $1, 2, 3, \dots, 10$ samalla tavalla kuin aiempi silmukka.

Mutta mitä hyötyä rekursiosta on? Se selviää seuraavista esimerkeistä, joissa käytämme rekursiota aidoissa ongelmissa.

1.3.1 Osajoukkojen läpikäynti

Aloitamme ongelmasta, jossa haluamme käydä läpi kaikki lukujen $1, 2, \dots, n$ osajoukot. Esimerkiksi kun $n = 3$, osajoukot ovat \emptyset (tyhjä joukko), $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ ja $\{1, 2, 3\}$. Osajoukkoja on yhteensä 2^n , koska jokaisen luvun kohdalla on kaksi vaihtoehtoa: se tulee tai ei tule mukaan osajoukkoon. Esimerkiksi kun $n = 3$, osajoukkoja on $2^3 = 8$.

Seuraava rekursiivinen proseduuri muodostaa lukujen $1, 2, \dots, n$ osajoukot. Ideana on, että teemme kunkin luvun kohdalla *päätöksen*, tuleeeko se mukaan osajoukkoon vai ei. Toteutamme tämän niin, että proseduuri kutsuu



Kuva 1.1: Osajoukkojen muodostaminen rekursiivisesti ($n = 3$).

itseään molemmissa tapauksissa rekursiivisesti.

```

procedure haku(k)
  if k == n+1
    // käsittele osajoukko
  else
    mukana[k] = true
    haku(k+1)
    mukana[k] = false
    haku(k+1)

```

Proseduurin `haku` parametri k ilmaisee, minkä luvun kohtalon päätämme seuraavaksi. Kutsumme aluksi proseduuria parametrilla $k = 1$. Jos $k = n + 1$, olemme saaneet osajoukon valmiiksi ja voimme käsitellä sen haluamallamme tavalla. Muuten haaraudumme kahteen osaan sen mukaan, tuleeeko luku k mukaan osajoukkoon vai ei. Molemmissa tapauksissa kutsumme proseduuria parametrilla $k + 1$, jolloin siirrymme käsittelemään seuraavan luvun.

Proseduuri merkitsee globaaliin taulukkoon `mukana`, mitkä luvut ovat mukana osajoukossa. Voimme hyödyntää tämän taulukon sisältöä, kun käsittelemme osajoukon tapauksessa $k = n + 1$. Esimerkiksi voimme tulostaa osajoukossa olevat luvut seuraavasti:

```

for i = 1 to n
  if mukana[i]
    print(i)

```

Kuva 1.1 näyttää, miten osajoukot muodostuvat tapauksessa $n = 3$. Jokaisessa proseduurin kutsussa ylempi haara ottaa luvun mukaan osajoukkoon ja alempi haara ei ota lukua mukaan osajoukkoon. Kuvan oikeassa reunassa näkyy kussakin tapauksessa muodostettu osajoukko.

1.3.2 Permutaatioiden läpikäynti

Tarkastellaan sitten toista ongelmaa, jossa haluammekin käydä läpi lukujen $1, 2, \dots, n$ *permutaatiot* eli kaikki mahdolliset tavat asettaa luvut johonkin järjestykseen. Esimerkiksi tapauksessa $n = 3$ permutaatiot ovat $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ ja $(3, 2, 1)$. Luvuista $1, 2, \dots, n$ voi muodostaa kaikkiaan $n!$ permutaatiota. Esimerkiksi tapauksessa $n = 3$ permutaatioiden määrä on $3! = 6$.

Myös permutaatioiden läpikäynti onnistuu kätevästi rekursiolla. Ideana on valita joka askeleella permutaation loppuun jokin luku, joka ei vielä kuulu siihen. Voimme toteuttaa haun näin:

```
procedure haku(k)
  if k == n+1
    // käsittele permutaatio
  else
    for i = 1 to n
      if not mukana[i]
        mukana[i] = true
        permutaatio[k] = i
        haku(k+1)
        mukana[i] = false
```

Parametri k tarkoittaa, mihin permutaation kohtaan valitsemme seuraavaksi luvun. Haku alkaa, kun kutsumme proseduuria parametrilla $k = 1$. Jos $k = n + 1$, olemme saaneet permutaation valmiiksi ja voimme käsitellä sen. Muuten valitsemme permutaation kohtaan k tulevan luvun käymällä läpi silmukalla luvut $1 \dots n$. Taulukko *mukana* kertoo, mitkä luvut ovat jo mukana permutaatiossa. Jos luku i ei ole vielä mukana, haaraudumme tapaukseen, jossa valitsemme sen kohtaan k , ja jatkamme hakua kohtaan $k + 1$.

Kun olemme saaneet permutaation valmiiksi, voimme käsitellä sen esimerkiksi tulostamalla lukujen järjestyksen näin:

```
for i = 1 to n
  print(permutaatio[i])
```

1.3.3 Peruuttava haku

Peruuttava haku (*backtracking*) on yleinen rekursiivinen menetelmä, jota käyttäen voimme muodostaa kaikki ratkaisut annettuun ongelmaan. Ideana on aloittaa tyhjästä ratkaisusta ja käydä joka askeleella läpi rekursiivisesti kaikki mahdolliset tavat, kuinka ratkaisua voi laajentaa.

Peruuttava haku on raa'an voiman algoritmi, ja voimme käyttää sitä vain silloin, kun ratkaisujen määrä on niin pieni, että ehdimme käydä läpi kaikki ratkaisut. Kuitenkin jos voimme käyttää peruuttavaa hakua, se on mainio tekniikka, koska voimme olla varmoja, että oikein toteutettu peruuttava haku löytää kaikki ratkaisut.

Tarkastellaan esimerkkinä tehtävää, jossa haluamme käydä läpi kaikki kokoa $n \times n$ olevat *latinalaiset neliöt* eli ruudukot, joissa kullakin vaaka- ja pystyrivillä esiintyy tarkalleen kerran jokainen luku $1, 2, \dots, n$. Kyseessä on siis yksinkertaistus tutusta sudoku-tehtävästä. Esimerkiksi kuva 1.2 näyttää kaikki 12 latinalaista neliötä kokoa 3×3 .

Toteutamme peruuttavan haun niin, että valitsemme joka askeleella ruudukon kohtaan (y, x) tulevan luvun. Numeroimme ruudukon vaaka- ja pystyrivit kokonaisluvuin $1, 2, \dots, n$. Aloitamme haun ruudukon vasemmasta yläkulmasta ja etenemme rivi kerrallaan alaspäin. Seuraava rekursiivinen algoritmi toteuttaa haun, kun sitä kutsutaan parametreilla $(1, 1)$:

```

procedure haku(y,x)
  if y == n+1
    // käsittele ratkaisu
  else if x == n+1
    haku(y+1,1)
  else
    for i = 1 to n
      if not vaaka[y][i] and not pysty[x][i]
        vaaka[y][i] = pysty[x][i] = true
        nelio[y][x] = i
        haku(y,x+1)
        vaaka[y][i] = pysty[x][i] = false

```

Algoritmin alussa on kaksi erikoistapausta: jos $y = n + 1$, olemme saaneet muodostettua yhden latinalaisen neliön. Jos taas $x = n + 1$, olemme saaneet jonkin vaakarivin valmiiksi ja alamme muodostaa seuraavaa vaakariviä. Muuten kyseessä on perustapaus, jossa haluamme valita kohtaan (y, x) tulevan luvun. Käymme läpi kaikki mahdolliset tavat for-silmukalla, jossa i on valittava luku. Koska jokainen luku saa esiintyä vain kerran kullakin vaaka- ja pystyrivillä, käytämme kahta aputaulukkoa: **vaaka** $[y][i]$ kertoo, onko vaakarivillä y jo lukua i , ja vastaavasti **pysty** $[x][i]$ kertoo, onko pystyrivillä x jo lukua i . Jos voimme sijoittaa luvun i kohtaan (y, x) , merkitsemme tämän taulukkoon **nelio** $[y][x]$ ja lisäksi päivitämme taulukoita **vaaka** ja **pysty**. Sitten jatkamme hakua rekursiivisesti seuraavaan oikealla olevaan ruutuun.

Kun olemme saaneet muodostettua latinalaisen neliön, voimme tulostaa sen sisällön taulukon **nelio** perusteella tai vain kasvattaa laskurin arvoa,

1 2 3 2 3 1 3 1 2	1 2 3 3 1 2 2 3 1	1 3 2 2 1 3 3 2 1	1 3 2 3 2 1 2 1 3	2 1 3 1 3 2 3 2 1	2 1 3 3 2 1 1 3 2
2 3 1 1 2 3 3 1 2	2 3 1 3 1 2 1 2 3	3 1 2 1 2 3 2 3 1	3 1 2 2 3 1 1 2 3	3 2 1 1 3 2 2 1 3	3 2 1 2 1 3 1 3 2

Kuva 1.2: Kaikki 12 latinalaista neliötä kokoa 3×3 .

ruudukon koko n	neliöiden määrä
1	1
2	2
3	12
4	576
5	161280
6	812851200

Taulukko 1.2: Latinalaisten neliöiden määrät, kun $n = 1, 2, \dots, 6$.

jolloin saamme lasketuksi, montako neliötä on olemassa kaikkiaan. Taulukko 1.2 sisältää latinalaisten neliöiden määrät tapauksissa $n = 1, 2, \dots, 6$, jotka pystymme laskemaan nopeasti tässä kuvatulla algoritmilla. Suuremmilla n :n arvoilla pelkkä peruuttava haku on kuitenkin liian hidas, koska ratkaisuja on valtava määrä emmekä voi enää käydä niitä läpi yksitellen.

Luku 2

Tehokkuus

Algoritmien suunnittelussa tavoitteemme on saada aikaan algoritmeja, jotka toimivat *tehokkaasti*. Haluamme luoda algoritmeja, joiden avulla voimme käsitellä myös suuria aineistoja ilman, että joudumme odottamaan kauan aikaa. Ajattelemmekin, että algoritmi on *hyvä*, jos se kykenee antamaan meille nopean vastauksen myös silloin, kun annamme sille paljon tietoa.

Tässä luvussa tutustumme työkaluihin, joiden avulla voimme arvioida algoritmien tehokkuutta. Keskeinen käsite on *aikavaativuus*, joka antaa tiiviissä muodossa kuvauksen algoritmin ajankäytöstä. Aikavaativuuden avulla voimme muodostaa arvion algoritmin tehokkuudesta sen rakenteen perusteella, eikä meidän tarvitse toteuttaa ja testata algoritmia vain saadaksemme tietää, miten nopea se on.

2.1 Aikavaativuus

Algoritmin tehokkuus riippuu siitä, montako askelta se suorittaa. Tavoitteemme on nyt arvioida algoritmin askelten määrää suhteessa syötteen kokoon n . Esimerkiksi jos syötteenä on taulukko, n on taulukon koko, ja jos syötteenä on merkkijono, n on merkkijonon pituus.

Tarkastellaan esimerkkinä seuraavaa algoritmia, joka laskee, montako kertaa alkio x esiintyy n lukua sisältävässä taulukossa.

```
1 laskuri = 0
2 for i = 0 to n-1
3     if luvut[i] == x
4         laskuri++
```

Voimme arvioida algoritmin tehokkuutta tutkimalla jokaisesta rivistä, montako kertaa algoritmi suorittaa sen. Rivi 1 suoritetaan vain kerran al-

goritmin alussa. Tämän jälkeen alkaa silmukka, jossa rivit 2 ja 3 suoritetaan molemmat n kertaa ja rivi 4 puolestaan suoritetaan $0 \dots n$ kertaa riippuen siitä, kuinka usein luku x esiintyy taulukossa. Algoritmi suorittaa siis vähintään $2n + 1$ ja enintään $3n + 1$ askelta.

Näin tarkka analyysi ei ole kuitenkaan yleensä tarpeen, vaan meille riittää määrittää karkea *yläraja* ajankäytölle. Sanomme, että algoritmi toimii ajassa $O(f(n))$ eli sen *aikavaativuus* (*time complexity*) on $O(f(n))$, jos se suorittaa enintään $cf(n)$ askelta aina silloin kun $n \geq n_0$, missä c ja n_0 ovat vakioita. Esimerkiksi yllä oleva algoritmi toimii ajassa $O(n)$, koska se suorittaa selkeästi enintään $4n$ askelta kaikilla n :n arvoilla.

2.1.1 Laskusääntöjä

Aikavaativuuden mukavana puolena on, että voimme yleensä päätellä aikavaativuuden helposti algoritmin rakenteesta. Tutustumme seuraavaksi laskusääntöihin, joiden avulla tämä on mahdollista.

Yksittäiset komennot

Jos koodissa ei ole silmukoita vaan vain yksittäisiä komentoja, sen aikavaativuus on $O(1)$. Näin on esimerkiksi seuraavassa koodissa:

```
c = a+b
if c >= 0
    print(c)
```

Silmukat

Merkitsemme \dots koodia, jonka aikavaativuus on $O(1)$. Jos koodissa on yksi silmukka, joka suorittaa n askelta, sen aikavaativuus on $O(n)$:

```
for i = 1 to n
    ...
```

Jos tällaisia silmukoita on kaksi sisäkkäin, aikavaativuus on $O(n^2)$:

```
for i = 1 to n
    for j = 1 to n
        ...
```

Yleisemmin jos koodissa on vastaavalla tavalla k sisäkkäistä silmukkaa, sen aikavaativuus on $O(n^k)$.

Huomaa, että vakiokertoimet ja matalammat termit eivät vaikuta aikavaativuuteen. Esimerkiksi seuraavissa koodissa silmukoissa on $2n$ ja $n - 1$ askelta, mutta kummankin koodin aikavaativuus on $O(n)$.

```
for i = 1 to 2*n
    ...
```

```
for i = 1 to n-1
    ...
```

Peräkkäiset osuudet

Jos koodissa on peräkkäisiä osuuksia, sen aikavaativuus on suurin yksittäisen osuuden aikavaativuus. Esimerkiksi seuraavan koodin aikavaativuus on $O(n^2)$, koska sen osuuksien aikavaativuudet ovat $O(n)$, $O(n^2)$ ja $O(n)$.

```
for i = 1 to n
    ...
for i = 1 to n
    for j = 1 to n
        ...
for i = 1 to n
    ...
```

Monta muuttujaa

Joskus aikavaativuus riippuu useammasta tekijästä, jolloin kaavassa on monta muuttujaa. Esimerkiksi seuraavan koodin aikavaativuus on $O(nm)$:

```
for i = 1 to n
    for j = 1 to m
        ...
```

Rekursiiviset algoritmit

Rekursiivisessa algoritmissa laskemme, montako rekursiivista kutsua tehdään ja kauanko yksittäinen kutsu vie aikaa. Tarkastellaan esimerkkinä seuraavaa aliohjelmää, jota kutsutaan parametrilla n :

```

procedure f(n)
  if n == 1
    return
  f(n-1)

```

Aliohjelmaa kutsutaan yhteensä n kertaa ja jokainen kutsu vie aikaa $O(1)$. Saamme selville aliohjelman aikavaativuuden kertomalla nämä arvot keskenään, joten aliohjelma vie aikaa $O(n)$.

Tarkastellaan sitten seuraavaa aliohjelmaa:

```

procedure g(n)
  if n == 1
    return
  g(n-1)
  g(n-1)

```

Tässä tapauksessa jokainen aliohjelman kutsu tuottaa kaksi uutta kutsua, joten aliohjelmaa kutsutaan kaikkiaan

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

kertaa. Jokainen kutsu vie aikaa $O(1)$, joten aikavaativuus on $O(2^n)$.

2.1.2 Yleisiä aikavaativuuksia

Tietyt aikavaativuudet esiintyvät usein algoritmeissa. Käymme seuraavaksi läpi joukon tällaisia aikavaativuuksia.

$O(1)$ (vakioaikainen)

Vakioaikainen (*constant time*) algoritmi suorittaa kiinteän määrän komentoja, eikä syötteen suuruus vaikuta algoritmin nopeuteen. Esimerkiksi seuraava algoritmi laskee summan $1 + 2 + \dots + n$ vakioajassa kaavalla $n(n+1)/2$:

```

summa = n*(n+1)/2

```

$O(\log n)$ (logaritminen)

Logaritminen (*logarithmic*) algoritmi puolittaa usein syötteen koon joka askeleella. Esimerkiksi seuraavan algoritmin aikavaativuus on $O(\log n)$:


```
laskuri = 0
while n >= 1
    laskuri++
    n /= 2
```

Tärkeä seikka logaritmeihin liittyen on, että $\log n$ on *pieni* luku, kun n on mikä tahansa tyypillinen algoritmeissa esiintyvä luku. Esimerkiksi $\log 10^6 \approx 20$ ja $\log 10^9 \approx 30$, kun logaritmin kantaluku on 2. Niinpä jos algoritmi tekee jotain logaritmisessa ajassa, siinä ei kulu kauan aikaa.

$O(n)$ (lineaarinen)

Lineaarinen (*linear*) algoritmi voi käydä läpi syötteen kiinteän määrän kertoja. Esimerkiksi seuraava $O(n)$ -algoritmi laskee taulukon lukujen summan:

```
summa = 0
for i = 0 to n-1
    summa += taulu[i]
```

Kun algoritmin syötteenä on aineisto, jossa on n alkia, lineaarinen aikavaativuus on yleensä paras mahdollinen, minkä voimme saavuttaa. Tämä johtuu siitä, että algoritmin täytyy käydä syöte ainakin kerran läpi, ennen kuin se voi ilmoittaa vastauksen.

$O(n \log n)$ (järjestäminen)

Aikavaativuus $O(n \log n)$ viittaa usein siihen, että algoritmin osana on *järjestämistä*, koska tehokkaat järjestämisalgoritmit toimivat ajassa $O(n \log n)$. Esimerkiksi seuraava $O(n \log n)$ -aikainen algoritmi tarkastaa, onko taulukossa kahta samaa alkia:

```
sort(taulu) // järjestäminen
samat = false
for i = 1 to n-1
    if taulu[i] == taulu[i-1]
        samat = true
```

Algoritmi järjestää ensin taulukon, minkä jälkeen yhtä suuret alkiot ovat vierekkäin ja ne on helppoa löytää. Järjestäminen vie aikaa $O(n \log n)$ ja silmukka vie aikaa $O(n)$, joten algoritmi vie yhteensä aikaa $O(n \log n)$.

Tutustumme tarkemmin järjestämiseen ja sitä käytettäviin algoritmeihin seuraavassa luvussa 3.

$O(n^2)$ (neliöllinen)

Neliöllinen (quadratic) algoritmi voi käydä läpi kaikki tavat valita kaksi alkioita syöttestä. Esimerkiksi seuraava $O(n^2)$ -algoritmi tutkii, onko taulukossa kahta lukua, joiden summa on x .

```
ok = false
for i = 0 to n-1
  for j = i+1 to n-1
    if taulu[i]+taulu[j] == x
      ok = true
```

 $O(n^3)$ (kuutiollinen)

Kuutiollinen (cubic) algoritmi voi käydä läpi kaikki tavat valita kolme alkioita syöttestä. Esimerkiksi seuraava $O(n^3)$ -algoritmi tutkii, onko taulukossa kolmea lukua, joiden summa on x .

```
ok = false
for i = 0 to n-1
  for j = i+1 to n-1
    for k = j+1 to n-1
      if taulu[i]+taulu[j]+taulu[k] == x
        ok = true
```

 $O(2^n)$ (osajoukot)

Aikavaativuus $O(2^n)$ viittaa usein siihen, että algoritmi käy läpi syötteen alkoiden osajoukot.

 $O(n!)$ (permutaatiot)

Aikavaativuus $O(n!)$ viittaa usein siihen, että algoritmi käy läpi syötteen alkoiden permutaatiot.

2.1.3 Tehokkuuden arviointi

Mitä hyötyä on määrittää algoritmin aikavaativuus? Hyötynä on, että aikavaativuus antaa meille arvion siitä, kuinka *hyvä* algoritmi on eli miten suuria syötteitä sillä voi käsitellä tehokkaasti. Kun meille kertyy kokemusta algoritmien suunnittelusta, meille alkaa muodostua selkeä kuva, mitä eri aikavaativuudet tarkoittavat käytännössä.

syötteen kokoluokka n	tarvittava aikavaativuus
10	$O(n!)$
20	$O(2^n)$
500	$O(n^3)$
5000	$O(n^2)$
10^6	$O(n)$ tai $O(n \log n)$
suuri	$O(1)$ tai $O(\log n)$

Taulukko 2.1: Kuinka suuren syötteen algoritmi voi käsitellä nopeasti?

Aikavaativuutta voi ajatella samalla tavalla kuin vaikkapa hotellin tähtiluokitusta: se kertoo tiiviissä muodossa, mistä asiassa on kysymys, eikä meidän tarvitse ottaa selvää yksityiskohdista. Jos meille tarjotaan majoitusta neljän tähden hotellissa, saamme heti jonkin käsityksen huoneen tasosta tähtiluokituksen ansiosta, vaikka emme saisi tarkkaa listausta huoneen varustelusta. Vastaavasti jos kuulemme, että jonkin algoritmin aikavaativuus on $O(n \log n)$, voimme heti arvioida karkeasti, miten suuria syötteitä voimme käsitellä, vaikka emme tuntisi tarkemmin algoritmin toimintaa.

Yksi kiinnostava näkökulma algoritmin tehokkuuteen on, miten suuren syötteen algoritmi voi käsitellä *nopeasti* (alle sekunnissa). Tämä on hyvä vaatimus, kun haluamme käyttää algoritmia jossakin käytännön sovelluksessa. Taulukossa 2.1 on joitakin hyödyllisiä arvioita, kun Java-kielellä toteutettu algoritmi suoritetaan nykyaikaisella tietokoneella. Esimerkiksi jos meillä on $O(n^2)$ -algoritmi, voimme käsitellä sillä nopeasti syötteen, jossa on luokkaa 5000 alkiota. Jos haluamme käsitellä tehokkaasti suurempia syötteitä, meidän tulisi löytää $O(n)$ - tai $O(n \log n)$ -aikainen algoritmi.

Kannattaa silti pitää mielessä, että nämä luvut ovat vain arvioita ja algoritmin todelliseen ajankäyttöön vaikuttavat monet asiat. Saman algoritmin hyvä toteutus saattaa olla kymmeniä kertoja nopeampi kuin huono toteutus, ja suuri merkitys on myös ohjelmointikielellä, jolla algoritmi on toteutettu. Tässä kirjassa analysoimme algoritmeja sekä aikavaativuuksien avulla että mittaamalla todellisia suoritusajoja.

2.1.4 Esimerkki: Merkkijonot

Tehtävän ratkaisemiseen on usein monenlaisia algoritmeja. Seuraavaksi ratkaisemme saman tehtävän kahdella algoritmilla, joista ensimmäinen on suoraviivainen raa'an voiman algoritmi, joka toimii ajassa $O(n^2)$. Toinen algoritmi on puolestaan tehokas algoritmi, joka vie aikaa vain $O(n)$.

Tehtävämme on seuraava: Annettuna on merkkijono, jonka pituus on n ja jokainen merkki on 0 tai 1. Haluamme laskea, monellako tavalla voimme vali-

ta kaksi kohtaa niin, että vasen merkki on 0 ja oikea merkki on 1. Esimerkiksi merkkijonossa 01001 tapoja on neljä: 01001, 01001, 01001 ja 01001.

$O(n^2)$ -algoritmi

Voimme ratkaista tehtävän raa'alla voimalla käymällä läpi kaikki mahdolliset tavat valita vasen ja oikea kohta. Tällöin voimme laskea yksi kerrallaan, monessako tavassa vasen merkki on 0 ja oikea merkki on 1. Seuraava koodi toteuttaa algoritmin:

```
laskuri = 0
for i = 0 to n-1
    for j = i+1 to n-1
        if merkit[i] == 0 and merkit[j] == 1
            laskuri++
print(laskuri)
```

Algoritmin aikavaativuus on $O(n^2)$, koska siinä on kaksi sisäkkäistä silmukkaa, jotka käyvät läpi syötteen.

$O(n)$ -algoritmi

Kuinka voisimme ratkaista tehtävän tehokkaammin? Meidän tulisi keksiä tapa, jolla saisimme pois toisen silmukan koodista.

Tässä auttaa lähestyä ongelmaa hieman toisesta näkökulmasta: kun olemme tiettyssä kohdassa merkkijonoa, monellako tavalla voimme muodostaa parin, jonka oikea merkki on nykyisessä kohdassamme? Jos olemme merkin 0 kohdalla, pareja ei ole yhtään, mutta jos merkinä on 1, voimme valita *minkä tahansa* vasemmalla puolella olevan merkin 0 pariin.

Tämän havainnon ansiosta meidän riittää käydä läpi merkkijono kerran vasemmalta oikealle ja pitää kirjaa, montako merkkiä 0 olemme nähneet. Sittem jokaisen merkin 1 kohdalla kasvatamme vastausta tämänhetkisellä merkkien 0 määrällä. Seuraava koodi toteuttaa algoritmin:

```
laskuri = 0
nollat = 0
for i = 0 to n-1
    if merkit[i] == 0
        nollat++
    else
        laskuri += nollat
print(laskuri)
```

syötteen koko n	$O(n^2)$ -algoritmi	$O(n)$ -algoritmi
10	0.00 s	0.00 s
10^2	0.00 s	0.00 s
10^3	0.00 s	0.00 s
10^4	0.14 s	0.00 s
10^5	1.66 s	0.00 s
10^6	172.52 s	0.01 s

Taulukko 2.2: Algoritmien suoritusaikojen vertailu.

Algoritmissa on vain yksi silmukka, joka käy syötteen läpi, joten sen aikavaativuus on $O(n)$.

Algoritmien vertailua

Meillä on nyt siis kaksi algoritmia, joiden aikavaativuudet ovat $O(n^2)$ ja $O(n)$, mutta mitä tämä tarkoittaa käytännössä? Saamme tämän selville toteuttamalla algoritmit jollakin oikealla ohjelmointikielellä ja mittaamalla niiden suoritusaikoja erikokoisilla syötteillä.

Taulukko 2.2 näyttää vertailun tulokset, kun algoritmit on toteutettu Javalla ja syöteinä on satunnaisia merkkijonoja. Pienillä n :n arvoilla molemmat algoritmit toimivat hyvin tehokkaasti, mutta suuremmilla syötteillä on nähtävissä huomattavia eroja. Raakaan voimaan perustuva $O(n^2)$ -algoritmi alkaa hidastua selvästi testistä $n = 10^4$ alkaen, ja testissä $n = 10^6$ sillä kuluu jo lähes kolme minuuttia aikaa. Tehokas $O(n)$ -algoritmi taas selvittää suuretkin testit salamannopeasti.

Tämän kurssin jatkuvana teemana on luoda algoritmeja, jotka toimivat tehokkaasti myös silloin, kun niille annetaan suuria syötteitä. Tämä tarkoittaa käytännössä sitä, että algoritmin aikavaativuuden tulisi olla $O(n)$ tai $O(n \log n)$. Jos algoritmin aikavaativuus on esimerkiksi $O(n^2)$, se on auttamatta liian hidas suurien syötteiden käsittelyyn.

2.2 Lisää algoritmien analysoinnista

Aikavaativuuksissa esiintyvä O -merkintä on yksi monista merkinnöistä, joiden avulla voimme arvioida funktioiden kasvunopeutta. Tutustumme seuraavaksi tarkemmin näihin merkintöihin.

2.2.1 Merkinnät O , Ω ja Θ

Algoritmien analysoinnissa usein esiintyviä merkintöjä ovat:

- *Yläraja*: Funktio $g(n)$ on luokkaa $O(f(n))$, jos on olemassa vakiot c ja n_0 niin, että $g(n) \leq cf(n)$ aina kun $n \geq n_0$.
- *Alaraja*: Funktio $g(n)$ on luokkaa $\Omega(f(n))$, jos on olemassa vakiot c ja n_0 niin, että $g(n) \geq cf(n)$ aina kun $n \geq n_0$.
- *Tarkka arvio*: Funktio $g(n)$ on luokkaa $\Theta(f(n))$, jos se on sekä luokkaa $O(f(n))$ että luokkaa $\Omega(f(n))$.

Vakion c tarkoituksena on, että saamme arvion kasvunopeuden suuruusluokalle välittämättä vakiokertoimista. Vakion n_0 ansiosta meidän riittää tarkastella kasvunopeutta suurilla n :n arvoilla. Voimme myös kirjoittaa $g(n) = O(f(n))$, kun haluamme ilmaista, että funktio $g(n)$ on luokkaa $O(f(n))$, ja vastaavasti Ω - ja Θ -merkinnöissä.

Kun sanomme, että algoritmi toimii ajassa $O(f(n))$, tarkoitamme, että se suorittaa *pahimmassa tapauksessa* $O(f(n))$ askelta. Tämä on yleensä hyvä tapa ilmoittaa algoritmin tehokkuus, koska silloin annamme takuun siitä, että algoritmin ajankäytöllä on tietty yläraja, vaikka syöte olisi valittu mahdollisimman ikävästi algoritmin kannalta.

Tarkastellaan esimerkkinä seuraavaa algoritmia, joka laskee taulukon lukujen summan:

```
summa = 0
for i = 0 to n-1
    summa += taulu[i]
```

Tämä algoritmi toimii samalla tavalla riippumatta taulukon sisällöstä, koska se käy aina läpi koko taulukon. Niinpä yläraja ajankäytölle on $O(n)$ ja alaraja ajankäytölle on samoin $\Omega(n)$, joten voimme sanoa, että algoritmi vie aikaa $\Theta(n)$ kaikissa tapauksissa.

Tarkastellaan sitten seuraavaa algoritmia, joka selvittää, onko taulukossa lukua x :

```
ok = false
for i = 0 to n-1
    if taulu[i] == x
        ok = true
        break
```

Tässä algoritmin pahin ja paras tapaus eroavat. Ajankäytön yläraja on $O(n)$, koska algoritmi joutuu käymään läpi kaikki taulukon alkioit silloin, kun luku x ei esiinny taulukossa. Toisaalta ajankäytön alaraja on $\Omega(1)$, koska jos luku x on taulukon ensimmäinen alkio, algoritmi pysähtyy heti taulukon

alussa. Voimme myös sanoa, että algoritmi suorittaa pahimmassa tapauksessa $\Theta(n)$ askelta ja parhaassa tapauksessa $\Theta(1)$ askelta.

Huomaa, että O -merkinnän antama yläraja voi olla *mikä tahansa* yläraja, ei välttämättä tarkka yläraja. On siis oikein sanoa esimerkiksi, että äskeinen algoritmi vie aikaa $O(n^2)$, vaikka on olemassa parempi yläraja $O(n)$. Miksi sitten käytämme O -merkintää, vaikka voisimme usein myös ilmaista tarkan ajankäytön Θ -merkinnällä? Tämä on vakiintunut ja käytännössä toimiva tapa. Olisi hyvin harhaanjohtavaa antaa algoritmille yläraja $O(n^2)$, jos näemme suoraan, että aikaa kuluu vain $O(n)$.

Asiaa voi ajatella niin, että O -merkintää käytetään algoritmin *markkinoinnissa*. Jos annamme liian suuren ylärajan, algoritmista tulee väärä käsitys yleisölle. Vertauksena jos myymme urheiluautoa, jonka huippunopeus on 250 km/h, on sinänsä paikkansa pitävä väite, että autolla pystyy ajamaan 100 km/h. Meidän ei kuitenkaan kannata antaa tällaista vähättelevää tietoa, vaan kertoa, että autolla pystyy ajamaan 250 km/h.

2.2.2 Tilavaativuus

Merkintöjä O , Ω ja Θ voi käyttää kaikenlaisissa yhteyksissä, ei vain algoritmin ajankäytön arvioinnissa. Esimerkiksi voimme sanoa, että algoritmi suorittaa silmukkaa $O(\log n)$ kierrosta tai että taulukossa on $O(n^2)$ lukua.

Aikavaativuuden lisäksi kiinnostava tieto algoritmista voi olla sen *tilavaativuus* (*space complexity*). Tämä kuvaa sitä, miten paljon algoritmi käyttää muistia syötteen *lisäksi*. Jos tilavaativuus on $O(1)$, algoritmi tarvitsee muistia vain yksittäisille muuttujille. Jos tilavaativuus on $O(n)$, algoritmi voi varata esimerkiksi aputaulukon, jonka koko vastaa syötteen kokoa.

Tarkastellaan esimerkkinä tehtävää, jossa taulukossa on luvut $1, 2, \dots, n$ yhtä lukuun ottamatta, ja tehtävämme on selvittää puuttuva luku. Yksi tapa ratkaista tehtävä $O(n)$ -ajassa on luoda aputaulukko, joka pitää kirjaa mukana olevista luvuista. Tällaisen ratkaisun tilavaativuus on $O(n)$, koska aputaulukko vie $O(n)$ muistia.

```
for i = 0 to n-2
    mukana[taulu[i]] = true
for i = 1 to n
    if not mukana[i]
        puuttuva = i
```

Tehtävään on kuitenkin olemassa myös toinen algoritmi, jossa aikavaativuus on edelleen $O(n)$ mutta tilavaativuus on vain $O(1)$. Tällainen algoritmi laskee ensin lukujen $1, 2, \dots, n$ summan ja vähentää sitten taulukossa esiin-

tyvät luvut siitä. Jäljelle jäävä luku on puuttuva luku.

```
summa = 0
for i = 1 to n
    summa += i
for i = 0 to n-2
    summa -= taulu[i]
puuttuva = summa
```

Käytännössä tilavaativuus on yleensä sivuroolissa algoritmeissa, koska jos algoritmi vie vain vähän aikaa, se ei *ehdi* käyttää kovin paljon muistia. Eri-tyisesti tilavaativuus ei voi olla suurempi kuin aikavaativuus. Niinpä meidän riittää tavallisesti keskittyä suunnittelemaan algoritmeja, jotka toimivat nopeasti, ja vertailla algoritmien aikavaativuuksia.

2.2.3 Rajojen todistaminen

Jos haluamme todistaa täsmällisesti, että jokin raja pätee, meidän täytyy löytää vakiot c ja n_0 , jotka osoittavat asian. Jos taas haluamme todistaa, että raja ei päde, meidän täytyy näyttää, että mikään vakioiden c ja n_0 valinta ei ole kelvollinen.

Jos haluamme todistaa rajan pätemisen, tämä onnistuu yleensä helposti valitsemalla vakio c tarpeeksi suureksi ja arvioimalla summan osia ylöspäin tarvittaessa. Esimerkiksi jos haluamme todistaa, että $3n + 5 = O(n)$, meidän tulee löytää vakiot c ja n_0 , joille pätee, että $3n + 5 \leq cn$ aina kun $n \geq n_0$. Tässä tapauksessa voimme valita esimerkiksi $c = 8$ ja $n_0 = 1$, jolloin voimme arvioida $3n + 5 \leq 3n + 5n = 8n$, kun $n \geq 1$.

Jos haluamme todistaa, että raja ei päde, tilanne on hankalampi, koska meidän täytyy näyttää, että ei ole olemassa *mitään* kelvollista tapaa valita vakioita c ja n_0 . Tässä auttaa tyypillisesti vastaoletuksen tekeminen: oletamme, että raja pätee ja voimme valita vakiot, ja näytämme sitten, että tämä oletus johtaa ristiriitaan.

Todistetaan esimerkkinä, että $n^2 \neq O(n)$. Jos pätsi $n^2 = O(n)$, niin olisi olemassa vakiot c ja n_0 , joille $n^2 \leq cn$ aina kun $n \geq n_0$. Voimme kuitenkin osoittaa, että tämä aiheuttaa ristiriidan. Jos $n^2 \leq cn$, niin voimme jakaa epäyhtälön molemmat puolet n :llä ja saamme $n \leq c$. Tämä tarkoittaa, että n on aina enintään yhtä suuri kuin vakio c . Tämä ei ole kuitenkaan mahdollista, koska n voi olla miten suuri tahansa, joten ei voi päteä $n^2 = O(n)$.

Määritelmistä lähtevä todistaminen on sinänsä mukavaa ajanvietettä, mutta sille on äärimmäisen harvoin tarvetta käytännössä, kun haluamme tutkia algoritmien tehokkuutta. Voimme koko kurssin ajan huoletta päätellä

algoritmin aikavaativuuden katsomalla, mikä sen rakenne on, kuten olemme tehneet tämän luvun alkuosassa.

Luku 3

Järjestäminen

Järjestäminen (*sorting*) on keskeinen algoritmiikan ongelma, jossa tehtävänä on järjestää n alkioita sisältävä taulukko suuruusjärjestykseen. Esimerkiksi jos meillä on taulukko $[5, 2, 4, 2, 6, 1]$ ja järjestämme sen alkiot pienimmästä suurimpaan, tuloksena on taulukko $[1, 2, 2, 4, 5, 6]$.

Tavoitteemme on toteuttaa järjestäminen *tehokkaasti*. On helppoa järjestää taulukko ajassa $O(n^2)$, mutta tämä on liian hidasta suurella taulukolla. Tässä luvussa opimme kaksi tehokasta järjestämisalgoritmia, jotka vievät aikaa vain $O(n \log n)$. Toisaalta osoittautuu, että ei ole olemassa yleistä järjestämisalgoritmia, joka toimisi nopeammin kuin $O(n \log n)$.

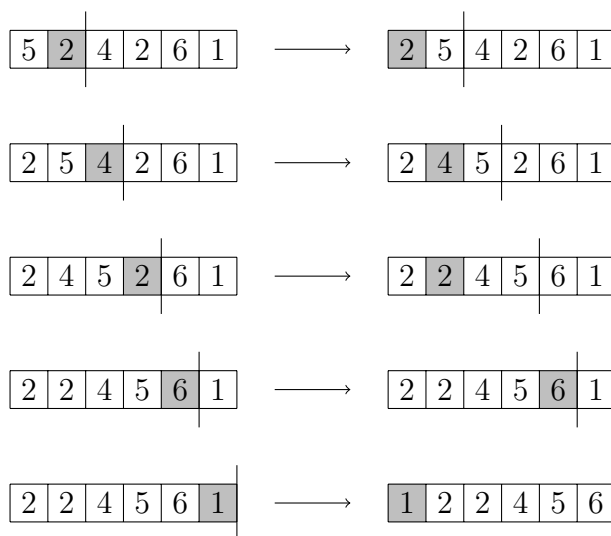
Voimme käyttää järjestämistä monella tavalla algoritmien suunnittelussa. Usein voimme helpottaa ongelman ratkaisemista järjestämällä ensin aineiston. Esimerkiksi jos haluamme tutkia, mikä alkio esiintyy useimmiten taulukossa, voimme järjestää ensin taulukon, jolloin yhtä suuret alkiot päätyvät vierekkäin. Tämän jälkeen meidän riittää käydä taulukko läpi ja etsiä siitä pisin samaa alkioita toistava osuus.

3.1 Järjestäminen ajassa $O(n^2)$

Tutustumme aluksi yksinkertaiseen järjestämisalgoritmiin, joka järjestää n -alkioisen taulukon ajassa $O(n^2)$ kahden silmukan avulla. Vaikka algoritmi ei ole nopea, se on tutustumisen arvoinen ja antaa hyvän lähtökohdan tehokkaampien algoritmien suunnittelemiselle.

3.1.1 Lisäysjärjestäminen

Lisäysjärjestäminen (*insertion sort*) käy läpi taulukon vasemmalta oikealle. Kun algoritmi tulee tiettyyn taulukon kohtaan, se siirtää kyseisessä kohdassa



Kuva 3.1: Lisäysjärjestäminen taulukolle $[5, 2, 4, 2, 6, 1]$.

olevan alkion oikeaan paikkaan taulukon alkuosassa niin, että taulukon alkuosa on tämän jälkeen järjestyksessä. Niinpä kun algoritmi pääsee taulukon loppuun, koko taulukko on järjestyksessä. Toisin sanoen algoritmilla on voimassa seuraava *invariantti*: kun algoritmi on käsitellyt taulukon kohdan i , taulukon alkuosa kohtaan i asti sisältää samat alkiot kuin ennenkin mutta ne ovat järjestyksessä. Invariantti on väittämä, joka pätee jokaisessa algoritmin vaiheessa ja voi auttaa perustelemaan, miksi algoritmi toimii oikein.

Kuva 3.1 näyttää esimerkin lisäysjärjestämisen toiminnasta, kun järjestettävänä on taulukko $[5, 2, 4, 2, 6, 1]$. Jokaisella rivillä algoritmi siirtää harmaataustaisen alkion sen oikealle paikalle taulukon alkuosassa. Pystyviiva ilmaisee kohdan, johon asti taulukko on järjestyksessä siirron jälkeen. Algoritmin päätteeksi tuloksena on järjestetty taulukko $[1, 2, 2, 4, 5, 6]$.

Seuraava koodi toteuttaa lisäysjärjestämisen:

```

for i = 1 to n-1
  j = i-1
  while j >= 0 and taulu[j] > taulu[j+1]
    swap(taulu[j], taulu[j+1])
    j--

```

Koodi käy läpi taulukon kohdat $1 \dots n-1$ ja siirtää aina kohdassa i olevan alkion oikeaan paikkaan. Tämä tapahtuu sisäsilmutalla, jonka jokainen askel vaihtaa keskenään alkion ja sen vasemmalla puolella olevan alkion. Tässä komento `swap` ilmaisee, että alkiot vaihdetaan keskenään.

Lisäysjärjestämisen tehokkuus riippuu siitä, mikä on järjestettävän taulukon sisältö. Algoritmi toimii sitä paremmin, mitä lähempänä järjestystä taulukko on valmiiksi. Jos taulukko on järjestyksessä, aikaa kuluu vain $O(n)$, koska ei tarvitse siirtää mitään alkioita. Pahin tapaus algoritmille on kuitenkin, että taulukko on *käänteisessä* järjestyksessä, jolloin jokainen alkio täytyy siirtää taulukon alkuun ja aikaa kuluu $O(n^2)$.

3.1.2 Inversiot

Hyödyllinen käsite järjestämisalgoritmien analysoinnissa on *inversio*: kaksi taulukossa olevaa alkioita, jotka ovat väärässä järjestyksessä. Esimerkiksi taulukossa $[3, 1, 4, 2]$ on kolme inversiota: $(3, 1)$, $(3, 2)$ ja $(4, 2)$. Inversioiden määrä kertoo taulukon järjestyksestä: mitä vähemmän inversioita taulukossa on, sitä lähempänä se on järjestystä. Erityisesti taulukko on järjestyksessä tarkalleen silloin, kun siinä ei ole yhtään inversiota.

Kun järjestämisalgoritmi järjestää taulukon, se *poistaa* siitä inversioita. Tarkemmin sanoen aina kun algoritmi vaihtaa kaksi alkioita keskenään, se poistaa taulukosta yhden inversion. Niinpä lisäysjärjestämisen työmäärä on yhtä suuri kuin järjestettävän taulukon inversioiden määrä.

Olemme jo todenneet, että pahin mahdollinen syöte lisäysjärjestämiselle on käänteisessä järjestyksessä oleva taulukko. Tällaisessa taulukossa jokainen alkio pari muodostaa inversion, joten inversioiden määrä on

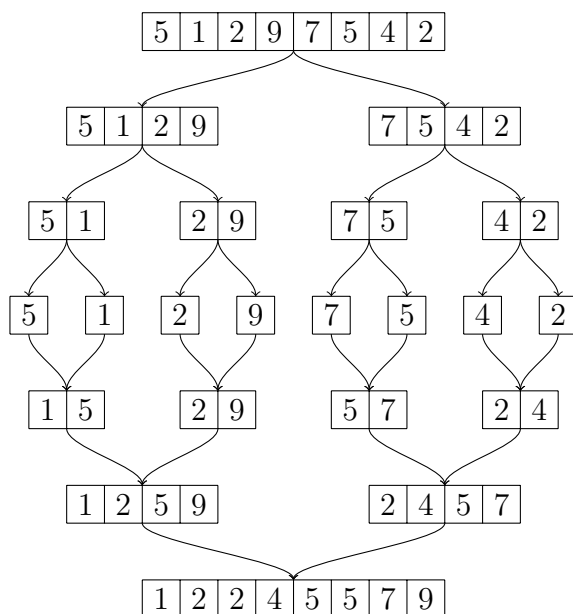
$$\frac{n(n-1)}{2} = O(n^2).$$

Entä kuinka hyvin lisäysjärjestäminen toimii *keskimäärin*? Jos oletamme, että taulukossa on n eri alkioita satunnaisessa järjestyksessä, jokainen taulukossa oleva alkio pari muodostaa inversion todennäköisyydellä $1/2$. Niinpä inversioiden määrän *odotusarvo* on

$$\frac{n(n-1)}{4} = O(n^2),$$

eli aikaa kuluu neliöllinen määrä myös keskimääräisessä tapauksessa.

Syy lisäysjärjestämisen hitauteen on siis, että se ei poista taulukosta inversioita riittävän tehokkaasti. Jos haluamme kehittää paremman järjestämisalgoritmin, meidän täytyy suunnitella se niin, että se voi poistaa useita inversioita *yhtä aikaa*. Käytännössä algoritmin täytyy pystyä siirtämään väärässä paikassa oleva alkio tehokkaasti taulukon toiselle puolelle.



Kuva 3.2: Lomitusjärjestäminen taulukolle $[5, 1, 2, 9, 7, 5, 4, 2]$.

3.2 Järjestäminen ajassa $O(n \log n)$

Seuraavaksi tutustumme kahteen tehokkaaseen järjestämisalgoritmiin, jotka perustuvat rekursioon. Molemmista algoritmeista on ideana, että kun haluamme järjestää taulukon, jaamme sen kahteen pienempään osaan ja järjestämme ne rekursiivisesti. Tämän jälkeen yhdistämme järjestetyt osataulukot kokonaiseksi järjestetyksi taulukoksi.

3.2.1 Lomitusjärjestäminen

Lomitusjärjestäminen (*merge sort*) on rekursiivinen järjestämisalgoritmi, joka perustuu taulukon puolituksiin. Kun saamme järjestettäväksi n -kokoisen taulukon, jaamme sen keskeltä kahdeksi osataulukoksi, joissa molemmissa on noin $n/2$ alkia. Tämän jälkeen järjestämme osataulukot erikseen rekursiivisesti ja *lomitamme* sitten järjestetyt osataulukot niin, että niistä muodostuu kokonainen järjestetty taulukko. Rekursio päättyy tapaukseen $n = 1$, jolloin taulukko on valmiiksi järjestyksessä eikä tarvitse tehdä mitään.

Seuraava koodi esittää tarkemmin lomitusjärjestämisen toiminnan:

```
procedure jarjesta(a,b)
  if a == b
    return
  k = (a+b)/2
  jarjesta(a,k)
  jarjesta(k+1,b)
  lomita(a,k,k+1,b)
```

Proseduuri `jarjesta` järjestää taulukon välin $a \dots b$ (osataulukon kohdasta a kohtaan b), eli kun haluamme järjestää koko taulukon, kutsumme proseduuria parametreilla $a = 0$ ja $b = n - 1$. Proseduuri tarkastaa ensin, onko osataulukossa vain yksi alkio, ja jos on, se päättyy heti. Muuten se laskee muuttujaan k järjestettävän välin keskikohdan ja järjestää vasemman ja oikean puoliskon rekursiivisesti. Lopuksi se kutsuu proseduuria `lomita`, joka yhdistää järjestetyt puoliskot. Tämän proseduurin voi toteuttaa näin:

```
procedure lomita(a1, b1, a2, b2)
  a = a1, b = b2
  for i = a to b
    if a2 > b2 or (a1 <= b1 and taulu[a1] < taulu[a2])
      apu[i] = taulu[a1]
      a1++
    else
      apu[i] = taulu[a2]
      a2++
  for i = a to b
    taulu[i] = apu[i]
```

Parametreina annetaan välit $a_1 \dots b_1$ ja $a_2 \dots b_2$, missä $b_1 + 1 = a_2$. Proseduuri olettaa, että näillä väleillä olevat taulukon alkiot on järjestetty, ja se lomittaa alkiot niin, että taulukon koko väli $a_1 \dots b_2$ on järjestetty. Proseduurin perustana on silmukka, joka käy läpi välejä $a_1 \dots b_1$ ja $a_2 \dots b_2$ rinnakkain ja valitsee aina seuraavaksi pienimmän alkion lopulliseen järjestykseen. Jotta lomitusta ei sotke taulukkoa, proseduuri käyttää globaalia aputaulukkoa, johon se ensin muodostaa järjestetyn osataulukon, ja kopioi sitten alkiot aputaulukosta varsinaiseen taulukkoon.

Kuva 3.2 näyttää, miten lomituserjestäminen toimii, kun sille annetaan taulukko $[5, 1, 2, 9, 7, 5, 4, 2]$. Algoritmi puolittaa ensin taulukon kahdeksi osataulukoksi $[5, 1, 2, 9]$ ja $[7, 5, 4, 2]$ ja järjestää molemmat osataulukot kutsumalla itseään. Kun algoritmi saa sitten järjestettäväksi taulukon $[5, 1, 2, 9]$, se jakaa taulukon edelleen osataulukoiksi $[5, 1]$ ja $[2, 9]$, jne. Lopul-

ta jäljellä on vain yhden alkion kokoisia osataulukoita, jotka ovat valmiiksi järjestyksessä. Tällöin rekursiivinen jakautuminen päättyy ja algoritmi alkaa koota järjestettyjä osataulukkoja pienimmästä suurimpaan.

Nyt voimme määrittää, kuinka tehokas lomitusjärjestäminen on. Koska jokainen proseduurin *jarjesta* kutsu puolittaa taulukon koon, rekursiosta muodostuu $O(\log n)$ tasoa (kuva 3.2). Ylimmällä tasolla on taulukko, jossa on n alkioita, seuraavalla tasolla on kaksi taulukkoa, joissa on $n/2$ alkioita, seuraavalla tasolla on neljä taulukkoa, joissa on $n/4$ alkioita, jne. Proseduurin *lomita* toimii lineaarisessa ajassa, joten kullakin tasolla taulukoiden lomittamiset vievät yhteensä aikaa $O(n)$. Niinpä algoritmin kokonaisaikaavaatavuus on $O(n \log n)$.

3.2.2 Pikajärjestäminen

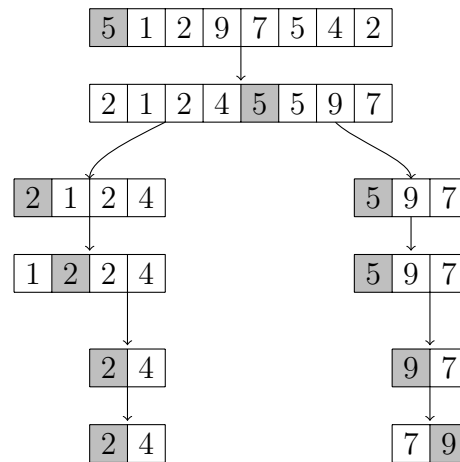
Pikajärjestäminen (*quick sort*) tarjoaa toisenlaisen rekursiivisen lähestymistavan taulukon järjestämiseen. Kun saamme järjestettäväksi taulukon, valitsemme ensin jonkin sen alkioista *jakoalkioksi* (*pivot*). Tämän jälkeen siirrämme alkioita niin, että jakoalkiota pienemmät alkiot ovat sen vasemmalla puolella, suuremmat alkiot ovat sen oikealla puolella ja yhtä suuret alkiot voivat olla kummalla tahansa puolella. Lopuksi järjestämme rekursiivisesti osataulukot, jotka muodostuvat jakoalkion vasemmalle ja oikealle puolelle.

Seuraava koodi esittää pikajärjestämisen toiminnan:

```
procedure jarjesta(a, b)
  if a >= b
    return
  k = jako(a,b)
  jarjesta(a,k-1)
  jarjesta(k+1,b)
```

Proseduuri *jarjesta* järjestää taulukon välillä $a \dots b$ olevat alkiot. Jos väli on tyhjä tai siinä on vain yksi alkio, proseduurin ei tee mitään. Muuten se kutsuu funktiota *jako*, joka valitsee jakoalkion, siirtää taulukon alkioita sen mukaisesti ja palauttaa sitten kohdan k , jossa jakoalkio on siirtojen jälkeen. Tämän jälkeen taulukon vasen osa (väli $a \dots k-1$) ja oikea osa (väli $k+1 \dots b$) järjestetään rekursiivisesti.

Funktion *jako* voi toteuttaa monella tavalla, koska voimme valita minkä tahansa alkion jakoalkioksi ja lisäksi on monia tapoja siirtää alkioita. Käytämme tässä esimerkkinä seuraavaa toteutusta:



Kuva 3.3: Pikajärjestäminen taulukolle $[5, 1, 2, 9, 7, 5, 4, 2]$.

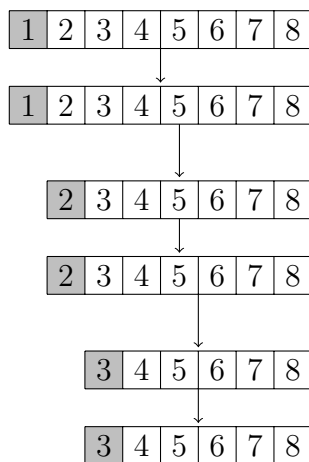
```

function jako(a,b):
    k = a
    for i = a+1 to b
        if taulu[i] < taulu[a]
            k += 1
            swap(taulu[i],taulu[k])
    swap(taulu[a],taulu[k])
    return k
  
```

Tässä jakoalkio on aina välin ensimmäinen alkio, joka on kohdassa a . Funktio käy läpi välin alkiot ja siirtää jakoalkiota pienempiä alkioita taulukon alkuosaan. Muuttuja k määrittää kohdan, johon seuraava pienempi alkio siirretään. Lopuksi jakoalkio itse siirretään keskelle kohtaan k , jonka funktio myös palauttaa.

Kuva 3.3 näyttää, miten pikajärjestäminen toimii, kun sille annetaan taulukko $[5, 1, 2, 9, 7, 5, 4, 2]$. Jokaisessa vaiheessa harmaa tausta osoittaa jakoalkion sijainnin. Aluksi koko taulukon jakoalkio on 5 ja algoritmi siirtää alkioita niin, että jakoalkion vasemmalla puolella ovat alkiot $[2, 1, 2, 4]$ ja oikealla puolella alkiot $[5, 9, 7]$. Tämän jälkeen vasen ja oikea osataulukko järjestetään vastaavasti rekursiivisesti.

Pikajärjestämisen tehokkuuteen vaikuttaa, miten alkiot jakautuvat jakoalkion eri puolille. Jos hyvin käy, jakoalkion kummallekin puolelle siirretään suunnilleen yhtä monta alkioita. Tällöin taulukon koko puolittuu jokaisen jaon jälkeen ja pikajärjestäminen toimii tehokkaasti. Koska funktio `jako` toimii lineaarisessa ajassa, pikajärjestäminen vie tässä tapauksessa ai-



Kuva 3.4: Pikajärjestämisen pahin tapaus: jokaisessa jaossa kaikki alkiot jäävät jakoalkion toiselle puolelle.

kaa $O(n \log n)$ samaan tapaan kuin lomitussjärjestäminen. Uhkana on kuitenkin, että jakoalkio jakaa taulukon osiin *epätasaisesti*. Kuva 3.4 näyttää tilanteen, jossa jokaisessa jaossa kaikki alkiot jäävät jakoalkion oikealle puolelle. Tällöin pikajärjestäminen viekin aikaa $O(n^2)$, koska rekursiivisia tasoja on $O(n)$. Selvästikään *ei* ole kaikissa tilanteissa hyvä tapa valita taulukon ensimmäinen alkio jakoalkioksi. Esimerkiksi valmiiksi järjestyksessä olevan taulukon järjestäminen vie silloin aikaa $O(n^2)$.

Oma lukunsa on tilanne, jossa taulukossa on paljon samoja alkioita. Ääritapauksena *jokainen* alkio on sama, mikä on käytännössä hyvin mahdollinen syöte. Tällöin tässä esitetty funktio `jako` tuottaa aina huonon jaon, jossa kaikki alkiot menevät jakoalkion oikealle puolelle. Ei siis riitä, että jakoalkio on valittu hyvin, vaan myös tapa, jolla alkioita siirretään taulukossa, vaikuttaa algoritmin tehokkuuteen.

Miten meidän tulisi sitten toteuttaa pikajärjestäminen? Algoritmista on kehitetty vuosien aikana suuri määrä muunnelmia, jotka pyrkivät parantamaan sen toimintaa eri tilanteissa. Yksi kehittyneempi tapa valita jakoalkio on ottaa tarkasteluun taulukon ensimmäinen, keskimäinen ja viimeinen alkio ja valita jakoalkioksi järjestyksessä keskimäinen näistä kolmesta alkios-ta. Tällainen valinta toimii käytännössä hyvin monissa tilanteissa. Parempi tapa toteuttaa alkioden siirtäminen on puolestaan käydä läpi rinnakkain alkioita alusta ja lopusta ja pitää huoli siitä, että jakokohta jää keskelle, jos kaikki alkiot ovat yhtä suuria. Pikajärjestämisen toteuttaminen hyvin ei ole helppo tehtävä, vaan vaatii paljon huolellisuutta.

3.2.3 Algoritmien vertailua

Meillä on nyt siis kaksi rekursiivista järjestämisalgoritmia: lomitusjärjestäminen toimii *aina* ajassa $O(n \log n)$, kun taas pikajärjestäminen toimii *ehkä* ajassa $O(n \log n)$, mutta saattaa viedä aikaa $O(n^2)$. Vaatii monenlaista virittelyä, ennen kuin pikajärjestämisen saa toimimaan tehokkaasti edes tapauksissa, joissa taulukko on valmiiksi järjestyksessä tai kaikki alkiot ovat samoja. Miksi haluaisimme koskaan käyttää epävarmaa pikajärjestämistä, kun voimme käyttää myös varmasti tehokasta lomitusjärjestämistä?

Syynä on, että pikajärjestämisen *vakiokertoimet* ovat pienet. Kokemus on osoittanut, että kun toteutamme lomitusjärjestämisen ja pikajärjestämisen ja mittaamme algoritmien todellisia suoritusajoja, pikajärjestäminen toimii usein nopeammin. Näin tapahtuu siitä huolimatta, että pikajärjestämisen pahimman tapauksen aikavaativuus on $O(n^2)$. Käytännössä pahin tapaus on kuitenkin harvinainen, jos jakoalkion valinta ja alkioden siirtäminen on toteutettu huolellisesti.

Jos taulukko on pieni, $O(n^2)$ -aikainen lisäysjärjestäminen toimii käytännössä nopeammin kuin rekursiiviset algoritmit, koska sen vakiokertoimet ovat hyvin pienet. Yksi mahdollisuus onkin toteuttaa *hybridialgoritmi*, jossa suuret taulukot järjestetään rekursiivisesti ja pienet taulukot järjestetään lisäysjärjestämisellä. Tällöin eri algoritmien hyvät puolet pääsevät osaksi kokonaisalgoritmia. Käytännössä hybridialgoritmin voi tehdä niin, että valitaan jokin sopiva raja k ja taulukko järjestetään rekursiivisesti, jos siinä on ainakin k alkioita, ja muuten lisäysjärjestämisellä.

3.3 Järjestämisen alaraja

Olisiko mahdollista luoda järjestämisalgoritmi, joka toimisi nopeammin kuin $O(n \log n)$? Osoittautuu, että tämä *ei* ole mahdollista, jos oletamme, että algoritmin tulee perustua taulukon alkioden vertailuihin. Vertailuihin perustuva järjestämisalgoritmi järjestää taulukon tekemällä joukon vertailuja muotoa ”onko alkio x suurempi kuin alkio y ?”.

Vertailuihin perustuva järjestämisalgoritmi on *yleiskäyttöinen*: se pystyy järjestämään mitä tahansa alkioita, kunhan meillä on keino saada selville kahden alkion suuruusjärjestys. Tämä on ominaisuus, jota yleensä ottaen toivomme järjestämisalgoritmilta, joten vertailuihin perustuminen on luonteva rajoitus. Kaikki tähän mennessä käsittelemämme järjestämisalgoritmit ovat olleet vertailuihin perustuvia

3.3.1 Alarajatodistus

Voimme ajatella vertailuihin perustuvaa järjestämistä *prosessina*, jossa jokainen vertailu antaa meille tietoa taulukosta ja auttaa meitä viemään taulukkoa lähemmäs järjestystä. Oletamme seuraavaksi, että taulukko muodostuu alkioista $1, 2, \dots, n$, jolloin meillä on $n!$ vaihtoehtoa, mikä on taulukon alkuperäinen järjestys. Jotta järjestämisalgoritmi voisi toimia oikein, sen täytyy käsitellä jokainen järjestys eri tavalla.

Esimerkiksi jos $n = 3$, taulukon mahdolliset järjestykset alussa ovat $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$ ja $[3, 2, 1]$. Algoritmi voi vertailla ensin vaikkapa ensimmäistä ja toista alkioita. Jos ensimmäinen alkio on pienempi, voimme päätellä, että mahdolliset taulukot ovat $[1, 2, 3]$, $[1, 3, 2]$ ja $[2, 3, 1]$. Jos taas ensimmäinen alkio on suurempi, mahdolliset taulukot ovat $[2, 1, 3]$, $[3, 1, 2]$ ja $[3, 2, 1]$. Tämän jälkeen voimme jatkaa vertailuja samaan tapaan ja saada lisää tietoa taulukosta. Algoritmi voi päättyä vasta silloin, kun jäljellä on vain yksi mahdollinen taulukko, jotta voimme olla varmoja, että olemme järjestäneet taulukon oikein.

Tärkeä seikka on, että jokaisessa vertailussa ainakin puolet jäljellä olevista taulukoista voi täsmätä vertailun tulokseen. Niinpä jos algoritmilla käy huono tuuri, se voi enintään puolittaa taulukoiden määrän joka askeleella. Tämä tarkoittaa, että algoritmi joutuu tekemään pahimmassa tapauksessa ainakin $\log(n!)$ vertailua. Logaritmien laskusääntöjen perusteella

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

Saamme tälle summalle alarajan ottamalla huomioon vain $n/2$ viimeistä termiä ja arvioimalla niitä alaspäin niin, että jokaisen termin suuruus on vain $\log(n/2)$. Tuloksena on alaraja

$$\log(n!) \geq (n/2) \log(n/2),$$

mikä tarkoittaa, että algoritmi joutuu tekemään pahimmassa tapauksessa $\Omega(n \log n)$ vertailua.

3.3.2 Laskemisjärjestäminen

Millainen olisi sitten järjestämisalgoritmi, joka ei perustu vertailuihin ja toimii tehokkaammin kuin $O(n \log n)$? *Laskemisjärjestäminen* (*counting sort*) on $O(n)$ -aikainen järjestämisalgoritmi, jonka toiminta perustuu oletukseen, että taulukon alkioita ovat sopivan pieniä kokonaislukuja. Algoritmi olettaa, että jokainen alkio on kokonaisluku välillä $0 \dots k$, missä $k = O(n)$.

Algoritmi luo *kirjanpidon*, joka kertoo, montako kertaa mikäkin mahdollinen luku välillä $0 \dots k$ esiintyy taulukossa. Seuraavassa koodissa kirjanpito

tallennetaan taulukkoon `laskuri` niin, että `laskuri[x]` ilmaisee, montako kertaa luku x esiintyy taulukossa. Tämän kirjanpidon avulla voimme luoda suoraan lopullisen järjestetyn taulukon.

```
for i = 0 to n-1
    laskuri[taulu[i]]++
i = 0
for x = 0 to k
    for j = 1 to laskuri[x]
        taulu[i] = x
        i++
```

Algoritmin molemmat vaiheet vievät aikaa $O(n)$, joten se toimii ajassa $O(n)$ ja on käytännössä hyvin tehokas. Algoritmi ei ole kuitenkaan yleinen järjestämisalgoritmi, koska sitä voi käyttää vain silloin, kun taulukon kaikki alkiot ovat sopivan pieniä kokonaislukuja.

3.4 Järjestäminen Javassa

Vaikka on hyödyllistä tuntea järjestämisen teoriaa, käytännössä ei ole hyvä idea toteuttaa itse järjestämisalgoritmia, koska nykypäivän ohjelmointikielessä on valmiit työkalut järjestämiseen. Valmiin algoritmin käyttämisessä on etuna, että se on varmasti hyvin toteutettu ja tehokas. Lisäksi meiltä säästyy aikaa, kun emme joudu toteuttamaan algoritmia itse.

Javan standardikirjasto sisältää metodin `Arrays.sort`, joka järjestää sille annetun taulukon. Esimerkiksi seuraava koodi järjestää kokonaislukuja sisältävän taulukon:

```
int[] taulu = {4,2,5,8,2,1,5,6};
Arrays.sort(taulu);
```

Kiinnostava kysymys on, mitä algoritmia Java käyttää taulukon järjestämiseen. Asian voi tarkastaa Javan standardikirjaston dokumentaatiosta. Yllättävää kyllä, Javan käyttämä algoritmi riippuu siitä, minkä *tyyppistä* tietoa taulukossa on. Jos taulukon alkiot ovat alkeistyyppisiä (esimerkiksi `int`), Java käyttää pikajärjestämisen muunnelmaa, jossa on kaksi jakoalkiota. Jos taas alkiot ovat oliotyyppisiä (esimerkiksi `String`), algoritmina on optimoitu lomitusjärjestäminen.

Kun `Arrays.sort` järjestää olioita sisältävän taulukon, se kutsuu metodia `compareTo` aina, kun se haluaa selvittää kahden alkion suuruusjärjestyksen. Metodien tulee palauttaa negatiivinen arvo, nolla tai positiivinen arvo sen

mukaan, onko olio itse pienempi, yhtä suuri vai suurempi kuin parametrina annettu olio. Javan omissa luokissa tällainen metodi on olemassa valmiina. Esimerkiksi voimme tutkia merkkijonojen suuruusjärjestystä näin:

```
System.out.println("apina".compareTo("banaani")); // -1
System.out.println("banaani".compareTo("apina")); // 1
System.out.println("apina".compareTo("apina")); // 0
```

Tämän ansiosta voimme järjestää suoraan taulukoita, joissa on esimerkiksi merkkijonoja. Jos haluamme, että Java pystyy järjestämään omia olioitamme, meidän täytyy toteuttaa itse luokkaan metodi `compareTo` ja merkitä, että luokka toteuttaa rajapinnan `Comparable`. Esimerkiksi seuraava koodi toteuttaa luokan `Piste`, johon voidaan tallentaa pisteen x- ja y-koordinaatit. Luokassa on metodi `compareTo`, joka määrittelee, että pisteet järjestetään ensisijaisesti x-koordinaatin ja toissijaisesti y-koordinaatin mukaan.

```
public class Piste implements Comparable<Piste> {
    public int x, y;

    public int compareTo(Piste p) {
        if (this.x != p.x) {
            return this.x-p.x;
        } else {
            return this.y-p.y;
        }
    }
}
```

Metodin `compareTo` avulla voimme myös konkreettisesti tarkastella, mitä Java tekee järjestäessään taulukon. Seuraava luokka sisältää vain yhden luvun, mutta ilmoittaa meille aina, kun Java kutsuu `compareTo`-metodia:

```
public class Luku implements Comparable<Luku> {
    public int luku;

    public int compareTo(Luku x) {
        System.out.println("vertailu: " + luku + " " + x.luku);
        return this.luku-x.luku;
    }
}
```

Esimerkiksi kun järjestettävänä taulukkona on `[4, 1, 3, 2]`, saamme tietää, että Java tekee seuraavat vertailut:

```
vertailu: 1 4
vertailu: 3 1
vertailu: 3 4
vertailu: 3 1
vertailu: 2 3
vertailu: 2 1
```

3.5 Järjestämisen sovelluksia

Järjestämisen merkitys algoritmikassa on siinä, että voimme ratkaista monia ongelmia tehokkaasti, kunhan aineisto on järjestyksessä. Niinpä yleinen tapa luoda tehokas algoritmi on järjestää ensin syöte ajassa $O(n \log n)$ ja hyödyntää sitten tavalla tai toisella järjestystä algoritmin loppuosassa.

3.5.1 Taulukkoalgoritmeja

Järjestämisen avulla voimme ratkaista ajassa $O(n \log n)$ monia taulukoihin liittyviä tehtäviä. Käymme seuraavaksi läpi kaksi tällaista tehtävää.

Ensimmäinen tehtävämme on laskea, montako *eri* alkia annettussa taulukossa on. Esimerkiksi taulukko $[2, 1, 4, 2, 4, 2]$ sisältää kolme eri alkia: 1, 2 ja 4. Voimme ratkaista tehtävän järjestämällä ensin taulukon, minkä jälkeen yhtä suuret alkut ovat vierekkäin. Tämän jälkeen saamme laskettua vastauksen helposti, koska riittää tutkia, monessako taulukon kohdassa on *vierekkäin* kaksi eri alkia. Voimme toteuttaa algoritmin näin:

```
sort(taulu)
laskuri = 1
for i = 1 to n-1
    if taulu[i-1] != taulu[i]
        laskuri++
print(laskuri)
```

Algoritmi järjestää ensin taulukon ajassa $O(n \log n)$, minkä jälkeen se käy läpi taulukon sisällön for-silmukalla ajassa $O(n)$. Tämän ansiosta algoritmi vie aikaa yhteensä $O(n \log n)$.

Entä jos haluammekin selvittää, mikä on taulukon *yleisin* alkio? Esimerkiksi taulukon $[2, 1, 4, 2, 4, 2]$ yleisin alkio on 2, joka esiintyy kolme kertaa taulukossa. Tämäkin tehtävä ratkeaa järjestämisen avulla, koska järjestämisen jälkeen yhtä suuret alkut ovat peräkkäin ja meidän riittää etsiä pisin samaa alkia toistava osuus. Voimme toteuttaa algoritmin seuraavasti:

```
sort(taulu)
maara = 1
suurin = 1
yleisin = taulu[0]
for i = 1 to n-1
    if taulu[i-1] != taulu[i]
        maara = 0
    maara++
    if maara > suurin
        suurin = maara
        yleisin = taulu[i]
print(yleisin)
```

Tässäkin algoritmissa järjestäminen vie aikaa $O(n \log n)$ ja for-silmukka vie aikaa $O(n)$, joten algoritmin kokonaisaikaavaativuus on $O(n \log n)$.

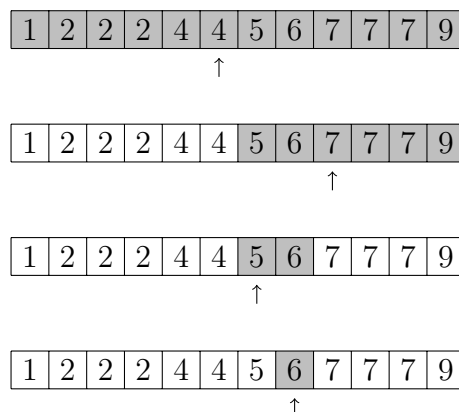
3.5.2 Binäärihaku

Binäärihaku (*binary search*) on menetelmä, jonka avulla voimme löytää alkion järjestetystä taulukosta ajassa $O(\log n)$. Ideana on pitää yllä hakuväliä, jossa etsittävä alkio voi olla, ja puolittaa väli joka askeleella tutkimalla välin keskimmäisenä olevaa alkioita. Koska taulukko on järjestyksessä, voimme aina päätellä, kumpaan suuntaan hakua tulee jatkaa.

Seuraava koodi etsii binäärihaulla taulukosta alkioita x :

```
a = 0
b = n-1
while a <= b
    k = (a+b)/2
    if taulu[k] == x
        // alkio löytyi
        break
    if taulu[k] < x
        a = k+1
    if taulu[k] > x
        b = k-1
```

Algoritmi pitää yllä hakuväliä $[a, b]$, joka on aluksi $[0, n - 1]$, koska alkio x saattaa olla missä tahansa kohdassa taulukossa. Joka askeleella algoritmi tarkastaa välin keskellä kohdassa $k = \lfloor (a + b)/2 \rfloor$ olevan alkion. Jos kyseinen alkio on x , haku päättyy. Jos taas alkio on pienempi kuin x , haku jatkuu välin oikeaan puoliskoon, ja jos alkio on suurempi kuin x , haku jatkuu välin va-



Kuva 3.5: Luvun 6 etsiminen järjestetystä taulukosta binäärihaun avulla. Harmaa alue vastaa väliä $[a, b]$ ja nuoli osoittaa kohdan k .

sempaan puoliskoon. Koska välin koko puolittuu joka askeleella, binäärihaun aikavaativuus on $O(\log n)$.

Kuva 3.5 näyttää esimerkin binäärihaun toiminnasta, kun etsimme lukua 6 järjestetystä taulukosta. Aluksi hakuvälinä on koko taulukko ja puolivälin kohdalla on luku 4. Niinpä voimme päätellä, että jos taulukossa on luku 6, sen täytyy esiintyä taulukon loppuosassa. Tämän jälkeen hakuvälinä on taulukon loppuosa ja puolivälin kohdalla on luku 7. Nyt voimme taas päätellä, että luvun 6 täytyy olla tämän kohdan vasemmalla puolella. Hakuväli puolittuu joka askeleella, ja kun jatkamme vastaavasti, kahden askeleen kuluttua hakuvälillä on vain yksi luku, joka on juuri haettu luku 6. Olemme löytäneet halutun luvun ja algoritmi päättyy.

Voimme selvittää binäärihaun avulla esimerkiksi tehokkaasti, onko taulukossa kahta alkia a ja b niin, että $a + b = x$, missä x on annettu arvo. Ideana on järjestää ensin taulukko ja käydä sitten läpi kaikki taulukon luvut. Jokaisen luvun kohdalla tutkimme, voisiko kyseinen luku olla a . Tällöin taulukossa pitäisi olla toinen luku b niin, että $a + b = x$, eli taulukossa pitäisi olla luku $x - a$. Pystymme tarkastamaan tämän binäärihaulla ajassa $O(\log n)$. Tuloksena on algoritmi, joka vie aikaa $O(n \log n)$, koska sekä järjestäminen että binäärihakua käyttävä läpikäynti vievät aikaa $O(n \log n)$.

Luku 4

Lista

Lista (*list*) on taulukkoa muistuttava tietorakenne, joka sisältää joukon alkioita, jotka ovat peräkkäin tietyssä järjestyksessä. Esimerkiksi $[3, 7, 2, 5]$ on lista, joka sisältää neljä alkioita. Haluamme toteuttaa listan niin, että pääsemme käsiksi listalla olevaan alkioon sen kohdan perusteella ja lisäksi pystymme lisäämään ja poistamaan alkioita.

Tässä luvussa tutustumme kahteen lähestymistapaan listan luomiseen. Ensin toteutamme taulukkolistan, jossa listan alkiot tallennetaan taulukkoon. Tämän jälkeen toteutamme linkitetyn listan, joka muodostuu toisiinsa viittaavista solmuista. Kuten tulemme huomaamaan, molemmissa listan toteutuksissa on omat hyvät ja huonot puolensa.

4.1 Taulukkolista

Taulukkolista (*array list*) on lista, joka on tallennettu taulukkona. Koska taulukon alkiot sijaitsevat aina peräkkäin muistissa, pääsemme käsiksi mihin tahansa listan alkioon ajassa $O(1)$. Haasteena toteutuksessa on kuitenkin, että taulukon koko on *kiinteä* ja jos haluamme muuttaa listan kokoa, meidän täytyy varata uusi taulukko ja kopioida sinne vanhan taulukon sisältö.

4.1.1 Muutokset lopussa

Toteutamme ensin taulukkolistan, jossa alkioden lisäykset ja poistot tapahtuvat listan lopussa. Tallennamme listan taulukkona niin, että tietty määrä alkioita taulukon alussa on listan käytössä ja loput tyhjät kohdat on varattu tuleville alkiuille. Tämän ansiosta pystymme lisäämään uuden alkion listalle ajassa $O(1)$, jos taulukossa on tilaa, koska meidän riittää vain ottaa käyttöön seuraava vapaana oleva kohta taulukosta.

(a)	3	7	2	5	—	—	—	—
(b)	3	7	2	5	6	—	—	—

Kuva 4.1: (a) Lista $[3, 7, 2, 5]$ tallennettuna taulukkoon. (b) Listan loppuun lisätään alkio 6.

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

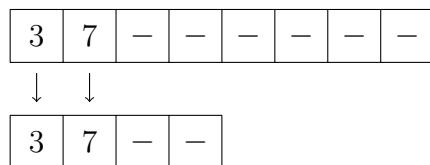
Kuva 4.2: Taulukkoon ei mahdu enää uutta alkioita. Meidän täytyy varata uusi suurempi taulukko ja kopioida vanhan taulukon sisältö sinne.

Kuva 4.1 näyttää esimerkin, jossa taulukossa on tilaa yhteensä kahdeksalle alkioille ja siihen on tallennettu lista $[3, 7, 2, 5]$. Taulukon neljä ensimmäistä kohtaa ovat siis listan käytössä ja muut ovat varalla tulevia alkioita varten. Kun lisäämme listan loppuun uuden alkion 6, otamme käyttöön taulukosta uuden kohdan, johon alkio sijoitetaan.

Mitä tapahtuu sitten, kun jossain vaiheessa koko taulukko on täynnä eikä uusi listalle lisättävä alkio mahdu enää taulukkoon? Tällöin meidän täytyy ensin varata uusi suurempi taulukko ja kopioida kaikki vanhan taulukon alkiot siihen. Vasta tämän jälkeen voimme lisätä uuden alkion listalle. Tämä vie aikaa $O(n)$, koska kopioimme kaikki listan alkiot uuteen paikkaan muistissa. Esimerkiksi kuvassa 4.2 uusi alkio 4 ei mahdu taulukkoon, joten joudumme varaamaan uuden taulukon ja kopioimaan alkiot.

Olemme saaneet siis aikaan listan, jossa lisääminen vie aikaa *joko* $O(1)$ tai $O(n)$ riippuen siitä, mahtuuko alkio nykyiseen taulukkoon vai täytyykö meidän varata uusi taulukko. Jotta lista olisi käyttökelpoinen, hidas $O(n)$ -operaatio ei saisi esiintyä liian usein. Osoittautuu, että saavutamme tämän tavoitteen, kunhan varaamme uuden taulukon aina reilusti aiempaa suuremmaksi. Tavanomainen ratkaisu on *kaksinkertaistaa* taulukon koko aina, kun varaamme uuden taulukon. Kun toimimme näin, jokaisen alkion lisääminen listalle vie *keskimäärin* vain $O(1)$ aikaa.

Miksi aikaa kuluu keskimäärin vain $O(1)$? Tarkastellaan tilannetta, jossa listaan lisätään alkioita ja taulukkoa kasvatetaan viimeisen kerran, kun listassa on n alkioita. Haluamme arvioida, montako alkioita kopioidaan taulukosta toiseen prosessin aikana. Koska taulukon koko kaksinkertaistuu joka vaihees-



Kuva 4.3: Poistojen jälkeen taulukon koko on käynyt tarpeettoman suureksi, ja puolitamme taulukon koon.

sa, kopioinnista tuleva lisäkustannus on $n + n/2 + n/4 + n/8 + \dots = O(n)$, eli keskimääräinen kustannus alkiota kohden on $O(1)$. Taulukon kasvattamisen vaikutus kokonaisuuteen on siis pieni.

Voimme poistaa alkion listan lopusta aina $O(1)$ -ajassa, koska taulukon kokoa ei tarvitse koskaan suurentaa. Tässä voi kuitenkin tulla ongelmaksi, että monien poistojen jälkeen taulukossa on turhan paljon tyhjää tilaa lopussa. Voimme soveltaa tässä käänteisesti samaa ideaa kuin lisäämisessä: jos poistamisen jälkeen vain *neljännes* taulukosta on käytössä, puolitamme taulukon koon. Kuva 4.3 näyttää esimerkin tällaisesta tilanteesta. Tällä tavalla myös poistamiset vievät keskimäärin aikaa $O(1)$.

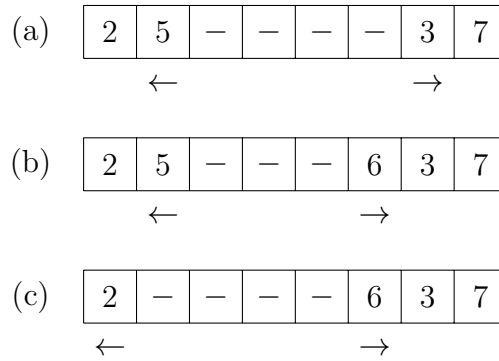
Miksi emme voisi varata heti aluksi niin suurta taulukkoa, että lopullinen lista mahtuisi siihen varmasti? Tässä olisi huonona puolena, että listamme tuhlaisi paljon muistia. Algoritmissa saattaa olla samaan aikaan käytössä monia listoja, ja haluamme, että listalle varattu taulukko on samaa kokoluokkaa kuin listan todellinen sisältö.

4.1.2 Muutokset alussa ja lopussa

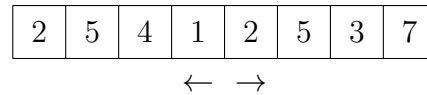
Melko samaan tapaan voimme myös luoda taulukkolistan, joka sallii tehokkaat alkioden lisäykset ja poistot sekä listan alussa että lopussa. Jotta tämä onnistuisi, muutamme listan tallennustapaa niin, että lista voi alkaa ja päättyä missä tahansa taulukon kohdassa ja listan sisältö voi tarvittaessa jatkua taulukon lopusta alkuun.

Kuva 4.4 näyttää esimerkin listan $[3, 7, 2, 5]$ uudesta tallennustavasta. Merkki \rightarrow osoittaa kohdan, josta lista alkaa, ja merkki \leftarrow osoittaa kohdan, johon lista päättyy. Kun haluamme lisätä alkion listan alkuun, siirrymme vasemmalle kohdasta \rightarrow , ja kun haluamme lisätä alkion listan loppuun, siirrymme oikealle kohdasta \leftarrow . Kun haluamme poistaa alkioita listasta, menetelmämme käänteisesti.

Jos kohdat \rightarrow ja \leftarrow ovat vierekkäin, taulukko on täynnä, emmekä voi enää lisätä uutta alkiota listan alkuun tai loppuun. Kuva 4.5 näyttää esimerkin tällaisesta tilanteesta. Tällöin meidän täytyy varata uusi suurempi



Kuva 4.4: (a) Lista $[3, 7, 2, 5]$ tallennettuna taulukkoon. (b) Listan alkuun lisätään alkio 6. (c) Listan lopusta poistetaan alkio 5.



Kuva 4.5: Lista täyttää koko taulukon, emmekä voi lisätä uutta alkioita. Ratkaisuna on varata listalle uusi suurempi taulukko.

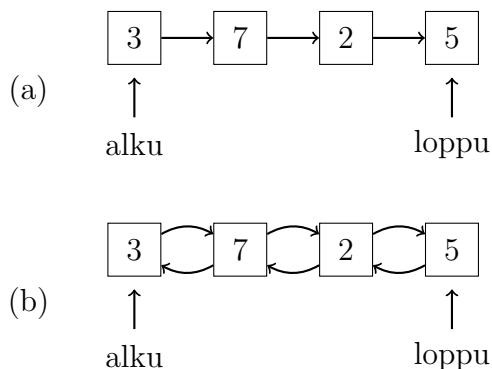
taulukko, johon listan sisältö siirretään. Voimme menetellä samalla tavalla kuin aiemmin ja esimerkiksi kaksinkertaistaa taulukon koon joka vaiheessa, jolloin operaatiot vievät keskimäärin aikaa $O(1)$.

4.2 Linkitetty lista

Linkitetty lista (*linked list*) muodostuu solmuista, joista jokainen sisältää yhden listan alkion. Linkitetty lista voi olla yhteen tai kahteen suuntaan linkitetty. Yhteen suuntaan linkitetyssä listassa jokaisesta solmusta on viittaus seuraavaan solmuun, ja kahteen suuntaan linkitetyssä listassa jokaisesta solmusta on viittaus sekä seuraavaan että edelliseen solmuun.

Kuva 4.6 näyttää esimerkkinä listan $[3, 7, 2, 5]$ yhteen ja kahteen suuntaan linkitettyinä. Molemmissa listoissa tiedossamme on viittaukset listan alkuun ja loppuun. Yhteen suuntaan linkitetyssä listassa voimme käydä läpi listan alkiot alusta loppuun, kun taas kahteen suuntaan linkitetyssä listassa voimme kulkea sekä alusta loppuun että lopusta alkuun.

Kaksisuuntainen linkitys on käytännössä järkevä tapa toteuttaa linkitetty lista, ja oletamme jatkossa, että listamme on kahteen suuntaan linkitetty ja meillä on tiedossa viittaukset listan alkuun ja loppuun.



Kuva 4.6: Lista [3, 7, 2, 5] linkitettyä listana. (a) Yhteen suuntaan linkitetty lista. (b) Kahteen suuntaan linkitetty lista.

4.2.1 Linkitetyt rakenteet

Jokaisessa ohjelmointikielessä on omat keinonsa linkitetyn rakenteen toteuttamiseen. Javassa voimme toteuttaa linkitetyn rakenteen niin, että jokainen solmu on oma olionsa. Esimerkiksi voimme toteuttaa seuraavan luokan `Solmu`, jonka oliot toimivat linkitetyn listan solmuina:

```
public class Solmu {
    public int arvo;
    public Solmu seuraava;
    public Solmu edellinen;

    public Solmu(int arvo, Solmu seuraava, Solmu edellinen) {
        this.arvo = arvo;
        this.seuraava = seuraava;
        this.edellinen = edellinen;
    }
}
```

Kenttä `arvo` kertoo solmun arvon, kenttä `seuraava` osoittaa seuraavaan solmuun ja kenttä `edellinen` osoittaa edelliseen solmuun. Jos seuraavaa tai edellistä solmua ei ole, viittauksen tilalla on arvo `null`. Esimerkiksi seuraava koodi luo solmun, jonka arvona on 5 ja joka ei viittaa mihinkään:

```
Solmu s = new Solmu(5, null, null);
```

Seuraava koodi puolestaan näyttää, kuinka voimme luoda tämän luokan avulla linkitetyn listan [3, 7, 2, 5]:

```
Solmu s1, s2, s3, s4;  
s1 = new Solmu(3, s2, null);  
s2 = new Solmu(7, s3, s1);  
s3 = new Solmu(2, s4, s2);  
s4 = new Solmu(5, null, s3);
```

Tämän jälkeen voimme käydä listan läpi näin alusta loppuun:

```
Solmu s = s1;  
while (s != null) {  
    System.out.println(s.arvo);  
    s = s.seuraava;  
}
```

Koodin tulostus on seuraava:

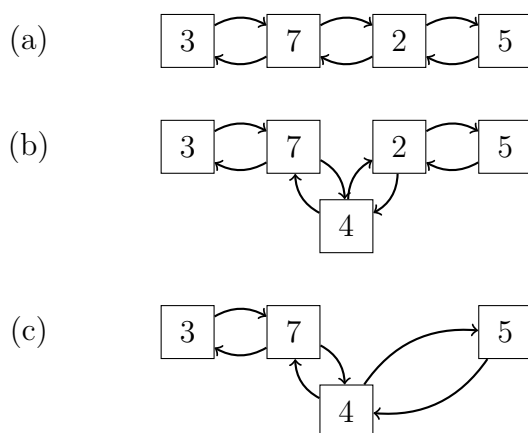
```
3  
7  
2  
5
```

4.2.2 Listan operaatiot

Linkitetyn listan etuna on, että voimme lisätä ja poistaa alkioita $O(1)$ -ajassa kaikissa listan kohdissa. Kun haluamme lisätä listalle alkion, luomme ensin uuden solmun ja muutamme sitten sen vieressä olevien solmujen viittauksia niin, että ne viittaavat uuteen solmuun. Vastaavasti kun haluamme poistaa alkion, muutamme viittauksia niin, että solmu ohitetaan.

Kuva 4.7 näyttää esimerkin linkitetyn listan käsittelystä. Listan sisältönä on aluksi $[3, 7, 2, 5]$. Sitten lisäimme listan keskelle alkion 4, jolloin luomme ensin uuden solmun alkioille ja muutamme sitten viittauksia alkioden 7 ja 2 välillä niin, että alkio 4 tulee niiden väliin. Lopuksi poistamme listasta alkion 2, jolloin yhdistämme alkiot 4 ja 5 suoraan toisiinsa.

Pääsemme listan ensimmäiseen ja viimeiseen solmuun tehokkaasti, koska meillä on muistissa viittaukset niihin. Sen sijaan jos haluamme päästä johonkin muuhun listan kohtaan, meidän täytyy aloittaa listan alusta tai lopusta ja kulkea askel kerrallaan viittauksia seuraten. Niinpä listan keskellä olevaan kohtaan pääseminen vie aikaa $O(n)$. Joudumme liikkumaan solmuihin linkkejä pitkin, koska solmut voivat olla eri puolilla muistia eikä meillä ole keinoa tietää suoraan, mihin mikäkin solmu on tallennettu.



Kuva 4.7: (a) Alkuperäinen lista $[3, 7, 2, 5]$. (b) Listan keskelle lisätään alkio 4. (c) Listasta poistetaan alkio 2.

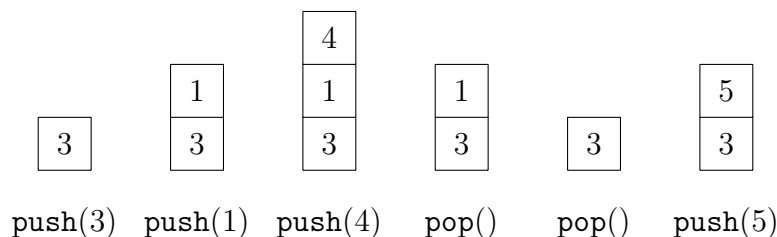
operaatio	taulukkolista	linkitetty lista
pääsy listan alkuun	$O(1)$	$O(1)$
pääsy listan loppuun	$O(1)$	$O(1)$
pääsy listan keskelle	$O(1)$	$O(n)$
lisäys/poisto listan alussa	$O(1)$	$O(1)$
lisäys/poisto listan lopussa	$O(1)$	$O(1)$
lisäys/poisto listan keskellä	$O(n)$	$O(1)$

Taulukko 4.1: Taulukkolistan ja linkitetyn listan operaatioiden aikavaativuuksia.

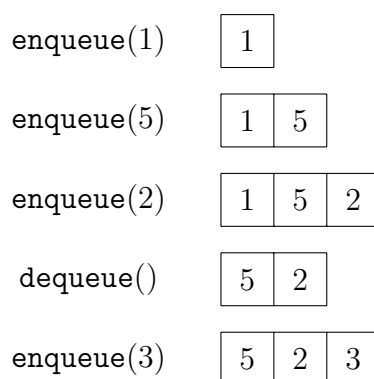
4.2.3 Listojen vertailua

Taulukko 4.1 esittää yhteenvedon taulukkolistan ja linkitetyn listan ominaisuuksista. Kummassakin toteutuksessa on yksi operaatio, joka ei ole tehokas. Taulukkolistassa pääsemme tehokkaasti mihin tahansa listan kohtaan, mutta on hidasta muokata listaa keskeltä. Linkitettyssä listassa voimme muokata listaa mistä tahansa, mutta keskelle pääseminen on hidasta.

Huomaa, että keskelle pääsemisen hitaus rajoittaa melko paljon linkitetyn listan käyttämistä. Vaikka pystymme sinänsä muokkaamaan listaa mistä tahansa kohdasta tehokkaasti, meidän tulee ensin *päästä* kyseiseen kohtaan. Jos meillä on jostain syystä etukäteen tiedossa viittaus listan keskelle, voimme muokata kyseistä kohtaa tehokkaasti, mutta muuten meidän tulee ensin kulkea haluttuun kohtaan, missä kuluu aikaa $O(n)$.



Kuva 4.8: Esimerkki pinon käsittelystä: lisäämme tyhjään pinoon alkiot 3, 1 ja 4, poistamme kaksi ylintä alkioita ja lisäämme lopuksi alkion 5.



Kuva 4.9: Esimerkki jonon käsittelystä: lisäämme tyhjään jonon alkiot 1, 5 ja 2, poistamme yhden alkion ja lisäämme vielä alkion 3.

4.3 Pino ja jono

Listan avulla voimme toteuttaa myös kaksi erikoistunutta tietorakennetta, pinon ja jonon, jotka sisältävät vain osan listan ominaisuuksista eli niiden operaatiot ovat listaa rajoittuneempia.

Pino (*stack*) on tietorakenne, jonka operaatiot ovat alkion lisääminen pinon päälle (**push**), ylimmän alkion poistaminen (**pop**) sekä ylimmän alkion hakeminen. Esimerkiksi kuvassa 4.8 lisäämme ensin tyhjään pinoon kolme alkioita, poistamme sitten kaksi alkioita ja lisäämme vielä yhden alkion.

Jono (*queue*) on tietorakenne, jossa voimme lisätä alkioita jonon loppuun (**enqueue**), poistaa alkioita jonon alusta (**dequeue**) ja hakea alkioita molemmista päistä. Esimerkiksi kuvassa 4.9 lisäämme ensin tyhjään jonoon kolme alkioita, poistamme sitten yhden alkion ja lisäämme vielä yhden alkion.

Pystymme toteuttamaan sekä pinon että jonon helposti listana niin, että niiden operaatiot toimivat ajassa $O(1)$. Mutta mitä järkeä on luoda uusia tietorakenteita, jotka ovat *huonompia* kuin lista? Listassa voimme käsitellä mitä tahansa alkioita, mutta pinossa ja jonossa emme pääse käsiksi keskellä

oleviin alkioihin. Selitys on siinä, että pino ja jono ovat hyödyllisiä *käsitteitä* algoritmien suunnittelussa. Voimme usein ajatella algoritmissa tarvittavaa tietorakennetta pinona tai jonona ja toteuttaa sen sitten listana.

Tarkastellaan esimerkkinä tehtävää, jossa annettuna on n merkin pituinen *sulkulauseke*, joka muodostuu kaarisulkeista `()` sekä hakasulkeista `[]`. Haluamme selvittää, onko lauseke *oikein muodostettu* eli onko jokaiselle aloitettavalle sululle vastaava lopettava pari. Esimerkiksi lauseke `[(())]` on oikein muodostettu, kun taas lauseke `[(())]` ei ole.

Voimme ratkaista tehtävän $O(n)$ -ajassa pinon avulla käymällä läpi lausekkeen merkit vasemmalta oikealle. Kun vastaan tulee aloittava sulku `(` tai `[`, lisäämme sen pinoon. Kun taas vastaan tulee lopettava sulku `)` tai `]`, tutkimme, mikä on pinossa ylimpänä oleva merkki. Jos merkki on vastaava aloittava sulku, poistamme sen pinosta, ja muuten toteamme, että lauseke on virheelinen. Jos lausekkeen läpikäynnin aikana ei esiinny virheitä ja pino on lopuksi tyhjä, lauseke on oikein muodostettu.

4.4 Javan toteutukset

Javan standardikirjastossa on monia listojen toteutuksia, jotka pohjautuvat taulukkolistaan tai linkitettyyn listaan. Seuraavaksi tutustumme rakenteisiin, joista on usein hyötyä algoritmien toteutuksessa.

4.4.1 ArrayList-rakenne

`ArrayList` on taulukkolista, joka sallii tehokkaat lisäykset ja poistot listan lopussa. Esimerkiksi seuraava koodi luo listan, lisää siihen alkiot 1, 2 ja 3 ja tulostaa listan sisällön.

```
ArrayList<Integer> lista = new ArrayList<>();
lista.add(1);
lista.add(2);
lista.add(3);
System.out.println(lista); // [1, 2, 3]
```

Metodi `add` lisää oletuksena alkion listan loppuun ja toimii siinä tapauksessa keskimäärin ajassa $O(1)$.

Koska lista on tallennettu taulukkona, pääsemme myös tehokkaasti käsiksi sen alkioihin kohdan perusteella. Metodi `get` hakee tietyssä kohdassa olevan arvon, ja metodi `set` muuttaa arvoa. Esimerkiksi seuraava koodi tulostaa ensin listan kohdassa 1 olevan alkion ja muuttaa sitten sen arvoksi 5.

```
System.out.println(lista.get(1)); // 2
lista.set(1,5);
System.out.println(lista.get(1)); // 5
```

Luokassa `Collections` on hyödyllisiä metodeita, joiden avulla voimme muuttaa `ArrayList`-rakenteen alkioden järjestystä. Seuraava esimerkkikoodi järjestää ensin listan, muuttaa sitten sen järjestyksen käänteiseksi ja sekoittaa lopuksi järjestyksen.

```
Collections.sort(lista);
Collections.reverse(lista);
Collections.shuffle(lista);
```

4.4.2 ArrayDeque-rakenne

`ArrayDeque`-rakenne on taulukkolista, joka sallii tehokkaat lisäykset ja poistot sekä listan alussa että lopussa. Alkioita voi lisätä metodeilla `addFirst` ja `addLast` ja poistaa metodeilla `removeFirst` ja `removeLast`:

```
ArrayDeque<Integer> lista = new ArrayDeque<>();
lista.addLast(1);
lista.addFirst(2);
lista.addLast(3);
System.out.println(lista); // [2, 1, 3]
lista.removeFirst();
System.out.println(lista); // [1, 3]
```

Lisäksi voimme hakea listan ensimmäisen ja viimeisen alkion metodeilla `getFirst` ja `getLast`:

```
ArrayDeque<Integer> lista = new ArrayDeque<>();
lista.addLast(1);
lista.addLast(2);
lista.addLast(3);
System.out.println(lista.getFirst()); // 1
System.out.println(lista.getLast()); // 3
```

Kaikki nämä metodit toimivat keskimäärin ajassa $O(1)$. Rajoituksena on kuitenkin, että emme pääse käsiksi listan keskellä oleviin alkioihin, vaan voimme käsitellä vain listan alkua ja loppua.

4.4.3 LinkedList-rakenne

`LinkedList`-rakenne toteuttaa kaksisuuntaisen linkitetyn listan, jossa voimme helposti lisätä ja poistaa alkioita listan alussa ja lopussa. Seuraava koodi esittelee asiaa:

```
LinkedList<Integer> lista = new LinkedList<>();
lista.addLast(1);
lista.addFirst(2);
lista.addLast(3);
System.out.println(lista); // [2, 1, 3]
lista.removeFirst();
System.out.println(lista); // [1, 3]
```

Jos haluamme tehdä lisäyksiä ja poistoja muualla listassa, meidän täytyy ottaa käyttöön *iteraattori*, joka osoittaa haluttuun kohtaan. Seuraava koodi luo iteraattorin, joka osoittaa ensin listan alkuun. Sitten siirrämme iteraattoria kaksi askelta eteenpäin ja lisäämme alkion 5 iteraattorin kohdalle eli listan toisen ja kolmannen alkion väliin.

```
ListIterator<Integer> x = lista.listIterator(0);
x.next();
x.next();
x.add(5);
```

`LinkedList` tarjoaa myös metodit `get` ja `set`, joiden avulla pääsemme käsiksi tiettyssä kohdassa listalla olevaan alkioon. Nämä metodit vievät kuitenkin aikaa $O(n)$, koska joudumme kulkemaan ensin oikeaan kohtaan listan alusta tai lopusta. Tämän vuoksi `LinkedList` ei ole hyvä valinta, jos haluamme käsitellä alkioita kohdan perusteella.

4.5 Tehokkuusvertailu

Tärkeä kysymys on, miten tehokkaita taulukkolista ja linkitetty lista ovat *käytännössä* ja kumpaa meidän kannattaa käyttää, jos voimme valita. Seuraavaksi vertailemme Javan taulukkolistan (`ArrayList`) ja linkitetyn listan (`LinkedList`) käytännön tehokkuutta.

Testissä luomme ensin taulukon, joka sisältää luvut $1, 2, \dots, n$ satunnaisessa järjestyksessä, sekä tyhjän listan. Tämän jälkeen käymme taulukon läpi vasemmalta oikealle ja lisäämme kunkin luvun listalle sen oikealle paikalle niin, että lista säilyy järjestettynä. Esimerkiksi jos listalla on ennestään alkiot $[2, 3, 6]$ ja seuraava taulukon alkio on 4, listasta tulee $[2, 3, 4, 6]$. Teemme

parametri n	ArrayList	LinkedList
10000	0.13 s	0.38 s
20000	0.48 s	1.20 s
30000	0.99 s	2.70 s
40000	1.72 s	5.15 s
50000	3.14 s	8.99 s

Taulukko 4.2: Listarakenteiden tehokkuusvertailu.

saman testin taulukkolistalle ja linkitetylle listalle.

Taulukkolistan tapauksessa käymme listaa läpi alusta muuttujalla k , kunnes tulemme kohtaan, johon uusi alkio kuuluu. Tämän jälkeen lisäämme alkion kohtaan k kutsumalla metodia `add`.

```
for (int i = 0; i < n; i++) {
    int k = 0;
    while (k < lista.size() && lista.get(k) < taulu[i]) {
        k++;
    }
    lista.add(k,taulu[i]);
}
```

Linkitetyn listan tapauksessa luomme iteraattorin, jonka avulla etsimme uuden alkion kohdan listan alusta lähtien. Tämän jälkeen lisäämme alkion listalle iteraattorin osoittamaan kohtaan.

```
for (int i = 0; i < n; i++) {
    ListIterator<Integer> x = lista.listIterator(0);
    while (x.hasNext()) {
        if (x.next() > taulu[i]) {
            x.previous();
            break;
        }
    }
    x.add(taulu[i]);
}
```

Taulukkolistassa sekä oikean kohdan etsiminen että alkion lisääminen vievät aikaa $O(n)$, kun taas linkitettyssä listassa oikean kohdan etsiminen vie aikaa $O(n)$, mutta alkion lisääminen vie aikaa vain $O(1)$. Mutta kuinka nopeasti koodit toimivat käytännössä?

Taulukko 4.2 näyttää testin tulokset. Osoittautuu, että taulukkolista on selvästi *nopeampi* kuin linkitetty lista. Näin käy siitä huolimatta, että tau-

	3	7	2	5			

	3					2	
				7			
		5					

Kuva 4.10: Taulukkolista ja linkitetty lista tietokoneen muistissa.

lukkolistassa alkion lisääminen vie aikaa $O(n)$, mutta linkitettyssä listassa aikaa kuluu $O(1)$. Kysymys kuuluukin:

Milloin kannattaa käyttää linkitettyä listaa?

Tietorakenteiden maailmassa jokaiselle tietorakenteelle on yleensä omat tietyt käyttötarkoituksensa, joissa se erottuu edukseen muista tietorakenteista. Linkitetty lista muodostaa kuitenkin poikkeuksen tähän sääntöön: *sitä ei kannata käyttää yleensä koskaan*.

Syynä tähän on, että nykyaikaiset tietokoneet suosivat taulukkolistan käyttämistä linkitetyn listan sijaan. Kuvassa 4.10 näkyy, miten taulukkolista ja linkitetty lista asettuvat tietokoneen muistissa. Taulukkolistan alkiot ovat peräkkäin, kun taas linkitetyn listan alkiot voivat olla eri puolilla muistia sekalaisessa järjestyksessä. Nykyaikaisen prosessorin välimuistit ja komentojen ennustus on toteutettu niin, että ne ovat parhaimmillaan silloin, kun tieto on tallennettu muistissa peräkkäin – eli juuri kuten taulukkolistassa. Tämä näkyy käytännössä siinä, että taulukkolistan käsittely on selvästi tehokkaampaa kuin linkitetyn listan käsittely.

Vaikka linkitetty lista ei ole käytännössä hyvä tietorakenne, linkitetyn rakenteen idea on hyödyllinen ja tarvitsemme sitä myöhemmin monimutkaisemmissa tietorakenteissa.

Luku 5

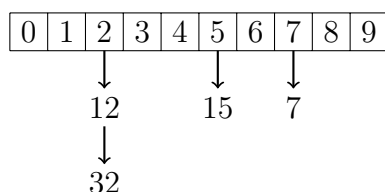
Hajautustaulu

Hajautustaulu (*hash table*) on tietorakenne, joka pitää yllä alkioden joukkoa. Voimme tarkastaa, kuuluuko tietty alkio joukkoon, sekä lisätä ja poistaa alkioita. Kuten matematiikassa, jokainen alkio voi esiintyä enintään kerran joukossa. Hajautustaulussa kaikki yllä mainitut operaatiot toimivat tehokkaasti, mikä tekee siitä kätevän työkalun algoritmien toteuttamisessa.

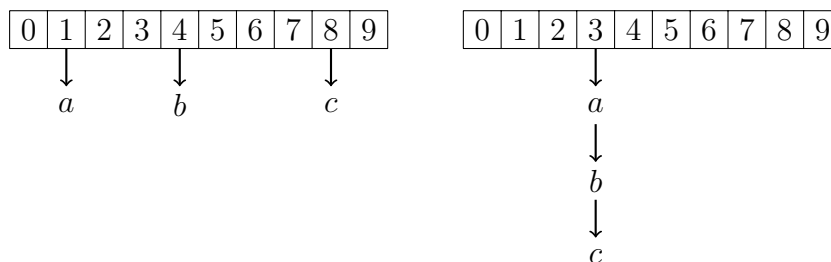
Tutustumme tässä luvussa ensin hajautustaulun toimintaan ja sen tehokkuuteen vaikuttaviin seikkoihin. Tämän jälkeen käsittelemme Javan tietorakenteet `HashSet` ja `HashMap`, jotka perustuvat hajautustauluun. Lopuksi käymme läpi esimerkkejä tilanteista, joissa voimme käyttää hajautustaulua algoritmien suunnittelussa.

5.1 Hajautustaulun toiminta

Toteutamme hajautustaulun taulukkona, jonka jokaisessa kohdassa on lista joukkoon kuuluvia alkioita. Jotta voimme käyttää hajautustaulua, tarvitsemme *hajautusfunktion* (*hash function*) f , joka antaa *hajautusarvon* (*hash value*) $f(x)$ mille tahansa joukon alkioille x . Hajautusarvo on kokonaisluku väliltä $0, 1, \dots, N - 1$, missä N on hajautustaulun koko. Tallennamme



Kuva 5.1: Hajautustaulu, joka vastaa joukkoa $\{7, 12, 15, 32\}$. Hajautusfunktiona on $f(x) = x \bmod 10$.



Kuva 5.2: Kaksi hajautustaulua joukolle $\{a, b, c\}$. Vasen tilanne on paras mahdollinen, oikea tilanne taas huonoin mahdollinen.

hajautustaulun kohdassa k olevaan listaan kaikki ne joukon alkiot, joiden hajautusarvo on k .

Kuvassa 5.1 on esimerkkinä hajautustaulu, jonka kokona on $N = 10$. Olemme tallentaneet hajautustauluun joukon $\{7, 12, 15, 32\}$ käyttäen hajautusfunktioita $f(x) = x \bmod 10$. Tämä tarkoittaa, että alkion x hajautusarvo on sen jakojäännös 10:llä eli luvun viimeinen numero. Esimerkiksi alkiot 12 ja 32 ovat kohdassa 2, koska niissä viimeinen numero on 2, ja alkiot 15 ja 7 ovat vastaavasti kohdissa 5 ja 7. Kaikki muut hajautustaulun listat ovat tällä hetkellä tyhjiä.

Kun haluamme tarkastaa, onko joukossa alkioita x , laskemme ensin sen hajautusarvon $f(x)$. Tämän jälkeen käymme läpi kaikki kohdan $f(x)$ listassa olevat alkiot ja tarkastamme, onko jokin niistä alkio x . Vastaavasti kun haluamme lisätä alkion x joukkoon tai poistaa alkion x joukosta, teemme muutoksen kohdassa $f(x)$ olevaan listaan. Jokaisen operaation aikavaativuus on $O(m)$, missä m on listan alkioden määrä. Hajautustaulu toimii siis tehokkaasti, jos jokainen siinä oleva lista on lyhyt.

5.1.1 Hajautusfunktio

Hajautusfunktio $f(x)$ määrittää, mihin kohtaan hajautustaulua alkio x sijoitetaan. Sen täytyy antaa jokaiselle mahdolliselle alkiolle hajautusarvo eli kokonaisluku väliltä $0, 1, \dots, N - 1$, missä N on hajautustaulun koko. Muilta osin meillä on periaatteessa vapaat kädet hajautusfunktion suunnitteluun. Mutta millainen olisi hyvä hajautusfunktio?

Huomamme, että hajautusfunktio jakaa alkioita *tasaisesti* hajautustaulun eri puolille. Jos onnistumme tässä, kaikki listat ovat lyhyitä ja hajautustaulun operaatiot ovat tehokkaita. Kuva 5.2 näyttää kaksi hajautustaulua, jotka vastaavat joukkoa $\{a, b, c\}$ kahdella eri hajautusfunktioilla. Vasemmassa taulussa hajautus on onnistunut täydellisesti ja jokainen alkio on omassa listassaan. Oikeassa taulussa taas kaikki alkiot ovat joutuneet samaan lis-

taan eikä hajautuksesta ole mitään hyötyä. Tavoitteemme on saada aikaan hajautusfunktio, jonka toiminta on lähempänä vasenta tilannetta.

Jos hajautettavat alkiot ovat kokonaislukuja, suoraviivainen hajautusfunktio on $f(x) = x \bmod N$, mikä tarkoittaa, että otamme jakojäännöksen hajautustaulun koolla N . Tämä on hyvin toimiva hajautusfunktio, kunhan aineistossa esiintyy tasaisesti eri jakojäännöksiä. Entä jos alkiot ovat jotain muuta tyyppiä kuin kokonaislukuja? Tällöin meidän täytyy päättää ensin jokin järkevä tapa, kuinka muutamme alkion kokonaisluvuksi, minkä jälkeen otamme jakojäännöksen N :llä.

Tarkastellaan esimerkkinä tilannetta, jossa haluamme hajauttaa merkkijonoja eli meidän täytyy löytää keino muuttaa merkkijono kokonaisluvuksi. Oletamme, että merkkijonossa on k merkkiä, joiden merkkikoodit ovat c_0, c_1, \dots, c_{k-1} . Esimerkiksi jos merkkijono on **apina**, merkkikoodit¹ ovat $c_0 = 97$, $c_1 = 112$, $c_2 = 105$, $c_3 = 110$ ja $c_4 = 97$. Yksi tapa muuttaa merkkijono kokonaisluvuksi on laskea merkkikoodien summa

$$c_0 + c_1 + \dots + c_{k-1},$$

jolloin merkkijonoa **apina** vastaa kokonaisluku

$$97 + 112 + 105 + 110 + 97 = 521.$$

Tämä on sinänsä järkevä tapa, mutta siinä on yksi ongelma: kaksi merkkijonoa saavat aina saman hajautusarvon, jos niissä on samat merkit eri järjestyksessä. Pystymme parantamaan hajautusarvon laskentaa lisäämällä summaan *kertoimet* käyttäen kaavaa

$$A^{k-1}c_0 + A^{k-2}c_1 + \dots + A^0c_{k-1},$$

missä A on vakio. Esimerkiksi jos $A = 7$, merkkijonoa **apina** vastaa kokonaisluku

$$7^4 \cdot 97 + 7^3 \cdot 112 + 7^2 \cdot 105 + 7^1 \cdot 110 + 7^0 \cdot 97 = 277325.$$

Tämä menetelmä, jota kutsutaan nimellä *polynominen hajautus* (*polynomial hashing*), on käytännössä hyvä merkkijonon hajautustapa, joka on käytössä esimerkiksi Javan standardikirjastossa.

¹Käytämme tässä merkkien ASCII-koodeja. Esimerkiksi Javassa char-merkin `c` koodin saa selville kirjoittamalla `(int)c`, eli esimerkiksi `(int)'a'` on 97.

5.1.2 Hajautuksen tehokkuus

Hajautustaulun operaatiot vievät aikaa $O(m)$, jossa m on hajautustaulussa olevan listan pituus. Mutta kuinka suuri m on? Tämä riippuu siitä, mikä on alkioden määrä n , hajautustaulun koko N sekä hajautusfunktio f .

Jos kaikki sujuu hyvin ja hajautusfunktio jakaa alkioita tasaisesti hajautustaulun eri puolille, jokaisessa listassa on noin n/N alkioita. Niinpä jos valitsemme hajautustaulun koon niin, että N on samaa luokkaa kuin n , operaatiot toimivat tehokkaasti ajassa $O(1)$. Kuitenkin on mahdollista että hajautus epäonnistuu ja alkiot jakautuvat hajautustauluun epätasaisesti. Pahimmassa tapauksessa kaikki alkiot saavat saman hajautusarvon ja ne kaikki tallennetaan samaan listaan, jolloin operaatiot vievät aikaa $O(n)$.

Voimme helposti vaikuttaa hajautustaulun kokoon N , mutta hajautusfunktion suunnittelu on epämääräisempi ala. Miten voimme tietää, että valitsemamme hajautusfunktio toimii hyvin? Itse asiassa emme voi olla koskaan varmoja tästä. Vaikka meillä olisi erittäin hyvä hajautusfunktio, *ilkeä vastustaja* voi kuitenkin antaa meille joukon alkioita, jotka kaikki saavat saman hajautusarvon. Tämä riski on aina hajautuksessa, koska mahdollisten hajautusarvojen määrä on paljon pienempi kuin mahdollisten alkioden määrä. Tämän vuoksi emme voi mitenkään suunnitella hajautusfunktiota niin, että se jakaisi alkiot *varmasti* tasaisesti hajautustauluun.

Kaikeksi onneksi hajautus toimii yleensä aina *käytännössä* hyvin ja voimme ajatella, että hajautustaulun operaatiot ovat $O(1)$ -aikaisia, kunhan hajautustaulun koko on riittävän suuri ja hajautusfunktio on toteutettu järkevästi. Vaikka on mahdollista, että hajautus epäonnistuu, tämän riski on niin pieni, että meidän ei tarvitse murehtia siitä käytännössä.

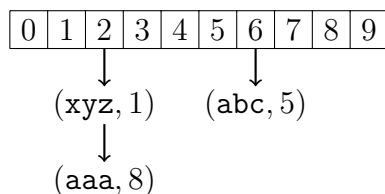
5.1.3 Hajautustaulu hakemistona

Voimme tallentaa hajautustauluun myös avain-arvo-pareja, joissa avaimen hajautusarvo määrittää, mihin hajautustaulun listaan pari sijoitetaan. Saamme näin aikaan tietorakenteen *hakemisto* (*dictionary*), jota voi ajatella taulukon yleistykseenä. Taulukossa avaimet ovat kokonaisluvut $0, 1, \dots, n-1$, mutta hakemistossa ne voivat olla mitä tahansa arvoja.

Kuvassa 5.3 on esimerkkinä hajautustauluun tallennettu hakemisto, joka vastaa seuraavaa "taulukkoa":

<pre>taulu["abc"] = 5 taulu["xyz"] = 1 taulu["aaa"] = 8</pre>

Tässä tapauksessa hakemiston avaimet ovat merkkijonoja ja arvot ovat



Kuva 5.3: Hakemiston tallentaminen hajautustauluun.

kokonaislukuja. Hajautustaulun ansiosta voimme käsitellä hakemistoa taulukon tavoin niin, että operaatiot vievät aikaa $O(1)$.

Voimme myös toteuttaa hakemiston, jonka avaimet ovat kokonaislukuja. Tällaisessa tietorakenteessa on järkeä, jos avaimet ovat niin suuria, että emme voi käyttää sen sijasta tavallista taulukkoa. Kuitenkin jos avaimet ovat pieniä kokonaislukuja, taulukko on paljon parempi valinta, koska sen vakiokertoimet ovat huomattavasti hajautustaulua pienemmät.

5.2 Javan toteutukset

Javassa on kaksi hajautustaulua käyttävää tietorakennetta: `HashSet` pitää yllä alkioden joukkoa ja `HashMap` toteuttaa hakemiston, jossa on avain-arvopareja. Kummankin rakenteen operaatiot toimivat ajassa $O(1)$.

5.2.1 HashSet-rakenne

`HashSet` on alkioden joukko, johon voi lisätä alkion metodilla `add` ja josta voi poistaa alkion metodilla `remove`. Esimerkiksi seuraava koodi luo joukon, jossa voi olla kokonaislukuja, ja lisää siihen luvut 3, 5 ja 8. Tämän jälkeen koodi poistaa luvun 5 joukosta.

```
HashSet<Integer> joukko = new HashSet<>();
joukko.add(3);
joukko.add(5);
joukko.add(8);
System.out.println(joukko); // [3, 5, 8]
joukko.remove(5);
System.out.println(joukko); // [3, 8]
```

Metodi `contains` kertoo, esiintyykö tietty alkio x joukossa:

```
if (joukko.contains(x)) {  
    System.out.println("alkio on joukossa");  
} else {  
    System.out.println("alkiota ei ole joukossa");  
}
```

Huomaa, että jokainen alkio voi esiintyä vain kerran joukossa. Esimerkiksi vaikka seuraava koodi lisää luvun 5 kolmesti joukkoon, se menee sinne vain ensimmäisellä kerralla ja muut lisäykset jätetään huomiotta.

```
HashSet<Integer> joukko = new HashSet<>();  
joukko.add(5);  
joukko.add(5);  
joukko.add(5);  
System.out.println(joukko); // [5]
```

5.2.2 HashMap-rakenne

HashMap luo hakemiston, jossa on avain-arvo-pareja. Hakemiston määrittelyssä tulee antaa avaimen ja arvon tyyppi. Metodi **put** lisää uuden avain-arvo-parin, ja metodi **get** hakee arvon avaimen perusteella.

Esimerkiksi seuraava koodi luo sanakirjan, jossa sekä avaimet että arvot ovat merkkijonoja. Syötämme sanakirjaan merkkijonopareja, jotka kertovat sanan käännöksen suomesta englanniksi.

```
HashMap<String,String> sanakirja = new HashMap<>();  
  
sanakirja.put("apina","monkey");  
sanakirja.put("banaani","banana");  
sanakirja.put("cembalo","harpsichord");  
  
System.out.println(sanakirja.get("banaani")); // banana
```

Hyödyllinen on myös metodi **containsKey**, jonka avulla voi tarkastaa, onko tietylle avaimelle tallennettu arvoa:

```
if (sanakirja.containsKey(sana)) {  
    System.out.println(sanakirja.get(sana));  
} else {  
    System.out.println("Sana puuttuu sanakirjasta!");  
}
```

5.2.3 Omat luokat

Javan luokissa on metodi `hashCode`, jonka avulla olio kertoo pyydettyä hajautusarvonsa. Voimme esimerkiksi selvittää merkkijonon `apina` hajautusarvon seuraavasti:

```
System.out.println("apina".hashCode());
```

Tämä koodi tulostaa luvun 93022541, joka on siis merkkijonon `apina` hajautusarvo Javassa. On tunnettua, että Java käyttää merkkijonon hajautusarvon laskemiseen polynomista hajautusta vakiolla $A = 31$, joten voimme laskea Javan hajautusarvon myös itse kaavalla

$$31^4 \cdot 97 + 31^3 \cdot 112 + 31^2 \cdot 105 + 31^1 \cdot 110 + 31^0 \cdot 97 = 93022541.$$

Jos haluamme käyttää omia olioitamme hajautustauluissa, meidän täytyy toteuttaa luokkaan kaksi metodia: `hashCode`, joka antaa olion hajautusarvon, sekä `equals`, joka ilmaisee, ovatko kaksi oliota samat. Metodi `hashCode` riittää toteuttaa niin, että se palauttaa jonkin kokonaisluvun. Metodi `equals` on tarpeen, jotta Java pystyy varmistamaan, ovatko saman hajautusarvon antavat oliot todella samat.

5.3 Hajautustaulun käyttäminen

Hajautustaulun ansiosta voimme käyttää algoritmeissamme joukkoja ja hakemistoja, joiden operaatiot toimivat tehokkaasti. Voimme alkajaisiksi ratkoa mukavammin ajassa $O(n)$ sellaisia ongelmia, jotka olemme ratkoneet aiemmin järjestämisen avulla ajassa $O(n \log n)$.

Aloitamme ongelmasta, jossa haluamme selvittää, montako eri alkioita taulukko sisältää. Luvussa 3.5.1 ratkaisimme ongelman järjestämällä taulukon ja tutkimalla sen jälkeen vierekkäisiä alkioita. Nyt kun käytössämme on hajautustaulu, voimme vain lisätä kaikki alkiot joukkoon ja hakea lopuksi joukon koon. Näin saamme aikaan seuraavan algoritmin:

```
alkiot = []  
for i = 0 to n-1  
    alkiot.add(taulu[i])  
print(alkiot.size())
```

Tässä `alkiot` on hajautustaulua käyttävä joukko, minkä ansiosta algoritmi toimii ajassa $O(n)$.

Tarkastellaan sitten ongelmaa, jossa haluamme selvittää taulukon yleisimmän alkion. Ratkaisimme tämänkin ongelman aiemmin järjestämällä taulukon, mutta hajautustaulun avulla voimme lähestyä ongelmaa toisella tavalla luomalla hakemiston, jonka avaimet ovat taulukon alkioita ja arvot niiden esiintymiskertoja. Nyt voimme vain käydä läpi taulukon sisällön ja pitää kirjaa, montako kertaa mikäkin alkio esiintyy taulukossa:

```
laskuri = []
suurin = 0
for i = 0 to n-1
    laskuri[taulu[i]]++
    if laskuri[taulu[i]] > suurin
        suurin = laskuri[taulu[i]]
        yleisin = taulu[i]
print(yleisin)
```

Tässä hakemisto `laskuri` on toteutettu hajautustaulun avulla, jolloin avaimet voivat olla mitä tahansa lukuja ja operaatiot toimivat ajassa $O(1)$. Tuloksena on algoritmi, jonka aikavaativuus on $O(n)$.

Kuten nämä esimerkit osoittavat, hajautustaulu *helpottaa* algoritmien luomista, koska meidän ei tarvitse pukea ongelmia järjestämisen muotoon vaan voimme käsitellä niitä suoremmin. Mutta toisaalta olemme ratkoneet vain uudestaan tehtäviä, jotka ovat hoituneet mainiosti myös järjestämisen avulla. Antaisiko hajautustaulu meille jotain todellisia uusia mahdollisuuksia algoritmien suunnittelussa?

Hajautustaulu osoittaa todelliset kyntensä silloin, kun haluamme pitää yllä aidosti *dynaamista* tietorakennetta eli haluamme vuorotellen muuttaa tietorakennetta ja hakea sieltä tietoa. Tällöin emme voi enää toteuttaa algoritmia, joka järjestää koko aineiston kerran alussa. Esimerkki tällaisesta tehtävästä on, että käytössämme on funktio `haeLuku`, joka antaa lukuja yksi kerrallaan. Jokaisen luvun jälkeen meidän tulee ilmoittaa, montako eri lukua olemme saaneet tähän mennessä, ennen kuin voimme pyytää funktiolta seuraavan luvun. Voimme ratkaista tehtävän seuraavasti ajassa $O(n)$ hajautustaulun avulla:

```
alkiot = []
for i = 1 to n
    luku = haeLuku()
    alkiot.add(luku)
print(alkiot.size())
```

Tällaisesta algoritmista käytetään joskus nimeä *online-algoritmi*. Tämä

tarkoittaa, että algoritmille annetaan syötettä alkio kerrallaan ja algoritmi pystyy ilmoittamaan senhetkisen vastauksen joka alkion käsittelyn jälkeen. Vastaavasti *offline-algoritmi* tarvitsee käyttöönsä heti koko syötteen, jotta se voi käsitellä syötettä kokonaisuutena, kuten järjestää sen. Monissa tehtävissä online-algoritmi on vaikeampi keksiä kuin offline-algoritmi.

Luku 6

Binäärihakupuu

Binäärihakupuu (*binary search tree*) on tietorakenne, joka pitää yllä alkioiden joukkoa, samaan tapaan kuin hajautustaulu. Binäärihakupuu eroaa hajautustaulusta kuitenkin siinä, että se säilyttää alkioita *järjestyksessä*. Tämän ansiosta voimme esimerkiksi etsiä tehokkaasti joukon pienimmän tai suurimman alkion, mikä ei ole mahdollista hajautustaulussa.

Aloitamme luvun tutustumalla binääripuiden teoriaan, minkä jälkeen perehdymme binäärihakupuun toimintaan. Käsitlemme binäärihakupuun tehokkaasta toteutuksesta esimerkkinä AVL-puun. Sitten käymme läpi Javan tietorakenteet `TreeSet` ja `TreeMap`, jotka perustuvat binäärihakupuuhun, ja lopuksi vertailemme joukkorakenteita ja järjestämistä.

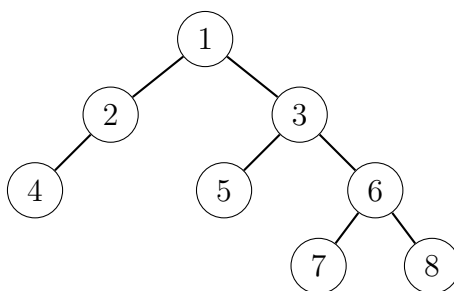
6.1 Taustaa binääripuista

Binäärihakupuun taustalla on yleisempi tietorakenne *binääripuu* (*binary tree*). Ennen kuin tutustumme binäärihakupuuhun, meidän onkin hyvä selvittää ensin, mikä on binääripuu ja mitä ominaisuuksia siihen liittyy.

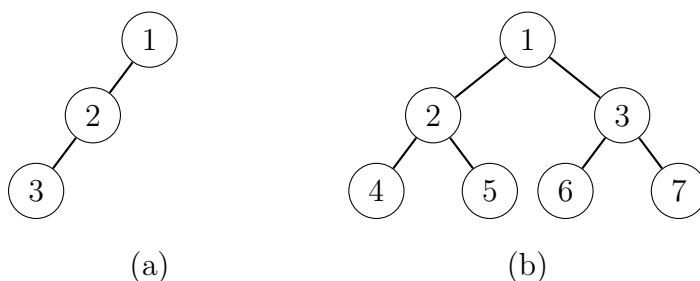
Binääripuu muodostuu n solmusta. Puussa ylimpänä on solmu, jota kutsutaan *juureksi* (*root*). Jokaisella solmulla voi olla vasen ja oikea *lapsi* (*child*), ja kaikilla solmuilla juurta lukuun ottamatta on yksikäsitteinen *vanhempi* (*parent*). Puun *lehtiä* (*leaf*) ovat solmut, joilla ei ole lapsia.

Binääripuun rakenne on rekursiivinen: jokainen solmu toimii juurena *alipuulle* (*subtree*), joka on myös binääripuu. Solmun x alipuu sisältää solmun x sekä kaikki solmut, joihin pääsemme laskeutumalla alaspäin solmusta x . Voimme myös ajatella asiaa niin, että jokaisen binääripuun solmun vasen ja oikea lapsi on toinen (mahdollisesti tyhjä) binääripuu.

Kuvassa 6.1 on esimerkki binääripuusta, jossa on 8 solmua. Solmu 1 on puun juuri, ja solmut 4, 5, 7 ja 8 ovat puun lehtiä. Solmun 3 vasen lapsi on



Kuva 6.1: Binääripuu, jossa on 8 solmua. Puun juuri on solmu 1, ja puun lehtiä ovat solmut 4, 5, 7 ja 8.



Kuva 6.2: (a) Vähiten solmuja sisältävä korkeuden 2 binääripuu. (b) Eniten solmuja sisältävä korkeuden 2 binääripuu.

solmu 5, oikea lapsi on solmu 6 ja vanhempi on solmu 1. Solmun 3 alipuu sisältää solmut 3, 5, 6, 7 ja 8.

Binääripuun juuren *syvyys* (*depth*) on 0 ja jokaisen muun solmun syvyys on yhtä suurempi kuin sen vanhemman syvyys. Binääripuun *korkeus* (*height*) on puolestaan suurin puun solmussa esiintyvä syvyys eli toisin sanoen suurin askelten määrä juuresta alaspäin lehteen. Esimerkiksi kuvan 6.1 puun korkeus on 3, koska solmujen 7 ja 8 syvyys on 3.

Jos binääripuun korkeus on h , siinä on vähintään $h+1$ solmua, jolloin puu on pelkkä solmujen lista, ja enintään $2^{h+1} - 1$ solmua, jolloin kaikilla tasoilla on kaikki mahdolliset solmut. Kuva 6.2 näyttää esimerkit näistä tapauksista, kun puun korkeus on 2.

6.1.1 Binääripuun käsittely

Voimme toteuttaa binääripuun linkitettyinä rakenteena niin, että jokainen puun solmu on olio, jossa on viittaus vasempaan ja oikeaan lapseen sekä mahdollisesti muita kenttiä, kuten solmuun liittyvä arvo. Jos solmulla ei ole vasenta tai oikeaa lasta, viittauksena on `null`.

Rekursio on luonteva tapa toteuttaa monia binääripuun käsittelyyn liittyviä operaatioita. Esimerkiksi seuraava funktio laskee, montako solmua sille annetussa puussa on:

```
function laskeSolmut(solmu)
  if solmu == null
    return 0
  return 1 + laskeSolmut(solmu.vasen) +
    laskeSolmut(solmu.oikea)
```

Funktiolle annetaan parametrina solmu, joka vastaa puun juurta. Jos puu on tyhjä, siinä ei ole yhtään solmua. Muuten puussa on juurisolmu sekä vasemman ja oikean alipuun solmut. Pystymme laskemaan alipuiden solmut rekursiivisesti kutsumalla samaa funktiota uudestaan.

Seuraava funktio puolestaan selvittää, mikä on puun korkeus. Huomaa, että jos puu on tyhjä, tulkintana on, että sen korkeus on -1 .

```
function korkeus(solmu)
  if solmu == null
    return -1
  return 1 + max(korkeus(solmu.vasen), korkeus(solmu.oikea))
```

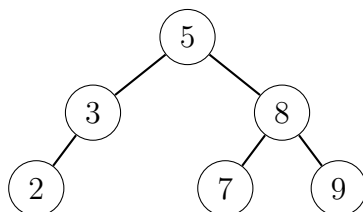
6.1.2 Läpikäyntijärjestykset

Voimme käydä läpi binääripuun solmut rekursiivisesti juuresta alkaen. Solmujen läpikäyntiin on kolme tavallista järjestystä:

- *esijärjestys (pre-order)*: käsittelemme ensin juuren, sitten vasemman alipuun ja lopuksi oikean alipuun
- *sisäjäjestys (in-order)*: käsittelemme ensin vasemman alipuun, sitten juuren ja lopuksi oikean alipuun
- *jälkijärjestys (post-order)*: käsittelemme ensin vasemman alipuun, sitten oikean alipuun ja lopuksi juuren

Esimerkiksi kuvan 6.1 puussa esijärjestys on $[1, 2, 4, 3, 5, 6, 7, 8]$, sisäjäjestys on $[4, 2, 1, 5, 3, 7, 6, 8]$ ja jälkijärjestys on $[4, 2, 5, 7, 8, 6, 3, 1]$.

Voimme käydä binääripuun solmut läpi kaikissa yllä mainituissa järjestyksissä rekursion avulla. Esimerkiksi seuraava proseduuri tulostaa puun solmut sisäjäjärjestyksessä, kun sille annetaan parametrina puun juuri:



Kuva 6.3: Joukkoa $\{2, 3, 5, 7, 8, 9\}$ vastaava binäärihakupuu.

```

procedure tulosta(solmu)
  if solmu == null
    return
  tulosta(solmu.vasen)
  print(solmu.arvo)
  tulosta(solmu.oikea)
  
```

6.2 Binäärihakupuun toiminta

Binäärihakupuu on binääripuu, jonka kukin solmu vastaa yhtä joukon alkia. Solmut on järjestetty niin, että jokaisessa solmussa kaikki vasemman alipuun solmut ovat arvoltaan pienempiä ja vastaavasti kaikki oikean alipuun solmut ovat arvoltaan suurempia. Tämän ansiosta voimme löytää kätevästi halutun alkion puusta aloittamalla haun puun juuresta.

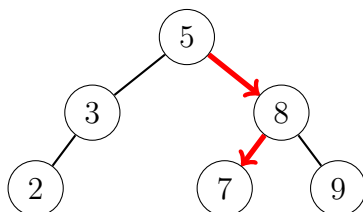
Kuvassa 6.3 on esimerkkinä joukkoa $\{2, 3, 5, 7, 8, 9\}$ vastaava binäärihakupuu, jonka juurena on alkio 5. Vasemmassa alipuussa on kaikki alkioita 5 pienemmät alkio, eli se vastaa joukkoa $\{2, 3\}$. Oikeassa alipuussa taas on kaikki alkioita 5 suuremmat alkio, eli se vastaa joukkoa $\{7, 8, 9\}$. Huomaa, että tämä on yksi monista tavoista muodostaa binäärihakupuu kyseiselle joukolle ja voisimme valita myös minkä tahansa muun alkion puun juureksi.

6.2.1 Operaatioiden toteutus

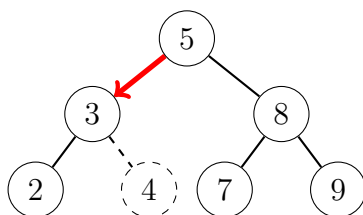
Seuraavaksi käymme läpi, kuinka voimme toteuttaa binäärihakupuun avulla operaatioita joukon alkioden käsittelemiseen. Osoittautuu, että voimme toteuttaa kaikki operaatiot ajassa $O(h)$, missä h on puun korkeus.

Alkion etsiminen

Kun haluamme etsiä joukosta alkioita x , lähdemme liikkeelle puun juuresta ja kuljemme alaspäin puussa. Kun olemme solmussa, jossa on alkio a , vaih-



Kuva 6.4: Alkion 7 etsiminen joukosta $\{2, 3, 5, 7, 8, 9\}$ juuresta alkaen.



Kuva 6.5: Alkion 4 lisääminen joukkoon $\{2, 3, 5, 7, 8, 9\}$.

toehtoja on kolme. Jos $a = x$, olemme löytäneet halutun alkion, jos $a > x$, jatkamme hakua solmun vasempaan lapseen, ja jos $a < x$, jatkamme hakua solmun oikeaan lapseen. Jos kuitenkin solmulla ei ole lasta, johon meidän tulisi edetä, toteamme, ettei joukossa ole alkioita x .

Kuva 6.4 näyttää, kuinka löydämme alkion 7 joukosta $\{2, 3, 5, 7, 8, 9\}$. Juurena on alkio 5, joten alkion 7 täytyy olla juuren oikeassa alipuussa. Tämän alipuun juurena on alkio 8, joten nyt taas tiedämme, että alkion 7 täytyy olla vasemmassa alipuussa, josta se löytyykin.

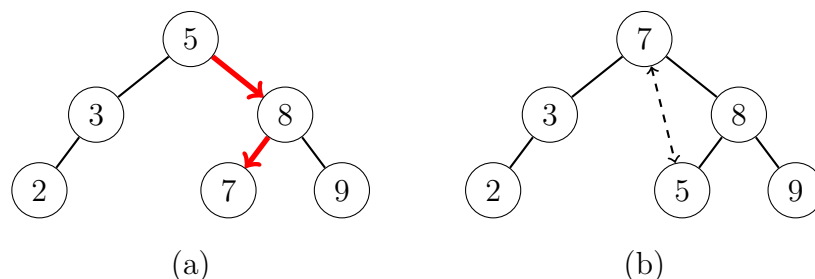
Alkion lisääminen

Kun haluamme lisätä joukkoon alkion x , jota ei vielä ole joukossa, kuljemme ensin puussa aivan kuin etsisimme alkioita x . Sitten kun olemme päässeet solmuun, jolla ei ole lasta, johon meidän tulisi edetä, luomme uuden solmun alkioille x ja lisäämme sen tähän kohtaan lapseksi.

Kuva 6.5 näyttää, kuinka lisäämme alkion 4 joukkoon $\{2, 3, 5, 7, 8, 9\}$. Kun haemme puusta alkioita 4, päädyimme solmuun, jossa on alkio 3 ja jolla ei ole oikeaa lasta. Niinpä luomme alkioille 4 uuden solmun, jonka asetamme alkion 3 solmun oikeaksi lapseksi.

Pienin alkio / suurin alkio

Kun haluamme löytää joukon pienimmän alkion, lähdemme liikkeelle juuresta ja etenemme joka askeleella solmun vasempaan lapseen. Kun solmulla ei



Kuva 6.6: Alkion 5 poistaminen joukosta $\{2, 3, 5, 7, 8, 9\}$. (a) Koska alkiolla 5 on kaksi lasta, etsimme seuraavan suuremman alkion 7. (b) Vaihdamme keskenään alkiot 5 ja 7, minkä jälkeen voimme poistaa helposti alkion 5.

ole enää vasenta lasta, olemme löytäneet joukon pienimmän alkion.

Vastaavalla tavalla löydämme joukon suurimman alkion etenemällä koko ajan oikeaan lapseen juuresta.

Seuraava suurempi alkio / edellinen pienempi alkio

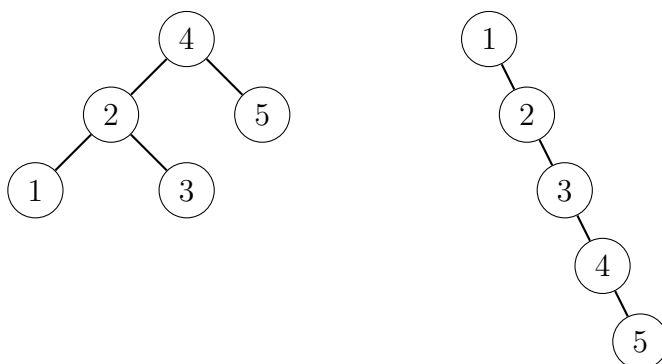
Kun haluamme löytää joukon pienimmän alkion, joka on suurempi kuin x , lähdemme liikkeelle puun juuresta. Kun olemme solmussa, jossa on alkio a , etenemme vasempaan lapseen, jos $a > x$, ja oikeaan lapseen, jos $a \leq x$. Jatkamme näin, kunnes emme voi edetä alemmas. Haluttu alkio on pienin alkiota x suurempi alkio kaikista alkioista, joiden kautta kuljimme.

Kun haluamme vastaavasti löytää joukon suurimman alkion, joka on pienempi kuin x , menettelemme käänteisesti edelliseen nähden.

Alkion poistaminen

Kun haluamme poistaa joukosta alkion x , etsimme ensin alkiota x vastaavan solmun tavalliseen tapaan. Jos solmulla ei ole lapsia tai vain yksi lapsi, meidän on helppoa poistaa solmu puusta ja säilyttää puun rakenne muuten ennallaan. Jos kuitenkin solmulla on kaksi lasta, tilanne on hankalampi. Tällöin etsimme alkion y , joka on pienin x :ää suurempi alkio, ja vaihdamme keskenään alkiot x ja y puussa. Tämän jälkeen meidän on helppoa poistaa solmu, jossa on nyt alkio x , koska sillä ei voi olla kahta lasta (jos solmulla olisi vasen lapsi, y ei olisi pienin x :ää suurempi alkio).

Kuva 6.6 näyttää, kuinka poistamme joukosta $\{2, 3, 5, 7, 8, 9\}$ alkion 5. Alkio on puun juuresta ja solmulla on kaksi lasta, joten meidän tulee etsiä ensin pienin alkiota 5 suurempi alkio, joka on 7. Vaihdamme sitten keskenään arvot 5 ja 7, minkä jälkeen meidän on helppoa poistaa alkio 5.



Kuva 6.7: Kaksi binäärihakupuuta joukolle $\{1, 2, 3, 4, 5\}$. Vasemman puun korkeus on 2 ja oikean puun korkeus on 4.

6.2.2 Operaatioiden tehokkuus

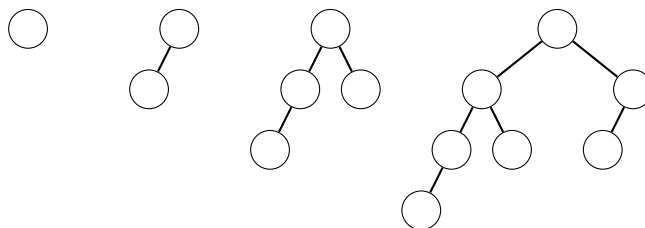
Binäärihakupuun operaatiot vievät aikaa $O(h)$, missä h on puun korkeus, joten operaatioiden tehokkuus riippuu puun korkeudesta. Operaatioiden tehokkuuteen vaikuttaa siis, miten olemme rakentaneet puun. Esimerkiksi kuvassa 6.7 on kaksi mahdollista binäärihakupuuta joukolle $\{1, 2, 3, 4, 5\}$. Vasemman puun korkeus on 2, kun taas oikean puun korkeus on 4.

Jotta binäärihakupuu toimisi tehokkaasti, haluamme, että puun korkeus ei kasva liian suureksi. Tarkemmin ottaen tavoittemme on, että solmut ovat jakautuneet tasaisesti puun eri puolille ja puun korkeus on $O(\log n)$. Tällöin sanomme, että puu on *tasapainoinen* (*balanced*). Jos onnistumme tässä, kaikki puun operaatiot toimivat tehokkaasti ajassa $O(\log n)$. Saavutamme tavoittemme lisäämällä puuhun ehtoja, jotka rajoittavat sen korkeutta sopivasti.

Binäärihakupuun tasapainottamiseen tunnetaan monia menetelmiä. Tutustumme seuraavaksi AVL-puuhun, joka on varhaisin tunnettu tasapainoinen binäärihakupuu. AVL-puu on yksinkertaisempi kuin monet myöhemmin kehitetyt rakenteet, minkä vuoksi se sopii hyvin esittelemään puiden tasapainotuksen ideoita. Javan ja muiden ohjelmointikielten standardikirjastoissa käytetään kuitenkin muita rakenteita, kuten punamustaa puuta.

6.3 AVL-puu

AVL-puu (*AVL tree*) on tasapainoinen binäärihakupuu, jonka korkeus on aina $O(\log n)$, minkä ansiosta puun operaatiot toimivat tehokkaasti ajassa $O(\log n)$. AVL-puussa jokaiseen solmuun liittyy *tasapainoehto*, joka takaa, että puu on tasapainoinen. Kun päivitämme puuta, meidän täytyy pitää huolta siitä, että tasapainoehto säilyy voimassa kaikissa solmuissa.



Kuva 6.8: Vähiten solmuja sisältävät AVL-puut korkeuksille 0, 1, 2 ja 3.

6.3.1 Tasapainoehto

AVL-puun tasapainoehtona on, että *jokaisessa solmussa vasemman ja oikean lapsen alipuiden korkeusero on enintään 1*.

Esimerkiksi kuvan 6.7 vasen puu on AVL-puu, kun taas oikea puu ei ole. Oikea puu ei ole AVL-puu, koska esimerkiksi solmussa 1 vasemman lapsen alipuun korkeus on -1 mutta oikean lapsen alipuun korkeus on 3. Korkeuksien erona on siis 4, vaikka ero saisi olla enintään 1.

Kutsumme AVL-puun tasapainoehtoa *AVL-ehdoksi*. Osoittautuu, että jos binäärihakupuu täyttää AVL-ehdon, sen korkeus on $O(\log n)$. Eli jos pystymme toteuttamaan puun operaatiot niin, että AVL-ehto säilyy, saamme aikaan binäärihakupuun, jonka operaatiot toimivat ajassa $O(\log n)$.

Miksi sitten AVL-ehto takaa, että binäärihakupuun korkeus on $O(\log n)$? Voimme lähestyä asiaa pahimman tapauksen kautta: kun tiedämme, että AVL-puussa on n solmua, mikä on sen *suurin mahdollinen* korkeus? Voimme selvittää tämän laskemalla ensin käänteisesti, mikä on *pienin mahdollinen* solmujen määrä AVL-puussa, jonka korkeus on h .

Merkitään $f(h)$:lla korkeutta h olevan AVL-puun pienintä mahdollista solmujen määrää. Kuvan 6.8 mukaisesti funktion ensimmäiset arvot ovat $f(0) = 1$, $f(1) = 2$, $f(2) = 4$ ja $f(3) = 7$. Yleisemmin

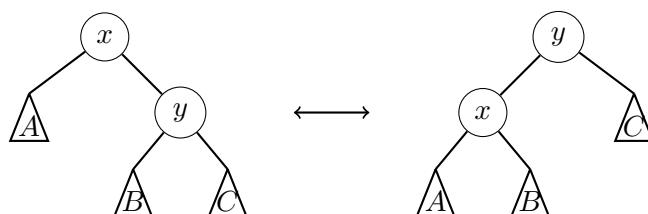
$$f(h) = 1 + f(h-1) + f(h-2),$$

kun $h \geq 2$, koska jos haluamme rakentaa AVL-puun korkeutta h , jossa on mahdollisimman vähän solmuja, meidän kannattaa laittaa juuren lapsiksi AVL-puut korkeutta $h-1$ ja $h-2$ niin, että kummassakin alipuussa on mahdollisimman vähän solmuja. Funktiolle pätee

$$f(h) \geq 2f(h-2),$$

eli funktion arvo ainakin kaksinkertaistuu kahden askeleen välein. Voimme ilmaista tämän alarajan

$$f(h) \geq 2^{h/2},$$



Kuva 6.9: Kierrot, joiden avulla korjaamme AVL-puuta.

jonka voimme taas muuttaa ylärajaksi

$$h \leq 2 \log f(h).$$

Tarkastellaan sitten puuta, jossa on n solmua ja jonka korkeus on h . Korkeudelle täytyy päteä $f(h) \leq n$, koska korkeutta h olevassa puussa on vähintään $f(h)$ solmua. Niinpä saamme ylärajan

$$h \leq 2 \log n,$$

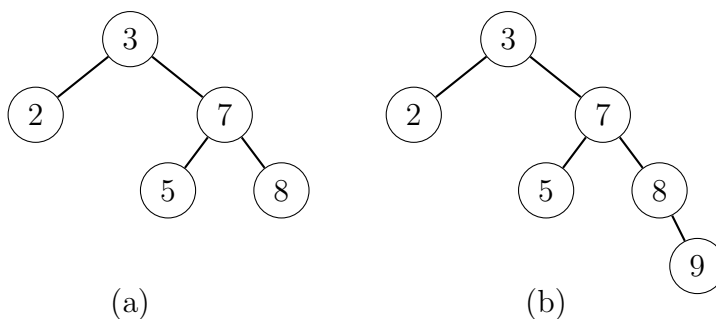
mikä tarkoittaa samaa kuin $h = O(\log n)$.

6.3.2 Kiertojen toteuttaminen

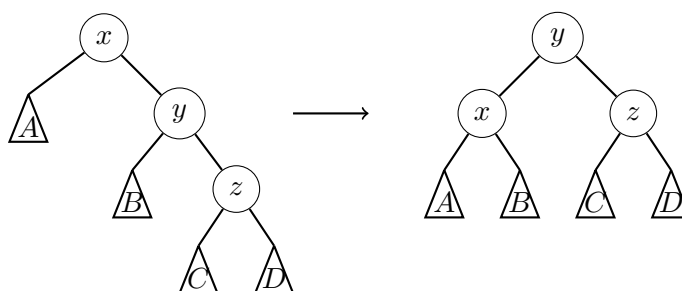
Voimme toteuttaa AVL-puun operaatiot muuten samaan tapaan kuin yleisessä binäärihakupuussa, mutta meidän täytyy varmistaa alkion lisäämisen ja poistamisen jälkeen, että AVL-ehto on edelleen voimassa. Tämä onnistuu tekemällä sopivia *kiertoja* (*rotation*), jotka muuttavat puun rakennetta. Jotta voimme toteuttaa kierrot, pidämme jokaisessa solmussa tietoa siitä, mikä on solmusta alkavan alipuun korkeus.

Osoittautuu, että voimme korjata puun rakenteen kaikissa tilanteissa käyttäen kahta kiertotyyppiä, jotka on esitetty kuvassa 6.9. Kierrämme solmuja x ja y , joihin liittyvät alipuut A , B ja C . Voimme tehdä kierron joko vasemmalta oikealle, jolloin solmu y nousee ylöspäin, tai oikealta vasemmalle, jolloin solmu x nousee ylöspäin.

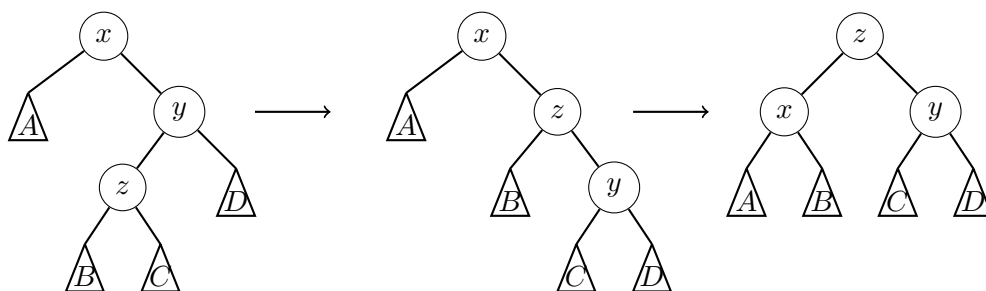
Kun lisäämme AVL-puuhun solmun, jonkin solmun AVL-ehto voi rikkoontua. Tämä ilmenee niin, että jossain solmussa lasten alipuiden korkeudet ovat ennen lisäämistä h ja $h + 1$ ja lisäämisen jälkeen h ja $h + 2$. Kuva 6.10 näyttää esimerkin tällaisesta tilanteesta. Vasemmassa puussa solmun 3 lasten korkeudet ovat 0 ja 1, joten AVL-ehto on kunnossa. Oikeassa puussa olemme lisänneet solmun 9, minkä seurauksena solmun 3 lasten korkeudet ovat 0 ja 2 eikä AVL-ehto enää päde.



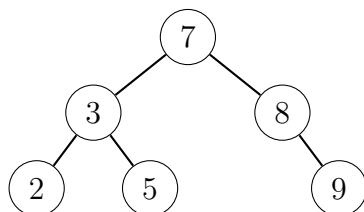
Kuva 6.10: (a) Jokaisessa puun solmussa pätee AVL-ehto. (b) AVL-ehto menee rikki solmussa 3, kun lisäämme puuhun solmun 9.



Kuva 6.11: Tapaus 1: nostamme solmua y ylöspäin.



Kuva 6.12: Tapaus 2: nostamme solmua z kahdesti ylöspäin.



Kuva 6.13: Kierrämme solmun 7 puun juureksi, jolloin AVL-ehto pätee taas.

Solmun lisäämisen jälkeen kuljemme puussa ylöspäin lisäystä solmusta juureen ja päivitämme solmujen korkeudet. Jos jokin solmu ei täytä AVL-ehtoa, korjaamme asian tekemällä yhden tai kaksi kiertoa. Oletetaan, että x on alimpana puussa oleva solmu, jossa AVL-ehto ei päde, y on x :n lapsi, jonka alipuussa on lisätty solmu, ja z on puolestaan y :n lapsi, jonka alipuussa on lisätty solmu. Tapauksia on kaksi: jos y ja z ovat samanpuoleisia lapsia, kierrämme solmua y kerran ylöspäin (kuva 6.11), ja muuten kierrämme solmua z kahdesti ylöspäin (kuva 6.12). Tämän korjauksen jälkeen AVL-ehto on jälleen voimassa kaikissa puun solmuissa, eli meidän riittää aina korjata ehto alimmassa solmussa, jossa se ei ole voimassa.

Kuvassa 6.10(b) AVL-ehto ei ole voimassa solmussa 3, koska vasemman alipuun korkeus on 0 ja oikean alipuun korkeus on 2. Tässä tapauksessa $x = 3$, $y = 7$ ja $z = 8$. Koska y on x :n oikea lapsi ja z on y :n oikea lapsi, meidän riittää tehdä yksi kierto, joka nostaa solmua 7 ylöspäin puun juureksi. Tuloksena on kuvan 6.13 mukainen puu, jossa AVL-ehto on jälleen voimassa, koska solmun 7 kummankin alipuun korkeus on nyt 1.

Kun poistamme puusta solmun, menettelemme melko samalla tavalla kuin lisäämisessä. Nyt on mahdollista, että ennen poistoa jossakin solmussa lasten alipuiden korkeudet ovat h ja $h - 1$ ja poiston jälkeen h ja $h - 2$. Poiston jälkeen nousemme puussa ylöspäin poistetun solmun vanhemmas- ta alkaen, ja jos vastaan tulee solmu x , jossa AVL-ehto ei päde, korjaamme asian. Tällä kertaa valitsemme solmut y ja z niin, että y on x :n lapsi, jonka korkeus on suurin, ja samoin z on y :n lapsi, jonka korkeus on suurin. Jos y :n kummankin lapsen korkeus on sama, valitsemme z :n niin, että se on samanpuoleinen lapsi kuin y . Sitten korjaamme AVL-ehdon kierroilla kuten solmun lisäämisessä. Toisin kuin lisäämisessä, saatamme joutua korjaamaan ehdon useassa solmussa, koska ehdon korjaaminen yhdessä solmussa voi rikkoa sen jossain ylemmässä solmussa. Kun lopulta saavumme juureen, AVL-ehto pätee jälleen kaikissa solmuissa.

Koska AVL-puun korkeus on $O(\log n)$ ja jokainen kierto tapahtuu vakio- ajassa, pystymme korjaamaan tasapainon sekä lisäämisen että poistamisen jälkeen ajassa $O(\log n)$. Lisäämisen jälkeen kuljemme puuta ylöspäin $O(\log n)$

askelta ja teemme enintään kaksi kiertoa. Poistamisen jälkeen taas kuljemme puuta ylöspäin $O(\log n)$ askelta ja teemme enintään $O(\log n)$ kiertoa.

6.4 Javan toteutukset

Javan tietorakenteet `TreeSet` ja `TreeMap` pohjautuvat punamustaan puuhun, joka on AVL-puun tapainen mutta monimutkaisempi binäärihakupuu. Ne muistuttavat rakenteita `HashSet` ja `HashMap`, mutta erona on, että pystymme lisäksi etsimään alkioita niiden järjestyksen perusteella. Rakenteiden operaatiot toimivat ajassa $O(\log n)$.

6.4.1 TreeSet-rakenne

Seuraava koodi luo `TreeSet`-rakenteen, joka pitää yllä lukujen joukkoa. Koodi lisää joukkoon alkioita ja tulostaa sitten sen sisällön. Huomaa, että tulostuksessa joukon alkiot näkyvät järjestyksessä pienimmästä suurimpaan, koska binäärihakupuu pitää niitä järjestyksessä.

```
TreeSet<Integer> joukko = new TreeSet<>();
joukko.add(4);
joukko.add(1);
joukko.add(8);
joukko.add(7);
System.out.println(joukko); // [1, 4, 7, 8]
```

Koska joukko on järjestyksessä, pystymme etsimään tehokkaasti pienimmän ja suurimman alkion metodeilla `first` ja `last`:

```
System.out.println(joukko.first()); // 1
System.out.println(joukko.last()); // 8
```

Pystymme myös etsimään seuraavan tiettyä alkioita suuremman tai pienemmän alkion metodeilla `higher` ja `lower`:

```
System.out.println(joukko.higher(5)); // 7
System.out.println(joukko.lower(5)); // 4
```

6.4.2 TreeMap-rakenne

Seuraava koodi luo sanakirjan `TreeMap`-rakenteen avulla:

```
TreeMap<String,String> sanakirja = new TreeMap<>();  
sanakirja.put("apina","monkey");  
sanakirja.put("banaani","banana");  
sanakirja.put("cembalo","harpsichord");
```

Tämä sanakirja muodostuu avain-arvo-pareista, joissa avaimet ovat suomen kielen sanoja ja arvot ovat englannin kielen sanoja. Sanakirjan sisältö on järjestetty avaimien perusteella. Esimerkiksi voimme selvittää, mikä on aakkosjärjestyksessä ensimmäinen ja viimeinen avain:

```
System.out.println(sanakirja.firstKey()); // apina  
System.out.println(sanakirja.lastKey()); // cembalo
```

Samoin voimme selvittää lähinnä tiettyä avainta olevat avaimet:

```
System.out.println(sanakirja.higherKey("biisoni")); // cembalo  
System.out.println(sanakirja.lowerKey("biisoni")); // banaani
```

6.4.3 Omat luokat

Jos haluamme käyttää omia olioitamme `TreeSet`- ja `TreeMap`-rakenteissa, meidän tulee toteuttaa luokkaan kaksi metodia: `equals` ilmaisee, ovatko kaksi oliota samat, ja `compareTo` kertoo kahden olion suuruusjärjestyksen. Jälkimmäisen metodin ansiosta luokka toteuttaa rajapinnan `Comparable`.

Miksi oikeastaan metodi `equals` on tarpeen? Metodi `compareTo` palauttaa arvon 0 tarkalleen silloin, kun alkiot ovat yhtä suuret, joten myös sen avulla voi tarkastaa asian. Syynä on, että `TreeSet` toteuttaa rajapinnan `Set`, joka edellyttää, että luokassa on oikein toimiva metodi `equals`, vaikka siitä ei sinänsä olekaan hyötyä tässä tapauksessa.

6.5 Tehokkuusvertailu

Monissa ongelmissa meillä on kaksi mahdollista lähestymistapaa: voimme käyttää joko joukkorakenteita tai sitten taulukoita ja järjestämistä. Vaikka molemmat tavat johtavat tehokkaaseen ratkaisuun, vakiokertoimissa voi olla merkittäviä eroja, jotka vaikuttavat käytännön tehokkuuteen.

Tarkastelemme seuraavaksi ongelmaa, jossa meille on annettu n lukua sisältävä taulukko, ja haluamme selvittää, montako eri lukua taulukossa on. Ratkaisemme ongelman kolmella eri tavalla ja tutkimme sitten ratkaisujen tehokkuutta.

Ratkaisu 1: TreeSet

Ensimmäinen tapa ratkaista tehtävä on luoda `TreeSet`, johon lisäämme taulukon luvut. Koska jokainen luku voi esiintyä joukossa vain kerran, joukon koko ilmaisee meille, montako eri lukua taulukossa on. Tämä ratkaisu vie aikaa $O(n \log n)$, koska jokainen `add`-operaatio vie aikaa $O(\log n)$.

```
TreeSet<Integer> joukko = new TreeSet<>();
for (int i = 0; i < taulu.length; i++) {
    joukko.add(taulu[i]);
}
System.out.println(joukko.size());
```

Ratkaisu 2: HashSet

Emme tarvitse `TreeSet`-rakenteen alkioiden järjestystä, joten saamme toisen ratkaisun käyttämällä sen sijaan `HashSet`-rakennetta. Koodi säilyy muuten täysin samanlaisena. Tämä ratkaisu vie aikaa $O(n)$ hajautuksen ansiosta.

```
HashSet<Integer> joukko = new HashSet<>();
for (int i = 0; i < taulu.length; i++) {
    joukko.add(taulu[i]);
}
System.out.println(joukko.size());
```

Ratkaisu 3: järjestäminen

Kolmas tapa ratkaista tehtävä on käyttää järjestämistä: kopioimme ensin luvut uuteen taulukkoon, järjestämme tämän taulukon ja tutkimme sitten, monessako kohdassa järjestetyssä taulukossa luku vaihtuu. Tämä ratkaisu vie aikaa $O(n \log n)$, koska taulukon järjestäminen vie aikaa $O(n \log n)$.

```
int[] kopio = taulu.clone();
Arrays.sort(kopio);
int laskuri = 1;
for (int i = 1; i < kopio.length; i++) {
    if (kopio[i-1] != kopio[i]) laskuri++;
}
System.out.println(laskuri);
```


taulukon koko n	TreeSet	HashSet	järjestäminen
10^6	0.74 s	0.25 s	0.09 s
$2 \cdot 10^6$	1.60 s	0.45 s	0.19 s
$4 \cdot 10^6$	5.60 s	1.56 s	0.52 s
$8 \cdot 10^6$	12.19 s	4.50 s	0.97 s

Taulukko 6.1: Algoritmien suoritusaikojen vertailu.

Vertailun tulokset

Taulukko 6.1 esittää tehokkuusvertailun tulokset. Jokaisessa testissä taulukossa on satunnaisia lukuja väliltä $1 \dots 10^9$.

Osoittautuu, että ratkaisujen välillä on merkittäviä tehokkuuseroja. Ensimmäkin **HashSet**-ratkaisu on noin kolme kertaa nopeampi kuin **TreeSet**-ratkaisu. Tämä onkin odotettavaa, koska hajautustaulun operaatiot vievät aikaa $O(1)$, kun taas binäärihakupuun operaatiot vievät aikaa $O(\log n)$. Selvästi nopein ratkaisu on kuitenkin kolmas järjestämistä käyttävä ratkaisu, joka on noin kymmenen kertaa **TreeSet**-ratkaisua nopeampi.

Miten on mahdollista, että sekä **TreeSet**-ratkaisun että järjestämisratkaisun aikavaativuus on $O(n \log n)$, mutta järjestämisratkaisu on kymmenen kertaa nopeampi? Tämä johtuu siitä, että taulukon järjestäminen on hyvin kevyt operaatio ja se tehdään vain kerran. Binäärihakupuussa jokainen lisäys muuttaa puun rakennetta, mikä aiheuttaa suuret vakiokertoimet.

Vaikka joukot ovat käteviä, niitä ei siis kannata käyttää turhaan. Jos haluamme ratkaista ongelman todella tehokkaasti, kannattaa miettiä, voisimmeko käyttää tavalla tai toisella järjestämistä joukkojen sijaan.

Luku 7

Keko

Keko (*heap*) on tietorakenne, jonka operaatiot ovat alkion lisääminen sekä pienimmän tai suurimman alkion etsiminen ja poistaminen. Vaikka voisimme toteuttaa nämä operaatiot myös binäärihakupuun avulla, keon etuna on, että saamme aikaan yleistä joukkorakennetta *kevyemmän* rakenteen, kun rajoitumme tilanteeseen, jossa käytössämme on vain nämä operaatiot.

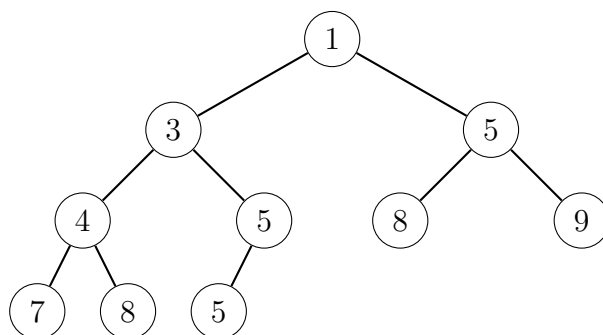
Tutustumme tässä luvussa binäärikeko-rakenteeseen, joka on tavallisimmin käytetty kekorakenne. Binäärikeko toteutetaan binääripuuna, ja se mahdollistaa alkion etsimisen ajassa $O(1)$ sekä alkioden lisäykset ja poistot ajassa $O(\log n)$. Javassa voimme käyttää tietorakennetta `PriorityQueue`, joka perustuu binäärikekoon.

7.1 Binäärikeko

Binäärikeko (*binary heap*) on binääripuu, jonka kaikki tasot alinta tasoa lukuun ottamatta ovat täynnä solmuja. Alimman tason solmut on puolestaan sijoitettu mahdollisimman vasemmalle ylempien solmujen lapsiksi.

Kun luomme keon, meidän täytyy päättää, onko se *minimikeko* vai *maksimikeko*. Minimikeossa voimme etsiä ja poistaa pienimmän alkion, kun taas maksimikeossa voimme etsiä ja poistaa suurimman alkion. Keon toiminta perustuu siihen, että jokainen keon solmu täyttää *kekoehdon*. Minimikeossa ehtona on, että jokaisen solmun arvo on pienempi tai yhtä suuri kuin sen lapsen arvo. Maksimikeossa puolestaan jokaisen solmun arvo on suurempi tai yhtä suuri kuin sen lapsen arvo. Kekoehdon ansiosta minimikeon juuressa on keon pienin alkio ja maksimikeon juuressa on keon suurin alkio.

Kuvassa 7.1 on minimikeko, johon on tallennettu kymmenen alkiota. Keon kolme ensimmäistä tasoa ovat täynnä ja neljännellä tasolla kolme ensimmäistä kohtaa on käytetty. Keon juurena on joukon pienin alkio 1, ja



Kuva 7.1: Minimikeko, joka sisältää alkiot $[1, 3, 4, 5, 5, 5, 7, 8, 8, 9]$.

kaikki solmut täyttävät kekoehdon. Huomaa, että sama alkio voi esiintyä monta kertaa keossa, kuten tässä keossa alkio 5 ja 8.

7.1.1 Keon tallentaminen

Tallennamme binäärikeon *taulukkona*, joka sisältää keon solmujen arvot järjestyksessä ylhäältä alaspäin ja vasemmalta oikealle. Tämä tehokas tallennustapa on mahdollinen, koska keon kaikki tasot ovat täynnä solmuja. Tallennamme keon taulukkoon kohdasta 1 alkaen, koska tämä helpottaa keon operaatioiden toteuttamista.

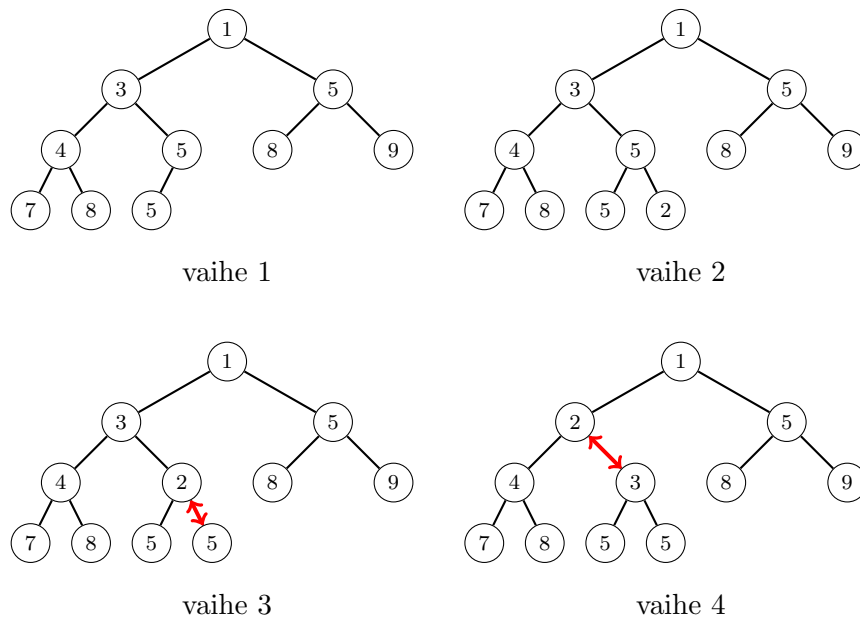
Esimerkiksi tallennamme kuvan 7.1 keon taulukkona seuraavasti:

keko = $[0, 1, 3, 5, 4, 5, 8, 9, 7, 8, 5]$

Huomaa, että taulukon ensimmäinen alkio on 0, koska emme käytä kohtaa 0 keon tallentamiseen.

Taulukkototeutuksen etuna on, että voimme laskea helposti, missä kohdissa keon alkio on taulukossa. Ensinnäkin keon juuri eli pienin tai suurin alkio on aina kohdassa 1. Lisäksi jos tiedämme, että tietty solmu on kohdassa k , niin solmun vasen lapsi on kohdassa $2k$, solmun oikea lapsi on kohdassa $2k + 1$ ja solmun vanhempi on kohdassa $\lfloor k/2 \rfloor$. Esimerkissämme solmu 3 on taulukossa kohdassa 2, joten sen vasen lapsi on kohdassa 4, oikea lapsi on kohdassa 5 ja vanhempi on kohdassa 1.

Käytännössä haluamme yleensä, että pystymme lisäämään keoon uusia alkioita, jolloin saattaa olla tarpeen suurentaa taulukkoa. Voimme toteuttaa tämän samalla tavalla kuin taulukkolistassa, jolloin taulukon suurentaminen ei hidasta keon operaatioita.



Kuva 7.2: Lisäämme alkion 2 kekoon ja nostamme sitä ylöspäin, kunnes kekoehto tulee jälleen voimaan.

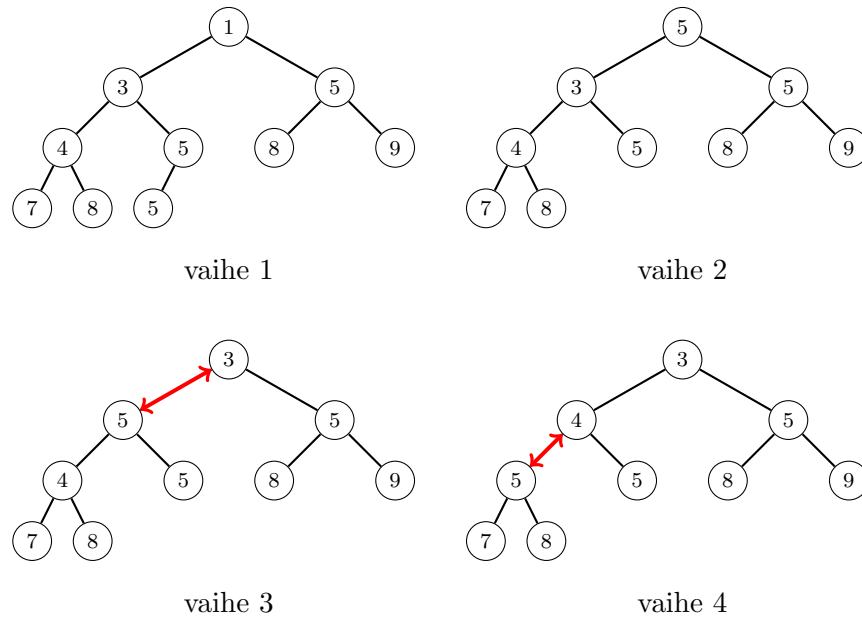
7.1.2 Operaatioiden toteutus

On helppoa etsiä minimikeon pienin alkio tai maksimikeon suurin alkio $O(1)$ -ajassa, koska tämä alkio on aina keon juuressa. Seuraavaksi näemme, kuinka voimme toteuttaa alkion lisäämisen sekä pienimmän tai suurimman alkion poistamisen $O(\log n)$ -ajassa.

Alkion lisääminen

Kun lisäämme uuden alkion kekoon, lisäämme sen ensin seuraavaan vapaana olevaan paikkaan puussa. Jos alimmalla tasolla on tilaa, lisäämme sen sinne mahdollisimman vasemmalle, ja muuten aloitamme uuden tason, jossa on toistaiseksi vain lisättävä solmu. Alkion lisäämisen jälkeen meidän täytyy varmistaa, että kekoehto säilyy edelleen voimassa. Tämä tapahtuu siirtämällä alkioita ylöspäin keossa, kunnes kekoehto tulee voimaan.

Kuva 7.2 näyttää, mitä tapahtuu, kun lisäämme alkion 2 esimerkikekoomme. Lisäämme alkion ensimmäiseen vapaaseen kohtaan keon alimmalla tasolla. Koska alkio 2 on pienempi kuin sen vanhempi 5, vaihdamme nämä alkioit keskenään. Tämän jälkeen alkio 2 on pienempi kuin sen vanhempi 3, joten vaihdamme myös nämä alkioit keskenään. Nyt kekoehto on voimassa eikä meidän tarvitse enää tehdä muutoksia kekoon.



Kuva 7.3: Poistamme keon juuresta olevan alkion korvaamalla sen viimeisellä alkiolla ja laskettamalla sitä alaspäin puussa.

Alkion lisääminen kekkoon vie aikaa $O(\log n)$, koska keossa on $O(\log n)$ tasoa ja kuljemme aina ylöspäin keon pohjalta huippua kohden, kunnes olemme löytäneet alkiolle sopivan paikan keosta.

Alkion poistaminen

Kun haluamme poistaa keon juuresta olevan alkion, siirrämme ensin keon viimeisen alkion keon juureksi ja poistamme sille kuuluneen solmun. Tämän jälkeen lasketamme juureen nostettua alkiota alaspäin keossa, kunnes kekohto on jälleen voimassa kaikkialla. Koska solmulla voi olla kaksi lasta, voi olla kaksi vaihtoehtoa, kumman lapsista nostamme ylemmäs. Jos keko on minimikeko, valitsemme lapsen, jossa on pienempi arvo, ja jos keko on maksimikeko, valitsemme vastaavasti lapsen, jossa on suurempi arvo.

Kuva 7.3 näyttää, kuinka poistamme esimerkikekostamme pienimmän alkion eli juuresta olevan alkion 1. Aluksi korvaamme alkion 1 keon viimeisellä alkiolla 5 ja poistamme keosta alkiolle 5 kuuluneen solmun. Tämän jälkeen vaihdamme keskenään alkion 5 ja sen vasemman lapsen alkion 3, ja sitten vielä alkion 5 ja sen vasemman lapsen alkion 4. Tämän jälkeen kekohto on voimassa ja olemme onnistuneet poistamaan pienimmän alkion keosta.

Alkion poistaminen keosta vie aikaa $O(\log n)$, koska keossa on $O(\log n)$ tasoa ja kuljemme polkua alaspäin keon huipulta pohjaa kohden.

7.2 Prioriteettijono

Monissa ohjelmointikielissä kekoa vastaava tietorakenne tunnetaan nimellä *prioriteettijono* (*priority queue*). Näin on myös Javassa, jonka standardikirjastoon kuuluu tietorakenne `PriorityQueue`. Se on binäärikekoon perustuva tietorakenne, joka toteuttaa oletuksena minimikeon.

Seuraava koodi esittelee Javan prioriteettijonon käyttämistä. Metodi `add` lisää alkion jonoon, metodi `peek` hakee pienimmän alkion ja metodi `poll` hakee ja poistaa pienimmän alkion.

```
PriorityQueue<Integer> jono = new PriorityQueue<>();
jono.add(5);
jono.add(3);
jono.add(8);
jono.add(7);
System.out.println(jono.peek()); // 3
System.out.println(jono.poll()); // 3
System.out.println(jono.poll()); // 5
```

Jos haluamme luoda prioriteettijonon, joka onkin maksimikeko, voimme tehdä sen seuraavaan tapaan:

```
PriorityQueue<Integer> jono =
    new PriorityQueue<>(10, Collections.reverseOrder());
```

Tässä tilanteessa meidän täytyy antaa konstruktorille kaksi tietoa: keolle alussa muistista varattava tila (tässä 10) sekä alkioiden järjestämistapa (tässä käänteinen järjestys). Huomaa, että Java varaa keolle tarvittaessa myöhemmin uuden suuremman muistialueen, joten tämä määrittely ei tarkoita, että keossa voisi olla enintään 10 alkioita.

Jos haluamme tallentaa `PriorityQueue`-rakenteeseen omia olioitamme, meidän tulee toteuttaa luokkaan metodi `compareTo` ja merkitä, että luokka toteuttaa rajapinnan `Comparable`.

7.3 Tehokkuusvertailu

Mitä hyötyä keosta oikeastaan on? Meillähän on olemassa jo binäärihakupuuh, jonka avulla voimme toteuttaa kaikki keon operaatiot ja *enemmänkin*. Keossa voimme hakea ja poistaa vain pienimmän tai suurimman alkion, mutta binäärihakupuussa voimme käsitellä myös muita alkioita.

Keon etuna on, että siinä on tehokkaan taulukkototeutuksen ansiosta pienemmät *vakiokertoimet* kuin binäärihakupuussa. Jos meille riittää, että

voimme hakea ja poistaa vain pienimmän tai suurimman alkion, voi siis olla hyvä ratkaisu käyttää kekoa binäärihakupuun sijasta. Mutta kuinka suuria erot ovat käytännössä?

Tästä antaa kuvaa seuraava testi, jossa vertailemme keskenään Javan tietorakenteita `PriorityQueue` ja `TreeSet`. Testissä meillä on taulukko, jossa on satunnaisessa järjestyksessä luvut $1, 2, \dots, n$. Lisäämme ensin taulukon $n/2$ ensimmäistä lukua joukkoon. Tämän jälkeen käymme läpi loput $n/2$ lukua, ja jokaisen luvun kohdalla lisäämme sen joukkoon ja poistamme joukon pienimmän luvun. Laskemme lisäksi samalla summaa poistetuista luvuista. Käytämme testissä seuraavia koodeja:

```
PriorityQueue<Integer> jono = new PriorityQueue<>();
for (int i = 0; i < n/2; i++) {
    jono.add(luvut[i]);
}
long summa = 0;
for (int i = n/2; i < n; i++) {
    jono.add(luvut[i]);
    summa += jono.poll();
}
System.out.println(summa);
```

```
TreeSet<Integer> joukko = new TreeSet<>();
for (int i = 0; i < n/2; i++) {
    joukko.add(luvut[i]);
}
long summa = 0;
for (int i = n/2; i < n; i++) {
    joukko.add(luvut[i]);
    summa += jono.pollFirst();
}
System.out.println(summa);
```

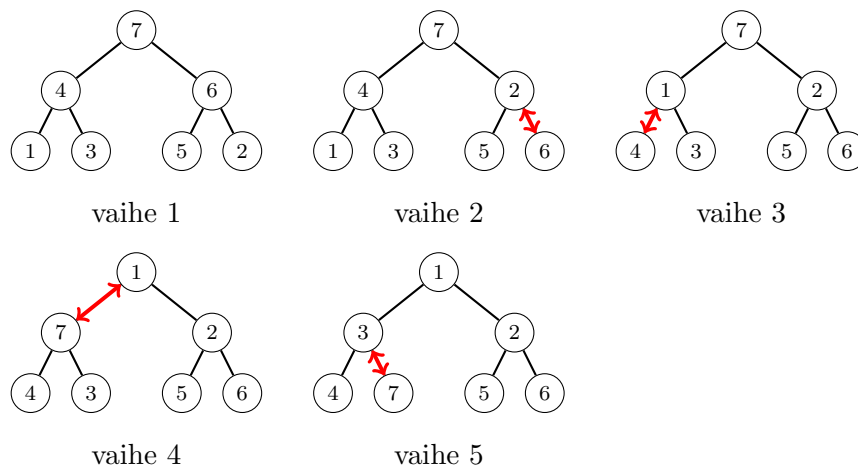
Taulukko 7.1 näyttää testin tulokset. Tämän testin perusteella näyttää siltä, että keon käyttämisestä on todellista hyötyä, koska `PriorityQueue` toimii 2–3 kertaa nopeammin kuin `TreeSet`.

7.4 Lisää keosta

Koska keko on tallennettu taulukkona, voimme tulkita minkä tahansa taulukon kekona, kunhan vain kekoehto on voimassa taulukon kaikissa kohdissa.

taulukon koko n	PriorityQueue	TreeSet
10^6	0.29 s	0.78 s
$2 \cdot 10^6$	0.71 s	1.50 s
$4 \cdot 10^6$	1.56 s	3.72 s
$8 \cdot 10^6$	3.68 s	9.43 s

Taulukko 7.1: Algoritmien suoritusaikojen vertailu.



Kuva 7.4: Muutamme taulukon keoksi korjaamalla kekoehdon alipuissa.

Käymme seuraavaksi läpi menetelmän, jonka avulla voimme *muuttaa* taulukon keoksi $O(n)$ -ajassa. Tämän jälkeen tutustumme kekojärjestämiseen, joka on $O(n \log n)$ -aikainen järjestämisalgoritmi.

7.4.1 Taulukosta keoksi

Oletetaan, että meillä on n alkioita sisältävä taulukko ja haluamme muuttaa sen keoksi. Suoraviivainen tapa on luoda tyhjä keko ja lisätä jokainen taulukon alkio siihen erikseen $O(\log n)$ -ajassa. Tällä tavalla saamme rakennettua keon $O(n \log n)$ -ajassa. Osoittautuu kuitenkin, että pystymme myös muuttamaan taulukon *suoraan* keoksi tehokkaammin ajassa $O(n)$.

Ideana on järjestää alkuperäisen taulukon alkioita uudestaan niin, että kekoehto tulee voimaan taulukon jokaiseen kohtaan – jolloin taulukko on muutunut keoksi. Käymme läpi taulukon alkiot lopusta alkuun ja varmistamme jokaisessa kohdassa, että kekoehto on voimassa kyseisestä kohdasta alkavassa alipuussa. Jos kekoehto ei ole voimassa, korjaamme sen laskettamalla kyseisen kohdan alkioita alaspäin keossa. Kun lopulta pääsemme taulukon alkuun, kekoehto on voimassa koko taulukossa.

Kuva 7.4 näyttää esimerkin, jossa muutamme taulukon $[7, 4, 6, 1, 3, 5, 2]$ minimikeoksi. Kun tulkitsemme taulukon kekona, kekoehdo on aluksi rikki monessa taulukon kohdassa. Ensin korjaamme kekoehdon tason 2 alipuissa vaihtamalla keskenään alkio 2 ja 6 ja sitten alkio 1 ja 4. Tämän jälkeen korjaamme kekoehdon tason 1 alipuissa eli koko keossa laskettamalla alkion 7 keon huipulta pohjalle. Nyt kekoehdo on voimassa kaikkialla taulukossa, joten olemme onnistuneet muuttamaan taulukon keoksi.

Miksi sitten tämä vie aikaa vain $O(n)$? Oletetaan, että keossa on h tasoa ja kaikki tasot ovat täynnä solmuja, eli keossa on $n = 2^h - 1$ solmua. Laskemme jokaiselle tasolle, montako alkioita laskeutuu enintään jostakin tämän tason solmusta alaspäin. Ensinnäkin tasolta 1 tasolle 2 laskeutuu enintään 1 alkio – juuressa oleva alkio. Vastaavasti tasolta 2 tasolle 3 laskeutuu enintään $1 + 2$ alkioita ja tasolta 3 tasolle 4 laskeutuu enintään $1 + 2 + 4$ alkioita. Yleisemmin tasolta k tasolle $k+1$ laskeutuu enintään $1 + 2 + \dots + 2^{k-1} = 2^k - 1$ alkioita. Koska tasoja on h ja alimmalta tasolta ei voi laskeutua alaspäin, kokonaistyömäärä on enintään

$$(2^1 - 1) + (2^2 - 1) + \dots + (2^{h-1} - 1) = 2^h - h \leq n,$$

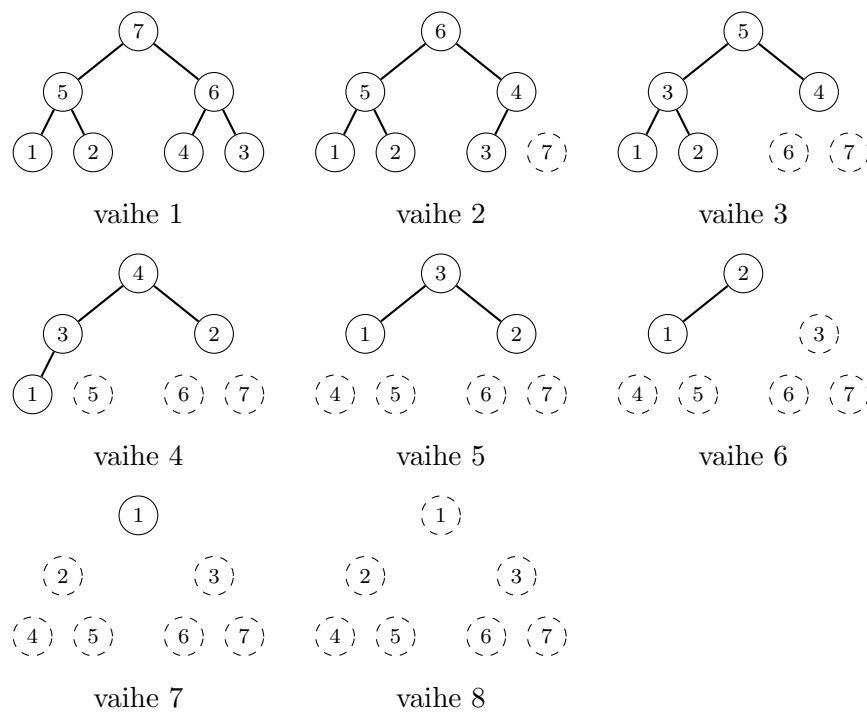
joten aikaa kuluu vain $O(n)$.

7.4.2 Kekojärjestäminen

Kekojärjestäminen (*heap sort*) on järjestämisalgoritmi, jonka toiminta perustuu kekoon. Ideana on muuttaa järjestettävä taulukko ensin keoksi ja sen jälkeen poistaa alkio keosta yksi kerrallaan järjestyksessä. Kekojärjestäminen vie aikaa $O(n \log n)$, koska taulukon muuttaminen keoksi vie aikaa $O(n)$ ja n alkion poistaminen keosta vie aikaa $O(n \log n)$.

Kuva 7.5 näyttää esimerkin kekojärjestämisestä, kun järjestämme taulukon $[5, 2, 3, 1, 7, 4, 6]$ pienimmästä suurimpaan. Muutamme ensin taulukon maksimikeoksi, jolloin taulukosta tulee $[7, 5, 6, 1, 2, 4, 3]$. Tämän jälkeen poistamme yksi kerrallaan keon juuressa olevan alkion vaihtamalla sen keon viimeisen alkion kanssa. Tämän seurauksena keosta poistuneet alkio (merkitty katkoviivoilla) muodostavat lopulta järjestetyn taulukon.

Kekojärjestäminen ei ole käytännössä yhtä tehokas algoritmi kuin lomitussjärjestäminen tai pikajärjestäminen, minkä vuoksi se ei ole saavuttanut samanlaista asemaa järjestämisalgoritmien joukossa. Siinä on kuitenkin yksi kiinnostava ominaisuus: jos haluamme selvittää vain taulukon k pienintä tai suurinta alkioita, tämä onnistuu ajassa $O(n + k \log n)$, koska meidän riittää poistaa keosta k kertaa pienin tai suurin alkio ajassa $O(\log n)$.



Kuva 7.5: Esimerkki kekojärjestämisestä.

Luku 8

Algoritmien suunnittelu

Kuinka voi suunnitella hyvän algoritmin? On selvää, ettei tähän kysymykseen ole yhtä helppoa vastausta. Yhtä hyvin voisi kysyä, kuinka voi kirjoittaa hyvän kirjan tai säveltää hyvää musiikkia. Algoritmien suunnittelu on taito, jonka oppiminen vie aikaa.

Algoritmisen ongelman ratkaisemisessa on usein kaksi vaihetta. Ensimmäinen vaihe on keksiä algoritmin yleisidea, mikä vaatii havaintoja ja oivalluksia ongelmasta. Tämän jälkeen toinen vaihe on löytää hyvä ja tehokas tapa toteuttaa algoritmi. Tässä voimme käyttää apuna järjestämistä, tietorakenteita ja muita tekniikoita.

Kuten aiemminkin, haluamme saada aikaan tehokkaita algoritmeja, jotka vievät aikaa $O(n)$ tai $O(n \log n)$. Tämä tarkoittaa käytännössä sitä, että voimme käydä läpi ja järjestää syötettä sekä käyttää tietorakenteita, joissa on tehokkaita operaatioita. Tämä ohjaa algoritmin suunnittelua, koska vaatimus tehokkuudesta rajoittaa, mitä voimme tehdä algoritmissa.

8.1 Ratkaisun vaiheet

Aloitamme ongelmasta, jossa n lasta haluaa mennä maailmanpyörään. Jokaisessa korissa voi istua yksi tai kaksi lasta, ja korissa istujien yhteispaino saa olla enintään x . Lisäksi tiedämme jokaisen lapsen painon. Mikä on pienin mahdollinen määrä koreja, joka riittää lapsille? Oletamme, ettei yhdenkään lapsen paino ylitä korin maksimipainoa.

Esimerkiksi jos lasten määrä on $n = 5$, lasten painot ovat $[2, 2, 4, 5, 8]$ ja korin painoraja on $x = 8$, pienin mahdollinen korien määrä on kolme. Tässä tapauksessa voimme muodostaa korit $[2, 4]$, $[2, 5]$ ja $[8]$. Tämä on optimaalinen ratkaisu, koska selvästikään ei riitä, että koreja olisi vain kaksi.

Vaihe 1: Yleisidean keksiminen

On valtava määrä mahdollisia tapoja, miten voimme sijoittaa lapsia koreihin, minkä vuoksi olisi liian hidasta käydä läpi kaikkia vaihtoehtoja. Jotta saamme aikaan tehokkaan algoritmin, meidän tulee keksiä jokin periaate, jonka avulla voimme päättää nopeasti, ketkä lapset ovat yhdessä koreissa.

Itse asiassa meidän riittää keskittyä ongelmaan, jossa haluamme löytää seuraavaan koriin tulevat lapset. Jos onnistumme tässä, voimme vain toistaa samaa, kunnes kaikki lapset ovat koreissa. Vaikeutena on kuitenkin, kuinka saamme tehtyä valinnan niin, että tuloksena on varmasti optimaalinen ratkaisu. Jos valitsemme väärällä tavalla koriin tulevat lapset, saatamme vahingossa käyttää liikaa koreja.

Osoittautuu, että tässä tehtävässä toimiva algoritmi on valita aina seuraavaan koriin *painavin* lapsi sekä sen pariksi jokin toinen lapsi niin, että painojen summa ei ylitä korin maksimipainoa. Jos ei ole mitään tapaa valita toista lasta, laitamme painavimman lapsen koriin yksin. Tällainen algoritmi on *ahne* (*greedy*), eli se tekee aina jonkin hyvältä tuntuvan valinnan, joka vie ratkaisua eteenpäin, eikä peruuta koskaan tehtyjä valintoja.

Tarkastellaan taas esimerkkiä, jossa lasten painot ovat $[2, 2, 4, 5, 8]$ ja korin maksimipaino on 8. Kun käytämme yllä kuvattua algoritmia, aloitamme lapsesta, jonka paino on 8. Tämä lapsi saa oman korin, koska emme voi laittaa samaan koriin ketään toista lasta. Tämän jälkeen vuoroon tulee lapsi, jonka paino on 5, ja se saa pariksi lapsen, jonka paino on 2. Lopuksi käsittelemme lapsen, jonka paino on 4, ja sen pariksi tulee lapsi, jonka paino on 2. Tuloksena on ratkaisu, jossa on korit $[8]$, $[2, 5]$ ja $[2, 4]$.

Mutta miksi tämä ahne algoritmi toimii? Tässä auttaa tarkastella, mitä tapahtuu, kun algoritmi tekee ensimmäisen valintansa. Painavin lapsi täytyy sijoittaa johonkin koriin, ja meidän kannattaa tietenkin laittaa samaan koriin toinenkin lapsi, jos tämä on mahdollista. Tässä ei ole väliä, minkä toisen lapsen valitsemme, koska jos jokin lapsi on riittävän kevyt toimiakseen parina painavimman lapsen kanssa, niin se voi toimia parina myös minkä tahansa muun lapsen kanssa. Niinpä algoritmin ensimmäinen askel on optimaalinen. Tämän jälkeen voimme toistaa vastaavan päättelyn algoritmin seuraavissa askelissa, mikä tarkoittaa, että koko algoritmi toimii oikein.

Vaihe 2: Tehokas toteutus

Meillä on nyt toimiva algoritmi, mutta meidän täytyy vielä keksiä hyvä tapa toteuttaa se. Jotta algoritmista tulee tehokas, sen täytyy löytää joka askeleella nopeasti seuraavaan koriin tuleva painavin lapsi sekä mahdollinen sen pariksi tuleva lapsi. Ehkä helpoin tapa saavuttaa tämä tavoite on käyttää

apuna järjestämistä.

Seuraava algoritmi järjestää ensin lasten painot ja alkaa sitten jakaa lapsia koreihin. Muuttuja a lähtee liikkeelle taulukon alusta ja osoittaa aina kevyimmän lapsen painoon. Muuttuja b puolestaan lähtee liikkeelle taulukon lopusta ja osoittaa aina painavimman lapsen painoon. Algoritmi laskee muuttujaan k , montako koria tarvitaan yhteensä. Jos kevyimmän ja painavimman lapsen painojen summa on enintään x , lapset sijoitetaan samaan koriin, a liikkuu oikealle ja b liikkuu vasemmalle. Jos taas painojen summa on suurempi kuin x , painavin lapsi saa oman korin ja vain b liikkuu vasemmalle.

```
sort(painot)
a = 0, b = n-1
k = 0
while a <= b
    if painot[a]+painot[b] <= x
        a++, b--
    else
        b--
    k++
print(k)
```

Algoritmi järjestää ensin taulukon, missä menee aikaa $O(n \log n)$. Tämän jälkeen **while**-silmukka vie aikaa $O(n)$, koska joka kierroksella muuttujat a ja b siirtyvät ainakin askeleen lähemmäs toisiaan. Niinpä algoritmin kokonais-aikavaativuus on $O(n \log n)$.

8.2 Algoritmien aineksia

Seuraavaksi käymme läpi *aineksia*, joita esiintyy usein tehokkaissa algoritmeissa. Nämä ainekset muodostavat hyvän perustan algoritmien suunnitteluun: kun saamme vastaamme uuden ongelman, voimme miettiä, voisimmeko hyödyntää aineksia jotenkin ongelman ratkaisemisessa.

8.2.1 Järjestäminen

Tehokas algoritmi perustuu usein tavalla tai toisella järjestämiseen, joka voi näyttäytyä monella tavalla algoritmissa.

Tarkastellaan esimerkkinä tehtävää, jossa annettuna on n kolikkoa ja haluamme löytää pienimmän rahamäärän, jota *emme* voi muodostaa summana kolikoista. Jokaisella kolikolla on tietty kokonaislukuarvo, ja muodostettava rahamäärä on myös kokonaisluku. Esimerkiksi jos kolikot ovat $[1, 2, 2, 7]$,

voimme muodostaa rahamäärät 1, 2, 3, 4 ja 5, mutta emme voi muodostaa rahamäärää 6, joten oikea vastaus on 6.

Algoritmien suunnittelussa auttaa usein lähteä liikkeelle helpoista tapauksista. Kun meillä on käytössämme tietyt kolikot, hyvä ensimmäinen tavoite on koettaa muodostaa rahamäärä 1. Jotta onnistumme tässä, meillä on pakko olla kolikko, jonka arvo on 1. Entä milloin voimme muodostaa rahamäärän 2, jos tiedämme, että voimme muodostaa rahamäärän 1? Tässä on kaksi vaihtoehtoa: meillä täytyy olla toinen kolikko, jonka arvo on 1, jolloin saamme summan $1 + 1 = 2$, tai sitten kolikko, jonka arvo on 2.

Tämä on hyvää päättelyä, mutta jotta saamme aikaan algoritmin, meidän täytyy pystyä yleistämään se kaikkiin tapauksiin. Voimme ajatella asiaa hie-man toisesta näkökulmasta: Meillä on joukko kolikoita, joiden avulla voimme muodostaa rahamäärät $1, 2, \dots, k$. Miten voisimme muodostaa myös rahamäärän $k + 1$? Ratkaisu on, että tarvitsemme uuden kolikon, jonka arvo on *enintään* $k + 1$. Kun uuden kolikon arvo on $u \leq k + 1$, voimme tämän jälkeen muodostaa rahamäärät $1, 2, \dots, k + u$. Tämän havainnon ansiosta voimme alkaa muodostaa ratkaisua kolikko kerrallaan. Lisäksi jos mahdollisia uusia kolikoita on useita, voimme aina valita ahneesti *pienimmän* kolikon, koska pystymme valitsemaan suuremmat kolikot myöhemminkin.

Nyt meillä on kaikki ainekset tehokasta algoritmia varten. Järjestämme ensin kolikot ja muodostamme sitten ratkaisun käymällä kolikot läpi pienimmästä suurimpaan. Aloitamme tyhjästä ratkaisusta, jossa $k = 0$. Jokaisen kolikon kohdalla tiedämme, että voimme muodostaa tällä hetkellä rahamäärät $1, 2, \dots, k$, joten voimme parantaa ratkaisua kolikon avulla, jos sen arvo on enintään $k + 1$. Muuten lopetamme ratkaisun muodostamisen, koska kaikki tulevat kolikot ovat vielä suurempia. Lopuksi toteamme, että $k + 1$ on pienin rahamäärä, jota emme voi muodostaa.

```
sort(kolikot)
k = 0
for i = 0 to n-1
    u = kolikot[i]
    if u <= k+1
        k += u
    else
        break
print(k+1)
```

Algoritmi järjestää ensin kolikot ajassa $O(n \log n)$ ja tämän jälkeen käy ne läpi ajassa $O(n)$, joten algoritmin aikavaativuus on $O(n \log n)$.

8.2.2 Tietorakenteet

Olemme tutustuneet kirjassa jo moniin tietorakenteisiin: listaan, hajautus-tauluun, binäärihakupuuhun ja kekkoon. Näissä tietorakenteissa kannattaa kiinnittää erityisesti huomiota siihen, mitkä operaatiot toimivat tehokkaasti $O(1)$ - tai $O(\log n)$ -ajassa. Nämä ovat operaatioita, joita voimme käyttää tehokkaissa algoritmeissa.

Tarkastellaan esimerkkinä tehtävää, jossa haluamme laskea, montako tapaa on valita taulukosta yhtenäinen osataulukko, jossa lukujen summa on x . Esimerkiksi jos taulukko on $[3, 1, 3, 4, 5]$ ja $x = 4$, niin tapoja on kolme: $[3, 1]$, $[1, 3]$ ja $[4]$. On helppoa ratkaista tehtävä kahdella silmukalla ajassa $O(n^2)$, mutta miten saisimme aikaan tehokkaan algoritmin?

Tässä auttaa muotoilla hieman toisin, mitä tarkoittaa, että osataulukolla on tietty summa. Merkitään $s(i)$:llä osataulukon summaa taulukon alusta kohtaan i asti, ja oletetaan lisäksi, että $s(-1) = 0$. Tätä merkintää käyttäen osataulukon summa kohdasta a kohtaan b on

$$s(b) - s(a - 1),$$

eli meidän riittää itse asiassa keskittyä vain taulukon alusta alkavien osataulukoiden summiin.

Koska haluamme saada tehokkaan algoritmin, hyvä tavoite olisi käydä taulukko läpi vain kerran. Kun olemme taulukon kohdassa i , monessako tähän kohtaan päättyvässä osataulukossa lukujen summa on x ? Tämä tarkoittaa, että haluamme etsiä kohdat $0 \leq j \leq i$, joille pätee

$$s(i) - s(j - 1) = x$$

eli

$$s(j - 1) = s(i) - x.$$

Saamme laskettua tällaisten kohtien määrän tehokkaasti, kun pidämme taulukon läpikäynnissä kirjaa, montako kertaa mikäkin summa on esiintynyt taulukon alkuosassa. Voimme toteuttaa algoritmin seuraavasti:

```
kerrat[0] = 1
laskuri = 0
summa = 0
for i = 0 to n-1
    summa += taulu[i]
    laskuri += kerrat[summa-x]
    kerrat[summa]++
print(laskuri)
```

Joka askeleella muuttuja `summa` sisältää summan $s(i)$. Hakemisto `kerrat` puolestaan pitää kirjaa siitä, montako kertaa mikäkin alkuosan summa on esiintynyt tähän mennessä. Voimme toteuttaa hakemiston hajautustaulun tai binäärihakupuun avulla. Näin saamme aikaan ratkaisun, joka vie aikaa $O(n)$ tai $O(n \log n)$ riippuen valitusta tietorakenteesta.

8.2.3 Binäärihaku

Binäärihaun tunnetuin käyttötarkoitus on alkion etsiminen järjestetystä taulukosta ajassa $O(\log n)$. Tämä on kuitenkin vain alkusoittoa sille, mitä binäärihaulla pystyy tekemään ja mikä sen merkitys on algoritmien suunnittelussa. Binäärihaun todellinen hyöty piilee siinä, että pystymme etsimään sen avulla tehokkaasti funktion *muutoskohdan*.

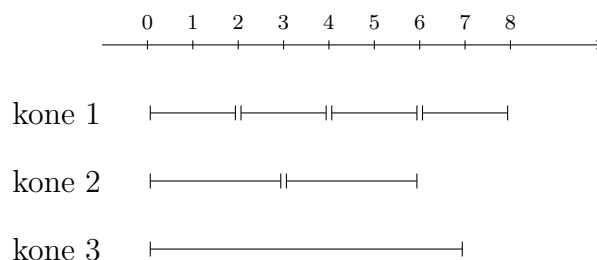
Oletetaan, että meillä on funktio `ok(x)`, joka kertoo, onko x kelvollinen ratkaisu tehtävään. Lisäksi pätee `ok(x) = false`, kun $x < u$, ja `ok(x) = true`, kun $x \geq u$. Binäärihaun avulla voimme etsiä tehokkaasti, mikä on funktion muutoskohta u eli ensimmäinen kohta, jossa funktio saa arvon `true`. Seuraava koodi toteuttaa haun, kun oletamme, että u on välillä $1 \dots N$:

```
a = 1, b = N
while a < b
    k = (a+b)/2
    if ok(k)
        b = k
    else
        a = k+1
u = a
```

Haun joka vaiheessa tiedämme, että muutoskohta on välillä $[a, b]$. Laskemme keskikohdan $k = \lfloor (a + b)/2 \rfloor$ ja tutkimme funktion arvoa kohdassa k . Jos pätee `ok(k)`, niin muutoskohdan on oltava välillä $[a, k]$, ja muuten sen täytyy olla välillä $[k + 1, b]$. Lopulta välillä on vain yksi alkio, jolloin olemme löytäneet muutoskohdan. Koska välin koko puolittuu joka askeleella, kutsumme funktiota `ok` yhteensä $O(\log N)$ kertaa.

Mutta mitä hyötyä on siitä, että löydämme tehokkaasti funktion muutoskohdan? Tämä selviää seuraavassa tehtävässä: Käytössämme on n konetta ja haluamme valmistaa niiden avulla m tavaraa. Tiedämme jokaisesta koneesta, montako minuuttia kestää valmistaa yksi tavara konetta käyttäen, ja haluamme löytää aikataulun, jota seuraamalla pystymme valmistamaan m tavaraa mahdollisimman nopeasti.

Kuva 8.1 näyttää parhaan aikataulun esimerkkitapauksessa, jossa koneiden määrä on $n = 3$, koneiden nopeudet ovat $[2, 3, 7]$ ja valmistettavien tava-



Kuva 8.1: Optimaalinen tapa valmistaa 7 tavaraa vie 8 minuuttia, kun koneiden nopeudet ovat $[2, 3, 7]$.

roiden määrä on $m = 7$. Käynnistämme koneen 1 neljästi kahden minuutin välein, koneen 2 kahdesti kolmen minuutin välein ja koneen 3 kerran. Viimeisenä pysähtyy kone 1, kun aloittamisesta on kulunut kahdeksan minuuttia.

Voimme pukea tehtävän binäärihauille sopivaan muotoon määrittämällä, että $\text{ok}(x) = \text{true}$ tarkalleen silloin, kun voimme valmistaa *ainakin* m tavaraa ajassa x . Tällöin funktion muutoskohta u vastaa tehtävän ratkaisua. Entä miten voimme laskea funktion ok arvon? Jos meillä on x minuuttia aikaa ja koneella i kestää a_i minuuttia valmistaa yksi tavara, pystymme valmistamaan $\lfloor x/a_i \rfloor$ tavaraa kyseistä konetta käyttäen. Kun sitten käytössämme on kaikki koneet, pystymme valmistamaan yhteensä

$$s = \sum_{i=1}^n \lfloor x/a_i \rfloor$$

tavaraa. Niinpä $\text{ok}(x) = \text{false}$, jos $s < m$, ja $\text{ok}(x) = \text{true}$, jos $s \geq m$. Voimme toteuttaa tämän käytännössä seuraavasti:

```
function ok(x)
  s = 0
  for i = 1 to n
    s += x/a[i]
  return s >= m
```

Voimme kytkeä tämän funktion suoraan binäärihakuun, jolloin tuloksena on tehokas algoritmi tehtävän ratkaisemiseen. Meidän täytyy kuitenkin vielä valita arvo N , joka on jokin yläraja kohdalle u . Tässä tehtävässä helppo valinta on

$$N = a_1 \cdot m,$$

mikä vastaa ratkaisua, jossa käytämme vain ensimmäistä konetta. On varmaa, että oikea u :n arvo on korkeintaan N , joten binäärihaku kutsuu $O(\log N)$

kertaa funktiota ok . Koska jokainen funktion kutsu vie aikaa $O(n)$, tuloksena on algoritmi, jonka aikavaativuus on $O(n \log N)$.

Huomaa, että $\log N$ on käytännössä pieni luku riippumatta siitä, kuinka suuri luku N on. Niinpä meidän ei tarvitse murehtia siitä, kuinka suuria a_1 ja m ovat, vaan voimme luottaa siihen, että algoritmi on tehokas.

8.2.4 Tasoitettu analyysi

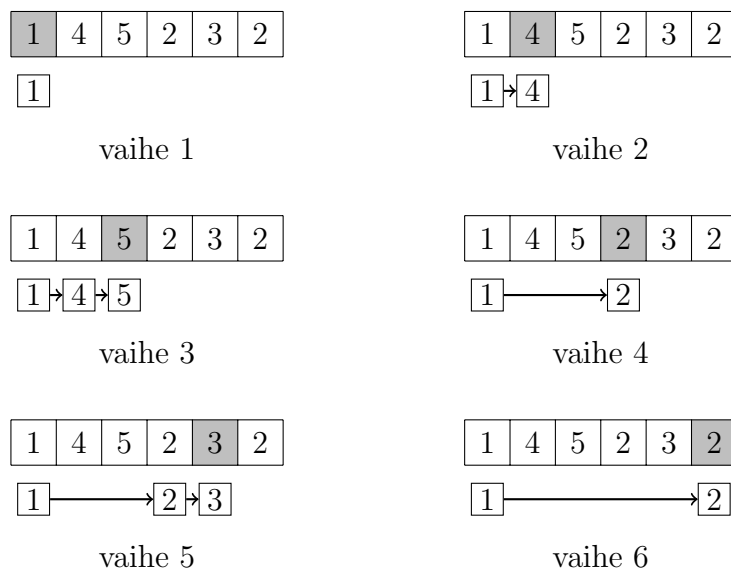
Voimme yleensä määrittää algoritmin aikavaativuuden helposti katsomalla jokaisesta silmukasta, montako kertaa siinä olevaa koodia suoritetaan. Joskus näin suoraviivainen analyysi ei anna kuitenkaan oikeaa kuvaa algoritmin tehokkuudesta, koska silmukan suorituskertojen määrä saattaa vaihdella algoritmin eri vaiheissa. Tutustumme seuraavaksi tekniikkaan nimeltä *tasoitettu analyysi* (*amortized analysis*), jossa koetamme arvioida tarkemmin, montako kertaa silmukassa olevaa koodia suoritetaan *yhteensä* algoritmin aikana.

Tasoitettuun analyysiin liittyy yleensä jokin tietorakenne, jonka operaatioiden määrää haluamme arvioida. Tarkastellaan esimerkkinä tehtävää, jossa meillä on n lukua sisältävä taulukko ja haluamme muodostaa toisen taulukon, jossa on jokaisen luvun *lähin pienempi edeltäjä*. Tämä tarkoittaa, että haluamme löytää jokaiselle luvulle pienemmän luvun, joka on mahdollisimman lähellä aiemmin taulukossa. Esimerkiksi taulukon $[1, 4, 5, 2, 3, 2]$ lähimmät pienemmät edeltäjät ovat $[-, 1, 4, 1, 2, 1]$. Koska luvulla 1 ei ole pienempää edeltäjää, sen kohdalla on merkki $-$.

Saamme ratkaistua tehtävän tehokkaasti algoritmilla, joka käy läpi taulukkoa vasemmalta oikealle ja pitää yllä *pinoa*, jossa on lista taulukon lukuja. Pino on muodostettu niin, että jokainen alkio on edellistä suurempi. Jokaisessa kohdassa i poistamme ensin pinon lopusta lukuja niin kauan kuin pinon viimeinen luku on suurempi tai yhtä suuri kuin kohdan i luku. Tämän jälkeen kirjaamme muistiin, että kohdan i luvun lähin pienempi edeltäjä on pinon viimeinen luku (jos pino ei ole tyhjä) ja lisäämme kohdan i luvun pinon loppuun. Tuloksena on seuraava algoritmi:

```
pino = []
for i = 0 to n-1
    while not pino.empty() and pino.top() >= taulu[i]
        pino.pop()
    if not pino.empty()
        edeltaja[i] = pino.top()
    pino.push(taulu[i])
```

Kuva 8.2 näyttää, kuinka algoritmi käsittelee taulukon $[1, 4, 5, 2, 3, 2]$.



Kuva 8.2: Etsimme lähimmät pienemmät edeltäjät pinon avulla.

Alussa pino on tyhjä, joten toteamme, ettei luvulla 1 ole lähintä pienempää edeltäjää ja lisäämme sen pinoon. Sitten vuoroon tulee luku 4, jonka lähin pienempi edeltäjä on pinon päällä oleva luku 1. Tämän jälkeen lisäämme luvun 4 pinoon. Vastaavasti luvun 5 lähin pienempi edeltäjä on luku 4, ja lisäämme luvun 5 pinoon. Luvun 2 kohdalla poistamme pinosta luvut 5 ja 4 ja toteamme, että luvun 2 lähin pienempi edeltäjä on luku 1. Lopuksi käsittelemme vielä vastaavasti luvut 3 ja 2.

Algoritmin tehokkuus riippuu siitä, montako kertaa suoritamme `while`-silmukassa olevan koodin. Oleellinen havainto on, että voimme poistaa pinosta korkeintaan niin monta alkia kuin olemme lisänneet siihen, eli emme voi kutsua `pop`-funktia useammin kuin `push`-funktia. Koska lisäämme pinon n alkia, suoritamme `while`-silmukassa olevaa koodia siis enintään n kertaa algoritmin aikana. Niinpä koko algoritmi vie aikaa vain $O(n)$.

Luku 9

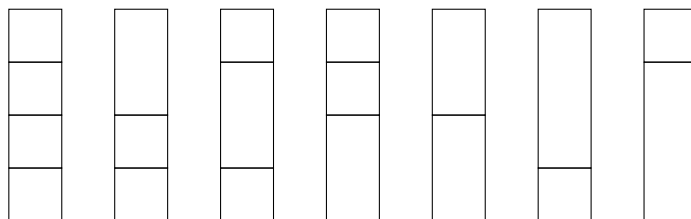
Dynaaminen ohjelmointi

Dynaaminen ohjelmointi (*dynamic programming*) on tekniikka, jonka avulla voi monessa tilanteessa laskea ongelman ratkaisujen yhteismäärän tai löytää ratkaisun, joka on jollain mittarilla optimaalinen. Ideana on muotoilla ongelma rekursiivisesti niin, että voimme laskea halutun asian saman ongelman pienempien osaongelmien avulla. Tämän jälkeen saamme aikaan tehokkaan algoritmin, kun käsittelemme jokaisen osaongelman vain kerran.

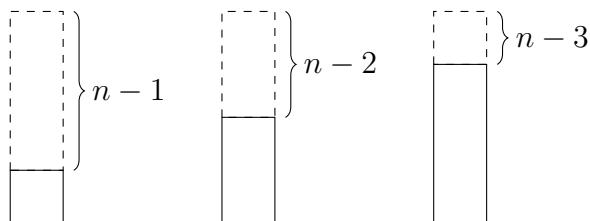
Tässä luvussa tutustumme ensin dynaamisen ohjelmoinnin perusteisiin käyttäen esimerkkinä tehtävää, jossa rakennamme torneja palikoista. Tämän jälkeen käymme läpi kokoelman muita tehtäviä, jotka esittelevät lisää dynaamisen ohjelmoinnin mahdollisuuksia.

9.1 Perustekniikat

Aloitamme dynaamiseen ohjelmointiin tutustumisen tehtävästä, jossa haluamme rakentaa palikoista tornin, jonka korkeus on n . Kunkin palikan korkeus on 1, 2 tai 3, ja jokaista palikkatyyppiä on saatavilla rajattomasti. Monellako tavalla voimme rakentaa tornin?



Kuva 9.1: Voimme rakentaa korkeuden 4 tornin 7 tavalla palikoista, joiden korkeudet ovat 1, 2 ja 3.



Kuva 9.2: Rekursiivinen idea: kun alamme rakentaa tornia, voimme laittaa pohjalle korkeuden 1, 2 tai 3 palikan.

Esimerkiksi kun tornin korkeus on $n = 4$, voimme rakentaa sen 7 tavalla kuvan 9.1 mukaisesti. Jos n on pieni, voimme laskea tornien määrän helposti käymällä läpi kaikki tavat, mutta tornien määrä kasvaa nopeasti emmekä voi käyttää raakaa voimaa suuremmilla n :n arvoilla. Seuraavaksi ratkaisemmekin ongelman tehokkaasti dynaamisella ohjelmoinnilla.

9.1.1 Rekursiivinen esitys

Jotta voimme käyttää dynaamista ohjelmointia, meidän täytyy pystyä esittämään ongelma *rekursiivisesti* niin, että saamme laskettua ongelman ratkaisun käyttäen osaongelmina pienempiä vastaavia ongelmia. Tässä tehtävässä luonteva rekursiivinen funktio on `tornit(n)`: monellako tavalla voimme rakentaa tornin, jonka korkeus on n ? Esimerkiksi `tornit(4) = 7`, koska voimme rakentaa korkeuden 4 tornin 7 tavalla.

Funktion pienten arvojen laskeminen on helppoa. Ensinnäkin `tornit(0) = 1`, koska on tarkalleen yksi tapa rakentaa tyhjä torni: siinä ei ole mitään palikoita. Sitten `tornit(1) = 1`, koska ainoa tapa rakentaa korkeuden 1 torni on valita palikka, jonka korkeus on 1, ja `tornit(2) = 2`, koska voimme rakentaa korkeuden 2 tornin valitsemalla joko kaksi palikkaa, jonka kummankin korkeus on 1, tai yhden palikan, jonka korkeus on 2.

Kuinka voisimme sitten laskea funktion arvon *yleisessä* tapauksessa, kun tornin korkeus on n ? Tässä voimme miettiä, kuinka tornin rakentaminen alkaa. Meillä on kolme mahdollisuutta: voimme laittaa ensin palikan, jonka korkeus on 1, 2 tai 3. Jos aloitamme korkeuden 1 palikalla, meidän täytyy rakentaa sen päälle korkeuden $n - 1$ torni. Vastaavasti jos aloitamme korkeuden 2 tai 3 palikalla, meidän täytyy rakentaa sen päälle torni, jonka korkeus on $n - 2$ tai $n - 3$. Kuva 9.2 havainnollistaa tämän idean. Niinpä voimme laskea tornien määrän rekursiivisesti kaavalla

$$\text{tornit}(n) = \text{tornit}(n - 1) + \text{tornit}(n - 2) + \text{tornit}(n - 3),$$

korkeus n	<code>tornit(n)</code>
0	1
1	1
2	2
3	4
4	7
5	13
6	24
7	44
8	81
9	149

Taulukko 9.1: Tornien määrät, kun korkeus n on $0, 1, \dots, 9$.

kun $n \geq 3$. Esimerkiksi voimme laskea

$$\text{tornit}(3) = \text{tornit}(2) + \text{tornit}(1) + \text{tornit}(0) = 4$$

ja

$$\text{tornit}(4) = \text{tornit}(3) + \text{tornit}(2) + \text{tornit}(1) = 7,$$

jolloin olemme saaneet laskettua esimerkkitapaustamme vastaavasti, että voimme rakentaa korkeuden 4 tornin 7 tavalla.

Taulukko 9.1 näyttää funktion `tornit(n)` arvot, kun $n = 0, 1, \dots, 9$. Kuten taulukosta voi huomata, funktion arvo kasvaa nopeasti: se lähes kaksinkertaistuu joka askeleella. Kun n on suuri, meillä onkin valtavasti mahdollisuuksia tornin rakentamiseen.

9.1.2 Tehokas toteutus

Nyt kun olemme saaneet aikaan rekursiivisen funktion, voimme toteuttaa sen ohjelmoimalla seuraavasti:

```
function tornit(n)
  if (n == 0) return 1
  if (n == 1) return 1
  if (n == 2) return 2
  return tornit(n-1)+tornit(n-2)+tornit(n-3)
```

Tämä on toimiva ratkaisu, mutta siinä on yksi ongelma: funktion arvon laskeminen vie kauan aikaa, jos n on vähänkin suurempi. Käytännössä laskenta alkaa hidastua parametrin $n = 30$ tienoilla. Esimerkiksi arvon `tornit(40)`

laskeminen vie aikaa noin minuutin ja arvon `tornit(50)` vie aikaa niin kauan, että emme jaksakaan odottaa laskennan valmistumista.

Syynä laskennan hitauteen on, että funktiota `tornit` kutsutaan uudestaan ja uudestaan samoilla parametreilla ja tornien määrä lasketaan loppujen lopuksi summaksi luvuista 1 ja 2 pohjatapauksista. Niinpä kun tornien määrä on suuri, laskenta on tuomittu viemään kauan aikaa. Voimme kuitenkin selviytyä ongelmasta toteuttamalla laskennan hieman toisella tavalla.

Tässä astuu kuvaan dynaamisen ohjelmoinnin keskeinen idea *taulukointi* (*memoization*): laskemme funktion arvon kullekin parametrille vain kerran ja tallennamme tulokset taulukkoon myöhempiä käyttöä varten. Tätä varten luomme taulukon `tornit`, jossa kohtaan `tornit[i]` tallennetaan funktion arvo `tornit(i)`. Kun haluamme laskea korkeuden n tornien määrän, täytämme taulukon kohdat $0, 1, \dots, n$. Seuraava koodi toteuttaa laskennan:

```
tornit[0] = 1
tornit[1] = 1
tornit[2] = 2
for i = 3 to n
    tornit[i] = tornit[i-1]+tornit[i-2]+tornit[i-3]
```

Koodin suorituksen jälkeen taulukon arvo `tornit[n]` kertoo meille, monellako tavalla voimme rakentaa korkeuden n tornin.

Tämän toteutuksen etuna on, että se on huomattavasti nopeampi kuin rekursiivinen funktio. Koska koodissa on vain yksi `for`-silmukka, se vie aikaa vain $O(n)$, eli voimme käsitellä tehokkaasti myös suuria n :n arvoja. Esimerkiksi voimme nyt laskea salamannopeasti, että

$$\text{tornit}(50) = 10562230626642,$$

eli on yli 10562 miljardia tapaa rakentaa korkeuden 50 torni.


9.2 Esimerkkejä

Olemme nyt tutustuneet dynaamisen ohjelmoinnin perusideaan, mutta tämä on vasta alkua sille, mitä kaikkea tekniikan avulla pystyy tekemään. Seuraavaksi käymme läpi kokoelman tehtäviä, jotka esittelevät lisää dynaamisen ohjelmoinnin mahdollisuuksia.

9.2.1 Pisin nouseva alijono

Ensimmäinen ongelmamme on selvittää, kuinka pitkä on n alkiota sisältävän taulukon *pin nouseva alijono*. Tämä tarkoittaa, että meidän tulee valita

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


Kuva 9.3: Taulukon pisin nouseva alijono on $[2, 5, 7, 8]$.

taulukosta mahdollisimman pitkä jono alkioita niin, että seuraava alkio on aina edellistä suurempi. Kuvassa 9.3 on esimerkki taulukosta, jonka pisin nouseva alijono $[2, 5, 7, 8]$ on pituudeltaan 4.

Voimme lähestyä tehtävää laskemalla jokaiselle taulukon kohdalle $k = 0, 1, \dots, n - 1$ arvon $\text{pisin}(k)$: kuinka pitkä on pisin nouseva alijono, joka päättyy kohtaan k . Kun olemme laskeneet kaikki nämä arvot, suurin arvoista kertoo meille, kuinka pitkä on pisin nouseva alijono koko taulukossa. Esimerkiksi kuvan 9.3 taulukossa $\text{pisin}(6) = 4$, koska kohtaan 6 päättyvä pisin nouseva alijono on pituudeltaan 4.

Millainen on sitten pisin kohtaan k päättyvä alijono? Yksi mahdollisuus on, että alijonossa on vain kohdan k alkio, jolloin $\text{pisin}(k) = 1$. Muussa tapauksessa alijonossa on ensin kohtaan x päättyvä pisin nouseva alijono, missä $x < k$, ja sitten vielä kohdan k alkio. Tämä edellyttää, että kohdan x alkio on pienempi kuin kohdan k alkio. Tuloksena olevan alijonon pituus on $\text{pisin}(x) + 1$. Tämä antaa mahdollisuuden dynaamiseen ohjelmointiin: kun haluamme laskea arvon $\text{pisin}(k)$, käymme läpi kaikki mahdolliset tavat valita kohta x ja valitsemme niistä parhaan vaihtoehdon.

Seuraava koodi laskee jokaiselle $k = 0, 1, \dots, n - 1$ pisimmän kohtaan k päättyvän alijonon pituuden yllä kuvattua ideaa käyttäen. Koodi olettaa, että taulukon sisältö on taulukossa `taulu`, ja se muodostaa taulukon `pisin`, jossa on pisimpien alijonojen pituudet.

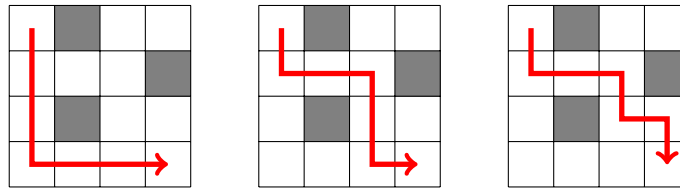
```

for k = 0 to n-1
  pisin[k] = 1
  for x = 0 to k-1
    if taulu[x] < taulu[k] and pisin[x]+1 > pisin[k]
      pisin[k] = pisin[x]+1

```

Koodin suorituksen jälkeen pisimmän nousevan alijonon pituus on siis suurin taulukon `pisin` arvoista. Tämä algoritmi vie aikaa $O(n^2)$, koska käymme jokaisessa kohdassa k läpi kaikki taulukon edelliset kohdat.

Mitä jos haluaisimme selvittää pisimmän nousevan alijonon pituuden lisäksi, mistä alkioista se muodostuu? Tämä onnistuu laajentamalla hie-
man koodia. Rakennamme taulukon `aiempi`, joka kertoo jokaisessa kohdassa,



Kuva 9.4: Mahdolliset reitit vasemmasta yläkulmasta oikeaan alakulmaan.

missä on tähän kohtaan päättyvän pisimmän alijonon edellinen alkio. Voimme muodostaa taulukon seuraavasti:

```
for k = 0 to n-1
  pisin[k] = 1
  aiempi[k] = -1
  for x = 0 to k-1
    if taulu[x] < taulu[k] and pisin[x]+1 > pisin[k]
      pisin[k] = pisin[x]+1
      aiempi[k] = x
```

Tämän jälkeen jokaisessa kohdassa k arvo $\text{aiempi}[k]$ kertoo pisimmän alijonon edellisen alkion kohdan. Kuitenkin jos alijonon pituus on 1, taulukossa on arvo -1 . Voimme nyt selvittää kohtaan k päättyvän alijonon alkioit käänteisesti seuraavasti:

```
while k != -1
  print(taulu[k])
  k = aiempi[k]
```

Voimme käyttää vastaavaa tekniikkaa dynaamisessa ohjelmoinnissa aina, kun haluamme selvittää, mistä aineksista paras ratkaisu muodostuu.

9.2.2 Reitti ruudukossa

Olemme $n \times n$ -ruudukon vasemmassa yläkulmassa ja haluamme päästä oikeaan alakulmaan. Jokaisella vuorolla voimme siirtyä askeleen alaspäin tai oikealle. Kuitenkin joissakin ruuduissa on este, emmekä voi kulkea sellaisen ruudun kautta. Montako mahdollista reittiä on olemassa? Esimerkiksi kuvassa 9.4 on 4×4 -ruudukko, jossa on kolme mahdollista reittiä ruudukon vasemmasta yläkulmasta oikeaan alakulmaan.

Tässä tehtävässä osaongelmat ovat *kaksiulotteisia*, koska olemme reitin joka vaiheessa tietyn rivin tietyllä sarakkeella. Niinpä määrittelemme rekursiivisen funktion, jolla on kaksi parametria: $\text{reitit}(y, x)$ kertoo, montako

reittiä on vasemmasta yläkulmasta ruutuun (y, x) . Numeroimme rivit ja sarakkeet $1, 2, \dots, n$ ja haluamme laskea arvon $\text{reitit}(n, n)$, joka on reittien määrä vasemmasta yläkulmasta oikeaan alakulmaan.

Hyödyllinen havainto on, että jokaisessa ruudussa on kaksi mahdollisuutta, kuinka reitti voi tulla ruutuun, koska voimme tulla joko ylhäältä tai vasemmalta. Kun haluamme laskea reittien määrää, laskemmekin yhteen ylhäältä ja vasemmalta tulevat reitit. Rajoituksena jos ruudussa on este, siihen tulevien reittien määrä on aina nolla. Tämän perusteella saamme aikaan seuraavan dynaamisen ohjelmoinnin algoritmin:

```
for y = 1 to n
  for x = 1 to n
    if este[y][x]
      reitit[y][x] = 0
    else if y == 1 and x == 1
      reitit[y][x] = 1
    else
      reitit[y][x] = reitit[y-1][x] + reitit[y][x-1]
```

Koodi laskee jokaiseen ruutuun reittien määrän vasemmasta yläkulmasta kyseiseen ruutuun. Jos ruudussa on este, reittien määrä on 0, koska mitään reittiä ei ole olemassa. Jos ruutu on vasen yläkulma (eli $y = 1$ ja $x = 1$), reittien määrä on 1, koska reitti alkaa siitä ruudusta. Muuten reittien määrä saadaan laskemalla yhteen ylhäältä ja vasemmalta tulevat reitit. Tuloksena on algoritmi, joka vie aikaa $O(n^2)$.

Huomaa, että tässä käytämme taulukon kohtia $1, 2, \dots, n$ ja oletamme, että kohdissa 0 on arvona nolla. Tämä on kätevää, koska meidän ei tarvitse tehdä erikoistapauksia ruudukon yläreunaa ja vasenta reunaa varten.

9.2.3 Repunpakkaus

Termi *repunpakkaus* (*knapsack*) viittaa ongelmiin, jossa meillä on joukko tavaroita, joilla on tietyt painot, ja haluamme selvittää, millaisia yhdistelmiä niistä voi muodostaa. Ongelmasta on monia muunnelmia, joiden yhdistävänä tekijänä on, että ne voi ratkaista tehokkaasti dynaamisella ohjelmoinnilla.

Seuraavaksi keskitymme ongelmaan, jossa meillä on n tavaraa, joilla on painot p_1, p_2, \dots, p_n . Haluamme selvittää kaikki mahdolliset yhteispainot, jotka voimme muodostaa valitsemalla jonkin osajoukon tavaroista. Esimerkiksi jos tavaroiden painot ovat $[1, 3, 3, 4]$, niin mahdolliset yhteispainot ovat $[0, 1, 3, 4, 5, 6, 7, 8, 10, 11]$. Esimerkiksi yhteispaino 6 on listalla, koska saamme sen painoista $3 + 3 = 6$, ja yhteispaino 8 on listalla, koska saamme sen painoista $1 + 3 + 4 = 8$.

	0	1	2	3	4	5	6	7	8	9	10	11
$k = 0$	✓											
$k = 1$	✓	✓										
$k = 2$	✓	✓		✓	✓							
$k = 3$	✓	✓		✓	✓		✓	✓				
$k = 4$	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓

Kuva 9.5: Mahdolliset yhteispainot, kun painot ovat $[1, 3, 3, 4]$.

On helppoa laskea, mikä on suurin mahdollinen tavaroiden yhteispaino, koska saamme sen valitsemalla kaikki tavarat. Suurin yhteispaino on siis $s = p_1 + p_2 + \dots + p_n$. Tämä on meille hyödyllinen yläraja, koska tiedämme nyt, että tavaroiden yhteispaino on aina jokin luku välillä $0 \dots s$.

Voimme lähestyä tehtävää dynaamisella ohjelmoinnilla määrittelemällä funktion $\text{painot}(k)$: mitkä kaikki yhteispainot voimme muodostaa, jos käytössämme on tavarat $1, 2, \dots, k$? Oletamme, että funktio palauttaa *taulukon*, jossa on $s + 1$ alkia: jokaiselle yhteispainolle $0, 1, \dots, s$ tieto siitä, voimme muodostaa sen painoista p_1, p_2, \dots, p_k . Tapaus $\text{painot}(0)$ on helppo, koska kun meillä ei ole mitään tavaroita, ainoa mahdollinen yhteispaino on 0. Tämän jälkeen pystymme laskemaan tapauksen $\text{painot}(k)$ ottamalla lähtökohdaksi tapauksen $\text{painot}(k - 1)$ ja selvittämällä, mitä uusia yhteispainoja voimme muodostaa, kun saamme käyttää myös painoa p_k .

Kuva 9.5 näyttää dynaamisen ohjelmoinnin taulukoiden sisällön esimerkissämme, jossa painot ovat $[1, 3, 3, 4]$. Ensimmäisellä rivillä $k = 0$, joten ainoa yhteispaino on 0. Toisella rivillä $k = 1$, joten saamme käyttää painoa $p_1 = 1$ ja voimme muodostaa yhteispainot 0 ja 1. Kolmannella rivillä $k = 2$, jolloin saamme käyttöömmme painon $p_2 = 3$ ja voimme muodostaa yhteispainot 0, 1, 3 ja 4. Viimeisellä rivillä käytössämme ovat kaikki painot, joten se vastaa ongelman ratkaisua.

Voimme toteuttaa dynaamisen ohjelmoinnin kätevästi niin, että koodissa on vain yksi boolean-tila painot , jossa on $s + 1$ alkia. Taulukko kertoo laskennan jokaisessa vaiheessa, mitkä yhteispainot ovat mahdollisia sillä hetkellä. Aluksi taulukon kohdassa 0 on arvo `true` ja kaikissa muissa kohdissa on arvo `false`. Tämän jälkeen päivitämme taulukkoa lisäämällä mukaan painoja yksi kerrallaan.

```

painot[0] = true
for i = 1 to n
  for j = s to 0
    if painot[j]
      painot[j+p[i]] = true

```

Tärkeä yksityiskohta algoritmissa on, että käymme jokaisen painon kohdalla taulukon läpi *lopusta alkuun*. Syynä tähän on, että tällä tavalla saamme laskettua oikealla tavalla, mitä uusia yhteispainoja voimme muodostaa, kun saamme käyttää uutta painoa kerran. Laskennan jälkeen voimme tulostaa kaikki mahdolliset yhteispainot näin:

```

for i = 0 to s
  if painot[i]
    print(i)

```

Tuloksena olevan algoritmin aikavaativuus on $O(ns)$. Algoritmin tehokkuus riippuu siis paitsi tavaroiden määrästä, myös niiden painoista. Jotta algoritmi on käyttökelpoinen, painojen summan s täytyy olla niin pieni, että voimme varata niin suuren taulukon.

9.2.4 Binomikertoimet

Binomikerroin $\binom{n}{k}$ ilmaisee, monellako tavalla voimme muodostaa n alkion joukosta k alkion osajoukon. Esimerkiksi $\binom{5}{3} = 10$, koska voimme muodostaa joukosta $\{1, 2, 3, 4, 5\}$ seuraavat 3 alkion osajoukot:

- $\{1, 2, 3\}$ • $\{1, 3, 5\}$ • $\{2, 4, 5\}$
- $\{1, 2, 4\}$ • $\{1, 4, 5\}$ • $\{3, 4, 5\}$
- $\{1, 2, 5\}$ • $\{2, 3, 4\}$
- $\{1, 3, 4\}$ • $\{2, 3, 5\}$

Binomikertoimien laskemiseen on monia tapoja. Dynaamisen ohjelmoinnin kannalta kiinnostava tapa on rekursiivinen kaava

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Voimme perustella kaavan tarkastelemalla k alkion osajoukon muodostamista joukosta $\{1, 2, \dots, n\}$. Jos otamme osajoukkoon mukaan alkion n , meidän

tulee muodostaa vielä $k - 1$ alkion osajoukko joukosta $\{1, 2, \dots, n - 1\}$. Jos taas emme ota osajoukkoon mukaan alkia n , meidän tulee muodostaa k alkion osajoukko joukosta $\{1, 2, \dots, n - 1\}$. Lisäksi pohjatapauksina

$$\binom{n}{0} = 1,$$

koska voimme muodostaa tyhjän osajoukon yhdellä tavalla, ja

$$\binom{n}{k} = 0, \text{ jos } k > n,$$

koska n alkista ei voi muodostaa osajoukkoa, jossa on yli n alkia.

Tämä rekursiivinen kaava tarjoaa meille tavan laskea tehokkaasti binomikertoimia dynaamisen ohjelmoinnin avulla. Voimme toteuttaa laskennan seuraavasti:

```
binom[0][0] = 1
for i = 1 to n
  binom[i][0] = 1
  for j = 1 to k
    binom[i][j] = binom[i-1][j-1] + binom[i-1][j]
```

Koodin suorituksen jälkeen taulukon kohdassa `binom[n][k]` on binomikerroin $\binom{n}{k}$. Algoritmi vie aikaa $O(nk)$, joten sitä voi käyttää melko suurten binomikertoimien laskemiseen.

Luku 10

Verkkojen perusteet

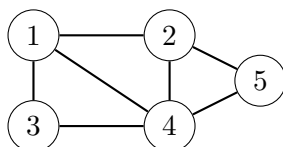
Voimme ratkaista monia algoritmiikan ongelmia esittämällä tilanteen *verkko* (*graph*) ja käyttämällä sitten sopivaa verkkoalgoritmia. Tyypillinen esimerkki verkosta on tieverkosto, joka muodostuu kaupungeista ja niiden välisistä teistä. Tällaisessa verkossa ongelmana voi olla selvittää vaikkapa, kuinka voimme matkustaa kaupungista a kaupunkiin b .

Tässä luvussa aloitamme verkkoihin tutustumisen käymällä läpi verkkojen käsitteitä sekä tapoja esittää verkkoja ohjelmoinnissa. Tämän jälkeen näemme, miten voimme tutkia verkkojen rakennetta ja ominaisuuksia syvyys- ja leveyshaun avulla. Seuraavissa kirjan luvuissa jatkamme verkkojen käsittelyä ja opimme lisää verkkoalgoritmeja.

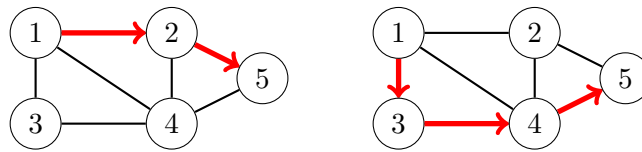
10.1 Verkkojen käsitteitä

Verkko muodostuu *solmuista* (*node* tai *vertex*) ja niiden välisistä *kaarista* (*edge*). Kuvassa 10.1 on verkko, jossa on viisi solmua ja seitsemän kaarta. Merkitsemme verkon solmujen määrää kirjaimella n ja kaarten määrää kirjaimella m . Lisäksi numeroimme verkon solmut kokonaisluvuin $1, 2, \dots, n$.

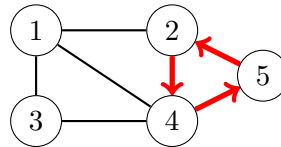
Sanomme, että kaksi solmua ovat *vierekkäin* (*adjacent*), jos niiden välillä on kaari. Solmun *naapureja* (*neighbor*) ovat kaikki solmut, joihin se on yhteydessä kaarella, ja solmun *aste* (*degree*) on sen naapureiden määrä. Kuvassa



Kuva 10.1: Verkko, jossa on viisi solmua ja seitsemän kaarta.



Kuva 10.2: Kaksi polkua solmusta 1 solmuun 5.

Kuva 10.3: Verkossa on sykli $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$.

10.1 solmun 2 naapurit ovat 1, 4 ja 5, joten solmun aste on 3.

Polku ja sykli

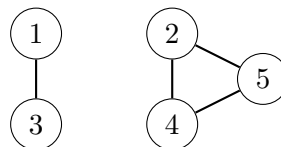
Verkossa oleva *polku* (*path*) on kaaria pitkin kulkeva reitti lähtösolmusta kohdesolmuun. Kuva 10.2 näyttää kaksi mahdollista polkua solmusta 1 solmuun 5 esimerkiverkossamme. Ensimmäinen polku on $1 \rightarrow 2 \rightarrow 5$ ja toinen polku on $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$.

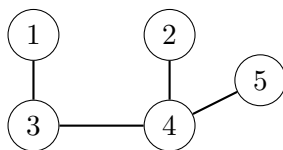
Polku on *sykli* (*cycle*), jos sen alku- ja loppusolmu on sama, siinä on ainakin yksi kaari eikä se kulje kahta kertaa saman solmun tai kaaren kautta. Kuvassa 10.3 on esimerkkinä sykli $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$. Jos verkossa ei ole yhtään sykliä, sanomme, että verkko on *sykkitön* (*acyclic*).

Yhtenäisyys ja komponentit

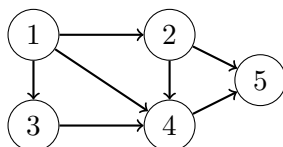
Verkko on *yhtenäinen* (*connected*), jos minkä tahansa kahden solmun välillä on polku. Kuvan 10.1 verkko on yhtenäinen, mutta kuvan 10.4 verkko ei ole yhtenäinen, koska esimerkiksi solmujen 1 ja 2 välillä ei ole polkua.

Voimme esittää verkon aina kokoelmana yhtenäisiä *komponentteja* (*component*). Kuvassa 10.4 yhtenäiset komponentit ovat $\{1, 3\}$ ja $\{2, 4, 5\}$.

Kuva 10.4: Verkon yhtenäiset komponentit ovat $\{1, 3\}$ ja $\{2, 4, 5\}$.



Kuva 10.5: Puu eli yhtenäinen, syklitön verkko.



Kuva 10.6: Suunnattu verkko.

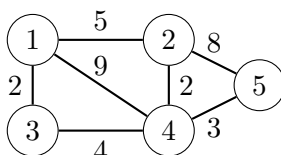
Verkko on *puu* (*tree*), jos se on sekä yhtenäinen että syklitön. Puussa kaarten määrä on aina yhden pienempi kuin solmujen määrä, ja jokaisen kahden solmun välillä on yksikäsitteinen polku. Kuvassa 10.5 on esimerkkinä puu, jossa on viisi solmua ja neljä kaarta.

Suunnattu verkko

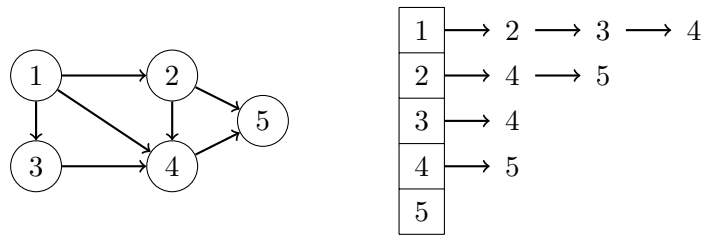
Suunnatussa (*directed*) verkossa jokaisella kaarella on tietty suunta, jonka mukaisesti meidän täytyy kulkea kaarta pitkin. Suunnat rajoittavat siis kulkemistamme verkossa. Kuvassa 10.6 on esimerkkinä suunnattu verkko, jossa voimme kulkea solmusta 1 solmuun 5 polkua $1 \rightarrow 2 \rightarrow 5$, mutta emme voi kulkea mitenkään solmusta 5 solmuun 1.

Painotettu verkko

Painotetussa (*weighted*) verkossa jokaiseen kaareen liittyy jokin paino, joka kuvaa tyypillisesti kaaren pituutta. Kun kuljemme polkua painotetussa verkossa, polun pituus on kaarten painojen summa. Kuvassa 10.7 on esimerkkinä painotettu verkko, jossa polun $1 \rightarrow 2 \rightarrow 5$ pituus on $5 + 8 = 13$ ja polun $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ pituus on $2 + 4 + 3 = 9$.



Kuva 10.7: Painotettu verkko.



Kuva 10.8: Verkon vieruslistaesitys.

10.2 Verkot ohjelmoinnissa

Verkon esittämiseen ohjelmoinnissa on monia mahdollisuuksia. Sopivan esitystavan valintaan vaikuttaa, miten haluamme käsitellä verkkoa algoritmisissa, koska jokaisessa esitystavassa on omat etunsa. Seuraavaksi käymme läpi kolme tavallista esitystapaa.

10.2.1 Vieruslistaesitys

Tavallisin tapa esittää verkko on luoda kullekin solmulle *vieruslista* (*adjacency list*), joka kertoo, mihin solmuihin voimme siirtyä solmusta kaaria pitkin. Kuvassa 10.8 on verkko ja sitä vastaava vieruslistaesitys. Jos haluamme tallentaa verkon vieruslistoina Javassa, voimme luoda taulukon

```
ArrayList<Integer>[] verkko = new ArrayList[n+1];
```

ja alustaa vieruslistat näin:

```
for (int i = 1; i <= n; i++) {
    verkko[i] = new ArrayList<>();
}
```

Tämän jälkeen lisäämme kaaret listoille näin:

```
verkko[1].add(2);
verkko[1].add(3);
verkko[1].add(4);
verkko[2].add(4);
verkko[2].add(5);
verkko[3].add(4);
verkko[4].add(5);
```

Vieruslistaesitys on monessa tilanteessa hyvä tapa tallentaa verkko, koska haluamme usein selvittää, mihin solmuihin pääsemme siirtymään tietystä

solmusta kaaria pitkin. Esimerkiksi seuraava koodi käy läpi solmut, joihin voimme siirtyä solmusta x kaarella:

```
for (Integer s : verkko[x]) {  
    // käsittele solmu s  
}
```

Jos verkko on suuntaamaton, eli voimme kulkea kaaria molempiin suuntiin, voimme tallentaa verkon vastaavalla tavalla, kunhan tallennamme kunkin kaaren molempiin suuntiin. Jos taas verkko on painotettu, voimme tallentaa jokaisesta kaaresta sekä kohdesolmun että painon.

10.2.2 Kaarilistaesitys

Toinen tapa tallentaa verkko on luoda *kaarilista* (*edge list*), joka sisältää kaikki verkon kaaret. Javassa voimme luoda listan

```
ArrayList<Kaari> kaaret = new ArrayList<>();
```

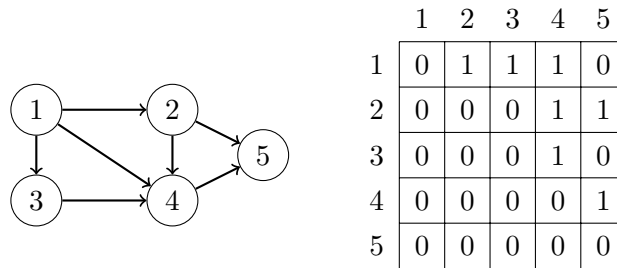
jossa on seuraavanlaisia kaaria:

```
public class Kaari {  
    public int alku, loppu;  
  
    public Kaari(int alku, int loppu) {  
        this.alku = alku;  
        this.loppu = loppu;  
    }  
}
```

Seuraava koodi luo esimerkkiverkkoamme vastaavan kaarilistan:

```
kaaret.add(new Kaari(1,2));  
kaaret.add(new Kaari(1,3));  
kaaret.add(new Kaari(1,4));  
kaaret.add(new Kaari(2,4));  
kaaret.add(new Kaari(2,5));  
kaaret.add(new Kaari(3,4));  
kaaret.add(new Kaari(4,5));
```

Kaarilista on hyvä esitystapa algoritmeissa, joissa haluamme pystyä käymään helposti läpi kaikki verkon kaaret eikä ole tarvetta selvittää tietystä solmusta lähteviä kaaria.



Kuva 10.9: Verkon vierusmatriisiesitys.

10.2.3 Vierusmatriisiesitys

Vierusmatriisi (*adjacency matrix*) on kaksiulotteinen taulukko, joka kertoo jokaisesta verkon kaaresta, esiintyykö se verkossa. Matriisin rivin a sarakkeessa b oleva arvo ilmaisee, onko verkossa kaarta solmusta a solmuun b . Kuvassa 10.9 on esimerkki verkon vierusmatriisiesityksestä. Tässä tapauksessa matriisin jokainen arvo on 0 (ei kaarta) tai 1 (kaari).

Javassa voimme määritellä vierusmatriisin seuraavasti:

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

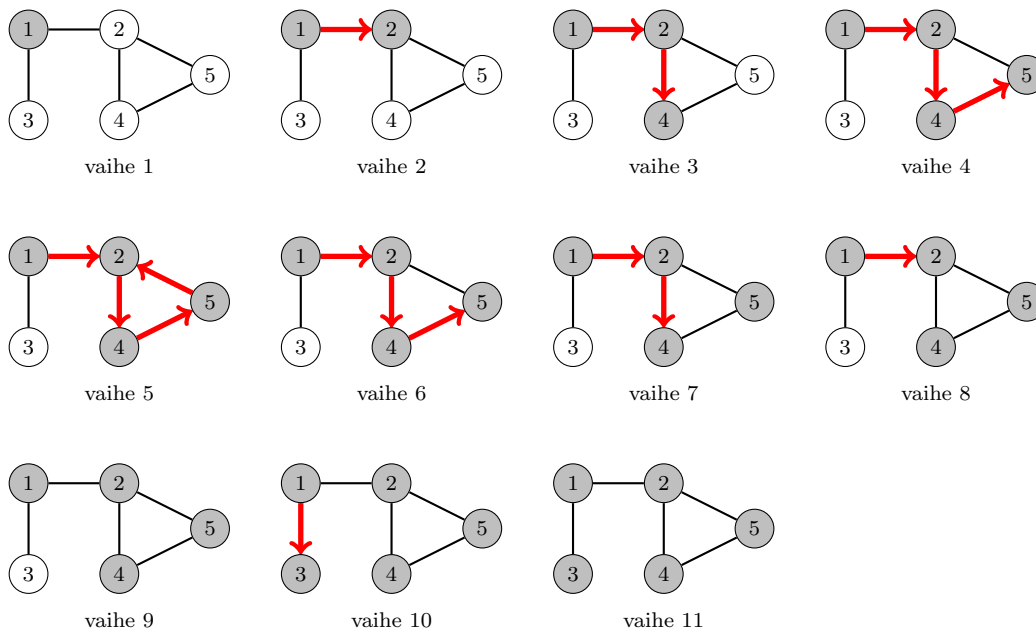
Tämän jälkeen muodostamme matriisiin näin:

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

Jos verkko on painotettu, voimme vastaavasti merkitä kaarten painot vierusmatriisiin. Vierusmatriisin etuna on, että voimme tarkastaa helposti, onko tietty kaari verkossa. Esitystapa kuluttaa kuitenkin paljon muistia, eikä sitä voi käyttää, jos verkon solmujen määrä on suuri.

10.3 Verkon läpikäynti

Tutustumme seuraavaksi kahteen keskeiseen verkkoalgoritmiin, jotka käyvät läpi verkossa olevia solmuja ja kaaria. Ensin käsittelemme syvyyshakua, joka on yleiskäyttöinen algoritmi verkon läpikäyntiin, ja sen jälkeen leveyshakua, jonka avulla voimme löytää lyhimpiä polkuja verkossa.



Kuva 10.10: Esimerkki syvyyshaun toiminnasta.

10.3.1 Syvyyshaku

Syvyyshaku (*depth-first search* eli *DFS*) on verkkojen käsittelyn yleistyökalu, jonka avulla voimme selvittää monia asioita verkon rakenteesta. Kun alamme tutkia verkkoa syvyyshaulla, meidän tulee päättää ensin, mistä solmusta haku lähtee liikkeelle. Haku etenee vuorollaan kaikkiin solmuihin, jotka ovat saavutettavissa lähtösolmusta kulkemalla kaaria pitkin.

Syvyyshaku pitää yllä jokaisesta verkon solmusta tietoa, onko solmussa jo vierailtu. Kun haku saapuu solmuun, jossa se ei ole vierailut aiemmin, se merkitsee solmun vierailuksi ja alkaa käydä läpi solmusta lähteviä kaaria. Jokaisen kaaren kohdalla haku etenee verkon niihin osiin, joihin pääsee kaaren kautta. Lopulta kun haku on käynyt läpi kaikki kaaret, se perääntyy taaksepäin samaa reittiä kuin tuli solmuun.

Kuvassa 10.10 on esimerkki syvyyshaun toiminnasta. Jokaisessa vaiheessa harmaat solmut ovat solmuja, joissa haku on jo vierailut. Tässä esimerkissä haku lähtee liikkeelle solmusta 1, josta pääsee kaarella solmuihin 2 ja 3. Haku etenee ensin solmuun 2, josta pääsee edelleen solmuihin 4 ja 5. Tämän jälkeen haku ei enää löydä uusia solmuja tästä verkon osasta, joten se perääntyy takaisin kulkemaansa reittiä solmuun 1. Lopuksi haku käy vielä solmussa 3, josta ei pääse muihin uusiin solmuihin. Nyt haku on käynyt läpi kaikki solmut, joihin pääsee solmusta 1.

Syvyyshaku on mukavaa toteuttaa rekursiivisesti seuraavaan tapaan:

```
procedure haku(solmu)
  if vierailtu[solmu]
    return
  vierailtu[solmu] = true
  for naapuri in verkko[solmu]
    haku(naapuri)
```

Haku käynnistyy, kun kutsumme aliohjelmaa `haku` parametrina haun lähtösolmu. Jokaisessa kutsussa aliohjelma tarkistaa ensin, olemmeko jo käyneet parametrina annetussa solmussa, ja päättyy heti tässä tilanteessa. Muuten aliohjelma merkitsee, että olemme nyt käyneet solmussa ja etenee rekursiivisesti kaikkiin sen naapureihin. Haku vie aikaa $O(n + m)$, koska käsittelemme jokaisen solmun ja kaaren enintään kerran.

Mihin voisimme sitten käyttää syvyyshakua? Seuraavassa on joitakin esimerkkejä syvyysshaun käyttökohteista:

Polun etsiminen

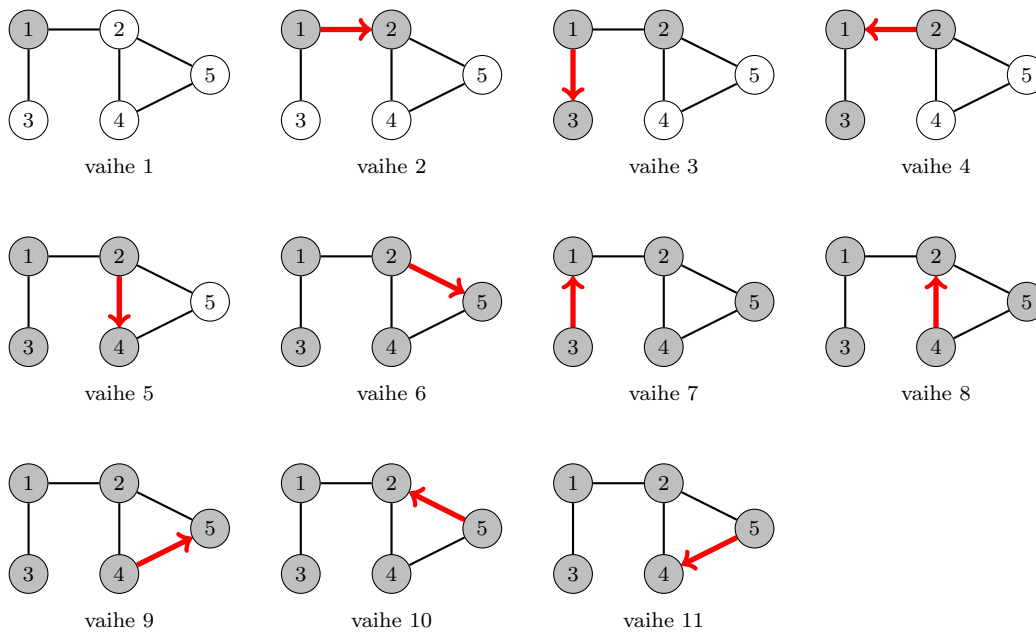
Syvyysshaun avulla voimme etsiä verkosta polun solmusta a solmuun b , jos tällainen polku on olemassa. Tämä tapahtuu aloittamalla haku solmusta a ja pysähtymällä, kun vastaan tulee solmu b . Jos polkuja on useita, syvyyshaku löytää jonkin niistä riippuen solmujen käsittelyjärjestyksestä.

Yhtenäisyys ja komponentit

Suuntaamaton verkko on yhtenäinen, jos kaikki solmut ovat yhteydessä toisiinsa. Voimmekin tarkastaa verkon yhtenäisyyden aloittamalla syvyysshaun jostakin solmusta ja tutkimalla, saavuttaako haku kaikki verkon solmut. Lisäksi voimme löytää verkon yhtenäiset komponentit käymällä läpi solmut ja aloittamalla uuden syvyysshaun aina, kun vastaan tulee vierailematon solmu. Jokainen syvyyshaku muodostaa yhden komponentin.

Syklin etsiminen

Jos suuntaamaton verkko sisältää syklin, huomaamme tämän syvyysshaun aikana siitä, että tulemme toista kautta johonkin solmuun, jossa olemme käyneet jo aiemmin. Niinpä löydämme syvyysshaun avulla jonkin verkossa olevan syklin, jos sellainen on olemassa.



Kuva 10.11: Esimerkki leveyshaun toiminnasta.

10.3.2 Leveyshaku

Leveyshaku (*breadth-first search* eli *BFS*) käy syvyysshaun tavoin läpi kaikki verkon solmut, joihin pääsee kulkemaan kaaria pitkin annetusta lähtösolmusta. Erona on kuitenkin, missä järjestyksessä solmut käydään läpi. Leveyshaku käy solmuja läpi *kerroksittain* niin, että se käsittelee solmut siinä järjestyksessä kuin ne ovat tulleet ensimmäistä kertaa vastaan haun aikana.

Vaikka voisimme käyttää leveyshakua yleisenä algoritmina verkon läpi-käyntiin syvyysshaun tavoin, käytämme sitä tavallisesti silloin, kun olemme kiinnostuneita verkon *lyhimmistä poluista*. Leveyshaun avulla pystymme nimittäin määrittämään lyhimmän polun pituuden eli *etäisyyden* lähtösolmusta kuhunkin haun aikana kohtaamaamme solmuun. Tässä oletamme, että polun pituus tarkoittaa sen kaarten määrää eli lyhin polku on mahdollisimman vähän kaaria sisältävä polku.

Kuvassa 10.11 on esimerkki leveyshaun toiminnasta, kun aloitamme haun solmusta 1 lähtien. Käsittelemme ensin solmun 1, josta pääsemme uusiin solmuihin 2 ja 3. Tämä tarkoittaa, että lyhimmat polut solmuihin 2 ja 3 ovat $1 \rightarrow 2$ ja $1 \rightarrow 3$ eli etäisyys näihin solmuihin on 1. Sitten käsittelemme solmun 2, josta pääsemme uusiin solmuihin 4 ja 5. Tämä tarkoittaa, että lyhimmat polut solmuihin 4 ja 5 ovat $1 \rightarrow 2 \rightarrow 4$ ja $1 \rightarrow 2 \rightarrow 5$ eli etäisyys näihin solmuihin on 2. Lopuksi käsittelemme vielä solmut 3, 4 ja 5, joista

emme kuitenkaan pääse enää uusiin solmuihin.

Tavallinen tapa toteuttaa leveyshaku on käyttää *jonoa*, jossa on käsittelyä odottavia solmuja. Jonon ansiosta pystymme käymään läpi solmut siinä järjestyksessä kuin olemme löytäneet ne leveyshaun aikana. Oletamme, että jonossa on metodi `enqueue`, joka lisää alkion jonon loppuun, sekä metodi `dequeue`, joka hakee ja poistaa jonon ensimmäisen alkion. Seuraava koodi suorittaa leveyshaun lähtösolmusta `alku` alkaen:

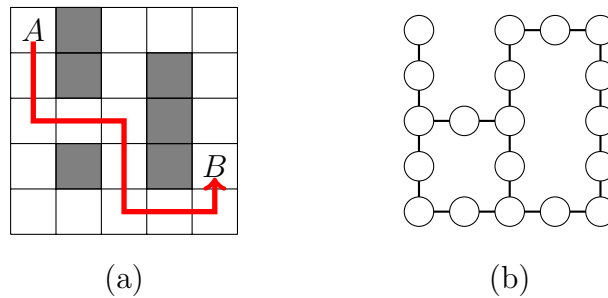
```
jono.enqueue(alku)
vierailtu[alku] = true
etaisyys[alku] = 0
while not jono.empty()
    solmu = jono.dequeue()
    for naapuri in verkko[solmu]
        if vierailtu[naapuri]
            continue
        jono.enqueue(naapuri)
        vierailtu[naapuri] = true
        etaisyys[naapuri] = etaisyys[solmu]+1
```

Lisäämme ensin jonoon lähtösolmun ja merkitsemme, että olemme vierailleet siinä ja että etäisyys siihen on 0. Tämän jälkeen alamme käsitellä solmuja siinä järjestyksessä kuin ne ovat jonossa. Käsitlemme solmun käymällä läpi sen naapurit. Jos emme ole aiemmin käyneet naapurissa, lisäämme sen jonoon ja päivitämme taulukoita. Haku vie aikaa $O(n+m)$, koska käsitlemme jokaisen solmun ja kaaren enintään kerran.

10.3.3 Esimerkki: Labyrintti

Olemme labyrintissa ja haluamme päästä ruudusta A ruutuun B . Joka vuorolla voimme siirtyä yhden askeleen ylöspäin, alaspäin, vasemmalle tai oikealle. Pystymmekö pääsemään ruudusta A ruutuun B , ja jos pystymme, mikä on lyhin mahdollinen reitti? Esimerkiksi kuvassa 10.12(a) lyhin reitti ruudusta A ruutuun B muodostuu 9 askeleesta.

Voimme esittää ongelman verkkona niin, että jokainen lattiaruutu on yksi verkon solmuista ja kahden solmun välillä on kaari, jos vastaavat ruudut ovat vierekkäin labyrintissa. Kuva 10.12(b) näyttää esimerkkilabyrinttimme verkona. Tätä esitystapaa käyttäen ruudusta A on reitti ruutuun B tarkalleen silloin, kun vastaavat verkon solmut kuuluvat samaan yhtenäiseen komponenttiin. Voimme siis tarkastaa syvyyshaulla, onko ruudusta A reittiä ruutuun B . Lyhin reitti ruudusta A ruutuun B löytyy puolesta leveyshaulla, joka lähtee liikkeelle ruudusta A .



Kuva 10.12: (a) Lyhin reitti labyrintissa ruudusta A ruutuun B . (b) Labyrintin esittäminen verkkona.

Huomaa, että meidän ei tarvitse erikseen muuttaa labyrinttia verkoksi, vaan voimme toteuttaa haut *implisiittiseen* verkkoon. Tämä tarkoittaa, että teemme haun labyrinttiin sen omassa esitysmuodossa. Käytännössä labyrintti on kätevää tallentaa kaksiulotteisena taulukkona, joka kertoo, mitkä ruudut ovat seinäruutuja. Tällöin voimme toteuttaa esimerkiksi syvyyshaun seuraavan tyyliä:

```

procedure haku(y,x)
  if y < 0 or x < 0 or y >= n or x >= n
    return
  if seinä[y][x] or vierailtu[y][x]
    return
  vierailtu[y][x] = true
  haku(y+1,x)
  haku(y-1,x)
  haku(y,x+1)
  haku(y,x-1)

```


Luku 11

Lyhimmät polut

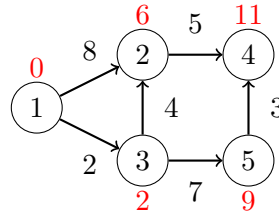
Monissa verkkoihin liittyvissä ongelmissa on kysymys siitä, että haluamme löytää *lyhimmän polun* (*shortest path*) verkon solmusta toiseen. Esimerkiksi voimme haluta selvittää, mikä on nopein reitti kahden katuosoitteen välillä tai mikä on halvin tapa lentää kaupungista toiseen. Näissä ja muissa soveluksissa on tärkeää, että löydämme lyhimmän polun tehokkaasti.

Olemme käyttäneet aiemmin leveyshakua lyhimpien polkujen etsimiseen. Tämä onkin hyvä ratkaisu, kun haluamme löytää polut, joiden kaarten määrä on pienin. Tässä luvussa keskitymme kuitenkin vaikeampaan tilanteeseen, jossa verkko on *painotettu* ja haluamme löytää polut, joissa painojen summa on pienin. Tällöin emme voi enää käyttää leveyshakua vaan tarvitsemme kehittyneempiä menetelmiä.

Lyhimpien polkujen etsimiseen painotetussa verkossa on monia algoritmeja, joilla on erilaisia ominaisuuksia. Tässä luvussa käymme läpi ensin Bellman–Fordin algoritmin ja Dijkstran algoritmin, jotka etsivät lyhimmät polut annetusta lähtösolmusta kaikkiin verkon solmuihin. Tämän jälkeen tutustumme Floyd–Warshallin algoritmiin, joka etsii samanaikaisesti lyhimmät polut kaikkien verkon solmujen välillä.

11.1 Lyhimmät polut lähtösolmusta

Tavallisin tilanne käytännön verkko-ongelmissa on, että haluamme löytää lyhimmän polun solmusta a solmuun b . Yksittäisen lyhimmän polun etsiminen vaatii usein kuitenkin, että etsimme sitä ennen muitakin lyhimpiä polkuja. Niinpä keskitymme alusta asti yleisempään ongelmaan, jossa olemme valinneet jonkin solmun lähtösolmuksi ja haluamme määrittää *jokaiselle* verkon solmulle, kuinka pitkä on lyhin polku lähtösolmusta solmuun eli mikä on solmun etäisyys lähtösolmusta.



Kuva 11.1: Lyhimpien polkujen pituudet solmusta 1 alkaen.

Kuvassa 11.1 on esimerkkinä verkko, jossa lähtösolmuna on solmu 1 ja jokaisen solmun viereen on merkitty sen etäisyys. Esimerkiksi solmun 5 etäisyys on 9, koska lyhin polku solmusta 1 solmuun 5 on $1 \rightarrow 3 \rightarrow 5$, jonka pituus on $2 + 7 = 9$. Käytämme tätä verkkoa esimerkkinä, kun tutustumme seuraavaksi kahteen algoritmiin lyhimpien polkujen etsimiseen.

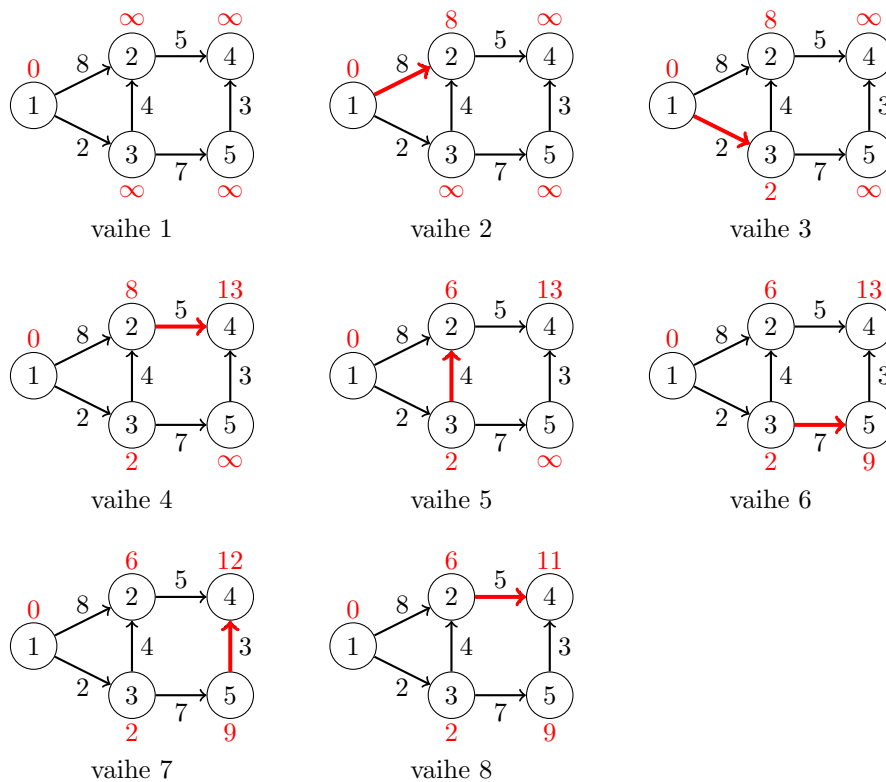
11.1.1 Bellman–Fordin algoritmi

Bellman–Fordin algoritmi etsii lyhimvät polut annetusta lähtösolmusta kaikkiin verkon solmuihin. Algoritmi muodostaa taulukon, joka kertoo jokaiselle verkon solmulle sen etäisyyden lähtösolmusta. Algoritmi toimii missä tahansa verkossa, kunhan verkossa ei ole negatiivista sykliä eli sykliä, jonka painojen summa on negatiivinen.

Bellman–Fordin algoritmi pitää yllä *arvioita* solmujen etäisyyksistä niin, että aluksi etäisyys lähtösolmuun on 0 ja etäisyys kaikkiin muihin solmuihin on ääretön. Tämän jälkeen algoritmi alkaa parantaa etäisyyksiä etsimällä verkosta kaaria, joiden kautta kulkeminen lyhentää polkuja. Jokaisella askeleella algoritmi etsii kaaren $a \rightarrow b$, jolle pätee, että pääsemme solmuun b aiempaa lyhempää polkua kulkemalla kaarella solmusta a . Kun mitään etäisyysarviota ei voi enää parantaa, algoritmi päättyy ja kaikki etäisyydet vastaavat todellisia lyhimpien polkujen pituuksia.

Kuva 11.2 näyttää esimerkin Bellman–Fordin algoritmin toiminnasta, kun lähtösolmuna on solmu 1. Jokaisen solmun vieressä on ilmoitettu sen etäisyysarvio: aluksi etäisyys solmuun 1 on 0 ja etäisyys kaikkiin muihin solmuihin on ääretön. Jokainen etäisyyden muutos näkyy kuvassa omana vaiheenaan. Ensin parannamme etäisyyttä solmuun 2 kulkemalla kaarta $1 \rightarrow 2$, jolloin etäisyydeksi tulee 8. Sitten parannamme etäisyyttä solmuun 3 kulkemalla kaarta $1 \rightarrow 3$, jolloin solmun uudeksi etäisyydeksi tulee 2. Jatkamme samalla tavalla, kunnes emme voi enää parantaa mitään etäisyyttä ja kaikki etäisyydet vastaavat lyhimpien polkujen pituuksia.

Bellman–Fordin algoritmi on mukavaa toteuttaa käyttäen verkon kaarilistaesitystä, jossa jokaisesta kaaresta on tallennettu alku- ja loppusolmu sekä



Kuva 11.2: Esimerkki Bellman–Fordin algoritmin toiminnasta.

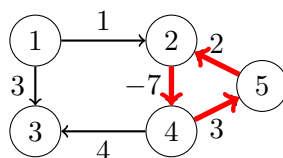
paino. Toteutamme algoritmin niin, että se muodostuu *kierroksista*, joista jokainen käy läpi kaikki verkon kaaret ja koettaa parantaa etäisyysarvioita niiden avulla. Kuten pian huomaamme, algoritmia riittää suorittaa $n - 1$ kierrosta, missä n on verkon solmujen määrä. Tämän jälkeen algoritmi on varmasti löytänyt kaikki lyhimmet polut lähtösolmusta alkaen. Voimme siis toteuttaa algoritmin seuraavasti:

```

for i = 1 to n-1
  for kaari in kaaret
    nyky = etaisyys[kaari.loppu]
    uusi = etaisyys[kaari.alku]+kaari.paino
    if uusi < nyky
      etaisyys[kaari.loppu] = uusi

```

Algoritmi käy jokaisella kierroksella läpi verkon kaaret ja tutkii kunkin kaaren kohdalla, mikä on nykyinen etäisyys kaaren kohdesolmuun sekä mikä on uusi etäisyys, jos kuljemmekin solmuun kaaren kautta. Jos uusi etäisyys on pienempi, päivitämme sen solmun etäisyysarvioksi.



Kuva 11.3: Negatiivinen sykli $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$, jonka avulla voimme lyhentää polkuja loputtomasti.

Algoritmin analyysi

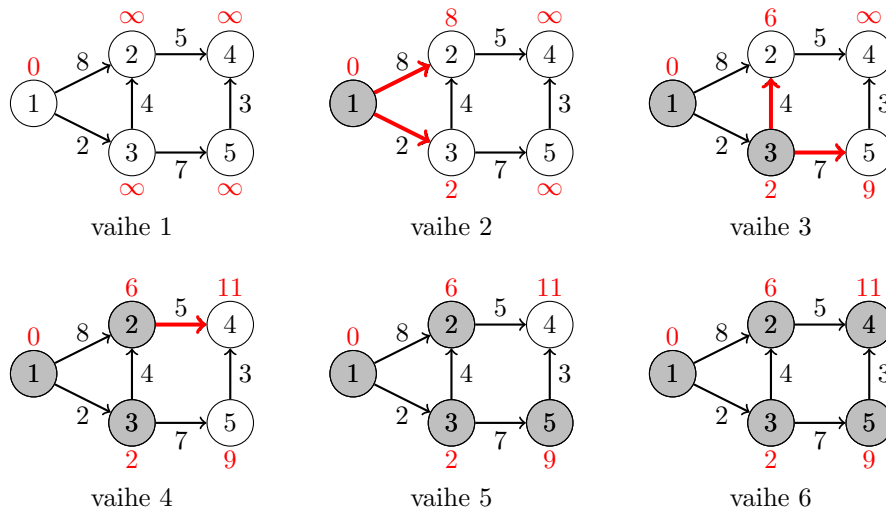
Olemme nyt kuvailleet ja toteuttaneet Bellman–Fordin algoritmin, mutta miten voimme olla varmoja, että se löytää kaikissa tilanteissa lyhimvät polut $n - 1$ kierroksen kuluessa? Jotta voimme vastata tähän kysymykseen, tarvitsemme kaksi havaintoa koskien verkon lyhimpiä polkuja.

Ensimmäinen havainto on, että jos $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_k$ on lyhin polku solmusta s_1 solmuun s_k , niin myös $s_1 \rightarrow s_2$ on lyhin polku solmusta s_1 solmuun s_2 , $s_1 \rightarrow s_2 \rightarrow s_3$ on lyhin polku solmusta s_1 solmuun s_3 , jne., eli jokainen lyhimmän polun alkuosa on myös lyhin polku vastaavaan solmuun. Jos näin ei olisi, voisimme parantaa lyhintä polkua solmusta s_1 solmuun s_k parantamalla jotain polun alkuosaa, mikä aiheuttaisi ristiriidan.

Toinen havainto on, että n solmun verkossa jokainen lyhin polku voi sisältää enintään $n - 1$ kaarta, kun oletamme, että verkossa ei ole negatiivista sykliä. Jos polkuun kuuluisi n kaarta tai enemmän, jokin solmu esiintyisi polulla monta kertaa. Tämä ei ole kuitenkaan mahdollista, koska ei olisi järkeä kulkea monta kertaa saman solmun kautta, kun haluamme saada aikaan lyhimmän polun.

Tarkastellaan nyt, mitä tapahtuu algoritmin kierroksissa. Ensimmäisen kierroksen jälkeen olemme löytäneet lyhimvät polut, joissa on enintään yksi kaari. Toisen kierroksen jälkeen olemme löytäneet lyhimvät polut, joissa on enintään kaksi kaarta. Sama jatkuu, kunnes $n - 1$ kierroksen jälkeen olemme löytäneet lyhimvät polut, joissa on enintään $n - 1$ kaarta. Koska missään lyhimässä polussa ei voi olla enempää kaaria, olemme löytäneet kaikki lyhimvät polut. Algoritmin riittää suorittaa siis $n - 1$ kierrosta, joista jokainen käy läpi kaikki verkon kaaret ajassa $O(m)$. Niinpä algoritmi löytää kaikki lyhimvät polut ajassa $O(nm)$.

Mitä tapahtuu sitten, jos verkossa on negatiivinen sykli? Esimerkiksi kuvan 11.3 verkossa on negatiivinen sykli $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$, jonka paino on -2 . Tässä tilanteessa Bellman–Fordin algoritmi ei anna oikeaa tulosta, koska voimme lyhentää loputtomasti syklin kautta kulkevia polkuja. Oikeastaan ongelma on siinä, että lyhin polku ei ole mielekäs käsite, jos polun osana on



Kuva 11.4: Esimerkki Dijkstran algoritmin toiminnasta.

negatiivinen sykli. Voimme kuitenkin tarkastaa, onko verkossa negatiivista sykliä, suorittamalla Bellman–Fordin algoritmia n kierrosta tavallisen $n - 1$ kierroksen sijasta. Jos jokin etäisyys paranee vielä viimeisellä kierroksella, verkossa on negatiivinen sykli.

11.1.2 Dijkstran algoritmi

Dijkstran algoritmi on Bellman–Fordin algoritmin tehostettu versio, jonka toiminta perustuu oletukseen, että verkossa ei ole negatiivisen painoisia kaaria. Bellman–Fordin algoritmin tapaan Dijkstran algoritmi pitää yllä arvioita etäisyyksistä lähtösolmusta muihin solmuihin. Erona on kuitenkin tapa, miten Dijkstran algoritmi parantaa etäisyyksiä.

Dijkstran algoritmissa verkon solmut kuuluvat kahteen luokkaan: käsittelemättömiin ja käsiteltyihin. Aluksi kaikki solmut ovat käsittelemättömiä. Algoritmi etsii joka askeleella käsittelemättömän solmun, jonka etäisyysarvio on pienin. Sitten algoritmi käy läpi kaikki solmusta lähtevät kaaret ja koettaa parantaa etäisyyksiä niiden avulla. Tämän jälkeen solmu on käsitelty eikä sen etäisyys enää muutu, eli aina kun olemme käsitelleet solmun, olemme saaneet selville sen lopullisen etäisyyden.

Kuva 11.4 näyttää esimerkin Dijkstran algoritmin toiminnasta. Solmun harmaa väri tarkoittaa, että se on käsitelty. Aluksi valitsemme käsittelemättömiä solmun 1, koska sen etäisyys 0 on pienin. Sitten jäljellä ovat solmut 2, 3, 4 ja 5, joista valitsemme käsittelemättömiä solmun 3, jonka etäisyys 2 on pienin. Tämän jälkeen valitsemme käsittelemättömiä solmun 2, jonka etäisyys on 6. Sama jatkuu,

kunnes olemme käsitelleet kaikki verkon solmut.

Dijkstran algoritmissa etsimme n kertaa käsittelemättömän solmun, jonka etäisyysarvio on pienin. Koska haluamme saada algoritmista tehokkaan, meidän täytyy pystyä löytämään solmut nopeasti. Tavallinen tapa toteuttaa Dijkstran algoritmi on käyttää *kekoa*, jonka avulla löydämme joka vaiheessa pienimmän etäisyyden solmun logaritmisessa ajassa. Tallennamme kekaan pareja, joissa on solmun etäisyys ja tunnus ja jotka järjestetään etäisyyden mukaan pienimmästä suurimpaan. Aluksi keossa on vain lähtösolmua vastaava solmu, jonka etäisyys on 0. Tämän jälkeen haemme joka askeleella keosta solmun, jonka etäisyys on pienin. Jos solmu on jo käsitelty, emme tee mitään. Muuten käymme läpi kaikki solmusta lähtevät kaaret ja tarkastamme, voimmeko parantaa etäisyyksiä niiden avulla. Aina kun voimme parantaa etäisyyttä, lisäämme uuden etäisyyden kekaan.

Dijkstran algoritmi on mukavaa toteuttaa käyttäen verkon vieruslistaesitystä. Voimme toteuttaa algoritmin seuraavasti:

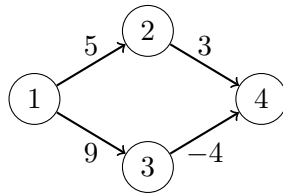
```
keko.push((0,alku))
while not keko.empty()
    solmu = keko.pop()[1]
    if kasitelty[solmu]
        continue
    kasitelty[solmu] = true
    for kaari in verkko[solmu]
        nyky = etaisyys[kaari.loppu]
        uusi = etaisyys[solmu]+kaari.paino
        if uusi < nyky
            etaisyys[kaari.loppu] = uusi
            keko.push((uusi,kaari.loppu))
```

Huomaa, että keossa voi olla samaan aikaan *useita* etäisyyksiä samalle solmulle, koska lisäämme kekaan uuden solmun aina etäisyyden parantuessa. Käsittelemme kuitenkin jokaisen solmun vain kerran, koska aina kun olemme hakeneet uuden solmun keosta käsittelyä varten, varmistamme ensin, että emme ole käsitelleet sitä aiemmin.

Algoritmin analyysi

Dijkstran algoritmi on ahne algoritmi, koska se etsii joka vaiheessa vielä käsittelemättömän solmun, jonka etäisyys on pienin, minkä jälkeen kyseisen solmun etäisyys ei enää muutu. Miten voimme olla varmoja, että olemme löytäneet tässä vaiheessa oikean etäisyyden?

Voimme ajatella asiaa siltä kannalta, että jos etäisyyttä olisi mahdollista



Kuva 11.5: Dijkstran algoritmi ei toimi oikein negatiivisen kaaren takia.

parantaa, niin verkossa olisi oltava jokin toinen vielä käsittelemätön solmu, jonka kautta voisimme muodostaa lyhemmän polun. Kuitenkin tiedämme, että kaikkien muiden tarjolla olevien solmujen etäisyydet ovat suurempia tai yhtä suuria eivätkä etäisyydet voi lyhentyä, koska verkossa ei ole negatiivisia kaaria. Tästä syystä voimme turvallisesti valita käsittelyyn pienimmän etäisyyden solmun ja kiinnittää sen etäisyyden.

Dijkstran algoritmi toimii siis oikein, jos verkossa ei ole negatiivisia kaaria, mutta kuinka nopeasti algoritmi toimii? Ensinnäkin algoritmi käy läpi verkon solmut ja kaaret, missä kuluu aikaa $O(n + m)$. Lisäksi algoritmista on joukko kekon liittyviä operaatioita, jotka vaikuttavat tehokkuuteen. Pahimmassa tapauksessa lisäämme jokaisen kaaren kohdalla kekon uuden alkion, eli lisäykset kekon vievät aikaa $O(m \log m)$. Toisaalta poistamme kaikki alkiot aikanaan keosta, mihin menee myös aikaa $O(m \log m)$. Algoritmin kokonaisaikaavaativuus on siis $O(n + m \log m)$.

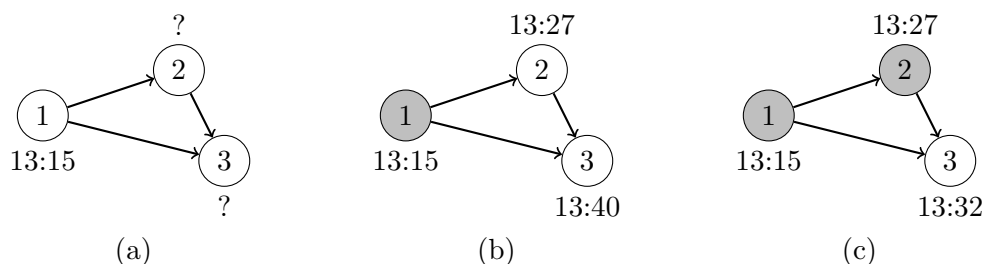
Voimme vielä hieman siistiä aikaavaativuutta, kun teemme luontevan oletuksen, että verkossa ei ole kahta kaarta, joiden alku- ja loppusolmu ovat samat. Tällöin $m \leq n^2$, jolloin $\log m = \log(n^2) = 2 \log n$ ja saamme algoritmin aikaavaativuudeksi $O(n + m \log n)$.

Entä mitä tapahtuu, jos verkossa kuitenkin on negatiivinen kaari? Tällöin Dijkstran algoritmi ei toimi välttämättä oikein. Kuva 11.5 näyttää esimerkin tällaisesta tilanteesta. Algoritmi seuraa ahneesti ylem্পää polkua ja toteaa, että pienin etäisyys solmusta 1 solmuun 4 on 8. Kuitenkin parempi tapa olisi kulkea alem্পaa polkua, jolloin polun pituus on vain 5.

11.1.3 Esimerkki: Reittiopas

Nyt meillä on tarvittavat tiedot, joiden avulla voimme luoda *reittioppaan* eli järjestelmän, jonka avulla voi etsiä eri kulkuvälineitä käyttäviä reittejä kaupungissa. Voimme mallintaa kaupungin verkkona, jonka solmut ovat kaupungin paikkoja ja kaaret ovat mahdollisia yhteyksiä paikkojen välillä. Reittioppaan tulisi ilmoittaa *nopein* reitti paikasta a paikkaan b .

Reittioppaan toteuttamiseen liittyy yksi lisähaaste: kulkuvälineillä on tietyt aikataulut, jotka rajoittavat niiden käyttöä. Esimerkiksi bussilinjan



Kuva 11.6: Nopeimman reitin etsiminen paikasta 1 paikkaan 3. (a) Matka alkaa kello 13:15. (b) Käymme läpi paikasta 2 lähtevät yhteydet. (c) Käymme läpi paikasta 3 lähtevät yhteydet.

lähtöjä saattaa olla 10 minuutin välein. Meidän tulee siis ottaa huomioon reitin etsimisessä, mihin aikaan saavumme mihinkin paikkaan. Voimme toteuttaa tämän tallentamalla verkon kaaret muodossa ”matka alkaa ajanhetkenä x ja päättyy ajanhetkenä y ”.

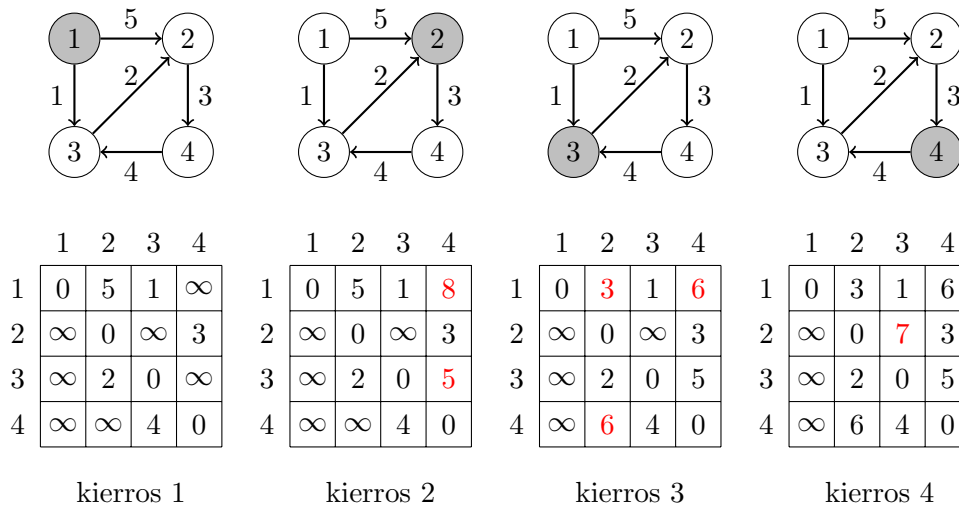
Koska kaikissa matkoissa kuluu positiivinen määrä aikaa, voimme etsiä reittejä Dijkstran algoritmin avulla. Toteutamme algoritmin niin, että määritämme jokaiseen paikkaan *varhaisimman* ajan, jolloin voimme päästä kyseiseen paikkaan. Alussa tiedämme, että olemme lähtöpaikassa matkamme alkuhetkenä. Sitten kun otamme käsittelyyn uuden paikan, käymme läpi siitä lähteviä yhteyksiä, joihin ehdimme optimaalista aikataulua noudattaen. Saamme tehostettua hakua merkittävästi ottamalla huomioon jokaisesta linjasta vain ensimmäisen lähdön, johon ehdimme. Tämä on perusteltua, koska ei voi missään tilanteessa olla hyvä idea odottaa myöhempään lähtöön.

Kuva 11.6 näyttää esimerkkitalanteen, jossa haluamme kulkea paikasta 1 paikkaan 3 ja lähdemme matkaan kello 13:15. Paikasta 1 lähtee 10 minuutin välein yhteys paikkaan 2 (kesto 7 minuuttia) sekä 30 minuutin välein yhteys paikkaan 3 (kesto 10 minuuttia). Niinpä pääsemme paikkaan 2 kello 13:27 ja paikkaan 3 kello 13:40. Sitten paikasta 2 lähtee 5 minuutin välein yhteys paikkaan 3 (kesto 2 minuuttia). Tämän avulla pääsemme paikkaan 3 kello 13:32, eli nopein reitti kulkee paikan 2 kautta.

Käytännössä hakua voisi vielä tehostaa monella tavalla. Esimerkiksi kun löydämme jonkin reitin kohdepaikkaan, voimme siitä lähtien hylätä kaikki paikat, joihin ehdimme myöhemmin kuin kohdepaikkaan.

11.2 Kaikki lyhimmat polut

Tarkastellaan seuraavaksi ongelmaa, jossa haluamme etsiä lyhimmat polut verkon *kaikista* solmuista *kaikkiin* solmuihin. Yksi tapa ratkaista tehtävä oli-



Kuva 11.7: Esimerkki Floyd–Warshallin algoritmin toiminnasta.

si suorittaa Bellman–Fordin tai Dijkstran algoritmi jokaisesta verkon solmusta alkaen. Voimme kuitenkin ratkaista tehtävän suoremmin etsimällä kaikki polut *samanaikaisesti* Floyd–Warshallin algoritmilla.

11.2.1 Floyd–Warshallin algoritmi

Floyd–Warshallin algoritmi muodostaa $n \times n$ -kokoisen *etäisyysmatriisin* (*distance matrix*), jossa rivin a sarakkeessa b on lyhimmän polun pituus solmusta a solmuun b . Algoritmi alustaa ensin matriisin niin, että siihen on merkitty vain etäisyydet, jotka toteutuvat kulkemalla yksittäistä kaarta, ja kaikissa muissa matriisin kohdissa etäisyys on ääretön. Sitten algoritmi suorittaa n kierrosta, jotka on numeroitu $1, 2, \dots, n$. Kierroksella k algoritmi etsii polkuja, joissa on välisolmuna solmu k sekä mahdollisesti muita välisolmuja joukosta $1, 2, \dots, k-1$. Jos tällainen polku parantaa etäisyyttä, päivitämme uuden etäisyyden matriisiin. Lopulta jokainen solmu on ollut välisolmuna poluilla, jolloin olemme saaneet selville kaikki lyhimvät polut.

Kuva 11.7 näyttää esimerkin Floyd–Warshallin algoritmin toiminnasta. Kierroksella 1 etsimme polkuja, joissa solmu 1 on välisolmuna. Tällaisia polkuja ei ole, koska solmuun 1 ei pääse mistään solmusta, joten matriisi ei muutu. Kierroksella 2 huomaamme, että voimme kulkea solmun 2 kautta solmusta 1 solmuun 4, jolloin saamme etäisyyden 8. Samoin voimme kulkea solmun 2 kautta solmusta 3 solmuun 4, jolloin saamme etäisyyden 5. Jatkamme vastaavasti, kunnes kierroksen 4 jälkeen olemme saaneet selville kaikki etäisyydet ja etäisyysmatriisi on lopullinen.

Floyd–Warshallin algoritmin mukavana puolena on, että se on hyvin helppoa toteuttaa. Meidän riittää luoda kolme sisäkkäistä for-silmukkaa, jotka toteuttavat matriisin päivitykset. Seuraavassa koodissa muuttuja k kertoo, mikä kierros on kyseessä eli mitä solmua käytämme välisolmuna. Jokaisella kierroksella käymme läpi kaikki solmuparit (i, j) ja koetamme parantaa niiden etäisyyksiä kulkemalla solmun k kautta.

```

for k = 1 to n
  for i = 1 to n
    for j = 1 to n
      etaisyys[i][j] = min(etaisyys[i][j],
                           etaisyys[i][k]+etaisyys[k][j])

```

Algoritmin aikavaativuus on selkeästi $O(n^3)$, koska se muodostuu kolmesta sisäkkäisestä for-silmukasta.

Algoritmin analyysi

Miksi Floyd–Warshallin algoritmi toimii? Voimme ymmärtää algoritmia tarkastelemalla sen toimintaa ”käänteisesti” rekursiivisesti. Kun verkossa on lyhin polku solmusta a solmuun b , millainen tämä polku voi olla?

Yksi mahdollisuus on, että polku on vain kaari solmusta a solmuun b . Tällöin sen pituus on merkitty etäisyysmatriisiin algoritmin alussa. Muussa tapauksessa polussa on yksi tai useampi välisolmu. Oletetaan, että x on välisolmu, jonka tunnus on suurin. Saamme nyt kaksi osatehtävää: meidän tulee ensin kulkea solmusta a solmuun x ja sen jälkeen solmusta x solmuun b niin, että kummallakin polulla jokaisen välisolmun tunnus on pienempi kuin x . Voimme käsitellä nämä osatehtävät rekursiivisesti.

Floyd–Warshallin algoritmi muodostaa joka vaiheessa polkuja, joissa voi olla välisolmuina solmuja $1, 2, \dots, i$. Kun haluamme muodostaa lyhimmän polun solmusta a solmuun b , meillä on kaksi vaihtoehtoa: Jos solmu i on välisolmuna, yhdistämme lyhimmän polun solmusta a solmuun i ja solmusta i solmuun b . Jos taas solmu i ei ole välisolmuna, olemme käsitelleet polun jo aiemmin. Algoritmin päätteeksi välisolmuina voi olla solmuja $1, 2, \dots, n$, eli mikä tahansa verkon solmu voi olla välisolmu.

11.2.2 Algoritmien vertailua

Olemme nyt käyneet läpi monia algoritmeja lyhimpien polkujen etsintään ja voimme alkaa muodostaa yleiskuvaa aiheesta. Taulukko 11.1 näyttää yhteenvedon algoritmien tehokkuudesta ja ominaisuuksista.

algoritmi	aikavaativuus	erityistä
leveyshaku	$O(n + m)$	ei salli painoja kaarissa
Bellman–Fordin algoritmi	$O(nm)$	
Dijkstran algoritmi	$O(n + m \log n)$	ei salli negatiivisia kaaria
Floyd–Warshallin algoritmi	$O(n^3)$	etsii kaikki polut

Taulukko 11.1: Algoritmit lyhimpien polkujen etsimiseen.

Käytännössä leveyshaku ja Dijkstran algoritmi ovat yleisimmin tarvittavat algoritmit: jos kaarilla ei ole painoja, käytämme leveyshakua, ja muuten Dijkstran algoritmia. Dijkstran algoritmin rajoituksena on, että verkossa ei saa olla negatiivisia kaaria, mutta tällä rajoituksella ei ole yleensä merkitystä käytännön ongelmissa, koska kaarten painot eivät useimmiten voi olla negatiivisia. Esimerkiksi selvästikään tien pituus tai lennon hinta ei voi olla negatiivinen. Jos kuitenkin verkossa voi olla negatiivisia kaaria, voimme turvautua Bellman–Fordin algoritmiin.

Miten sitten Floyd–Warshallin algoritmi vertautuu muihin algoritmeihin? Tämä riippuu siitä, onko verkko *harva* (*sparse*) vai *tiheä* (*dense*). Harvassa verkossa on vähän kaaria ja $m \approx n$, kun taas tiheässä verkossa on paljon kaaria ja $m \approx n^2$. Floyd–Warshallin algoritmi on parhaimmillaan silloin, kun verkko on tiheä, koska sen aikavaativuus ei riipu kaarten määrästä. Esimerkiksi jos etsimme kaikki lyhimmät polut suorittamalla n kertaa Dijkstran algoritmin, harvassa verkossa aikaa kuluu $O(n^2 \log n)$, mutta tiheässä verkossa aikaa kuluu $O(n^3 \log n)$. Siis harvassa verkossa aikavaativuus on parempi kuin Floyd–Warshallin algoritmilla, mutta tiheässä verkossa se on huonompi. Toisaalta Floyd–Warshallin algoritmin vakiokertoimet ovat hyvin pienet sen yksinkertaisen rakenteen ansiosta, minkä ansiosta algoritmi voi toimia käytännössä yllättävänkin nopeasti.

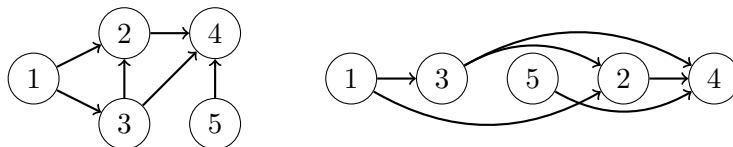
Luku 12

Suunnatut syklittömät verkot

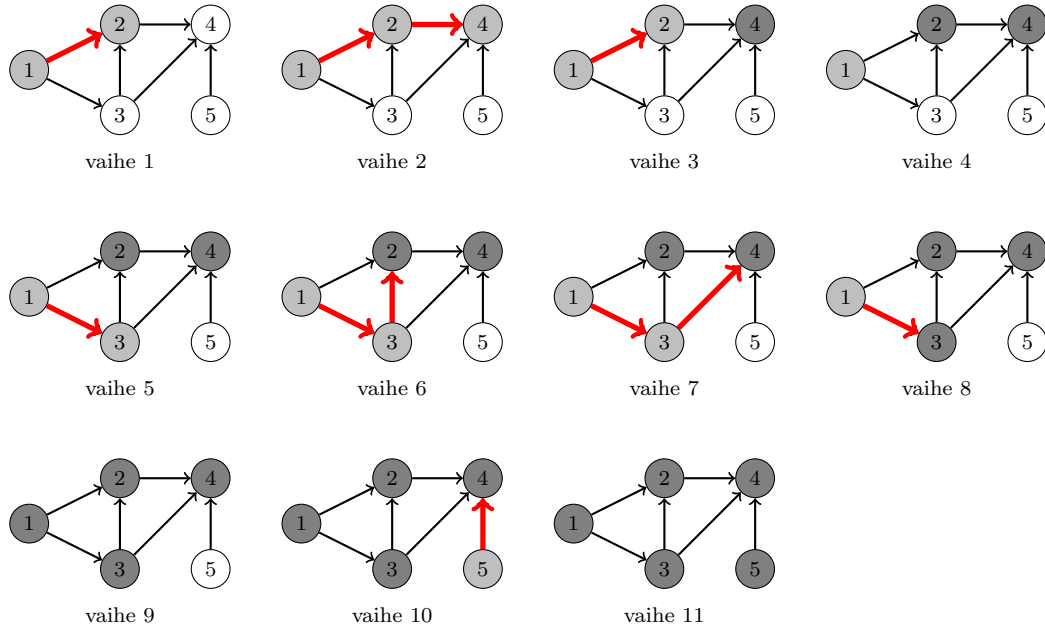
Verkkoalgoritmien suunnittelussa aiheuttavat usein vaikeuksia verkossa olevat syklit, ja monen ongelman ratkaiseminen on vaikeaa nimenomaan sen takia, että meidän täytyy ottaa huomioon, mitä tapahtuu sykleissä. Tässä luvussa katsomme, miten asiat muuttuvat, kun voimmekin olettaa, että käsiteltävänä on suunnattu verkko, jossa *ei ole* syklejä.

Kun verkko on suunnattu ja syklitön, voimme muodostaa sille aina *topologisen järjestyksen*, joka antaa luontevan järjestyksen käsitellä verkon solmut niiden riippuvuuksien mukaisesti. Tämän ansiosta voimme hyödyntää dynaamista ohjelmointia verkon polkujen käsittelyssä. Itse asiassa tulemme huomaamaan, että *mikä tahansa* dynaamisen ohjelmoinnin algoritmi voidaan nähdä suunnatun syklittömän verkon käsittelynä.

Entä jos verkossa kuitenkin on syklejä? Osoittautuu, että voimme silti esittää sen syvärakenteen syklittömänä verkkona muodostamalla verkon *vahvasti yhtenäiset komponentit*. Tämän ansiosta voimme tietyissä tilanteissa käsitellä verkkoa mukavasti syklittömän verkon tavoin, vaikka se ei olisi-kaan alun perin syklitön.



Kuva 12.1: Verkko ja yksi sen topologinen järjestys $[1, 3, 5, 2, 4]$.



Kuva 12.2: Esimerkki topologisen järjestyksen muodostamisesta.

12.1 Topologinen järjestys

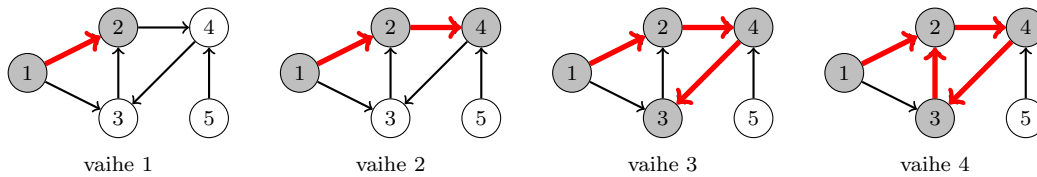
Topologinen järjestys (*topological sort*) on suunnatun verkon solmujen järjestys, jossa pätee, että jos solmusta a on kaari solmuun b , niin solmu a on ennen solmua b järjestyksessä. Topologinen järjestys voidaan esittää listana, joka ilmaisee solmujen järjestyksen. Kuvassa 12.1 on esimerkkinä verkko ja yksi sen topologinen järjestys $[1, 3, 5, 2, 4]$.

Osoittautuu, että voimme muodostaa suunnatulle verkolle topologisen järjestyksen tarkalleen silloin, kun verkko on syklitön (*directed acyclic graph* eli *DAG*). Tutustumme seuraavaksi tehokkaaseen algoritmiin, joka muodostaa topologisen järjestyksen tai toteaa, ettei järjestystä voi muodostaa verkossa olevan syklin takia.

12.1.1 Järjestyksen muodostaminen

Voimme muodostaa topologisen järjestyksen suorittamalla joukon syvyyshakuja, joissa jokaisella solmulla on kolme mahdollista tilaa:

- tila 0 (valkoinen): solmussa ei ole käyty
- tila 1 (harmaa): solmun käsittely on kesken
- tila 2 (musta): solmun käsittely on valmis



Kuva 12.3: Topologista järjestystä ei voi muodostaa syklin takia.

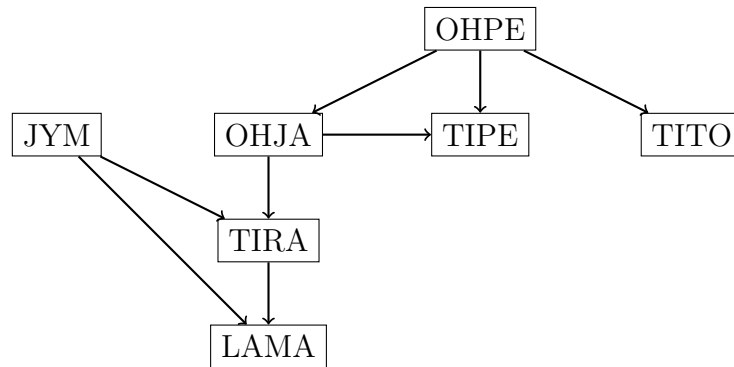
Algoritmin alussa jokainen solmu on valkoinen. Käymme läpi kaikki verkon solmut ja aloitamme aina syvyysshaun solmusta, jos se on valkoinen. Aina kun saavumme uuteen solmuun, sen väri muuttuu valkoisesta harmaaksi. Sitten kun olemme käsitelleet kaikki solmusta lähtevät kaaret, solmun väri muuttuu harmaasta mustaksi ja lisäämme solmun listalle. Tämä lista käänteisessä järjestyksessä on verkon topologinen järjestys. Kuitenkin jos saavumme jossain vaiheessa toista kautta harmaaseen solmuun, verkossa on sykli eikä topologista järjestystä ole olemassa.

Kuva 12.2 näyttää, kuinka algoritmi muodostaa topologisen järjestyksen esimerkiverkossamme. Tässä tapauksessa suoritamme kaksi syvyyshakua, joista ensimmäinen alkaa solmusta 1 ja toinen alkaa solmusta 5. Algoritmin tuloksena on lista $[4, 2, 3, 1, 5]$, joten käänteinen lista $[5, 1, 3, 2, 4]$ on verkon topologinen järjestys. Huomaa, että tämä on eri järjestys kuin kuvassa 12.1 – topologinen järjestys ei ole yksikäsitteinen ja voimme yleensä muodostaa järjestyksen monella tavalla. Kuva 12.3 näyttää puolestaan tilanteen, jossa topologista järjestystä ei voi muodostaa verkossa olevan syklin takia. Tässä verkossa on sykli $2 \rightarrow 4 \rightarrow 3 \rightarrow 2$, jonka olemassaolon huomaamme siitä, että tulemme uudestaan harmaaseen solmuun 2.

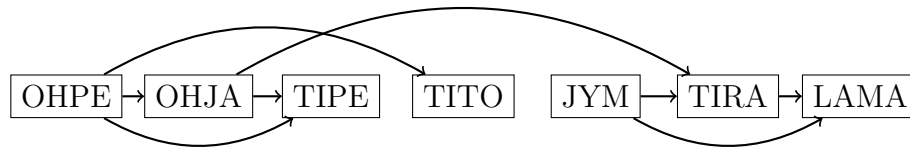
Algoritmin analyysi

Miksi algoritmi toimii? Tarkastellaan ensin tilannetta, jossa verkossa on sykli. Jos algoritmi saapuu uudestaan harmaaseen solmuun, on selvää, että verkossa on sykli, koska algoritmi on onnistunut pääsemään harmaasta solmusta itseensä kulkemalla jotain polkua verkossa. Toisaalta jos verkossa on sykli, algoritmi tulee jossain vaiheessa ensimmäistä kertaa johonkin syklin solmuun x . Tämän jälkeen se käy läpi solmusta lähtevät kaaret ja aikanaan käy varmasti läpi kaikki syklin solmut ja saapuu uudestaan solmuun x . Niinpä algoritmi tunnistaa kaikissa tilanteissa oikein verkossa olevan syklin.

Jos sitten verkossa ei ole sykliä, algoritmi lisää jokaisen solmun listalle sen jälkeen, kun se on käsitellyt kaikki solmusta lähtevät kaaret. Jos siis verkossa on kaari $a \rightarrow b$, solmu b lisätään listalle ennen solmua a . Lopuksi lista käännetään, jolloin solmu a tulee ennen solmua b . Tämän ansiosta jokaiselle



Kuva 12.4: Kurssien esitietovaatimukset verkkona.



Kuva 12.5: Topologinen järjestys antaa kurssien suoritusjärjestyksen.

kaarelle $a \rightarrow b$ pätee, että solmu a tulee järjestykseen ennen solmua b , eli algoritmi muodostaa kelvollisen topologisen järjestyksen.

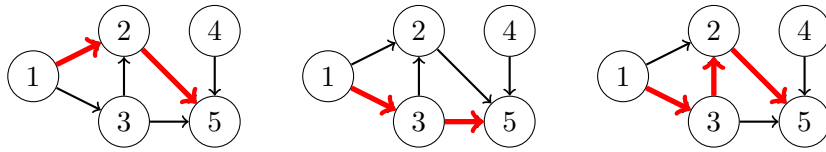
Algoritmin aikavaativuus on $O(n + m)$, koska se käy kaikki verkon solmut ja kaaret läpi syvyyshaun avulla.

12.1.2 Esimerkki: Kurssivalinnat

Yliopiston kurssit ja niiden esitietovaatimukset voidaan esittää suunnattuna verkkona, jonka solmut ovat kursseja ja kaaret kuvaavat, missä järjestyksessä kurssit tulisi suorittaa.

Kuvassa 12.4 on esimerkkinä joitakin tietojenkäsittelytieteen kandiohjelman kursseja. Tällaisen verkon topologinen järjestys antaa meille yhden tavan suorittaa kurssit esitietovaatimusten mukaisesti. Kuvassa 12.5 näkyy esimerkkinä topologinen järjestys, joka vastaa suoritusjärjestyksestä OHPE, OHJA, TIPE, TITO, JYM, TIRA, LAMA.

On selvää, että kurssien ja esitietovaatimusten muodostaman verkon tulee olla syklitön, jotta kurssit voi suorittaa halutulla tavalla. Jos verkossa on sykli, topologista järjestystä ei ole olemassa eikä meillä ole mitään mahdollisuutta suorittaa kursseja esitietovaatimusten mukaisesti.



Kuva 12.6: Mahdolliset polut solmusta 1 solmuun 5.

12.2 Dynaaminen ohjelmointi

Kun tiedämme, että suunnattu verkko on syklitön, voimme ratkaista helposti monia verkon polkuihin liittyviä ongelmia *dynaamisen ohjelmoinnin* avulla. Tämä on mahdollista, koska topologinen järjestys antaa meille selkeän järjestyksen, jossa voimme käsitellä solmut.

12.2.1 Polkujen laskeminen

Tarkastellaan esimerkkinä ongelmaa, jossa haluamme laskea, montako polkua suunnatussa syklittömässä verkossa on solmusta 1 solmuun n . Esimerkiksi kuvassa 12.6 solmusta 1 solmuun 5 on kolme mahdollista polkua: $1 \rightarrow 2 \rightarrow 5$, $1 \rightarrow 3 \rightarrow 5$ ja $1 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Polkujen määrän laskeminen on vaikea ongelma yleisessä verkossa, jossa voi olla syklejä. Itse asiassa tehtävä ei ole edes mielekäs sellaisenaan: jos verkossa on sykli, voimme kiertää sykliä miten monta kertaa tahansa ja tuottaa aina vain uusia polkuja, joten polkuja tulee äärettömästi. Nyt kuitenkin oletamme, että verkko on syklitön, jolloin polkujen määrä on rajoitettu ja voimme laskea sen tehokkaasti dynaamisella ohjelmoinnilla.

Jotta voimme käyttää dynaamista ohjelmointia, meidän täytyy määritellä ongelma rekursiivisesti. Sopiva funktio on $\text{polut}(x)$, joka antaa polkujen määrän solmusta 1 solmuun x . Tätä funktiota käyttäen $\text{polut}(n)$ kertoo, montako polkua on solmusta 1 solmuun n . Esimerkiksi kuvan 12.6 tilanteessa funktion arvot ovat seuraavat:

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

Nyt meidän täytyy enää löytää tapa laskea funktion arvoja. Pohjatapauksessa olemme solmussa 1, jolloin on aina yksi tyhjä polku:

$$\text{polut}(1) = 1$$

Entä sitten, kun olemme jossain muussa solmussa x ? Tällöin käymme läpi kaikki solmut, joista pääsemme solmuun x kaarella, ja laskemme yhteen näihin solmuihin tulevien polkujen määrät. Kun oletamme, että solmuun x on kaari solmuista u_1, u_2, \dots, u_k , saamme seuraavan rekursiivisen kaavan:

$$\text{polut}(x) = \text{polut}(u_1) + \text{polut}(u_2) + \dots + \text{polut}(u_k)$$

Esimerkiksi kuvan 12.6 verkossa solmuun 5 on kaari solmuista 2, 3 ja 4, joten

$$\text{polut}(5) = \text{polut}(2) + \text{polut}(3) + \text{polut}(4) = 2 + 1 + 0 = 3.$$

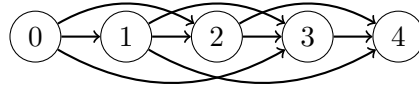
Koska tiedämme, että verkko on sykliton, voimme laskea funktion arvoja tehokkaasti dynaamisella ohjelmoinnilla. Oleellista on, että emme voi joutua koskaan silmukkaan laskeessamme arvoja, ja käytännössä laskemme arvot jossakin solmujen topologisessa järjestyksessä.

12.2.2 Ongelmat verkkoina

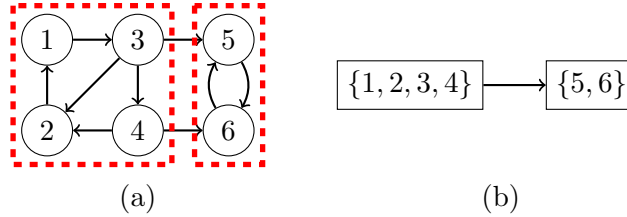
Itse asiassa voimme esittää *minkä tahansa* dynaamisen ohjelmoinnin algoritmin suunnatun syklittömän verkon käsittelynä. Ideana on, että muodostamme verkon, jossa jokainen solmu on yksi osaongelma ja kaaret ilmaisevat, miten osaongelmat liittyvät toisiinsa.

Tarkastellaan esimerkkinä luvusta 9.1 tuttua tehtävää, jossa haluamme laskea, monellako tavalla voimme muodostaa korkeuden n tornin, kun voimme käyttää palikoita, joiden korkeudet ovat 1, 2 ja 3. Voimme esittää tämän tehtävän verkkona niin, että solmut ovat tornien korkeuksia ja kaaret kertovat, kuinka voimme rakentaa tornia palikoista. Jokaisesta solmusta x on kaari solmuihin $x + 1$, $x + 2$ ja $x + 3$, ja polkujen määrä solmusta 0 solmuun n on yhtä suuri kuin tornin rakentamistapojen määrä. Esimerkiksi kuva 12.7 näyttää verkon, joka vastaa tapausta $n = 4$. Solmusta 0 solmuun 4 on yhteensä 7 polkua, eli voimme rakentaa korkeuden 4 tornin 7 tavalla.

Olemme saaneet siis uuden tavan luonnehtia dynaamista ohjelmointia: *voimme käyttää dynaamista ohjelmointia, jos pystymme esittämään ongelman suunnattuna syklittömänä verkkona.*



Kuva 12.7: Tornitehtävän tapaus $n = 4$ esitettynä verkkona.



Kuva 12.8: (a) Verkon vahvasti yhtenäiset komponentit. (b) Komponentti-verkko, joka kuvaa verkon syvärakenteen.

12.3 Vahvasti yhtenäisyys

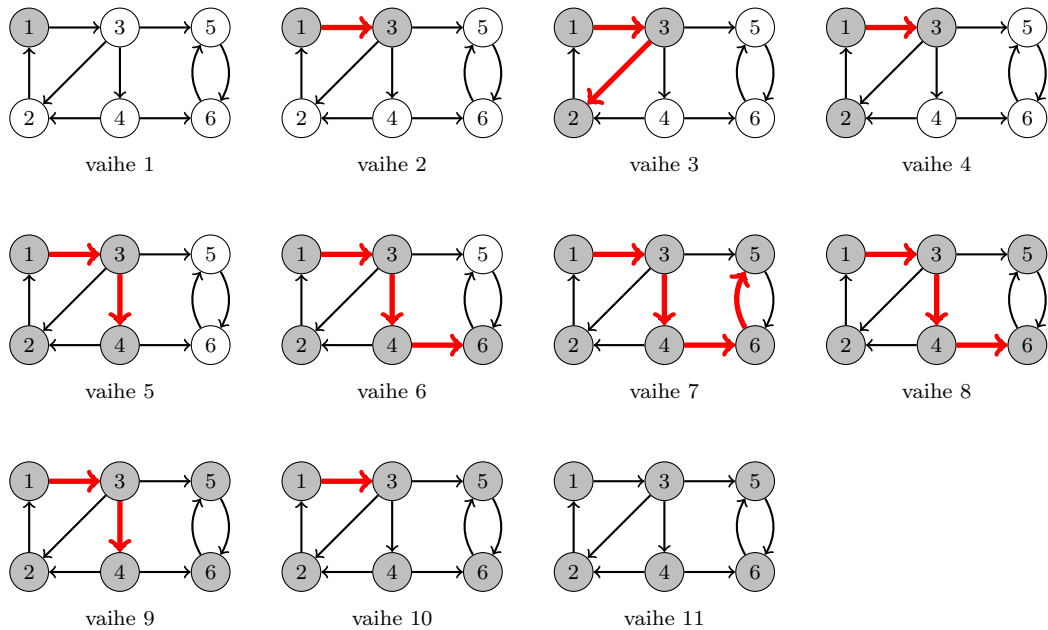
Jos suunnatussa verkossa on sykli, emme voi muodostaa sille topologista järjestystä emmekä käyttää dynaamista ohjelmointia. Mikä neuvoksi, jos kuitenkin haluaisimme tehdä näin?

Joskus voimme selviytyä tilanteesta käsittelemällä verkon vahvasti yhtenäisiä komponentteja. Sanomme, että suunnattu verkko on *vahvasti yhtenäinen* (*strongly connected*), jos mistä tahansa solmusta on polku mihin tahansa solmuun. Voimme esittää suunnatun verkon aina yhtenä tai useampana vahvasti yhtenäisenä komponenttina, joista muodostuu syklitön *komponenttiverkko*. Tämä verkko esittää alkuperäisen verkon syvärakenteen.

Kuvassa 12.8 on esimerkkinä verkko, joka muodostuu kahdesta vahvasti yhtenäisestä komponentista. Ensimmäinen komponentti on $\{1, 2, 3, 4\}$ ja toinen komponentti on $\{5, 6\}$. Komponenteista muodostuu syklitön komponenttiverkko, jossa on kaari solmusta $\{1, 2, 3, 4\}$ solmuun $\{5, 6\}$. Tämä tarkoittaa, että voimme liikkua miten tahansa joukon $\{1, 2, 3, 4\}$ solmuissa sekä joukon $\{5, 6\}$ solmuissa. Lisäksi pääsemme joukosta $\{1, 2, 3, 4\}$ joukkoon $\{5, 6\}$, mutta emme pääse takaisin joukosta $\{5, 6\}$ joukkoon $\{1, 2, 3, 4\}$.

12.3.1 Kosarajun algoritmi

Kosarajun algoritmi on tehokas algoritmi, joka muodostaa suunnatun verkon vahvasti yhtenäiset komponentit. Algoritmissa on kaksi vaihetta, joista kumpikin käy läpi verkon solmut syvyyshaulla. Ensimmäinen vaihe muistuttaa topologisen järjestyksen etsimistä ja tuottaa listan solmuista. Toinen vaihe



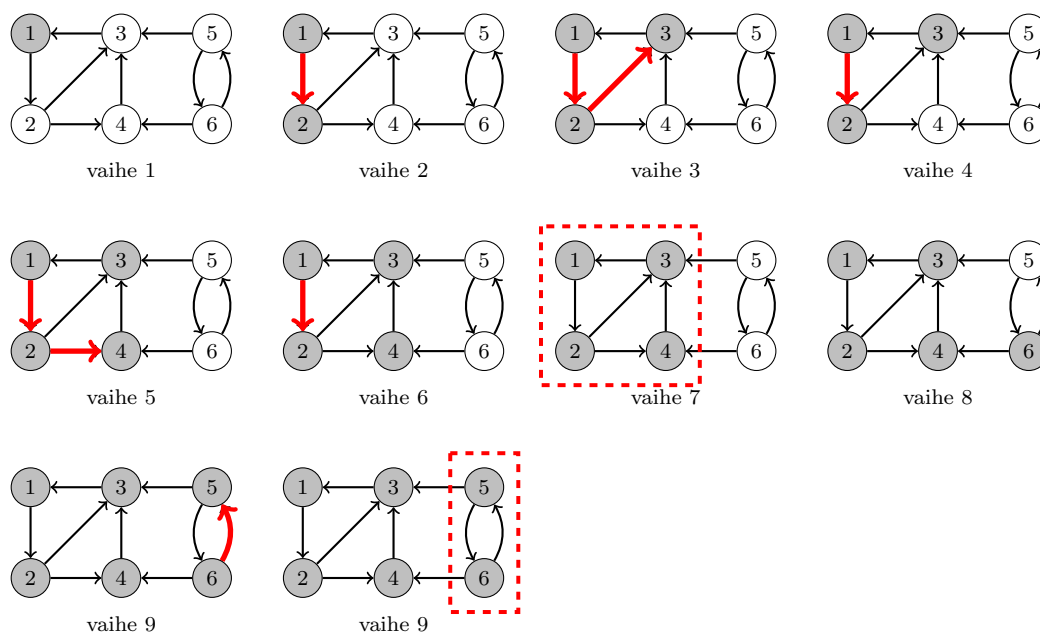
Kuva 12.9: Kosarajun algoritmin ensimmäinen vaihe.

muodostaa vahvasti yhtenäiset komponentit tämän listan perusteella.

Algoritmin ensimmäisessä vaiheessa käymme läpi verkon solmut ja aloitamme uuden syvyyshaun aina, jos emme ole vielä käyneet solmussa. Syvyyshaun aikana lisäämme solmun listalle, kun olemme käyneet läpi kaikki solmusta lähtevät kaaret. Toimimme siis kuten topologisen järjestyksen muodostamisessa, mutta emme välitä, jos tulemme toista reittiä solmuun, jota ei ole vielä käsitelty loppuun.

Algoritmin toisen vaiheen alussa käännämme jokaisen verkon kaaren suunnan. Tämän jälkeen käymme läpi käänteisessä järjestyksessä listalla olevat solmut. Aina kun vuoroon tulee solmu, jossa emme ole vielä käyneet, aloitamme siitä solmusta syvyyshaun, joka muodostaa uuden vahvasti yhtenäisen komponentin. Lisäämme komponenttiin kaikki solmut, joihin pääsemme syvyyshaun aikana ja jotka eivät vielä kuulu mihinkään komponenttiin.

Tarkastellaan seuraavaksi, kuinka Kosarajun algoritmi toimii esimerkkiverkossamme. Kuva 12.9 näyttää algoritmin ensimmäisen vaiheen, joka muodostaa solmuista listan $[2, 5, 6, 4, 3, 1]$. Kuva 12.10 näyttää algoritmin toisen vaiheen, jossa käännämme ensin verkon kaaret ja käymme sitten läpi solmut järjestyksessä $[1, 3, 4, 6, 5, 2]$. Vahvasti yhtenäiset komponentit syntyvät solmuista 1 ja 6 alkaen. Kaarten kääntämisen ansiosta solmusta 1 alkava vahvasti yhtenäinen komponentti ei ”vuoda” solmujen 5 ja 6 alueelle.



Kuva 12.10: Kosarajun algoritmin toinen vaihe.

Algoritmin analyysi

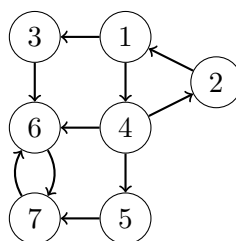
Kosarajun algoritmin kriittinen osa on sen toinen vaihe, jossa algoritmi muodostaa vahvasti yhtenäiset komponentit. On selvää, että jokainen syvyyssha-ku löytää vahvasti yhtenäiseen komponenttiin kuuluvat solmut, mutta miksi komponenttiin ei voi tulla lisäksi ylimääräisiä solmuja?

Voimme tarkastella asiaa muodostettavan komponenttiverkon näkökulmasta. Jos meillä on komponentti A , josta pääsee kaarella komponenttiin B , algoritmin ensimmäisessä vaiheessa jokin A :n solmu lisätään listalle kaikkien B :n solmujen jälkeen. Kun sitten käymme läpi listan käänteisessä järjestyksessä, jokin A :n solmu tulee vastaan ennen kaikkia B :n solmuja. Niinpä alamme rakentaa ensin komponenttia A emmekä mene komponentin B puolelle, koska verkon kaaret on käännetty. Sitten kun myöhemmin muodostamme komponentin B , emme mene käännettyä kaarta komponenttiin A , koska komponentti A on jo muodostettu.

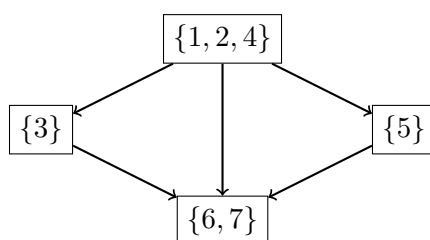
Algoritmin aikavaativuus on $O(n + m)$, koska se käy kahdesti verkon solmut ja kaaret läpi syvyysshaun avulla.

12.3.2 Esimerkki: Luolapeli

Olemme pelissä luolastossa, joka muodostuu n luolasta ja joukosta käytäviä niiden välillä. Jokainen käytävä on yksisuuntainen. Jokaisessa luolassa on



Kuva 12.11: Luolasto, jossa on 7 luolaa ja 10 käytävää. Haluamme kulkea luolasta 1 luolaan 7 keräten mahdollisimman paljon aarteita.



Kuva 12.12: Luolaston vahvasti yhtenäiset komponentit.

yksi aarre, jonka voimme ottaa mukaamme, jos kuljemme luolan kautta. Peli alkaa luolasta 1 ja päättyy luolaan n . Montako aarretta voimme saada, jos valitsemme parhaan mahdollisen reitin? On sallittua kulkea saman luolan kautta monta kertaa tarvittaessa.

Voimme mallintaa tilanteen verkkona, jonka solmut ovat luolia ja kaaret ovat käytäviä. Haluamme löytää reitin solmusta 1 solmuun n niin, että kuljemme mahdollisimman monen solmun kautta. Esimerkiksi kuva 12.11 näyttää verkkona luolaston, joka muodostuu seitsemästä luolasta ja kymmenestä käytävästä. Yksi optimaalinen reitti luolasta 1 luolaan 7 on $1 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7$, jota seuraten saamme kerättyä kaikki aarteet paitsi luolassa 5 olevan aarteen. Ei ole olemassa reittiä, jota noudattamalla saisimme haltuunme kaikki luolaston aarteet.

Voimme ratkaista ongelman tehokkaasti määrittämällä ensin verkon vahvasti yhtenäiset komponentit. Tämän jälkeen riittää löytää sellainen polku alkusolmun komponentista loppusolmun komponenttiin, että komponenttien kokojen summa on suurin mahdollinen. Koska verkko on sykliton, tämä onnistuu dynaamisella ohjelmoinnilla.

Kuva 12.12 näyttää vahvasti yhtenäiset komponentit esimerkkiverkossamme. Tästä esityksestä näemme suoraan, että optimaalisia reittejä on olennaisesti kaksi: voimme kulkea joko luolan 3 tai luolan 5 kautta.

Luku 13

Komponentit ja virittävät puut

Tähän mennessä olemme tarkastelleet verkkoja, joiden rakenne säilyy samana koko algoritmin ajan. Mitä tapahtuu sitten, jos verkkoon tulee *muutoksia*, kuten lisäämme verkkoon uusia kaaria?

Tutustumme tässä luvussa union-find-rakenteeseen, joka on hyödyllinen työkalu verkkojen käsittelyssä. Rakenteen avulla voimme pitää kirjaa verkon yhtenäisistä komponenteista ja päivittää rakennetta tehokkaasti, kun lisäämme verkkoon kaaria. Voimme esimerkiksi tarkkailla, montako yhtenäistä komponenttia verkossa on milläkin hetkellä.

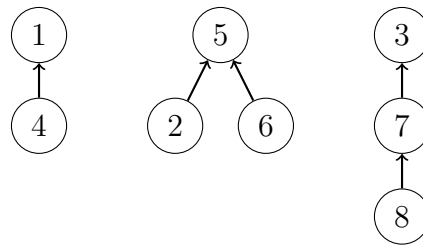
Käsitlemme myös pienimmän virittävän puun ongelmaa, jossa haluamme kytkeä verkon solmut toisiinsa kaaria käyttäen niin, että kaarten yhteispaino on pienin. Voimme ratkaista ongelman tehokkaasti Kruskalin algoritmilla, joka perustuu union-find-rakenteeseen, tai Primin algoritmilla, joka muistuttaa Dijkstran algoritmia.

13.1 Union-find-rakenne

Union-find-rakenne on tietorakenne, joka pitää yllä kokoelmaa alkioiden joukkoja ja tarjoaa seuraavat tehokkaat operaatiot:

- tarkasta, ovatko kaksi alkioita samassa joukossa
- yhdistä kaksi joukkoa samaksi joukoksi

Oletamme, että alkiot ovat $1, 2, \dots, n$, ja jokainen alkio kuuluu tarkalleen yhteen joukkoon. Esimerkiksi kun $n = 8$, joukot voivat olla vaikkapa $A = \{1, 4\}$, $B = \{2, 5, 6\}$ ja $C = \{3, 7, 8\}$. Tällä hetkellä esimerkiksi alkiot 1 ja 2 ovat eri joukoissa. Kun yhdistämme sitten joukot A ja B , niistä syntyy joukko $\{1, 2, 4, 5, 6\}$. Tästä lähtien alkiot 1 ja 2 ovatkin samassa joukossa.



Kuva 13.1: Union-find-rakenne joukoille $\{1, 4\}$, $\{2, 5, 6\}$ ja $\{3, 7, 8\}$.

13.1.1 Rakenteen toteutus

Toteutamme union-find-rakenteen niin, että jokaisessa joukossa yksi alkioista on joukon *edustaja*. Kutakin joukkoa vastaa puu, jonka juurena on joukon edustaja ja muut alkiot viittaavat edustajaan yhden tai useamman kaaren kautta. Kun haluamme tarkastaa, ovatko kaksi alkioita samassa joukossa, selvitämme niiden edustajat ja vertaamme niitä toisiinsa.

Kuvassa 13.1 on esimerkkinä union-find-rakenne, joka vastaa joukkoja $A = \{1, 4\}$, $B = \{2, 5, 6\}$ ja $C = \{3, 7, 8\}$. Tässä tapauksessa joukkojen edustajat ovat 1, 5 ja 3. Esimerkiksi jos haluamme tarkastaa, ovatko alkiot 2 ja 6 samassa joukossa, selvitämme ensin alkioiden edustajat kulkemalla polkuja $2 \rightarrow 5$ ja $6 \rightarrow 5$. Kummankin alkion edustaja on 5, joten toteamme, että alkiot ovat samassa joukossa.

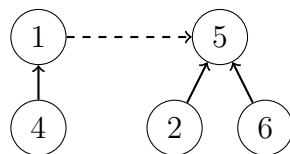
Jotta saamme toteutettua union-find-rakenteen, pidämme yllä jokaiselle alkioille x arvoa `vanhempi[x]`, joka kertoo seuraavan alkion ylempänä puussa. Kuitenkin jos x on joukon edustaja, `vanhempi[x] = x`. Esimerkiksi kuvassa 13.1 `vanhempi[2] = 5` ja `vanhempi[5] = 5`. Tämän ansiosta pystymme selvittämään alkion x edustajan seuraavasti:

```
function edustaja(x)
    while x != vanhempi[x]
        x = vanhempi[x]
    return x
```

Tämän jälkeen voimme tarkastaa seuraavalla operaatiolla, ovatko alkiot a ja b samassa joukossa. Alkiot ovat samassa joukossa täsmälleen silloin, kun niillä on sama edustaja:

```
function sama(a,b)
    return edustaja(a) == edustaja(b)
```

Haluamme toteuttaa vielä operaation, jolla voimme yhdistää kaksi joukkoa toisiinsa. Tämän operaation toteutus ratkaisee, kuinka tehokas raken-



Kuva 13.2: Tehokas yhdistäminen. Alkion 1 joukon koko on 2 ja alkion 5 joukon koko on 3, joten yhdistämme alkion 1 alkioon 5.

teemme on. Alkion edustajan etsiminen vie aikaa $O(k)$, missä k on polun pituus, joten haluamme toteuttaa yhdistämiset niin, että puussa on vain lyhyitä polkuja. Saavutamme tämän tavoitteen yhdistämällä kaksi joukkoa aina asettamalla *pienemmän* joukon edustajan osoittamaan *suuremman* joukon edustajaan. Jos joukot ovat yhtä suuria, voimme toteuttaa yhdistämisen kummin päin vain.

Kuva 13.2 näyttää, mitä tapahtuu, kun yhdistämme joukot $A = \{1, 4\}$ ja $B = \{2, 5, 6\}$. Joukon A edustaja on 1 ja siinä on kaksi alkiota, kun taas joukon B edustaja on 5 ja siinä on kolme alkiota. Koska joukko A on pienempi, asetamme joukon A edustajan osoittamaan joukon B edustajaan. Tämän jälkeen kaikki alkiot kuuluvat samaan joukkoon ja alkio 5 on tästä lähtien koko joukon edustaja.

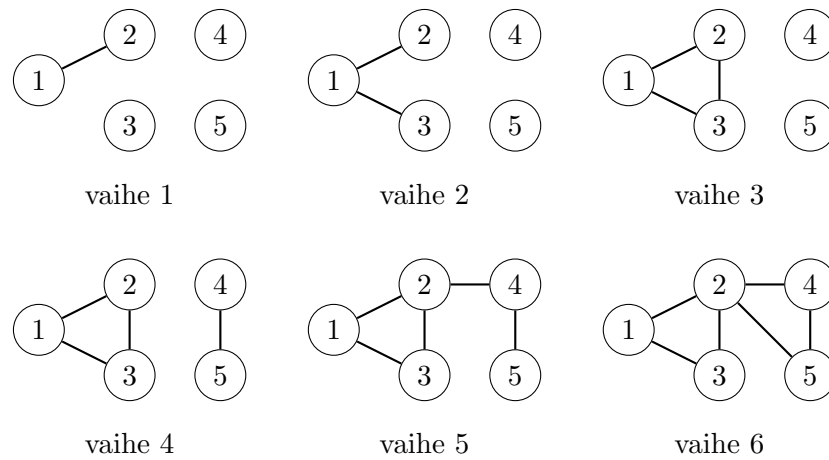
Nyt olemme valmiita toteuttamaan operaation, joka yhdistää toisiinsa joukot, joissa on alkiot a ja b . Oletamme, että alkiot ovat eri joukoissa ennen yhdistämistä. Jotta voimme toteuttaa yhdistämisen tehokkaasti, meidän täytyy myös pitää kirjaa kunkin joukon koosta. Seuraavassa toteutuksessa `koko[x]` kertoo, montako alkiota alkion x edustama joukko sisältää.

```

procedure yhdistä(a,b)
  a = edustaja(a)
  b = edustaja(b)
  if koko[a] < koko[b]
    swap(a,b)
  vanhempi[b] = a
  koko[a] += koko[b]

```

Kun toteutamme yhdistämiset tällä tavalla, jokainen puussa esiintyvä polku sisältää vain $O(\log n)$ alkiota. Tämä johtuu siitä, että aina kun kuljemme polkua askeleen ylöspäin alkion a alkioon b , `koko[b] $\geq 2 \cdot$ koko[a]` eli edustajaa vastaavan joukon koko ainakin *kaksinkertaistuu*. Koska joukossa on enintään n alkiota, kuljemme siis yhteensä enintään $O(\log n)$ askelta. Niinpä kaikki union-find-rakenteen operaatiot toimivat ajassa $O(\log n)$.



Kuva 13.3: Esimerkki kaupunkien yhdistämisestä teillä. Vaiheen 5 jälkeen kaikki kaupungit ovat yhteydessä toisiinsa.

13.1.2 Esimerkki: Kaupungit

Bittimaassa on n kaupunkia, joiden välillä ei ole vielä yhtään tietä. Sitten teitä aletaan rakentaa yksi kerrallaan, yhteensä m tietä. Jokainen tie yhdistää kaksi kaupunkia toisiinsa. Minkä tien rakentamisen jälkeen kaikki kaupungit ovat ensimmäistä kertaa yhteydessä toisiinsa?

Kuva 13.3 näyttää esimerkitapauksen, jossa $n = 5$, $m = 6$ ja tiet rakennetaan järjestyksessä $(1, 2)$, $(1, 3)$, $(2, 3)$, $(4, 5)$, $(2, 4)$ ja $(2, 5)$. Kaikki kaupungit ovat yhteydessä toisiinsa vaiheen 5 jälkeen.

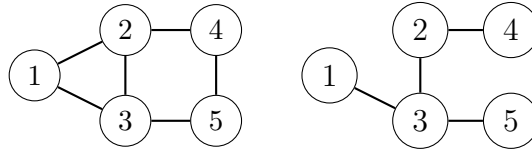
Ratkaisu 1: Union-find-rakenne

Pidämme yllä verkon komponentteja union-find-rakenteen avulla. Aluksi jokainen solmu on omassa komponentissaan eli joukot ovat $\{1\}, \{2\}, \dots, \{n\}$. Sitten jokaisen kaaren kohdalla tarkastamme, ovatko sen päätesolmut eri joukoissa, ja jos ovat, yhdistämme joukot. Kun lopulta kaikki solmut ovat samassa joukossa, verkko on tullut yhtenäiseksi.

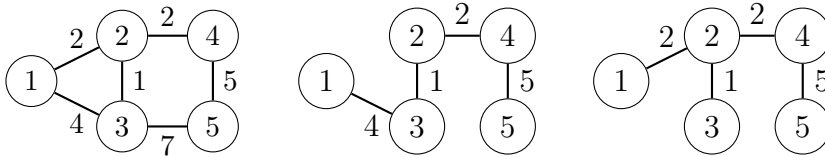
Tuloksena oleva algoritmi vie aikaa $O(n + m \log n)$, koska luomme ensin n komponenttia ajassa $O(n)$ ja käsittelemme tämän jälkeen m kaarta. Jokaisen kaaren kohdalla suoritamme enintään kaksi operaatiota union-find-rakenteessa ajassa $O(\log n)$.

Ratkaisu 2: Binäärihaku

Toinen tapa ratkaista tehtävä on hyödyntää *binäärihakua*. Jos meillä on arvaus, että kaikki kaupungit ovat yhteydessä x lisäyksen jälkeen, voimme tar-



Kuva 13.4: Verkko ja yksi sen virittävistä puista.

Kuva 13.5: Painotettu verkko ja kaksi virittävää puuta, joiden painot ovat $4 + 1 + 2 + 5 = 12$ ja $2 + 1 + 2 + 5 = 10$.

kistaa helposti, pitääkö arvaus paikkansa: lisäämme ensin x ensimmäistä tietä tyhjään verkkoon ja tarkastamme sitten, onko verkko yhtenäinen. Tämä vie aikaa $O(n + m)$ käyttäen syvyyshakua.

Jos verkko on yhtenäinen ensimmäistä kertaa vaiheessa k , selvästikin verkko ei ole yhtenäinen vaiheissa $1, 2, \dots, k - 1$ ja on yhtenäinen vaiheissa $k, k + 1, \dots, m$, koska kaarten lisääminen ei voi poistaa verkon yhtenäisyyttä. Tämän ansiosta voimme etsiä arvon k binäärihaun avulla. Binäärihaku suorittaa $O(\log m)$ askelta ja ratkaisu vie aikaa $O((n + m) \log m)$.

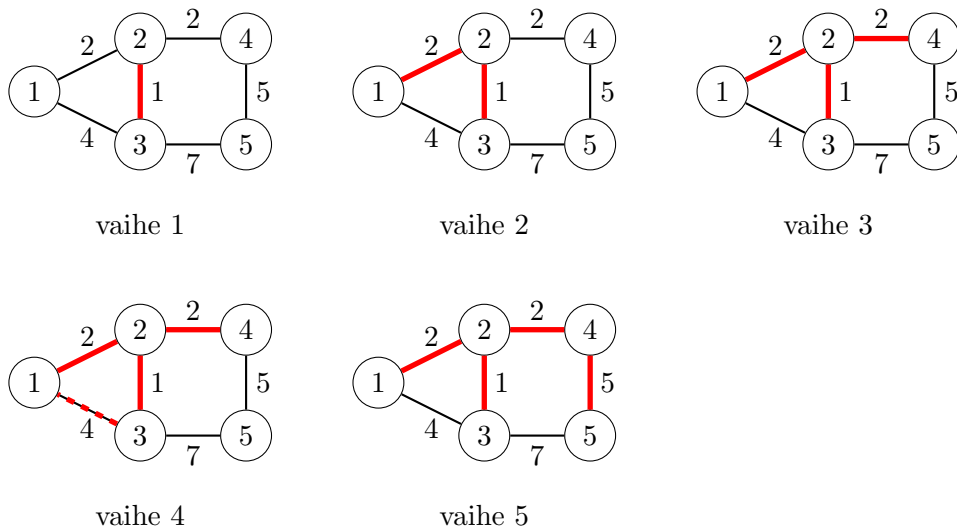
13.2 Pienin virittävä puu

Verkon *virittävä puu* (*spanning tree*) on kokoelma verkon kaaria, jotka kytkevät kaikki verkon solmut toisiinsa. Kuten puut yleensäkin, virittävä puu on yhtenäinen ja syklitön eli jokaisen kahden solmun välillä on yksikäsitteinen polku. Kuvassa 13.4 on esimerkkinä verkko ja yksi sen virittävistä puista.

Jos verkko on painotettu, kiinnostava ongelma on etsiä verkon *pienin virittävä puu* (*minimum spanning tree*). Tämä on virittävä puu, jonka kaarten painojen summa on mahdollisimman pieni. Esimerkiksi kuvassa 13.5 on painotettu verkko ja kaksi sen virittävää puuta, joiden painot ovat 12 ja 10. Näistä jälkimmäinen on verkon pienin virittävä puu.

13.2.1 Kruskalin algoritmi

Kruskalin algoritmi muodostaa verkon pienimmän virittävän puun aloittamalla tyhjästä verkosta, jossa on vain verkon solmut, ja lisäämällä siihen



Kuva 13.6: Esimerkki Kruskalin algoritmin toiminnasta.

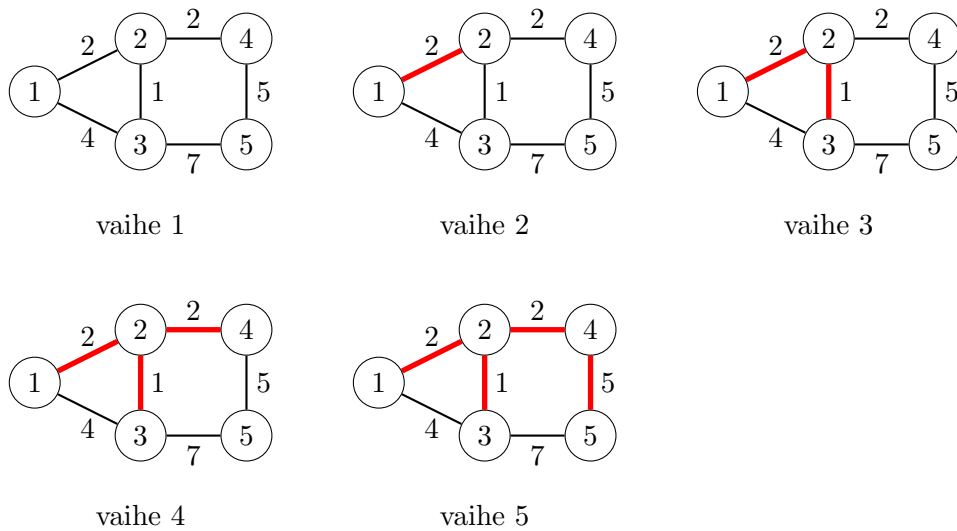
kaaria. Algoritmi käy läpi tarjolla olevat kaaret järjestyksessä niiden painon mukaan kevyimmästä raskaimpaan. Jokaisen kaaren kohdalla algoritmi ottaa kaaren mukaan, jos se yhdistää kaksi eri komponenttia. Kun kaikki komponentit on yhdistetty, pienin virittävä puu on valmis.

Kuva 13.6 näyttää, kuinka Kruskalin algoritmi löytää pienimmän virittävän puun esimerkkiverkossamme. Verkon kaaret järjestyksessä kevyimmästä raskaimpaan ovat:

kaari	paino
(2, 3)	1
(1, 2)	2
(2, 4)	2
(1, 3)	4
(4, 5)	5
(3, 5)	7

Algoritmi käsittelee ensin kaaren (2, 3). Solmut 2 ja 3 ovat eri komponenteissa, joten kaari otetaan mukaan puuhun. Tämän jälkeen algoritmi käsittelee kaaret (1, 2) ja (2, 4), jotka valitaan myös puuhun. Seuraavaksi vuorossa on kaari (1, 3), mutta tämä kaari ei tule puuhun, koska solmut 1 ja 3 ovat jo samassa komponentissa. Lopuksi algoritmi ottaa mukaan kaaren (4, 5), jolloin pienin virittävä puu on valmis.

Voimme toteuttaa Kruskalin algoritmin tehokkaasti käyttäen union-find-rakennetta. Algoritmin alussa järjestämme kaaret painojärjestykseen, missä



Kuva 13.7: Esimerkki Primin algoritmin toiminnasta.

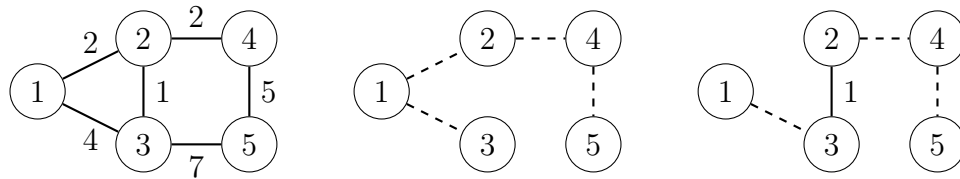
kuluu aikaa $O(m \log m)$. Kun oletamme, että jokaisen solmuparin välillä on enintään yksi kaari, tämä sievenee muotoon $O(m \log n)$. Tämän jälkeen luomme kullekin solmulle komponentin, käymme kaaret läpi ja jokaisen kaaren kohdalla otamme kaaren mukaan, jos se yhdistää kaksi eri komponenttia. Tässä kuluu aikaa $O(n + m \log n)$, kun käytämme union-find-rakennetta. Algoritmi vie siis yhteensä aikaa $O(n + m \log n)$.

13.2.2 Primin algoritmi

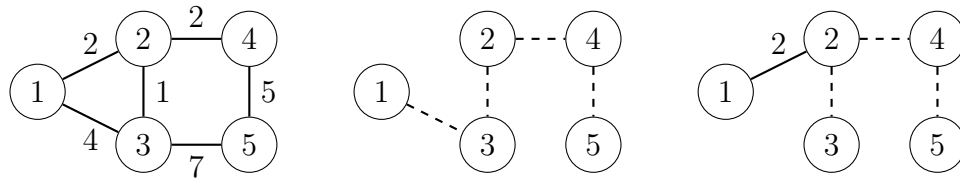
Primin algoritmi tarjoaa toisen lähestymistavan pienimmän virittävän puun muodostamiseen. Algoritmi aloittaa puun muodostamisen tilanteesta, jossa puussa on vain yksi solmu. Tämän jälkeen se etsii joka vaiheessa kevyimmän kaaren, jonka toinen päätesolmu kuuluu puuhun ja toinen päätesolmu on vielä puun ulkopuolella, ja lisää puuhun tämän kaaren. Kun kaikki solmut on lisätty puuhun, pienin virittävä puu on valmis.

Kuva 13.7 näyttää esimerkin Primin algoritmin toiminnasta. Voimme aloittaa puun rakentamisen mistä tahansa solmusta; tässä esimerkissä aloitamme solmusta 1. Solmuun 1 on yhteydessä kaksi kaarta (1, 2) ja (1, 3), joista valitsemme kaaren (1, 2), koska se on kevyempi. Seuraavaksi tarjolla ovat kaaret (1, 3), (2, 3) ja (2, 4), joista valitsemme kaaren (2, 3). Tämän jälkeen lisäämme puuhun vastaavalla tavalla kaaret (2, 4), (4, 5), minkä jälkeen pienin virittävä puu on valmis.

Primin algoritmi muistuttaa paljon Dijkstran algoritmia. Erona on, että Dijkstran algoritmista valitsemme seuraavaksi solmun, jonka etäisyys al-



Kuva 13.8: Kruskalin algoritmi: Pienin virittävä puu sisältää varmasti kaaren $(2, 3)$, koska muuten saisimme paremman ratkaisun sen avulla.



Kuva 13.9: Primin algoritmi: Pienin virittävä puu sisältää varmasti kaaren $(1, 2)$, koska muuten saisimme paremman ratkaisun sen avulla.

kusolmuun on pienin, mutta Primin algoritmista valitsemme solmun, jonka etäisyys *johonkin solmuun* puussa on pienin. Voimme myös toteuttaa Primin algoritmin tehokkaasti samaan tapaan kuin Dijkstran algoritmin keon avulla, jolloin algoritmi vie aikaa $O(n + m \log n)$. Aikavaativuus on siis sama kuin Kruskalin algoritmista, ja on käytännössä makuasia, kumman algoritmin valitsemme.

13.2.3 Miksi algoritmit toimivat?

Kruskalin ja Primin algoritmit ovat ahneita algoritmeja: ne lisäävät joka askeleella kevyimmän mahdollisen kaaren puuhun. Miksi algoritmit tuottavat pienimmän virittävän puun joka tilanteessa?

Voimme ajatella asiaa näin: Jos meillä on kaksi solmua a ja b , jotka ovat eri komponenteissa, meidän on yhdistettävä ne jotenkin samaan komponenttiin algoritmin aikana. Jos kevyin saatavilla oleva kaari on solmujen a ja b välillä, meidän kannattaa valita se, koska muuten joutuisimme yhdistämään komponentit myöhemmin käyttäen raskaampaa kaarta.

Tarkastellaan ensin Kruskalin algoritmia. Mitä tapahtuu, jos emme valitse kevyintä kaarta algoritmin alussa? Kuvassa 13.8 näkyy kuvitteellinen tilanne, jossa katkoviivoilla esitetty pienin virittävä puu ei sisällä kevyintä kaarta $(2, 3)$, jonka paino on 1. Ei ole kuitenkaan mahdollista, että tämä olisi todellisuudessa pienin virittävä puu, koska voisimme vaihtaa jonkin puun kaaren kaareen $(2, 3)$, jolloin puun paino pienenee. Voimme siis huoletta va-

lita kevyimmän kaaren puuhun Kruskalin algoritmin alussa. Samasta syystä voimme tämän jälkeen valita seuraavaksi kevyimmän kaaren, jne.

Primin algoritmissa voimme käyttää melko samanlaista päättelyä. Oletetaan, että algoritmi lähtee liikkeelle solmusta 1, ja tarkastellaan ensimmäisen kaaren valintaa. Kuvassa 13.9 näkyy kuvitteellinen pienin virittävä puu, jossa emme ole valinneet alussa kevyintä kaarta $(1, 2)$. Saamme kuitenkin aikaan paremman ratkaisun, kun tutkimme, minkä kaaren kautta solmu 1 on yhteydessä solmuun 2, ja korvaamme tämän kaaren kaarella $(1, 2)$. Niinpä Primin algoritmin alussa on optimaalista valita puuhun kevyin kaari. Voimme soveltaa vastaavaa päättelyä algoritmin joka vaiheessa.

Huomaa, että Kruskalin ja Primin algoritmit toimivat myös silloin, kun verkossa on negatiivisia kaaria, koska ainoastaan kaarten painojärjestys merkitsee. Tämän ansiosta voimme etsiä algoritmien avulla myös verkon *suurimman virittävän puun* (*maximum spanning tree*) eli virittävän puun, jossa kaarten painojen summa on suurin. Tämä onnistuu muuttamalla ensin jokaisen verkossa olevan kaaren paino käänteiseksi ja etsimällä sitten pienimmän virittävän puun. Toinen tapa on vain toteuttaa Kruskalin tai Primin algoritmi niin, että joka vaiheessa valitaan painavin mahdollinen kaari.

Luku 14

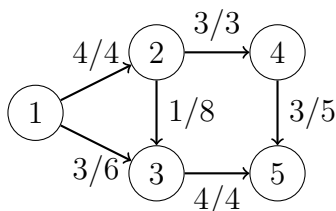
Maksimivirtaus

Tässä luvussa tarkastelemme ongelmaa, jossa haluamme välittää mahdollisimman paljon *virtausta* (*flow*) verkon solmusta toiseen, kun jokaisella kaarella on tietty kapasiteetti, jota emme saa ylittää. Voimme esimerkiksi haluta siirtää tietokoneverkossa tietoa mahdollisimman tehokkaasti koneesta toiseen, kun tiedämme verkon rakenteen ja yhteyksien nopeudet.

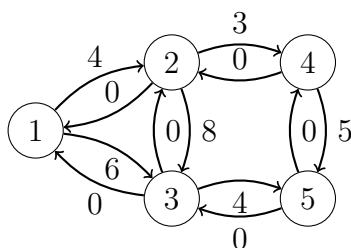
Tutustumme aluksi Ford–Fulkersonin algoritmiin, jonka avulla voimme sekä selvittää maksimivirtauksen että ymmärtää paremmin, mistä ongelmasa on kysymys. Tämän jälkeen tarkastelemme joitakin verkko-ongelmia, jotka pystymme ratkaisemaan *palauttamalla* ne maksimivirtaukseen.

14.1 Maksimivirtauksen laskeminen

Käsitlemme suunnattua verkkoa, jossa on kaksi erityistä solmua: *lähtösolmu* (*source*) ja *kohdesolmu* (*sink*). Haluamme muodostaa verkkoon mahdollisimman suuren virtauksen eli *maksimivirtauksen* (*maximum flow*) lähtösolmusta kohdesolmuun niin, että jokaiseen välisolmuun tuleva virtaus on yhtä suuri kuin solmusta lähtevä virtaus. Virtausta rajoittaa, että kullakin verkon kaarella on *kapasiteetti*, jota virtauksen määrä kaarta pitkin ei saa ylittää.



Kuva 14.1: Maksimivirtaus solmusta 1 solmuun 5 on 7.



Kuva 14.2: Verkon esitysmuoto Ford–Fulkersonin algoritmossa.

Kuvassa 14.1 näkyy esimerkkinä verkon maksimivirtaus, kun lähtösolmu on 1 ja kohdesolmu 5. Tässä tapauksessa maksimivirtauksen suuruus on 7. Jokaisessa kaaressa merkintä v/k tarkoittaa, että kaaren kautta kulkee virtausta v ja kaaren kapasiteetti on k . Solmusta 1 lähtevä virtauksen määrä on $4 + 3 = 7$, solmuun 5 saapuva virtauksen määrä on $3 + 4 = 7$ ja kaikissa muissa solmuissa saapuva virtaus on yhtä suuri kuin lähtevä virtaus.

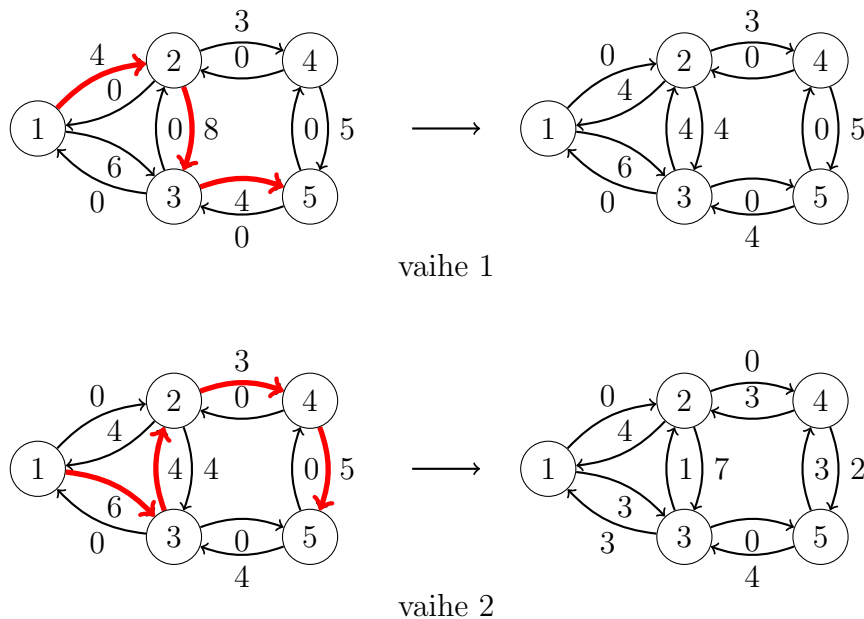
14.1.1 Ford–Fulkersonin algoritmi

Ford–Fulkersonin algoritmi on tavallisin menetelmä verkon maksimivirtauksen etsimiseen, ja tutustumme seuraavaksi tämän algoritmin toimintaan. Algoritmi muodostaa lähtösolmusta kohdesolmuun polkuja, jotka kasvattavat virtausta pikkuhiljaa. Kun mitään polkua ei voi enää muodostaa, algoritmi on saanut valmiiksi maksimivirtauksen.

Jotta voimme käyttää algoritmia, esitämme verkon erityisessä muodossa, jossa jokaista alkuperäisen verkon kaarta vastaa *kaksi* kaarta: alkuperäinen kaari, jonka painona on kaaren kapasiteetti, sekä sille käänteinen kaari, jonka painona on 0. Käänteisten kaarten avulla pystymme tarvittaessa *peruuttamaan* virtausta algoritmin aikana. Kuva 14.2 näyttää, kuinka esitämme esimerkiverkkomme algoritmossa.

Algoritmin jokaisessa vaiheessa muodostamme polun lähtösolmusta kohdesolmuun. Polku voi olla mikä tahansa, kunhan jokaisen kaaren paino polulla on positiivinen. Polun muodostamisen jälkeen virtaus lähtösolmusta kohdesolmuun kasvaa p :llä, missä p on pienin kaaren paino polulla. Lisäksi jokaisen polulla olevan kaaren paino vähenee p :llä ja jokaisen niille käänteisen kaaren paino kasvaa p :llä. Etsimme tällä tavalla uusia polkuja, kunnes mitään sallittua polkua ei voi enää muodostaa.

Kuva 14.3 näyttää, kuinka Ford–Fulkersonin algoritmi muodostaa maksimivirtauksen esimerkiverkossamme. Algoritmi muodostaa ensin polun $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$, jossa pienin paino on 4. Tämän seurauksena virtaus kasvaa 4:llä, polulla olevien kaarten paino vähenee 4:llä ja käänteisten kaarten paino kas-



Kuva 14.3: Esimerkki Ford–Fulkersonin algoritmin toiminnasta.

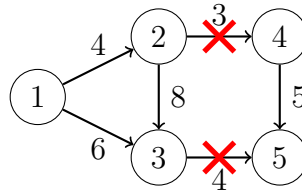
vaa 4:llä. Tämän jälkeen algoritmi muodostaa polun $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5$, joka kasvattaa virtausta 3:lla. Huomaa, että tämä polku peruuttaa kaarta $2 \rightarrow 3$ menevää virtausta, koska se kulkee käänteisen kaaren $3 \rightarrow 2$ kautta. Tämän jälkeen algoritmi ei enää pysty muodostamaan mitään polkua solmusta 1 solmuun 5, joten maksimivirtaus on $4 + 3 = 7$.

Kun olemme saaneet maksimivirtauksen muodostettua, voimme selvittää jokaisessa alkuperäisessä kaaressa kulkevan virtauksen tutkimalla, miten kaaren paino on muuttunut algoritmin aikana. Kaarta pitkin kulkeva virtaus on yhtä suuri kuin kaaren painon vähennys algoritmin aikana. Esimerkiksi kuvassa 14.3 kaaren $4 \rightarrow 5$ paino on alussa 5 ja algoritmin suorituksen jälkeen 2, joten kaarta pitkin kulkevan virtauksen määrä on $5 - 2 = 3$.

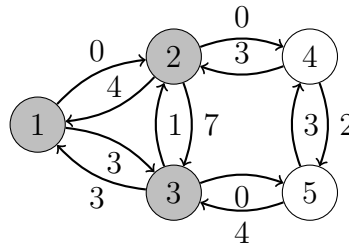
14.1.2 Yhteys minimileikkaukseen

Ford–Fulkersonin algoritmin toimintaidea on sinänsä järkevä, koska polut lähtösolmusta kohdesolmuun kasvattavat virtausta, mutta ei ole silti todellakaan päältä päin selvää, miksi algoritmi löytää varmasti *suurimman* mahdollisen virtauksen. Jotta voimme ymmärtää paremmin algoritmin toimintaa, tarkastelemme seuraavaksi toista verkko-ongelmaa, joka antaa meille uuden näkökulman maksimivirtaukseen.

Lähtökohtanamme on edelleen suunnattu verkko, jossa on lähtösolmu ja



Kuva 14.4: Minimileikkaus, jossa poistamme kaaret $2 \rightarrow 4$ ja $3 \rightarrow 5$.



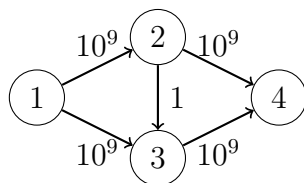
Kuva 14.5: Solmut 1, 2 ja 3 ovat saavutettavissa lähtösolmusta.

kohdesolmu. Sanomme, että joukko kaaria muodostaa *leikkauksen* (*cut*), jos niiden poistaminen verkosta estää kulkemisen lähtösolmusta kohdesolmuun. *Minimileikkaus* (*minimum cut*) on puolestaan leikkaus, jossa kaarten yhteispaino on mahdollisimman pieni. Kuvassa 14.4 on minimileikkaus, jossa poistamme kaaret $2 \rightarrow 4$ ja $3 \rightarrow 5$ ja jonka paino on $3 + 4 = 7$.

Osoittautuu, että verkon maksimivirtaus on aina yhtä suuri kuin minimileikkaus, ja tämä yhteys auttaa perustelemaan, miksi Ford–Fulkersonin algoritmi toimii. Ensinnäkin voimme havaita, että *mikä tahansa* verkon leikkaus on yhtä suuri tai suurempi kuin maksimivirtaus. Tämä johtuu siitä, että virtauksen täytyy ylittää jokin leikkaukseen kuuluva kaari, jotta se pääsee lähtösolmusta kohdesolmuun. Esimerkiksi kuvassa 14.4 virtaus voi päästä solmusta 1 solmuun 5 joko kulkemalla kaarta $2 \rightarrow 4$ tai kaarta $3 \rightarrow 5$. Niinpä virtaus ei voi olla suurempi kuin näiden kaarten painojen summa.

Toisaalta Ford–Fulkersonin algoritmi muodostaa sivutuotteenaan myös verkon leikkauksen, joka on yhtä suuri kuin maksimivirtaus. Löydämme leikkauksen etsimällä ensin kaikki solmut, joihin pääsemme lähtösolmusta positiivisia kaaria pitkin algoritmin lopputilanteessa. Esimerkkiverkossamme nämä solmut ovat 1, 2 ja 3 kuvan 14.5 mukaisesti. Kun valitsemme sitten alkuperäisen verkon kaaret, jotka johtavat näiden solmujen ulkopuolelle ja joiden kapasiteetti on käytetty kokonaan, saamme aikaan verkon leikkauksen. Esimerkissämme nämä kaaret ovat $2 \rightarrow 4$ ja $3 \rightarrow 5$.

Koska olemme löytäneet virtauksen, joka on yhtä suuri kuin leikkaus, ja toisaalta virtaus ei voi olla mitään leikkausta suurempi, olemme siis löytäneet



Kuva 14.6: Verkko, joka voi aiheuttaa ongelmia algoritmille.

maksimivirtauksen ja minimileikkauksen, joten Ford–Fulkersonin algoritmi toimii oikein.

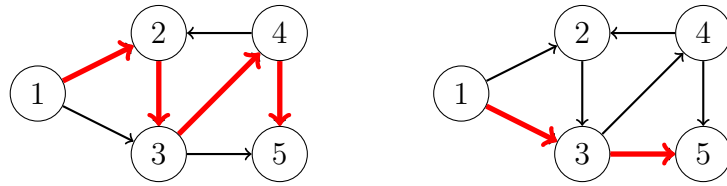
14.1.3 Polkujen valitseminen

Voimme muodostaa polkuja Ford–Fulkersonin algoritmin aikana miten tahansa, mutta polkujen valintatapa vaikuttaa algoritmin tehokkuuteen. Riippumatta polkujen valintatavasta on selvää, että jokainen polku kasvattaa virtausta ainakin *yhdeällä* yksiköllä. Niinpä tiedämme, että joudumme etsimään enintään f polkua, kun verkon maksimivirtaus on f . Jos muodostamme polut syvyyshaulla, jokaisen polun muodostaminen vie aikaa $O(m)$, joten saamme algoritmin ajankäytölle ylärajan $O(fm)$.

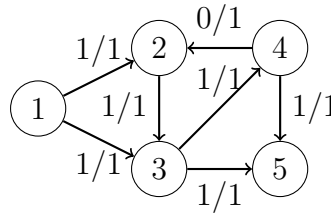
Voiko todella käydä niin, että jokainen polku parantaa virtausta vain yhdellä? Tämä on mahdollista, ja kuva 14.6 tarjoaa esimerkin asiasta. Jos muodostamme polut syvyyshaulla ja valitsemme haussa aina solmun, jonka tunnus on pienin, muodostamme vuorotellen polkuja $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ja $1 \rightarrow 3 \rightarrow 2 \rightarrow 4$ niin, että lisäämme ja peruutamme edestakaisin kaarta $2 \rightarrow 3$ kulkevaa virtausta. Tämän vuoksi joudumme muodostamaan $2 \cdot 10^9$ polkua, ennen kuin olemme saaneet selville verkon maksimivirtauksen.

Voimme kuitenkin estää tämän ilmiön määrittelemällä tarkemmin, miten valitsemme polkuja algoritmin aikana. *Edmonds–Karpin algoritmi* on Ford–Fulkersonin algoritmin versio, jossa muodostamme polut *leveyshaulla*. Tämä tarkoittaa, että valitsemme aina polun, jossa on mahdollisimman vähän kaaria. Leveyshakua käyttäen muodostamme kuvan 14.6 verkossa vain kaksi polkua $1 \rightarrow 2 \rightarrow 4$ ja $1 \rightarrow 3 \rightarrow 4$, jotka antavat suoraan maksimivirtauksen.

Kun käytämme Edmonds–Karpin algoritmia, meidän täytyy muodostaa aina vain $O(nm)$ polkua, joten algoritmi vie aikaa $O(nm^2)$. Saamme tämän rajan tarkastelemalla kullakin polulla jotakin kaarta, jonka kapasiteetti on käytetty kokonaan polulla. Jotta voimme käyttää kyseistä kaarta uudestaan jollain tulevalla polulla, meidän täytyy sitä ennen peruuttaa virtausta kulke-malla kaarta käänteiseen suuntaan, mikä kasvattaa etäisyyttä lähtösolmusta kaareen. Koska valitsemme polut leveyshaun avulla, kukin m kaaresta voi



Kuva 14.7: Kaksi erillistä polkua solmusta 1 solmuun 5.



Kuva 14.8: Erilliset polut tulkittuna maksimivirtauksena.

tämän vuoksi olla vain $O(n)$ kertaa kaarena, jonka kapasiteetti käytetään kokonaan, ja saamme polkujen määrälle ylärajan $O(nm)$.

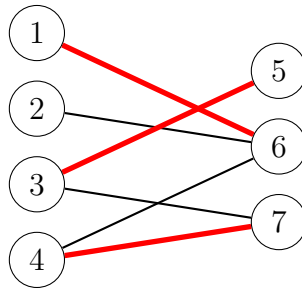
14.2 Maksimivirtauksen sovelluksia

Maksimivirtauksen etsiminen on tärkeä ongelma, koska pystymme *palauttamaan* monia verkko-ongelmia maksimivirtaukseen. Tämä tarkoittaa, että muutamme toisen ongelman jotenkin sellaiseen muotoon, että se vastaa maksimivirtausta. Tutustumme seuraavaksi joihinkin tällaisiin ongelmiin.

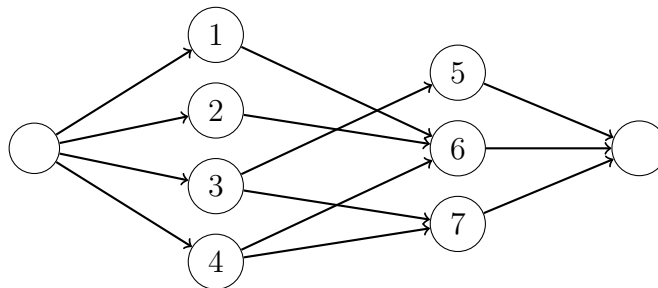
14.2.1 Erilliset polut

Ensimmäinen tehtävämme on muodostaa mahdollisimman monta *erillistä* polkua verkon lähtösolmusta kohdesolmuun. Erillisyys tarkoittaa, että jokainen verkon kaari saa esiintyä enintään yhdellä polulla. Saamme kuitenkin halutessamme kulkea saman solmun kautta useita kertoja. Esimerkiksi kuvassa 14.7 voimme muodostaa kaksi erillistä polkua solmusta 1 solmuun 5, mutta ei ole mahdollista muodostaa kolmea erillistä polkua.

Voimme ratkaista ongelman tulkitsemalla erilliset polut maksimivirtauksena. Ideana on etsiä maksimivirtaus lähtösolmusta kohdesolmuun olettaen, että jokaisen kaaren kapasiteetti on 1. Tämä maksimivirtaus on yhtä suuri kuin suurin erillisten polkujen määrä. Kuva 14.8 näyttää maksimivirtauksen esimerkkiverkossamme.



Kuva 14.9: Kaksijakoisen verkon maksimiparitus.



Kuva 14.10: Maksimiparitus tulkittuna maksimivirtauksena.

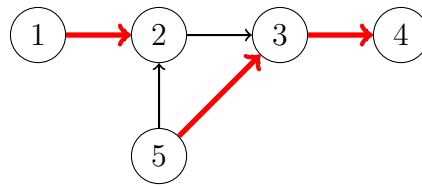
Miksi maksimivirtaus ja erillisten polkujen määrä ovat yhtä suuret? Ensinnäkin erilliset polut muodostavat yhdessä virtauksen, joten maksimivirtaus ei voi olla pienempi kuin erillisten polkujen määrä. Toisaalta jos verkossa on virtaus, jonka suuruus on k , voimme muodostaa k erillistä polkua valitsemalla kaaria ahneesti lähtösolmusta alkaen, joten maksimivirtaus ei voi olla suurempi kuin erillisten polkujen määrä. Ainoa mahdollisuus on, että maksimivirtaus ja erillisten polkujen määrä ovat yhtä suuret.

14.2.2 Maksimiparitus

Verkon *paritus* (*matching*) on joukko kaaria, joille pätee, että jokainen solmu on enintään yhden kaaren päätepisteenä. *Maksimiparitus* (*maximum matching*) on puolestaan paritus, jossa on mahdollisimman paljon kaaria. Keskitymme tapaukseen, jossa verkko on *kaksijakoinen* (*bipartite*) eli voimme jakaa verkon solmut vasempaan ja oikeaan ryhmään niin, että jokainen kaari kulkee ryhmien välillä.

Kuvassa 14.9 on esimerkkinä kaksijakoinen verkko, jonka maksimiparitus on 3. Tässä vasen ryhmä on $\{1, 2, 3, 4\}$, oikea ryhmä on $\{5, 6, 7\}$ ja maksimiparitus muodostuu kaarista $(1, 6)$, $(3, 5)$ ja $(4, 7)$.

Voimme tulkita maksimiparituksen maksimivirtauksena lisäämällä verk-



Kuva 14.11: Polkupeite, joka muodostuu poluista $1 \rightarrow 2$ ja $5 \rightarrow 3 \rightarrow 4$.

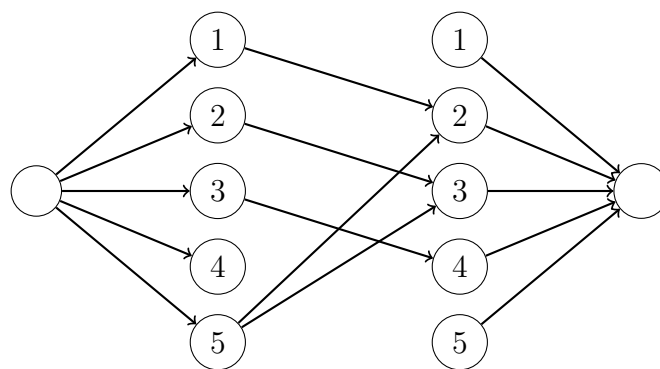
koon kaksi uutta solmua: lähtösolmun ja kohdesolmun. Lähtösolmusta pääsee kaarella jokaiseen vasemman ryhmän solmuun, ja jokaisesta oikean ryhmän solmusta pääsee kaarella kohdesolmuun. Lisäksi suuntaamme alkuperäiset kaaret niin, että ne kulkevat vasemmasta ryhmästä oikeaan ryhmään. Kuva 14.10 näyttää tuloksena olevan verkon esimerkissämme. Maksimivirtaus tässä verkossa vastaa alkuperäisen verkon maksimiparitusta.

14.2.3 Pienin polkupeite

Polkupeite (*path cover*) on joukko verkon polkuja, jotka kattavat yhdessä kaikki verkon solmut. Oletamme, että verkko on suunnattu ja sykliton, ja haluamme muodostaa mahdollisimman pienen polkupeitteen niin, että jokainen solmu esiintyy tarkalleen yhdessä polussa. Kuvassa 14.11 on esimerkkinä verkko ja sen pienin polkupeite, joka muodostuu kahdesta polusta.

Voimme ratkaista pienimmän polkupeitteen etsimisen ongelman maksimivirtauksen avulla muodostamalla verkon, jossa jokaista alkuperäistä solmua vastaa kaksi solmua: vasen ja oikea solmu. Vasemmasta solmusta on kaari oikeaan solmuun, jos alkuperäisessä verkossa on vastaava kaari. Lisäämme vielä verkkoon lähtösolmun ja kohdesolmun niin, että lähtösolmusta pääsee kaikkiin vasempiin solmuihin ja kaikista oikeista solmuista pääsee kohdesolmuihin. Tämän verkon maksimivirtaus antaa meille alkuperäisen verkon pienimmän solmupeitteen.

Kuva 14.12 näyttää tuloksena olevan verkon esimerkissämme. Ideana on, että maksimivirtaus etsii, mitkä kaaret kuuluvat polkuihin: jos kaari solmusta a solmuun b kuuluu virtaukseen, niin vastaavasti polkupeitteessä on polku, jossa on kaari $a \rightarrow b$. Koska virtauksessa voi olla valittuna vain yksi kaari, joka alkaa tietyistä solmista tai päättyy tiettyyn solmuun, tuloksena on varmasti joukko polkuja. Toisaalta mitä enemmän kaaria saamme laitettua polkuihin, sitä pienempi on polkujen määrä.



Kuva 14.12: Polkupeitteen etsiminen maksimivirtauksen avulla.

Luku 15

NP-ongelmat

Olemme tässä kirjassa tutustuneet moniin algoritmeihin, jotka toimivat tehokkaasti. Kuitenkin on myös suuri määrä ongelmia, joiden ratkaisemiseen ei tällä hetkellä tunneta mitään tehokasta algoritmia. Jos vastaamme tulee tällainen ongelma, hyvät neuvot ovat kalliit.

Vaikeiden ongelmien yhteydessä esiintyy usein kirjainyhdistelmä NP. Eri-tyisen tunnettu on kysymys P vs. NP, jonka ratkaisijalle on luvattu miljoonan dollarin potti. Hankalalta tuntuvasta ongelmasta saatetaan arvella, että se on NP-täydellinen tai NP-vaikea. Nyt on aika selvittää, mitä nämä käsitteet oikeastaan tarkoittavat.

15.1 Vaativuusluokat

Laskennallisia ongelmia luokitellaan *vaativuusluokkiin* (*complexity class*), joista jokaisessa on joukko ongelmia, joiden ratkaisemisen vaikeudessa on jotain yhteistä. Tunnetuimmat vaativuusluokat ovat P ja NP.

Keskitymme tässä luvussa *päätösongelmiin* (*decision problem*), joissa algoritmin tulee antaa aina vastaus ”kyllä” tai ”ei”. Esimerkiksi ongelma ”onko verkossa polkua solmusta a solmuun b ?” on päätösongelma. Tulemme huomaamaan, että voimme muotoilla monenlaisia ongelmia päätösongelmina eikä tämä rajoita juurikaan, mitä ongelmia voimme tarkastella.

15.1.1 Luokka P

Luokka P sisältää päätösongelmat, joiden ratkaisemiseen on olemassa *polynominen* algoritmi eli algoritmi, jonka aikavaativuus on enintään $O(n^k)$, missä k on vakio. Lähes kaikki tässä kirjassa esitetyt algoritmit ovat olleet polynomisia. Tuttuja polynomisia aikavaativuuksia ovat esimerkiksi $O(1)$, $O(\log n)$

$O(n)$, $O(n \log n)$, $O(n^2)$ ja $O(n^3)$.

Esimerkiksi ongelma ”onko verkossa polkua solmusta a solmuun b ?” kuuluu luokkaan P, koska voimme ratkaista sen monellakin tavalla polynomisessa ajassa. Voimme vaikkapa aloittaa syvyysshaun solmusta a ja tarkastaa, pääsemmekö solmuun b . Tuloksena on algoritmi, joka toimii lineaarisessa ajassa, joten ongelma kuuluu luokkaan P.

Luokan P tarkoituksena on kuvata ongelmia, jotka voimme ratkaista jossain mielessä *tehokkaasti*. Tässä tehokkuuden määritelmä on varsin karkea: pidämme algoritmia tehokkaana, jos sillä on mikä tahansa polynominen aikavaativuus. Onko $O(n^{100})$ -aikainen algoritmi siis tehokas? Ei, mutta käytännössä vakio k on yleensä pieni ja polynominen aikavaativuus on osoittautunut toimivaksi tehokkuuden mittariksi.

15.1.2 Luokka NP

Luokka NP sisältää päätösongelmat, joissa jokaisessa ”kyllä”-tapauksessa on olemassa *todiste*, jonka avulla voimme *tarkastaa* polynomisessa ajassa, että vastaus todellakin on ”kyllä”. Todiste on merkkijono, jonka koko on polynominen suhteessa syötteeseen, ja se antaa meille lisätietoa siitä, minkä takia ”kyllä”-vastaus pitää paikkansa syötteelle.

Esimerkki luokkaan NP kuuluvasta ongelmasta on ”onko verkossa polkua solmusta a solmuun b , joka kulkee tasan kerran jokaisen verkon solmun kautta?”. Tämän ongelman ratkaisemiseen ei tunneta polynomista algoritmia, mutta jokaisessa ”kyllä”-tapauksessa on olemassa todiste: halutunlainen polku solmusta a solmuun b . Voimme tarkastaa helposti polynomisessa ajassa, että todisteen kuvaamalla polulla on vaaditut ominaisuudet.

Jos vastaus syötteeseen on ”ei”, tähän ei tarvitse liittyä mitään todistetta. Usein olisikin hankalaa antaa todiste siitä, että jotain asiaa *ei* ole olemassa. Esimerkiksi jos etsimme verkosta tietynlaista polkua, on helppoa todistaa polun olemassaolo, koska voimme vain näyttää kyseisen polun, mutta ei ole vastaavaa keinoa todistaa, että polkua ei ole olemassa.

Huomaa, että kaikki luokan P ongelmat kuuluvat myös luokkaan NP. Tämä johtuu siitä, että luokan P ongelmissa voimme tarkastaa ”kyllä”-vastauksen *tyhjän* todisteen avulla: voimme saman tien ratkaista koko ongelman alusta alkaen polynomisessa ajassa.

15.1.3 P vs. NP

Äkkiseltään voisi kuvitella, että luokassa NP täytyy olla enemmän ongelmia kuin luokassa P. Luokassa NP meidän riittää vain tarkastaa ”kyllä”-vastauksen todiste, mikä tuntuu helpommalta kuin muodostaa ongelman rat-



Kuva 15.1: P vs. NP: Voisiko olla yhtä helppoa muodostaa ratkaisu tyhjästä kuin tarkastaa, onko annettu ratkaisu oikein?

kaisu tyhjästä (kuva 15.1). Monet uskovatkin, että luokka NP on suurempi kuin luokka P, mutta kukaan ei ole onnistunut todistamaan asiaa.

Tietojenkäsittelytieteen merkittävä avoin kysymys on, päteekö $P = NP$ vai $P \neq NP$. Monet tutkijat ovat tarttuneet haasteeseen 70-luvulta lähtien, mutta tähän mennessä kaikki ovat epäonnistuneet. Ongelman ratkaisija saisi maineen ja kunnian lisäksi myös tuntuvan rahallisen korvauksen, koska Clay-instituutti on luvannut miljoonan dollarin palkinnon sille, joka todistaa, että $P = NP$ tai $P \neq NP$. Voi olla kuitenkin, että tämä on yksi *vaikeimmista* tavoista ansaita miljoona dollaria.

Jos pätee $P \neq NP$, kuten uskotaan, vaikeutena on keksiä keino todistaa, että jotakin luokan NP ongelmaa on mahdotonta ratkaista polynomisessa ajassa. Tämän todistaminen on vaikeaa, koska meidän pitää näyttää, että tehokasta algoritmia ei ole olemassa, vaikka laatisimme algoritmin miten tahansa. Vaikka moni on koettanut tuloksetta ratkoa tunnettuja NP-ongelmia, kysymys saattaa silti olla siitä, että tehokas algoritmi olisi olemassa mutta kukaan ei vain ole vielä löytänyt sitä.

15.1.4 Muita luokkia

P ja NP ovat tunnetuimmat vaativuusluokat, mutta niiden lisäksi on suuri määrä muitakin luokkia. Yksi tällainen luokka on PSPACE, joka sisältää ongelmat, joiden ratkaisuun riittää polynominen määrä *muistia*. PSPACE sisältää kaikki luokan NP ongelmat, mutta siinä on myös ongelmia, joiden ei tiedetä kuuluvan luokkaan NP.

Luokat BPP ja ZPP puolestaan liittyvät satunnaisuuteen. Luokka BPP sisältää ongelmat, joiden ratkaisuun on polynomiaikainen algoritmi, joka antaa oikean vastauksen ainakin todennäköisyydellä $2/3$. Luokka ZPP taas sisältää ongelmat, joiden ratkaisuun on algoritmi, jonka suoritusajan *odotusarvo* on polynominen. Sekä BPP että ZPP sisältävät luokan P, jonka ongelmat voi ratkaista polynomisessa ajassa ilman satunnaisuuttakin.

Vaativuusluokkien suhteet toisiinsa tunnetaan yleensä ottaen huonosti. Vaikka kysymys P vs. NP on saanut eniten huomiota, myös vastaavat kysymykset esimerkiksi luokkien PSPACE, BPP ja ZPP kohdalla ovat avoimia

ongelmia. Kukaan ei tiedä, onko näissä luokissa loppujen lopuksi mitään ongelmaa, joka ei kuuluisi myös luokkaan P.

15.2 NP-täydellisyys

Sanomme, että ongelma on *NP-täydellinen* (*NP-complete*), jos se kuuluu luokkaan NP ja mikä tahansa luokan NP ongelma voidaan *palauttaa* siihen polynomisessa ajassa. NP-täydelliset ongelmat ovat luokan NP vaikeimpia ongelmia: jos voisimme ratkaista jonkin NP-täydellisen ongelman tehokkaasti, voisimme ratkaista minkä tahansa luokan NP ongelman tehokkaasti.

Kiinnostava ilmiö on, että lähes kaikki tunnetut luokan NP ongelmat joko kuuluvat myös luokkaan P tai ovat NP-täydellisiä. Nykyään tunnetaankin tuhansia erilaisia NP-täydellisiä ongelmia. Jos keksisimme mihin tahansa niistä polynomisessa ajassa toimivan ratkaisun, olisimme samalla onnistuneet todistamaan, että $P = NP$.

15.2.1 SAT-ongelma

Ensimmäinen löydetty NP-täydellinen ongelma oli SAT-ongelma, jossa annettuna on konjunktiiivisessa normaalimuodossa oleva looginen kaava ja haluamme selvittää, voimmeko valita muuttujien arvot niin, että kaava on tosi. Konjunktiiivinen normaalimuoto tarkoittaa, että kaava koostuu lausekkeista, jotka on yhdistetty ja-operaatiolla (\wedge), ja jokainen lauseke muodostuu muuttujista ja niiden negaatioista, jotka on yhdistetty tai-operaatiolla (\vee).

Esimerkiksi kaava

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

on mahdollista saada todeksi, koska voimme esimerkiksi asettaa muuttujat x_1 ja x_2 epätosi ja muuttujan x_3 todeksi. Vastaavasti kaava

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$$

ei ole tosi, vaikka valitsisimme muuttujien arvot miten tahansa.

Kun haluamme osoittaa, että SAT on NP-täydellinen ongelma, meidän täytyy näyttää, että se kuuluu luokkaan NP ja mikä tahansa luokan NP ongelma voidaan palauttaa siihen. Luokkaan NP kuuluminen on helppoa nähdä: ”kyllä”-tapauksessa todiste on kullekin muuttujalle valittu arvo. Huomattavasti vaikeampaa on osoittaa, että *jokainen* luokan NP ongelma voidaan palauttaa SAT-ongelmaan polynomisessa ajassa.

Tässä kirjassa emme käsittele todistusta yksityiskohtaisesti, mutta voimme kuitenkin kuvailla sen perusidea. Tarkastellaan tiettyä luokan NP ongelmaa, joka meidän täytyy pystyä palauttamaan SAT-ongelmaan. Koska ongelma voi olla mikä tahansa luokkaan NP kuuluva, tiedämme siitä vain, että on olemassa algoritmi, joka tarkastaa polynomisessa ajassa ”kyllä”-tapauksen todisteen. Tämä vastaa sitä, että on olemassa *epädeterministinen* Turingin kone, joka rakentaa ja tarkastaa tällaisen todisteen polynomisessa ajassa. Nyt kun haluamme tarkastaa annetusta syötteestä, onko vastaus siihen ”kyllä”, voimme muodostaa konjunkttiivisessa normaalimuodossa olevan loogisen kaavan, joka luonnehtii Turingin koneen laskentaa, kun koneelle annetaan kyseinen syöte. Voimme muodostaa kaavan niin, että sen voi saada todeksi tarkalleen silloin, kun vastaus syötteeseen on ”kyllä”. Niinpä olemme onnistuneet palauttamaan alkuperäisen ongelman SAT-ongelmaan.

15.2.2 Ongelmien palautukset

Kun tiedämme, että SAT-ongelma on NP-täydellinen, voimme osoittaa muita ongelmia NP-täydellisiksi palautusten avulla. Ideana on, että jos ongelma A on NP-täydellinen ja voimme palauttaa sen polynomisessa ajassa ongelmaksi B , myös ongelma B on NP-täydellinen.

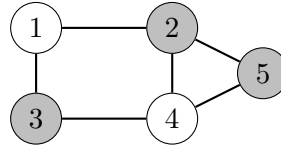
Palautus $\text{SAT} \rightarrow 3\text{SAT}$

Aloitamme osoittamalla, että 3SAT-ongelma on NP-täydellinen. 3SAT-ongelma on SAT-ongelman erikoistapaus, jossa jokaisessa \wedge -merkeillä yhdistetyssä lausekkeessa on tarkalleen kolme muuttujaa. Esimerkiksi kaava

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

on kelvollinen 3SAT-ongelman syöte. Jotta saamme palautettua SAT-ongelman 3SAT-ongelmaan, meidän on näytettävä, että voimme muuttaa polynomisessa ajassa minkä tahansa SAT-ongelman syötteen 3SAT-ongelman syötteeksi, jonka totuusarvo on sama.

Ideana on muokata jokaista SAT-ongelman syötteen lauseketta niin, että tuloksena on yksi tai useampia kolmen muuttujan lausekkeita. Merkitään k :lla lausekkeen muuttujien määrää. Jos $k = 1$ tai $k = 2$, toistamme viimeistä muuttujaa uudestaan, jotta saamme lausekkeeseen kolme muuttujaa. Esimerkiksi jos lauseke on (x_1) , muutamme sen muotoon $(x_1 \vee x_1 \vee x_1)$, ja jos lauseke on $(x_1 \vee x_2)$, muutamme sen muotoon $(x_1 \vee x_2 \vee x_2)$. Jos $k = 3$, meidän ei tarvitse tehdä mitään, koska lausekkeessa on valmiiksi kolme muuttujaa. Jos sitten $k > 3$, jaamme lausekkeen osiin, jotka ketjutetaan uusien



Kuva 15.2: Solmut $\{2, 3, 5\}$ muodostavat solmupeitteen.

apumuuttujien avulla. Ketjun jokaisessa kohdassa vasemman lausekkeen viimeinen muuttuja on a_i ja oikean lausekkeen ensimmäinen muuttuja on $\neg a_i$. Tämä takaa, että ainakin yksi alkuperäinen muuttuja saa oikean arvon. Esimerkiksi jos lauseke on $(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5)$, muutamme sen kolmeksi lausekkeeksi $(x_1 \vee x_2 \vee a_1)$, $(\neg a_1 \vee x_3 \vee a_2)$ ja $(\neg a_2 \vee x_4 \vee x_5)$.

Tämä palautus osoittaa, että 3SAT on NP-täydellinen ongelma, eli SAT-ongelman oleellinen vaikeus syntyy jo siitä, että lausekkeissa voi olla kolme muuttujaa¹. Palautuksen hyötynä on myös se, että myöhemmissä todistuksissa meidän on helpompaa käsitellä kolmen muuttujan lausekkeitä kuin vaihtelevan pituisia lausekkeitä.

Palautus 3SAT \rightarrow solmupeite

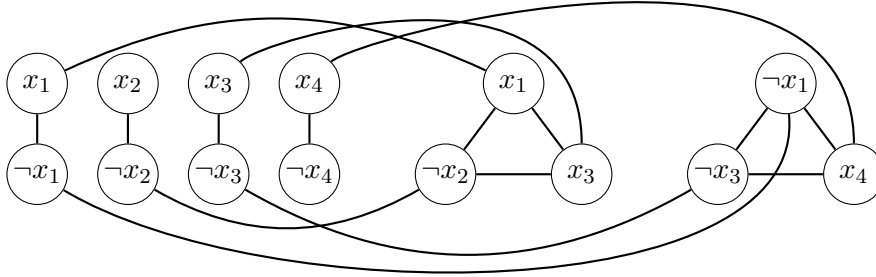
Seuraavaksi osoitamme, että on NP-täydellinen ongelma tarkastaa, onko verkossa *solmupeitetä* (*vertex cover*), jossa on k solmua. Solmupeite on verkon solmujen osajoukko, joka on valittu niin, että jokaisessa kaaressa ainakin toinen päätesolmu kuuluu solmupeitteeseen. Esimerkiksi kuvassa 15.2 on verkko ja sen solmupeite, johon kuuluu kolme solmua.

Kun haluamme palauttaa 3SAT-ongelman solmupeiteongelmaan, meidän täytyy näyttää, että voimme tulkitä minkä tahansa 3SAT-ongelman tapauksen verkon solmupeitteen etsimisenä. Meidän tulee keksiä systemaattinen tapa muuttaa looginen kaava verkoksi, jonka k solmun solmupeite vastaa sitä, että kaava on totta.

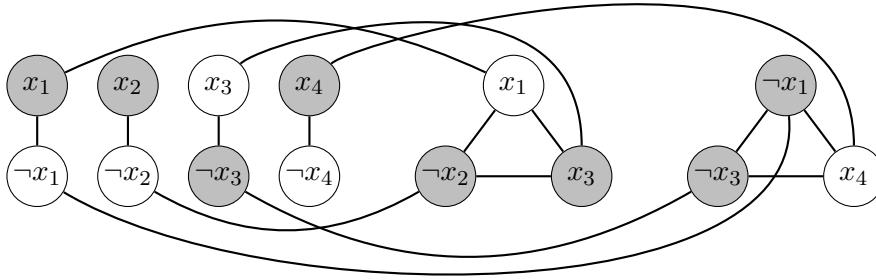
Oletamme, että kaavassa esiintyy n muuttujaa x_1, x_2, \dots, x_n ja siinä on m lauseketta. Muodostamme verkon, jossa on ensinnäkin n solmuparia, jotka vastaavat kaavan muuttujia. Kussakin solmuparissa on muuttujat x_i ja $\neg x_i$, joiden välillä on kaari. Lisäksi verkossa on m kolmen solmun ryhmää, jotka vastaavat lausekkeitä. Jokaisessa ryhmässä kaikki solmut ovat yhteydessä toisiinsa, minkä lisäksi kukin solmu on yhteydessä sitä vastaavaan solmuun pareissa. Esimerkiksi kaavaa

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4),$$

¹Entä ongelma 2SAT, jossa jokaisessa lausekkeessa on kaksi muuttujaa? Tämä *ei* ole NP-täydellinen ongelma, vaan kuuluu luokkaan P.



Kuva 15.3: Kaava $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$ verkkona.

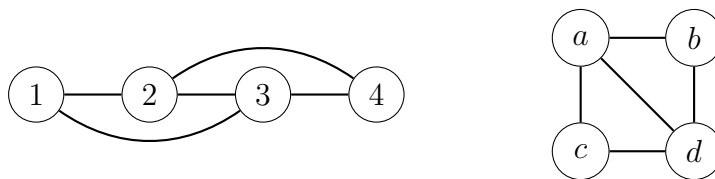


Kuva 15.4: Ratkaisu, jossa x_1 , x_2 ja x_4 ovat tosia ja x_3 on epätosi.

vastaa kuvan 15.3 mukainen verkko.

Osoittautuu, että voimme saada kaavan todeksi tarkalleen silloin, kun verkossa on solmupeite, jossa on enintään $k = n + 2m$ solmua. Tällaiseen peitteeseen kuuluu toinen solmu jokaisesta parista x_i ja $\neg x_i$. Tämä määrittää, miten muuttujien arvot asetetaan. Lisäksi peitteeseen kuuluu kaksi solmua jokaisesta kolmen solmun ryhmästä. Koska ryhmässä on yksi solmu, joka ei kuulu peitteeseen, kyseisen solmun täytyy olla yhteydessä kaarella peitteeseen kuuluvaan solmuun. Tämä varmistaa, että jokaisessa lausekkeessa ainakin yksi kolmesta muuttujasta on asetettu oikein. Kuva 15.4 näyttää esimerkin solmupeitteestä, joka saa kaavan todeksi. Tämä vastaa ratkaisua, jossa x_1 , x_2 ja x_4 ovat tosia ja x_3 on epätosi.

Olemme siis onnistuneet palauttamaan 3SAT-ongelman solmupeiteongelmaksi niin, että verkon koko on polynominen suhteessa kaavan pituuteen,



Kuva 15.5: Kaksi verkkoa, jotka ovat isomorfiset.

joten solmupeiteongelma on NP-täydellinen.

15.2.3 Lisää ongelmia

Palautusten avulla on onnistuttu löytämään tuhansia NP-täydellisiä ongelmia. Esimerkiksi myös seuraavat ongelmat ovat NP-täydellisiä:

- onko verkossa k -kokoista *klikkiä* (*clique*) eli k solmun joukkoa, jossa jokaisen kahden solmun välillä on kaari?
- voimmeko värittää verkon solmut kolmella värillä niin, että jokaisen kaaren päätesolmut ovat eri värisiä?
- onko verkossa polkua, joka kulkee tasan kerran jokaisen verkon solmun kautta (eli *Hamiltonin polkua* (*Hamiltonian path*))?
- voiko annetuista n luvusta valita osajoukon, jonka summa on x ?
- onko olemassa k -merkkistä merkkijonoa, jonka alijonoja ovat kaikki annetut merkkijonot?

Entä millainen olisi luokan NP ongelma, joka ei kuulu luokkaan P eikä ole NP-täydellinen? Kukaan ei tiedä, onko tällaista ongelmaa olemassa, koska ei edes tiedetä, ovatko P ja NP eri luokat. Yksi ehdokas tällaiseksi ongelmaksi on kuitenkin ongelma, jossa haluamme tarkastaa, ovatko kaksi verkkoa *isomorfiset* eli onko verkkojen rakenne samanlainen, jos solmut asetetaan vastaamaan toisiaan sopivalla tavalla. Esimerkiksi kuvassa 15.5 olevat verkot ovat isomorfiset, koska voimme valita solmuille vastaavuudet $(1, c)$, $(2, a)$, $(3, d)$ ja $(4, b)$.

Verkkojen isomorfisuuden ongelma kuuluu luokkaan NP, koska on helppoa tarkastaa, onko verkoilla sama rakenne, jos tiedämme solmujen vastaavuudet. Ongelmaan ei kuitenkaan tunneta polynomiaikaista algoritmia, joten sen ei tiedetä kuuluvan luokkaan P. Toisaalta emme myöskään osaa tehdä mitään palautusta, joka osoittaisi ongelman olevan NP-täydellinen.

15.2.4 Optimointiongelmat

Käytännössä haluamme usein ratkaista päätösongelman sijasta *optimointiongelman* (*optimization problem*): haluamme etsiä pienimmän tai suurimman mahdollisen ratkaisun. Esimerkiksi emme halua tarkastaa, onko verkossa enintään k solmun solmupeitettä (pätösongelma), vaan haluamme etsiä *pienimmän* solmupeitteen (optimointiongelma). Osoittautuu kuitenkin, että päätösongelmat ja optimointiongelmat ovat loppujen lopuksi hyvin lähellä toisiaan.

Oletetaan, että meillä on keino tarkastaa tehokkaasti, onko verkossa k solmun solmupeitettä. Miten voimme menetellä, jos haluammekin etsiä pienimmän solmupeitteen? Ratkaisuna on käyttää *binäärihakua*: etsimme pienimmän arvon x , jolle pätee, että verkossa on x solmun solmupeite. Tämä tarkoittaa, että verkon pienin solmupeite sisältää x solmua. Koska käytämme binäärihakua, meidän riittää ratkaista päätösongelma vain logaritminen määrä kertoja, joten saamme ratkaistua optimointiongelman lähes yhtä tehokkaasti kuin päätösongelman.

Sanomme, että ongelma on *NP-vaikea* (*NP-hard*), jos voimme palauttaa kaikki luokan NP ongelmat siihen mutta ongelman ei tarvitse kuulua luokkaan NP. Jos ongelma on NP-vaikea, se on siis ainakin yhtä vaikea kuin luokan NP vaikeimmat ongelmat. Tyypillisiä NP-vaikeita ongelmia ovat NP-täydellisten päätösongelmien optimointiversiot, koska voimme palauttaa niihin NP-täydellisiä ongelmia mutta ne eivät kuulu luokkaan NP.

Käytännössä termejä ei käytetä aina näin täsmällisesti ja optimointiongelmaa saatetaan sanoa NP-täydelliseksi, vaikka se oikeastaan on NP-vaikea. Voimme myös vain puhua yleisesti NP-ongelmista, kun tarkoitamme NP-täydellisiä ja NP-vaikeita ongelmia.

15.3 Ongelmien ratkaiseminen

Jos saamme ratkaistavaksemme NP-vaikean ongelman, tilanne ei näytä hyvältä, koska edessämme on silloin ongelma, johon ei tunneta mitään tehokasta algoritmia. Emme voi toivoa, että osaisimme ratkaista tehokkaasti ongelmaa, jota kukaan muukaan ei ole osannut. Peli ei ole kuitenkaan välttämättä vielä menetetty, vaan voimme koettaa lähestyä ongelmaa monella tavalla.

Tarkastellaan seuraavaksi ongelmaa, jossa meille annetaan n kokonaislukua ja haluamme jakaa luvut kahteen ryhmään niin, että ryhmien summat ovat mahdollisimman lähellä toisiaan. Esimerkiksi jos $n = 5$ ja luvut ovat $[1, 2, 4, 5, 7]$, paras ratkaisu on muodostaa ryhmät $[1, 2, 7]$ ja $[4, 5]$, joiden summat ovat $1 + 2 + 7 = 10$ ja $4 + 5 = 9$. Tällöin summien ero on 1, eikä

ole olemassa ratkaisua, jossa ero olisi 0.

Tämä ongelma on NP-vaikea, joten kukaan ei tiedä tehokasta algoritmia sen ratkaisemiseen. Seuraavaksi käymme läpi joukon tapoja, joiden avulla voimme kuitenkin yrittää ratkaista ongelmaa.

Raaka voima

Vaikka ongelma olisi NP-vaikea, voimme silti ratkaista sen pieniä tapauksia. Tässä tehtävässä voimme tehdä algoritmin, joka käy läpi kaikki tavat jakaa luvut kahteen ryhmään ja valitsee parhaan vaihtoehdon. Tämä vastaa sitä, että käymme läpi kaikki tavat valita toiseen ryhmään tulevat luvut eli kaikki lukujen osajoukot.

Koska n luvusta voi muodostaa 2^n osajoukkoa, tällainen algoritmi vie aikaa $O(2^n)$. Voimme toteuttaa algoritmin esimerkiksi rekursiolla samaan tapaan kuin teimme luvussa 1. Tuloksena oleva algoritmi on käyttökelpoinen, jos n on niin pieni, että ehdimme käydä kaikki osajoukot läpi. Käytännössä voimme ratkaista tehokkaasti tapauksia, joissa n on enintään noin 20.

Heuristiikat

Yksi mahdollisuus selviytyä vaikeasta ongelmasta on tyytyä optimaalisen ratkaisun sijasta johonkin *melko hyvään* ratkaisuun, joka ei välttämättä ole optimaalinen. Saamme tällaisen ratkaisun aikaan keksimällä jonkin *heuristiikan* (*heuristic*), joka muodostaa ratkaisun. Heuristiikka on jokin järkevä sääntö, jonka avulla algoritmi rakentaa ratkaisua eteenpäin askel kerrallaan.

Tässä tehtävässä yksinkertainen heuristiikka on käydä luvut läpi suurimmasta pienimpään ja sijoittaa luku aina ryhmään, jonka summa on sillä hetkellä pienempi. Jos kummankin ryhmän summa on yhtä suuri, sijoitamme luvun ensimmäiseen ryhmään. Tämä on järkevä heuristiikka, koska näin saamme monessa tapauksessa jaettua lukuja melko tasaisesti ryhmiin.

Esimerkiksi jos luvut ovat $[1, 2, 4, 5, 7]$, sijoitamme ensin luvun 7 ensimmäiseen ryhmään, sitten luvut 5 ja 4 toiseen ryhmään ja lopuksi luvut 2 ja 1 ensimmäiseen ryhmään. Saamme muodostettua näin ryhmät $[1, 2, 7]$ ja $[4, 5]$, joiden summien ero on 1 eli ratkaisu on optimaalinen.

Tällainen algoritmi ei kuitenkaan tuota aina optimaalista ratkaisua. Kun tunnemme algoritmin toiminnan, voimme keksiä tilanteita, joissa se antaa huonomman ratkaisun. Tällainen tapaus on esimerkiksi $[2, 2, 2, 3, 3]$. Heuristiikkaa käyttävä algoritmi tuottaa ryhmät $[2, 3]$ ja $[2, 2, 3]$, joiden summien ero on 2. Kuitenkin optimaalinen ratkaisu olisi muodostaa ryhmät $[2, 2, 2]$ ja $[3, 3]$, joiden summien ero on 0.

Satunnaisuus

Voimme hyödyntää ongelman ratkaisemisessa myös *satunnaisuutta*. Voimme tehdä algoritmin, joka tekee satunnaisia valintoja, ja lisäksi toistaa algoritmia monta kertaa ja valita parhaan ratkaisun.

Esimerkiksi voimme parantaa äskeistä heuristista algoritmia niin, että se sekoittaa alussa lukujen järjestyksen. Tämän jälkeen voimme toistaa algoritmia vaikkapa miljoona kertaa ja valita parhaan ratkaisun. Tämä voi parantaa merkittävästi algoritmin toimintaa, ja on vaikeampaa keksiä tapauksia, joissa algoritmi toimii huonosti.

Rajoittaminen

Joskus voimme saada aikaan tehokkaan algoritmin, kun rajoitamme sopivalla tavalla tehtävää. Esimerkiksi tässä tehtävässä on mahdollista luoda tehokas dynaamisen ohjelmoinnin ratkaisu, kunhan oletamme, että luvut ovat sopivan pieniä kokonaislukuja.

Ideana on laskea tarkastella osaongelmia muotoa ”voimmeko valita a ensimmäisestä luvusta osajoukon, jonka lukujen summa on b ?” Jos luvut ovat pieniä, niin myös summat ovat pieniä, joten voimme ratkaista kaikki tällaiset osaongelmat rekursiivisesti. Tämän jälkeen riittää etsiä mahdollinen jako, jossa $a = n$ ja b on mahdollisimman lähellä lukujen summan puoliväliä.

Tällainen ratkaisu vie aikaa $O(ns)$, missä s on lukujen summa. Sanomme, että algoritmi on *pseudopolynominen*, koska sen aikavaativuus on polynominen mutta riippuu lukuarvojen suuruudesta. Yleisessä tapauksessa syötteenä annetut luvut voivat kuitenkin olla suuria kokonaislukuja, jolloin emme voi käyttää tällaista ratkaisua.

Liite A

Matemaattinen tausta

Tämä liite käy läpi matematiikan merkintöjä ja käsitteitä, joita käytetään kirjassa algoritmien analysoinnissa.

Merkintöjä

Merkinnät $\lfloor x \rfloor$ ja $\lceil x \rceil$ tarkoittavat, että pyöristämme luvun x alaspäin ja ylöspäin kokonaisluvuksi. Esimerkiksi $\lfloor 5.23 \rfloor = 5$ ja $\lceil 5.23 \rceil = 6$.

Kertoma $n!$ lasketaan $1 \cdot 2 \cdot 3 \cdots n$. Esimerkiksi $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

Merkintä $a \bmod b$ tarkoittaa, mikä on jakojäännös, kun a jaetaan b :llä. Esimerkiksi $32 \bmod 5 = 2$, koska $32 = 6 \cdot 5 + 2$.

Summakaavat

Voimme laskea lukujen $1, 2, \dots, n$ summan kaavalla

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

Esimerkiksi

$$1 + 2 + 3 + 4 + 5 = \frac{5 \cdot 6}{2} = 15.$$

Kaavan voi ymmärtää niin, että laskemme yhteen n lukua, joiden suuruus on *keskimäärin* $(n+1)/2$.

Toinen hyödyllinen kaava on

$$2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1.$$

Esimerkiksi

$$1 + 2 + 4 + 8 + 16 = 32 - 1.$$

Tässä voimme ajatella, että aloitamme luvusta 2^n ja lisäämme siihen aina puolet pienemmän luvun lukuun 1 asti. Tämän seurauksena pääsemme yhtä vaille lukuun 2^{n+1} asti.

Logaritmi

Logaritmin määritelmän mukaan $\log_b n = x$ tarkalleen silloin kun $b^x = n$. Esimerkiksi $\log_2 32 = 5$, koska $2^5 = 32$.

Logaritmi $\log_b n$ kertoo, montako kertaa meidän tulee jakaa luku n luvulla b , ennen kuin pääsemme lukuun 1. Esimerkiksi $\log_2 32 = 5$, koska tarvitsemme 5 puolitusta:

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

Tässä kirjassa oletamme, että logaritmin kantaluku on 2, jos ei ole toisin mainittu, eli voimme kirjoittaa lyhyesti $\log 32 = 5$.

Logaritmeille pätevät kaavat

$$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$$

ja

$$\log_b(x/y) = \log_b(x) - \log_b(y).$$

Ylemmästä kaavasta seuraa myös

$$\log_b(x^k) = k \log_b(x).$$

Lisäksi voimme vaihtaa logaritmin kantalukua kaavalla

$$\log_u(x) = \frac{\log_b(x)}{\log_b(u)}.$$

Odotusarvo

Odotusarvo kuvaa, mikä on keskimääräinen tulos, jos satunnainen tapahtuma toistuu monta kertaa eli mitä tulosta voimme tavallaan odottaa.

Kun tapahtumalla on n mahdollista tulosta, odotusarvo lasketaan kaavalla

$$p_1 t_1 + p_2 t_2 + \cdots + p_n t_n,$$

missä p_i ja t_i ovat tapahtuman i todennäköisyys ja tulos. Esimerkiksi kun heitämme noppaa, tuloksen odotusarvo on

$$1/6 \cdot (1 + 2 + 3 + 4 + 5 + 6) = 7/2.$$

Hakemisto

- O -merkintä, 12, 19
- Ω -merkintä, 19
- Θ -merkintä, 19
- 3SAT-ongelma, 169

- ahne algoritmi, 92
- aikavaativuus, 11
- algoritmi, 1
- aliohjelma, 5
- alipuu, 65
- aste, 111
- AVL-ehto, 72
- AVL-puu, 71

- Bellman–Fordin algoritmi, 124
- binäärihaku, 38, 96
- binäärihakupuu, 65
- binäärikeko, 81
- binääripuu, 65
- binomikerroin, 109
- BPP-luokka, 167

- Dijkstran algoritmi, 127
- dynaaminen ohjelmointi, 101, 139

- Edmonds–Karpin algoritmi, 159
- ehtolause, 4
- erilliset polut, 160
- esijärjestys, 67
- etäisyys, 119, 123
- etäisyysmatriisi, 131

- Floyd–Warshallin algoritmi, 131
- Ford–Fulkersonin algoritmi, 156
- funktio, 5

- hajautusarvo, 55
- hajautusfunktio, 55
- hajautustaulu, 55
- hakemisto, 58
- Hamiltonin polku, 172
- harva verkko, 133
- heuristiikka, 174
- hybridialgoritmi, 33

- implisiittinen verkko, 120
- invariantti, 25
- inversio, 27
- isomorfiset verkot, 172
- iteraattori, 51

- jälkijärjestys, 67
- järjestäminen, 25, 93
- jakojäännös, 177
- jono, 48
- juuri, 65

- kaari, 111
- kaarilista, 115
- kaksijakoinen verkko, 161
- keko, 81
- kekoehto, 81
- kekojärjestäminen, 88
- kertoma, 177
- kierto, 73
- klikki, 172
- komponentti, 112
- komponenttiverkko, 141
- Kosarajun algoritmi, 141
- Kruskalin algoritmi, 149

- kuutiollinen algoritmi, 16
- lähin pienempi edeltäjä, 98
- lapsi, 65
- laskemisjärjestäminen, 34
- latinalainen neliö, 9
- lehti, 65
- leikkaus, 157
- leveyshaku, 119
- lineaarinen algoritmi, 15
- linkitetty lista, 44
- lisäysjärjestäminen, 25
- lista, 41
- logaritmi, 178
- logaritminen algoritmi, 14
- lomitussjärjestäminen, 28
- lyhin polku, 119, 123
- maksimikeko, 81
- maksimiparitus, 161
- maksimivirtaus, 155
- metodi, 5
- minimikeko, 81
- minimileikkaus, 157
- muuttuja, 4
- naapuri, 111
- neliöllinen algoritmi, 15
- NP-luokka, 166
- NP-ongelma, 165
- NP-täydellinen ongelma, 168
- NP-vaikea ongelma, 173
- odotusarvo, 178
- offline-algoritmi, 62
- online-algoritmi, 62
- optimointiongelma, 173
- osajoukko, 6
- P vs. NP, 166
- P-luokka, 165
- päätösongelma, 165
- painotettu verkko, 113
- palautus, 160, 169
- paritus, 161
- permutaatio, 8
- peruuttava haku, 8
- pienin polkupeite, 162
- pienin virittävä puu, 149
- pikajärjestäminen, 30
- pino, 48, 98
- psin nouseva alijono, 104
- polku, 112
- polkujen laskeminen, 139
- polkupeite, 162
- polynominen algoritmi, 165
- polynominen hajautus, 57
- Primin algoritmi, 151
- prioriteettijono, 85
- proseduuri, 5
- pseudokoodi, 2
- pseudopolynominen algoritmi, 175
- PSPACE-luokka, 167
- puu, 112
- rekursio, 6
- repunpakkaus, 107
- SAT-ongelma, 168
- satunnainen algoritmi, 175
- silmukka, 4
- sisäjärjestys, 67
- solmu, 111
- solmupeite, 170
- summakaava, 177
- suunnattu sykliton verkko, 135
- suunnattu verkko, 113
- suurin virittävä puu, 153
- syöte, 1
- sykli, 112
- syklin etsiminen, 118, 136
- sykliton verkko, 112
- syvyyshaku, 117

tasapainoinen puu, 71
tasoitettu analyysi, 98
taulukko, 4
taulukkolista, 41
taulukointi, 104
tietorakenne, 4, 95
tiheä verkko, 133
tilavaativuus, 21
todiste, 166
topologinen järjestys, 136
tuloste, 1

union-find-rakenne, 145

väritys, 172
vaativuusluokka, 165
vahvasti yhtenäinen komponentti, 141
vahvasti yhtenäinen verkko, 141
vakioaikainen algoritmi, 14
vakiokerroin, 12, 33
vanhempi, 65
verkko, 111
vieruslista, 114
vierusmatriisi, 116
virittävä puu, 149
virtaus, 155

yhtenäinen komponentti, 112
yhtenäinen verkko, 112

ZPP-luokka, 167