Algoritmit 1

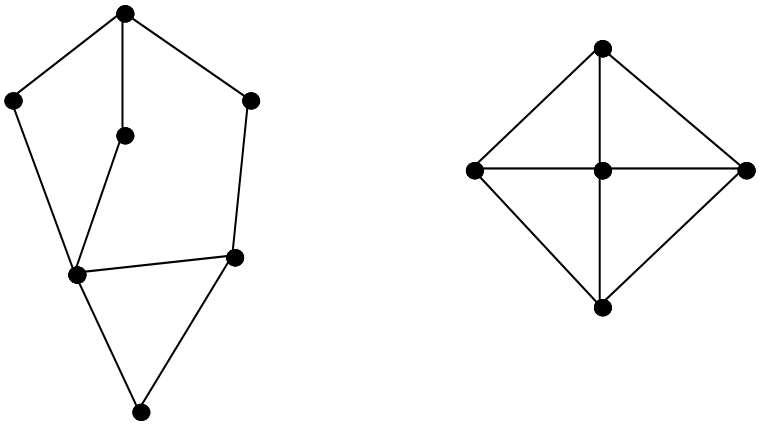
Syksy 2008

1 Algoritmit ja algoritmien analysointi

Algoritmilla tarkoitetaan askel askeleelta suoritettavaa ohjetta jonkin tehtävän suorittamiseksi tai jonkun ongelman ratkaisemiseksi. ”Hyvän” algoritmin on oltava myös tehokas ja se tarvitsee toteutusta varten sille sopivan tietorakenteen. Tietorakenne on systemaattinen tapa tallettaa algoritmin tarvitsemaa tietoa.

1.1 Muutamia esimerkkejä ongelmista

- ***P*$_1$**: Annettu kokonaisluvut n ja m. Laske tulo $n \cdot m$. Ongelmalle löytyy tehokkaita ratkaisualgoritmeja.
- ***P*$_2$**: Annettu kokonaislukujono $A = a_1, a_2, \ldots, a_n$. Järjestä luvut nousevaan järjestykseen. Ongelmalle löytyy runsaasti erilaisia algoritmeja ja myös tehokkaita ratkaisualgoritmeja.
- ***P*$_3$**: Annettu verkko G. Sisältääkö G Eulerin polun? Eulerin polku on yhtenäinen polku, joka sisältää verkon jokaisen tien täsmälleen kerran. Myös tälle ongelmalle löytyy tehokkaita ratkaisualgoritmeja.



- ***P*$_4$**: Annettu verkko G. Sisältääkö G Hamiltonin kehän? Hamiltonin kehä on verkon silmukka, joka sisältää verkon kaikki solmut täsmälleen kerran. Ongelma kuluu niin sanottuun NP-täydellisten ongelmien joukkoon. Sille ei luultavasti ole olemassa tehokasta ratkaisualgoritmia.
- ***P*$_5$**: Annettu C-kielinen funktio A ja sen syöte x. Pysähtyykö A:n suoritus syötteellä x? Ongelma P_5 on algoritmisesti ratkeamaton (Turing/1936).

  Todistus:
  Oletetaan, että P_5:llä olisi ratkaisualgoritmi, so. kaikilla syötteillä pysähtyvä testifunktio:

```
boolean H(text p, text x)
  // p ohjelmateksti, x merkkijono,
  // H(p, x) = true,  jos p pysähtyy syötteellä x
  // H(p, x) = false, jos p ei pysähdy syötteellä x
{
  ...
```

```
}
```

Määritellään tätä käyttäen uusi funktio:

```
boolean K(text p)
{
  if (H(p, p))
    while true do ; // ikuinen silmukka
  else
    return true;
}
```

Merkitään funktion `K` tekstiä `k`:lla ja tarkastellaan laskentaa `K(k)`:

`K(k)` pysähtyy, jos ja vain jos `H(k, k)` = false ja `H(k, k)` = false, jos ja vain jos `K(k)` ei pysähdy. Saadusta ristiriidasta seuraa, ettei oletetun kaltaista pysähtymisen testausfunktiota `H` voi olla olemassa.

Ongelmalle P_5 saadaan kuitenkin osittaisratkaisu yksinkertaisesti simuloimalla:

```
boolean K'(text p)
{
  // simuloi ohjelman p toimintaa syötteellä x;
  // jos p pysähtyy niin palauta true
}
```

Jos `p` pysähtyy syötteellä `x`, funktio `K'` palauttaa arvon true, muuten se ei pysähdy.

- ***P*$_6$**: Annettu C-kielinen funktio *A*. Pysähtyykö *A*:n suoritus kaikilla syötteillä? Ongelmalla P_6 ei ole edes tällaista osittaisratkaisua.

1.2 Johdatusta algoritmin analysointiin

Algoritmille ei ole mitään yleisesti hyväksyttyä määritelmää. Yleensä *algoritmi*lla tarkoitetaan jonoa ristiriidattomia askel askeleelta suoritettavia käskyjä, joiden avulla saadaan jokin ongelma ratkaistua tai jokin tehtävä suoritettua. Mutta yleisesti algoritmilla voidaan tarkoittaa ohjetta, suunnitelmaa, reseptiä, ... Tällä kurssilla algoritmilla tarkoitetaan yleensä lopulta tietokoneohjelmana toteutettavaa ongelman ratkaisumenetelmää.

Algoritmilla on syöttötietoa ja se tuottaa arvon tai joukon arvoja tulostuksena. Algoritmi siis muodostuu jonosta askelia, jotka muuntavat syöttötiedon tulostustiedoiksi. Täten puhutaan usein *laskennallisesta ongelma*sta ja sellaisen ratkaisemisesta. Yleensä vaatimuksena on, että algoritmin suorittaman askelmäärän tulisi olla äärellinen. Olennaisena osana tietokoneohjelmana toteutettavaa algoritmia on systemaattinen tapa säilyttää ja käyttää ongelman ratkaisussa tarvittavaa tietoa eli ohjelman tietojen talletusrakenne eli *tietorakenne*.

Algoritmin olennaisena vaatimuksena on, että se antaa oikean vastauksen. Joskus haetaan parasta ratkaisua useiden mahdollisten ratkaisujen joukosta. Paras ratkaisu voidaan tunnistaa jonkin kriteerin mukaan. Tällöin tämän kaikista parhaan ratkaisun selvittäminen voi olla niin työlästä, että tyydytään ”vain” suhteellisen hyvään ratkaisuun, vaikka se ei olisikaan välttämättä paras mahdollinen. Kyseessä on silloin likiarvoisesti eli approksimatiivisesti hyvän ratkaisun selvittäminen.

Algoritmi tehdään ongelman ratkaisua varten. Kun ongelmalle annetaan tarkat lähtötiedot, niin tarkoitetaan ongelman tiettyä *esiintymä*ä. Eri lähtötiedoilla saadaan ongelmalle eri esiintymiä.

Ongelma voidaan ratkaista aina useammalla erilaisella algoritmilla. Täten tavoitteeksi voidaan asettaa mahdollisimman hyvän eli jonkin mittarin mukaan tehokkaan algoritmin löytäminen. Algoritmin tehokkuuden selvittämisellä tullaan tarkoittamaan *algoritmin analysointi*a. Algoritmin analysoinnissa yritetään selvittää tai ainakin ennustaa algoritmin tarvitsemia resursseja. Näitä resursseja voivat olla esimerkiksi muistitilan käyttö, kommunikaatioväylän kaistanleveys tai muita tietokonelaitteiston ominaisuuksia. Yleisin tarkasteltava resurssi on algoritmin tarvitsema tietokoneen *suoritusaika*.

Jos ratkaistavana on yksi tai kaksi "pientä" ongelman esiintymää, niin ratkaisuun käytettävän algoritmin tarvitsemien resurssien määrällä ei ole suurta merkitystä. Silloin algoritmin valintaan ei yleensä tarvitse kiinnittää suurempaa huomiota. Tietysti silloinkin algoritmi tulee osata tarvittaessa tehdä. Voidaan kuitenkin valita jokin yksinkertainen tai jokin jo valmiina oleva toteutettu algoritmi. Jos kuitenkin on ratkaistava useampia ongelman esiintymiä tai ongelma on vaikeasti ratkaistava tai ”suurikokoinen”, niin algoritmin valinta on tehtävä huolella.

Miten sitten tulisi selvittää algoritmin suoritusaika eli miten mitata algoritmin suoritusaikaa? Jos algoritmi on implementoitu, niin sen suoritusaika on periaatteessa selvitettävissä testaamalla. Tällöin suorittamalla algoritmi erilaisilla syötteillä voidaan mitata ”tarkat” suoritusajat näille ongelman esiintymille. Tavallisesti ollaan kiinnostuneita suoritusajan riippuvuudesta verrattuna erilaisiin ongelman syöttötietoihin, varsinkin erisuuruisiin syöttötiedon määriin. Siten algoritmi tulee suorittaa erilaisilla ja erikokoisilla ongelman esiintymillä.

Normaalisti suoritusaika kasvaa syöttötiedon määrän kasvaessa. Mutta myös samalla syöttötiedon koolla suoritusajat voivat vaihdella. Suoritusaikaan vaikuttaa syöttötiedon lisäksi tietysti käytettävä laitteistoympäristö. Mutta niin myös se ohjelmistoympäristö, jossa algoritmi on toteutettu. Lisäksi huomattava, että tällaisia suoritusaikamittauksia voidaan tehdä vain rajoitetulle joukolle ongelman esiintymiä. Kun verrataan kahta erilaista algoritmia, niin testit tulisi suorittaa täsmälleen samanlaisilla laitteistoilla ja täsmälleen samoissa ohjelmistoympäristöissä. Edelleen haittapuolena on se, että tarkkojen testien suorittamiseksi on algoritmit välttämättä toteuttava tietokoneella.

Yllä esitetyn kaltainen aikamittaukseen perustuva suoritusajan analysointi on siis monessa mielessä vaikeaa ja ongelmallista. Algoritmin suoritusajalle olisi hyvä löytää

mahdollisimman yleinen mittari, joka ensinnäkin huomioisi kaikki mahdolliset syöttötiedot ja toisaalta mahdollistaisi kahden algoritmin suhteellisen tehokkuuden vertailun riippumatta laitteisto- ja ohjelmointiympäristöstä. Edelleen algoritmin tehokkuutta tulisi voida mitata siten, että algoritmia ei tarvitsisi toteuttaa millään ohjelmointikielellä eikä siis toteuttaa sitä tietokoneella.

Yleisempää algoritmin suoritusajan vaativuutta voidaan selvittää laskemalla esimerkiksi algoritmin suorittamien operaatioiden lukumäärää tai askeleiden määrää.

Yritetään selvittää alla olevan metodin tehokkuutta suoritusajan suhteen. Metodi laskee taulukossa olevien kokonaislukujen summan. Olkoon taulukon alussa `n` kappaletta kokonaislukuja.

```
int laskeSumma(int [] t, int n) {
  int summa = 0;
  for (int i = 0; i < n; i++) {
    summa = summa + t[i];
  }
  return summa;
}
```

Metodin toiminta on aina samanlainen taulukon alkioista riippumatta. Muuttuja `summa` alustetaan ja siihen lisätään `n` kokonaislukua. Metodin suoritus voidaan selvittää melko tarkasti. Jos taulukon alkioiden lukumäärä muuttuu niin silti pystymme laskemaan kunkin rivin ja operaation suorituskertojen lukumäärät tarkasti alkioiden lukumäärän `n` (`n > 0`) funktiona. Ohjelman suoritusaika täten riippuu taulukon alkioiden lukumäärästä `n` eli syöttötiedon koosta. Kun alkioita on n kappaletta niin suoritusaika `T(n)` voisi yhtälönä olla muotoa $T(n) = n*t_1 + t_2$., missä t_1 on for-silmukan yhden kierroksen viemä aika ja t_2 on muuhun suoritukseen kuluva aika.

Tarkastellaan toista esimerkkiä, joka hakee taulukon `n`:sta alkiosta suurimman.

```
int haeSuurin(int [] t, int n) {
  int suurin = t[0];
  for (int i = 1; i < n; i++) {
    if (suurin < t[i])
      suurin = t[i];
  }
  return suurin;
}
```

Tässä esimerkissä for-silmukan runko suoritetaan aina `n-1` kertaa. Kuitenkin sijoitus `suurin = t[i]` suoritetaan vaihteleva määrä kertoja riippuen siitä millaisia taulukon alkiot ovat. Muuttujalle `suurin` voidaan asettaa arvo vaihteleva määrä esiintymästä riippuen. Täten suoritusaika riippuu myös itse datan sisällöstä eli ongelman esiintymästä. Tässä suoritusaika vaihtelee, mutta for-silmukan runko suoritetaan joka tapauksessa `n-1` kertaa.

Tarkastellaan seuraavaksi esimerkkiä, joka hakee `n` alkiota sisältävästä taulukosta jonkin määrätyn alkion esiintymispaikkaa taulukossa. Taulukon alkiot voivat olla vapaasti missä järjestyksessä tahansa, joten suoritetaan taulukon alkioille peräkkäishaku taulukon alusta loppua kohti.

```
int haeAlkio(int [] t, int n, int alkio) {
  for (int i = 0; i < n; i++) {
    if (t[i] == alkio)
      return i;
  }
  return -1;
}
```

Metodi palauttaa arvon -1, jos alkiota ei löydy taulukosta. Tässä esimerkissä ei voida tietää kuinka monta kertaa for-silmukan runko suoritetaan. Haettava alkio voi olla taulukon missä alkiossa tahansa, joten silmukka suoritetaan vähintään kerran ja enintään `n` kertaa.

Metodin suoritusaika siis vaihtelee ongelman esiintymän mukaan. Voidaankin arvioida erilaisia suoritusaikoja. Paras mahdollinen suoritusaika tässä esimerkissä on vakioaika eli kun haettava alkio löytyy taulukon ensimmäisestä alkiosta. Pahin mahdollinen suoritusaika tapahtuu silloin, kun haettavaa alkiota ei löydy lainkaan, jolloin `for` silmukan runko suoritetaan `n` kertaa. Tämä antaa rajan, jota ei ylitetä. Yksi mahdollisuus on selvittää keskimääräinen suoritusaika. Silloin tarkastelu voidaan jakaa kahteen osaan. Jos haettavaa alkiota ei löydy taulukosta, niin silmukka suoritetaan aina `n` kertaa. Toisaalta jos haettava alkio löytyy, niin haettava alkio voi olla mikä tahansa taulukon alkioista. Jos haettava alkio on taulukon ensimmäinen alkio, niin suoritetaan yksi vertailu, jos haettava alkio on taulukon toinen alkio, niin suoritetaan kaksi vertailua ja yleisesti, jos haettava alkio on taulukon `k`:s alkio, niin suoritetaan k vertailua. Jos kaikkia alkioita haetaan samalla todennäköisyydellä, `1/n`, niin voidaan laskea keskimääräinen vertailujen lukumäärä (tässä on käytetty summakaavaa ($1+2+3+\cdots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$)

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

Joskus ollaan kiinnostuneita nimenomaan keskimääräisestä suoritusajasta. Silloin eräs vaikeus on ongelmatilanteen keskimääräisen syöttöaineiston määrittäminen. Usein silloin oletetaan, että kaikki mahdolliset syöttöesiintymät ovat yhtä todennäköisiä.

Tarkan suoritusajan laskenta ei siis useimmiten ole mahdollista ja jos se on mahdollista niin se saattaa olla varsin hankalaa. Lisäksi tarkka lopputulos on varsin hankalaa muotoa.

Pyritään yksinkertaiseen ratkaisuun suorittaen algoritmin tehokkuuden tarkastelu käyttäen analyyttisempää lähentymistapaa, joka sopii suoraan algoritmin selkeämpään esitystapaan mielellään ilman ohjelmakoodia. Tehokkuuden mittauksessa ei silloin käytetä ulkoisista tekijöistä riippuvaa mittayksikköä kuten sekunteja. Tällöin voidaan

laskea algoritmien suorittamien operaatioiden tai askeleiden lukumääriä. Kaikkia operaatioita ei kannata laskea. Valitaan jokin perusoperaatio tai joitakin perusoperaatiota ja lasketaan niiden lukumääriä.

Perusoperaation valinta ei yleensä ole vaikeaa; se voisi olla eniten aikaa vievä operaatio algoritmin sisimmässä silmukassa tai useimmiten suoritettava operaatio. Esimerkiksi esitetyssä alkion hakuongelmassa perusoperaatio voisi olla vertailuoperaatio tai silmukkamuuttujan arvon sijoitusoperaatio tai molemmat.

Jos tarkastellaan vaikkapa suurimman alkion etsimisongelmaa, niin havaitaan, että mitä enemmän taulukossa on alkioita niin sitä kauemmin etsiminen luultavasti kestää. Täten perusoperaatioiden lukumäärä riippuu tässä tapauksessa varmasti tutkittavien alkioiden lukumäärästä. Perusoperaatioiden lukumäärä lasketaankin *syöttötiedon koon funktiona*, esimerkiksi tutkittavien alkioiden lukumäärän funktiona. Jos syöttötiedon koko on *n*, niin perusoperaatioiden lukumäärä voidaan kirjoittaa funktiomuodossa *f(n)*. Syöttötiedon koko on tavallisesti helposti havaittava parametri. Jos *n* kappaletta alkioita halutaan laskea yhteen, niin syöttötiedon koko on *n*. Jos *n*:n alkion joukosta halutaan hakea joku määrätty alkio, niin syöttötiedon koko on myös *n*.

Algoritmeja voidaan esittää usealla eri tavalla. Ohjelmointikieli on eräs tapa. Se on tarkka, mutta monesti yksityiskohdat häivyttävät olennaisia algoritmin ideoita. Tämän vuoksi usein käytetään *pseudokoodi*a. Siinä voidaan käyttää osittain tavallisimpia ohjelmointikielten lauseiden rakenteita, mutta usein mukana on myös vapaasti tekstiä kuvaamaan joitakin algoritmin toimintoja. Analysointia varten on kuitenkin oltava selvyys myös vapaana tekstinä kuvattujen vaiheiden tehokkuuksista. Joskus algoritmin askeleet eli vaiheet kuvataan vapaasti tekstillä ja askeleet numeroidaan. Siirtyminen muualle kuin seuraavaan askeleeseen ilmoitetaan askeleiden numeroiden avulla. Erilaisia kaavioita ja päätöspuita näkee myös usein käytettävän.

Analysoitaessa algoritmin tehokkuutta ei siis käytetä mittayksikkönä mitään standardia aikayksikköä. Käytetään aikaisemman mukaan perusoperaatioiden lukumäärää. Tällöin ikään kuin oletetaan kaikkien operaatioiden suoritusaikojen olevan suunnilleen samaa luokkaa. Tällainen lähestymistapa algoritmien kuvauksen käytössä on laskennallinen malli nimeltä RAM (Random Access Machine). Se on eräänlainen teoreettinen vastine olemassa oleville tietokoneille. Voidaan ajatella käytettävän myös muita formalismeja, Turingin koneet [Turing 1936], eri ohjelmointikielet, (FORTRAN, BASIC, PASCAL, C, C++, JAVA, ...), λ-kalkyyli, rekursiiviset funktiot, jne.

Algoritmien kuvauksen kannalta tärkeän *Churchin teesin* mukaan kuitenkin:
kaikki edellä mainitut algoritmikäsitteen formalisoinnit ovat yhtä voimakkaita.

Hyväksytään analysointia varten seuraava ”invarianssiperiaate”: minkä tahansa algoritmin ”järkevät” toteutukset poikkeavat aikavaativuudeltaan enintään vakiotekijän verran. Tähän periaatteeseen nojaten jätetään algoritmianalyysissä usein huomiotta vaativuusfunktioiden vakiokertoimet ja tyydytään analysoimaan vain menetelmän tehokkuuden kertaluokkaa ja muistetaan "kätkettyjen vakioiden" olemassaolo.

Mikä on tapauksen koko?

Yllä todettiin, että algoritmin suoritusaika yleensä kasvaa syöttötiedon koon kasvaessa. Täten algoritmin tehokkuus on jonkin syöttötiedon koon ilmaisevan parametrin, *n*, funktio. Tällainen syöttötiedon koko on tavallisesti helposti valittavissa oleva parametri.

Laskennallisen ongelman esiintymän *x* syöttötiedon koolla tarkoitetaan $|x| = x$:n ”tiiviin” esityksen vaatimaa talletustilaa = ratkaisualgoritmin syöttötiedon pituus. Syöttötiedon pituutta $|x|$ merkitään usein kirjaimella *n*.

Teoreettisesti: bittejä.
Käytännössä: jokin syötteen kokoa ilmaiseva luonteva suure.

Tyypillisiä esimerkkejä:
järjestäminen: *n* = järjestettävien alkioiden määrä,
verkko-ongelmat: *n* = verkon solmujen tai kaarien määrä tai jopa molemmat,
matriisilaskenta: *n* = matriisin suurin dimensio tai matriisin alkioiden lukumäärä.

Millaista aikaa analysoidaan?

Yllä tuli esiin jo kolme eri mahdollisuutta: paras mahdollinen suoritusaika, huonoin mahdollinen suoritusaika tai keskimääräinen suoritusaika.

Näistä ensimmäinen ei yleensä ole juurikaan kiinnostava. Hitain eli pahin mahdollinen suoritusaika on tavallisin ja monesti myös tarpeellinen. Lisäksi se on usein suhteellisen helposti laskettavissa. Keskimääräinen suoritusaika on usein kiinnostavin, mutta sen ongelmana on syöttötiedon jakauman selvittäminen ja lisäksi sen laskenta voi tuottaa hankaluuksia. Joskus voidaan myös suorittaa niin sanottu tasoitettu analyysi; tämän ideana on, että "pitkissä" toimenpidejonoissa niin sanotut pahimmat tapaukset voivat olla erittäin harvinaisia ja tämän vuoksi kustannuksia jaetaan niiden esiintymiskertojen mukaisessa suhteessa.

1.3 Funktion kertaluokka

Algoritmeja vertailtaessa pyritään usein ennustamaan pahimman tapauksen suoritusajan kasvua, kun *n* eli järjestettävien alkioiden lukumäärä kasvaa. Olkoon pahimman tapauksen suoritusaika vaikkapa muotoa $an^2 + bn + c$ tietyillä vakioilla *a* (> 0), *b* ja *c*. Silloin suurilla *n*:n arvoilla termi an^2 on huomattavasti suurempi kuin loput lausekkeen termit $bn + c$ ja täten suoritusaikaa voidaankin arvioida pelkästään termillä an^2. Edelleen voidaan havaita, että jos alkioiden lukumäärä *n* kasvaa kaksinkertaiseksi niin väliinsijoitusmenetelmän pahimman tapauksen suoritusaika tulee noin nelinkertaiseksi. Tehdäänkin lisäksi vielä yksinkertaistus, että meitä kiinnostaa ainoastaan suoritusajan *funktion kasvunopeus*. Silloin riittää siis tietää funktion korkeimman potenssin termi. Myös tämän korkeimman potenssin termin vakiokerroin jätetään silloin pois, koska se on vähemmän merkitsevä funktion kasvunopeuden yhteydessä. Tällöin kiinnostuksen kohteena on vain se mitä suuruusluokkaa, kertaluokkaa, tämä kasvunopeus on, minkä potenssin mukaan

suoritusaika kasvaa. Esimerkiksi alkion haulla *n*:n alkion taulukosta pahimman tapauksen suoritusajan kasvunopeus on siis kertaluokkaa *n*.

Kertaluokka antaa varsin yksinkertaisessa muodossa kuvan algoritmin tehokkuudesta. Sen mukaan voidaan helposti verrata algoritmien suhteellisia tehokkuuksia. Kun syöttötiedon koko, esimerkiksi hakuongelmassa taulukossa olevien alkioiden lukumäärä, kasvaa tarpeeksi suureksi, niin tämä suoritusajan funktion kertaluokka tulee merkittäväksi tekijäksi. Tällöin on kyseessä niin sanottu algoritmin *asymptoottinen tehokkuus*. Tavallisesti algoritmi, joka on asymptoottisesti tehokkaampi, on hyvä valinta ehkä aivan pieniä syöttötiedon kokoja lukuun ottamatta.

Olkoon meillä kaksi ei-negatiivista parametrin *n* funktiota $t_A(n)$ ja $t_B(n)$. Jos

$$\lim_{n\to\infty} \frac{t_A(n)}{t_B(n)} = 0$$

Niin funktio $t_B(n)$ kasvaa nopeammin kuin $t_A(n)$ ja silloin sanotaan, että $t_A(n)$ on asymptoottisesti pienempi kuin $t_B(n)$. Vastaavasti tällöin $t_B(n)$ on asymptoottisesti suurempi kuin $t_A(n)$. Jos tämä raja-arvo on nollasta eroava vakio, niin $t_A(n)$ ja $t_B(n)$ ovat asymptoottisesti yhtä suuria eli funktioiden kertaluokat ovat samoja.

Nyt tarvitsemme merkintätavan kuvaamaan kasvunopeutta.

Käytetään $O(g(n))$ merkintää kaikista niistä funktioista, jotka ovat asymptoottisesti pienempiä tai yhtä suuria kuin funktio $g(n)$. Tässä yhteydessä käytetään yleisesti =-merkintää ja esimerkiksi $10n + 8 = O(n)$ eli $g(n) = n$. Vastaavasti $4n^2 + 7n + 35 = O(n^2)$ eli $g(n) = n^2$. Myös $5n + 8 = O(n^2)$ eli tässäkin $g(n) = n^2$.

Esitetyn mukaan funktiot voidaan nyt asettaa asymptoottiseen suuruusjärjestykseen. Tämä voidaan tehdä, kuvaavat funktiot sitten suoritusaikaa, muistinkäyttöä tai mitä resurssia tahansa.

Annetaan seuraavassa tarkempi, formaali määrittely *O* merkinnälle. Funktioiden määrittelyalueena on luonnollisten lukujen joukko eli $N = 0, 1, 2, \ldots$. Tämä sopii hyvin algoritmien suoritusajalle ja myös muistinkäytölle, koska syöttötiedon koko *n* on tällöin ei-negatiivinen kokonaisluku.

Tavallisimmin käytetty on Big-O (iso O) -merkintä, $f(n) = O(g(n))$ ja tällöin tarkoitetaan, että $f(n)$ on asymptoottisesti pienempi tai yhtä suuri kuin $g(n)$ eli saadaan asymptoottinen yläraja $g(n)$. Tarkka määrittely on seuraava; funktiolle $g(n)$ merkinnällä $O(g(n))$ tarkoitetaan funktioita $\{f(n)$, joilla $\exists$ positiiviset vakiot c ja n_0 siten, että $0 \le f(n) \le cg(n)$ kaikilla arvoilla $n \ge n_0\}$.

Tällöin sanotaan, että funktion $f(n)$ vaativuus tai kompleksisuus on $O(g(n))$ ja kirjoitetaan $f(n) = O(g(n))$.

Esimerkiksi funktiolle $f(n) = 100n^2 + 30n + 45$ on voimassa, että $f(n) = O(n^2)$, mutta ei ole voimassa $f(n) = O(n)$ eli $f(n) \neq O(n)$.

Esimerkkejä:

$n^2 = O(n^3)$ (oikeastaan siis tulisi käyttää merkintää $n^2 \in O(n^3)$).
$12n^3 + 4n^2 = O(n^3)$.
$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0 = O(n^k)$, kun $k > 0$ ja $a_k > 0$.
$\log_a n = O(\log_b n)$ kaikilla $a, b > 1$.
$\log_2 (n!) = \sum_{i=1}^{n} \log_2 i = O(n \log_2 n)$.
$\sum_{i=0}^{n} \frac{1}{2^i} = O(1)$ eli vakio.

Ominaisuuksia:

Jos $f(n) = O(g(n))$ ja $g(n) = O(h(n))$ niin $f(n) = O(h(n))$.
$O(cf(n)) = O(f(n))$ kaikilla vakioilla $c > 0$.
Jos $f_1(n) = O(g_1(n))$ ja $f_2(n) = O(g_2(n))$ niin $f_1(n)+f_2(n) = O(\max(g_1(n), g_2(n)))$.

Jos algoritmi vaatii suurin piirtein saman ajan riippumatta probleeman koosta, niin se vie vakioajan. Silloin algoritmin suoritusajan vaativuus on vakio ja siitä käytetään merkitään $O(1)$. Jos algoritmin pahimman tapauksen suoritusajan vaativuus on $an^2 + bn + c$, niin vaativuus on $\mathrm{O}(n^2)$. Huomattava, että myös lineaarinen funktio eli esimerkiksi $3n + 15 = \mathrm{O}(n^2)$. Tarkoituksena on tietysti löytää aina mahdollisimman tiukka ylärajafunktio ja tässä tapauksessa tulisi kirjoittaa, että $3n + 15 = O(n)$.

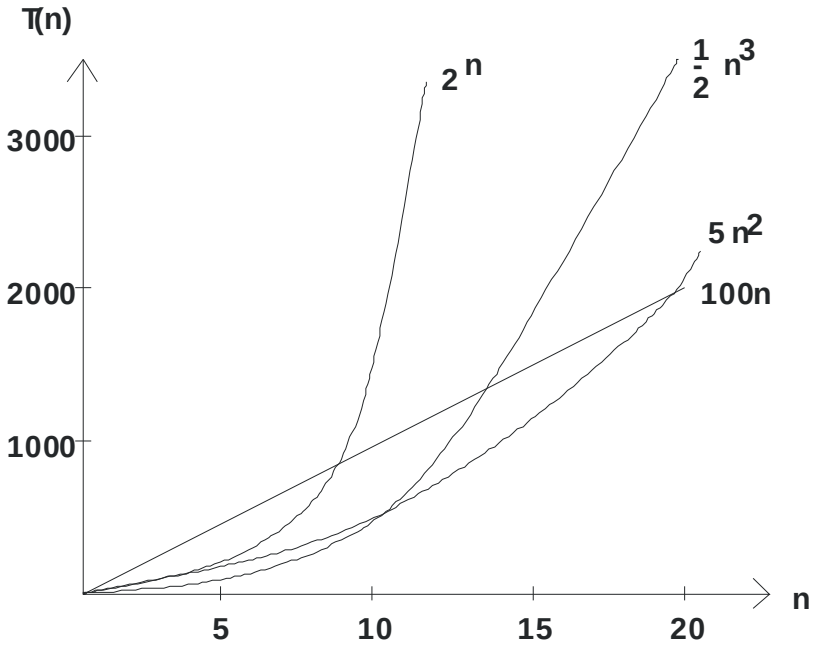
On olemassa myös muita kertaluokkafunktioiden merkintöjä, asymptoottinen alaraja $\Omega(f(n))$, tarkka raja $\Theta(f(n))$ aito yläraja $o(f(n))$ ja aito alaraja $\omega(f(n))$. Tällä opintojaksolla käytetään kuitenkin vain O-kertaluokkafunktiota.

Huom. Usein eri kirjoissa käytetään hiukan erilaisia määritelmiä ja kannattaa aina tarkistaa kirjasta tarkoin millaista merkintää käytetään.

Asymptoottisten funktioiden käytöstä

Oletetaan, että jonkun ongelman ratkaisemiseksi on kaksi algoritmia; algoritmi A, jonka suoritusajan vaativuus pahimmassa tapauksessa on $T_A(n) = O(n)$ ja algoritmi B, jonka suoritusajan vaativuus pahimmassa tapauksessa on $T_B(n) = O(n^2)$. Jos pahimman tapauksen suoritusajan vaativuus on ratkaiseva algoritmin valinnassa, niin asymptoottisen analyysin mukaan algoritmi A on parempi, vaikka pienillä syöttötiedon koon n arvoilla algoritmi B voikin olla nopeampi.

Seuraavassa on annettu neljän algoritmin suoritusaikojen funktiot, tässä poikkeuksellisesti siis tarkat funktiot, ja niiden kuvaajat. Nähdään, että pienillä n:n arvoilla funktioiden arvot ei juuri eroa toisistaan. Suuremmilla n:n arvoilla erot eri funktioiden arvojen välillä kasvavat hyvin suuriksi ja mitä suurempi n on, niin sitä merkittävämpi on funktion kertaluokka.



Tarkastellaan vielä näitä neljää algoritmia. Alla on kuvattu mikä on suurin ongelma joka kullakin algoritmilla voidaan ratkaista tietyn ajan kuluessa. Nähdään, että ensimmäisellä algoritmilla ratkaisuajan kasvaessa kymmenkertaiseksi saadaan siitä täysi hyöty eli myös suurin sallittu ongelman koko kasvaa kymmenkertaiseksi. Sen sijaan eksponentiaalisen ratkaisuajan algoritmilla, aikafunktio 2^n, suurimman sallitun probleeman koko kasvaa vain vakiolla 3.

	max *n*		
T(n)	10^3	10^4	lisäys
100n	10	100	× 10
$5n^2$	14	45	× 3.2
$(½)n^3$	12	27	× 2.3
2^n	10	13	+3

Onkin ratkaisevan tärkeää, onko algoritmin aikavaativuusfunktio polynomisesti rajoitettu vai ylipolynominen. Ylipolynomisia algoritmeja käytettäessä ei pidemmän ajoajan salliminen tai laitteiston tehostaminen juuri lisää ratkaisumahdollisuuksia. Sen sijaan tehokkaamman algoritmin löytäminen auttaa huomattavasti ongelman ratkaistavuuden suhteen.

Peruskertaluokkia

Suoritusajan kasvunopeutta voidaan siis käyttää kuvaamaan kuinka algoritmin vaativuus muuttuu syöttötiedon koon muuttuessa. Se on käyttökelpoinen myös algoritmien vertailussa. Aikaisemman mukaan, koska saman ongelman ratkaisuun voidaan käyttää algoritmia *A*, jonka pahimman tapauksen aikavaativuus on $O(n)$ ja algoritmia *B*, jonka pahimman tapauksen aikavaativuus on $O(n^2)$, niin algoritmi *A* on parempi pahimman tapauksen mukaan kuin algoritmi *B* riittävän suurella syöttötiedon koon arvolla *n*. Tässä tulee olla varovainen sen suhteen mitä riittävän suurella tarkoitetaan.

Seuraavassa taulukossa on esitetty kuuden eri funktion arvoja muutamilla pienillä muuttujan *n* arvoilla.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296
6	64	384	4096	262144	$\approx 1.8*10^{19}$

Tästäkin taulukosta voidaan havaita eri suuruusluokan funktioiden erottuvan nopeasti toisistaan. Jos jokin algoritmi on aikavaativuudeltaan esimerkiksi eksponentiaalinen, niin sen voidaan odottaa soveltuvan vain syöttötiedon koon suhteen varsin pienten ongelmien ratkaisuun.

Muutamat funktioiden suuruusluokat ovat erittäin tavallisia algoritmien analysoinnin yhteydessä. Seuraavassa on luetteloitu tavallisimpia suuruusluokkafunktioita:

1	Tämä vaatii siis *vakioajan* eli on riippumaton syötön koosta.
$\log n$	Tämä, *logaritminen*, on usein tuloksena, kun ongelma voidaan jakaa pienempään osaongelmaan pienentäen ongelman kokoa murto-osaan alkuperäisestä. Logaritmin kantaluku vaikuttaa vain termin vakiokertoimeen, joten kantaluvulla ei ole vaikutusta suuruusluokkaan.
n	Tällöin yleensä jokin operaatio tehdään kaikille alkioille. Tämä *lineaarinen* vaativuus on optimaalinen tapauksissa, joissa kaikki syöttötieto käsiteltävä.
$n \log n$	Tämä on yleensä tuloksena, kun ongelma jaetaan tasaisesti osiin, ratkaistaan kukin erikseen ja lopuksi vielä kootaan osaratkaisuista lopullinen tulos.
n^2	Tämä *neliöllinen* vaativuus on käyttökelpoinen vielä melko suuren syöttötiedon koon omaaville ongelmille. Tällöin yleensä algoritmi käsittelee ongelman kaikki alkioparit.
n^3	Tämä *kuutiollinen* vaativuus sopii vain suhteellisen pienen syöttötiedon koon omaaville ongelmille.
n^k	Tässä *k* on vakio ja tämä tarkoittaa yleisesti *polynomista* vaativuutta.
2^n	Tällä tarkoitetaan *eksponentiaalista* vaativuutta. Nämä ovat usein hyvin paljon resursseja vaativia ja melko tavallisia käytännön ongelmiin.

Nämä kaksi viimeistä suuruusluokkaa ovat teorian kannalta varsin merkittäviä. Hyvin yleisesti sanotaan, että jos ongelmalle löytyy algoritmi, jonka pahimman tapauksen suoritusaika on polynominen, niin kyseinen ongelma on (hyvin) ratkeava. Jos taas ongelmalle ei löydy tällaista polynomista algoritmia eli löydetään vain eksponentiaalisen suoritusajan algoritmeja, niin ongelmaa sanotaan ratkeamattomaksi tai huonosti ratkaistavissa olevaksi. On olemassa joukko ongelmia, joille ei ole löydetty lainkaan polynomisia algoritmeja. Eräs tällainen ongelma on jo aikaisemmin mainittu Hamiltonin kehäongelma, kuten myös tunnettu kauppamatkustajan ongelma.

Suuri joukko huonosti ratkeavia ongelmia, joille tunnetaan vain eksponentiaalisen aikavaativuuden algoritmeja, on osoitettu kuuluvan yhteiseen ongelmien joukkoon; NP (nondeterministic polynomial-time) -täydellisten ongelmien joukkoon. Millekään näistä NP-täydellisen joukon ongelmista ei ole löydetty polynomisen aikavaativuuden ratkaisualgoritmia. Jos yhdellekin tämän joukon ongelmista löydetään polynomisen ajan algoritmi, niin kaikki tämän joukon ongelmat ovat silloin ratkaistavissa polynomisen ajan algoritmilla. Mutta toisaalta kukaan ei ole osoittanut, että tällaista polynomista algoritmia ei voisi olla olemassa. Ongelmat, joille on löydetty jokin polynominen algoritmi, muodostavat polynomisessa ajassa ratkeavien ongelmien joukon, jota merkitään P:llä. Teoreettisen tietojenkäsittelyn suuri kysymys onkin, että onko P = NP.

1.4 Algoritmien analysointia

Ei-rekursiivinen algoritmi

Ohjelmaa tai algoritmia analysoitaessa määritellään ensin ohjelman yksittäisten käskyjen tai algoritmien yksittäisten askelien asymptoottinen vaativuus. Tämän jälkeen ohjelman tai algoritmin vaativuus saadaan yhdistämällä peräkkäin suoritettavien käskyjen tai askelien vaativuudet huomioiden silmukat ja muut rakenteet. Ohjelmaa ja algoritmia analysoitaessa voidaan käyttää seuraavia sääntöjä, tietysti huomioiden perusoperaatiot:

- Sijoituslause vie yleensä vakioajan *O(1)*.
- Jonoon peräkkäisiä lauseita käytetään aikaisemmin esitettyä summasääntöä.
- if-lauseen aika on ehtolausekkeen vaatima suoritusaika + ehdollisesti suoritetun lauseen maksimisuoritusaika.
- Silmukan suoritusaika on silmukkalauseen lausekkeiden vaatima suoritusaika + silmukan rungon suoritusaika summattuna yli kaikkien silmukan suorituskertojen.
- Aliohjelmakutsujen suoritusaikaa varten on ensin selvitettävä aliohjelman vaatima suoritusaika. Jos aliohjelma ei ole rekursiivinen niin tulee etsiä ensin sellainen aliohjelma, joka ei enää sisällä aliohjelmakutsuja. Lasketaan sen suoritusaika ja sen jälkeen voidaan laskea sellaisen aliohjelman suoritusaika, joka sisältää vain sellaisten aliohjelmien kutsuja, joiden suoritusaika on jo selvitetty. Näin jatkamalla saadaan aliohjelmakutsujen suoritusajat selvitettyä. Rekursiiviset aliohjelmat johtavat useimmiten seuraavan esimerkin mukaisesti rekursiivisiin suoritusaikafunktioihin.

Rekursiivinen algoritmi

Rekursio on yksi vahvimmista ongelman ratkaisumenetelmistä. Sen perustana on se, että pienempi ongelma on helpommin ratkaistavissa kuin suurempi. Kun ongelma on tarpeeksi pieni, niin sen ratkaisu on erittäin helposti muodostettavissa. Tällainen ratkaisutapa sopii usealle ongelmalle ja monet ongelmat ovat joko luonnostaan rekursiivisia tai ne voidaan kuvata rekursiivisesti. Tällöin niiden ratkaisemiseen voidaan etsiä algoritmia vaikkapa osittamisperiaatteen (hajoita-ja-hallitse, divide-and-conquer) mukaisesti. Siinä probleema jaetaan pienempiin vastaavanlaisiin

osaongelmiin, jotka ratkaistaan rekursiivisesti ja tämän jälkeen yhdistetään näiden osaongelmien ratkaisuista alkuperäisen ongelman ratkaisu.

Useat rekursiiviset algoritmit perustuvat seuraaville periaatteille

- Mahdollisesti tarvittavat globaalit muuttujat alustetaan. Sitten suoritetaan rekursiivisen algoritmin aloituskutsu.
- Algoritmissa tarkistetaan ensin, että kyseessä ei ole pieni yksinkertainen ongelma jäljellä. Jos näin on, niin ratkaistaan se ja palautetaan tämän antama ratkaisu.
- Muutoin tehdään tämänhetkisestä ongelmasta yksi tai useampia pienempiä vastaavanlaisia ongelmia ja ratkaistaan ne rekursiivisten kutsujen avulla. Ratkaisut pannaan talteen.
- Muodostetaan näiden pienempien ongelmien ratkaisuista tämän kutsun ongelman ratkaisu.

Järjestämisongelma voidaan ratkaista myös tällä periaatteella ja näin tehdään lomitusmenetelmässä. Periaate on silloin seuraava:

- Oletetaan, että järjestettävien alkioiden lukumäärä $n = 2^k$.
- Jaetaan järjestettävät n alkiota kahteen $n/2$ alkion osajoukkoon.
- Järjestetään molempien osajoukkojen alkiot rekursiivisesti siis käyttäen tätä samaa menetelmää.
- Yhdistetään kahden järjestetyn osajoukon alkiot lomittamalla, jolloin tuloksena saadaan alkuperäisten alkioiden järjestetty joukko.
- Rekursiivinen alkioiden järjestäminen voidaan lopettaa, kun osajoukossa on vain yksi alkio.

Kokoamalla esitetyt periaatteet sopivasti saadaan seuraavanlainen algoritmi:

```
// Lomitusmenetelmä
Merge_sort(a, l, h)
{
  if (l < h) {
    k = (l+h)/2;
    Merge_sort(a, l, k);
    Merge_sort(a, k+1, h);
    Merge(a, l, k, h);
  }
}
```

Tässä *a* on järjestettävät alkiot sisältävä taulukko, *l* on järjestettävän taulukon osan ensimmäinen eli alimmainen indeksi ja *h* vastaavasti viimeinen eli ylimmäinen indeksi. `Merge` lomittaa taulukon järjestyksessä olevat osat *a[l…k]* ja *a[k+1…h]* järjestykseen taulukon alkioihin *a[l…h]*. Algoritmin toiminta selvinnee seuraavasta esimerkistä, joka järjestää taulukon *a* alkiot 0..7. Tässä on pystyviivalla kuvattu aina senhetkiset taulukon jakopaikat.

```
(20,   6,   12,   17,    14,   9,   23,   15)
(20,   6,   12,   17, |  14,   9,   23,   15)
(20,   6, | 12,   17, |  14,   9,   23,   15)
(20, | 6, | 12,   17, |  14,   9,   23,   15)
(6,   20, | 12, | 17, |  14,   9,   23,   15)
(6,   20, | 12,   17, |  14,   9,   23,   15)
(6,   12,   17,   20, |  14,   9,   23,   15)
(6,   12,   17,   20, |  14,   9, | 23,   15)
(6,   12,   17,   20, |  14, | 9, | 23,   15)
(6,   12,   17,   20, |   9,  14, | 23, | 15)
(6,   12,   17,   20, |   9,  14, | 15,   23)
(6,   12,   17,   20, |   9,  14,   15,   23)
(6,    9,   12,   14,    15,  17,   20,   23)
```

Rekursiivisten algoritmien suoritusajan vaativuutta laskettaessa käytetään yleensä rekursiivisia funktioita. Merkitään *T(n)* on algoritmin suoritusajan pahimman tapauksen vaativuus, kun syöttötiedon koko on *n* alkiota. If-lauseen suoritus vie vakioajan *b* ja samoin seuraavan rivin suoritus. Seuraavan rivin rekursiivisen kutsun suoritusajan vaativuus on $T(n/2)$ ja samoin toiselle rekursiivisella kutsulla seuraavalla rivillä. `Merge` eli lomitus voidaan tehdä lineaarisessa ajassa alkioiden lukumäärän suhteen eli ajassa *cn*. Täten koko algoritmin suoritusajan vaativuus voidaan kirjoittaa muodossa

$$T(n) = \begin{cases} b, & \text{jos } n = 1, \\ 2T(n/2) + cn, & \text{jos } n > 1. \end{cases}$$

Voidaan osoittaa, että *T(n) = O(n* log *n)*, missä $\log n$ tarkoittaa kaksikantaista logaritmia.

Tarpeeksi suurella syöttötiedon koolla *n* lomitusmenetelmä *O(n* log *n)* on huomattavasti nopeampi kuin jokin yksinkertainen järjestämismenetelmä, jonka pahimman tapauksen suoritusajan vaativuus on $O(n^2)$. Olkoon meillä tietokone, joka suorittaa 100 000 000 käskyä sekunnissa ja olkoon jokin yksinkertainen järjestämismenetelmä ohjelmoitu koodiksi, joka tarvitsee noin $2n^2$ tämän tietokoneen käskyä järjestämään *n* alkiota. Vastaavasti lomitusmenetelmän ohjelmakoodi vie $50n \log n$ tietokoneen käskyä. Tällöin miljoonan alkion järjestäminen vie yksinkertaisella menetelmällä 5.56 tuntia kun taas lomitusmenetelmä veisi 10 sekuntia.

Aikavaativuusanalyysin käytettävyydestä

Käytettäessä esitettyä pahimman tapauksen aikavaativuusanalyysia tulee huomioida seuraavat seikat:

1) Joidenkin (monien?) algoritmien käyttäytyminen on keskimäärin huomattavasti parempaa kuin pahimmassa tapauksessa (esim. lineaarisen optimoinnin simplex-algoritmi).

2) Algoritmia käytetään vain kerran tai pari. Tällöin ohjelman kehityskustannukset hallitsevat kokonaiskustannuksia. Helpoimmin toteutettava algoritmi voi olla paras valinta.
3) Algoritmia toteutetaan vain pienillä syöttötiedon koon arvoilla. Tällöin saattaa suoritusajan funktion termien kertoimilla olla suurempi vaikutus kuin funktion kasvunopeudella.
4) Tehokkain algoritmi voi olla niin monimutkainen, että sitä ei voida tai ei kannata toteuttaa.
5) Algoritmin muistitilan tarve voi olla niin suuri, että joudutaan käyttämään toissijaista muistia, joka taasen hidastaa algoritmia. Myös virtuaalimuistin käytön merkitys voi olla hämmästyttävän suuri.
6) Algoritmeilla voi olla joitain muita tärkeitä vaatimuksia. Esimerkiksi numeerisilla algoritmeilla saattaa numeerinen tarkkuus asettaa lisävaatimuksia.
7) Swap- ja cache-muistien käyttö voi tuottaa yllätyksiä.

Muistitilan analysointi

Muistitilan käyttöä voidaan analysoida vastaavasti. Muistitilaa tarvitaan ensinnäkin itse ohjelmakoodia varten. Toisaalta muistitilaa tarvitaan ohjelman tietoja varten; ohjelmassa esiintyy erilaista ohjelman kannalta vakiotietoa ja algoritmin käyttämää työtietoa. Tieto voi olla joko staattisesti tai dynaamisesti varattua. Lisäksi ohjelma käyttää muistia ohjelmointiympäristöä varten, sieltä se varaa muistia lähinnä aliohjelmien kutsujen yhteydessä.

Kokonaisuudessaan ohjelman käyttämä muistitila voidaan jakaa kahteen osaan. Kiinteä osa muistitilaa, jonka koko ei riipu lainkaan ongelman esiintymästä; se sisältää lähinnä ohjelmakoodin. Vaihteleva osa muistitilaa käsittää lähinnä esiintymän datan, ohjelman muuttujia ja esimerkiksi aliohjelmakutsujen vaatiman muistitilan. Se mikä eri algoritmien toteutuksissa kiinnostaa on erityisesti tämä vaihtuvan osan muistitila.

Muistitilan tarpeen laskenta on yleensä yksinkertaisempaa kuin suoritusajan laskenta. Tosin mukana on usein piilevää muistitilan käyttöä.

Algoritmin suoritusajan mittaaminen

Yllä esitetty matemaattinen analysointitapa eli suoritusajan vaativuuden laskemismenetelmä voi joskus olla riittämätön. Voidaan esimerkiksi löytää useampia algoritmeja, joiden vaativuus pahimmassa tapauksessa kuuluu samaan kertaluokkaan. Tällöin voidaan algoritmin valintaa varten suorittaa tarkempi aika-analyysi toteuttamalla algoritmi jollain ohjelmointikielellä. Ja implementoidulla algoritmilla suoritetaan empiirinen analyysi mittaamalla suoritusaikoja.

Implementoidulla algoritmilla voidaan helposti selvittää myös muuta kuin suoritusaikaa. On mahdollista laskea laskuria käyttämällä tiettyjen operaatioiden lukumääriä tai muistiviittausten lukumääriä. Voidaan myös selvittää ajallisesti ohjelman pullokaulapaikat, jonka jälkeen tehostaminen voidaan kohdistaa oikeisiin paikkoihin.

Suoritusaikaa mitattaessa tulee muistaa, että systeemin antamat kelloajat eivät kuvaa tarkasti pelkästään algoritmin suoritusaikaa. Täten joudutaan yleensä suorittamaan useita testiajoja ja laskemaan niistä esimerkiksi keskiarvo.

Itse testiajoja varten tulee muodostaa testiaineistot. On päätettävä minkä kokoisilla aineistoilla testejä suoritetaan eli mitä tulevat olemaan syöttötietojen koon *n* arvot. Kannattaa aloittaa melko pienillä *n*:n arvoilla ja kasvattaa sitä tarpeen mukaan. Kullakin *n*:n arvolla on muodostettava itse syöttötiedot, joiden tulisi noudattaa oletettavissa olevan syöttötiedon jakaumaa. Tässä voidaan käyttää historiatietoja ja jos sellaisia ei ole, niin usein käytetään apuna satunnaislukugeneraattorin avulla muodostettuja aineistoja. Se mahdollistaa helposti suurienkin syöttötietojen muodostamisen.

Saatujen tulosten perusteella pyritään muodostamaan suoritusajalle lauseke syöttötiedon koon funktiona. Näin voidaan saada funktioita, joissa on mukana myös erilaisia vakiokertoimia ja muitakin termejä kuin korkeimman potenssin termi. Tämän lausekkeen avulla voidaan myös ennustaa suoritusaikaa muilla syöttötiedon koon arvoilla. Tulostietojen esittäminen taulukkomuodossa on hyvin tavallista, mutta ehkä selkeämmin tuloksia voidaan nähdä graafisesta tulostuksesta.

Tärkein ero empiirisen analyysin ja matemaattisen analyysin kohdalla on se, että matemaattinen menetelmä on riippumaton tietyistä syöttötiedoista. Toisaalta pahimman ajan matemaattinen analyysi voi antaa huonon kuvan algoritmin keskimääräisestä käyttäytymisestä eikä anna minkäänlaista todellista kelloaika-arviota algoritmin suoritusajalle.

2 Perustietorakenteita

Algoritmeja toteutettaessa tulee ongelman tiedot ja algoritmin tarvitsemat tiedot tallettaa jonkinlaiseen tietorakenteeseen käyttöä varten. Samoin algoritmia analysoitaessa on tiedettävä tietojen talletustapa, koska se vaikuttaa suoritusaikaan ja muistitilan tarpeeseen. Tätä varten tässä tarkastellaan muutamia perustietorakenteita ja niiden eri talletusmuotojen käyttämiä algoritmeja. Tietorakenteita tarkastellaan abstrakteina tietotyyppeinä.

Abstrakti tietotyyppi

Tietorakenne pyritään määrittelemään abstraktina tietotyyppinä, jolloin siitä annetaan seuraavat ominaisuudet.

- tiedot eli mitä tietoalkioita rakenteessa on ja
- operaatiot eli mitä tiedoilla voidaan tehdä.

Abstraktin tietotyypin määrittely on riippumaton tiedon esitysmuodosta ja toteutuksessa käytettävästä ohjelmointikielestä. Tässä vaiheessa ei vielä välitetä siitä miten rakenne toteutetaan. Toteutuksia ja niiden yksityiskohtaisia ohjelmointimahdollisuuksia on monia, joista useimmat ovat huonoja. Tunnusomaista rakenteiden perusmäärittelyille ja toteutuksille on, että harvoin löytyy yksiselitteisesti parasta vaihtoehtoa. Miltei aina on tyydyttävä kompromissiin (nopeus, muistitila, toteutuksen yksinkertaisuus, käytön helppous).

2.1 Pino

Pino (stack, push down list, LIFO) on tietorakenne, missä alkio voidaan lisätä ensimmäiseksi, pinon päällimmäiseksi, ja poistossa vain pinon päällimmäinen eli viimeksi lisätty voidaan poistaa. Pinojen yhteydessä näistä operaatioista käytetään nimityksiä *push* (lisäys pinon huipulle) ja *pop* (pinon huippualkion poisto ja palautus). Joskus huippualkion palautus suoritetaan omana operaationaan, *top*, ja poisto omana operaationaan, *pop*. Usein on mukana myös operaatio *isEmpty*, jolla voidaan testata onko pino tyhjä ja operaatio *size*, joka palauttaa pinossa tällä hetkellä olevien alkioiden lukumäärän. Implementoinnista riippuen voidaan ehkä tehdä muitakin toimenpiteitä, mutta varsinaisesti pinoon kuuluvat vain edellä mainitut.

Pinolle on siis ominaista, että siitä voidaan poistaa vain se alkio, joka lisättiin siihen viimeksi. Tästä tulee nimitys LIFO, Last-In-First-Out. Yksinkertaisin esimerkki on pino kirjoja. Uusi kirja voidaan helposti lisätä vain pinon päällimmäiseksi. Samoin vain tämä voidaan poistaa helposti.

Yksinkertaisuudestaan huolimatta pino on keskeinen tietorakenne.

- Aliohjelmia kutsuttaessa paluuosoite kutsuvaan aliohjelmaan ja osa parametreista asetetaan pinoon. Samasta pinosta kutsuttu aliohjelma allokoi tilan automatic-tyyppisille muuttujille. Tämä menetelmä sopii myös rekursiivisille aliohjelmille, jolloin puhutaan usein rekursiopinosta.

- Kun suorittimessa tapahtuu keskeytys, osa koneen rekistereistä talletetaan käyttöjärjestelmän hallitsemaan pinoon. Jos keskeytystä palveleva ohjelma keskeytetään jonkin etuoikeutetun tapahtuman johdosta, tulee pinoon lisää uusi kerros.
- Yksinkertainen esimerkki pinon käytöstä. Olkoon merkkijono, missä on erilaisia sulkumerkkejä {, }, [,] ,(,) muiden merkkien lisäksi. On testattava, että sulkumerkit sulkeutuvat oikein. Silloin esimerkiksi seuraava järjestys on oikein: ()([()()]{()}). Seuraava taas on väärin:[()(]). Oikeantyyppiset sulkumerkkijonot muodostavat kontekstivapaan kielen. Seuraavalla algoritmilla tarkastetaan sulkumerkkien oikeellisuus käyttäen pinoa, jonka alkiot voivat olla vasempia sulkumerkkejä.

    1) Alustetaan pino tyhjäksi. Asetutaan merkkijonon alkuun.
    2) Jos on tultu merkkijonon loppuun ja pino on tyhjä, ovat sulkumerkit oikein ja lopetetaan.
    3) Otetaan merkkijonosta seuraava merkki `a`, jota käsitellään seuraavien vaihtoehtojen 4,5,6 mukaisesti.
    4) Jos `a` on jokin vasen sulku, niin `push(a)`.
    5) Jos `a` on jokin oikea sulku ja pinon ensimmäinen (`top()`) merkki on samaa tyyppiä oleva vasen sulku, niin `pop()`. Muussa tapauksessa sulkumerkit ovat väärin, jolloin lopetetaan.
    6) Jos `a` on muu merkki, ei tehdä mitään.
    7) Toistetaan kohdasta 2.

Alla on pinon tilanne edellä mainitussa algoritmissa, kun käsitellään merkkijonoa ()([()()]{()}). Merkki | edustaa pinon pohjaa. Pinon päällimmäinen merkki on vasemmalla ennen kaksoispistettä. Kaksoispiste ei kuulu pinoon. Kaksoispisteen jälkeen on vuorossa oleva merkki kun ollaan kohdassa 3.

```
| :(     |( :)    | :(     |( :[     |([ :(    |([( :)   |([ :(    |([( :)
|([ :]   |( :{    |({ :(   |({( :)   |({ :}    |( :)     | :
```

Pinon implementoinnista

Ohjelmointikielissä on nykyään valmiina pino talletusrakenne käytettävissä. Jos kuitenkin haluaa tai tarvitsee itse implementoida pino, niin tällöin on lähinnä kaksi vaihtoehtoa; taulukko tai dynaaminen muistin käyttö.

Pino talletetaan taulukkoon `p` siten, että pinon viimeinen alkio on aina positiossa 0 ja pinon alkioiden lukumäärän ilmoittaa muuttuja `n`. Silloin pinon ensimmäinen alkio on positiossa `p[n-1]`.

- Kun pinoon lisätään alkio, niin ensin varmistetaan, että `n` on pienempi kuin `PMAX`, joka on taulukon koko.
- Sitten ensimmäisen alkion tiedot jäljennetään positioon `n`, jonka jälkeen suoritetaan `n=n+1`.

- Kun pinosta poistetaan alkio, varmistetaan ensin, että `n>0` (pino ei ole tyhjä). Sitten suoritetaan `n=n-1` ja otetaan taulukon tietueesta `p[n]` tarvittavat tiedot.

Jos ohjelmassa käsitellään useita pinoja, niin jokaista pinoa kohti tarvitaan oma taulukko.

Taulukkoesityksen etuna on yksinkertaisuus. Haittana taas on se, että jos pinossa samanaikaisesti olevien alkioiden maksimimäärää ei tiedetä tarkasti, voi pinon sallittu koko ylittyä. Tätä haittaa voidaan pienentää seuraavilla tavoilla.

- Pyritään laskemaan ohjelman alkuarvojen perusteella yläraja pinon koolle. Itse taulukko varataan dynaamisesti.
- Varataan tilaa reilusti. Haittana on tietenkin mahdollinen muistitilan loppuminen.
- Kun taulukko tulee täyteen, laajennetaan sitä varaamalla uusi suurempi taulukko, jonne senhetkinen pino sitten kopioidaan. Tämän haittana on se, että laajennuksen yhteydessä on tilaa oltava sekä vanhalle että uudelle taulukolle yhtaikaa ennen kuin vanha taulukko jäljennetään uuteen. Siis nytkin voi muisti loppua kesken. Yhtä hyvin voitaisiin alun perin varata tilaa riittävästi.

Kaksi pinoa voidaan kätevästi tallettaa samaan taulukkoon siten, että ensimmäisen pinon pohja on taulukon alussa ja pino huippu kasvaa kohti taulukon loppupäätä. Toinen pino sijoitetaan vastaavasti siten, että sen pohja on taulukon lopussa ja pino kasvaa kohti taulukon alkupäätä.

Pino voidaan tallettaa myös linkittämällä dynaamiseen muistiin. Tällöin tulee linkitys hoitaa pinon huipulta pohjaa kohti.

Käytetään kumpaa talletustapaa tahansa niin pino-operaatiot ovat erittäin tehokkaita. Poisto, lisäys ja huippualkion arvon selvittäminen voidaan tehdä vakioajassa.

2.2 Jono

Jono (queue, FIFO) on tietorakenne, missä tavallisesti on operaatiot *lisää jonon viimeiseksi* (*enqueue*) ja *poista jonon alusta* (*dequeue*). Tässä voi olla samoin kuin pinossa erikseen operaatio, joka vain palauttaa jonon keula-alkion arvon. Lisäksi on usein operaatiot *size*, joka palauttaa jonossa olevien alkioiden lukumäärän ja *isEmpty*, joka palauttaa boolean arvon tosi, jos jono on tyhjä ja muutoin arvon epätosi. Jono on loogisesti samankaltainen kuin tavaratalon kassajono.

Alla on esimerkki kirjaimia sisältävän jonon toimintahistoriasta. Jonon alku on vasemmalla. Nuoli vasemmalla tarkoittaa, että jonosta on juuri poistettu nuolen osoittama merkki. Nuoli jonon peräpäässä kuvaa sitä, että jonoon ollaan juuri lisäämässä merkkiä. Jonon merkit ovat merkkien | ja : välissä, jotka eivät kuulu itse jonoon.

|: ←A |A: ←C A← |C: |C: ← F |CF: ←X C← |FX: F← |X: X← |:

Jonon yleistyksenä on kaksipäinen jono (dequeue), missä voidaan lisätä alkuun tai loppuun ja myös poistaa alusta tai lopusta.

Tietotekniikassa on seuraavia jonojen sovelluksia.

- Vuoroaan odottavien tehtävien jono moniajokäyttöjärjestelmässä.
- I/O-komentojen jono käyttöjärjestelmän levypalvelijassa.
- Tapahtumajonot (Event Queue) (hiiren ja näppäimistön painallukset) graafisten käyttäjäliittymien ohjausohjelmistoissa (Windows, XWindows).
- Todellisten jonojen mallittaminen simulointiohjelmistoissa.

Myös jono on nykyään suoraan käytettävissä useimmissa ohjelmointikielissä. Mutta jos toteuttaa sen itse niin toteutus on jälleen suhteellisen helppoa.

Taulukossa talletus olisi yksinkertaisinta siten, että taulukossa `q`, jonka pituus on `QMAX`, jonon ensimmäinen alkio olisi aina positiossa 0 ja jonon viimeinen alkio positiossa `n-1`. Tässä `n` on jonon alkioiden lukumäärä. Jonon ensimmäisen alkion poisto kuitenkin edellyttäisi taulukon kaikkien alkioiden siirtämistä eteenpäin. Tästä syystä ainoa järkevä jonon taulukkotalletusmuoto on taulukossa kiertävä jono. Periaatteena on, että kun jono on täyttynyt taulukon viimeiseen positioon saakka, asetetaan seuraava alkio taulukon positioon 0. Kun jonon ensimmäinen alkio poistetaan, siirtyy jonon ensimmäinen positio askeleen eteenpäin. Jos silloin taulukon loppu ohitetaan, siirtyy uusi jonon alkukohta positioon 0. Jotta voitaisiin testata koska jono on tyhjä tai täysi, talletetaan taulukkoon, jonka koko olkoon `QMAX`, enintään `Q1 = QMAX-1` alkiota. Jonon ensimmäisen alkion indeksi taulukossa olkoon `f` (front) ja viimeistä seuraavan paikan indeksi `b` (back). Indeksi `b` osoittaa aina tyhjää paikkaa. Toinen tapa tyhjän jonon tarkasteluun on käyttää laskurikenttää, jossa pidetään yllä koko ajan jonossa olevien alkioiden lukumäärää.

Kun `f = b`, on jono tyhjä. Jono on täysi, kun joko `f = 0` ja `b = Q1`, tai kun `f > 0` ja `f-b = 1`.

Jono-operaatioiden suoritus:

- Lisäys jonoon (enqueue)
  - Tarkistetaan, että jono ei ole täysi.
  - Jäljennetään uuden alkion tiedot positioon `b`.
  - Jos `b < Q1`, niin `b = b+1`. Jos `b = Q1`, niin asetetaan `b = 0`.
- Poisto jonosta (dequeue)
  - Tarkistetaan, että jono ei ole tyhjä.
  - Otetaan jonon ensimmäisestä alkiosta `f` tarvittavat tiedot.
  - Jos `f<Q1`, niin `f=f+1`. Jos `f = Q1`, niin asetetaan `f=0`.

Todetaan, että nyt kumpaankin operaatioon menee aika, joka on riippumaton jonon alkioiden lukumäärästä eli vakioaika. Jono voidaan implementoida myös käyttäen linkitystä dynaamisessa muistissa. Silloin linkitys hoidettava jonon keulasta kohti jono viimeistä alkiota. Tällöin myös operaatioiden suoritusaika on vakio. Oikeastaan ainoa

huono puoli jonon toteuttamisessa taulukossa on, että jonon maksimikapasiteetti tulee tuntea etukäteen.

2.3 Lista

Lista on erittäin monipuolinen ja yleinen tietorakenne. Se koostuu joukosta samantyyppisiä alkioita (tietueita). Seuraavia määrittelyjä ei ole yritetty tehdä matemaattisessa mielessä minimaalisiksi.

- Lista `l` voi olla tyhjä.
- Ei tyhjässä listassa on kaksi erikoisasemassa olevaa alkiota, ensimmäinen alkio `first(l)` ja viimeinen alkio `last(l)`. Jos listassa `l` on vain yksi alkio, niin `first(l) = last(l)`.
- Listan alkiot muodostavat täydellisesti järjestetyn joukon. Tätä järjestystä kutsutaan myös talletusjärjestykseksi. Silloin jokaisella alkiolla `a`, paitsi viimeisellä, on välitön seuraaja `succ(a)`. Samoin jokaisella alkiolla `a`, paitsi ensimmäisellä, on välitön edeltäjä `pred(a)`.
- Listan jokaiselle alkiolle voidaan antaa järjestysnumero. Niinpä alkion `first(l)` järjestysnumero on 0 (tai 1). Jos alkion `a` järjestysnumero on `i`, niin alkiolla `succ(a)` se on `i+1` ja alkiolla `pred(a)` se on `i-1`, edellyttäen, että vastaavat alkiot ovat olemassa.
- Listan ensimmäinen (ja mahdollisesti viimeinen) alkio ovat välittömästi käytettävissä.

Listan alkioilla on siis tietty rakenteellinen järjestysnumero ja tämä järjestys asettaa listan alkioille lineaarisen järjestyksen. Näin ollen puhutaan lineaarisista listoista.

Listoille suoritettavat operaatiot vaihtelevat tarpeen mukaan. Kuinka tehokkaita (vähän aikaa vieviä) eri operaatiot ovat, riippuu implementoinnista. Kaikkia operaatioita ei välttämättä tarvitsekaan käyttää. Seuraavassa on lueteltu tavallisimpia listaoperaatioita.

- Listan kulku eteenpäin. Olkoon listan alkio `a` käytettävissä. Se tarkoittaa, että siihen on referenssi. Silloin saadaan käytettäväksi sen seuraaja `succ(a)`. Jos `a` oli kuitenkin listan viimeinen, saadaan indikaatio siitä (esimerkiksi funktiolla `succ(a)` on jokin erikoisarvo).
- Listan kulku taaksepäin. Edellinen alkio on `pred(a)`. Jos `a` oli listan ensimmäinen, saadaan indikaatio siitä, että funktio ei ole määritelty.
- Alkion lisäys listaan. Se voidaan lisätä ensimmäiseksi, viimeiseksi tai johonkin muualle, jolloin paikan valintaan voi olla erilaisia perusteita. Oletetaan, että uusi alkio lisätään alkion `a`, jonka järjestysnumero on `i`, jälkeen. Silloin uuden alkion järjestysnumero on `i+1`.
- Alkion poisto listasta. Taaskin voidaan poistaa ensimmäinen, viimeinen tai jokin muu alkio. Lista ei tietenkään saa olla tyhjä. Jos poistettiin alkio `a`, vähenevät tätä alkiota seuraavien alkioiden järjestysnumerot yhdellä.
- Alkion käyttö (saanti, access). Tämä tarkoittaa sitä, että saadaan referenssi johonkin listan alkioon, jonka tietoja voidaan sitten käyttää tai muuttaa.

Alkiota ei kuitenkaan poisteta listasta. Erikoistapauksina ovat ensimmäisen ja viimeisen alkion saannit. Alkio voi olla annettu myös järjestysnumerollaan. Listan yleisen alkion saanti voi olla hidas, koska listan alkio saadaan esille vain edellisestä (tai mahdollisesti seuraavasta) alkiosta käsin.

- Listan läpikäynti etenevästi. Olkoot lista-alkiot etenevässä järjestyksessä $a_0, a_1, \ldots, a_{n-1}$. Läpikäynti tarkoittaa sitä, että suoritetaan jotkin toimenpiteet $f(a_0), f(a_1), \ldots, f(a_{n-1})$ listan alkioille tässä järjestyksessä. Vastaavasti voitaisiin määritellä listan läpikäynti takaperin. Periaatteessa listan läpikäynti suoritetaan seuraavasti:

    - Suoritetaan ensimmäisen alkion saanti. Tuloksena saadaan referenssi alkioon `first(l)`. Tämän avulla saadaan alkion tiedot käyttöön ja voidaan suorittaa toimenpide `f`. Jos lista on tyhjä, saadaan tuloksena `null` tms. indikaatio.
    - Sen jälkeen siirrytään vuorossa olevasta alkiosta `a` seuraavaan käyttäen funktiota `succ(a)`. Jos `a` oli listan viimeinen, on `succ(a) = null`.

- Listojen yhdistäminen. Silloin kahden listan l_1 ja l_2 alkiot viedään kolmanteen listaan l_3. Listat l_1 ja l_2 voivat hävitä toimenpiteen johdosta. Voidaan myös vaatia, että ne säilyvät ennallaan, jolloin tiedot kopioidaan uusiin tietueisiin. Erikoistapaus yhdistämisestä on katenointi. Silloin esimerkiksi lista l_2 voitaisiin liittää listan l_1 perään.
- Listan perustaminen ja alustaminen tyhjäksi. Listan tyhjennys. Muistitilan varaaminen listalle. Muistitilan vapauttaminen.

Edellä määritellyissä operaatioissa ei oteta mitään kantaa siihen miten lista on esitetty tietokoneessa. Esitystapa vaikuttaa siihen, miten nopeasti eri operaatiot voidaan suorittaa.

Voidaan havaita, että pino ja jono ovat erikoistapauksia listasta. Niissä operaatiot kohdistuvat vain määrättyihin kohtiin listaa.

Järjestyt listat

Oletetaan, että listan alkioille on määritelty järjestysrelaatio alkion jonkin kentän tietosisällön, järjestysavaimen, perusteella. Toisaalta listalla on sen alkioiden keskinäinen (rakenteellinen) järjestys, jonka määrittelee funktio `succ`. Oletetaan listan alkioina olevan pelkästään avainkentän arvoja. Olkoon `a` ja `b` listan alkioita, ja `b = succ(a)`. Jos aina $a \leq b$, niin lista on järjestetty. Järjestys voi tietysti olla myös laskeva.

Järjestysavain voi koostua myös useammasta komponentista. Silloin niitä vertaillaan yleensä leksikografisesti. Olkoot esimerkiksi tietueiden 1 ja 2 järjestysavaimina kolmikot (a_1, b_1, c_1) ja (a_2, b_2, c_2). Tietue 1 on pienempi kuin tietue 2, kun jokin seuraavista ehdoista on voimassa:

$a_1 < a_2$,
$a_1 = a_2$ ja $b_1 < b_2$ tai
$a_1 = a_2$ ja $b_1 = b_2$ ja $c_1 < c_2$.

Useamman komponentin määräämässä järjestyksessä ei kaikkien komponenttien järjestyksen tarvitse olla samansuuntainen. Esimerkiksi henkilöluettelo voitaisiin järjestään ensisijaisesti laskevaan ikäjärjestykseen. Samanikäiset järjestetään aakkosjärjestykseen suku- ja etunimien mukaan.

Jos järjestetystä listasta poistetaan alkioita, niin se pysyy edelleen järjestettynä. Sen sijaan uuden alkion lisäys on määriteltävä niin, että lisäyksen jälkeenkin lista on järjestetty. Näin on laita, kun uusi alkio `u` lisätään seuraavien ehtojen mukaisesti:

- Jos `u < first(l)`, niin `u` lisätään listan ensimmäiseksi.
- Jos `last(l)` $\leq$ `u`, niin `u` lisätään listan viimeiseksi.
- Muussa tapauksessa haetaan sellaiset alkiot `a` ja `b = succ(a)`, joilla `a` $\leq$ `u < b` . Alkio `u` lisätään listassa alkioiden `a` ja `b` väliin, jolloin siis lisäyksen jälkeen `u = succ(a)` ja `b = succ(u)`.

Jos on olemassa alkioita, joilla on voimassa yhtä suuruus `a = b`, ei lisäämispaikka ole yksikäsitteinen. Jos uusi alkio lisätään aina mahdollisimman lähelle järjestetyn listan loppua, on lisääminen *stabiili*. Poistettaessa alkioita valitaan taasen yhtä suurista alkioista ensimmäinen järjestetyn listan alusta lukien.

Eräs järjestetyille listoille luontainen operaatio on lomitus (merge). Järjestetyistä samanlaisia alkioita sisältävistä listoista l_1 ja l_2 muodostetaan järjestetty lista l_3.

Järjestettyjen listojen limitysalgoritmi

1) Alustetaan lista l_3 tyhjäksi. Aloitetaan kasvavasti järjestettyjen listojen l_1 ja l_2 alusta.
2) Toistetaan seuraavaa:
   - Jos jompi kumpi listoista l_1 tai l_2 on lopussa, jatka kohdasta 3.
   - Olkoot listojen l_1 ja l_2 tällä hetkellä ensimmäisinä olevat alkiot a_1 ja a_2. Siirretään pienempi niistä listan l_3 viimeiseksi.
   - Toistetaan kohdan 2 alusta.
3) Toinen listoista on lopussa. Mikäli toinen lista ei ole tyhjä, siirretään sen jäljellä oleva osa sellaisenaan listan l_3 jatkoksi.

Listan implementointi

On olemassa kaksi päävaihtoehtoa lineaarisen listan implementoinnissa; taulukon käyttö ja dynaaminen muistinvaraus.

Taulukon käyttö

Lista talletetaan esimerkiksi niin, että sen ensimmäinen tietue on positiossa 0 ja muut ovat siitä eteenpäin. Yhtä hyvin voitaisiin listan ensimmäinen alkio tallettaa taulukon

viimeiseen positioon ja seuraavat siitä taulukon alkupäätä kohti. On huomattava, että vain sillä on merkitystä, että listaoperaatiot voidaan suorittaa määritelmien mukaisesti.

Järjestetyssä listassa alkioiden järjestyksen listassa määrää alkioiden järjestysavaimen mukainen järjestys.

- Tietueen u lisääminen taulukkoon.
  - Lähtien taulukon loppupäästä alkuun päin haetaan ensimmäinen indeksi `i`, jolla `t[i-1]` $\leq$ `u`. Jos tällaista ei ole, `i = 0`.
  - Siirretään taulukon loppupäätä yhdellä askeleella eteenpäin alkaen positiosta `n-1` ja lopettaen positioon `i` (tämä mukaan luettuna). Uusi tietue tulee positioon `i`. Paikan haku ja siirto voidaan yhdistää.
  - Pahimman tapauksen aikavaativuus siis on *O(n)*.
- Poisto taulukosta. Olkoon poistettavan tietueen indeksi `i`. Silloin taulukon loppupäätä siirretään alkuun päin yhdellä askeleella alkaen positiosta `i+1` ja lopettaen positioon `n-1`. Pahimman tapauksen aikavaativuus siis on *O(n)*.
- Peräkkäishaku. Kuljettaessa taulukon alusta loppuun päin, voidaan haku lopettaa, kun vastaan tulee tietue, jonka avain on suurempi kuin hakuavain. Silloin haettua tietuetta ei löydy. Keskimäärin joudutaan käymään läpi puolet alkioista. Pahimman tapauksen vaativuus on *O(n)*. Peräkkäishaun vaativuus on sama riippumatta siitä löytyykö tietue vai ei.

Huomaa, että jos alkiot vain talletetaan taulukkoon ja siis ei pidetä yllä listaa niin alkioita voidaan siirrellä vapaasti. Silloin lisäys voidaan vakioajassa suorittaa listan viimeiseksi alkioksi, edellyttäen, että tilaa on. Tämä tietysti edellyttää, että lisäys todella suoritetaan ja ei tarvitse erikseen tutkia, onko sama alkio jo listassa, jolloin lisäystä kenties ei tehtäisikään. Samoin poisto voidaan tällöin suorittaa vakioajassa kopioimalla listan viimeinen alkio poistettavan alkion tilalle. Tämä edellyttää, että poistettavan alkion paikka on jo aikaisemmin selvitetty.

Binäärihaku

Binäärihaku eli puolitushaku perustuu siihen, että ensin tutkitaan onko haettava tietue järjestetyn taulukon alkupuolessa vai loppupuolessa vertaamalla hakuavainta taulukon keskimmäisen tietueen avaimeen. Sitten rajoitutaan siihen puoliskoon, missä haettava tietue on, mikäli sitä ylipäänsä on taulukossa. Jatketaan samalla tavalla tarkasteluvälin puolituksella, kunnes välin pituudeksi tulee 1 alkio.

Alla olevassa ohjelmanpätkässä indeksi `i` ja `j` rajoittavat sen taulukon osan, missä haettavan tietueen tulisi olla. Indeksi `k` on edellisten keskiarvo.

```
// Binäärihaku taulukosta t. Hakuavain a
// Järjestetyt tietueet taulukon alkioissa 0..n-1
i = -1; j = n;
while (j-i > 1) {
  k  = (i + j)/2;
  // Invariantti: t[i].key < a <= t[j].key
  if (a <= t[k].key)
    j = k;
```

```
  else
    i = k;
}
if (j < n and a == t[j].key) {  // Löytyi paikasta j
  // ... Toimenpiteet, kun löytyi …
} else {
  // ... Toimenpiteet, kun ei löytynyt ...
}
```

Jos hakukierrokset päättyvät tilanteeseen, missä `j = n`, on hakuavain suurempi kuin mikään taulukossa olevan tietueen avain. Jos hakuavain on pienempi kuin taulukon avaimet, on lopussa `i = -1` ja `j = 0`, jonka jälkeen todetaan, että `t[0]` $\neq$ `a`.

Binäärihakuohjelman oikeaksi todistaminen

while-silmukassa on koko ajan voimassa seuraava silmukkainvariantti eli algoritmin suorituksen aikana silmukan lopetusehtoa tutkittaessa oleva muuttumaton ominaisuus: hakuavain on suurempi kuin tietueen `t[i]` avain ja pienempi tai yhtä suuri kuin tietueen `t[j]` avain. Osoitetaan tämä.

- Aluksi asetetaan `i = -1` ja `j = n` (taulukon tietueiden lukumäärä). Voidaan ajatella, että näissä taulukon ulkopuolisissa positioissa avainten arvot ovat $-\infty$ ja $+\infty$. Näitä ei ohjelmassa kuitenkaan koskaan käytetä. Siis invariantti on tosi alussa.
- Oletetaan, että invariantti on tosi **while**-silmukan alussa. Silloin siis indeksien erotus on ainakin 2, joten niiden keskiarvo `k` ei ole sama kuin `i` tai `j`. Jos hakuavain `a` $\leq$ `t[k].key`, on uusi yläraja `j = k`. Muussa tapauksessa uusi alaraja `i = k`. Siis silmukan lopussa invariantti on edelleen tosi. Tämä pitää paikkansa myös viimeisellä kierroksella.
- Kun lopuksi **while**-silmukka lopetetaan, on silloin `j = i + 1` ja `t[i].key < a` $\leq$ `t[j].key`. Jos siis avaimen `a` omaavia tietueita ylipäänsä on taulukossa, on ensimmäinen niistä positiossa `j`.

Binäärihaun vaativuus

Lause. Olkoon järjestetyssä taulukossa n tietuetta. Silloin binäärihakuun tarvittavien hakuavaimen ja tietueavainten järjestysvertailujen määrä on joko $v(n) + 1$ tai $v(n) + 2$, missä $v(n) = \lfloor \log_2 n \rfloor$.

Tod. Todetaan ensiksikin, että kun taulukon pituus kasvaa, niin tarvittavien vertailujen määrän suurin mahdollinen arvo pysyy ennallaan tai kasvaa.

Olkoon taulukon pituus $n = 2^k - 1$. Kun aloitetaan **while**-silmukkaa (ennen lopetustestiä) sen jälkeen, kun on suoritettu h kierrosta, todistetaan, että indeksi i muotoa $m2^{k-h} - 1$ $(m \geq 0)$ ja $j = (m + 1)2^{k-h} - 1$. Todetaan, että näin laita alussa (0 kierrosta), jolloin $i = 0 \cdot 2^k - 1 = -1$ ja $j = 1 \cdot 2^k - 1$.

Induktio-oletus: Indeksit *i* ja *j* ovat olleet mainittua tyyppiä kun on suoritettu *h* kierrosta. Määrätään indeksit *h+1* kierroksen jälkeen. Keskiarvo $\lfloor (i + j)/2 \rfloor = (2m + 1)2^{k-h-1}$. Sen jälkeen on joko $i = 2m\,2^{k-h-1} - 1$ ja $j = (2m + 1)2^{k-h-1} - 1$ tai $i = (2m + 1)2^{k-h-1} - 1$ ja $j = (2m + 2)2^{k-h-1} - 1 = n$.

Kun on suoritettu viimeinen kierros, ovat indeksit muotoa $m2^0 - 1$ ja $(m + 1)2^0 - 1$. Siis kierroksia suoritetaan *k*. Kun taulukon koko *n* kasvaa arvosta $2^k - 1$ arvoon $2^{k+1} - 1$, kasvaa kierrosten määrä arvosta *k* arvoon *k+1*. Sillä välillä kierroksia tarvitaan *k* tai *k+1* riippuen haettavan alkion paikasta. Kun *n* on välillä $[2^k, 2^{k+1} - 1]$, on kierrosmäärä välillä $[k, k + 1] = [v(n), v(n) + 1]$. Jokaisella kierroksella suoritetaan yksi vertailu. Lopuksi vielä tutkitaan hakuavaimen ja tietueen *j* avaimen yhtä suuruus.□

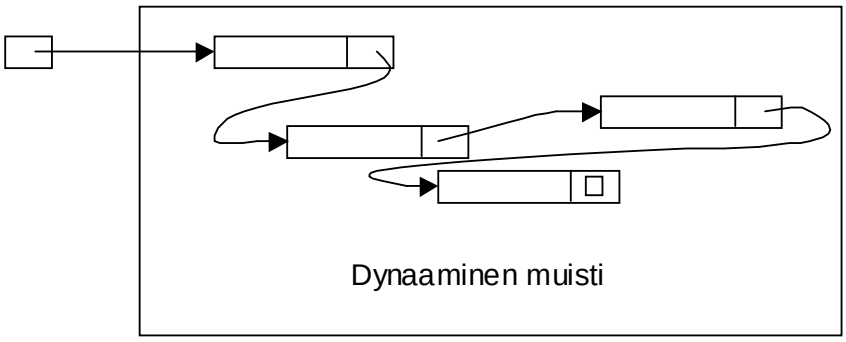
Jos listaan kohdistuu paljon hakuoperaatioita, niin listan järjestämisestä saadaan selvää hyötyä. Silloin voidaan siis käyttää binäärihakua. Olkoon taulukon pituus 50. Binäärihaussa tarvitaan silloin enintään 7 vertailua ja peräkkäishaussa keskimäärin 25. Jos taulukon pituus on 1000, binäärihaku tarvitsee enintään 11 vertailua, mutta peräkkäishaku keskimäärin peräti 500.

Dynaamisen muistin käyttö eli linkitys

Kun lineaarinen lista talletettiin taulukkoon, oli eräänä haittana uusien tietueiden lisäämisen ja poistamisen yhteydessä suoritettavat siirrot. Samoin listan koon arviointi etukäteen saattaa olla hankalaa tai jopa mahdotonta. Linkitettyjen rakenteiden käyttö vähentää näitä ongelmia.

Periaatteena on, että listan tietueet eivät välttämättä ole fyysisesti vierekkäin, vaan ne voivat olla missä tahansa niille varatussa alueessa, dynaamisen muistin alueessa. Jokaisessa tietueessa on referenssi, osoitin, linkki, joka ilmoittaa listan seuraavan alkion paikan. Listan viimeisellä tietueella tämän referenssin arvo on esimerkiksi `null`.

Jokaisella listalla on listareferenssi, joka ilmoittaa listan ensimmäisen alkion paikan. Tätä kutsutaan yleensä listanpääksi. Yleensä graafisissa esityksissä referenssit, osoittimet, linkit, pointterit jne., kuvataan nuolilla, joiden kärkipää näyttää mitä referenssi osoittaa.



Listan alkioille varataan siis tilaa dynaamisesta muistista. Dynaamisen muistin alueella voi samanaikaisesti olla useampiakin listoja. Käytettäessä dynaamista muistia tulee listoista poistettujen tietueiden muistilohkot tarvittaessa vapauttaa uudelleen käytettäviksi. Listojen koko voi vaihdella suuresti ohjelman kuluessa. Tietenkin koneen ja käyttöjärjestelmän sallima kokonaismuistitila on aina rajana.

Tietueen lisääminen linkitettyyn listaan

Olkoon referenssi listan ensimmäiseen alkioon `plista`. Uutta tietuetta osoittaa referenssi `pnew`.

Ennen kuin listaan viedään uusi tietue, on sille allokoitava tila dynaamisesta muistista, ja yleensä tietueeseen asetetaan muut tiedot paitsi seuraajareferenssi. Itse lisäyksessä on tilanteesta riippuen huolellisesti hoidettava kolme eri tapausta.

- Lisääminen listan alkuun. Referenssin `pnew` osoittama tietue lisätään listan ensimmäiseksi seuraavilla lauseilla. Lista voi olla myös tyhjä (`plista = null`).

  ```
  pnew.next = plista;
  plista = pnew;
  ```

- Olkoon tietue (referenssi `pnew`) lisättävä listan keskelle referenssin `pj` antaman tietueen jälkeen (`pj` voi olla myös viimeinen).

  ```
  pnew.next = pj.next;
  pj.next = pnew;
  ```

- Olkoon `pviim` referenssi listan viimeiseen alkioon. Uusi tietue lisätään listan viimeiseksi seuraavilla käskyillä (oletetaan, että lista ei ole tyhjä).

  ```
  pviim.next = pnew;
  pnew.next = null;
  pviim = pnew;
  ```

Linkityn listan kulku

Esimerkki listan kaikkien alkioiden läpikäynnistä.

```
for (h = plista; h != null; h = h.next)
  tulosta(h.nimi(), h.svuosi());
```

Tietueiden poisto

Tavallisesti tietorakenteesta poistettava alkio palautetaan, joten sijoitetaan referenssi poistettavaan alkioon, `pois`.

- Ensimmäisen tietueen poisto listasta. Olkoon `plista` referenssi listan ensimmäiseen tietueeseen. Ensimmäinen tietue voidaan poistaa seuraavasti

```
pois = plista;
plista= plista.next;

// Tehdään poistetulla tietueella mahdollisesti jotain
// ja tuhotaan se eli delete pois tai pois = null
```

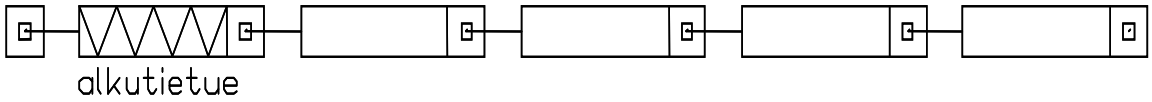
- Mielivaltaisen tietueen poisto listasta (referenssi ensimmäiseen tietueeseen on `plista`). Ensin on poistettava tietue haettava tai itse asiassa poistettavan tietueen edeltäjä. Tämä voidaan tehdä joko yhden referenssinmuuttujan tai kahden peräkkäin kulkevan referenssimuuttujan avulla. Olkoon referenssi poistettavaan tietueeseen `old`. Sen lisäksi täytyy olla tiedossa referenssi `prev` edeltävään tietueeseen. Jos poistettava tietue on listan ensimmäinen, sovitaan siitä että `prev = null`. Poiston suoritus:

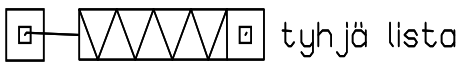
```
if (prev != null) prev.next = old.next;
else plista = old.next;
// Tehdään poistetulla tietueella mahdollisesti jotain
// Vapautetaan poistetun tietueen viemä tila
```

Vaihtoehtoisia listarakenteita

Alkutietueen käyttö

Tällöin listassa on ensimmäisenä tietue, missä ei ole mitään varsinaista tietoa. Muuten se on samaa tyyppiä kuin listan tietueet. Jos sen kenttien tyypit ovat sopivia, voidaan siihen tietenkin varastoida listan alkioille yhteistä tietoa. Tyhjässä listassa on vain alkutietue. Alkutietueen käyttö yksinkertaistaa hieman listaoperaatioita. Tarkastellaan esimerkkinä tietueen poistoa listasta, joka käyttää alkutietuetta.





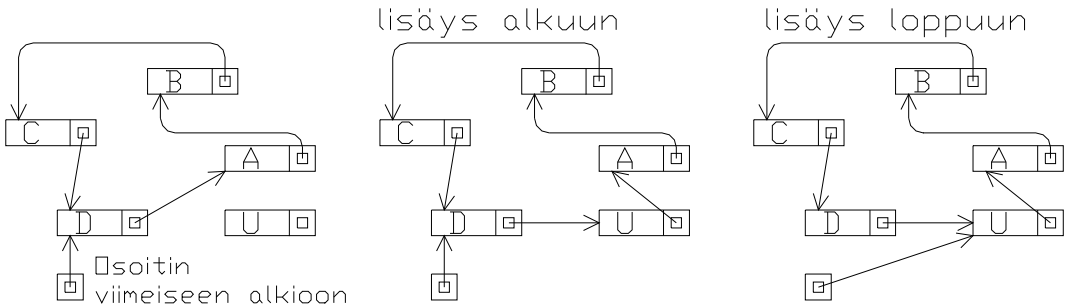

```
//haku ja poisto alkutietueella varustetusta listasta
tapaht poista(
  // palautetaan referenssi poistettuun tietueeseen
  // tai null, jos ei löydy
  int paika, // poistetaan eka tietue, jonka aika on tämä
  tapaht sl  // referenssi listan alkutietueeseen
  ){
    tapaht pi, pdel;
```

```
    // Haetaan tietue
    for (pi = sl; pi.next != null; pi = pi.next)
      if (paika == pi.next.aika) { // Poistetaan
        pdel = pi.next; // poistettava tietue
        pi.next = pdel.next;
        return pdel;
    }
    return null;  // Tietue ei ole listassa
  }
```

Edellä on huomattava, että referenssi `pi` osoittaa tutkittavan tietueen edeltäjään. Aluksi tämä on listan alussa oleva tyhjä alkutietue.

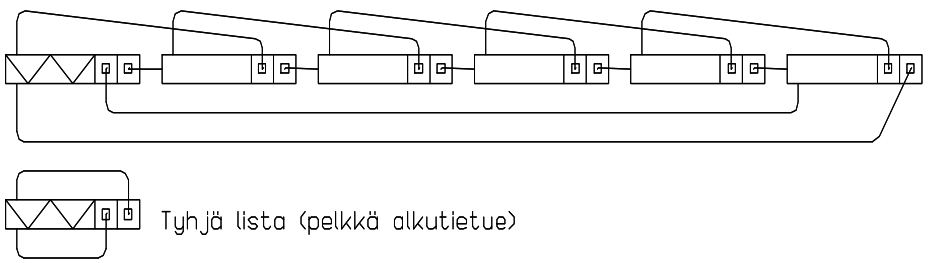
Rengaslista

Listan viimeisen alkion seuraajareferenssi asetetaan osoittamaan listan ensimmäistä alkiota, jolloin muodostuu rengas. Referenssi itse renkaaseen voi olla sen viimeiseen alkioon. Silloin renkaaseen voidaan lisätä sekä sen alkuun että loppuun. Renkaan ensimmäinen alkio voidaan poistaa helposti, mutta jos viimeinen alkio poistetaan, on sen edeltäjä haettava käymällä läpi koko rengas. Kun kuljetaan rengasta läpi, täytyy lopetus testata sillä, että tullaan takaisin samaan tietueeseen mistä lähdettiinkin.



Kaksoislinkitetty lista

Nytkin listan tietueissa on referenssi `next`, joka osoittaa listan seuraavaan tietueeseen, tai on `null`, jos tietue on viimeisenä. Lisätään tietueeseen referenssi `previous`. Se osoittaa tietuetta edeltävään tietueeseen, tai on `null`, jos tietue on listan ensimmäinen. Tällä rakenteella on se etu, että tietue voidaan aina poistaa listan mielivaltaisesta paikasta ilman että täytyy hakea sen naapureita. Naapurithan saadaan aina selville seuraajien ja edeltäjien avulla. Tämä on edullista silloin, kun poistettava tietue saadaan selville ilman että kuljetaan pitkin listaa. Lisääminen annetun tietueen edelle sujuu myös ilman, että täytyy hakea edeltävää tietuetta.



Haittana on kuitenkin, että lisäämisessä ja poistossa on monta listan päistä aiheutuvaa erikoistapausta. Kun kaksoislinkitetyn avoimen ketjun asemesta käytetään alkutietueella varustettua kaksoislinkitettyä rengasta, eivät päät ja tyhjä lista aiheuta mitään ongelmia.

Tietue uusi lisätään kaksoislinkitettyyn alkutietueella varustettuun renkaaseen ennen tietuetta seur seuraavasti.

```
uusi.previous = seur.previous; uusi.next = seur;
seur.previous.next = uusi; seur.previous = uusi;
```

Kaksoislinkitetty rengas ilman alkutietuetta on luonteva ratkaisu mm. monikulmioiden esittämisessä. Silloin on lisääminen tyhjään listaan ja listan tyhjentyminen poiston yhteydessä otettava erikseen huomioon.

2.4 Binääripuu

Binääripuun perusolioita ovat puun solmut. Määritellään binääripuu rekursiivisesti:

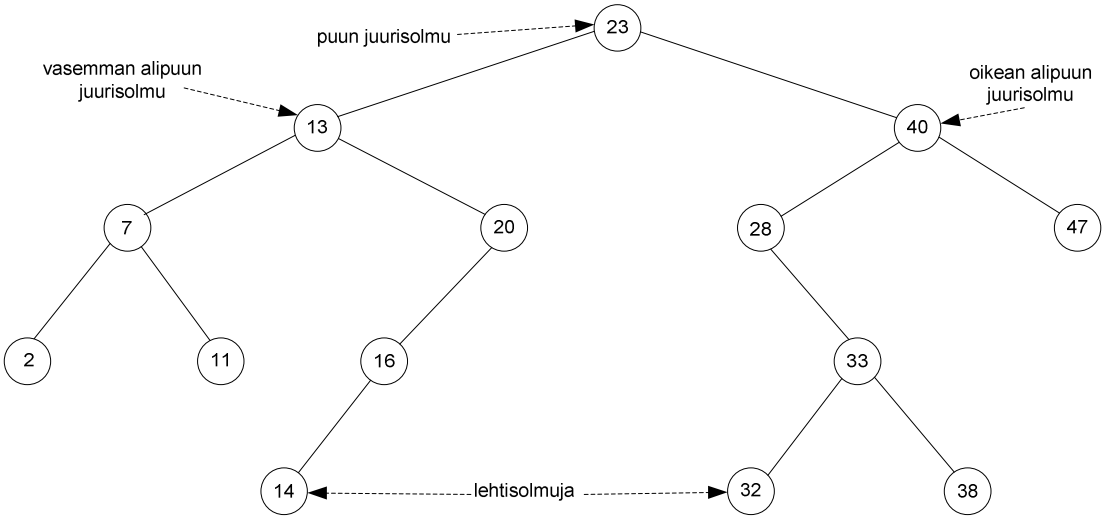
- Binääripuu voi olla tyhjä.
- Jos binääripuu ei ole tyhjä, on siinä juurisolmu, vasen alipuu ja oikea alipuu.
- Ei-tyhjän binääripuun vasen alipuu ja oikea alipuu ovat taasen binääripuita.

Binääripuun solmuille voidaan määritellä *rakenteellinen järjestys*. Jos solmu a edeltää solmua b rakenteellisessa järjestyksessä, niin käytetään merkitään $a \prec b$. Rakenteellinen järjestys määritellään seuraavasti.

- Jos solmu a on solmun b vasemmassa alipuussa, niin $a \prec b$.
- Jos solmu b on solmun a oikeassa alipuussa, niin $a \prec b$.
- Edellisistä tietysti seuraa, että jos a on solmun c vasemmassa alipuussa ja b solmun c oikeassa alipuussa, niin $a \prec c \prec b$, joten $a \prec b$.

Huomattakoon, että rakenteellisessa järjestyksessä on aina joko $a \prec b$ tai $b \prec a$. Mitään yhtä suuruutta muistuttavaa ei ole.

Olkoon sitten puun solmuille määritelty järjestys niiden tietosisällön perusteella. Tällöin voi siis olla $a < b$, $a > b$ tai $a = b$. Tavallisimmin tämä järjestys määritellään solmutietueiden avainten perusteella. Binääripuu on *järjestetty*, kun kaikilla sen solmupareilla a ja b: $a \prec b \rightarrow a \leq b$. Tällöin puhutaan usein *binäärisestä etsintäpuusta* tai *hakupuusta*. Huomattakoon, että jos binääripuussa on useampia yhtä suuria tietueita (tietosisällön perusteella), ei niiden keskinäinen rakenteellinen järjestys ole yksikäsitteinen.



Tästä määrittelystä seuraa, että kaikki järjestetyn binääripuun solmun c ei-tyhjässä vasemmassa alipuussa olevat solmut ovat ≤ alipuun juurisolmu c. Samoin kaikki binääripuun solmun c ei-tyhjässä oikeassa alipuussa olevat solmut ovat ≥ alipuun juurisolmu c. Edelleen kaikki solmun c vasemmassa alipuussa olevat solmut ovat pienempiä tai yhtä suuria kuin solmun c oikeassa alipuussa oleva solmut.

Jatkossa oletetaan, että binääripuut ovat järjestettyjä, jos ei erikseen toisin mainita.

Solmun vasemman alipuun juurisolmu on solmun *vasen (vasemmanpuoleinen) lapsi*. Vastaavasti puhutaan solmun *oikeasta (oikeanpuoleisesta) lapsesta*. Solmu on lastensa *vanhempi (isä)*. Saman solmun kaksi lasta ovat *sisaruksia*. *Solmun edeltäjiä (esivanhempia)* ovat solmu itse ja solmun vanhempi, tämän vanhempi jne. *Solmun seuraajia* ovat vastaavasti solmu itse, sen lapset, näiden lapset jne. Lapseton solmu on *lehtisolmu*. Muut kuin lehtisolmut ovat *sisäsolmuja*.

Binääripuu piirretään yleensä siten, että juurisolmu on ylhäällä ja sen vasen lapsi on tämän juurisolmun vasemmalla alapuolella ja vastaavasti oikea lapsi oikealla alapuolella. Solmu yhdistetään viivalla lapsiinsa ja tästä viivasta käytetään nimitystä *haara, kaari* tai *oksa*.

Polku solmusta a solmuun b on joukko solmuja v_1 (= a), v_2, …, v_{k-1}, v_k (= b), siten, että v_{i-1} on aina solmun v_i vanhempi. Vastaavasti voi olla polku siten, että v_{i-1} on aina solmun v_i lapsi. *Polun pituus* on polulla olevien haarojen lukumäärä. Puun solmulle voidaan määritellä *syvyys*. Solmun syvyys on kyseisestä solmusta koko puun juurisolmuun vievän polun pituus. Juurisolmun syvyys on 0 ja sen lapsien syvyys on 1. Solmun *korkeudella* taas tarkoitetaan pisimmän siitä lehtisolmuun vievän polun pituutta. Tällöin lehtisolmun tulee tietysti olla tarkasteltavan solmun seuraaja. Binääripuun korkeus on sen juurisolmun korkeus. Binääripuun solmuille voidaan määritellä myös *taso*, joka vastaa solmun syvyyttä. Juurisolmun taso on 0, juurisolmun lapsien taso on 1 ja niiden lapsien taso on jne. Solmun taso on siis koko binääripuun juurisolmusta tähän solmuun vievän polun pituus.

Binääripuiden implementointi

Binääripuiden solmut ovat sovelluksissa yleensä samantyyppisiä tietueita. Jos binääripuun solmutietueet allokoidaan dynaamisesta muistista, ja järjestys määritellään avaimen avulla, voivat ne olla esimerkiksi seuraavaa muotoa:

```
public class BinTreeNode{
  Object element;
  BinTreeNode left;    // referenssi vasemmalle tai null
  BinTreeNode right;   // referenssi oikealle tai null
  keyType key;         // Puun järjestysavain
                       // Muut mahdolliset määrittelyt
}
```

Usein käytetään myös talletustapaa, jolloin solmussa on kolme linkkikenttää. Tällöin vasen ja oikea linkkien lisäksi on vanhempi linkki, jossa on referenssi solmun vanhempaan. Tämä mahdollistaa täten kulkemisen binääripuussa myös lehtisolmuista kohti juurisolmua ilman, että talletetaan solmureferenssejä apumuistiin.

Puun solmut voidaan tallettaa myös taulukkoon, joka koostuu tietueista, tai tiedot on hajotettu erillisiin rinnakkaisiin taulukoihin. Silloin puuosoittimet ovat taulukoiden indeksejä. Jos puurakenne on varsin stabiili, niin erilaiset taulukkotalletusmuodot voivat olla käyttökelpoisia.

Binääripuun perusalgoritmit

Solmun haku binääripuusta

On haettava solmu, jonka avain on sama kuin annettu hakuavain, tai todettava, että binääripuussa ei ole sellaista. Yleisemmin on annettu referenssi hakutietueeseen, `h`, ja referoitu tietue sisältää tietueiden vertailussa käytettävät mallitiedot. Binääripuusta on haettava solmu, joka binääripuulle määritellyn järjestysrelaation mielessä on yhtä suuri kuin hakutietue.

Käytetään menetelmää, jonka etuna on, että jokaista kuljettua solmua kohti tehdään vain yksi solmujen järjestyksen vertailu. Yhtä suuruus haettavan solmun kanssa testataan vasta lopussa. Jos binääripuussa on useampia solmuja, jotka ovat yhtä suuria kuin `h`, niin niistä valitaan rakenteellisessa järjestyksessä ensimmäinen.

Algoritmi. Solmun haku binääripuusta
1) Asetetaan muuttujaan `p` referenssi puun juurisolmuun ja ehdokassolmu `c = null`.
2) Toistetaan seuraavaa kunnes `p = null`.
   - Jos `h.key > p.key` asetetaan, `p` = solmun `p` oikean alipuun juurisolmu.
   - Muuten asetetaan `c = p` ja `p` = solmun `p` vasemman alipuun juurisolmu.
3) Lopetus: Jos `c` ≠ `null` ja `c.key = h.key` niin `c` on haettu solmu. Muussa tapauksessa haettua solmua ei ole puussa.

Algoritmissa siis lähdetään juurisolmusta. Solmusta mennään vasemmalle, jos hakuavain on ≤ solmun avain. Muuten mennään oikealle. Kun yritetään mennä tyhjään haaraan, lopetetaan. Aina kun mennään vasemmalle, asetetaan se solmu, mistä lähdettiin, uudeksi ehdokassolmuksi. Lopuksi katsotaan onko ehdokassolmun avain yhtä suuri kuin hakuavain. Kuljettujen solmujen jono muodostaa hakupolun.

Voidaan osoittaa, että esitetty algoritmi löytää haettavan solmun, jos binääripuussa on sellainen. Ja jos on useampia sellaisia, joiden avaimen arvot ovat haettavan solmun avaimen arvoja niin tämä algoritmi löytää niistä rakenteellisessa järjestyksessä ensimmäisen. Lisäksi algoritmin päättyessä hakupolku päättyy haetun solmun vasemman alipuun rakenteellisessa järjestyksessä viimeiseen solmuun.

Solmun lisääminen binääripuuhun

Olkoon `u` referenssi lisättävään solmuun. Jos sallitaan, että binääripuussa voi olla yhtä suuria solmuja (annetun järjestyksen suhteen), uuden solmun pitää tulla viimeiseksi niistä. Silloin binääripuun järjestystä sanotaan *stabiiliksi*. Seuraavassa menettelyssä itse asiassa yritetään hakea viimeistä solmua, jonka avain on yhtä suuri kuin `u`:n referoiman solmun avain. Tätä varten modifioidaan esitettyä hakumenetelmä peilikuvakseen, jolloin siis lähdetään oikealle solmusta `x`, kun `u.key` ≥ `x.key` ja asetetaan uudeksi kandidaattisolmuksi `c = x`.

Algoritmi. Solmun lisääminen binääripuuhun
1) Alustus.
   - Asetetaan uudella solmulla `u` vasen ja oikea linkki arvoon `null`. Jos binääripuu on tyhjä, asetetaan uusi solmu binääripuun juureksi ja lopetetaan.
   - Asetetaan `p` = referenssi puun juurisolmuun. Solmu `c:= null`.
2) Suoritetaan vertailun mukaan toinen seuraavista kohdista.
   - Jos `u.key < p.key`, niin:
     - Jos solmun `p` vasen alipuu on tyhjä, jatketaan kohdasta 3.
     - Muussa tapauksessa asetetaan `p = p.left` ja toistetaan kohdasta 2.
   - Jos `u.key` ≥ `p`.key, niin:
     - Asetetaan `c = p`.
     - Jos solmun `p` oikea alipuu on tyhjä, jatketaan kohdasta 3.
     - Muussa tapauksessa asetetaan `p = p.right` ja toistetaan kohdasta 2.
3) Jos puussa ei saa esiintyä yhtä suuria solmuja ja `c` ≠ `null` ja `c.key = u.key` niin VIRHELOPETUS. Muussa tapauksessa:
   - Jos `c = p` niin uusi solmu `u` asetetaan solmun `p` oikeaksi alipuuksi.
   - Jos `c` ≠ `p` niin uusi solmu `u` asetetaan solmun `p` vasemmaksi alipuuksi.

Binääripuun ensimmäisen ja viimeisen solmun poisto

Binääripuun ensimmäisellä solmulla tarkoitetaan rakenteellisessa järjestyksessä ensimmäistä solmua. Vastaavasti binääripuun viimeinen solmu on rakenteellisessa järjestyksessä viimeinen solmu. Näihin kahteen kohdistuvat poisto-operaatiot ovat helpompia kuin yleinen minkä tahansa binääripuun solmun poisto.

Ensimmäinen solmu haetaan kulkemalla juurisolmusta jatkuvasti vasemmalle, kunnes tullaan solmuun `f`, jonka vasen alipuu on tyhjä. Solmu `f` on puun ensimmäinen solmu. Samalla tavalla puun viimeinen solmu löydetään kulkemalla juurisolmusta oikealle, kunnes tullaan solmuun `l`, jonka oikea alipuu on tyhjä.

Algoritmi. Binääripuun ensimmäisen solmun poisto

1) Käytetään kahta solmua (referenssiä) `p` ja `q`, jolloin `p = q.left` eli `p` on solmun `q` vasen seuraaja. Jos puu ei ole tyhjä, asetetaan aluksi `p` = puun juuri ja `q = null`.
2) Toistetaan seuraavaa:
   - Jos `p.left = null` niin `p` on puun ensimmäinen solmu. Jatketaan kohdasta 3.
   - Muuten edetään vasemmalle: `q = p` ja `p = p.left`. Toistetaan kohdasta 2.
3) Jos `q = null` niin poistettava solmu `p` on puun juuri. Silloin uudeksi juureksi tulee `p.right`. Muussa tapauksessa asetetaan solmun `p` oikea alipuu solmun `q` vasemmaksi alipuuksi, eli `q.left = p.right`.

Solmun poisto binääripuusta

Binääripuusta on poistettava solmu, jonka hakuavain on annettu tai yleisemmin se tietue, jonka avain annetun järjestyksen määrittelemällä tavalla on yhtä suuri kuin annetun hakutietueen (`h` referenssi tähän) avain. Yleisperiaatteena seuraavassa on, että ensin poistettava solmu haetaan ja sitten sen paikalle puuhun siirretään poistetun solmun vasemman alipuun viimeinen solmu. Käytettäessä hakuun aiemmin esitettyä menetelmää, päättyy haku juuri tähän solmuun. Mikäli poistettava solmu ei ole binääripuun juuri, täytyy lisäksi olla tiedossa poistettavan solmun edeltäjä sekä poistettavan solmun asema (vasemmalla tai oikealla) sen edeltäjään nähden. Monenlaiset erikoistapaukset tekevät poistamisalgoritmin hieman hankalaksi.

Algoritmi. Solmun haku ja poistaminen binääripuusta

1) Haussa käytetään kahta solmua (referenssiä) `p` ja `q` siten, että `q` on aina solmun `p` edeltäjä. Kandidaattisolmun `c` lisäksi talletetaan sen edeltäjä c_p. Jos binääripuu ei ole tyhjä, alustetaan `p` = binääripuun juuri ja `c = null`.
2) Haetaan poistettava solmu binääripuusta hakualgoritmilla.
   - Jos `h.key` $\leq$ `p.key` niin:
     Aseta `c = p`, c_p `= q`.
     Jos solmun `p` vasen alipuu on tyhjä, niin jatka kohdasta 3.
     Muutoin aseta `q = p` ja `p=p.left`. Toista kohdasta 2.
   - Muussa tapauksessa:
     Jos solmun `p` oikea alipuu on tyhjä, niin jatka kohdasta 3.
     Muutoin aseta `q = p`, `p = p.right`. Toista kohdasta 2.
3) Jos `c == null` tai `c` $\neq$ `null` ja `c.key` $\neq$ `h.key`, poistettavaa solmua ei ole puussa. VIRHELOPETUS.
4) Suoritetaan solmun `c` poisto:
   - Jos solmun `c` vasen alipuu on tyhjä, niin korvataan `c` solmulla `p = c.right`. Tämä voi olla tyhjä.

- Muussa tapauksessa korvaava solmu `p` on solmun `c` vasemman alipuun viimeinen solmu `p`, johon haku päättyi. Kytketään solmun `c` oikea alipuu solmun `p` oikeaksi haaraksi (solmun oikea haara oli ennestään tyhjä).
  Jos `p` ei ole solmun `c` vasen seuraaja (silloin siis `q` $\neq$ `c`) niin irrotetaan `p` solmun `c` vasemmasta alipuusta asettamalla solmun `p` vasen alipuu edeltävän solmun `q` oikeaksi alipuuksi (`q.right = p.left`). Sitten jäljellä oleva osa solmun `c` vasenta alipuuta kytketään solmun `p` vasemmaksi alipuuksi.
  Muussa tapauksessa (`p` on heti solmusta `c` vasemmalla) ei tehdä mitään.
- Kytketään korvaava solmu `p` solmun `c` tilalle:
  Jos `c` on puun juurisolmu, niin asetetaan `p` uudeksi juureksi.
  Muussa tapauksessa asetetaan solmu `p` solmun c_p vasemmaksi tai oikeaksi haaraksi riippuen siitä kumpaan se kuuluu.

Binääripuun solmujen läpikäynti

Binääripuun solmut voidaan käydä läpi useammassa eri järjestyksessä. Kolme tavallisinta järjestystä ovat esi-, sisä- ja jälkijärjestys. Eri järjestykset on helposti määriteltävissä rekursiivisesti ja määrittelyjen pohjalta on erittäin helppo kirjoittaa myös rekursiiviset algoritmit.

Esijärjestys (preorder):
1) Käsittele tarkasteltava solmu.
2) Jos tarkasteltavan solmun vasen alipuu ei ole tyhjä, niin tutki tämä vasen alipuu esijärjestyksessä.
3) Jos tarkasteltavan solmun oikea alipuu ei ole tyhjä, niin tutki tämä oikea alipuu esijärjestyksessä.

Sisäjärjestys (välijärjestys, inorder):
1) Jos tarkasteltavan solmun vasen alipuu ei ole tyhjä, niin tutki tämä vasen alipuu sisäjärjestyksessä.
2) Käsittele tarkasteltava solmu.
3) Jos tarkasteltavan solmun oikea alipuu ei ole tyhjä, niin tutki tämä oikea alipuu sisäjärjestyksessä.

Jälkijärjestys (postorder):
1) Jos tarkasteltavan solmun vasen alipuu ei ole tyhjä, niin tutki tämä vasen alipuu jälkijärjestyksessä.
2) Jos tarkasteltavan solmun oikea alipuu ei ole tyhjä, niin tutki tämä oikea alipuu jälkijärjestyksessä.
3) Käsittele tarkasteltava solmu.

Esimerkki. Binääripuun korkeuden määrittäminen

Seuraavassa rekursiivisessa algoritmissa selvitetään binääripuun korkeus. Tässähän puu käydään läpi jälkijärjestyksessä.

`binääripuunKorkeus(solmu)`
1) jos solmu on `null`, niin palauta -1.
2) jos solmun vasen alipuu on tyhjä, niin aseta `vasen = -1`

muutoin aseta `vasen = binääripuunKorkeus(solmu.left)`.
3) jos solmun oikea alipuu on tyhjä, niin aseta `oikea = -1`
muutoin aseta `oikea = binääripuunKorkeus(solmu.right)`.
4) jos `vasen > oikea`, niin palauta `vasen + 1`
muutoin palauta `oikea + 1`.

Binääripuun läpikäynti ilman rekursiota

Edellä esitettiin binääripuun läpikäynti rekursiivisen algoritmin avulla. Rekursiolla on omat haittansa ja voidaan haluta ei-rekursiivinen binääripuun läpikäyntialgoritmi. Rekursion voi poistaa useammalla tavalla, mutta tavallisin tapa on käyttää apupinoa.

Seuraavassa on algoritmi, joka käy binääripuun läpi sisäjärjestyksessä ilman rekursiota. Periaatteena on, että aina kun lähdetään solmusta vasemmalle, talletetaan pinoon solmun referenssi. Kun sitten vasen alipuu on käyty läpi, on pinossa ensimmäisenä se solmu, joka nyt tulee käsittelyvuoroon.

Algoritmi. Binääripuun läpikäynti sisäjärjestyksessä pinon avulla
1) Solmujen läpikäynnin aloitus. Aloittaen binääripuun juuresta kuljetaan vasemmalle kunnes tullaan solmuun, jonka vasen alipuu on tyhjä. Samalla kaikki solmut, joihin tullaan, talletetaan pinoon.
2) Vuorossa olevan solmun saanti.
   - Jos pino on tyhjä, niin lopetetaan. Muuten poistetaan pinosta sen päällimmäinen solmu `p`.
   - Olkoon `q` solmun `p` oikean alipuun juuri.
   - Jos `q` ei ole `null` niin talletetaan solmu `q` pinoon.
   - Sitten kuljetaan solmusta `q` vasemmalle niin pitkällä kuin päästään ja talletetaan samalla solmut pinoon.
   - Solmu `p` on nyt seuraava vuorossa oleva solmu.

Binääripuun operaatioiden vaativuus

Selvästi voidaan havaita, että binääripuiden perusalgoritmien, haku, lisäys, poisto, askelten määrä on ≤ pisin polku juurisolmusta binääripuun lehtisolmuihin.

Arvioidaan binääripuun korkeutta ja tarkastellaan ensin erikoistapauksia. Tasapainoisessa binääripuussa jokaisella solmulla on kaksi haaraa lukuun ottamatta korkeimman tason solmuja. Olkoon puun korkeus *h*. Silloin siinä on *h+1* tasoa eli kerrosta. Ylimmässä kerroksessa on vain juurisolmu. Seuraavassa kerroksessa on 2 solmua, sitä seuraavassa 4 solmua jne. Alimmassa kerroksessa on korkeintaan 2^h solmua. Binääripuussa on siis kaikkiaan $n = 1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1}-1$ solmua. Todetaan, että binääripuun korkeus $h = \log_2(n+1)-1$. Tämä pitää paikkansa myös, kun binääripuun korkeus on ylipäänsä mahdollisimman pieni solmujen lukumäärän ollessa kiinteä.

Toisaalta binääripuun muodostaminen voi myös täysin epäonnistua. Jos solmut lisätään esimerkiksi kasvavassa tai vähenevässä järjestyksessä, tulee binääripuusta

niin sanottu degeneroitunut binääripuu eli itse asiassa puhdas lineaarinen lista. Silloin sen korkeus on *n-1*.

On siis vältettävä tilannetta, missä binääripuu muodostetaan tietueista, jotka ovat melkein järjestyksessä.

Voidaan todistaa, että satunnaisesti muodostetussa binääripuussa hakupolun keskipituus on noin 1.39 $\log_2 n$. Jos siis solmut lisätään binääripuuhun satunnaisessa suuruusjärjestyksessä, ei syntyvä binääripuu ole yleensä huono.

Tarkastellaan yksinkertaista binääripuun tasapainossa pitomenetelmää, joka sopii lähinnä staattisiin tilanteisiin. Tällä tarkoitetaan sitä, että kun binääripuu jossain vaiheessa on kerran saatu rakennettua niin poistoja ja lisäyksiä ei enää kovin paljon tapahdu.

Kun halutaan tasapainottaa binääripuu, josta epäillään, että se ei ole enää tasapainossa, käydään binääripuun solmut läpi rekursiivisella algoritmilla kasvavassa järjestyksessä eli sisäjärjestyksessä ja talletetaan referenssit tietueisiin taulukkoon. Epätasapaino voidaan huomata laskemalla hakujen yhteydessä hakupolun solmujen lukumäärä. Jos tämä on riittävän paljon suurempi kuin teoreettinen keskiarvo, niin suoritetaan binääripuun uudelleen tasapainotus. Tasapainoisen binääripuun muodostus tapahtuu taulukosta seuraavasti. Asetetaan järjestyksessä keskimmäisestä solmusta uuden binääripuun juurisolmu. Sitä pienemmistä muodostetaan juurisolmun vasen alipuu ja sitä suuremmista juurisolmun oikea alipuu. Nämä molemmat voidaan muodostaa rekursiivisesti.

On olemassa lukuisia menetelmiä, joilla voidaan muodostaa likimäärin tasapainoinen puu sekä säilyttää se sellaisena myös lisäyksiä ja poistoja suoritettaessa. Tällaisia ovat mm. 2-3-puut, AVL-puut ja puna-mustat puut. Näissä tulee jokaisen solmun lisäyksen ja poiston yhteydessä säilyttää puulla tietty tasapainoisuus ominaisuus. Tämä tasapainoisuus on eri menetelmissä hiukan eri lailla määritelty, mutta aina on periaatteena, että jokaisen solmun vasen ja oikea alipuu eivät saa kovin paljon erota toisistaan korkeuden suhteen.

Binääripuun käyttö hakemistona

Hakemistolla (sanakirja, dictionary) tarkoitetaan tietorakennetta, jonka perusoperaatioita ovat lähinnä alkion lisäys, alkion poisto ja alkion haku. Binääripuuta käytettäessä operaatiot lisäys, poisto, haku, ensimmäisen ja viimeisen solmun saanti ja poisto voidaan suorittaa ajassa $O(\log n)$, kun käytetään nimenomaan tasapainoista binääripuuta. Järjestetyssä linkitetyssä listassa vastaavat ajat ovat O(n) lukuun ottamatta ensimmäisen solmun saantia ja poistoa, joka voidaan suorittaa vakioajassa. Jos esimerkiksi $n = 1000$, on tarvitaan haussa keskimäärin 14 avainten vertailua, kun puu on satunnainen. Linkitetyssä listassa vertailuja on sen sijaan keskimäärin 500.

2.5 Verkot

Suunnatussa graafissa on *solmu*jen joukko `N` ja *kaari*en joukko `A`. Kullakin kaarella `a` on tarkalleen yksi alkusolmu `t(a)` ja yksi loppusolmu `h(a)`. Jokaiseen kaareen täytyy kuulua sen alku- ja loppusolmu. Sen sijaan solmuun ei tarvitse liittyä yhtään kaarta.

Kuhunkin solmuun `j` liittyy kaksi kaarien joukkoa: lähtevät kaaret `A(j)` ja tulevat kaaret `B(j)`. Näiden kaarien toisia solmuja sanotaan solmun `j` naapurisolmuiksi. Olkoon graafissa joukko kaaria $a_1, a_2, \ldots, a_k$, joilla on se ominaisuus, että $h(a_i) = t(a_{i+1}), i = 1, \ldots, k-1$. Silloin nämä kaaret muodostavat *reitin* tai *polun*. Polku on *yksinkertainen*, jos mikään solmu ei toistu. Jos alkusolmu a_1 on sama kuin loppusolmu a_k niin reitti on *silmukka*.

Tavallisesti graafilla ymmärretään oliota, missä kaarien ja solmujen välillä on annettu vain niiden insidenssi (liittyminen toisiinsa). Jos solmuihin ja kaariin liittyy muitakin kuvauksia, sanotaan graafia myös *verkoksi*. Nimityksissä on kylläkin horjuvuutta. Niinpä kullekin kaarelle voidaan määritellä sen pituus. Silloin reitin pituus on reitillä olevien kaarien pituuksien summa. Verkon kaarilla voi olla myös virtaus, jolloin kaareen liittyvä luku kuvaa kaaren kapasiteettia. Kaaren virtaus tapahtuu kaaren suuntaisesti ja sen suuruus ei voi ylittää kaaren kapasiteettia. Jokaisella solmulla on voimassa, että solmuun tulevien virtausten summa on yhtä kuin solmusta lähtevien virtausten summa (Kirchhoffin laki).

Suuntaamattomalla graafilla kaarille ei ole kiinnitetty erikseen alkusolmua ja loppusolmua, vaan kaari yksinkertaisesti vain yhdistää kaksi nämä solmua. Tällöin kaaresta usein käytetäänkin nimitystä *tie*. Graafi on *yhdistetty*, jos jokaisen solmuparin välillä on polku. *Puu*lla tarkoitetaan suuntaamatonta graafia, joka on yhdistetty ja jossa ei ole yhtään silmukkaa.

Suuntaamattomien graafien käsittely suoritetaan useimmiten siten, että kullekin tielle annetaan nimellinen suunta. Suuntaamaton graafi esitetään tietokoneessa sitten suunnattuna ja usein yksi suuntaamaton tie voidaan esittää kahtena suunnattuna kaarena. Algoritmeissa otetaan huomioon, että tien molemmat suunnat ovat nyt tasa-arvoisia.

Kaaritaulukko

Kaikkein yksinkertaisin rakenne on taulukko verkon kaarista mielivaltaisessa järjestyksessä. Kullekin kaarelle on annettu sen alkusolmun ja loppusolmun tunnukset sekä muut kaaren tiedot, esimerkiksi pituus. Rakenne on muokattavissa myös

suuntaamattomille verkoille. Muistitilaa tämä menetelmä käyttää vain vakion verran verkon jokaista kaarta kohden.

Tätä talletustapaa voidaan käyttää etsittäessä erittäin yleisellä menetelmällä verkon lyhimmät polut ja näiden lyhimpien polkujen etäisyydet verkon jostain solmusta verkon muihin solmuihin. Jokaiselle suunnatun verkon kaarelle `h` on annettu kaaren pituus `l(h)`. Jotkin pituuksista voivat olla myös negatiivisia. *Polun pituus* on polulla olevien kaarien pituuksien summa. Verkossa ei kuitenkaan saa olla silmukkaa, jonka kaarien yhteispituus on negatiivinen.

Seuraavassa menetelmässä määrätään lyhimmät polut annetusta lähtösolmusta `a` verkon kaikkiin niihin solmuihin, jotka ovat saavutettavissa siitä. Kullekin solmulle `i` pidetään yllä sen etäisyyttä `d(i)` alkusolmusta `a` pitkin tähän mennessä tunnettua lyhintä polkua ja vastaavan polun viimeisen kaaren alkusolmua `t(i)` ennen solmua `i`.

Algoritmi. Lyhimpien polkujen määrääminen Ford-Fulkersonin menetelmällä

1) Alusta lyhimmät etäisyydet `d(i) = ∞` (suuri luku) kaikilla muilla solmuilla, paitsi lähtösolmulla `d(a) = 0`.
2) Käsittele kaikki verkon kaaret mielivaltaisessa järjestyksessä.
   Olkoon vuorossa oleva kaari `k`. Sen alkusolmu on `i` ja loppusolmu `j`. Kaaren pituus on `l(k)`.
   Jos `d(i) + l(k) < d(j),` niin aseta `d(j) = d(i) + l(k)` ja `t(j) = i`.
3) Jos edellä tapahtui jokin muutos yhdessäkään luvussa `d(j)`, niin toista kohdasta 2. Muussa tapauksessa lopeta. Silloin luvut `d(j)` antavat solmujen etäisyydet lähtösolmusta pitkin lyhimpiä polkuja, ja `t(j)` ilmoittavat edellisen välittömän solmun lyhimmällä polulla lähtösolmusta solmuun `j`. Jos solmuun `j` ei johda lainkaan reittiä alkusolmusta niin `d(j) = ∞`.

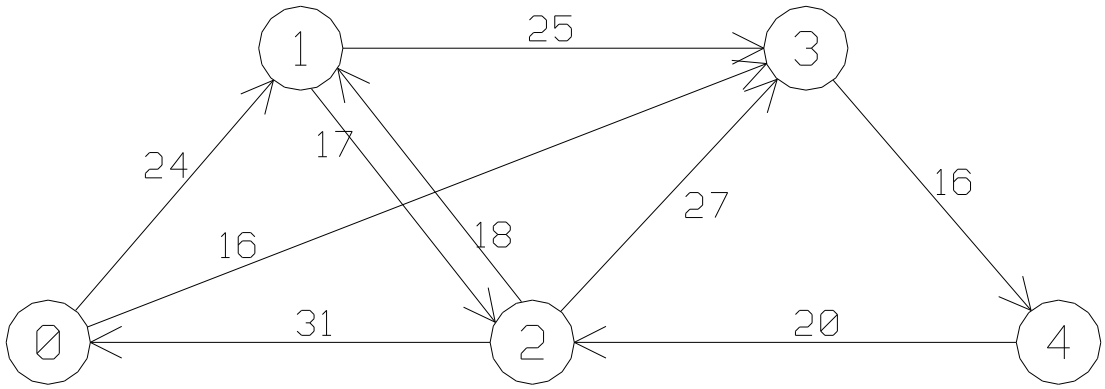
Tämä algoritmi ei ole huippunopea. Se on kuitenkin yksinkertainen ja toimii hyvin yksinkertaisella talletusrakenteella. Kohta 2 suoritetaan enintään `n` kertaa (`n` on solmujen lukumäärä). Kukin askel 2 suorittaa kierroksen kaarilistan läpi.

Bittimatriisi ja kaarien pituusmatriisi

Olkoon esitettävä puhdas graafi, missä ei ole annettu muuta kuin solmujen ja kaarien insidenssi. Jos graafissa ei ole rinnakkaisia kaaria, voidaan sen rakenne esittää `n×n` bittimatriisina `B`. Numeroidaan graafin solmut `0, …, n-1`. Silloin matriisin bitti B_{ij} `= 1`, jos solmusta `i` solmuun `j` johtaa kaari. Muutoin kyseinen bitti on 0.

Bittimatriisiesitys sopii, kun graafissa on paljon kaaria. Sen sijaan harvoissa graafeissa on probleemana se, että solmun naapureita etsittäessä täytyy tutkia kaikki `n` bittiä. Jos bitit ovat peräkkäin, voidaan kuitenkin pelkkiä 0-bittejä sisältävät tavut (sanat) ohittaa.

```
    0 1 2 3 4       0  1  2  3  4
0:  0 1 0 1 0       0 24  M 16  M
1:  0 0 1 1 0       M  0 17 25  M
2:  1 1 0 1 0      31 18  0 27  M
3:  0 0 0 0 1       M  M  M  0 16
4:  0 0 1 0 0       M  M 20  M  0
```



Oletetaan, että verkossa ei ole rinnakkaisia kaaria kahden solmun välillä. Silloin verkon rakenne ja kaarien pituudet voidaan esittää samalla matriisilla d. Olkoon solmusta i solmuun j johtavan kaaren pituus d_{ij}. Jos solmujen välillä ei ole kaarta, niin asetetaan $d_{ij} = \infty$ (käytännössä suuri luku). Päälävistäjän, vinorivi vasemmasta ylänurkasta oikeaan alanurkkaan, alkiot $d_{ii} = 0$. Suuntaamattomilla verkoilla matriisi on symmetrinen.

Suurilla ja harvoilla verkoilla matriisi tulee liikaa tilaa vieväksi, muistitilaa tarvitaan n^2 alkion verran. Edelleen määrättäessä solmun naapureita, joudutaan käymään läpi matriisin rivin kaikki n alkiota, vaikka naapureita olisi vain muutama, niin kuin verkoissa usein on asianlaita.

Matriisiesitys sopii kuitenkin Floydin menetelmään, jolla saadaan lyhimpien polkujen pituudet verkon kaikkien solmuparien välillä. Algoritmi perustuu siihen, että rajoitetaan sallittujen välisolmujen joukkoa solmuparin lyhimmällä polulla. Aluksi ei sallita välisolmuja lainkaan. Seuraavaksi sallitaan solmu 0 välisolmuksi. Yleisesti, oletetaan tunnetuksi lyhimpien polkujen pituudet, kun välisolmuina sallitaan solmut $0, 1, \ldots, k$. Seuraavaksi laajennetaan sallittujen välisolmujen joukkoa lisäämällä siihen solmu $k+1$. Tälle perustuu seuraava yksinkertainen Floydin menetelmä:

Floydin algoritmi. Etäisyysmatriisin määrääminen kaarien pituusmatriisista
1. Alusta lyhimpien etäisyyksien matriisi $D = d$, kun d on kaarien pituusmatriisi.
2. Suorita seuraava kaikilla arvoilla k, $k = 0,1, \ldots, n-1$:
- Kaikilla arvoilla $i, j, i, j = 0, 1, \ldots, n-1$:
  - Olkoon $z = D_{ik} + D_{kj}$.
  - Jos $z < D_{ij}$, niin asetetaan $D_{ij} = z$.

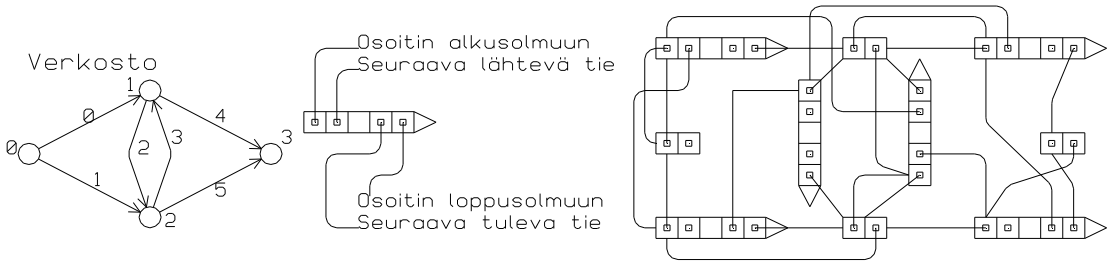
Tämän jälkeen matriisin D alkiot antavat lyhimpien reittien pituudet kaikkien solmujen välillä. Itse reitit saadaan huomaamalla, että ensimmäinen solmu lyhimmällä reitillä $i \rightarrow j$ on se solmu k, jolla on voimassa $D_{ij} = d_{ik} + D_{kj}$.

Lähtevien ja tulevien kaarien listat

Kullekin solmulle muodostetaan lista siitä lähtevistä kaarista ja tarpeen mukaan myös kyseiseen solmuun tulevista kaarista. Esityksestä käytetään myös nimitystä naapurilista.

Kutakin solmua kohti on solmutietue ja kullakin kaarella kaaritietue. Kaikki kaaret, jotka alkavat samasta solmusta muodostavat listan. Samoin kaikki kaaret, jotka päättyvät samaan solmuun, voivat muodostavat toisen listan. Listojen alkioiden järjestys on yleensä vapaa.

- Solmutietueen sisältö:
  - Referenssi lähtevien kaarien listan ensimmäiseen kaareen.
  - Referenssi tulevien kaarien listan ensimmäiseen kaaren, jos käytössä.
  - Referenssi graafin seuraavaan solmuun, jos graafin solmut on linkitetty listaksi.
  - Muut solmukohtaiset tiedot (solmun nimi, algoritmien työmuuttujat).
- Kaaritietueen sisältö:
  - Referenssit kaaren alkusolmuun ja loppusolmuun.
  - Referenssi seuraavaan kaareen, jolla on sama alkusolmu. Tämä on siis seuraajalinkki listassa, missä ovat alkusolmusta lähtevät kaaret.
  - Referenssi seuraavaan kaareen, jolla on sama loppusolmu, jos tämä käytössä. Tämä on seuraajalinkki listassa, missä ovat loppusolmuun tulevat kaaret.
  - Muut kaarikohtaiset tiedot (pituus, kapasiteetti, virtaus, algoritmien työmuuttujat).



Tämä rakenne sallii rinnakkaiset kaaret. Kaaret, joiden alku- ja loppusolmu on sama (self loop), voidaan periaatteessa esittää myös, mutta algoritmeissa niistä voi helposti tulla pulmia. Rakennetta voidaan käyttää myös suuntaamattomille verkoille. Silloin annetaan kaarille mielivaltainen suunta. Solmusta lähtevät ja siihen tulevat kaaret käsitellään tasa-arvoisina.

Itse tietueet ovat dynaamisessa muistissa tai ne on talletettu taulukoihin. Kaarilistarakenteet vievät paljon tilaa. Olkoon verkossa `n` solmua ja `m` kaarta. Silloin verkon rakennetiedot vaativat talletettavasta tiedosta hiukan riippuen `3n+4m` referenssiä. Rakenne on edullinen harvoilla verkoilla, jolloin kaarien määrä `m` ei ole suuri verrattuna solmujen määrään. Kommunikaatioverkot ovat usein tällaisia.

Forward Star-rakenne

Erittäin tiivis esimerkiksi lyhimpien reittien määräämiseen sopiva taulukkopohjainen esitys on Forward Star-rakenne. Se on tiivistetty esitys lähtevien kaarien listasta, naapurilistasta. Siinä kaaret on järjestetty alkusolmun numeron mukaan kasvavaan järjestykseen. Samasta solmusta lähtevät kaaret ovat siis kaaritaulukossa peräkkäin. Kaarista on annettu vain pituus ja loppusolmun numero.

Edelleen on solmutaulukko `s`, missä on `n+1` indeksiä. `s[i]` ilmoittaa missä paikassa kaaritaulukoissa on ensimmäinen kaari, jonka lähtösolmu on `i`. Tarvitaan myös `s[n+1]`, joka ilmoittaa, kaaritaulukon viimeistä kaarta seuraavan position. Silloin kaikilla solmuilla `i, i = 1, …, n` `s[i+1] - s[i]` ilmoittaa montako kaarta lähtee solmusta `i`. Huomattakoon, että kun solmusta `i` ei lähde yhtään kaarta, niin `s[i] = s[i+1]`.

```
Listaesitys taulukoissa                          Forward Star-esitys
---------------------------------------------    ---------------

  Solmut           Kaaret
  ffwa fbwa        tail head nxfw nxbw pit     s      head pit
0:   0  NIX      0:   0    1    1  NIX  43     0: 0  0:  1  43
1:   4    3      1:   0    2  NIX    2  24     1: 2  1:  2  24
2:   3    1      2:   1    2  NIX  NIX  26     2: 4  2:  2  26
3: NIX    5      3:   2    1    5    0  32     3: 6  3:  3  15
                 4:   1    3    2  NIX  15     4: 6  4:  1  32
                 5:   2    3  NIX    4  17           5:  3  17
                                                   6:NIX NIX
```

Forward Star-esitys vaatii tilaa `n+1` indeksille ja `m`:lle kaarialkiolle verkon rakenteen esittämiseen. Vastaava rakenne voidaan tehdä myös solmuun tulevien kaarien suhteen, jolloin kyse on backward star-esityksestä.

Forward Star rakenne sopii hyvin verkon talletukseen esimerkiksi saavutettavissa olevien solmujen selvittämisalgoritmille ja aikaisemmin esitetylle lyhimmän reitin algoritmille.

Tarkastellaan suuntaamatonta graafia. Jos solmusta `a` solmuun `b` on ainakin yksi polku, on solmu `b` saavutettavissa solmusta `a`. Seuraavalla algoritmilla selvitetään mitkä solmut ovat saavutettavissa annetusta lähtösolmusta. Algoritmia voidaan soveltaa myös suunnattuihin graafeihin, jolloin saavutettavuus määritellään suunnattujen polkujen avulla. Alla siis tarkastellaan suuntaamatonta graafia ja tie on silloin molempien päätesolmujen lähtevien kaarien listassa.

Algoritmi. Saavutettavissa olevat solmut

1) On määrättävä sellaiset solmut, jotka ovat saavutettavissa solmusta `a`. Olkoon `R` saavutettavissa olevien solmujen joukko ja `T` tutkittavien solmujen joukko. Asetetaan alustuksessa `T = R = {a}`.
2) Jos joukko `T = ∅`, niin lopetetaan. Joukko `R` käsittää silloin kaikki solmut, joihin johtaa polku solmusta `a`.
   Muussa tapauksessa poistetaan joukosta `T` jokin siihen kuuluva solmu `x`.
3) Käy läpi solmusta `x` lähtevät kaaret ja käsittele kukin kaari `(x, j)` seuraavasti:
   Jos `j` ∈ `R` niin ei tehdä mitään.
   muussa tapauksessa liitetään solmu `j` joukkoon `R` ja joukkoon `T`.

Jatka kohdasta 2.

Edellä esitettyä menetelmää voidaan soveltaa myös yleiseen verkon läpikäyntiin eli verkon kaikkien solmujen ja kaarien tutkintaan. Silloin kohtaa 2 tulee muuntaa siten, että joukon T tullessa tyhjäksi tulee vielä selvittää, onko verkon kaikki solmut tutkittu. Jos ei, niin valitaan jokin tutkimaton solmu uudeksi lähtösolmuksi ja toistetaan algoritmia, kunnes kaikki verkon solmut on käyty läpi. Itse asiassa joka kerta kun askeleessa 2 tulee joukko T tyhjäksi, niin on löydetty yksi uusi verkon yhdistetty komponentti. Esitetyllä tavalla voidaan selvittää verkon yhdistetyt komponentit.

Jos yllä joukkona T on jono (lisäys loppuun, poisto alusta), kutsutaan algoritmia myös *leveäksi hauksi* (leveyssuuntainen haku, Breadth-First Search). Tässä verkon läpikäynti tehdään siten, että ensin tutkitaan kaikki solmut, jotka ovat yhden kaaren päässä lähtösolmusta. Sen jälkeen ne solmut, jotka ovat kahden kaaren päässä lähtösolmusta jne. Jos joukkona T on pino (lisäys alkuun, poisto alusta), kyseessä on *syvähaku* (syvyyssuuntainen haku, Depth-First Search). Tällöin verkon tutkiminen jatkuu aina viimeksi tutkitusta solmusta eteenpäin, syvemmälle, kunnes tutkittavasta solmusta ei enää päästä eteenpäin. Silloin palataan jatkamaan jostain jo aikaisemmin käydystä solmusta.

Forward star rakenteella voidaan tehostaa myös aikaisemmin esitettyä lyhimpien polkujen Ford-Fulkersonin menetelmää.

2.6 Muita perustietorakenteita

Tässä ei ole tarkoitus käsitellä tarkemmin muita perustietorakenteita. On olemassa useita tärkeitä puurakenteita; kekorakenteet, b-puut, tasapainoiset puurakenteet, jne. Tämän lisäksi tärkeä tietorakenne on hajautusmenetelmän (hashing) tuntemus. Varsinkin, jos haku on tärkein operaatio, niin hajautus on huomioonotettava mahdollisuus. Jos kullakin solmulla on olemassa tärkeyskerroin eli prioriteetti niin prioriteettijono voi tulla kyseeseen. Tällöin tärkeimpiä operaatioita ovat alkion lisääminen ja pienimmän (tai suurimman) prioriteetin omaavan alkion poisto.

3 Algoritminen ongelman ratkaisu

Algoritmit ovat perustana ratkaistaessa ongelmia tietokoneella ohjelmallisesti. Ne eivät ole suoria vastauksia, vaan ne muodostuvat ohjeista, joiden avulla ratkaisu on saavutettavissa. Eräs mahdollisuus algoritmiseen ongelman ratkaisuun on seuraavien vaiheiden mukainen.

Ensimmäisenä vaiheena ongelman ratkaisussa on itse ongelman ymmärtäminen. Ongelman kuvaus on tutkittava tarkoin ja selvitettävä epäselvyydet. Pienten esimerkkitapausten läpikäynti paperilla on usein hyödyllistä. Ongelman voidaan todeta vastaavan jotain usein esiintyvää ongelmaa, jolloin kannattaa käyttää kirjallisuudesta löytyviä valmiita ratkaisumenetelmiä. Tällaisten tyyppiongelmien ja niiden ratkaisumenetelmien tunteminen auttaa muunkinlaisten ongelmien ratkaisemisessa.

Seuraavaksi tarkastellaan ratkaisumahdollisuuksia eli asettaako ratkaisuympäristö jotain rajoituksia. Onko esimerkiksi käytettävässä laitteistossa rinnakkaisprosessointi mahdollista? Onko ratkaisun selvittämisessä aikarajoituksia?

Lisäksi tulee selvittää halutaanko tarkka paras mahdollinen ratkaisu vai tyydytäänkö kenties likiarvoiseen, approksimatiiviseen ratkaisuun. Joitakin ongelmia ei voida lainkaan ratkaista tarkasti ja toisaalta joidenkin ongelmien tarkka ratkaisu voi olla äärettömän hidasta.

Algoritmin suunnitteluun liittyy hyvin tiukasti myös käytettävien tietorakenteiden suunnittelu ja valinta.

Itse algoritmin suunnittelussa käytetään yleensä menetelmiä, jotka ovat osoittautuneet hyviksi usein eri alojen algoritmisissa ratkaisuissa. Tällaiset suunnittelumenetelmät sopivat monen eri alan ongelmien ratkomiseen.

Itse algoritmin kuvaamiseen voidaan käyttää eri tapoja. Eräs usein käytetty tapa on sanallinen askel askeleelta kuvaus. Tämän heikkous on riittävän tarkkuuden löytäminen. Toinen suosittu tapa on pseudokoodin käyttö. Siinä pyritään käyttämään mahdollisen lähellä ohjelmointikieltä olevaa esitystapaa, mutta sallitaan myös vapaa sanallinen kuvaus.

Kun algoritmi on saatu valmiiksi, niin se tulisi osoittaa oikeaksi. Algoritmi on myös analysoitava, tärkein lienen tehokkuus suoritusajan suhteen, mutta myös muistitilan käyttö tai jonkin muun resurssin käyttö voi asettaa rajoituksia. Likiarvoisen, approksimatiivisen ratkaisun tuottaville algoritmeille tulisi tarkastella niiden antamien ratkaisujen hyvyyksiä. Voidaan myös pyrkiä tutkimaan koko algoritmin optimaalisuutta.

Jos algoritmi on tarkoitus toteuttaa tietokoneella, niin se tulee lisäksi koodata ohjelmaksi. Tähän liittyy kaikki laadukkaiden ohjelmien tekoon liittyvät vaiheet.

Jatkossa esitetään erilaisia algoritmien suunnittelumenetelmiä. Huomattava, että usein algoritmi voidaan katsoa kuuluvan useankin eri suunnittelumenetelmän mukaiseksi. Lisäksi kaikkien algoritmien ei voida katsoa kuuluvan mihinkään eri suunnittelumenetelmäluokkaa. Monesti kaikkein paras algoritmi käyttää paljon ongelman alaan kuuluvaa erikoistietoa.

3.1 Raa'an voiman käyttö

Raa'an voiman käyttö perustuu ongelman suoraviivaiseen ratkaisemiseen. Siinä tutkitaan yleensä kaikkia mahdollisia ratkaisuvaihtoehtoja ja valitaan niistä paras. Menetelmän algoritmit ovat yksinkertaisia ja ehkäpä helpoimman tuntuisia.

Jos ongelmana on laskea a^n, missä n on positiivinen kokonaisluku niin määritelmän

$$a^n = \prod_{i=1}^{n} a = \underbrace{a * a * \cdots * a}_{n \text{ kertaa}}$$

mukaan voidaan kirjoittaa algoritmi, joka muodostuu n kierrosta suoritettavasta silmukasta. Tämä suoraan määritelmän mukaisen algoritmin aikavaativuus on selvästi $O(n)$. Tämä voidaan kuitenkin laskea paljon nopeamminkin.

Tarkastellaan seuraavaksi paria esimerkkiä järjestämisongelman ratkaisemisesta. Olkoon taulukko, missä on n samaa tyyppiä olevaa tietuetta. Tietueiden kesken on annettu järjestys, joka tavallisimmin määritellään yhden tai useamman avaintiedon perusteella. Olkoot taulukon tietueet aluksi mielivaltaisessa järjestyksessä. Järjestämistehtävässä yleensä taulukon tietueiden järjestys vaihdetaan sellaiseksi, että taulukossa peräkkäiset tietueet ovat avainkentän mukaan halutun mukaisessa, esimerkiksi kasvavassa, järjestyksessä.

Tarkastellaan vain avainkentän sisältäviä tietueita. Olkoot tietueiden R_i ja R_j alkuperäiset paikat taulukossa $[i_0]$ ja $[j_0]$ ja $R_i = R_j$. Jos kaikilla tällaisilla tietuepareilla aina kun $i_0 < j_0$ seuraa, että $i < j$, on järjestämismenetelmä *stabiili*. Tällöin siis ne tietueet, joilla on sama (avainkentän) arvo, säilyttävät alkuperäisen järjestyksen mukaiset paikkansa toistensa suhteen.

Paitsi taulukoissa, tarvitaan järjestämistehtävää myös linkitetyissä listoissa. Kun järjestäminen tapahtuu kokonaisuudessa keskusmuistissa, niin sitä sanotaan sisäiseksi järjestämiseksi. Lisäksi voidaan järjestää tiedostoissa olevia tietoja ja silloin puhutaan ulkoisesta järjestämisestä. Tässä tutkitaan vain sisäistä järjestämistä.

Jos taulukon tietueet ovat suuria, voi olla edullista antaa järjestys indeksitaulukon tai referenssitaulukon avulla. Jos esimerkiksi käytetään indeksitaulukkoa *xt*, niin alustetaan se ensin kokonaislukuarvoihin *xt[i] = i (i = 0, … , n-1)*. Sitten indeksitaulukko järjestetään siten, että jos $i < j$, niin indeksien *xt[i]* ja *xt[j]* ilmoittamat tietueet ovat vastaavassa järjestyksessä. Tässä on se etu, että ei tarvitse siirrellä suuria tietueita, vaan lyhyitä indeksejä tai referenssejä. Edelleen samalle tietuetaulukolle (tiedostolle) voidaan antaa useampia järjestyksiä samanaikaisesti käyttämällä useaa indeksitaulukkoa.

Linkitettyjen listojen alkioille voidaan järjestys tehdä joko linkityksiä muuttamalla tai sitten referenssitaulukon avulla.

Järjestämistä kutsutaan usein lajitteluksi (sorting). Tämä johtuu siitä, että ennen reikäkortit laitettiin järjestykseen reikäkorttilajittelijalla (sorter).

Järjestäminen on eräs tärkeimpiä ohjelmointiongelmia. Järjestämistä tarvitaan itse järjestyksen luomisen takia, mutta sen avulla voidaan myös nopeuttaa useita muita algoritmeja. Erilaisia järjestämisalgoritmeja on erittäin paljon, toiset tehokkaampia kuin toiset.

Kuplamenetelmä

Tämä on erittäin yksinkertainen ja nimenomaan raakaan voimaan perustuva menetelmä.

Kuplamenetelmä

1) Olkoon taulukossa *t* alkioita *n* kappaletta. Suoritetaan kohta 2 indeksin *i* arvoilla $i = 1, \ldots, n-1$.
2) Käydään läpi tietueita $t[j], j = n-2, \ldots, i-1$ seuraavasti:
   Jos $t[j] > t[j+1]$ niin vaihdetaan tietueiden $t[j]$ ja $t[j+1]$ paikkaa.

Arvioidaan kuplamenetelmän vaativuutta. Kuplamenetelmässä suoritetaan vertailuja aina sama määrä eli $(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2$ kappaletta. Pahimmassa tapauksessa jokainen vertailu voi johtaa myös tietueiden vaihtoon, jolloin myös niitä suoritetaan yhtä paljon. Kuplamenetelmän aikavaativuus on siten $O(n^2)$.

Väliinsijoitusmenetelmä

Tämä toinen yksinkertainen menetelmä selvitettiin ja analysoitiin jo luentomonisteen alussa. Se on hidas suurilla taulukoilla. Se on kuitenkin varteenotettava, kun taulukossa on vähän tietueita.

Väliinsijoitusmenetelmä

1) Olkoon taulukossa *t* alkioita *n* kappaletta. Suoritetaan kohta 2 indeksin *i* arvoilla $i = 1, \ldots, n-1$. Aina ennen kohdan 2 suoritusta oletetaan, että taulukossa *t* tietueet $t[0], \ldots, t[i]$ ovat jo toisiinsa nähden kasvavassa järjestyksessä. Todetaan, että tämä on ilman muuta tosi, kun $i = 0$.
2) Kopioidaan aputietueeseen *p* tietue $t[i]$. Sitten käydään läpi tietueita $t[j], j = i-1, i-2, \ldots, 1, 0$ tarvittaessa seuraavasti:
   Jos $j \geq 0$ ja $p < t[j]$, kopioidaan tietue $t[j]$ seuraavaan taulukon positioon $j+1$ ja siirrytään seuraavaan arvoon $j = j-1$.
   Muussa tapauksessa kopioidaan tietue *p* positioon $j+1$ ja lopetetaan tietueiden siirtely.

Kohdan 2 periaatteena on siis, että ennen sitä tietueet positioissa $0, \ldots, i-1$ ovat toisiinsa nähden järjestyksessä. Sitten tietue $t[i]$ siirretään syrjään. Siitä alkuun päin

siirretään tietueita eteenpäin yhdellä askeleella kunnes tullaan kohtaan, mihin ennen kohdassa *i* ollut tietue voidaan sijoittaa, jotta taulukon alkupään järjestys olisi oikein. Tämän jälkeen tietueet positioissa *0, …, i* ovat toisiinsa nähden järjestyksessä.

Arvioidaan väliinsijoitusmenetelmän vaativuutta. Pahin mahdollinen tilanne on, kun taulukko on aluksi vähenevässä järjestyksessä. Silloin kohta 2 suoritetaan joka kerta positioon 0 saakka. Pääkierroksella *i* tarvitaan siis *i* perusaskelta (avainten vertailu ja tietueen kopiointi). Lisäksi tarvitaan kaksi ylimääräistä kopiointia. Perusaskeleiden määrä on siten $1 + 2 + \ldots + n\text{-}1 = n(n\text{-}1)/2$. Näin ollen avainten vertailuja on pahimmillaan $n(n\text{-}1)/2$ ja tietueiden kopiointeja $(n\text{-}1)(n/2 + 2)$. Todetaan, että korkein termi lausekkeissa on $n^2/2$. Väliinsijoitusmenetelmän aikavaativuus on siis $O(n^2)$.

Valintamenetelmä

Tämäkin on toinen yksinkertainen järjestämismenetelmämenetelmä jota myös voidaan pitää raa'an voiman menetelmänä.

Valintamenetelmä

1) Olkoon taulukossa *t* alkioita *n* kappaletta. Suoritetaan kohtia 2 ja 3 indeksin *i* arvoilla $i = 0, \ldots, n\text{-}2$.
2) Kopioidaan aputietueeseen $p = t[i]$ ja asetetaan $k = i$. Sitten käydään läpi tietueita $t[j], j = i+1, i+2, \ldots, n\text{-}1$ seuraavasti:
   Jos $p < t[j]$, kopioidaan $p = t[j]$ ja asetetaan $k = j$.
3) Jos $k \neq i$ niin kopioi tietue $t[i]$ tietueeksi $t[k]$ ja kopioi tietue *p* tietueeksi $t[i]$.

Periaatteena on siis, että tietueet positioissa $0, 1, \ldots, i\text{-}1$ ovat jo lopullisesti oikeissa paikoissaan. Kohdassa 2 siis selvitetään taulukon loppuosan $i, \ldots, n$ tietueista pienin ja kohdassa 3 sijoitetaan tarvittaessa tämä tietue paikkaan *i*.

Arvioidaan valintamenetelmän vaativuutta. Vertailuja suoritetaan aina sama määrä eli $1 + 2 + \ldots + n\text{-}1 = n(n\text{-}1)/2$ eli $O(n^2)$. Tietueiden siirtojen suhteen huono tilanne on, kun taulukko on aluksi vähenevässä järjestyksessä. Silloin siirtoja voidaan joutua suorittamaan kertaluokan $O(n^2)$ verran. Siirtojen määrää voidaan kuitenkin vähentää, jos kohdassa 2 ei kopioida tietuetta, vaan pidetään ainoastaan yllä indeksiä *k* ja korvataan viittaus tietueeseen *p* viittauksella tietueeseen $t[k]$. Tällöin siirtoja tuleekin pahimmassa tapauksessa enintään kertaluokan $O(n)$ verran.

Lähin pistepari

Olkoon meillä annettuna *n* tason pistettä $p_i = (x_i, y_i), i = 1, \ldots, n$. Ongelmana on löytää kaksi toisiaan lähinnä olevaa pistettä. Pisteiden välinen etäisyys lasketaan tavallisena Euklidisena etäisyytenä eli $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Raa'an voiman algoritmi käy läpi kaikki pisteparit, laskee niiden välisen etäisyyden ja pitää yllä pienintä etäisyyttä.

Lähin pistepari

1) Olkoon taulukossa *t* pistepareja (x_i, y_i) *n* kappaletta.
   Aseta *pienin* = ∞.
   Suorita askel 2 indeksin *i* arvoilla *i = 0, … , n-2*.
2) Suorita seuraava indeksin *j* arvoilla *j = i+1, … , n-1*.
   Laske $e = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
   Jos *e* < *pienin*,
   niin *pienin = e, p1 = i* ja *p2 = j*.
3) Palauta pisteparin taulukkoindeksit *p1* ja *p2*.

Algoritmin perusoperaationa voidaan pitää kahden pisteen välisen etäisyyden laskemista. Itse asiassa neliöjuuren laskeminenkaan ei ole aivan helppo laskutoimenpide. Mutta se voidaankin välttää tässä tapauksessa. Perusoperaationa voidaan pitää luvun toisen potenssin laskemista. Tällaisia laskutoimenpiteitä suoritetaan *(n-1) + (n-2) + … + 2 + 1 = n(n-1)/2* kappaletta. Täten tämän menetelmän aikavaativuus on $O(n^2)$.

Raa'an voiman algoritmit perustuvat tietokoneen laskentatehon hyväksikäyttöön. Usein ne tutkivat mahdollisia ratkaisuvaihtoehtoja niin pitkälle kunnes joko haettu ratkaisu tai kaikki ratkaisuvaihtoehdot on tutkittu. Ratkaisun esitystavasta johtuen tämä voi tarkoittaa kaikkien osajoukkojen tai permutaatioiden tutkimista. Jos halutaan minimoida ylimääräisen työn määrää, niin tämä tulee tehdä järjestelmällisesti. Suurin ongelma raa'an voiman menetelmissä on ratkaisuvaihtoehtojen suuri määrä. Se estää näiden menetelmien käytön vähänkään suuremmille ongelmille. Esimerkkinä esitetyille järjestämisongelmalle ja myös lähimmän pisteparin ongelmalle löytyy tehokkaampiakin algoritmeja.

Kaikki permutaatiot

Kaikkien ongelman ratkaisuvaihtoehtojen tutkiminen on selvästi raa'an voiman menetelmä. Usein ongelman ratkaisu voidaan esittää peräkkäisten kokonaislukujen *1, 2, …, n* avulla. Yksi mahdollisuus ratkaisun esittämiseen näiden kokonaislukujen muodostama jono.

Olkoon meillä ongelma, jonka syöttötiedon koko on *n*. Edelleen siis oletetaan, että ongelman ratkaisut voidaan esittää kokonaislukujen *{1,2, …, n}* permutaationa, eli näiden kokonaislukujen erilaisina järjestyksinä. Tällaisia eri järjestyksiä on olemassa erittäin paljon eli kaikkiaan *n! = n*(n-1)*(n-2)*…*2*1* kappaletta.

Eräs tällainen ongelma on Hamiltonin kehä-ongelma. Annettuna on verkko; solmujen joukko *N = {1, 2, …, n}* ja verkossa olevien kaarien joukko $E = \{(i, j) | i \in N$ ja $j \in N\}$. Ongelmana on löytää sellainen *n*:n kaaren joukko, joka muodostaa silmukan verkon kaikkien *n*:n solmun kautta siten, että silmukka käy kussakin solmussa kerran ja vain kerran.

Hamiltonin kehäongelman ratkaisu voidaan esittää verkon solmujen permutaatioratkaisuja tutkimalla. Permutaatioratkaisu *{1,2, …, n}* tarkoittaisi

Hamiltonin kehää $1 \rightarrow 2 \rightarrow \ldots \rightarrow n \rightarrow 1$. Kun permutaatioratkaisu on muodostettu, niin tulee vielä tutkia, onko tarkasteltavassa verkossa olemassa kaikki ratkaisussa esitetyt kaaret.

Täten ongelma voidaan ratkaista raakaan laskentaan perustuvalla algoritmilla, jossa muodostetaan kaikki mahdolliset *n*:n eri solmun renkaat eli kaikki lukujen *{1,2, …, n}* permutaatiot ja tutkimalla vastaavan permutaatioratkaisun sallittavuus.

Kaikkien permutaatioiden muodostaminen sujuu helposti rekursion avulla. Oletetaan, että osaamme muodostaa kaikki permutaatiot *n-1*:lle kokonaisluvulle. Silloin voimme laajentaa menetelmää *n*:lle kokonaisluvulle *{1, 2, …, n}* seuraavasti.

Kiinnitetään luku 1 seuraavaksi vuorossa olevien permutaatioiden ensimmäiseksi luvuksi ja muodostetaan osaamallamme menetelmällä kaikki lukujen *{2, 3, …, n}* permutaatiot.

Seuraavaksi kiinnitetään kokonaisluku 2 vuorossa olevien permutaatioiden ensimmäiseksi luvuksi ja muodostetaan osaamallamme menetelmällä kaikki lukujen *{1, 3, …, n}* permutaatiot.

Tätä toistetaan kunnes lopulta kiinnitetään kokonaisluku *n* vuorossa olevien permutaatioiden ensimmäiseksi luvuksi ja muodostetaan osaamallamme menetelmällä kaikki lukujen *{1, 2, …, n-1}* permutaatiot.

Seuraava algoritmi muodostaa permutaatiot taulukkoon *p[1,n]* esitetyllä tavalla.

Algoritmi:

```
// muodostetaan kaikki lukujen {1, 2, …, n} permutaatiot.
permutaatio(m)
{
  if (m == n)
    seuraava permutaatio on nyt taulukossa p[1..n]
  else
    for (kaikilla arvoilla j arvosta m arvoon n) {
      swap(p[j], p[m]);
      permutaatio(m+1);
      swap(p[j], p[m]);
    }
}
```

Ensin tulee vielä alustaa taulukko *p* ja tehdä ensimmäinen kutsu seuraavasti.

```
for (kaikilla arvoilla j arvosta 1 arvoon n)
  p[j] = j;
permutaatio(1);
```

Koska *n*:n alkion permutaatioita on *n!* kappaletta, niin algoritmin tietysti tulee muodostaa ne kaikki. Kullekin permutaatiolle sitten tehdään ensimmäisessä **if**-osassa jotain joka vieköön ajan *O(n)*. Tällöin koko algoritmin aikavaativuudeksi voidaan laskea *O(n n!)*.

3.2 Osittaminen

Osittaminen eli hajoita-ja-hallitse periaate on ehkä parhaiten tunnettu algoritmien suunnittelutekniikka.

Idea

- Ongelman esiintymä ositetaan saman ongelman useaksi pienemmäksi esiintymäksi. Osaongelmat ovat yleensä keskenään suunnilleen samankokoisia.
- Osaongelmat ratkaistaan, yleensä rekursiivisesti.
- Tarvittaessa osaongelmien ratkaisuista kootaan alkuperäisen koko ongelman ratkaisu.

Yleinen muoto:

```
vastaus osittava (tapaus x)
{
  if (x on ''pieni'')
    y = pienen tapauksen ratkaisu;
  else {
    jaa x osiin x1, …, xm;
    // osat yleensä tasakokoisia, ei välttämättä erillisiä
    y1:=osittava(x1);
    …
    ym:=osittava(xm);
    kokoa x:n vastaus y osista y1, ..., ym;
  }
  return y;
}
```

Lomitusjärjestämismenetelmä

Jo luentomonisteen alussa esiteltiin raa'an voiman menetelmiä tehokkaampi järjestämismenetelmä; lomitusmenetelmä.

```
// Lomitusmenetelmä
MergeSort(t, l, h)}
{
  if (l < h) {
    k = (l+h)/2;
    MergeSort(t, l, k);
    MergeSort(t, k+1, h);
    Merge(t, l, k, h);
  }
}
```

Jos *T(n)* on algoritmin suoritusaika *n*:n alkion syöttötiedon koolla, niin voidaan kirjoittaa rekursiivinen yhtälö

$$T(n) = \begin{cases} c, & \text{kun } l \geq h \\ 2T(n/2) + dn, & \text{kun } l < h \end{cases}$$

Tästä voidaan ratkaista *T(n)* useammallakin eri tavalla.

Eräs mahdollisuus on pyrkiä purkamaan rekursiota. Oletetaan, että alkioiden lukumäärä $n = 2^k$. Tällöin taulukko jakautuu aina kahteen yhtä suureen osaan eli esimerkiksi ensimmäisellä kerralla $n/2 = 2^k/2 = 2^{k-1}$.

Soveltamalla rekursiokaavaa aina uudestaan saadaan

$$\begin{aligned} T(n) &= 2T(n/2) + dn = 2^1 T(2^{k-1}) + 1dn \\ &= 2(2T(2^{k-1}/2) + d(n/2)) + dn = 2^2 T(2^{k-2}) + 2dn \\ &= 2^2(2T(2^{k-2}/2) + d(n/4)) + 2dn = 2^3 T(2^{k-3}) + 3dn \\ &= \cdots \\ &= 2^k T(n/2^k) + kdn \\ &= nT(1) + kdn \\ &= nc + kdn. \end{aligned}$$

Tässä on vielä selvitettävä *k*n arvo ja koska $n = 2^k$ niin saadaan $\log_2 n = k$. Täten saadaan lopputulos $T(n) = nc + dn \log_2 n$ eli lomitusjärjestämisen aikavaativuus on pahimmassa tapauksessa kertaluokkaa $O(n \log n)$.

Toinen mahdollisuus on arvata lopputulos ja todistaa se oikeaksi induktiolla. Kolmas mahdollisuus rekursiivisten, muotoa $T(n) = aT(n/b) + f(n)$, olevien yhtälöiden ratkaisemiseen on käyttää ns. master-lausetta, joka antaa kaavana suoraan ratkaisun. Lausetta ei kuitenkaan esitetä tässä. Lisäksi voidaan käyttää erilaisia matemaattisia menetelmiä.

Suurten kokonaislukujen kertolasku

Ongelma: Annettu *n*-bittiset kokonaisluvut $X, Y \geq 0$ ($n = 2^k$, $k = 0, 1, 2, \ldots$). Laskettava tulo *XY*.

”Peruskoulualgoritmi”: Kerrotaan lukuja ”numero” kerrallaan. Tällöin tarvitaan $T(n)=O(n^2)$ bittioperaatiota.

(*Huom*. Kaksi *n*-bittistä lukua voidaan laskea yhteen $O(n)$ bittioperaatiolla.)

Osittava ratkaisu:

Merkitään
$X=A\cdot 2^{n/2}+B$ ja
$Y=C\cdot 2^{n/2}+D$,
missä *A, B, C* ja *D* ovat *n/2* bitin pituisia.

Algoritmi 1:

$XY = AC \cdot 2^n + (AD+BC)\, 2^{n/2} + BD$

Tällöin saadaan

$$T(n) = \begin{cases} c_1, & \text{kun } n = 1 \\ 4T(\frac{n}{2}) + c_2 n, & \text{kun } n = 2^k, k = 0,1,... \end{cases}$$

Tämän ratkaisuksi voidaan esimerkiksi master-lausetta soveltaen saada $T(n)=O(n^{\log 4})=O(n^2)$, siis ei parannusta.

Algoritmi 2 (Karatsuba & Ofman 1962): Käytetään hyväksi seuraavaa kaavaa.

$XY=AC \cdot 2^n + [(A-B)(D-C)+AC+BD] \cdot 2^{n/2} + BD$

Tarvitaan siis 3 kpl *n/2* bitin kertolaskua, 4 kpl *n* bitin yhteenlaskua, 2 kpl *n/2* bitin vähennyslaskua ja 2 sivuttaissiirtoa.

$$T(n) = \begin{cases} c_1, & \text{kun } n = 1 \\ 3T(\frac{n}{2}) + c_2 n, & \text{kun } n = 2^k, k = 0,1,... \end{cases}$$

Nyt vastaavasti esimerkiksi master-lauseeen avulla saadaan, että $T(n) = O(n^{\log 3}) = O(n^{1,59...})$.

Huomautuksia:

- Aikavaativuusfunktioiden vakiotekijöiden takia peruskoulualgoritmi on itse asiassa K & O -algoritmia parempi noin 500-bittisiin lukuihin asti. Toisaalta näin suuria ja suurempiakin lukuja tarvitaan nykyisin mm. salakirjoitusjärjestelmissä rutiininomaisesti.
- Teoriassa hyvin suurten lukujen kertolasku on mahdollista suorittaa vielä nopeammin eli ajassa $O(n \log n \operatorname{loglog} n)$ (ns. Schönhage - Strassen -algoritmi). On avoin ongelma, voidaanko kertolasku toteuttaa ajassa $O(n)$.
- Toteutuksesta: Käytännössä lukujen osittamista ei kannata jatkaa 1-bittisiin saakka, vaan vain käytettävän tietokoneen sanapituuteen asti, jolloin osien kertolasku voidaan toteuttaa yhdellä konekäskyllä.

Quicksort eli pikajärjestämismenetelmä

Erilaisia järjestämismenetelmiä on hyvin paljon. Tärkein niistä on Quicksort (Hoare 1960) eli pikajärjestämismenetelmä. Se on monissa kokeissa todettu yleensä tehokkaimmaksi yleiskäyttöiseksi taulukon järjestämismenetelmäksi. Pikamenetelmä perustuu osittamiseen (divide and conquer) ja siten myös rekursioon.

- Jos taulukon pituus on tarpeeksi pieni, järjestetään se väliinsijoitusmenetelmällä. Myöhemmin esitettävän algoritmien kannalta on oleellista, että väliinsijoitusmenetelmää käytetään ainakin silloin, kun taulukon pituus on < 3.

- Valitaan jokin taulukon *t* (*n* tietuetta) tietue *p* (jakotietue).
- Ositusvaihe (partition). Siirrellään taulukon tietueita siten, että tämän vaiheen jälkeen jakotietue on positiossa *j*. Paikka *j* määräytyy siten, että ositusvaiheen siirtojen jälkeen tietueet, jotka edeltävät jakotietuetta, ovat ennen positiota *j*. Tietueet, jotka seuraavat jakotietuetta ovat position *j* jälkeen. Tietueet, joilla on sama avain kuin jakotietueella, voivat olla molemmilla puolilla.
- Järjestetään taulukon alkuosa *0, 1, … ,j-1* ja loppuosa *j+1, j+2, … , n-1* väliinsijoitusmenetelmällä, mikäli osan pituus on suurempi kuin 1, mutta pienempi kuin *v*. Muussa tapauksessa järjestäminen tehdään rekursiivisesti quicksort-menetelmällä. Raja *v* vaihtelee alkuperäisen taulukon koosta riippuen. Yleensä *v < 100*.

Periaatteena on siis, että taulukko jaetaan kahteen lyhyempään pätkään, jotka voidaan järjestää erikseen toisistaan riippumatta. Osien järjestämiseksi aliohjelma kutsuu itseään.

Jakotietueen valinta

Seuraavia menetelmiä on käytetty:

- Valitaan taulukon ensimmäinen tietue. Jos taulukko on satunnaisessa järjestyksessä, tämä kyllä käy. Jos kuitenkin taulukko on jo järjestyksessä, tulee jakotietue siirtelyn jälkeen taulukon positioon 0. Silloin taulukon loppuosan pituus olisi *n-1*. Täten jako kahteen osaan tulee erittäin epätasaiseksi ja se johtaa tehottomuuteen.
- Valitaan jakotietue satunnaisesti. Silloin jakotietueeksi voi tulla taulukon osan suurin tai pienin tietue todennäköisyydellä *2/n*. Täten epäonnistumisen todennäköisyys ei ole kovin suuri.
- Jakotietueeksi valitaan keskimmäinen tietue. Tällöin menetelmä ei voi epäonnistua, jos taulukko on järjestetty. Muuten epäonnistumisen todennäköisyys on sama kuin edellä.
- Mediaanimenetelmä. Silloin jakotietueeksi valitaan suuruudeltaan keskimmäinen taulukon ensimmäisestä, keskimmäisestä ja viimeisestä tietueesta (median of three).
- Muodostetaan yhdeksän tietueen pseudomediaani. Sitä varten valitaan taulukosta tasavälisesti yhdeksän tietuetta. Valitaan seuraavaksi näistä kolmen ensimmäisen tietueen mediaanialkio, sitten kolmen seuraavan ja myös vielä kolmen viimeisen mediaanialkio. Lopulliseksi jakotietueeksi valitaan näiden kolmen mediaanialkion mediaani. Tätä menetelmää käytetään, kun taulukko on riittävän suuri. Kokeet ovat osoittaneet, että 40 tietuetta on sopiva raja.

Taulukon jakaminen (partition)

Olkoon taulukossa *t* tietueita *n (n > 0)* kappaletta. Parametri *l* ilmoittaa tarkasteltavan taulukon osan vasemmanpuoleisimman alkion indeksin ja parametri *r* vastaavasti oikeanpuoleisimman alkion indeksin. Jakotietueeksi on saatu tavalla tai toisella *t[k] = p*. Taulukon jakaminen voidaan suorittaa seuraavasti (erilaisia versioita on kymmeniä).

```
int partition(int t[], int l, int r, int k)
{
  SWAP(l, k); // Jakotietue vaihdetaan taulukon alkuun
  for (i = l, j = r+1; ; ) {
    do ++i; while (i < r && t[l].key > t[i].key});
    do --j; while (t[l].key < t[j].key);
    if (j < i) break;
    SWAP(i, j); // Vaihdetaan tietueet t[i] ja t[j]
  }
  SWAP(l, j);    // Jakotietue lopulliselle paikalleen
  return j;
}
```

Tämän jälkeen järjestetään taulukon osat *[l..j-1]* ja *[j+1..r]* erikseen. Tätä varten aliohjelma kutsuu itseään rekursiivisesti.

Tarkastellaan erikoistapauksia.

- Olkoot kaikki tietueet pienempiä kuin jakotietue. Silloin ensimmäisessä **do**-silmukassa *i* kasvaa kunnes tullaan taulukon tarkasteltavan osan loppuun. Silloin toinen **do**-silmukka loppuu yhden kierroksen jälkeen tilanteeseen, missä *j = r-1* ja *i = r*, jolloin myös **for**-silmukka loppuu. Silloin jakotietue, joka myös on taulukon suurin tietue, siirretään lopuksi paikkaan *j = r-1*. Se tulee oikealle paikalleen. Taulukon alkuosa järjestetään sitten.
- Jos kaikki tietueet ovat suurempia kuin jakotietue, päättyy algoritmi tilanteessa, jolloin *i = l+1* ja *j = l*. Nytkin saadaan vain jakotietue oikealle paikalleen.

Kokonaisuudessaan pikajärjestäminen toimii seuraavasti

```
void quicksort(int[] t, int l, int r)
{
  if (r-l < alaraja)
    // järjestä väliinsijoitumenetelmällä
  else {
    k = pivot(t, l, r); // valitaan jakotietue
    j = partition(t, l, r, k);
    quicksort(t, l, j-1);
    quicksort(t, j+1, r);
  }
}
```

Pikajärjestämismenetelmän analysointia

Pikajärjestämismenetelmän tehokkuutta tarkasteltaessa jakotietueen valinnassa onnistuminen on varsin oleellista. Pahin tapaus on sellainen, jossa jako kahteen pienempään osaan tapahtuu mahdollisimman epätasaisesti. Olkoon esimerkiksi jakotietueen valinta sellainen, että valinnan tuloksena jakotietueeksi tulee avaimen arvoltaan pienin tietue. Silloin jakotietueen vasemmalle puolelle ei tule yhtään tietuetta ja oikealle puolelle tulee *n-1* tietuetta. Jos *T(n)* on pikajärjestämismenetelmän pahimman tapauksen suoritusaika, niin saadaan rekursiivinen yhtälö *T(n) = T(n-1) +*

cn. Jos sama toistuisi myös jatkossa, niin suoritusajaksi saadaan $T(n) = T(1) + c2 + c3 + c4 + \ldots + cn \leq cn(n+1)/2 = O(n^2)$.

Jos taasen jakotietueen valinta onnistuu aina siten, että alkioiden lukumäärä puolittuisi, niin suoritusajalle saataisiin rekursioyhtälö $T(n) = 2T(n/2) + n$, joka on tuttua muotoa lomitusmenetelmän yhteydestä eli tällöin suoritusaika olisi $O(n \log n)$.

Voidaan lisäksi osoittaa, että keskimäärin pikajärjestämismenetelmän suoritusajan vaativuus on $O(n \log n)$. Tämän todistuksessa oletetaan, että jokainen järjestettävä tietue voidaan valita jakotietueeksi samalla todennäköisyydellä. Jätetään tämän todistus väliin.

Osittamismenetelmän osittamisen tasapainoisuudesta

Yleensä ongelman tapaus kannattaa osittaa siten, että osatapaukset ovat mahdollisimman samankokoisia. Tämä voitiin havaita jo pikajärjestämismenetelmän yhteydessä.

Esimerkki: Lukujoukon mediaanin määrittäminen.

Ongelma: Annettu kokonaislukujoukko S, $|S|=n$, ja luku k. Määrättävä S:n k:nneksi pienin alkio. (Erityisesti mediaani: $k=\lceil n/2 \rceil$).

Triviaaliratkaisuja:

1) Määrätään järjestyksessä k pienintä alkiota: $T(n) = O(k \cdot n) = O(n^2)$.
2) Järjestetään S ja poimitaan k:nneksi pienin: $T(n) = O(n \log n)$.

Osittava ratkaisu:

```
alkio select(joukko S, int k)
{
  if (|S| == 1)
    return a;
  else {
    a = ''satunnainen'' S:n alkio;
    S1 = {x ∈ S|x<a}; // Oletetaan yksink. vuoksi
    S2 = {x ∈ S|x>a}; // että S:n alkiot erilliset
    if (|S1| ≥ k)
      return select(S1, k);
    else if (|S1| == k-1)
      return a;
    else
      return select(S2, k-| S1|-1);
  }
}
```

''Analyysi'': Jos jakoalkio *a* tulee aina valituksi "suunnilleen keskeltä", niin

$$T(n) \approx \begin{cases} c_1, & \text{kun } n = 1, \\ T(\frac{n}{2}) + c_2 n, & \text{kun } n > 1. \end{cases}$$

Ja tällöin saadaan $T(n) \approx O(n)$.

Jos jakoalkio tulee valittua äärimmäisen huonosti eli rekursiivisen kutsun osassa joukkoon jää aina vain yksi alkio vähemmän niin tuloksena on, että

$$T(n) \approx \begin{cases} c_1, & \text{kun } n = 1, \\ T(n-1) + c_2 n, & \text{kun } n > 1. \end{cases}$$

ja tällöin $T(n) \approx O(n^2)$.

Valitsemalla jakoalkio huolellisesti voidaan taata, että myös pahimmassa tapauksessa algoritmin vaativuus on $O(n)$

Järjestämisongelman alaraja

Yllä on esitetty useita ratkaisumenetelmiä järjestämistehtävälle. On esitetty menetelmiä, jotka vaativat joko pahimmassa tai keskimääräisessä tapauksessa ajan $O(n \log n)$. Voidaankin kysyä, että onko parempia menetelmiä yleensä olemassakaan?

Esitetyt menetelmät ovat perustuneet siihen, että järjestäminen on suoritettu aina kahden tietueen avainten arvoja vertaamalla. Tällaiset menetelmät ovat *vertailumenetelmiä*. Tällaisia ovat siis kaikki tähän asti esitetyt järjestämismenetelmät. Vertailujärjestämisessä verrataan kahden tietueen avainkenttiä, esimerkiksi kysytään onko $t_i < t_j$, ja saadaan vastaus kyllä tai ei. Tämän vastauksen perusteella menetelmä voi sitten tehdä joitain toimenpiteitä ja lopulta suorittaa uudelleen vastaavanlaisen vertailun. Täten tällainen vertailuun perustuva järjestämismenetelmä voidaan aina esittää (binäärisenä) päätöspuuna. Tämän päätöspuun jokainen sisäinen solmu, siis ei-lehtisolmu, vastaa yhtä vertailua. Sisäisellä solmulla on aina kaksi lasta vertailun kummankin tuloksen mukaisesti.

Minkä tahansa kahden tietueen vertailuun perustuvan järjestämismenetelmän käyttäytyminen voidaan esittää tällaisena päätöspuuna. Huomattava, että itse algoritmin ei tarvitse tietää mitään tällaisen päätöspuun olemassaolosta. Päätöspuu vain esittää kaikki mahdolliset vertailujen jonot, jotka algoritmi suorituksen aikana käy läpi. Algoritmi aloittaa juurisolmusta ja vertailun tuloksen mukaan jatkaa joko vasemmasta tai oikeasta haarasta ja jatkaa näin puuta eteenpäin päätyen johonkin lehtisolmuun, kun algoritmi lopettaa suorituksen ja on suorittanut järjestämisen valmiiksi.

Täten binäärisen päätöspuun korkeus kuvaa pisintä mahdollista polkua, jonka algoritmi suorittaa eli suurinta määrää vertailuja, joka voidaan tehdä algoritmin suorituksen aikana.

Jos ajatellaan mitä tahansa vertailumenetelmää järjestämisongelmalle, niin sen tulee pystyä järjestämään kaikki mahdolliset syöttöjärjestykset. Näitä erilaisia syöttöjärjestyksiä eli erilaisia lukujen *1, 2, …, n* permutaatioita on *n!* kappaletta. Täten

algoritmin päätöspuulla tulee olla ainakin *n!* lehtisolmua. Koska binääripuun kullakin solmulla voi olla korkeintaan 2 lasta niin jos binääripuulla *n!* lehtisolmua niin binääripuun korkeus on $\geq \lceil \log n! \rceil \geq \log n! > \sum_{i=1}^{n} \log i \geq \sum_{i=1}^{n/2} \log n/2 \geq n/2 \log n/2$.
Koska päätöspuun korkeus on suurempi tai yhtä suuri kertaluokka $n \log n$ niin minkä tahansa algoritmin, joka suorittaa kahden tietueen välisiä vertailuja, tulee suorittaa pahimmassa tapauksessa vähintään kertaluokan $n \log n$ verran vertailuja.

Täten esimerkiksi esitetty lomitusmenetelmä on tässä mielessä optimaalinen. On kuitenkin olemassa järjestämisongelmia, joissa avainkentän arvojen suhteen on olemassa rajoituksia ja järjestämismenetelmiä, joissa avainkenttien suhteen voidaan tehdä muutakin kuin binäärisiä vertailuja. Jos esimerkiksi avainten arvoalue on suhteellisen pieni, niin tietueen paikka voidaan selvittää suoraan laskemalla. Täten pyrittäessä mahdollisimman hyvään järjestämisalgoritmiin tulee sen valinta suhteellisen hankalaksi.

Järjestäminen laskemalla

Aikaisemmin esitetyt yksinkertaiset järjestämismenetelmät olivat pahimmassa tapauksessa kertaluokan $O(n^2)$ menetelmiä. Lisäksi tunnemme lomitusmenetelmän (mergesort), jonka aikavaativuus on $O(n \log n)$ ja pikamenetelmän (quicksort), jonka keskimääräinen aikavaativuus on $O(n \log n)$. Edelleen tiedämme, että kahden alkion vertailuun perustuvan algoritmin aikavaativuus pahimmassa tapauksessa on aina vähintään kertaluokkaa $n \log n$. Tarkastellaan tässä järjestämistä muutoin kuin vertailemalla ja esitetään menetelmiä, joilla tämä $n \log n$ aikaraja voidaan rikkoa.

Laskentamenetelmä

Aikaisemmin esitetyt järjestämismenetelmät suorittavat järjestämisen vertailun perusteella. Laskentajärjestämisessä oletetaan, että järjestettävänä on n kappaletta kokonaislukuja väliltä $0 \ldots k$.

Laskentamenetelmän perusideana on laskea kullekin järjestettävälle alkiolle a niiden alkioiden lukumäärä, jotka ovat pienempiä kuin a. Tämän tiedon perusteella alkio a voidaan sijoittaa omalle paikalleen.

Olkoot järjestettävät alkiot taulukossa *t[1..n]*. Järjestetyt alkiot ovat algoritmin päättyessä järjestyksessä toisessa taulukossa *s[1..n]* ja käytetään lisäksi työtilana kolmatta taulukkoa *u[0..k]*.

Saadaan seuraava laskentamenetelmä *n*:n tietueen järjestämiseksi, kun avainkenttien arvot ovat väliltä *0* .. *k*.

Algoritmi:

```
laskentamenetelmä alkioiden järjestämiseksi.
  1. Nollaa taulukko u.
  2. for (i = 1; i ≤ n; i++)  u[t[i]]++;
     // Tämän jälkeen u[j] sisältää alkioiden j lukumäärän.
  3. for (i = 1; i ≤ k; i++) u[i] = u[i] + u[i-1];
```

```
    // u[i] sisältää nyt tiedon siitä montako alkiota on
    // pienempiä tai yhtä suuria kuin i.
4. for (j = n; j ≥ 1; j--) s[u[t[j]]] = t[j] ja u[t[j]]--;
```

Tämän algoritmin vaativuus on helppo analysoida. Ensimmäinen silmukka eli askel 1 vie ajan *O(k)*. Askeleen 2 aikavaativuus on *O(n)*. Askeleen 3 *O(k)* ja askeleen 4 *O(n)*. Yhteensä laskentamenetelmän vaativuus pahimmassa tapauksessa on siis *O(k + n)*. Jos nyt *k = O(n)*, niin saadaan koko laskentamenetelmän vaativuudeksi *O(n)*. Näin järjestäminen voidaan suorittaa tässä erikoistapauksessa lineaarisessa ajassa alkioiden lukumäärän suhteen. Tämä menetelmä on myös stabiili.

Lokeromenetelmä (bin sort)

Lokeromenetelmässä järjestettävät alkiot pannaan lokeroihin. Yleisesti lokerossa voi olla useita alkioita ja yhdessä lokerossa on vain yhden avainarvon omaavia alkioita.

Yksinkertaisin tilanne on, kun avaimet ovat kokonaislukuja väliltä *0 ... n-1* ja kukin avain esiintyy täsmälleen yhden kerran. Tällöin lokerot voidaan esittää taulukon alkioina. Olkoon järjestettävänä taulukossa *t* olevat n alkiota taulukkoon *s*.

```
1. Nollaa taulukko s;
2. for (i = 0; i < n; i++) s[t[i]] = t[i];
```

Nyt alkuperäiset alkiot ovat kasvavassa järjestyksessä taulukossa *s*. Menetelmän aikavaativuus on selvästi *O(n)*.

Yleisemmässä tapauksessa sama avain voi esiintyä useamminkin kuin kerran tai ei lainkaan ja samaan lokeroon voi tulla useampia alkioita. Silloin lokeroa ei voida toteuttaa taulukon alkiona. Lisäksi mukana voi olla tyhjiä lokeroita, jolloin meidän tulee myös koota lopputulos eri lokeroista.

Olkoon meillä *n* tietuetta a_i, joiden avainkentän arvot, $a_i.avain$, ovat väliltä *[l - h]* ja merkitään *m = h - l + 1*. Järjestetään nämä alkiot listaan *l*.

Algoritmi:

```
lokeromenetelmä (bin sort)
1. for (i = 0; i < n; i++)
       Lisää alkio a_i lokeroon a_i.avain.
2. Luo tyhjä lista lis.
3. for (j = l; j <= h; j++)
       Lisää listan lis loppuun lokeron j alkiot.
```

Algoritmin päättyessä alkiot ovat järjestettyinä listassa *lis*. Algoritmin aikavaativuus on selvästi *O(n+m)*, jos operaatiot voidaan tehdä vakioajassa. Jos lokeroiden talletustapa on linkitetty lista, niin tämä aikavaativuus on saavutettavissa.

Lukujärjestelmäjärjestäminen (radix sort)

Mahdollisten avainkentän eri arvojen lukumäärän kasvaessa lokeromenetelmän aikavaativuus vastaavasti huononee. Jos esimerkiksi *m* on kertaluokkaa n^2, niin

esitetty menetelmäkin on silloin $O(n^2)$. Jos avainten joukko on sopivasti rajoitettu, niin voimme yleistää edellä esitettyä lokerojärjestämismenetelmää. Olkoot avainten sallitut arvot väliltä $0 \ldots (d^k-1)$, jollakin ennalta kiinnitetyllä k:n arvolla. Tässä d on lukujärjestelmän kantaluku.

Esimerkiksi avaimet voivat olla väliltä 0 - 999 olevia kokonaislukuja. Tässä siis avaimet ovat väliltä $0 \ldots (10^3-1)$ ja järjestettävänä olkoot seuraavat 10 lukua, avainta; 453, 9, 638, 812, 48, 561, 528, 239, 94 ja 184.

Suoritetaan alla järjestäminen kolmessa vaiheessa.

alku	vaihe 1	vaihe 2	vaihe 3
453	561	009	009
009	812	812	048
638	453	528	094
812	094	638	184
048	184	239	239
561	638	048	453
528	048	453	528
239	528	561	561
094	009	184	638
184	239	094	812

Merkitään ensin kaikki avaimet kolminumeroisina. Ensimmäisessä vaiheessa suoritetaan järjestäminen viimeisen numeron mukaan. Tämä voidaan tehdä lokeromenetelmällä käyttäen 10 eri lokeroa. Sijoitus lokeroihin tulee aina tehdä stabiilisti eli uusi alkio lisätään aina lokeronsa viimeiseksi alkioksi. Ensimmäisen lokerojärjestämisvaiheen jälkeinen tilanne on yllä taulukoituna vaiheen 1 alle. Tämän jälkeen toistetaan vastaavasti lokerojärjestämismenetelmä toiseksi viimeisen numeron suhteen. Jälleen tulee lisäys lokeroihin hoitaa stabiilisti. Tämän jälkeinen tilanne on yllä taulukossa vaiheen 2 kohdalla. Vielä kolmas kierros käyttäen nyt aina ensimmäisen numeron mukaista lokeroa ja saadaan alkiot vaiheen 3 sarakkeen mukaisessa järjestyksessä.

Olkoon avainkenttänä yleisesti esimerkiksi k-numeroinen kokonaisluku ja kukin numero on kymmenjärjestelmän luku. Tällöin voidaan käyttää esitetyn kaltaista monivaiheista järjestämismenetelmää, jota kutsutaan lukujärjestelmämenetelmäksi (radix sort).

Järjestäminen tehdään silloin k:ssa eri vaiheessa ja kussakin vaiheessa järjestetään tietueet aina yhden numeron mukaan. Aloitetaankin oikeanpuoleisimmasta eli vähiten merkitsevästä numerosta ja edetään vaihe kerrallaan kohti vasemmanpuoleisinta numeroa. Menetelmässä on siis kutakin kymmentä numeroa kohti olemassa lokero, jonka viimeiseksi alkioksi siihen tuleva alkio aina sijoitetaan.

Algoritmi:

```
Lukujärjestelmämenetelmä alkioiden järjestämiseksi
Alkiot a_h, h = 1, …, n ovat listassa lis.
Avainkenttä on k-numeroinen 10 järjestelmän kokonaisluku.
```

```
Vaiheessa i järjestetään alkiot avainkentän i:nnen
numeron mukaan, kun numeroiden järjestys lasketaan
oikealta vasemmalle.
  1. Suorita askeleet 2, 3 ja 4 eli vaihe i kaikilla
     arvoilla i = 1,2, …,k.
  2. for (j = 0; j < 10; j++) merkitse lokero j tyhjäksi;
  3. for (jokainen listan lis alkio a_h, h = 1, …, n)
       poista alkio a_h listasta ja lisää se avaimensa i:nnen
       numeron mukaisen lokeron viimeiseksi alkioksi.
  4. // alkioiden lista lis on nyt tyhjä
     for (j = 0; j < 10; j++)
       lisää lokeron j alkiot listan lis perään.
```

Esitetty algoritmi on helposti laajennettavissa huomioimaan erilaiset avainkentät. Eri lokeroiden kokoaminen tulee tehdä siten, että ensin ovat pienimmän lokeron alkiot, niitä seuraavat toiseksi pienimmän jne.

Lukujärjestelmämenetelmän suoritusajan vaativuus pahimmassa tapauksessa seuraa käytetyn stabiilin lokerojärjestämismenetelmän vaativuudesta. Jos avainten mahdolliset arvot ovat väliltä *0 ... (n^{k-1})*, niin algoritmi vaatii ajan *O(nk)*. Koska *k* on vakio, niin menetelmän aikavaativuus on *O(n)*. Aikavaativuuden laskelmat on laajennettavissa erilaisiin avainkenttiin. Algoritmin huonona puolena on työtilan käyttö lokeroita varten.

3.3 Taulukointi

Idea: Vältetään toistuvia samanlaisia rekursiivisia kutsuja samoilla parametrien arvoilla taulukoimalla jo laskettuja tuloksia. (Top-down → bottom-up.)

Esimerkki. Binomikertoimet $(n! = n*(n-1)*(n-2)*\cdots*2*1)$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, 0 \le k \le n$$

voidaan, kuten tunnettua, määrittää myös Pascalin kolmion palautuskaavalla

$$\binom{n}{k} = \begin{cases} 1, & \text{kun } k = 0 \text{ tai } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{kun } 0 < k < n. \end{cases}$$

Tähän perustuva suoraviivainen rekursiivinen algoritmi olisi seuraava:

```
int bin(int n, int k)
{
  if (k == 0 || k == n)
    return 1;
  else
    return bin(n-1, k-1) + bin(n-1, k);
}
```

Arvon bin(n, k) laskeminen tällä algoritmilla vaatii kuitenkin ainakin kertaluokkaa $\binom{n}{k}$ olevan ajan, joka kasvaa eksponentiaalisen nopeasti *n*:n ja *k*:n kasvaessa.

Syy algoritmin tehottomuuteen on, että se laskee samat binomikertoimen arvot moneen kertaan:

Uudelleenlaskenta voidaan välttää laskemalla arvot iteratiivisesti *taulukoimalla*:

```
int bin(int n, int k)
{
  // määritellään taulukko c[0..n][0..k]
  for (m = 0; m ≤ n; m++) {
    c[m][0] = 1;
    if (m ≤ k)
      c[m][m] = 1;
    for (j = 1; j ≤ min(m-1, k); j++)
      c[m][j] = c[m-1][j-1] + c[m-1][j];
  }
  return c[n][k]
}
```

Tämän algoritmin aikavaativuus on vain *O(nk)*.

Tätä menetelmää voi muokata muotoon, jossa kaksidimensioinen taulukko korvataan yksidimensioisella taulukolla. Silloin pidetään yllä vain juuri muodostettavaa riviä *m*. Algoritmin aikavaativuus ei tällöin muutu..

Tätä vastaavanlaista välitulosten taulukointimenetelmää käytettiin laskettaessa lyhimmät etäisyydet verkon kaikkien solmuparien välillä Floydin menetelmällä.

Esimerkki: Kapsäkkiongelma; matkailija lähtee reppumatkalle ja hän kantaa yhtä reppua mukanaan. Hän on asettanut ehdottoman kokonaispainorajan *W*, jota repun kuorman paino ei saa ylittää. Hänellä on valittavana *n* kappaletta erilaisia tavaroita, joita hän voi ottaa mukaansa. Jokaiselle mahdolliselle tavaralle *i* hän tuntee tavaran painon w_i ja hyötyarvon p_i. Oletetaan, että tavaroiden painot ja hyötyarvot ovat positiivisia kokonaislukuja. Matkailijan ongelmana on valita mahdollisimman suuren hyötyarvon omaava reppu *R*, jonka paino ei ylitä annettua painorajaa *W*. Oletuksena on, että kukin tavara, joko otetaan kokonaisuudessaan mukaan tai ei sitten lainkaan. Jos jokin tavara otetaan mukaan, niin sitä otetaan korkeintaan yksi kappale.

Ongelmana on siis valita reppu *R* ja ongelma voidaan kirjoittaa muotoon
max $\sum_{i \in R} p_i$, kun $\sum_{i \in R} w_i \leq W$,
tässä $i \in R$ tarkoittaa sitä, että tavara *i* on valittu reppuun mukaan *R*.

Tarkoituksena on tässä käyttää taulukointia tämän ongelman ratkaisuun. Tehtävänä on silloin tunnistaa minkälaisen osaongelman tuloksia taulukoidaan. Hyvän osaongelman valinta tässä onkin yleensä vaikein kohta taulukointimenetelmän yhteydessä.

Olkoon tavarat siis numeroitu järjestyksessä *1, 2, …, n*. Valitaan osaongelmaksi sellainen ongelma, jossa repun maksimipaino saa olla korkeintaan *r* ja meidän tulee valita mukaan tavarat vain osajoukosta *1, 2, …,m*. Käytetään tästä osaongelmasta merkintää *s(m, r)*. Ideana on, että taulukoidaan tällaisten osaongelmien *s(m,r)* ratkaisuja kaikilla arvoilla, kun *m = 1, 2, …, n* ja *r = 0, 1, 2, …, W*.

Koska kaikkien tavaroiden painot ovat positiivisia, niin voidaan asettaa lähtöarvoina ratkaisujen arvot *s(0, r) = 0*, kun *r = 0, 1, …, W* eli kun mitään tavaroita ei oteta mukaan reppuun, niin repun arvo on 0 kaikilla sallituilla painoarvoilla. Tämän jälkeen tarkastellaan tilannetta, jossa otetaan aina yksi uusi tavara mahdollisesti reppuun mukaan. Tämä tarkoittaa, että lasketaan repun arvo *s(k, r)*, kun aikaisemmin on jo laskettu arvot *s(k-1, r)*. Tätä tarkastellaan kaikilla repun koon arvoilla *r = 1, …, W*.

Otettaessa huomioon uusi tavara tulee vain tarkistaa kannattaako sitä ottaa mukaan vai ei. Täten yleisesti on voimassa seuraava

$$s(k,r) = \begin{cases} s(k-1,r), & \text{jos } w_k > r \\ \max\left(s(k-1,r),\, p_k + s(k-1,r-w_k)\right), & \text{jos } w_k \le r \end{cases}$$

Kaavan ylempi osa tarkoittaa sitä, että tavara *k* ei mahdu kokoa *r* olevaan reppuun ja alemassa osassa verrataan vaihtoehtoja, että tavaraa *k* ei oteta reppuun ja toisaalta otetaan reppuun. Parempi vaihtoehto huomioidaan.

Voidaan kirjoittaa seuraava algoritmi.

Algoritmi: Kapsäkkiongelman ratkaisu taulukoinnilla.

Taulukoidaan osaratkaisut *s(m, r)*; mukana tavarat *1, …, m* ja kapsäkin maksimipaino *r*.

```
1) Aseta s[i,0] = 0, i = 0, …, n.
2) Aseta s[0,j] = 0, j = 0, …, W.
3) Suorita seuraava arvoilla i = 1, …, n:
   suorita seuraava arvoilla j = 1, …, W:
     jos (w_i ≤ j)
        s[i,j] = max{s[i-1,j],s[i-1, j – w_i] + p_i}
     muutoin
        s[i,j] = s[i-1,j];
4) Palauta arvo s[n, W].
```

Selvästi voidaan havaita, että algoritmin aikavaativuus on *O(nW)*. Tilaa käytetään myös saman verran. Itse ratkaisu voidaan myös purkaa taulukon *s* ja ongelman tietojen avulla. Jos vain ratkaisun arvo kiinnostaa, niin muistitilan tarvetta voidaan pienentää.

3.4 Ahne menetelmä

Idea: Vastaus tarkasteltavana olevan ongelman tapaukseen rakennetaan peräkkäisistä paikallisesti parhaan näköisistä (lokaalisesti optimaalisista) valinnoista.

Joissakin ongelmissa lokaalisesti optimaaliset valinnat johtavat globaalistikin optimaaliseen vastaukseen; toisissa taas paikallisten valintojen optimointi voi johtaa "umpikujaan", so. lopulta huonoihin pakkovalintoihin.

Esimerkki: Rahanvaihto. Esitettävä 17 rahayksikköä mahdollisimman vähillä kolikoilla, kun käytettävissä on 10, 5 ja 1 rahayksikön kolikot.

Ratkaisu: 17 = 10 + 5 + 1 + 1 (4 kolikkoa).

Menetelmä: Kussakin vaiheessa valitaan suurin kolikko, joka vielä sopii jäljellä olevaan summaan.

Huom. Menetelmä vaatii toimiakseen sopivat kolikkojen arvot. Jos olisi edellä mainittujen lisäksi esim. 13 rahayksiön kolikko, niin ahne menetelmä johtaisi tässä tapauksessa suboptimaaliseen ratkaisuun: 13 + 1 + 1 + 1 + 1 (5 kolikkoa).

Yleinen muoto:

```
vastaus ahne(tapaus k)
{
  y := 0;            // "Tyhjä osittaisvastaus"
  while (y ei ole täydellinen vastaus) {
    valitse y:n mahdollisista täydennyksistä e1, ..., ek
    sellainen ei, että osittaisvastaus y täydennettynä
    ei :llä antaa ”suurimman arvon";
    if (y:n täydentäminen ei ole mahdollista)
      return fail;
    else
      täydennä osittaisratkaisua y täydennyksellä ei;
  }
  return y;
}
```

Esimerkki. Verkon lyhimmät polut yhdestä lähdesolmusta verkon muihin solmuihin.

Olkoon $G=(V, E, l)$ suunnattu, painotettu verkko siten, että kaikkien kaarien pituudet ovat ei-negatiivisia eli $l(e) \geq 0$ kaikilla kaarilla $e \in E$, ja olkoot $v_0 \in V$ jokin verkon kiinnitetty solmu ("lähdesolmu").

Tehtävänä on määrittää kaikille solmuille $v \in V$ etäisyys:
$D[v]$ = lyhimmän v_0:sta v:hen johtavan polun pituus.

Ratkaisumenetelmän (Dijkstra 1959) perustana on ylläpitää joukkoa S niistä solmuista, joiden lyhimmän polun pituus lähdesolmusta v_0 tunnetaan jo. Tätä joukkoa

S sitten kasvatetaan ahneeseen tyyliin yksi solmu kerrallaan. Periaate on siis seuraavan esityksen mukainen.

1) Aluksi $S = \{v_0\}$.
2) Kussakin laajennusaskeleessa joukkoon *S* lisätään se joukon $V \setminus S$ solmu, joka näyttää olevan seuraavaksi lähinnä solmua v_0. (Ahne laajennus.)
3) Lopulta $S = V$.

Algoritmi:

```
tulos Dijkstra(L[1..n][1..n],v0)  // Oletus V ={1, ...,n},
                                  // L verkon etäisyysmatriisi
  joukko V, S;                    // bittivektoreita
  int D[1..n];                    // lyhimpien
                                  // etäisyyksien taulu
  S = {v0}, V = {verkon muut solmut paitsi v0};
  D[v0]:=0;
  for ( kaikilla v ∈ V)           // alustusaskel
    D[v] = L[v0,v];               // voi olla L[v0,v]=.∞
  while (V ei ole tyhjä joukko) {
    valitse jokin u ∈ V, jolla D[u] on minimaalinen;
    lisää u joukkoon S ja poista se joukosta V;
    for (kaikilla v ∈ V)
      D[v] = min(D[v], D[u] + L[u, v]);
  }
  return D;
}
```

Algoritmin aikavaativuudeksi saadaan $O(n^2)$; keskeinen silmukka suoritetaan $n-1$ kertaa ja kukin kierros sujuu ajassa $O(n)$. Esittämällä verkko lähtevien kaarien listoilla tai forward star rakenteella ja käyttämällä tehokkaita tietorakenteita toteutusta voidaan parantaa harvoille verkoille eli sellaisille missä $|E| = e \ll n^2$. Tehokkain tunnettu toteutus toimii ajassa $O(n \log n + e)$. Ilmeinen alaraja ongelmalle on kertaluokkaa *e*.

Algoritmin oikeellisuuden todistaminen perustuu oleellisesti siihen, että kaikkien kaarien pituudet ovat ei-negatiivisia. Oikeellisuus tulisi tietysti tarkasti todistaa.

Esimerkki: Verkon lyhin virittävä puu.

Olkoon $G = (V, E)$ suuntaamaton verkko, *V* on verkon solmujen joukko ja *E* on verkon teiden joukko. *G*:n aliverkko $G'=(V', E')$, missä $V' \subseteq V$, $E' \subseteq E$, virittää *G*:n, jos $V'=V$. Jos G' ei sisällä renkaita, se on *G*:n *virittävä metsä*. Jos se on lisäksi yhtenäinen, se on *G*:n *virittävä puu*.

Huom. 1: Virittävän puun olemassaolo edellyttää, että *G* itse on yhtenäinen.
Huom. 2: Virittävää metsää voidaan tarkastella myös kokoelmana erillisiä virittäviä puita.

Olkoon tarkasteltavan verkon teille määritelty myös pituus eli etäisyysfunktio $l: E \rightarrow R$. Aliverkon $G'=(V', E')$ pituus on $l(G') = \sum_{e \in E'} l(e)$.

Yhtenäisen verkon *lyhin virittävä puu* on se virittävä puu, jonka pituus on minimaalinen.

Virittävillä puilla on sovelluksia mm. tietoverkkojen suunnitteluissa sekä muiden verkkoalgoritmien osana.

Triviaaliratkaisu lyhimmän virittävän puun määrittämisessä olisi käydä läpi kaikki virittävät puut ja valita niistä minimaalinen. Tämä vaatisi kuitenkin eksponentiaalisen ajan (esim. n - solmuisella täydellisellä verkolla on n^{n-2} virittävää puuta).

Alla esitetyn ahneen ratkaisualgoritmin ideana on aloittaa verkon $G=(V,E)$ triviaalilla virittävällä metsällä (V, T), missä $T = Ø$. Metsän puita yhdistetään sitten "ahneesti" teillä, kunnes tuloksena on yksi yhtenäinen puu. Tämä on niin kutsuttu Kruskalin algoritmi.

Algoritmi:

```
joukko Kruskal(int[] v, joukko E);
//Oletetaan, että verkko on yhtenäinen
{
  joukko T;
  T = tyhjä joukko;
  while (metsä <V, T> ei ole puu) {
    valitse se (jokin) e ∈ E, jolla l(e) on pienin;
    poista e teiden joukosta E;
    if (tien e päät kuuluvat metsän T eri puihin)
      lisää e lyhimmän virittävän puun teiden joukkoon T;
  }
  return T;
}
```

Algoritmin tarkka aikavaativuus riippuu minimaalisen tien valintatavasta ja joukko-operaatioiden toteutuksesta. Tehokkaita tietorakenteita käyttäen voidaan laatia toteutus, jonka vaativuus on $O(e \log e)$, missä $e = |E|$.

Algoritmin oikeellisuus perustuu siihen, että lyhin tie kaikista sellaisista, jotka yhdistävät metsän kaksi eri puuta, kuuluu ainakin yhteen lyhimpään virittävään puuhun: Tämä tulee tietysti todistaa oikeaksi.

Ahne algoritmi heuristiikkana

Jos ongelman tarkka ratkaiseminen on hidasta, voidaan joskus tinkiä ratkaisun optimaalisuudesta nopeuden hyväksi: etsitään likimääräinen ratkaisu nopealla algoritmilla. Likimääräisratkaisu voi perustua johonkin *heuristiikkaan*, joka "näyttää yleensä" toimivan hyvin. Ahne menetelmä sopii usein heuristisen menetelmän perustaksi, koska ahneet algoritmit ovat yleensä nopeita.

Esimerkki: Kapsäkkiongelma (knapsack problem)

Kapsäkkiongelma; matkailija lähtee reppumatkalle ja hän kantaa yhtä reppua mukanaan. Hän on asettanut kokonaispainon rajan *W*, jota repun kuorman paino ei saa ylittää. Hänellä on *n* kappaletta erilaisia tavaroita, joista hän voi valita mukaan otettavat. Jokaiselle mahdolliselle tavaralle *i* hän tuntee tavaran painon w_i ja hyötyarvon p_i. Oletetaan, että tavaroiden painot ja hyötyarvot ovat positiivisia kokonaislukuja. Matkailijan ongelmana on valita mahdollisimman suuren hyötyarvon omaava reppu *R*, jonka paino ei ylitä annettua painorajaa *W*. Oletuksena on, että kukin tavara, joko otetaan mukaan tai ei. Siis, jos se otetaan mukaan, niin se otetaan kokonaan ja sitä otetaan korkeintaan yksi kappale.

Ongelmana on siis valita reppu *R* ja ongelma voidaan kirjoittaa muotoon
max $\sum_{i \in R} p_i$, kun $\sum_{i \in R} w_i \leq W$.

Yksinkertaisin ahne heuristinen menetelmä on pakata tavaroita kapsäkkiin niin kauan kun niitä mahtuu. Tämä voi tuottaa hyvin huonon tuloksen, koska se ei huomioi mitenkään tavaroiden arvoja.

Toinen mahdollisuus onkin pakata tavaroita hyötyarvon mukaan laskevassa järjestyksessä. Tämän huonona puolena on taasen se, että se ei huomioi mitenkään tavaroiden painoja.

Parempi ratkaisu saadaan yleensä seuraavalla algoritmilla, joka ottaa huomioon sekä tavaran arvon että sen painon.

1) Laske kullekin tavaralle painoyksikön mukaiset arvot eli suhteet $r_i = p_i/w_i$.
2) Järjestä tavarat näiden suhteiden r_i mukaan laskevaan järjestykseen.
3) Täytä kapsäkkiä suhteiden r_i mukaan laskevassa järjestyksessä, niin kauan kun tavaroita mahtuu.

Tämä menetelmä on helposti toteutettavissa aikavaativuudella $O(n \log n)$. Mutta se voi palauttaa joskus erittäin huononkin ratkaisun.

Jatkuvalla kapsäkkiongelmalla tarkoitetaan sellaista kapsäkkiongelmaa, jossa tavaroita voidaan laittaa millainen murto-osa tahansa väliltä 0 – 1. Jos jatkuvaan kapsäkkiongelmaan sovelletaan edellä esitettyä ahnetta algoritmia, niin se antaa optimaalisen ratkaisun.

Esimerkki: Kauppamatkustajan ongelma (TSP)

Annettu täydellinen painotettu suuntaamaton verkko *G* (solmut vastaavat kaupunkeja, kaarten painot vastaavat kaupunkien välisiä etäisyyksiä). Määrättävä *G*:ssä Hamiltonin kehä, jonka kustannus on minimaalinen eli "lyhin reitti, joka kulkee kaikkien kaupunkien kautta täsmälleen kerran". Ongelmassa annetaan usein myös täysi etäisyysmatriisi.

Ongelma on ns. NP-täydellinen, joten kaikki sen tunnetut tarkat ratkaisumenetelmät vaativat pahimmassa tapauksessa kaupunkien lukumäärän (*n*) suhteen eksponentiaalisen suoritusajan.

Yksinkertaisin ahne heuristinen menetelmä selvittää jonkinlainen ratkaisu kauppamatkustajanongelmalle on lienen lähimmän naapurin menetelmä. Lähdetään liikkeelle jostain kaupungista ja edetään aina lähimpään vielä käymättömään kaupunkiin, kunnes kaikki kaupungit on käyty läpi ja palataan lopulta vielä lähtökaupunkiin.

Lähimmän naapurin menetelmä kauppamatkustajan ongelmalle:

1) Valitaan tämänhetkiseksi kaupungiksi t = kaupunki 1. Olkoon käymättömät kaupungit $K = \{2, 3, \ldots, n\}$.
2) Etsi kaupunki i siten, että $d(t, i) = \min\{d(t, j) \mid j \in K\}$.
   Lisää kaari (t, i) reittiin ja aseta $t = i$ ja $K = K \setminus \{i\}$.
   Jos K = tyhjä joukko niin siirry askeleeseen 3, muutoin toista askel 2
3) Lisää kaari (t, 1) reittiin ja lopeta.

Tässä on oletettu, että kaari (t, i) askeleessa 2 aina löytyy. Menetelmä on helppo toteuttaa siten, että sen aikavaativuus on $O(n^2)$. Mutta se voi palauttaa joskus erittäin huonon reitin verrattuna optimaaliseen ratkaisuun.

Toinen ahne heuristiikka kauppamatkustajan ongelmalle:

Tarkastellaan tässä suuntaamatonta verkkoa.

1) Käsittele alla verkon teitä pituuden mukaan kasvavassa järjestyksessä.
2) Lisää tie reittiin, jos
   a) sen kummastakin päätepisteestä alkaa enintään yksi aiemmin valittu tie; ja
   b) se ei muodosta rengasta aiemmin valittujen teiden kanssa lukuun ottamatta viimeistä tietä.

Voidaan osoittaa, että jos teiden pituudet toteuttavat kolmioepäyhtälön $(c(x, y) \leq c(x, z)+c(z, y))$, niin tämä heuristiikka takaa ratkaisun, jonka kustannus on enintään kaksi kertaa optimiratkaisun kustannus $(c \leq 2c_{opt})$.

Niin sanottu *Christofidesin heuristiikka* takaa $c \leq 3/2\ c_{opt}$; tämä on paras tunnettu polynominen algoritmi. Jos kolmioepäyhtälö ei ole voimassa ja $P \neq NP$, niin mikään polynominen algoritmi ei voi taata, että $c \leq k \cdot c_{opt}$, millään vakiolla k.

4 Esimerkki eräästä ongelman ratkaisusta

Tarkastellaan niin sanottua vakaan yhdistelyn ongelmaa. Esimerkiksi olkoon meillä kolme opiskelijaa, 1, 2 ja 3 ja kolme yritystä A, B ja C. Jokainen yritys palkkaa yhden näistä opiskelijoista kesätöihin. Vastaavasti nämä kolme opiskelijaa menevät kesätöihin nimenomaan yhteen näistä kolmesta yrityksestä. Ongelmana on siinä, että kuka menee mihin yritykseen.

Kukin opiskelija on laittanut yritykset oman mielensä mukaan suosituimmuusjärjestykseen.

Esimerkiksi

		yritykset		
opiskelijat	**1**	A	B	C
	2	C	B	A
	3	A	B	C

Samoin kukin yritys on laittanut opiskelijat yrityksen kannalta suosituimmuusjärjestykseen.

Esimerkiksi

		opiskelijat		
yritykset	**A**	2	3	1
	B	1	3	2
	C	2	1	3

Ongelmana olisi löytää opiskelijoiden kiinnitys yrityksiin, yksi opiskelija aina yhteen yritykseen eli opiskelijoiden *yhdistely* yrityksiin.

Olkoon meillä esimerkiksi yhdistely 1 – A, 2 – B, 3 – C. Tämä ei ole hyvä ratkaisu, sillä jos opiskelija 2 vaihtaisi yritykseen C ja opiskelija 3 vaihtaisi yritykseen B niin sekä opiskelijat että yritykset hyötyvät. Voidaankin kysyä, että onko olemassa sellaisia ratkaisuja, joissa mikään yrityksen ja opiskelijan vaihto ei paranna sekä opiskelijan että yrityksen asemaa.

Tätä ongelmaa ovat tutkineet Gale ja Shapley ja he ovat määritelleet yhdistelylle tällaisen ominaisuuden. Kun se on voimassa, niin yhdistelyn sanotaan olevan *vakaa*.

Jokaiselle yritykselle Y ja jokaiselle opiskelijalle O, joka ei ole yhdistetty yritykseen Y on voimassa ainakin toinen seuraavista
Y pitää siihen nyt yhdistettyä opiskelijaa parempana kuin opiskelijaa O tai
O pitää parempana nyt siihen yhdistettyä yritystä kuin yritystä Y.

Tämä ongelma voidaan esittää yleisesti seuraavassa muodossa, jossa meillä on n miestä eli miesten joukko $P = \{p_1, p_2, \ldots, p_n\}$ ja n naista eli naisten joukko $T = \{t_1, t_2, \ldots, t_n\}$. Jokainen henkilö yhdistetään täsmälleen yhteen vastakkaista sukupuolta

olevaan henkilöön eli pari vihitään keskenään avioliittoon. Merkitään $P \times T$:llä kaikkien mahdollisten järjestettyjen parien *(p, t),* $p \in P$ ja $t \in T$ joukkoa. *Yhdistely Y* on järjestettyjen avioparien joukko, missä jokainen pari kuuluu joukkoon $P \times T$ siten, että jokainen joukon *P* ja joukon *T* alkio on mukana yhdistelyssä enintään yhden kerran. Yhdistely *Y* on *täydellinen yhdistely*, jos jokainen joukon *P* ja jokainen joukon *T* alkio on mukana yhdistelyssä täsmälleen yhden kerran.

Määritellään seuraavaksi käsite *preferenssi* eli *etusija* yhdistelyssä. Jokainen mies $p \in P$ asettaa kaikki naiset järjestykseen suosituimmasta vähiten suosittuun. Silloin mies *p* antaa etusijan naiselle *t* naisen *t'* suhteen, jos miehellä *p* on nainen *t* järjestyksessä ennen naista *t'*. Tätä etusijajärjestystä kutsutaan miehen *m* preferenssilistaksi. Järjestyksessä ei sallita yhtä suuruuksia. Naisille on tietysti myös omat preferenssilistat.

Jos on annettuna täydellinen yhdistely, niin siinä voi olla aikaisemmin esitetyn kaltaisia epävakaita tapauksia. Tavoitteena on löytää *vakaa täydellinen yhdistely*. Esitetään seuraavaksi perusteet algoritmille, jolle annetaan syöttötietona miesten ja naisten preferenssilistat ja se palauttaa vakaan täydellisen yhdistelyn eli avioparien joukon.

Aluksi kaikki miehet ja naiset ovat naimattomia. Yhdistelemätön (vapaa, naimaton) mies *p* valitsee naisen *t*, kuka on korkeimmalla miehen *p* preferenssilistalla ja esittää naiselle kosinnan eli yhdistelytarjouksen. Nyt pari *(p, t)* ovat kihloissa eli esiyhdistetyt.

Yleisesti ollaan tilanteessa, jossa ainakin joku mies ja nainen ovat vapaita eli ei edes kihloissa. Valitaan joku vapaa mies *p*, joka sitten valitsee preferenssilistaltaan ensimmäisen naisen *t*, jolle mies ei vielä ole esittänyt kosintaa. Ja mies kosii häntä. Jos nainen *t* on vapaa niin pari *(p, t)* kihlautuu. Jos nainen *t* ei ole vapaa, niin hän on kihloissa jonkun miehen *p'* kanssa. Silloin nainen *t* päättää oman preferenssilistansa kautta kumpi miehistä *p* ja *p'* on järjestyksessä ennen hänen preferenssilistallaan. Hän valitsee preferenssilistalla edellä olevan ja kihlautuu hänen kanssa. Toinen miehistä tulee siten vapaaksi.

Lopulta algoritmi päättyy, kun ketään ei ole vapaana.

Alla on tämä pseudokoodina

```
Aluksi jokainen mies p∈P ja jokainen nainen t∈T on
vapaa
while (on olemassa mies p, joka on vapaa ja joka ei ole
       kosinut kaikkia naisia) {
  valitse tällainen vapaa mies p
  olkoon t ensimmäinen nainen miehen p
     preferenssilistalla, jota p ei ole kosinut
  if (t on vapaa) {
    pari (p, t) kihlataan
  } else {
       // olkoon kihlautunut pari (p', t)
    if (p' on ennen p:tä t:n preferenssilistalla) {
```

```
      p säilyy vapaana
  } else {
      aseta pari (p, t) kihlapariksi
      aseta mies p' vapaaksi
    }
  }
}
palauta kihlaparit aviopareina
```

Voidaan todistaa, että algoritmin while-silmukkaa suoritetaan korkeintaan n^2 kierrosta. Kun nainen kihlautuu, niin hän säilyy kihlattuna koko ajan aina algoritmin päättymiseen asti. Miesten suhteen algoritmi toimii erilailla eli kihlautunut mies voi vapautua ja jos mies kihlautuu uuden naisen kanssa, niin tämä uusi nainen on miehen preferenssilistalla huonommassa asemassa kuin edellinen. Jos mies on vapaa jollakin hetkellä, niin aina on olemassa vielä nainen, jota hän ei ole kosinut. Lopuksi voidaan osoittaa, että algoritmin päättyessä yhdistely on täydellinen ja lisäksi se on vakaa.

Algoritmin analysointia varten tulee sen toteutusta ajatella tarkemmin. Koska kierroksia tulee pahimmassa tapauksessa n^2 verran, niin tämä on alaraja. Jos jokainen kierros voidaan suorittaa vakioajassa, niin tästä saataisiin tarkka raja pahimmalle tapaukselle. Tarkastellaan tarvittavia talletusrakenteita. Miehet ja naiset voivat olla numeroituja välillä $1 \ldots n$, joten tunnistetaan heidät näillä kokonaisluvuilla. Jokaiselle miehelle ja naiselle tarvitaan preferenssilistat ja ne voivat olla matriiseissa `PPref[1..n, 1..n]` ja `TPref[1..n, 1..n]`.

Algoritmissa tulee tunnistaa aina vapaa mies. Tätä varten voidaan käyttää mitä tahansa yksinkertaista rakennetta, vaikkapa jonoa. Silloin lisääminen ja poistaminen tapahtuvat vakioajassa.

Seuraavaksi valitun vapaan miehen *p* tulee tunnistaa preferenssilistaltaan ensimmäinen nainen, jota hän ei ole kosinut. Tätä varten voisi olla vektori `Seuraava[1..n]`, jonka alkiossa `i` on järjestysnumero preferenssilistalla sille naiselle, jota mies `i` seuraavaksi kosii. Aluksi siis `Seuraava[i] = 1` kaikilla $i = 1,\ldots n$. Täten päästään taas vakioaikaiseen suoritukseen.

Jokaiselle naiselle tulee lisäksi selvittää, onko hän jo kihloissa ja jos on, niin kenen kanssa. Tätä varten voidaan naisille pitää yllä vektoria `Nykyinen[1..n]`, jonka alkiossa `i` on sen miehen tunnus, jonka kanssa nainen `i` on kihloissa. Arvo `0` ilmaisee sen, että nainen `i` on vapaa. Aluksi siis `Nykyinen[i] = 0` kaikilla $i = 1,\ldots n$.

Jos nainen `t` on saanut kosinnan mieheltä `p` ja nainen `t` on kihloissa miehen `p'` kanssa, niin tulisi saada selville kumpi miehistä `p` ja `p'` on korkeammalla naisen `t` preferenssilistalla. Naisen `t` preferenssilistan läpikäynnillä tämä voidaan selvittää, mutta se ei ole vakioaikainen toimenpide. Vakioaikaisuutta varten voidaan muodostaa taulukko `Jarjestys[1..n, 1..n]`, jonka alkiossa `[i, j]` on miehen `j` järjestys naisen `i` preferenssilistalla. Tällöin esitetty kahden miehen preferenssien vertailu saadaan vakioaikaiseksi. Tämä matriisin muodostaminen tehdään aloitusvaiheessa samalla luettaessa naisten preferenssilistoja.

Edellä esitettyjä talletusrakenteita käyttäen esitetty täydellisen vakaan yhdistelyn muodostaman algoritmin pahimman tapauksen aikavaativuus on $O(n^2)$. Samoin tilavaativuus on $O(n^2)$.

Kannattaa huomata, että tällaisia vakaita täydellisiä yhdistelyjä voi olla useita. Tällöin voisi olla jokin kriteeri, joka laittaa täydelliset yhdistelyt vielä jonkinlaiseen paremmuusjärjestykseen.