TIEA211 Algoritmit 2

Antti Valmari
Jyväskylän yliopisto

1. toukokuuta 2024

1 Johdanto

Tässä luvussa kerrotaan ensin tämän tekstin tavoitteista. Sitten näytetään, miten pieniä ohjelmanpätkiä voi toisinaan testata tehokkaasti tietokoneen avulla vaikka käytettävissä ei olisi mallitoteutusta. Sen jälkeen käytetään puolitushakua esimerkkinä siitä, kuinka vaikeaa on saada ohjelmista luotettavia, ja minkälaisilla keinoilla luotettavuutta voi parantaa. Luvussa käsitellään sekä puolitushaun testaamista että perustelemista toimivaksi huolellisella päättelyllä.

1.1 Tästä tekstistä

Tämä kirjoitus on tarkoitettu oppimateriaaliksi kurssille TIEA211 Algoritmit 2. Se poikkeaa monesta muusta algoritmien oppikirjasta sikäli, että tavoitteena ei ole pelkästään käydä läpi joukko tärkeimpiä algoritmeja ja tietorakenteita, vaan myös harjoitella laajemmin ohjelmoinnissa käyttökelpoisia ratkaisuja sekä ohjelmien saamista oikein toimiviksi. Tämä kirjoitus kannattaa ajatella ohjelmointitaidon, algoritmien ja tietorakenteiden oppimateriaaliksi.

Yleisimmät tietorakenteet ja algoritmit saa nykyisin ohjelmointikielten kirjastoista valmiina. Siksi, jos ajatellaan pelkästään niiden käyttämistä, ei niitä tarvitsi osata ohjelmoida itse. Mutta hyvän ohjelmointitaidon saavuttaminen vaatii harjoittelua ja monenlaisiin ratkaisuihin tutustumista. Yleisimmät tietorakenteet ja algoritmit ovat siihen hyvä kohde, sillä ne sisältävät monenlaisia huolellisesti mietittyjä ratkaisuja. Harjoittelevathan muusikotkin soittamaan kappaleita, jotka joku toinen on jo soittanut ja laittanut äänityksen nettiin.

Tavoitteena ei ole pelkästään oppia ohjelmointia yksityiskohtien tasolla, vaan myös kokonaisuuksien suunnittelua. Tätä havainnollistakoon opintojakson tietomalli eräässä jo useita vuosia sitten käytöstä poistetussa opetushallinnon tietojärjestelmässä.

Eräs tavoite oli varmistaa esitietoketjujen toimivuus rakentamalla järjestelmään vaatimus, että jos opintojakso pidetään, niin myös jokainen sen pakollinen esitieto pidetään. Järjestelmä ei kuitenkaan tuntenut pidempiä ajanjaksoja kuin yksi lukuvuosi. Niinpä sen mukaan jos A on B:n pakollinen esitieto ja B toteutetaan, niin A täytyy toteuttaa joskus samana lukuvuonna. Kun eräs syksyn opintojakso pidettiin viimeisen kerran, kieltäytyi järjestelmä vahvistamasta sen tietoja, koska sen pakollista esitietoa ei ollut opetusohjelmassa. Sitä ei ollut opetusohjelmassa,

koska se oli pidetty viimeisen kerran edellisenä keväänä. Sen sijaan järjestelmä ei olisi huomannut, jos uuden syksyn kurssin pakollinen esitieto olisi pidetty ensimmäisen kerran vasta seuraavana keväänä. Hyvä tavoite esitietoketjujen automaattisesta tarkastamisesta kääntyi nurinkuriseksi!

Opintojakson tiedot pystyi vahvistamaan vasta kun sen pakollisten esitietojen tiedot oli vahvistettu. Niinpä opettajat joutuivat koordinoimaan keskenään, missä järjestyksessä opintojaksojen tiedot vahvistetaan. Tämä onnistui kohtalaisesti saman oppiaineen sisällä, mutta huonommin oppiaineiden välillä. Lopulta yliopiston hallinto joutui määräämään mahtikäskyllä varhaisen päivämäärän, johon mennessä matematiikan alkeiskurssien tiedot piti vahvistaa.

Opintojakson aikataulua ei esitetty viikkorytminä vaan erityyppisten oppimistapahtumien luetteloina, joissa tapahtumasta kerrotaan laji, päivämäärä, alkamis- ja päättymisaika vuorokauden tunteina ja paikka. Tiedot voitiin kopioida lukuvuodelta seuraavalle niin että viikonpäivät säilyivät samoina (joten päivämäärät hieman muuttuivat). Edelliseltä vuodelta kopioitu oppimistapahtuma saattoi kuitenkin osua itsenäisyyspäivään, pääsiäispäivään, vappuun tai muuhun arkivapaaseen. Järjestelmä huomasi sellaiset, joten ne oli melko helppo poistaa. Sen sijaan sitä järjestelmä ei huomannut, että poistetun oppimistapahtuman kohdalta kopioitui sitä seuraavalle vuodelle aukko. Niinpä ympäri yliopistoa käytettiin joka vuosi valtava määrä työtunteja tarkastamalla opetusaikatauluja ja lisäämällä aukkojen kohdalta puuttumaan jääneitä oppimistapahtumia.

Parempi tietomalli olisi sellainen, jossa tavallisen opintojakson esitystavan lähtökohtana olisi viikkoaikataulu. Siinä kerrotaan oppimistapahtumien lajit, viikonpäivät, kellonajat ja paikat, esimerkiksi tiistaina 10–12 luento salissa 3, torstaina 14–16 luento salissa 3, maanantaina 12–14 harjoitus salissa 231 ja niin edelleen. Ihminen voi vaihtaa esimerkiksi salia siten, että muutos koskee jokaista viikkoa. Ihminen voi myös tehdä yksittäisiä poikkeuksia, kuten vaihtaa jonkin tiistain luennon toiseen saliin. Järjestelmä näyttää joko viikkoaikataulun ja poikkeusten luettelon, tai kunkin viikon erikseen siten että se on ottanut poikkeukset huomioon. Aikataulullisesti epäsäännöllisen opintojakson voi esittää pelkkänä poikkeusten luettelona.

Järjestelmä kopioi viikkoaikataulun seuraavalle vuodelle. Arkivapaille osuvia oppimistapahtumia ei poisteta viikkoaikataulusta, vaan järjestelmä vain jättää ne näyttämättä. Sen voi toteuttaa esimerkiksi tekemällä kopioinnin yhteydessä automaattisesti poikkeusten luettelon, jossa arkivapaiden kohdalle osuvat oppimistapahtumat on ilmoitettu poistetuiksi. Teknisesti ja mahdollisesti jopa loppukäyttäjienkin kannalta vielä helpompi ratkaisu olisi, että arkivapaiden kohdalle osuville oppimistapahtumille ei tehdä tietojärjestelmässä yhtään mitään, vaan oletetaan jokaisen opiskelijan ja opettajan tietävän ilmankin, että vappuna ei ole luentoa vaikka viikkoaikataulun mukaan siihen osuisi luento.

Vaikka tämä esimerkki on järjestelmätasolla ja tietorakenteet ja algoritmit ovat

yksityiskohtien tasolla tai vain vähän sitä ylempänä, on niiden taustalla silti samankaltaista ajattelutapaa. Tiedot kannattaa organisoida siten, että niistä saadaan tehokkaasti esille se mitä tarvitaan ja niitä voidaan muuttaa tehokkaasti. Ulospäin näkyvä esitystapa ei välttämättä ole tarkoituksenmukaisin esitystapa.

1. Tekstin seassa on kysymyksiä, jotka on numeroitu vasempaan marginaaliin. Tarkoitus on, että ennen lukemisen jatkamista lukija miettii tai ainakin yrittää miettiä vastauksen kysymykseen, ja sen jälkeen lukee mallivastauksen. Mallivastaukseen pääsee klikkaamalla kysymyksen numeroa ↩, ja sieltä pääsee takaisin klikkaamalla mallivastauksen kohdalla olevaa kysymyksen numeroa. Joissakin tekstinlukuohjelmissa mallivastauksen alku ilmestyy näkyviin, kun siirtää kursorin kysymyksen numeron kohdalle. Jollet jo käynyt lukemassa tämän kysymyksen mallivastausta, niin käy nyt!

   Kysymykseen saattaa olla useita oikeita vastauksia. Mallivastauksessa näytetään niistä vain yksi. Joidenkin mallivastausten yhteydessä on annettu lisätietoa, joka ei ole osa vastausta. Sen tunnistaa merkinnästä "*Lisätietoa*". Lisätiedoissa voi olla muun muassa perustelu mallivastaukselle, muita oikeita vastauksia tai jatkopohdintaa kysymyksen aihealueelta.

   Koska tavoitteena on harjoitella laajasti ohjelmoinnissa käyttökelpoisia ratkaisuja, ei osa kysymyksistä käsittele puheena olevaa algoritmia tai tietorakennetta, vaan jotakin muuta tärkeää asiaa, jonka puheena oleva asia motivoi.

   Kysymysten joukossa on myös sellaisia, joiden tavoitteena on valmistella pian tulossa olevien asioiden oppimista saamalla lukija pohtimaan aihepiiriä etukäteen. Tarkoitus ei siis ole, että lukija osaa vastata jokaiseen kysymykseen. Kysymykset, joiden numeron edessä on *, ovat muita vaikeampia tai vaativat tavallisten ohjelmointikurssien ulkopuolista taustatietoa. Niitä suositellaan vain kysymyksen aihepiiristä kiinnostuneille.

   Otetaan aluksi muutama lämmittelykysymys. Alla olevan ohjelmanpätkän tehtävänä on laskea tentin arvosana pisterajojen ja opiskelijan saaman pistemäärän perusteella.

```
1  int arvosana = 0;
2  while( arvosana < 5 && pisteet >= pisteraja[ arvosana ] ){
3    ++arvosana;
4  }
```

2. Millä ehdolla arvosanaksi pitää antaa 3? Vastaa käyttäen ohjelmanpätkän muuttujia.
3. Mikä ehto pisterajojen täytyy täyttää, jotta ohjelmanpätkän toiminta olisi mielekästä? Vihje: se ei ole mielekästä esimerkiksi kun pisterajat ovat $[12,9,15,6,9]$.
4. Perustele, että pisteet eivät riitä mihinkään suurempaan arvosanaan kuin minkä ohjelmanpätkä antaa!

5. Perustele, että pisteet riittävät siihen arvosanaan, jonka ohjelmanpätkä antaa!
6. Ohjelmanpätkästä poistetaan `arvosana < 5 &&` ja taulukkoa `pisteraja` muutetaan. Miten sitä pitää muuttaa, jotta ohjelman tuottamat lopputulokset eivät muuttuisi?

7. Käytettävissä on satunnaislukugeneraattori RANDOM(n), joka tuottaa satunnaisen kokonaisluvun väliltä 0, 1, ..., $n-1$. Jokainen vaihtoehto on yhtä todennäköinen. Kirjoita ohjelmanpätkä, joka tuottaa satunnaisen kokonaisluvun väliltä a, $a+1$, ..., y!

1.2 Arvosanan laskemisen testaaminen tietokoneella

Jos pitää toteuttaa ohjelmanpätkä arvosanan laskemiseksi pisterajojen ja pistemäärän perusteella, ja jos ei ole vielä paljoa ohjelmointikokemusta, niin miten voi varmistua, että se laskee oikein? Tässä luvussa esitellään hyvä keino. Se ei ole täysin aukoton, mutta täysin aukotonta keinoa ei ole olemassakaan.

Olkoon annettu pisterajat, vaikka $[14, 17, 20, 23, 26]$. Ensimmäinen numero eli `rajat[0]` on arvosanan 1 alaraja, seuraava eli `rajat[1]` on arvosanan 2 alaraja ja niin edelleen arvosanaan 5 saakka. Arvosanaa vastaava pisteraja on siis `rajat[ arvosana-1 ]`. Saatu arvosana on väärä jos ja vain jos pisteet eivät riitä siihen tai ne riittäisivät korkeampaankin arvosanaan. Tästä saadaan yritys testiksi, onko testattavan ohjelmanpätkän laskema arvosana oikea. Kohta näemme, että testimme ei ihan toimi, mutta jostain täytyy aloittaa.

```
int arvosana = testattava( rajat, pist );
if( pist < rajat[ arvosana-1 ] || pist >= rajat[ arvosana ] )
  { ... ilmoita virheestä }
```

Tietokone on helppo laittaa testaamaan kaikki pistemäärät johonkin kohtuulliseen maksimiin saakka, eikä siinä kulu paljoa aikaa.

```
for( int pist = 0; pist <= 30; ++pist ){
  ... edellä ollut koodinpätkä
}
```

Kun tätä kokeilee jollakin sellaisella ohjelmointikielellä, jossa taulukon indeksointien laillisuus tarkastetaan suoritusaikana, saa varsin pian ilmoituksen laittomasta indeksoinnista. Se johtuu siitä, että jos saatu arvosana on 0, niin `rajat[ arvosana-1 ]` indeksoi taulukon vasemman reunan ohi, ja jos saatu arvosana on 5, niin `rajat[ arvosana ]` indeksoi taulukon oikean reunan ohi. Virhe tulee siis silloin, kun ohjelma testaa riittävätkö pisteet arvosanaan 0 tai riittäisivätkö ne korkeampaan arvosanaan kuin 5. Kumpaakaan testiä ei tarvita, koska mikä tahansa pistemäärä riittää ainakin nollaan eikä mikään riitä viitosta suurempaan.

```
1  #include <iostream>
2  class taulukko{
3    static int A[5];
4  public:
5    int size(){ return 5; }
6    int operator[]( int i ){
7      if( 0 <= i && i < 5 ){ return A[i]; }
8      std::cout << i << " laiton indeksi\n"; return -1;
9    }
10 };
11 int taulukko::A[5] = {14, 17, 20, 23, 26};

12 int testattava( taulukko & rajat, int pist ){
13   for( int i=0; i < rajat.size()-1; i++ )
14     if( pist >= rajat[i] && pist < rajat[i+1] ) return i+1;
15   return 0;
16 }

17 int main(){
18   taulukko rajat;
19   for( int pist = 0; pist <= 30; ++pist ){
20     int arvosana = testattava( pist );
21     if(
22       arvosana < 0 || arvosana > 5 ||
23       ( arvosana > 0 && pist < rajat[ arvosana-1 ] ) ||
24       ( arvosana < 5 && pist >= rajat[ arvosana ] )
25     ){ std::cout << pist << " " << arvosana << "\n"; }
26   }
27 }
```

Kuva 1: Arvosanaohjelman testaaja ja virheellinen arvosanaohjelma

Ohjelmointikielen C++ suoritusympäristö huomaa laittomat indeksoinnit, jos taulukkoa indeksoidaan toiminnolla `.at(...)`, mutta ei huomaa, jos indeksoidaan toiminnolla `[...]`. Tämä voidaan kiertää luomalla oma taulukkotyyppi. Arvosanan laskentaohjelman testaamiseksi riittää, että siinä on toiminnot taulukon alkioiden ja koon lukemiseksi. Taulukon sisältö saa olla vakio. Kuvan 1 riveillä 2, ..., 11 luodaan tällainen taulukkotyyppi. Rivi 1 tarvitaan, jotta ohjelmassa voitaisiin tulostaa mitään.

Kuvan 1 riveillä 22, ..., 24 on korjattu testi. Se ei indeksoi pisterajataulukkoa laittomasti. Jos ohjelmointikielen suoritusympäristö ilmoittaa laittomista indeksoinneista tai rivit 2, ..., 11 ovat mukana, niin rivi 22 ei ole välttämätön. Se selkeyttää virheilmoitusta siinä tapauksessa, että testattavan aliohjelman tulos ei ole nollan ja viitosen välillä. Jos riviä 22 ei olisi, niin jos testattava aliohjelma palauttaa negatiivisen luvun niin rivi 24 indeksoisi laittomasti, ja viitosta suuremmalla

luvulla rivi 23 indeksoisi laittomasti. Silloinkin virhe jäisi kiinni, mutta virheilmoitus ei kertoisi selkeästi mikä oli vikana.

8. Toteuta testiympäristö jollakin itsellesi tutulla ohjelmointikielellä. Mitä se tulostaa, kun testattavana on rivien 15, . . . , 17 ohjelmanpätkä?
9. Kuvattuna loppukäyttäjälle tutuin käsittein, minkä virheen riveillä 15, . . . , 17 oleva ohjelmanpätkä tekee?
10. Jos riviltä 15 poistetaan `-1`, niin minkä virheen ohjelmanpätkä tekee?
11. Miten ohjelmanpätkän voi korjata ilman että mitään jo olemassa olevaa riviä muutetaan? (Pitää siis lisätä jotakin.)

    Testiympäristömme ei saa kiinni kaikkia virheellisiä ohjelmanpätkiä. Se ei saa kiinni esimerkiksi seuraavaa:

```
if( pist < rajat[0] ){ return 0; }
if( pist >= rajat[4] ){ return 5; }
return ( pist - rajat[0] ) / ( rajat[1] - rajat[0] ) + 1;
```

12. Anna jotkin pisterajat, joilla tämä ohjelmanpätkä jää kiinni.

* 13. Anna jotkin pisterajat, joissa rajat eivät ole tasavälein, ja joilla tämä ohjelmanpätkä ei jää kiinni.

Ei ole helppoa suunnitella testaamisen kannalta mahdollisimman tehokasta pisterajataulukkoa tai pisterajataulukkojen kokoelmaa. Se vaatii samankaltaista ajattelua kuin mitä vaatii virheiden löytäminen miettimällä. Edellä kehitetyn testaajan teho ei perustu siihen, että se olisi huolella mietitty, vaan siihen, että se tekee monta testiä. Luotettavia ohjelmia saa aikaan vain testaamisen ja miettimisen yhdistelmällä. Miettimisen taitokaan ei koskaan voi olla täydellinen, mutta kuten monessa muussakin taidossa, siinä voi kehittyä aloittamalla sopivan tasoisilla harjoituksilla ja siirtymällä asteittain vaativampiin. Miettimisen taitoa kehittää sekin, että tekee omille ohjelmilleen testaajia, löytää niiden avulla omista ohjelmistaan virheitä, miettii virheiden syitä ja korjaa ne.

1.3 Puolitushaku

Puolitushaku (***binary search***) on eräs kaikkein yleisimmin käytettyjä esimerkkejä tehokkaista algoritmeista. Se on erittäin nopea keino löytää alkio suuruusjärjestyksessä olevasta taulukosta. Etsittävää alkiota kutsutaan usein ***avaimeksi*** (***key***). Eräs sen versio toimii seuraavasti. Aluksi etsintäalueena on koko taulukko. Kullakin kierroksella avainta verrataan etsintäalueen keskimmäiseen alkioon. Jos ne ovat yhtäsuuret, on etsittävä alkio löytynyt. Jos avain on pienempi kuin verrattu, jatketaan etsintää etsintäalueen alkupuolikkaasta. Muussa tapauksessa etsintää jatketaan etsintäalueen loppupuolikkaasta. Kun etsintäalue on kaventunut nollan kokoiseksi, lopetetaan ja todetaan että etsittävää alkiota ei ole taulukossa.

Joka kierroksella etsintäalue suunnilleen puolittuu. Melko pieni määrä puolittamisia riittää kaventamaan isonkin alueen nollan kokoiseksi. Jos etsintäalueen koko on miljoona, niin 20 puolitusta riittää. Vaikka puolitus olisi jossain määrin monimutkainenkin toimenpide, on paljon nopeampaa tehdä 20 puolitusta kuin tarkastaa erikseen jokainen miljoonasta alkiosta.

Jon Bentley kertoo kirjassaan Programming Pearls (1986), kuinka hän antoi tämän koodattavaksi yli sadalle ammattilaisohjelmoijalle. Aikaa oli pari tuntia. Ohjelmoijat saivat itse valita ohjelmointikielen tai käyttää pseudokoodia. Melkein kaikki ilmoittivat onnistuneensa. Mutta kun Bentley testautti vastaukset, kävi ilmi, että enintään 10 % oli oikein!

Donald Knuth kertoo kirjassaan Sorting and Searching (1973), että puolitushaku julkaistiin ensimmäisen kerran 1946, mutta virheetön puolitushaku julkaistiin ensimmäisen kerran 1962. Richard Pattis havaitsi 1988 (Textbook Errors in Binary Searching), että 20:stä oppikirjasta 15:ssa puolitushaku oli väärin. Hän tunnisti yhteensä 22 erilaista virhettä, joista 11 liittyi toimintaperiaatteen toteutukseen ja 11 väärään parametrinvälitysmekanismiin.

Vuonna 2006 Joshua Bloch julkaisi blogin ”Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken”, jossa kerrottiin Javan standardikirjaston puolitushausta löytyneen ohjelmointivirheen tarina.[1] Kyseinen versio oli suora kopio Bentleyn testaamasta ja oikeaksi osoittamasta versiosta, mutta virhe liittyi ilmiöön, jota Bentley ei ottanut huomioon, koska sitä ei hänen aikanaan voinut esiintyä. Kyse oli lukualueen ylivuodosta kun lasketaan etsintäalueen puoliväli. Tarvittiin selvästi yli miljardin alkion kokoinen taulukko, että ylivuoto tapahtuisi. Bentleyn aikana niin suuria muisteja oli vain unelmissa, mutta tänään niitä on kännyköissä. Lukualueiden ylivuotojen välttäminen on oma osaamisen lajinsa, johon on algoritmikirjallisuudessa kiinnitetty huomiota vähän tai ei ollenkaan. Koska siitä on tullut ajankohtainen, käsitellään sitä tässä kirjoituksessa hieman mutta ei kovin paljoa.

Vuonna 2010 Mike Taylor toisti Bentleyn kokeen netissä.[2] Vastauksia tuli satoja. Niistä arviolta puolet oli oikeita. Tästä ei kuitenkaan voi tehdä kovinkaan varmoja johtopäätöksiä, koska ei voida tietää kuinka moni jätti kertomatta että oli yrittänyt mutta epäonnistunut, kuinka moni rikkoi sääntöä että koeajaa ei saa, eikä kuinka moni oli katsonut mallia jostakin.

Kuvassa 2 on C++-kielinen esimerkki puolitushausta, jossa on kaksi virhettä (ja lisäksi Blochin kertoma ylivuotovirhe). Tarkoitus on, että se palauttaa -1 jos avain ei ole taulukossa, ja muuten se palauttaa avaimen (jonkin) paikan. Yritä hetki löytää virheet, ja jatka sitten lukemista.

[1] `https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html`
[2] `https://reprog.wordpress.com/2010/04/21/binary-search-redux-part-1/`

```
1  int p_haku( const taulukko & A, int avain ){
2    int ala = 0, yla = A.size()-1;
3    while( ala < yla ){
4      int vali = (ala + yla)/2;
5      if( A[vali] == avain ){ return vali; }
6      else if( A[vali] > avain ){ yla = vali; }
7      else{ ala = vali; }
8    }
9    if( A[ala] == avain ){ return ala; }
10   else{ return -1; }
11 }
```

Kuva 2: Virheellinen puolitushaku

1.4 Puolitushakualiohjelman testaaminen tietokoneella

On melko helppoa laittaa tietokone testaamaan puolitushakualiohjelmia. Kuvassa 3 on aliohjelma, joka saa taulukon ja testaa puolitushakualiohjelmaa jokaisella avaimen arvolla nollasta yhdeksään. Jos taulukossa on vain lukuja ykkösestä kahdeksaan, niin testatuksi tulevat kaikki tapaukset joissa avain on taulukossa, tapaus jossa avain on taulukon kaikkia alkioita pienempi, tapaus jossa avain on kaikkia taulukon alkioita suurempi, sekä taulukon sisällöstä riippuen mahdollisesti muitakin tapauksia, joissa avain ei ole taulukossa.

(Yhdestä kirjaimesta koostuvat muuttujien nimet ovat osoittautuneet hankaliksi kun muuttujia tarvitsee etsiä esimerkiksi ohjelman korjaamiseksi, joten tuttujen `n` ja `i` tilalla on käytetty `nn` ja `ii`.)

Rivillä 4 tarkastetaan, että testikohteen tulos on joko -1 tai laillinen taulukon

```
1  void testaa( const taulukko & A ){
2    for( int avain = 0; avain <= 9; ++avain ){
3      int tulos = p_haku( A, avain ), nn = A.size();
4      bool ok = ( -1 <= tulos && tulos < nn );
5      if( ok ){
6        bool mukana = false;
7        for( int ii = 0; ii < nn; ++ii ){
8          if( A[ii] == avain ){ mukana = true; break; }
9        }
10       if( tulos == -1 ){ ok = !mukana; }
11       else{ ok = ( A[ tulos ] == avain ); }
12     }
13     if( !ok ){ ilmoita virheestä }
14   }
15 }
```

Kuva 3: Testin ajava aliohjelma

```
1  #include <iostream>
2  #include <vector>
3  typedef std::vector< int > taulukko;
   ... tähän testikohde ja aliohjelma testaa
4  int main(){
5    taulukko A;
6    A.push_back(3); testaa(A); A.push_back(6); testaa(A);
7    A.push_back(7); testaa(A); A.clear(); testaa(A);
8    A.push_back(3); A.push_back(3); testaa(A);
9    ...
10   A.clear();
11   for( unsigned ii = 1; ii < 9; ++ii ){
12     for( unsigned jj = 0; jj < ii; ++jj ){ A.push_back(ii); }
13   }
14 }
```

Kuva 4: Testaavan ohjelman alku ja loppu

indeksi. Jos se on, niin riveillä 6, ..., 9 selvitetään, onko avain oikeasti taulukossa. Jos testikohde vastasi, että avain ei ole taulukossa, niin rivillä 10 asetetaan `ok` sen mukaan oliko vastaus oikein. Jos testikohde vastasi, että avain on kohdassa `tulos`, niin rivillä 11 asetetaan `ok` sen mukaan, onko se totta.

Tällainen tapa testata riippuu jossain määrin siitä, että osataan kirjoittaa tarpeeksi luotettava testaava ohjelma. Tätä ongelmaa lievittää se, että osa testaavan ohjelman virheistä paljastuu siten, että se väittää testikohteen vastanneen väärin, mutta käsin tarkastuksessa paljastuu että testikohteen vastaus olikin oikein. Myös se auttaa, että testaavassa ohjelmassa voidaan käyttää yksinkertaisia hitaita ratkaisuja, kuten avaimen etsiminen selaamalla alkioita peräkkäin riveillä 6, ..., 9.

14. Miksi ei voida testata siten, että etsitään avaimen paikka (hitaalla mutta) luotettavalla tavalla, ja verrataan testikohteen vastausta siihen?

Vaikeaksi ongelmaksi jää muodostaa riittävän kattava testiaineisto. Se on tyypillisesti mahdotonta. Kuitenkin pienellä järjen käytöllä voidaan kattaa suuri joukko tapauksia. Kannattaa hyödyntää sitä, että tietokoneet ovat nopeita ja myös testiaineiston muodostamista voi automatisoida. Niinpä usein on mahdollista tehdä tuhansia testejä sekunnin osassa.

Kuvassa 4 on testausohjelman alkuosat ja pääohjelma. Riveillä 1, 2 ja 3 ladataan tarvittavat kirjastot ja luodaan taulukoille tyyppi. Testikohde ja aliohjelma `testaa` tulevat rivien 3 ja 4 väliin. Riveillä 6 ja 7 testataan taulukoilla $[3]$, $[3,6]$, $[3,6,7]$ ja tyhjällä taulukolla. Sitten testataan taulukolla jossa on kaksi samaa alkiota. Rivin 9 kohdalla testataan muutamalla taulukolla jossa sama alkio toistuu, sekä taulukolla $[1,2,3,4,5,6,7,8]$.

Lopuksi testataan taulukolla, jossa on kaikkia alkioita ykkösestä kahdeksaan, kutakin oman itsensä ilmoittama määrä kertaa. Se tuskin on paras mahdollinen ei

ihan pieni taulukko ja ainakaan se ei yksinään kata kaikkia virhemahdollisuuksia, mutta se on tyhjää parempi. Vielä ei kannata käyttää paljoa aikaa testaamisen miettimiseen. Tällä testiaineistolla saadaan osa virheistä kiinni. Sitten kun testikohde on niiden osalta korjattu, voidaan testauksen miettimistä jatkaa.

Kuvan 2 aliohjelma jäi testeissä ikuiseen silmukkaan. Lisäämällä kuvan 3 rivin 3 eteen testattavan taulukon koon ja avaimen tulostus paljastuu, että se tapahtui taulukolla $[3,6]$ avaimen ollessa 4. Alla on simuloitu suoritusta tällä syötteellä. Riviltä 3 mennään aina riville 4 koska $ala < yla$, ja sieltä riville 7 koska $A[vali] = A[0] = 3 < 4 = avain$. Rivillä 7 sijoitetaan *ala*:an 0, mutta se ei muuta mitään, koska siinä on jo 0. Sitten palataan riville 3 ja sama toistuu loputtomasti.

rivi	2	3	4	7	3	4	7	...
ala	0			0			0	...
vali			0			0		...
yla	1							...

Tämä vika voidaan korjata huolehtimalla, että $A[vali]$ jää hakualueen ulkopuolelle. Se onnistuu muuttamalla rivien 6 ja 7 sijoituksiksi `yla = vali-1;` ja `ala = vali+1;`.

Tämän korjauksen jälkeen testikohde jää kiinni tyhjällä taulukolla, kun avain on 3. Se väittää, että avain löytyy paikasta 0, vaikka tyhjässä taulukossa ei ole paikkaa 0.

Tyhjällä taulukolla testikohde aloittaa asettamalla *ala*:ksi 0 ja *yla*:ksi -1. Sitten se siirtyy riviltä 3 suoraan riville 9. Siellä se todellakin kokeilee, onko $A[0] = 3$. Näemmä se oli, koska testikohde palautti 0 eikä -1. Vaikuttaa siltä, että olematonta alkiota 0 vastaavassa kohdassa muistia oli 3 sen jäljiltä, että juuri ennen tyhjällä taulukolla testaamista oli testattu taulukolla $[3,6,7]$.

Kun tyhjällä taulukolla testaaminen siirrettiin ensimmäiseksi heti taulukon luonnin jälkeen, testikohde kaatui. Tällekin ilmiölle on keksittävissä uskottava selitys, mutta se edellyttää taustatietoja, jotka kerrotaan vasta luvussa 3.3.

Riippumatta siitä seuraako olemattoman alkion käyttämisestä väärä vastaus vai ohjelman kaatuminen, on se virhe joka pitää korjata. Se onnistuu lisäämällä rivin 9 testin alkuun `A.size() > 0 &&`. Niin korjattu aliohjelma läpäisee tähänastiset testit. Se on altis Blochin kertomalle lukualueen ylivuodosta aiheutuvalle virheelle, mutta muita virheitä siinä ei tiedetä olevan.

1.5 Toimiva puolitushaku

Edellä kuvatussa puolitushaussa on korjattunakin ominaisuus, joka vähentää sen käyttökelpoisuutta: jos avain on taulukossa monena kappaleena, se löytää jonkin niistä eikä välttämättä esimerkiksi ensimmäistä. Siitä seuraa, että jos halutaan käsitellä kaikki alkiot, joilla on sama avain, niin joudutaan tutkimaan löydetystä kohdasta molempiin suuntiin. Olisi yksinkertaisempaa, jos puolitushaku aina palaut-

taisi yhtäsuurista ensimmäisen, jolloin riittäisi tehdä jatkotutkimukset löydetystä kohdasta alkaen eteenpäin.

Lisäksi joissakin tilanteissa olisi hyötyä siitä, että jos avaimen kanssa täsmälleen yhtäsuurta alkiota ei taulukossa ole, niin puolitushaku palauttaisi lähinnä suuremman alkion paikan. Silloin käyttäjä voisi tutkia, onko siinä tai edellisessä paikassa oleva alkio (mikäli niissä on alkiot) riittävän hyvä likiarvo sille, jota etsittiin. Kun lasketaan tentin arvosanaa pistemäärän ja pisterajojen perusteella, niin etsitään yhtäsuuren tai lähinnä pienemmän alkion paikkaa. Se on muuten sama ongelma kuin edellä, mutta vastakkaiseen suuntaan.

Näillä pohdinnoilla tavoitteeksi tulee löytää vasemmalta alkaen ensimmäinen kohta, jossa oleva alkio on vähintään yhtäsuuri kuin avain. Jos mikään taulukon alkio ei ole niin suuri, niin pitää palauttaa taulukon viimeisen alkion jälkeen tuleva kohta, eli `A.size()`. Tämä kohta on itse asiassa se luku, joka `ii`:ssä olisi kuvan 3 rivien 7, ..., 9 silmukan jälkeen, jos `ii` olisi luotu ennen silmukkaa niin että se olisi olemassa myös silmukan jälkeen. Käyttäjän tehtäväksi jää tarkastaa, onko palautettu arvo `A.size()`, ja jollei ole niin onko siinä kohdassa taulukkoa avaimen kanssa yhtäsuuri vai avainta suurempi alkio.

Merkitsemme $n =$ `A.size()`. Löydetty kohta i on oikea, jos ja vain jos seuraavat kolme väitettä pätevät sille:

1. i on tulokselle sallitulla välillä: $0 \leq i \leq n$
2. i ei ole liian suuri: $i = 0$ tai $A[i-1] < avain$
3. i ei ole liian pieni: $i = n$ tai $A[i] \geq avain$

Kuvan 3 `testaa` saadaan muutettua tälle uudelle vaatimukselle korvaamalla rivit 4, ..., 12 seuraavalla:

```
bool ok = ( 0 <= tulos && tulos <= nn );
if( ok ){ ok &= ( tulos == 0 || A[ tulos-1 ] < avain ); }
if( ok ){ ok &= ( tulos == nn || A[ tulos ] >= avain ); }
```

Enää testaajan ei tarvitse selvittää, onko avain taulukossa!

Tätä testaajaa kokeiltiin kolmella virheellisellä uusien vaatimusten toteutusyrityksellä, ja se löysi virheet. Se tarina ei kuitenkaan toisi lisäarvoa siihen, jonka jo kerroimme edellisillä vaatimuksilla. Siksi siirrymme nyt versioon, jossa ei tiedetä olevan muita vikoja kuin Blochin kuvaama ylivuotovirhe. Se ja sen oikeellisuuden perustelu ovat kuvassa 5.

Sivusuunnassa suunnilleen keskellä kuvaa on muuttujien *ala*, *yla* ja *vali* arvoja koskevia väitteitä. Perustelemme, että kukin niistä pätee aina sen rivin lopussa, jonka kohdalla se on esitetty.

On selvää, että rivin 1 lopussa $ala = 0 \leq n = yla$. Siitä seuraa, että jos riville 2 tullaan riviltä 1, niin rivin 2 lopussa $0 \leq ala \leq yla \leq n$. Sieltä jatketaan riville

1	`int ala = 0, yla = A.size();`	$ala = 0 \leq n = yla$	$ala = 0$ ja $yla = n$
2	`while( ala < yla ){`	$0 \leq ala \leq yla \leq n$	(*)
3	`  int vali = (ala + yla)/2;`	$ala \leq vali < yla$	
4	`  if( A[vali] >= avain )`	$ala \leq vali < yla$	$A[vali] \geq avain$
5	`  { yla = vali; }`	yla pienenee; $ala \leq yla$	$A[yla] \geq avain$
6	`  else`	$ala < vali + 1 \leq yla$	$A[vali] < avain$
7	`  { ala = vali+1; }`	ala kasvaa; $ala \leq yla$	$A[ala-1] < avain$
8	`}`		
9	`return ala;`	$0 \leq ala = yla \leq n$	(*)

(*): ($ala = 0$ tai $A[ala-1] < avain$) ja ($yla = n$ tai $A[yla] \geq avain$)

Kuva 5: Toimiva puolitushaku ja sen oikeellisuuden perustelu

3 tai 9 riippuen siitä, päteekö $ala < yla$ vai $ala = yla$. Perustelemme kohta, että rivien 3, ..., 7 väitteet pätevät. Niistä seuraa, että kun riville 2 tullaan riviltä 8, niin silloinkin $0 \leq ala \leq yla \leq n$ pätee. Riville 2 ei voida tulla muualta kuin riviltä 1 tai 8. Siksi $0 \leq ala \leq yla \leq n$ pätee aina rivillä 2.

Rivillä 3 pätee $ala < yla$, koska muutoin riviltä 2 ei oltaisi tultu sinne. Siksi $ala < \frac{ala+yla}{2} < yla$. Luku $\frac{ala+yla}{2}$ voi olla puoliluku, kuten $5\frac{1}{2}$. Koska *vali* on kokonaislukutyyppiä, se ei pysty esittämään puolilukuja tarkasti. Siksi *vali*:in sijoitettava arvo on pyöristetty alaspäin lähimpään kokonaislukuun. Niinpä *vali*:in sijoitettava arvo on $\frac{ala+yla}{2}$ tai $\frac{ala+yla}{2} - \frac{1}{2}$. Siksi rivin 3 lopussa $ala \leq vali < yla$.

Rivin 4 väite $ala \leq vali < yla$ pätee siksi, että osoitimme sen pätevän rivin 3 lopussa, eikä sen jälkeen minkään muuttujan arvo ole muuttunut. Myös riville 6 on kopioitu rivin 3 lopun väitteen asiasisältö, mutta sen esitystapaa on muutettu. Koska *ala*, *vali* ja *yla* ovat kokonaislukuja, tarkoittaa $ala \leq vali$ samaa kuin $ala < vali + 1$ ja $vali < yla$ samaa kuin $vali + 1 \leq yla$.

Rivillä 5 *yla* pienenee, koska siihen sijoitetaan arvo joka rivin 4 väitteen mukaan on pienempi kuin sen aikaisempi arvo. Se arvo mikä rivillä 4 oli muuttujassa *vali* kopioitiin muuttujaan *yla* rivillä 5, joten, ja koska $ala \leq vali$ päti rivillä 4, pätee $ala \leq yla$ rivin 5 lopussa. Vastaavasti rivillä 7 *ala*:an sijoitetaan sen aikaisempaa arvoa suurempi arvo. Se arvo minkä lauseke $vali + 1$ tuotti rivillä 6 kopioitiin muuttujaan *ala* rivillä 7, joten, ja koska $vali + 1 \leq yla$ päti rivillä 6, pätee $ala \leq yla$ rivin 7 lopussa. Riveillä 2, ..., 8 pätevät koko ajan $0 \leq ala$ ja $yla \leq n$. Niitä ei ole toistettu rivin 2 jälkeen, jotta kuva säilyisi helpompana lukea.

Oikeanpuoleisen palstan ylimmän rivin väite on ilmeinen. Kun riville 2 tullaan riviltä 1, pätee (*) siten, että sen kummankin osan ensimmäinen vaihtoehto pätee. Rivien 4 ja 6 väitteet pätevät, koska rivillä 4 testattiin kumpi niistä pätee, ja jatkettiin sen mukaan rivin 4 loppuun tai riville 6. Rivin 5 väite seuraa rivin 5 sijoituksesta ja rivin 4 väitteestä keskimmäisen palstan kohdalla selostetulla tavalla. Niinpä rivillä 2 pätee $yla = n$ tai $A[yla] \geq avain$ sen mukaan, onko riviä 5 koskaan

suoritettu. Vastaava pätee riveille 7 ja 6, koska rivin 7 jälkeen $ala - 1$ tuottaa saman kuin *vali* tuotti rivillä 6. Niinpä rivillä 2 pätee $ala = 0$ tai $A[ala - 1] < avain$ sen mukaan, onko riviä 7 koskaan suoritettu.

Koska riville 9 tullaan riviltä 2 ilman että minkään muuttujan arvo muuttuu, pätee rivillä 9 kaikki mikä päti rivillä 2. Lisäksi rivillä 9 pätee $ala \geq yla$, koska muuten ei oltaisi menty riville 9 vaan riville 3. Tiedot $ala \leq yla$ ja $ala \geq yla$ tuottavat yhdessä $ala = yla$. Kun tietää että $ala = yla$, niin edellä mainitut puolitushaun tulokselta vaaditut ominaisuudet 1, 2 ja 3 voi melkein suoraan lukea rivillä 9 luvatuista asioista. Niinpä ne pätevät sille arvolle, jonka kuvan 5 puolitushaku palauttaa.

Koska $0 \leq ala \leq vali < yla \leq n$ rivillä 4, pätee $0 \leq vali < n$ rivillä 4, joten rivin 4 indeksointi on aina laillinen. Rivillä 7 ei voi tapahtua ylivuotoa, koska $vali + 1 \leq yla$. Rivillä 3 voi tapahtua Blochin kuvaama ylivuotovirhe. On melko helppo perustella, että se voidaan välttää kirjoittamalla rivin 3 oikea puoli uudelleen muotoon `ala + (yla-ala)/2`. Emme kuitenkaan mene nyt siihen, jotta emme joutuisi liian kauas pääasiasta. Rivillä 1 voi tapahtua ylivuoto, jos `A` on niin suuri, että sen koko ei mahdu `int`-tyyppiin. Mitään muuta laitonta ei kuvassa 5 voi tapahtua.

Vielä tarvitsee perustella, että riville 9 tullaan eli että rivien $2, \ldots, 8$ silmukka lopettaa joskus. Koska jokaisella kierroksella *yla* pienenee ilman että *ala* muuttuu, tai *ala* kasvaa ilman että *yla* muuttuu, pienenee $yla - ala$ joka kierroksella. Niinpä silmukka lopettaa viimeistään $yla - ala$ kierroksen jälkeen. Muilla keinoin voidaan osoittaa, että jos $yla - ala$ on suuri, niin se lopettaa huomattavasti nopeammin.

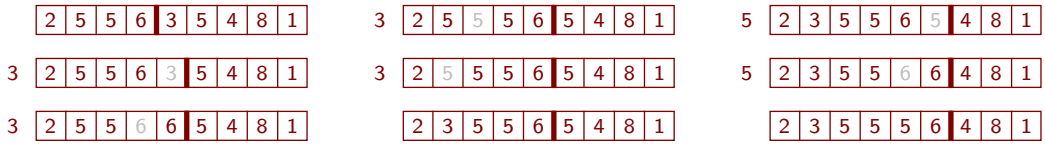
2 Silmukkainvariantit ja lisäysjärjestäminen

Silmukkainvariantti (***loop invariant***) on tehokas keino varmistaa, että ohjelmassa oleva silmukka toimii oikein. Tässä luvussa tutustutaan niiden käyttöön käyttäen esimerkkinä algoritmia, jonka nimi on ***lisäysjärjestäminen*** (***insertion sort***). Luvun lopussa tarkastellaan lisäysjärjestämistä myös muista näkökulmista, koska lisäysjärjestäminen on itsessäänkin tärkeä asia. Se on paras tunnettu yleiskäyttöinen menetelmä järjestää pieni taulukko suuruusjärjestykseen. Niinpä sitä tarvitaan usein ja monenlaisissa käyttötilanteissa. Se on niin yksinkertainen, että on helppoa toteuttaa se itse.

2.1 Lisäysjärjestäminen

Järjestettävä taulukko jakaantuu lisäysjärjestämisen toiminnan aikana kahteen osaan. Taulukon alkuosa on kasvavassa järjestyksessä, ja loppuosalle ei ole vielä tehty mitään. Kuvassa 6 on esimerkki, jossa osien välinen raja on esitetty paksulla pystyviivalla.

Aluksi alkuosassa on vain yksi alkio. Alkuosaa kasvatetaan yksi lokero kerrallaan kunnes jokainen alkio on alkuosassa ja loppuosa on tyhjä. Kasvattaminen tapahtuu ottamalla loppuosan ensimmäinen alkio sivuun, siirtämällä sen vasemmalla puolella olleet sitä suuremmat alkiot yksi kerrallaan yhden lokeron verran oikealle, ja sijoittamalla sivuun otettu alkio viimeisen siirretyn alkion alta vapautuneeseen lokeroon.



Kuva 6: Esimerkki lisäysjärjestämisen (insertion sort) toiminnasta

```
    INSERTIONSORT(&A)
1   for i := 1 to A.koko − 1 do
2     apu := A[i]; j := i
3     while j > 0 && A[j − 1].x > apu.x do
4       A[j] := A[j − 1]; j = j − 1
5     A[j] := apu
```

```
typedef std::vector< alkio > taulukko;
void InsertionSort( taulukko & A ){
  for( unsigned i=1; i<A.size(); ++i ){
    alkio apu = A[i]; unsigned j = i;
    while( j > 0 && A[j-1].x > apu.x ){
      A[j] = A[j-1]; --j;
    }
    A[j] = apu;
  }
}
```

Kuva 7: Lisäysjärjestäminen (insertion sort) pseudokoodina ja C++-koodina

Kuvassa 6 on näytetty ensin alkion 3 ja sitten alkion 5 siirtäminen loppuosasta alkuosaan. Kukin himmeänharmaa numero näyttää sellaisen lokeron edellisen sisällön, josta alkio on siirretty pois ja johon ei vielä ole siirretty muuta alkiota tilalle. Vaikka algoritmimme näkökulmasta alkioita siirretään, ohjelmointikielen tasolla kyse on todellisuudessa kopioimisesta. Niinpä myös himmeänharmaat numerot ovat todellisuudessa taulukossa, mutta algoritmimme ei enää käytä niitä.

15. Piirrä kuvan 6 taulukon sisältö sen jälkeen kun alkio 4 on siirretty alkuosaan. Piirrä myös sen jälkeen kun alkio 8 ja sen jälkeen kun alkio 1 on siirretty alkuosaan.

Kuvassa 7 on näytetty pseudokoodina ja C++-koodina lisäysjärjestäminen taulukolle A, jonka lokerot sijaitsevat paikoissa $0, 1, \ldots, n-1$. Luvun n saa kysyttyä A:lta toiminnolla $A.koko$. Merkki & ilmauksessa $\&A$ määrää, että INSERTIONSORT käyttää samaa taulukkoa kuin se ohjelman osa, jossa sitä kutsutaan. Siksi jos kutsuna on vaikka INSERTIONSORT(*opintojaksot*), niin INSERTIONSORT järjestää taulukon *opintojaksot*. Ilman merkkiä & INSERTIONSORT saisi alkuperäisestä taulukosta kopion. Siinä tapauksessa INSERTIONSORT järjestäisi kopion eli A:n, mutta *opintojaksot* jäisi muuttumatta.

Muuttujassa i on sen lokeron numero, joka siirtyy loppuosan alusta alkuosan loppuun. Sivuun otettu alkio on muuttujassa *apu*. Toiminnan selittämisen helpottamiseksi kutsumme ”aukoksi” sitä lokeroa, jossa ollut alkio on kopioitu yhden lokeron verran oikealle tai muuttujaan *apu*. Muuttuja j sisältää aukon numeron. Rivillä 2 kopioidaan lokerossa i oleva alkio muuttujaan *apu* ja asetetaan aukon numeroksi i. Rivillä 4 kopioidaan aukon vasemmalla puolella oleva alkio aukkoon, sekä asetetaan aukon numero osoittamaan sitä lokeroa, josta alkio kopioitiin. Sivuun otettu alkio sijoitetaan aukkoon rivillä 5.

On tavallista, että alkioiden järjestyksen määräämisessä käytetään vain osaa alkioiden sisällöstä. Esimerkiksi sähköposteja järjestettäessä käytetään usein lähetysaikaa, joskus otsikkoa ja toisinaan lähettäjän sähköpostiosoitetta. Tämä on otettu huomioon kuvassa 7 siten, että rivillä 3 ei verrata alkioita kokonaisuudes-

saan vaan ainoastaan niiden sisältämiä tietoja x. Alkioiden vertaamisessa käytettävää osaa alkion tiedoista kutsutaan usein ***avaimeksi*** (***key***). Kuvassa se on kuitenkin x, koska pitempi nimi aiheutti ongelmia algoritmin mahduttamisessa luentoruuduille sitä koskevien kommenttien yhteyteen.

Ehtolauseke *vasen* && *oikea* lasketaan siten, että ensin lasketaan *vasen*. Jos se tuottaa ”tosi”, lasketaan *oikea* ja palautetaan sen tulos koko lausekkeen tuloksena. Jos *vasen* tuottaa ”epätosi”, jätetään *oikea* laskematta ja *vasen* && *oikea* tuottaa ”epätosi”. Sen ansiosta rivillä 3 ei yritetä laskea $A[j-1].x$ silloin kun $j = 0$. Tämä on tärkeää, sillä sen laskemisessa luettaisiin lokeron -1 sisältö. Se on kiellettyä, sillä se voisi johtaa ohjelmamme kaatumiseen, koska A:ssa ei ole lokeroa -1.

16. Se, onko rivillä 3 koskaan INSERTIONSORT:in suorituksen aikana $j = 0$, riippuu taulukon A alkuperäisestä sisällöstä. Taulukot voidaan jakaa kahteen ryhmään: niihin, joilla rivillä 3 ei koskaan ole $j = 0$, ja niihin, joilla rivillä 3 on ainakin kerran $j = 0$. Kerro, minkälaisilla taulukoilla on ainakin kerran $j = 0$! Kuinka monta alkiota taulukossa on tällöin?

Kuvan 7 C++-koodissa `typedef std::vector< alkio > taulukko;` määrää, että `taulukko` on taulukko, jonka alkiot ovat tyyppiä `alkio`. Niinpä `apu` on samaa tyyppiä kuin taulukon alkiot. C++:ssa on kaksi erilaista taulukkoa: C-kielestä peräisin oleva jota kutsutaan englanniksi ”array” ja nykyaikaisempi `vector`. Jälkimmäisen kokoa voi muuttaa suorituksen aikana ja koon voi kysyä toiminnolla `size`.

C++:ssa `size()` on etumerkitöntä kokonaislukutyyppiä. Vertailun `i<A.size()` vuoksi C++-kääntäjä toivoo, että myös `i` on etumerkitön. On luontevaa, että `j` on samaa tyyppiä kuin `i`. Siksi muuttujien `i` ja `j` tyypiksi on valittu etumerkitön kokonaislukutyyppi `unsigned`. Sen luvataan kattavan ainakin kokonaisluvut $0, 1, \ldots, 65\,535$. Tavallisissa nykyaikaisissa tietokoneissa `unsigned` kattaa ainakin kokonaisluvut $0, 1, \ldots, 4\,294\,967\,295$, mutta yläraja voi olla suurempikin. Jos on vaara, että tämä ei riitä, niin kannattaa käyttää tyyppiä `std::size_t`. Se on etumerkitön kokonaislukutyyppi, jonka taataan aina olevan riittävän suuri taulukoiden indeksointiin.

2.2 Todistuksen jakaminen tapauksiin ja tapaus $n = 0$

Luvuissa 2.3, ..., 2.6 esitettävä todistus on pätevä kaikille muille taulukoille, mutta ei tyhjälle taulukolle. Siksi joudumme todistamaan erikseen, että INSERTIONSORT toimii oikein tyhjälle taulukolle.

Taulukko $A[0 \ldots n-1]$ on tyhjä jos ja vain jos $n = 0$. Tällöin **for**-silmukka ei kierrä yhtään kierrosta. Niinpä INSERTIONSORT ei tee A:lle mitään. Siis jos syötteenä on tyhjä taulukko, niin lopputuloksena on sama. Se on oikein, sillä tyhjä taulukko on jo valmiiksi oikeassa järjestyksessä.

17. Luvuissa 2.3, ..., 2.6 esitettävä todistus kattaa myös tapauksen $n = 1$, joten sitä ei tarvitsisi käsitellä erikseen. Sen käsitteleminen erikseen on kuitenkin hyödyl-

linen pieni harjoitustehtävä. Niinpä: miksi INSERTIONSORT toimii oikein silloin kun $n = 1$?

2.3 Ulompi silmukka paitsi sen runko

Silmukkainvariantti on väite, josta voidaan osoittaa kolme asiaa:

IE Se on tosi, kun silmukan alkuun tullaan silmukan edeltä.

IS Jos se on tosi kun ollaan silmukan alussa, ja jos lisäksi silmukan ehto on tosi, niin silmukkainvariantti on tosi myös kun alkuun tullaan uudelleen silmukan rungon suorittamisen jälkeen.

IH Jos se on tosi kun silmukan kiertäminen lopetetaan tai silmukka ohitetaan, niin silmukalta haluttu asia on tosi.

Kohdista IE ja IS seuraa, että silmukkainvariantti on tosi joka kerta kun ollaan silmukan alussa. Ne takaavat sen hiukan epäsuorasti, sillä tarkkaan ottaen ne takaavat vain, että koskaan ei ole ***ensimmäinen kerta***, jolloin silmukkainvariantti ei ole tosi silmukan alussa. Mutta se riittää, sillä jos ei ole olemassa ensimmäistä kertaa, niin ei ole olemassa myöskään toista, kolmatta, ... kertaa. Kohta IE takaa, että ensimmäinen kerta ei voi olla silloin, kun silmukan alkuun tullaan silmukan edeltä. Jos silmukkainvariantti ei ole tosi kun silmukan alkuun tullaan silmukan rungon suorittamisen jälkeen, niin kohdan IS vuoksi silmukkainvariantti ei ollut tosi silloin kun oltiin edellisen kerran silmukan alussa, joten nykyinen kerta ei ole ensimmäinen.

Merkinnällä $A[a \ldots y]$ tarkoitetaan taulukon A osaa, joka kattaa alkiot $A[a]$, $A[a+1]$, ..., $A[y]$. Jos $y = a-1$, niin $A[a \ldots y]$ on tyhjä. Tätä merkintää ei saa käyttää siten, että $y < a-1$. Merkintää voidaan käyttää myös ilmaisemaan koko taulukon nimi, ensimmäisen alkion paikka ja viimeisen alkion paikka. Taulukko, jonka INSERTIONSORT järjestää, on $A[0 \ldots n-1]$.

18. Kuinka monta alkiota on taulukossa $A[a \ldots y]$?

19. Taulukko $A[0 \ldots n-1]$ jaetaan alkuosaan, jossa on i alkiota, ja loppuosaan, jossa on loput alkiot. Ilmoita alkuosan ja loppuosan indeksialueet!

20. Mitkä arvot kysymyksen 19 i voi saada? Miksi vain ne?

Kuten luvussa 2.2 kerrottiin, tässä luvussa A ei ole tyhjä. Toisin sanoen $n \geq 1$. INSERTIONSORT:in ulomman silmukan eli **for**-silmukan toiminta voidaan osoittaa oikeaksi seuraavalla invariantilla. Siinä i:n arvona käytetään ensimmäisellä kerralla i:n ensimmäistä arvoa eli 1, ja muilla kerroilla i:n edellistä arvoa kasvatettuna yhdellä. Alkuperäisillä alkioilla ja alkuperäisellä järjestyksellä tarkoitetaan A:n sisältöä kun INSERTIONSORT:in suorittaminen alkoi.

1. Osassa $A[0 \ldots i-1]$ on alkuperäiset alkiot kasvavassa järjestyksessä.
2. Osassa $A[i \ldots n-1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä.

Pohdimme ensin, miksi tapausta $n = 0$ ei käsitellä tässä, vaan se käsiteltiin erikseen luvussa 2.2. INSERTIONSORT:in suorituksen alussa $i = 1$. Osa $A[0 \ldots i-1]$ on silloin $A[0 \ldots 0]$. Siinä on täsmälleen yksi alkio, nimittäin $A[0]$. Kun $n = 0$, ei A:ssa ole yhtään alkiota, joten $A[0]$ ei ole A:n alkio. Siinä tilanteessa ei ole totta, että osassa $A[0 \ldots i-1]$ on alkuperäiset alkiot. Niinpä invarianttimme osa 1 ei ole totta eikä invarianttiamme voi käyttää. Lisäksi osa $A[i \ldots n-1]$ on silloin $A[1 \ldots -1]$. Se rikkoo sitä, että ilmauksessa $A[a \ldots y]$ ei saa olla $y < a-1$. Siksikään invarianttiamme ei voi käyttää.

Tämä ei tarkoita että INSERTIONSORT toimisi väärin kun $n = 0$, vaan tämä tarkoittaa vain että luvuissa 2.3, ..., 2.6 esitettävä päättely ei ole pätevä kun $n = 0$. INSERTIONSORT on luvussa 2.2 osoitettu oikeaksi kun $n = 0$, ja luvuissa 2.3, ..., 2.6 se osoitetaan oikeaksi kun $n \geq 1$. Koska taulukossa on aina joko nolla alkiota tai vähintään yksi alkio, kattavat nämä kaksi todistusta kaikki mahdollisuudet, joten INSERTIONSORT toimii oikein jokaisella taulukolla.

Osoitamme seuraavaksi, että invarianttimme toteuttaa vaatimuksen IE. Ulomman silmukan alkuun tullaan sen edeltä INSERTIONSORT:in suorituksen aikana vain yhden kerran, nimittäin silloin kun suoritus alkaa. Silloin $i = 1$ ja A:ssa on alkuperäiset alkiot alkuperäisillä paikoillaan. Niinpä sekä osassa $A[0 \ldots i-1]$ että osassa $A[i \ldots n-1]$ on osan alkuperäiset alkiot alkuperäisessä järjestyksessä (ja kumpikin osa todella on A:n osa). Koska $i = 1$, on osassa $A[0 \ldots i-1]$ täsmälleen yksi alkio. Jokainen yhden alkion taulukko on kasvavassa suuruusjärjestyksessä, joten $A[0 \ldots i-1]$ on kasvavassa suuruusjärjestyksessä. Invariantin jokainen yksityiskohta on nyt osoitettu.

Vaatimuksen IS toteutumisen osoittaminen on monimutkaista, koska ulomman silmukan runko sisältää toisen silmukan. Siksi käsittelemme sen myöhemmin luvussa 2.6.

Vaatimuksen IH tarkastamiseksi tulee osoittaa, että kun ulommasta silmukasta tullaan ulos tai se ohitetaan, on siltä haluttu asia tosi. Tässä tapauksessa siltä halutaan, että koko taulukossa on alkuperäiset alkiot kasvavassa järjestyksessä.

Sitä varten osoitamme ensin, että kun ulompi silmukka on suoritettu tai ohitettu, pätee $i = n$. Ulommassa silmukassa ei ole lauseita **break**, **return** tai mitään muutakaan niiden kaltaista. Siksi ainoa tapa tulla siitä ulos on tulla riville 1 niin että $i \geq n$. Silloin $i = n$, koska i kasvaa yhden kerrallaan. Ulompi silmukka ohitetaan jos ja vain jos $i \geq n$ heti alussa. Heti alussa $i = 1$. Koska olemme oletuksen $n \geq 1$ piirissä, saamme $1 = i \geq n \geq 1$, joten $i = n = 1$.

Nyt kun tiedämme, että $i = n$, on helppo osoittaa, että jos invariantti pätee, niin koko taulukossa on alkuperäiset alkiot kasvavassa järjestyksessä. Invariantin

osa 1 lupaa, että osassa $A[0 \ldots i-1]$ on alkuperäiset alkiot kasvavassa järjestyksessä. Koska $i = n$, on osa $A[0 \ldots i-1]$ sama kuin koko taulukko. Kohta IH on nyt osoitettu.

2.4 Ulomman silmukan lopettaminen

Kohdat IE, IS ja IH takaavat, että haluttu asia pätee kun silmukka on lopettanut tai ohitettu. Ne eivät kuitenkaan takaa, että silmukka lopettaa. Sen osoittamiseksi on osoitettava, että silmukkaan ei jäädä pyörimään ikuisesti ja että silmukan rungon suoritus lopettaa.

Se, että ulompaan silmukkaan ei jäädä pyörimään ikuisesti, seuraa välittömästi siitä, että ulompi silmukka on niin sanottu ***aito for-silmukka***. Aito **for**-silmukka esitetään usein muodossa

for *i* := *alku* **to** *loppu* **do** *runko*

(On olemassa myös monimutkaisempia aitoja **for**-silmukoita, mutta niitä ei käsitellä nyt.) Siinä *i* on ***silmukkamuuttuja*** (***loop variable***). Silmukka kasvattaa sitä joka kierroksella alkaen lausekkeen *alku* arvosta kunnes se saavuttaa arvon, joka lausekkeella *loppu* oli silloin, kun silmukka aloitti, tai kunnes silmukasta poistutaan kesken kaiken esimerkiksi **return**-lauseella. Silmukassa ei saa olla mitään muuta, joka voisi mahdollisesti vaikuttaa silmukkamuuttujan arvoon. INSERTION-SORT:in ulomman silmukan silmukkamuuttuja on *i*, *alku* on 1 ja *loppu* on $n-1$.

Jos *loppu* $<$ *alku*, niin silmukka ei suorita yhtään kierrosta. Muussa tapauksessa silmukka suorittaa enintään *loppu* $-$ *alku* $+1$ kierrosta.

Silmukkamuuttujan arvoon ei saa vaikuttaa silmukan rungossa, koska jos saisi, niin esimerkiksi **for** $i := 1$ **to** 8 **do** $i := 5$ olisi päättymätön aito **for**-silmukka. Lauseketta *loppu* ei lasketa joka kierroksella uudelleen, koska jos laskettaisiin, niin esimerkiksi **for** $i := n$ **to** $n+5$ **do** $n := n+1$ olisi päättymätön aito **for**-silmukka. (Sitäpaitsi uudelleen laskeminen kuluttaisi myös turhaan aikaa.) Joissakin ohjelmointikielissä kääntäjä estää yritykset vaikuttaa silmukkamuuttujan arvoon ja huolehtii, että lopetus tarkastetaan lausekkeen *loppu* silmukan aloitushetkisen arvon mukaan. Muiden ohjelmointikielten tapauksessa ohjelmoijan on varmistuttava niistä itse.

Kuvan 7 oikealla puolella ei ole muita lauseita, jotka vaikuttaisivat muuttujan `i` arvoon, kuin `for`-silmukkaan kuuluva `++i` rivillä 1. Siinä ei myöskään ole lauseita, jotka vaikuttaisivat muuttujan `n` arvoon, joten vaikka jatkamisehto `i < n` lasketaan joka kierroksella uudelleen senhetkisellä `n`:n arvolla, on `n`:n arvo aina sama kuin alussa. Niinpä myös kuvan 7 `for`-silmukka on aito **for**-silmukka.

Siis kuvassa 7 sekä pseudokoodin että ohjelmakoodin ulompi silmukka on aito **for**-silmukka. Tämä tieto riittää takaamaan, että ne lopettavat.

Sen, että ulomman silmukan runko lopettaa, osoitamme luvussa 2.6.

2.5 Ei laittomia toimintoja

Siinä määrin kuin tarkoituksenmukaista, ohjelman osoittaminen oikein toimivaksi sisältää myös sen tarkastamisen, että minkään lauseen suorittamisen aikana ei tapahdu mitään laitonta. Esimerkiksi muistin loppumista vastaan suojautuminen ei ole tarkoituksenmukaista silloin, kun tiedetään syötteiden olevan pieniä verrattuna tietokoneessa olevan muistin määrään. Silloinkin voi silti olla hyvä kertoa, että ohjelma ei ole varautunut muistin loppumiseen. INSERTIONSORT:in tapauksessa voidaan ja riittää osoittaa, että A:sta ei yritetä käyttää alkioita joita siinä ei ole, ja että laskutoimituksissa ja sijoituksissa ei tapahdu numeerisia ylivuotoja eikä tyyppivirheitä.

Tarkastamme ensin A:n indeksoinnit. Koska ulompi silmukka on aito **for**-silmukka, riveillä $2, \ldots, 5$ pätee $1 \leq i < n$. Koska rivin 2 lopussa $j = i \geq 1$, sisempi silmukka ei muuta j:tä muuten kuin pienentämällä yksi kerrallaan, ja se lopettaa viimeistään kun $j = 0$, pätee kaikkialla $0 \leq j \leq i < n$. Siksi jokainen $A[i]$ ja $A[j]$ on laillinen. Indeksointien $A[j-1]$ kohdalle tullaan vain kun rivillä 3 on todettu, että $0 < j$. Sen jälkeen j ei ole muuttunut. Niinpä $0 \leq j - 1$. Koska $j < n$, pätee $j - 1 < n$. Siksi jokainen $A[j-1]$ laillinen.

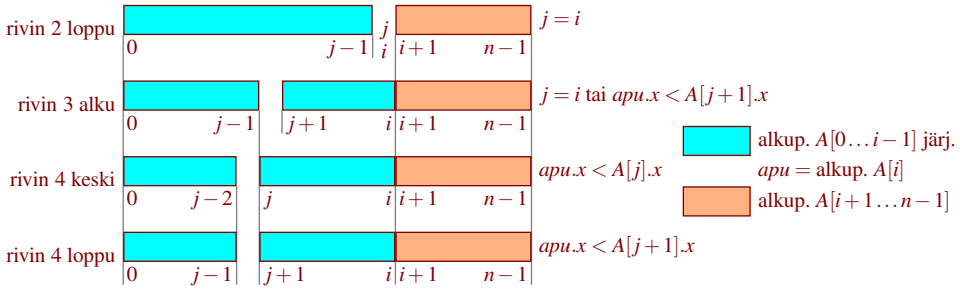
21. Miksi päättelyn kannalta on tärkeää että rivin 2 lopussa $j = i \geq 1$, miksei tieto $j = i$ riitä?

Muiden asioiden tarkastamiseksi tulisi tietää muuttujien tyypit. Kuten algoritmikirjallisuudessa on tapana, kuvan 7 pseudokoodin selkeyttämiseksi niitä ei kerrota siinä, vaan oletetaan, että ohjelmoija valitsee ne tarkoituksenmukaisesti. Ylivuotoja ja tyyppivirheitä ei tapahdu, jos *apu* on samaa tyyppiä kuin A:n alkiot, ja i, j ja n ovat niin suurta kokonaislukutyyppiä, että se kattaa ainakin luvut $0, \ldots, n$. Muuttujaa i kasvatetaan vain kun $i < n$, joten lopputulos ei voi olla suurempi kuin n. Ennen j:n pienentämistä on todettu että $j > 0$, joten lopputulos ei voi olla pienempi kuin 0. Koska $n - 1 < 0$ kun $n = 0$, täytyy lisäksi tavalla tai toisella varmistaa, että **for**-silmukka ei suorita yhtään kierrosta kun $n = 0$.

2.6 Ulomman silmukan runko

Vielä tarvitsee osoittaa, että jos ulomman silmukan invariantti pätee rivillä 1 ja $i < n$, niin ulomman silmukan invariantti pätee kun riville 1 tullaan uudelleen. Koska **for**-silmukka kasvattaa i:tä yhdellä, tarkoittaa tämä, että kun ulomman silmukan runko on suoritettu, täytyy päteä väite, joka saadaan invariantista käyttämällä i:n tilalla yhtä suurempaa lukua. Toisin sanoen, seuraavan väitteen täytyy päteä:

1. Osassa $A[0 \ldots i]$ on alkuperäiset alkiot kasvavassa järjestyksessä.
2. Osassa $A[i+1 \ldots n-1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä.



Kuva 8: Sisemmän silmukan invariantti ja sen säilyminen

Ulomman silmukan rungossa on kolme osaa: alkion $A[i]$ kopiointi muuttujaan *apu* ja sijoitus $j := i$ (rivi 2), sisempi silmukka (rivit 3 ja 4), sekä muuttujan *apu* kopiointi alkioksi $A[j]$ (rivi 5). Rungosta ei tulla ulos muualta kuin rivin 5 lopusta.

Osoitamme ensin invariantin osan 2 eli että osataulukossa $A[i+1\ldots n-1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä. Sen rajat eivät muutu riveillä 2, ..., 5, koska i:n ja n:n arvot eivät muutu niillä riveillä. Koska A:han ei sijoiteta muualle kuin lokeroon j, ja koska $j \leq i$, ei osan $A[i+1\ldots n-1]$ sisältö muutu. Siksi osassa $A[i+1\ldots n-1]$ on rivin 5 lopussa samat alkiot samassa järjestyksessä kuin rivin 2 alussa. Ne ovat alkuperäiset alkiot alkuperäisessä järjestyksessä, koska ulomman silmukan invariantin osa 2 lupaa niin osataulukolle $A[i\ldots n-1]$ ja oli voimassa rivin 2 alussa, ja koska $A[i+1\ldots n-1]$ on osan $A[i\ldots n-1]$ osa.

Vielä on osoitettava, että rivin 5 lopussa on osassa $A[0\ldots i]$ sen alkuperäiset alkiot kasvavassa järjestyksessä. Se onnistuu käyttämällä sisemmälle silmukalle seuraavaa invarianttia:

1. Osissa $A[0\ldots j-1]$ ja $A[j+1\ldots i]$ on yhteensä osan $A[0\ldots i-1]$ alkuperäiset alkiot kasvavassa järjestyksessä.
2. Alkuperäinen $A[i]$ on muuttujassa *apu*.
3. Joko $j = i$ tai $apu.x < A[j+1].x$.

Tätä invarianttia ja sen säilymistä on havainnollistettu kuvassa 8.

Silmukkainvarianttien vaatimus IE sanoo, että invariantin täytyy olla voimassa kun silmukkaan tullaan sen edeltä. On siis osoitettava, että sisemmän silmukan invariantti pätee kun riville 3 tullaan riviltä 2. Alkuperäinen $A[i]$ on muuttujassa *apu*, koska ulomman silmukan invariantin osa 2 lupaa että $A[i]$:llä oli alkuperäinen arvonsa rivillä 1, $A[i]$ ei ole muuttunut sen jälkeen, $A[i]$ kopioitiin *apu*:un rivillä 2, eikä *apu* ole muuttunut sen jälkeen. Osuus ”joko $j = i$ tai ...” on tosi, koska rivin 2 vuoksi $j = i$. Osassa $A[0\ldots i-1]$ on sen alkuperäiset alkiot kasvavassa järjestyksessä, koska ulomman silmukan invariantin osan 1 mukaan se päti rivillä 1, eivätkä osan $A[0\ldots i-1]$ rajat eikä sisältö ole muuttuneet sen jälkeen. Osa $A[j+1\ldots i]$

on tyhjä ja $A[0 \ldots j-1]$ on $A[0 \ldots i-1]$. Niinpä osissa $A[0 \ldots j-1]$ ja $A[j+1 \ldots i]$ on yhteensä osan $A[0 \ldots i-1]$ alkuperäiset alkiot kasvavassa järjestyksessä.

Edellä sanottiin useasti, että jokin ei ole muuttunut sen jälkeen kun suoritus oli jossakin kohdassa. Tällaisten asioiden tarkastaminen on yleensä helppoa eikä vie monta sekuntia. Todistuksissa mukana ollessaan ne vaikeuttavat olennaisempien asioiden lukemista. Niinpä tapana on jättää sellaiset mainitsematta ja luottaa siihen, että lukija huolehtii niistä itse. Sanotaan esimerkiksi "Alkuperäinen $A[i]$ on muuttujassa *apu* rivin 2 ja ulomman silmukan invariantin osan 2 vuoksi. Osassa $A[0 \ldots i-1]$ on sen alkuperäiset alkiot kasvavassa järjestyksessä ulomman silmukan invariantin osan 1 vuoksi."

Vaatimus IS sanoo, että invariantin tulee olla voimassa rivin 4 jälkeen, jos se oli voimassa rivillä 3 ja rivin 3 ehto toteutui. Koska se oli voimassa rivillä 3, on sijoituksen $A[j] := A[j-1]$ jälkeen osissa $A[0 \ldots j-2]$ ja $A[j \ldots i]$ osan $A[0 \ldots i-1]$ alkuperäiset alkiot kasvavassa järjestyksessä. Alkuperäinen $A[i]$ on yhä muuttujassa *apu*. Koska rivin 3 ehto oli voimassa ja $A[j] = A[j-1]$, pätee $A[j].x = A[j-1].x > apu.x$.

Se lukuarvo, joka juuri ennen sijoitusta $j := j-1$ oli muuttujassa j, saadaan heti sijoituksen jälkeen lausekkeella $j+1$. Niinpä ne mitkä juuri ennen sijoitusta olivat $A[0 \ldots j-2], A[j \ldots i]$ ja $A[j]$, ovat heti sijoituksen jälkeen $A[0 \ldots (j+1)-2]$, $A[j+1 \ldots i]$ ja $A[j+1]$. Siksi rivin 4 lopussa osissa $A[0 \ldots j-1]$ ja $A[j+1 \ldots i]$ on osan $A[0 \ldots i-1]$ alkuperäiset alkiot kasvavassa järjestyksessä, alkuperäinen $A[i]$ on muuttujassa *apu*, ja $apu.x < A[j+1].x$. Niinpä sisemmän silmukan invariantti pätee.

22. Perustele, että rivin 4 alussa $A[0 \ldots j-2]$ todella on A:n osan $A[0 \ldots i-1]$ osa.

Vaatimus IH sanoo, että sisemmän silmukan invariantista ja siitä, että $j > 0$ && $A[j-1].x > apu.x$ ei päde, täytyy rivi 5 suorittamalla päästä siihen, että osassa $A[0 \ldots i]$ on alkuperäiset alkiot kasvavassa järjestyksessä. Rivi 5 on $A[j]$:= *apu*. Sen lopussa osassa $A[0 \ldots i]$ ovat ne alkiot, jotka juuri ennen riviä 5 olivat yhteensä osissa $A[0 \ldots j-1]$ ja $A[j+1 \ldots i]$ sekä muuttujassa *apu*. Sisemmän silmukan invariantin osien 1 ja 2 mukaan ne ovat osan $A[0 \ldots i]$ alkuperäiset alkiot.

Vielä tarvitsee osoittaa, että ne ovat kasvavassa järjestyksessä. Sisemmän silmukan invariantin osa 1 lupaa sen muilta osin, mutta ei lupaa että $A[j]$ on oikeassa järjestyksessä suhteessa naapureihinsa osassa $A[0 \ldots i]$. Oikeanpuoleisen naapurin osalta tämä toteutuu, koska se on olemassa vain jos $j < i$, jolloin sisemmän silmukan invariantin osan 3 mukaan $apu.x < A[j+1].x$. Vasemmanpuoleisen naapurin osalta tämä toteutuu, koska se on olemassa vain jos $j > 0$, jolloin silmukasta ei olisi tultu ulos, ellei pätisi $A[j-1].x \leq apu.x$.

Sisempi silmukka lopettaa viimeistään i kierroksen jälkeen, koska j aloittaa arvolla i ja pienenee joka kierroksella yhdellä, silmukka lopettaa viimeistään kun $j = 0$, eikä silmukan rungossa ole mitään mikä voisi jäädä ikuiseen silmukkaan. Silmukka voi lopettaa tätä aikaisemmin ehdon $A[j-1].x > apu.x$ vuoksi.

2.7 Vakaus

Kuten luvussa 2.1 todettiin, alkioita verrattaessa ei välttämättä oteta huomioon kaikkea alkioon sisältyvää tietoa. Esimerkiksi sähköpostit järjestetään usein lähetysajan mukaan niin että mikään muu kuin lähetysaika ei vaikuta järjestämiseen.

Toisinaan halutaan järjestää usean eri perusteen mukaan. Esimerkiksi Wikipedia sanoo korkeushypystä ”Tasatuloksessa korkeammalle sijoittuu kilpailija, jolla on vähemmän hyppyjä siitä korkeudesta, jossa tasatilanne syntyi. Sen ollessa tasan vertaillaan pudotusten kokonaismäärää alemmista korkeuksista, ja vähimmillä pudotuksilla selvinnyt sijoittuu korkeammalle.” Tämä voidaan toteuttaa ottamalla alkioita verrattaessa kaikki perusteet huomioon halutussa järjestyksessä. Se on kuitenkin sikäli kömpelöä, että jokaiselle käytössä olevalle yhdistelmälle tarvitaan oma vertailutoiminto.

Järjestämisalgoritmi on ***vakaa*** (***stable***), jos ja vain jos se ei koskaan muuta kahden yhtäsuuren alkion keskinäistä järjestystä. Vakaudesta seuraa merkittävä etu: voidaan järjestää usean eri kriteerin mukaan ilman että toteutetaan halutulle yhdistelmälle erillinen vertailuoperaattori. Se tapahtuu järjestämällä ensin vähiten merkitsevän kriteerin mukaan, sitten seuraavaksi vähiten merkitsevän ja niin edelleen. Esimerkiksi korkeushypyssä järjestettäisiin ensin alempien korkeuksien pudotusten kokonaismäärän mukaan, sitten hyppääjän tuloskorkeudessaan käyttämien hyppyjen määrän mukaan, ja viimeiseksi takaperin tuloskorkeuden mukaan.

Kuvassa 9 on esimerkki sähköpostien järjestämisestä vakaalla algoritmilla. Aluksi viestit ovat lähetysajan mukaisessa järjestyksessä. Kun käyttäjä klikkaa ne aiheen mukaan kasvavaan järjestykseen, niin juoruviesti nousee ensimmäiseksi ja ”Tsekkaa tämä!” putoaa viimeiseksi. ”Syömään?”-viestit ovat peräkkäin. Ne säilyttivät keskinäisen järjestyksensä, joten ne ovat keskenään aikajärjestyksessä. Seuraavaksi käyttäjä klikkaa viestit lähettäjän mukaiseen kasvavaan järjestykseen. Späde Spämmärin viestit ovat nyt peräkkäin. Ne ovat keskenään aiheen mukaisessa järjestyksessä, koska niiden keskinäinen järjestys säilyi. Lopuksi käyttäjä klikkaa aihetta uudelleen. Nyt ”Syömään?”-viestit ovat vastakkaisessa järjestyksessä kuin edellisellä kerralla, koska nyt ne ovat keskenään lähettäjän mukaisessa järjestyksessä.

23. Erään työpaikan tietohallintovastaavan käyttämän sähköpostiohjelman järjestämistoiminnot ovat vakaat. Hän sai lukuisia sähköpostiviestejä otsikolla ”Ääkköset eivät toimi”. Analysoidakseen ongelmaa hän halusi lukea viestit lähettäjä kerrallaan, ja kunkin lähettäjän viestit hän halusi lukea aikajärjestyksessä vanhin ensin. Koska aiemmissa keskusteluissa järkevimmät kommentit olivat tulleet Aada-Aaro Bittiseltä ja typerimmät Örwell Öyhöttäjältä, hän halusi edetä lähettäjien nimien mukaisessa aakkosjärjestyksessä. Miten hän klikkaili sähköpostien järjestämiskomentoja?

INSERTIONSORT on vakaa. Vakaus jätettiin edellä käsittelemättä, koska se oli-

viestit aikajärjestyksessä

9:15	Späde Spämmäri	Tsekkaa tämä!
11:46	Yvonne Ystävä	Syömään?
12:03	Nippe Nälkäinen	Syömään?
13:30	Späde Spämmäri	Juoru...
17:41	Kalle Kaveri	Ongelma ratkesi
22:28	Späde Spämmäri	Kissavideo :-)

klikattu aihe

13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
17:41	Kalle Kaveri	Ongelma ratkesi
11:46	Yvonne Ystävä	Syömään?
12:03	Nippe Nälkäinen	Syömään?
9:15	Späde Spämmäri	Tsekkaa tämä!

klikattu lähettäjä

17:41	Kalle Kaveri	Ongelma ratkesi
12:03	Nippe Nälkäinen	Syömään?
13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
9:15	Späde Spämmäri	Tsekkaa tämä!
11:46	Yvonne Ystävä	Syömään?

klikattu aihe

13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
17:41	Kalle Kaveri	Ongelma ratkesi
12:03	Nippe Nälkäinen	Syömään?
11:46	Yvonne Ystävä	Syömään?
9:15	Späde Spämmäri	Tsekkaa tämä!

Kuva 9: Vakaa järjestäminen säilyttää yhtäsuurten alkioiden edellisen järjestyksen

si monimutkaistanut todistusta ja koska se on helppo tarkastaa jälkeenpäin erikseen. Nimittäin ainoa tapa, jolla kahden alkion järjestys voi vaihtua, on rivin 4 suorittaminen silloin kun toinen niistä on $A[j-1]$ ja toinen on *apu*. Mutta rivillä 3 olevan ehdon vuoksi riviä 4 ei suoriteta, jos $A[j-1].x$ ja $apu.x$ ovat yhtäsuuret.

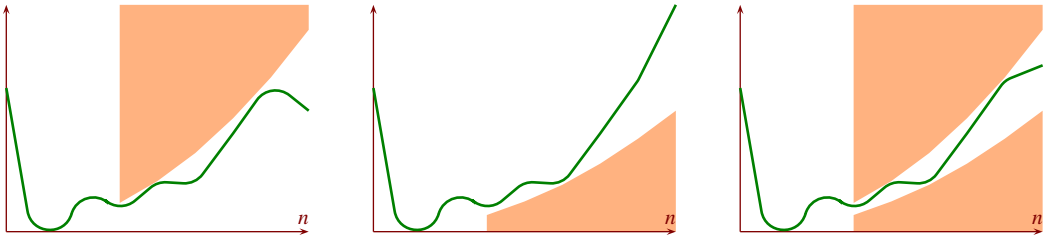
24. Millä pienellä muutoksella INSERTIONSORT:ista saataisiin sellainen, että yhtäsuurten alkioiden keskinäinen järjestys kääntyy takaperoiseksi?

2.8 O-, Ω- ja Θ-merkinnät

Ohjelman suoritusaika ei riipu vain ohjelmasta ja syötteestä, vaan myös tietokoneesta jossa ohjelma suoritetaan, kääntäjästä jolla ohjelma käännettiin, siitä mitä muuta tietokone tekee samalla ja niin edelleen. Toisaalta monen ohjelman suoritusaika kasvaa syötteen koon kasvaessa niin rajusti, että tietokoneen ja kääntäjän vaikutus on sen rinnalla toisarvoista, ja olennaisinta on suoritusajan kasvun muoto.

Tätä kuvataan O-, Ω- ja Θ-merkinnöillä. Niissä otetaan huomioon ainoastaan suoritusaikaan suurilla syötteillä eniten vaikuttava osuus ohjelman toiminnasta, ja sekin ilman vakiokertoimia. Syötteen kokoa kuvataan usein muuttujalla n, mutta algoritmista riippuen käytössä voi olla muukin muuttuja tai jopa useita muuttujia samanaikaisesti. Esimerkiksi kun etsitään reittiä maantiekartasta, niin n voi olla risteysten ja m niiden välisten tienpätkien määrä. O-merkintä asettaa ylärajan sille, kuinka jyrkästi suoritusaika saa suurilla syötteillä kasvaa n:n kasvaessa. Ω-merkintä asettaa alarajan, ja Θ-merkintä asettaa putken, joka rajoittaa sekä ylhäältä että alhaalta. Tätä on havainnollistettu kuvassa 10.

Nämä merkinnät antavat suoritusajoista vain abstraktia tietoa: ne eivät kerro aikayksikköä (kuten sekunti tai vuosi) eivätkä kuinka suuri syötteen täytyy olla, että se otettaisiin huomioon. Silti ne kertovat käytännön suoritusajoista usein niin hyvin, että muuta tietoa ei tarvitakaan. Ne eivät kerro mitä tapahtuu pienillä syöt-



Kuva 10: $O(n^2)$, $\Omega(n^2)$ ja $\Theta(n^2)$

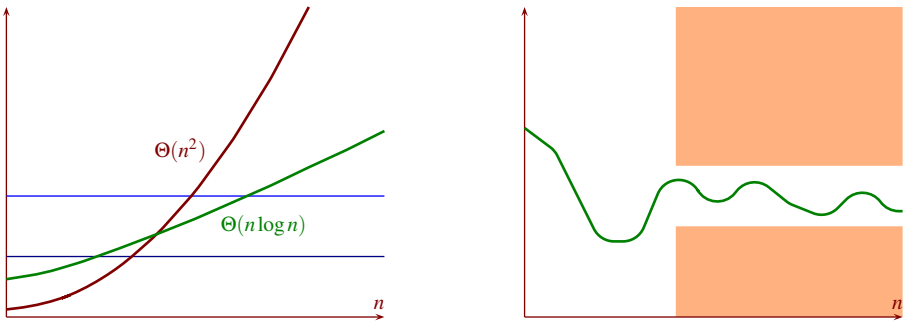
teillä, mutta tyypillisesti sillä ei ole väliä, koska pienillä syötteillä melkein mikä tahansa yhtään järkevä vaihtoehto on tarpeeksi nopea. Tyypillisesti ne ennustavat hyvin, mitä tulee tapahtumaan kun syötekoko kasvaa. Ne eivät ennusta mistä syötekoosta alkaen hitaus tulee olemaan ongelma, mutta ne ennustavat varsin hyvin, tuleeko se käytännössä olemaan ongelma.

Sitäpaitsi ne kertovat suoritusajoista etukäteen, ennen ohjelman toteuttamista. Niinpä toisin kuin mittaukset, ne ovat käytettävissä jo kun mietitään, mikä toteuttamisvaihtoehto kannattaa valita.

Varsin tavallista on, että tarjolla on yksinkertainen pienillä syötteillä erittäin nopea algoritmi sekä monimutkainen pienillä syötteillä selvästi hitaampi algoritmi, jonka Θ-merkinnällä ilmaistu suoritusaika on parempi. Kuvan 11 vasen puoli havainnollistaa tätä. Siinä yksinkertaisen algoritmin suoritusaika on $\Theta(n^2)$ ja monimutkaisen $\Theta(n \log n)$.[3] Kuten kuvasta näkyy, jostakin syötekoosta alkaen monimutkainen algoritmi on nopeampi, ja syötekoon kasvaessa nopeusero kasvaa niin rajusti, että monimutkaisen algoritmin suoritusaika pysyy siedettävän pienenä kauan sen jälkeen kun yksinkertainen algoritmi on muuttunut toivottoman hitaaksi. Sekunneissa ja minuuteissa esitetty esimerkki tästä tulee kuvassa 41.

Toisin kuin ehkä voisi luulla, tietokoneiden nopeutuminen ei ole vähentänyt

[3] Simo Juvaste on kehittänyt selkeän selityksen sille, mitä log tarkoittaa algoritmeissa. Kerromme sen sivulla 43. Siihen saakka riittää, että log on jotain nopeaa.



Kuva 11: $\Theta(n^2)$ verrattuna $\Theta(n \log n)$, sekä $\Theta(1)$

vaan lisännyt nopeiden algoritmien merkitystä loppukäyttäjien näkökulmasta. Ihmiselle ei ole väliä, tuleeko vastaus 0,1 vai 0,2 sekunnissa, mutta hänelle on väliä, tuleeko se 10 vai 20 sekunnissa. Kuvan 11 tummansininen viiva edustaa niiden askelten määrää, jotka ohjelma ehtii suorittaa ajassa, joka ei tunnu hänestä liian pitkältä. Vaalensininen viiva edustaa samaa sen jälkeen kun tietokone on vaihdettu nopeampaan. Ensimmäisessä tapauksessa $\Theta(n^2)$-algoritmi ehtii käsitellä siinä ajassa isomman syötteen kuin $\Theta(n \log n)$-algoritmi, mutta jälkimmäisessä tapauksessa päinvastoin.

Kuvasta 11 ei käy ilmi, että monessa tapauksessa suoritusaika ei riipu pelkästään syötteen koosta, vaan myös syötteen laadusta. Tästä tulee esimerkki jo luvussa 2.9, ja luvussa 5.1 tulee dramaattinen esimerkki. Jos tositilanteen syöte on erilaista kuin Θ-merkintää laskettaessa oletettiin, niin Θ-merkinnän antama kuva voi olla olennaisesti väärä. Sama toimii myös niinpäin, että jos mittauksissa käytetty syöte on erilaista kuin tositilanteen syöte, niin mittauksetkin voivat antaa väärän kuvan. Tavallisesti Θ-merkintöjä käytettäessä kiinnitetään eniten huomiota suoritusaikaan huonoimmassa mahdollisessa tapauksessa. Jos niin saatu suoritusaika on riittävän hyvä, niin suoritusaika on riittävän hyvä myös tositilanteessa riippumatta siitä, millaista syöte todellisuudessa on.

Vaikka edellä puhuttiin ajan kulutuksesta, käytetään O-, Ω- ja Θ-merkintöjä usein ilmaisemaan myös muistin kulutusta. Jos ohjelma on kunnolla toteutettu, niin näiden merkintöjen tarkkuudella ilmaistuna sen suoritusaika ja muistin kulutus ovat samat kuin sen toteuttaman algoritmin suoritusaika ja muistin kulutus. Niinpä on järkevää analysoida algoritmien ajan ja muistin kulutuksia O-, Ω- ja Θ-merkinnöillä. On kuitenkin muistettava, että sellaisiakin tapauksia on, joissa Θ-merkinnän mukaan paras ei ole käytännössä paras tai edes kelvollinen. Ääriesimerkki tästä ovat niin sanotut ”galaktiset algoritmit” (”galactic algorithms”).

Kuvan 11 oikealla puolella on havainnollistettu, mitä tarkoittaa $\Theta(1)$. Se tarkoittaa, että on olemassa kaksi positiivista lukua, joiden välissä kulutus on riippumatta syötteen koosta, paitsi että pienillä syötteillä sallitaan poikkeuksia. Kulutus saa vaihdella syötteen koon ja laadun mukaan, mutta se ei saa kasvaa rajatta eikä lähestyä nollaa syötteen koon kasvaessa, eikä edes heilahdella niin että suuret arvot kasvavat rajatta tai pienet arvot lähestyvät nollaa. Toisinaan tämä ilmaistaan sanomalla, että kulutus on vakio. Se on jossain määrin harhaanjohtavaa, sillä $\Theta(1)$ ei lupaa että kulutus ei vaihtele, vaan se lupaa vain että vaihtelu pysyy isoilla syötteillä syötteen koosta ja laadusta riippumattomissa rajoissa.

Se, että $\Theta(1)$ sallii poikkeuksia pienillä syötteillä, on kätevää puhuttaessa esimerkiksi siitä, kuinka kauan kestää poistaa pinosta ylin alkio. Pienin mahdollinen pino on tyhjä eli sen koko on nolla. Siitä ei saa yrittää poistaa ylintä alkiota, koska siinä ei ole ylintä alkiota. Niinpä kun pinon koko on nolla, ei ole järkevää vaatia, että poistamisen ajan kulutus olisi niiden kahden vakion välissä, joiden välissä $\Theta(1)$ lupaa sen olevan suurilla syötteillä. Toinen etu on, että voidaan käyttää

muun muassa merkintää $\Theta(n \log n)$, vaikka $\log 0$ ei ole määritelty.

Korjattakoon tässä yhteydessä myös se väärinkäsitys, että O-, Ω- ja Θ-merkinnät olisivat suunnilleen sama asia kuin matematiikan raja-arvo. Silloin kun matematiikan raja-arvo on olemassa, niin siitä saadaan O-, Ω- tai Θ-merkintä. Mutta raja-arvo kieltää vaihtelun perusteellisemmin kuin O-, Ω- ja Θ-merkinnät. Siksi ohjelmoinnissa tilanne on hyvin usein sellainen, että raja-arvoa ei ole olemassa, mutta O-, Ω- tai Θ-merkintä on.

2.9 Lisäysjärjestämisen suoritusaika

INSERTIONSORT:issa on kaksi silmukkaa, lukujen ja taulukon alkioiden sijoituksia, apumuuttujan ja taulukon alkioiden avainten suuruusjärjestysvertailuja, ykkösen lisäämistä ja vähentämistä sekä `&&`. Silmukoita lukuun ottamatta kukin näistä toiminnoista vie $\Theta(1)$ aikaa. Kuten edellä todettiin, se tarkoittaa, että on olemassa kaksi positiivista lukua, joiden välissä ajan kulutus on riippumatta taulukon koosta, kunhan koko on jotain rajaa isompi.

Vaikka taulukoiden alkioiden sijoitus ja vertaaminen vievätkin sitä enemmän aikaa mitä isompia alkiot ja niiden avaimet ovat, niidenkin ajan kulutuksen katsotaan olevan $\Theta(1)$, koska alkioiden ja avainten koko ei muutu INSERTIONSORT:in suorituksen aikana eikä riipu n:stä eli alkioiden määrästä.

Kuvasta 41 näkyy, että alkioiden koolla on olennainen merkitys käytännön suoritusajalle. Jos tämä halutaan ottaa Θ-merkinnässä huomioon, niin voidaan ottaa käyttöön toinen muuttuja k kuvaamaan alkion kokoa. Jos tälle tielle lähdetään, niin herää kysymys, tarvitsisiko myös avaimen koko oman muuttujan. Sehän voi olla olennaisesti suurempi kuin luvun koko mutta samalla olennaisesti pienempi kuin alkion koko, esimerkiksi kun avaimena on henkilön nimi ja alkiossa on paljon henkilön muita tietoja. Järjestämisen tapauksessa lisämuuttujien analyysille tuomasta lisätarkkuudesta olisi hyötyä vain jos ohjelman nopeudelle asetettavat vaatimukset ovat poikkeuksellisen tiukat. Siksi emme ota niitä käyttöön ennen lukua 7.4 (ja sielläkin vain toisen, ja eri syystä).

Koska tarkoitus on selvittää vain mitä tapahtuu suurilla syötteillä, voimme unohtaa tapauksen $n = 0$. Kun $n > 0$, kiertää **for**-silmukka $n - 1$ kierrosta, joten INSERTIONSORT kuluttaa ainakin lukuun n verrannollisesti aikaa. Tämä voidaan ilmaista sanomalla, että INSERTIONSORT:in suoritusaika on $\Omega(n)$.

Tämä ilmaus on samankaltainen kuin jos rautatieasemalla kuulutetaan että veturissa on ongelma, joten kestää ainakin 20 minuuttia ennen kuin juna lähtee. Se ei lupaa että juna lähtee 20 minuutin kuluttua, vaan se lupaa vain että juna ei lähde sitä aikaisemmin. Saattaa olla, että junan lähtöä joudutaan odottamaan 30 minuuttia, mutta koska kuuluttaja ei vielä tiedä tuleeko niin käymään, hän ei sano 30 minuuttia vaan 20 minuuttia. Jos on totta, että junan lähtöä joutuu odottamaan ainakin 20 minuuttia, niin myös on totta, että sitä joutuu odottamaan ainakin 5 mi-

nuuttia. Matkustajien kannalta tärkeää ei kuitenkaan ole pelkästään että kuulutukset ovat totta, vaan myös kuinka paljon niistä saa informaatiota. Kuuluttaja puhuu 20 minuutista eikä 5 minuutista siksi, että se kertoo matkustajille enemmän.

Samoin, koska INSERTIONSORT:in suoritusaika on $\Omega(n)$, on sen suoritusaika myös $\Omega(\sqrt{n})$ ja myös $\Omega((\log n)^3)$. Näistä eri vaihtoehdoista suosimme ilmausta $\Omega(n)$, koska se kertoo eniten. Jos myöhemmin osoittautuu että aikaa kuluu vaikka $\Omega(n\sqrt{n})$, siirrymme käyttämään ilmausta $\Omega(n\sqrt{n})$, koska se kertoo enemmän kuin $\Omega(n)$. Vielä emme voi käyttää sitä, koska vielä emme tiedä onko se totta.

Vastaavalla tavalla INSERTIONSORT:in suoritusajalle voidaan antaa yläraja, jota suoritusaika ei milloinkaan ylitä, mutta joka ei kiellä INSERTIONSORT:ia olemasta nopeampi toisinaan tai jopa aina. Rivi 1 suoritetaan n kertaa, rivit 2 ja 5 $n-1$ kertaa, ja rivien 3 ja 4 suorituskertojen määrä vaihtelee sen mukaan missä järjestyksessä A oli alun perin. Koska j ei ole koskaan suurempi kuin $n-1$ eikä pienempi kuin 0 ja vähenee jokaisella rivin 4 suorituksella, riviä 3 ei suoriteta ulomman silmukan millään kierroksella enemmän kuin n kertaa. Koska ulomman silmukan kierroksia on $n-1$, ei rivin 3 suoritusten määrä voi olla yhteensä enemmän kuin $(n-1)n = n^2 - n \leq n^2$. Rivi 4 suoritetaan enintään yhtä monesti kuin rivi 3. Niinpä mitään riviä ei suoriteta enemmän kuin n^2 kertaa. Siksi INSERTIONSORT:in suoritusaika on enintään muotoa n^2. Toisin sanoen, INSERTIONSORT:in suoritusaika on $O(n^2)$.

Ohjelmat, joiden suoritusaika on muotoa n^2, ovat hitaita suurilla syötteillä. Mutta $O(n^2)$ on vain ylälikiarvo. Kunnes toisin todistetaan, on mahdollista että INSERTIONSORT:in todellinen suoritusaika on parempi. Osoittautuu, että INSERTIONSORT:in todellisen suoritusajan muoto vaihtelee välillä $n, \ldots, n^2$ riippuen siitä, missä järjestyksessä A on alun perin. Jos A on alun perin takaperoisessa järjestyksessä, niin ensimmäinen alkio siirretään 0 askelta, toinen 1 askeleen, ... ja viimeinen $n-1$ askelta. Siirtoja tulee kaikkiaan $0+1+\ldots+(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$ askelta. Koska n kasvaa hitaammin kuin n^2, saa termin $-\frac{1}{2}n$ jättää pois. Koska myös vakiokertoimen $\frac{1}{2}$ saa jättää pois, jää jäljelle vain n^2. Niinpä $O(n^2)$ on paras mikä voidaan ilmaista O-muodossa. (Luvussa 3.5 kerrotaan, miten tällaisesta pitkästä yhteenlaskusta saa helposti Ω-merkinnän.)

Sen, että $O(n^2)$ on paras mikä voidaan ilmaista O-muodossa, voi sanoa toisellakin tavalla: INSERTIONSORT:in suoritusaika on ***hitaimmillaan*** $\Theta(n^2)$. Symbolin Θ käyttäminen symbolin O tilalla kertoo, että suoritusaika on muotoa n^2 eikä enempää tai vähempää. Toisaalta sana ”hitaimmillaan” kertoo, että näin huonoa suoritusaikaa ei luvata kaikille syötteille, vaan ainoastaan osalle syötteistä, mutta kuitenkin äärettömän monelle eri syötteelle. Ilmausten ”suoritusaika on $O(n^2)$” ja ”suoritusaika on hitaimmillaan $\Theta(n^2)$” ero on siinä, että edellinen jättää avoimeksi toteutuuko näin huono suoritusaika millään syötteellä, mutta jälkimmäinen kertoo, että kyllä toteutuu, eikä vain yhdellä tai viidellä eri syötteellä, vaan ääret-

tömän monella.

25. Onko INSERTIONSORT:in suoritusaika $O(n^3)$? Onko se hitaimmillaan $\Theta(n^3)$?

26. Onko INSERTIONSORT:in suoritusaika $O(\frac{n^2}{100})$? Onko se hitaimmillaan $\Theta(\frac{n^2}{100})$?

27. Mikä on INSERTIONSORT:in suoritusaika Θ-merkinnällä ilmaistuna?

    Vastaavasti tieto, että suoritusaika on $\Omega(n)$, kertoo että INSERTIONSORT ei koskaan ole muotoa n nopeampi, mutta jättää avoimeksi, onko INSERTIONSORT koskaan edes niin nopea. Kyllä se on, nimittäin ainakin silloin kun A on jo alun perin kasvavassa järjestyksessä. Silloin $A[j-1].x > apu.x$ ei toteudu koskaan, joten rivi 3 suoritetaan kaikkiaan vain $n-1$ kertaa eikä riviä 4 suoriteta kertaakaan. Niinpä INSERTIONSORT:in suoritusaika on nopeimmillaan $\Theta(n)$.

28. Oletetaan, että taulukossa on ensin $\frac{n}{3}$ ykköstä ja sitten $\frac{2n}{3}$ nollaa. Taulukko on siis esimerkiksi $[1,1,0,0,0,0]$. Mikä on INSERTION-SORT:n suoritusaika? Perustele vastauksesi!

29. Oletetaan, että taulukossa on ensin $\sqrt{n}$ ykköstä ja loput alkiot ovat nollia. Mikä on INSERTION-SORT:n suoritusaika? Perustele vastauksesi!

    Kiinnostavaa on myös, mikä on suoritusaika muissa kuin hitaimmassa ja nopeimmassa tapauksessa. Mikä on suoritusaika tyypillisessä tapauksessa? Valitettavasti on epäselvää, minkälainen tapaus on tyypillinen. Kuinka monella eri alkiolla voi tyypillisessä tapauksessa olla sama avain?

30. Mikä on INSERTION-SORT:n suoritusaika Θ-merkinnällä ilmaistuna, jos kaikki alkiot ovat yhtäsuuret? Miksi?

    Sen verran voidaan sanoa, että jos millään kahdella eri alkiolla ei voi olla sama avain ja jos alkiot ovat alun perin satunnaisessa järjestyksessä, niin suoritusaika on $\Theta(n^2)$. Nimittäin silloin kukin alkio siirtyy keskimäärin puolet A:n järjestetyn osan koosta, joten ajan kulutus on keskimäärin suunnilleen puolet siitä, mitä se on silloin kun A on alun perin takaperoisessa järjestyksessä. Tästä syystä sanotaan, että lisäysjärjestämisen keskimääräinen suoritusaika on $\Theta(n^2)$.

31. Kuvassa 12 oleva ohjelma poistaa kunkin alkion vuorollaan `A`:n loppuosasta, etsii puolitushaulla sille suuruusjärjestyksen mukaisen paikan `A`:n alkuosasta, ja lisää sen sinne. (Ohjelmassa on sekaisin indeksejä ja iteraattoreita, koska kirjastotoiminnot käyttävät iteraattoreita mutta `insert` mitätöi niiden voimassaolon, joten silmukkaa ei saa toteuttaa niillä.) Puolitushaku toimii ajassa $O(\log n)$, joten se on hyvin nopea. Kuinka nopea kuvan 12 ohjelma on nopeimmillaan? Kuinka nopea se on hitaimmillaan? Vertaa näitä nopeuksia INSERTIONSORT:in nopeuksiin.

2.10 Muistin tarve

INSERTIONSORT tarvitsee järjestettävän taulukon lisäksi hyvin vähän muistia, nimittäin muuttujille *apu*, *i*, *j* ja *n*. Niinpä sen lisämuistin tarve on $\Theta(1)$ ja muuten-

```
1  class vertaaja{
2  public:
3    bool operator()( const alkio & eka, const alkio & toka ){
4      return eka.x < toka.x;
5    }
6  };

7  void EraseInsert( taulukko & A ){
8    vertaaja vrt;
9    for( unsigned i = 0; i < A.size(); ++i ){
10     alkio apu = A[i]; A.erase( A.begin()+i );
11     taulukko::iterator j =
12       std::upper_bound( A.begin(), A.begin()+i, apu, vrt );
13     A.insert( j, apu );
14   }
15 }
```

Kuva 12: Järjestäminen poistolla, puolitushaulla ja lisäämisellä

kin erittäin pieni. Ilmaus ”$\Theta(1)$” tarkoittaa, että jotakin tarvitaan vähintään jonkin positiivisen vakion verran ja enintään jonkin positiivisen vakion verran riippumatta syötteen koosta ja laadusta. Sitä on selostettu sivulla 27.

2.11 Lisää kysymyksiä

Kuvassa 13 oleva järjestämisalgoritmi tunnetaan nimellä ***valintajärjestäminen*** (***selection sort***).

32. Miksi on tärkeää, että rivillä 1 lukee `i+1 < A.size()` eikä `i < A.size()-1`?
33. Olkoon $n =$ `A.size()`. Mitä i:n arvosta tiedetään rivin 2 alussa? Mitä i:n, j:n ja p:n arvoista tiedetään rivin 4 alussa? Mitä i:n ja p:n arvoista tiedetään rivin 6 alussa?
34. Anna valintajärjestämisen ulommalle silmukalle invariantti!

```
   void SelectionSort( taulukko & A ){
1    for( unsigned i = 0; i+1 < A.size(); ++i ){
2      unsigned p = i;
3      for( unsigned j = i+1; j < A.size(); ++j ){
4        if( A[j].x < A[p].x ){ p = j; }
5      }
6      alkio apu = A[i]; A[i] = A[p]; A[p] = apu;
7    }
8  }
```

Kuva 13: Valintajärjestäminen

35. Anna valintajärjestämisen sisemmälle silmukalle invariantti! (Ei tarvitse toistaa mitään siitä, mitä ulomman silmukan invariantti sanoo.)

36. Onko valintajärjestäminen vakaa? Anna esimerkki syötteestä, jolla se vaihtaa kahden samansuuruisen alkion keskinäisen järjestyksen, tai perustele, että sellaista syötettä ei voi olla olemassa!

37. Vaihdetaan rivillä 4 operaattorin < tilalle <=. Anna esimerkki syötteestä, jolla muutettu ohjelma vaihtaa kahden samansuuruisen alkion keskinäisen järjestyksen!

38. Kuinka paljon valintajärjestäminen käyttää aikaa nopeimmillaan ja hitaimmillaan Θ-merkinnällä ilmaistuna? Perustele lyhyesti.

* 39. Tämä kysymys on tarkoitettu erityisesti numeerisesta laskennasta kiinnostuneille. Kertolasku on monimutkaisempaa kuin yhteenlasku, joten kertolaskujen määrää pyritään toisinaan vähentämään. Polynomin $a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0$ arvon annetulla x:n arvolla saa laskettua pienellä määrällä kertolaskuja seuraavasti:

$$tulos := a_n; \textbf{ for } i := n-1 \textbf{ downto } 0 \textbf{ do } tulos := x \cdot tulos + a_i$$

Anna **for**-silmukalle invariantti.

40. Joskus kertakaikkisen typerä vitsi saa silkalla typeryydellään jonkin asian pysymään hyvin muistissa. Tämä kysymys ja sen vastaus yrittävät olla sellainen vitsi. O-, Θ- ja Ω-merkintöjen kannalta tärkeää on vain ja ainostaan se, mitä tapahtuu eräällä Suomen tunturilla, jossa on hiihtokeskus. Mikä on tämä tunturi?

3 Kekojärjestäminen ja prioriteettijono

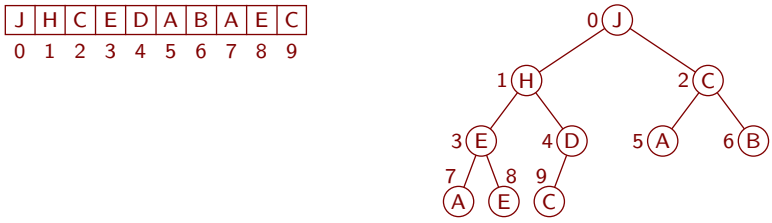
Keko (***heap***) tarkoittaa ohjelmoinnissa kahta eri asiaa. Toisaalta se tarkoittaa sitä aluetta muistissa, josta ohjelma voi esimerkiksi luoda olioita toiminnolla `new`. Toisaalta se tarkoittaa tietyllä tavalla järjestettyä taulukkoa tai puurakennetta, johon voi logaritmisessa ajassa lisätä alkion ja ottaa jonkin suurimman alkion pois (tai vaihtoehtoisesti jonkin pienimmän, mutta ei molempia). Tässä luvussa keskitytään taulukkomuotoiseen kekoon ja sen sovelluksiin. Tärkeimmät sovellukset ovat taulukon järjestäminen suuruusjärjestykseen nopeasti vähällä lisämuistilla sekä prioriteettijono, jolla voi valita seuraavaksi suoritettavia tehtäviä niiden tärkeyden mukaan. Luvussa 4 näytetään, miten prioriteettijonon avulla voi toteuttaa tehokkaan reitin etsinnän.

3.1 Keot

Kuvassa 14 on vasemmalla puolella esimerkki binäärikeosta jossa on jokin suurin ensimmäisenä, ja oikealla puolella sitä on havainnollistettu binääripuuna. Tapana on piirtää binääripuut haarautumaan ylhäältä alaspäin. Piirroksen ympyrät ovat ***solmuja*** (***vertex***) ja viivanpätkät ***kaaria*** (***edge***). Kaaren alapäässä oleva solmu on yläpäässä olevan solmun ***lapsi*** (***child***), ja yläpäässä oleva solmu on alapäässä olevan solmun ***vanhempi*** (***parent***). Ylintä solmua kutsutaan ***juureksi*** (***root***). Solmun ***syvyys*** (***depth***) on juuresta solmuun vievän polun pituus. Juuren syvyys on 0. Keskenään samalla syvyydellä olevat solmut muodostavat ***tason*** (***level***).

Kekoa havainnollistavan binääripuun alin taso solmuja pitää täyttää vasemmalta alkaen, ja jokaisen muun tason on oltava täysi. Taulukko on keko, jossa on jokin suurin ensimmäisenä, jos ja vain jos sen havainnollistuksessa kunkin solmun paitsi juuren avain on enintään yhtäsuuri kuin siihen kaarella yhdistetyn lähinnä ylemmän solmun avain. Toisin sanoen, vanhemman on aina oltava vähintään yhtäsuuri kuin lapsensa, missä suuruuksia verrataan avainten mukaan. Kuvassa solmun 5 avain A on pienempi kuin solmun 2 avain C, solmun 8 avain on yhtäsuuri kuin solmun 3 avain ja niin edelleen.

Keko voidaan toteuttaa myös jokin pienin ensimmäisenä. Silloin solmun avaimen pitää olla vähintään yhtäsuuri kuin vanhempansa avain.



Kuva 14: Esimerkki keosta ja sen havainnollistus binääripuuna

Keko on binäärikeko, jos ja vain jos kullakin solmulla on tasan kaksi lasta kunnes solmut loppuvat kesken. ***Vasen lapsi*** (***left child***) on lapsista vasemmanpuoleinen tai ainoa, ja toinen lapsi on ***oikea lapsi*** (***right child***). Koska binäärikekoja käytetään paljon enemmän kuin muita, jätetään etuliite ”binääri” tyypillisesti pois. Tästä eteenpäin ”keko” tarkoittaa binäärikekoa jossa on jokin suurin ensimmäisenä, ellei erikseen täsmennetä toisin.

41. Vähintään ja enintään kuinka monella solmulla on täsmälleen yksi lapsi? Mitä solmujen määrästä voidaan päätellä, jos jollakin solmulla on täsmälleen yksi lapsi?

42. Kuinka monella solmulla ei ole yhtään lasta? Perustele vastauksesi!

Koska keko toteutetaan taulukkona eikä ”solmu” ole taulukoihin liittyvä käsite, siirrymme puhumaan lokeroista solmujen sijaan. Lokeron 0 lapset ovat lokeroissa 1 ja 2, lokeron 1 lapset ovat lokeroissa 3 ja 4, lokeron 2 lapset ovat lokeroissa 5 ja 6 ja niin edelleen niin pitkälle kuin lapsia riittää. Siis lokeron i ne lapset jotka ovat olemassa, ovat lokeroissa $2i+1$ ja $2i+2$. Vasen lapsi on olemassa, jos ja vain jos $2i+1 \leq n-1$, ja oikea lapsi jos ja vain jos $2i+2 \leq n-1$. Myös lokeron itsensä täytyy olla olemassa. Siksi lokerolla i on vasen lapsi, jos ja vain jos $0 \leq i \leq \frac{n-2}{2}$, ja oikea lapsi, jos ja vain jos $0 \leq i \leq \frac{n-3}{2}$.

Vastaavasti lokeron i vanhempi, missä $1 \leq i \leq n-1$, on lokerossa $\frac{i-1}{2}$ jos i on pariton ja lokerossa $\frac{i-2}{2}$ jos i on parillinen. Arvo $i = 0$ jätettiin pois, koska ensimmäisellä lokerolla ei ole vanhempaa. Niinpä taulukko $A[0 \ldots n-1]$ on keko, jos ja vain jos jokaiselle parittomalle i väliltä $1, \ldots, n-1$ pätee $A[\frac{i-1}{2}].x \geq A[i].x$, ja jokaiselle parilliselle i väliltä $1, \ldots, n-1$ pätee $A[\frac{i-2}{2}].x \geq A[i].x$.

Puhuminen erikseen parittoman ja erikseen parillisen kokoisista taulukoista on kömpelöä. Sen voi välttää ottamalla käyttöön merkinnän $\lfloor x \rfloor$. Se tarkoittaa suurinta kokonaislukua, joka on enintään yhtäsuuri kuin x. Niinpä $\lfloor \frac{n}{2} \rfloor$ tarkoittaa $\frac{n}{2}$ pyöristettynä alaspäin. Jos n on parillinen, niin $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$, ja jos n on pariton, niin $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. Monissa ohjelmointikielissä, jos `n` ja `m` ovat kokonaislukutyyppiä, $\texttt{n} \geq 0$ ja $\texttt{m} > 0$, niin `n/m` pyöristää alas lähimpään kokonaislukuun. Niissä $\lfloor \frac{\texttt{n}}{\texttt{m}} \rfloor$ saadaan lausekkeella `n/m`. (Jos $\texttt{n} < 0$ tai $\texttt{m} < 0$, niin se mitä `n/m` laskee vaihtelee eri ohjelmointikielissä, katso Wikipedia ”Modulo” $\rightsquigarrow$ ”In programming languages”.)

Nyt voidaan ilmaista näppärämmin, mitä tarkoittaa, että $A[0\ldots n-1]$ on keko, jossa on jokin suurin ensimmäisenä:

Jokaisella i väliltä $1,\ldots,n-1$ pätee $A[\lfloor\frac{i-1}{2}\rfloor].x \geq A[i].x$.

Vastaavasti $A[0\ldots n-1]$ on keko, jossa on jokin pienin ensimmäisenä, jos ja vain jos jokaisella i väliltä $1,\ldots,n-1$ pätee $A[\lfloor\frac{i-1}{2}\rfloor].x \leq A[i].x$.

Monesti keoista puhuttaessa taulukot indeksoidaan ykkösestä eikä nollasta alkaen. Silloin vanhempi ja lapset pitää laskea hieman toisin. Ilmaus "jos $1 \leq i \leq n-1$, niin lokeron i vanhempi on lokerossa $\lfloor\frac{i-1}{2}\rfloor$" saadaan siirrettyä taulukolta $A[0\ldots n-1]$ taulukolle $A[1\ldots n]$ kolmessa vaiheessa. Ensin jokaista lokeron numeroa kasvatetaan yhdellä, jolloin saadaan "jos $1 \leq i \leq n-1$, niin lokeron $i+1$ vanhempi on lokerossa $\lfloor\frac{i-1}{2}\rfloor+1$". Sitten jokaisen i:n tilalle kirjoitetaan $(i-1)$, jolloin saadaan "jos $1 \leq (i-1) \leq n-1$, niin lokeron $(i-1)+1$ vanhempi on lokerossa $\lfloor\frac{(i-1)-1}{2}\rfloor+1$". Sieventämällä saadaan "jos $2 \leq i \leq n$, niin lokeron i vanhempi on lokerossa $\lfloor\frac{i}{2}\rfloor$". Koska päättelyvirheitä voi tapahtua, on hyödyllistä testata vastausta kokeilemalla muutamalla i:n arvolla. Lauseke $\lfloor\frac{i}{2}\rfloor$ antaa lokeroiden 2, 3, 4, 5 ja 6 vanhempien lokeroiksi 1, 1, 2, 2 ja 3, mikä on oikein.

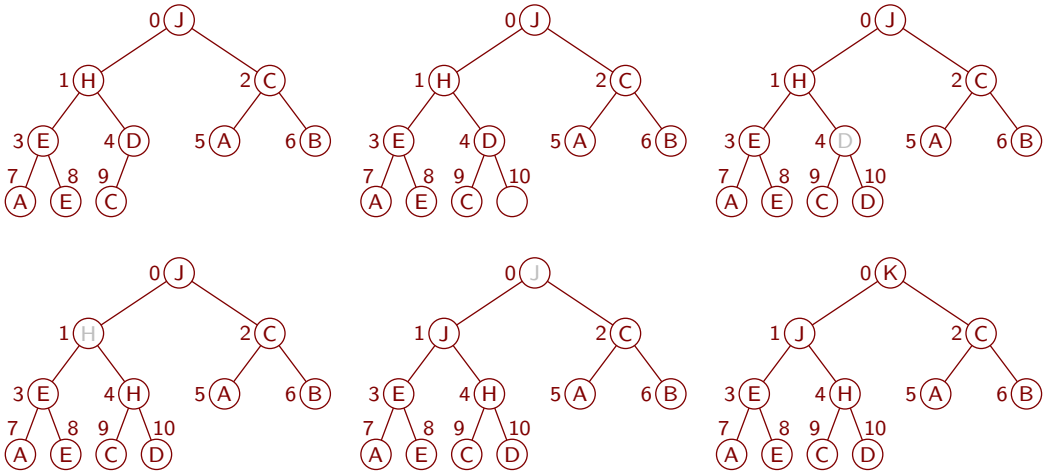
43. Tee sama muunnos ilmaukselle "jos $0 \leq i \leq \frac{n-2}{2}$ niin lokeron i vasen lapsi on lokerossa $2i+1$, ja jos $0 \leq i \leq \frac{n-3}{2}$ niin oikea lapsi on lokerossa $2i+2$"!

44. Binäärikekoa voi kutsua myös kaksihaaraiseksi keoksi. Tarkastelemme jälleen taulukkoa $A[0\ldots n-1]$. Kolmihaaraisessa keossa lokeron 0 lapset ovat lokeroissa 1, 2 ja 3, lokeron 1 lapset ovat lokeroissa 4, 5 ja 6 ja niin edelleen. Mitkä ovat lokeron i vanhemman, ensimmäisen lapsen ja kolmannen lapsen lokerot kolmihaaraisessa keossa, ja millä i:n arvoilla ne ovat olemassa?

45. Tarkastelemme yhä taulukkoa $A[0\ldots n-1]$. Olkoon d ykköstä suurempi kokonaisluku. Mitkä ovat lokeron i vanhemman, ensimmäisen lapsen ja d:nnen lapsen lokerot d-haaraisessa keossa, ja millä i:n arvoilla ne ovat olemassa?

3.2 Kekoon lisääminen ja jonkin suurimman poistaminen

Kuvassa 15 on havainnollistettu alkion lisäämistä kekoon. Ensin keon loppuun lisätään lokero. Tässä tapauksessa se tuli paikkaan 10. Sitten uuden lokeron vanhempi kopioidaan uuteen lokeroon, vanhemman vanhempi kopioidaan edellisessä kopioinnissa vapautuneeseen lokeroon ja niin edelleen, kunnes uusi alkio on enintään yhtäsuuri kuin vapaan lokeron vanhempi tai vapaalla lokerolla ei ole vanhempaa. Tässä tapauksessa uusi alkio K on suurempi kuin uuden lokeron vanhempi eli lokeron 4 sisältö D, joten D kopioidaan lisättyyn lokeroon. Sitten H kopioidaan lokerosta 1 lokeroon 4, J kopioidaan lokerosta 0 lokeroon 1, ja uusi alkio K laitetaan lokeroon 0.



Kuva 15: Esimerkki K:n lisäämisestä kekoon

Kuvassa 16 on lisäys pseudokoodina. Aloitamme sen pohtimisen tarkastamalla, että jokainen A:n indeksointi on laillinen. Sitä varten osoitamme, että $0 \leq i < n$ riveillä $3, \ldots, 5$, missä n on taulukon A koko rivin 2 jälkeen. Rivin 1 lopussa $0 \leq i$, ja rivin 2 lopussa lisäksi $i = n-1 < n$. Jos $0 < i < n$, niin $0 \leq \lfloor \frac{i-1}{2} \rfloor \leq \frac{i-1}{2} < \frac{i}{2} < i < n$. Koska rivin 3 loppuosaan ja riville 4 mennään vain jos $i > 0$, ja koska rivillä 3 ja rivin 4 alussa $j = \lfloor \frac{i-1}{2} \rfloor$, ovat rivien 3 ja 4 indeksoinnit laillisia ja rivin 4 sijoituksen $i := j$ jälkeenkin $0 \leq i < n$. Myös A:n indeksointi rivillä 5 on laillinen, koska sielläkin $0 \leq i < n$.

Olettaen, että A on keko kun LISÄÄKEKOON aloittaa, voidaan seuraavalla silmukkainvariantilla LISÄÄKEKOON osoittaa oikein toimivaksi:

1. Alkio j on alkion i vanhempi, eli $j = \lfloor \frac{i-1}{2} \rfloor$.
2. Osissa $A[0 \ldots i-1]$ ja $A[i+1 \ldots n-1]$ on yhteensä alkuperäiset alkiot.
3. Jokaisella k väliltä $1, \ldots, n-2$ on alkion k vanhempi vähintään yhtäsuuri kuin alkio k, eli $A[\lfloor \frac{k-1}{2} \rfloor].x \geq A[k].x$.
4. $i = n-1$ tai osan 3 väite pätee myös kun $1 \leq k = n-1$.

```
   LISÄÄKEKOON(&A, uusi)
1  i := A.koko; j := ⌊(i−1)/2⌋
2  A.kooksi(i+1)
3  while i > 0 && A[j].x < uusi.x do
4     A[i] := A[j]; i := j; j := ⌊(i−1)/2⌋
5  A[i] := uusi
```

Kuva 16: Alkion lisääminen kekoon

5. $i = n-1$ tai $A[i].x < uusi.x$.

Kun riville 3 tullaan ensimmäisen kerran, pätee $i = n-1$, koska n tarkoittaa A:n kokoa rivin 2 jälkeen. Niinpä invariantin osat 4 ja 5 pätevät. Myös osa 2 pätee, koska kun $i = n-1$, on $A[0 \ldots i-1]$ alkuperäinen taulukko ja $A[i+1 \ldots n-1]$ on tyhjä. Osa 3 sanoo, että $A[0 \ldots n-2]$ on keko. Keon ominaisuuden ilmaisemisessa käytettiin sivulla 35 muuttujaa i, mutta nyt käytetään muuttujaa k, koska i on varattu toiseen tarkoitukseen: i on ohjelman muuttuja. Osa 3 pätee, koska alkuperäinen taulukko on keko. Osa 1 pätee rivin 1 ansiosta. Siksi IE pätee.

Osoitamme seuraavaksi, että IH pätee. Rivin 5 lopussa on A:ssa ne alkiot mitkä pitääkin olla, koska invariantin osan 2 mukaan muut kuin uusi alkio ovat A:ssa, ja uusi alkio sijoitettiin rivillä 5 juuri siihen lokeroon, jonka osa 2 jättää käyttämättä. Keko-ominaisuus muille kuin viimeiselle alkiolle luvataan osassa 3. Jos $i \neq n-1$, niin se luvataan viimeiselle alkiolle osassa 4. Muussa tapauksessa $i = n-1$. Jos rivien 3 ja 4 silmukka pysähtyi siksi, että $A[j].x \geq uusi.x$, niin rivin 5 lopussa alkion i vanhempi on vähintään yhtäsuuri kuin $A[i]$, koska $A[i] = uusi$ ja alkion i vanhempi on lokerossa j. Muussa tapauksessa rivien 3 ja 4 silmukka pysähtyi siksi, että $i \leq 0$. Tällöin $n = i+1 \leq 1$. Rivin 2 vuoksi $n \geq 1$, joten $n = 1$. Siksi keko-ominaisuus pätee automaattisesti.

Vielä tarvitsee osoittaa IS. Invariantin osa 1 saatetaan voimaan rivin 4 lopussa. Osa 2 säilyy voimassa sijoituksessa $A[i] := A[j]$, koska se ei väitä $A[i]$:stä mitään. Se säilyy sijoituksessa $i := j$, koska i:n muuttumisen vuoksi osan 2 väitteen piiriin tuleva alkio on juuri kopioitu osan 2 piiristä poistuvasta alkiosta. Koska osa 2 ei mainitse j:tä, ei sijoitus $j := \lfloor \frac{i-1}{2} \rfloor$ vaikuta siihen. Osan 5 osuus $A[i].x < uusi.x$ ja samalla koko osa pätevät rivin 4 lopussa, koska silloin i:llä on sama arvo kuin j:llä oli rivillä 3, jossa testattiin $A[j].x < uusi.x$.

Invariantin osaan 3 vaikuttaa vain $A[i] := A[j]$. Osat 3 ja 4 ovat heti sen jälkeen voimassa seuraavista syistä. Jos $k = i$, niin $A[k]$ eli $A[i]$ on yhtäsuuri kuin vanhempansa eli $A[j]$. Jos i on k:n vanhemman lokero, niin $i < k \leq n-1$ ja ennen sijoitusta $A[i] := A[j]$ päti $A[k].x \leq A[i].x \leq A[j].x$ osien 3 ja 4 vuoksi. Siksi sijoituksen jälkeen $A[k].x \leq A[i].x$. Muilla k:n arvoilla osat 3 ja 4 säilyivät voimassa, koska alkio k ja sen vanhempi eivät muuttuneet. Sijoitus $i := j$ ei vaikuta osaan 4, koska jos $i = n-1$ niin juuri sitä ennen $A[n-1]$:een kopioitiin vanhempansa, joten osa 4 pätee i:n arvosta riippumattomalla tavalla.

While-silmukka lopettaa, koska i:n arvo pienenee jokaisella kierroksella, koska $\lfloor \frac{i-1}{2} \rfloor < i$ kun $i > 0$, kuten sivulla 36 todettiin.

Rivi 2 voi epäonnistua jos ohjelman suoritukselle varattu muisti ei riitä A:n kasvattamiseen tai $i+1$ ei mahdu käytettyyn tietotyyppiin. Tyypillistä on, että muistin loppumiseen ei kannata varautua mitenkään, koska niin tapahtuu harvoin eikä sille

voida tehdä mitään. Jos muisti loppuu, niin ohjelma kaatuu. Toki jos kyse on esimerkiksi lentokonetta lentävästä ohjelmasta, niin tarvitaan parempi ratkaisu kuin että annetaan ohjelman vain kaatua. Niiden toteutus on kuitenkin tapauskohtaista, eikä kuulu tämän kirjoituksen piiriin.

Jos $i = 0$ rivillä 1 tai 4, niin $\lfloor \frac{i-1}{2} \rfloor = -1$. Jos käytössä on etumerkitön kokonaislukutyyppi, niin -1 ei sisälly siihen. Mutta ainakaan C:ssä ja C++:ssa ohjelma ei tähän kaadu, vaan j saa arvokseen suurimman tyyppiin mahtuvan luvun. Se on algoritmin LISÄÄKEKOON näkökulmasta väärä arvo. Se ei kuitenkaan haittaa, koska jos $i = 0$, niin rivin 3 loppuosaan ja riville 4 ei mennä, joten j:n väärää arvoa ei käytetä mihinkään.

Olettaen että kaikki muuttujat ovat sopivaa tyyppiä, ei kuvassa 16 ole edellä käsiteltyjen lisäksi muuta, joka voisi aiheuttaa laittoman toiminnon.

Ylin alkio voidaan poistaa keosta nostamalla sen jompikumpi lapsi ylimmäksi, nostamalla niin vapautuneeseen lokeroon jompikumpi sen lapsista ja niin edelleen, kunnes vapaa lokero on keon viimeiselle alkiolle sopivassa paikassa. Sitten keon viimeinen alkio laitetaan vapaaseen lokeroon. Algoritmi tehostuu hiukan ja sen miettiminen helpottuu, jos oikean lokeron etsinnän aikana keon tulkitaan loppuvan juuri ennen viimeistä alkiota.

46. Jos vapaalla lokerolla on kaksi lasta, niin kumpi niistä pitää nostaa siihen? Miksi?

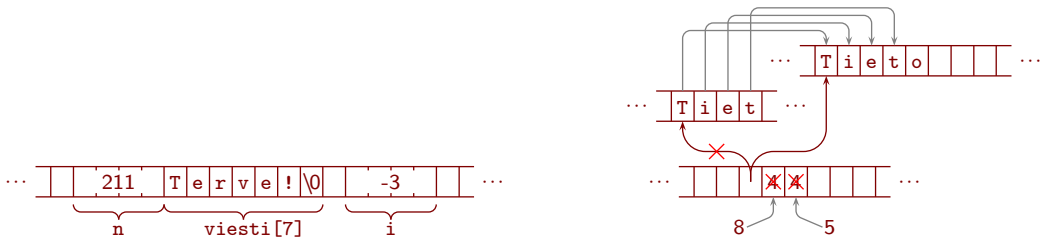
47. Kirjoita keosta poistaminen pseudo- tai ohjelmakoodina!

3.3 Kasvavat taulukot ja tasattu ajan kulutus

Tietokoneen muisti voidaan ajatella suurena taulukkona, josta varataan muuttujille sopivankokoisia pätkiä. Kuvassa 17 vasemmalla on varattu neljä tavua kokonaisluvulle `n`, seitsemän tavua C-tyyliselle merkkijonolle `viesti` ja neljä tavua kokonaisluvulle `i`. Nykyaikaisissa tietokoneissa pyritään sijoittamaan muuttujia neljällä jaollisiin osoitteisiin, joten `viesti`:n perään on jätetty käyttämätön tavu ennen seuraavaa muuttujaa. Muuttujien kohdalla muistissa ei ole tietoa, missä muuttuja alkaa ja loppuu, vaan muuttujaa käyttävä ohjelman osa on kirjoitettu siten että se käsittelee jostakin kohdasta alkavaa jonkin pituista muistialuetta. C-tyylisissä merkkijonoissa on aina lopussa nollatavu helpottamassa niiden käsittelyä, mutta se on muistin hallinnasta erillinen käytäntö.

Taulukon laittaminen muiden muuttujien sekaan kuten kuvassa vasemmalla johtaa siihen, että taulukkoa ei voi kasvattaa ohjelman suorituksen aikana. Sellaisen taulukon maksimikoko on siis tiedettävä etukäteen. Se tekee ohjelmoinnista kömpelöä.

Siksi on kehitetty toisenlainen, hitaampi mutta joustavampi tapa tallentaa taulukko muistiin. Sitä on havainnollistettu kuvassa 17 oikealla. Siinä taulukon si-



Kuva 17: Taulukoiden esityksiä muistissa

sällölle varataan tilaa erillisestä muistialueesta. (Sitäkin kutsutaan keoksi, vaikka sillä ei ole mitään tekemistä keko-nimisen tietorakenteen kanssa.) Taulukon kohdalla muiden muuttujien seassa on vain tieto mistä varsinaiselle sisällölle varattu muistialue alkaa, kuinka monelle alkiolle siinä on tilaa, ja kuinka monta alkiota taulukossa on. Alkioiden määrän tilalla voi olla osoitin sinne missä alkiot loppuvat, ja kapasiteetin tilalla voi olla osoitin sinne missä sisällölle varattu muistialue loppuu. Alkioiden käsittelyyn tulee lisävaiheena osoittimen osoittamaan muistipaikkaan siirtyminen.

Jos taulukko mahtuu kasvamaan sen sisällölle varatussa tilassa, niin kasvatus vie $\Theta(1)$ aikaa. Muussa tapauksessa kasvatus alkaa varaamalla taulukon sisällölle isompi muistialue ja kopioimalla sisältö sinne. Tätäkin on havainnollistettu kuvassa 17 oikealla. Silloin aikaa kuluu $\Theta(n)$.

Voidaan laskea esimerkiksi, että jos aloitetaan tyhjällä taulukolla, muistia varataan sadan alkion verran kerrallaan, ja lisättävien alkioiden määrä n on sadan monikerta, niin alkioita siirretään yhteensä $\frac{1}{200}n^2 - \frac{1}{2}n$ kertaa. Se on $\Theta(n^2)$. Jos n on iso niin sillä on suuri merkitys, vaikka vakiokerroin $\frac{1}{200}$ on melko pieni. Toisaalta jos ohjelmassa on paljon muutaman alkion kokoisia taulukoita ja jokaiselle varataan tilaa sadalle alkiolle, niin muistia tuhlaantuu merkittävästi.

Siksi tapana on, että uusi muistialue on kooltaan kaksinkertainen edelliseen verrattuna (paitsi jos edellinen koko oli 0). Siitä seuraa, että muistialueen kasvatuksia tapahtuu harvoin. Alla olevan taulukon keskimmäisellä rivillä on kullakin kasvatuskerralla tehtävien sijoitusten määrä (siirtäminen uuteen muistialueeseen plus viimeksi lisätyn alkion sijoittaminen taulukkoon), ja alimmalla rivillä on kaiken kaikkiaan tehtyjen sijoitusten määrä. Alla oletetaan, että pienin epätyhjälle taulukolle varattu tila on yhden alkion verran.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17
1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48

Taulukkoa katsomalla näyttää siltä, että sijoitusten kokonaismäärä vaihtelee välillä $2n-1$ (kun n on kahden potenssi), …, $3n-3$ (kun n on kahden potenssi

plus yksi). Osoitamme seuraavaksi, että kun $n > 1$, on sijoitusten määrä enintään $3n-3$.

Jos $n > 1$, niin muistialueesta toiseen siirretään ainakin kerran enintään $n-1$ alkiota. Muistialueen toisella, kolmannella ja niin edelleen kasvatuskerralla siirrettävistä alkioista täsmälleen puolet oli siirretty edelliselläkin kasvatuskerralla. Niinpä enintään $\frac{1}{2}(n-1)$ alkiota siirretään ainakin kahdesti, enintään $\frac{1}{2} \cdot \frac{1}{2}(n-1)$ ainakin kolmesti ja niin edelleen. Siirtojen kokonaismäärä on siis enintään $(1 + \frac{1}{2} + (\frac{1}{2})^2 + \ldots + (\frac{1}{2})^{k-1})(n-1)$, missä k on kasvatuskertojen määrä. Se on alle $2(n-1)$, sillä on helppo tarkastaa, että kun k on positiivinen kokonaisluku, pätee $1 + \frac{1}{2} + \ldots + (\frac{1}{2})^{k-1} = 2 - (\frac{1}{2})^{k-1}$. Se on siis enintään $2n-3$. Kun siihen lisätään alkuperäiset sijoitukset, saadaan että sijoituksia on kaikkiaan enintään $3n-3$.

Aikaa kuluu hieman myös kasvatustarpeen testaamiseen, uuden koon laskemiseen, muistin varaamiseen muistinhallintajärjestelmältä ja niin edelleen. Isoilla taulukoilla kuitenkin sijoitusten viemä aika hallitsee. Siitä seuraa, että jos aloitetaan tyhjällä taulukolla ja lisätään n alkiota, niin aikaa kuluu enintään noin kolminkertaisesti verrattuna siihen, kuinka paljon aikaa kuluisi, jos taulukolle olisi alun perin varattu tilaa n alkiolle.

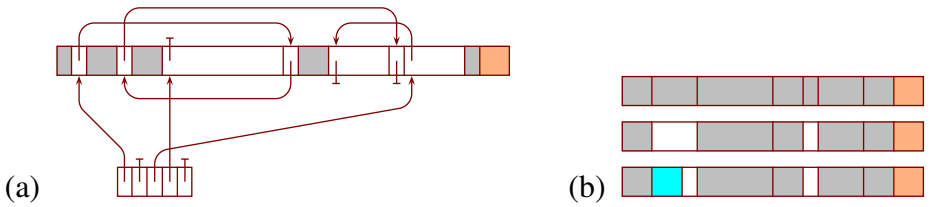
Siksi jos aloitetaan tyhjällä taulukolla ja lisätään n alkiota, niin aikaa ei kulu kaikkiaan $\Theta(n^2)$, vaan vain $O(n)$. Tästä on helppo jatkaa tulokseen, että aikaa kuluu kaikkiaan $\Theta(n)$.

48. Todista, että aikaa kuluu kaikkiaan $\Theta(n)$!

Niinpä vaikka ei voida luvata, että kukin alkion lisääminen vie enintään $\Theta(1)$ aikaa, voidaan silti luvata, että n:n alkion lisääminen vie yhteensä enintään $\Theta(n)$ aikaa. (Kuten sivulla 27 kerrotaan yksityiskohtaisemmin, $\Theta(1)$ tarkoittaa että suurilla syötteillä vähintään vakion verran ja enintään vakion verran.) Jos kokonaisaika jaettaisiin tasan lisäämisten kesken, niin kunkin lisäämisen osuus olisi $\Theta(1)$. Siksi sanotaan, että alkion lisääminen kasvavaan taulukkoon vie ***tasatusti*** (***amortized***) $\Theta(1)$ aikaa.

Tasatun suoritusajan käyttäytyminen on tärkeä ymmärtää esimerkiksi auton jarruja ohjaavaa ohjelmaa laadittaessa. Jos syntyy kolari siksi että jarruilla kesti kolme sekuntia reagoida, niin ei paljoa lohduta, että niin pitkä reagointiaika toteutuu vain harvakseltaan ja melkein aina muulloin jarrut reagoivat sekunnin murto-osassa. Jos huomataan, että jarrut ovat melkein aina nopeat mutta joskus harvoin hitaat, niin vikaa on vaikea etsiä, koska sitä on vaikea saada toistumaan. Jos vika ilmenee hyvin harvoin, niin voi olla vaikea jopa saada auton valmistaja uskomaan, että mitään vikaa on.

Jos taulukolle varattua tilaa halutaan vapauttaa taulukon pienentyessä, niin rajana olevan alkioiden määrän kannattaa olla olennaisesti pienempi kuin kasvatuksessa. Jos rajat olisivat samat, niin aikaa kuluisi hyvin paljon, jos vuorotellen lisättäisiin ja poistettaisiin alkio niin että joka lisäyksellä taulukon sisältö kopioitaisiin suurempaan ja joka poistolla pienempään muistialueeseen. Tilan vapauttaminen



Kuva 18: (a) vapaiden listat 2:n potensseille; (b) vapaan muistin pirstoutuminen

taulukon pienentyessä ei kuitenkaan ole yhtä tarpeellinen toiminto kuin tilan varaaminen taulukon kasvaessa, sillä nykyisissä tietokoneissa on niin paljon muistia, että sitä on yleensä varaa tuhlata. C++:n `vector`:in pienentyessä ei tilaa vapauteta automaattisesti. Ohjelmoija voi halutessaan toiminnolla `shrink_to_fit` käskeä erikseen, että tilaa pitää yrittää vapauttaa.

3.4 Hieman muistinhallinnasta

Ohjelmointikielen suoritusympäristö voi hyödyntää taulukon kasvattamisessa vapautuneen muistialueen esimerkiksi tavalla, jota matkitaan seuraavassa tehtävässä ja havainnollistetaan kuvassa 18 (a). Käyttöjärjestelmältä saatuja ja sille palautettavia muistialueita on tapana kutsua ***muistilohkoiksi*** (***memory block***). Oikeassa tilanteessa käytettäisiin tavuja, mutta tehtävässä käytetään alkoita, joiden tyyppi on `unsigned`. Kierrätettävissä oleva muisti esitetään taulukkona `muisti[` $0 \ldots$ `kaikki` $-\,1$ `]`. Sen alkuosassa on varattuja tai varattuina olleita muistilohkoja, ja loppuosassa kohdasta `loppu` alkaen on käyttämätöntä muistia. Jos muistilohkon koko on 2^E alkiota, niin tässä tehtävässä lukua E sanotaan muistilohkon eksponentiksi. Luku 2^E saadaan lausekkeella `(1u <<` (E)`)`.

Vapautettu muisti kierrätetään taulukon `vapaat` avulla. Alkiossa `vapaat[`E`]` on joko vapaan 2^E:n kokoisen muistilohkon alkukohdan numero tai `~0u` merkiksi siitä, että sen kokoisia ei ole vapaana. Merkintä `0u` tarkoittaa `unsigned`-tyypin nollaa, ja `~` kääntää sen jokaisen bitin päinvastaiseksi. Niinpä `~0u` on suurin `unsigned`-tyyppiin mahtuva luku. Vapaan muistilohkon ensimmäisessä alkiossa on jonkin muun vapaan samankokoisen muistilohkon alkukohdan numero tai `~0u`. Kaikki vapaat samankokoiset muistilohkot muodostavat siis linkitetyn listan. Kun tarvitaan jonkin kokoinen muistilohko, yritetään ensin saada sellainen taulukosta `vapaat`. Jos se ei onnistu, niin otetaan halutun kokoinen lohko käyttämättömästä muistista, jos siinä on riittävästi tilaa jäljellä.

49. Kirjoita aliohjelma `tuplaa`, joka saa parametreikseen muistilohkon alkukohdan numeron ja eksponentin, varaa kaksinkertaisen muistilohkon, kopioi sisällön sinne, ja palauttaa sen alkukohdan numeron! Jos kaksinkertaista muistilohkoa ei voi varata, niin `tuplaa` palauttaa `~0u`. Suurin sallittu eksponentti on `maks_eksp`.

Voit olettaa, että `unsigned`-tyypin lukualue ei lopu kesken.

Tässä tekstissä noudatetaan laajalti suositeltua kansainvälistä standardia, jonka mukaan ”kilo”, ”mega”, ”giga” ja niiden lyhenteet ”k”, ”M”, ”G” ja niin edelleen viittaavat kymmenen potensseihin, ja suunnilleen saman suuruisiin kahden potensseihin viitataan sanoilla ”kibi”, ”mebi”, ”gibi” ja lyhenteillä ”Ki”, ”Mi”, ”Gi” ja niin edelleen. Kuten Wikipedian sivulla ”Binary prefix” kerrotaan, sanojen ”kilo” ja niin edelleen ja niiden lyhenteiden käyttö tarkoittamaan kahden potensseja on aiheuttanut sekaannusta ja jopa oikeusjuttuja.

50. Jos kierrätettävissä olevaa muistia on yksi tebitavu eli 2^{40} tavua, niin kuinka monta alkiota taulukossa `vapaat` tarvitsee olla?

Edellisen tehtävän vastaus havainnollistaa sitä, että jos muistia varataan ja vapautetaan vain kahden potenssien kokoisina lohkoina, niin muistin kierrättämiseen riittää hyvin pieni määrä vapaiden muistilohkojen listoja. Jos muistia varattaisiin ja vapautettaisiin minkä tahansa kokoisina lohkoina, niin listojen määrä saattaisi kasvaa ongelmallisen suureksi. Samalla todennäköisyys sille olisi pieni, että vapautuneiden lohkojen joukossa olisi täsmälleen sopivan kokoinen. Silloin jouduttaisiin joko antamaan liian iso lohko, halkaisemaan liian iso lohko sopivan kokoiseksi ja loppuosaksi kuten kuva 18 (b) näyttää, tai ottamaan uusi lohko käyttämättömästä muistista. Ensimmäinen tuhlaa muistia. Jälkimmäiset aiheuttavat vapaan muistin ***pirstoutumista*** (***fragmentation***) eli jakautumista lukuisiksi lohkoiksi.

Kahden potenssien käyttö saattaa johtaa siihen, että melkein puolet annetusta muistista jää käyttämättä. Vastineeksi se vähentää pirstoutumista. On parempi, että korkeintaan puolet muistista tuhlataan, kuin että yritetään käyttää muisti tarkemmin mutta vähän kerrassaan muisti pirstotaan niin pieniin osiin, että lopulta paljon yli puolet on tuhlattu. Niin kauan kun muistia riittää, on muistin tuhlaaminen tyypillisesti pienempi paha kuin muistin nuukempaan käyttöön kuluva ylimääräinen aika, koska ei ole suurta eroa siinä, jääkö muisti käyttämättä siksi että ohjelma varaa mutta ei käytä sitä, vai siksi että ohjelma ei varaa sitä. Niinpä tavallisesti olennaista ei ole yrittää käyttää muistia mahdollisimman vähän, vaan voida tehdä käytettävissä olevalla muistilla mahdollisimman paljon.

Valitettavasti kahden potenssien käyttö ei sellaisenaan ratkaise kaikkia muistinhallinnan ongelmia. Eräs jäljelle jääneistä on, että jos varataan ja vapautetaan paljon pieniä muistilohkoja, niin kahden potenssien tapauksessakin muisti pirstoutuu erillisiksi lohkoiksi, joista ei pystytä antamaan yhtenäistä isoa lohkoa, vaikka vapaata muistia olisi paljon. Siksi on kehitetty menetelmiä tunnistaa vierekkäisiä vapaita lohkoja ja yhdistää niitä isommiksi vapaiksi lohkoiksi, sekä lisätä todennäköisyyttä sille, että vapaan lohkon vieressä on toinen, yhdistämiskelpoinen vapaa lohko. Niistä löytyy tietoa esimerkiksi hakusanoilla ”buddy memory allocation”. Kahden potenssien käyttö helpottaa tällaisten menetelmien suunnittelua.

Muistin hallinta on siis vaikeaa. Sen tekniikoita on hiottu vuosikymmeniä, joten nykyaikaiset käyttöjärjestelmät selviävät siitä kohtalaisen hyvin. Silti tarpee-

ton muistin varaaminen ja vapauttaminen saattavat nykyisinkin hidastaa ohjelmaa tuntuvasti. Tästä tulee esimerkki kuvassa 47. Jos ohjelma tarvitsee vain muutamankokoisia muistilohkoja, voi tehokkaan kierrätyksen toteuttaa helposti itse vapaiden lohkojen listoilla. Jos tarvitaan vain yhdenkokoisia muistilohkoja, niin ei ole sitä vaaraa, että lohkoa ei voi antaa vaikka vapaata muistia olisi riittävästi, koska se on erikokoisissa lohkoissa kuin mitä tarvittaisiin.

3.5 Kekoon lisäämisen ja poistamisen ajan kulutus

Luvussa 3.3 kerrotusta seuraa, että kuvan 16 rivillä 2 kuluu toisinaan aikaa $\Theta(n)$, mutta tyypillisesti vain $\Theta(1)$. Kukin muu yksittäinen toiminto kuvassa 16 kuluttaa $\Theta(1)$ aikaa. Laskemme seuraavaksi melko tarkan ylälikiarvon sille, miten monta kertaa rivien 3 ja 4 silmukka voi kiertää.

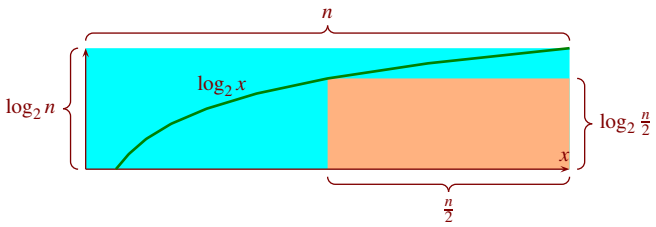
Koska sivun 36 mukaan $\lfloor \frac{i-1}{2} \rfloor < \frac{i}{2}$ kun $i > 0$, ja koska rivin 2 lopussa $i = n-1$, on silmukan kierrosten määrä enintään niiden kertojen määrä, jotka luku n täytyy puolittaa, jotta tulos olisi alle 1. Jos n on kahden potenssi, niin niiden kertojen määrä, jotka n täytyy puolittaa jotta tulos olisi tasan 1, on luvun n kaksikantainen logaritmi eli $\log_2 n$.[4] Jos $n > 0$ mutta n ei ole kahden potenssi, niin $\log_2 n$ ei ole kokonaisluku. Silloin $\lfloor \log_2 n \rfloor + 1$ puolitusta tuottaa tulokseksi alle 1. Niinpä rivien 3 ja 4 silmukan kierrosten määrä on aina enintään $1 + \log_2 n$ (paitsi kun $n = 0$).

Siksi niillä kerroilla, joilla A:lle varattua muistilohkoa ei kasvateta rivillä 2, on aliohjelman LISÄÄKEKOON suoritusaika $O(\log_2 n)$. Alaindeksi 2 on tapana jättää pois O-, Θ- ja Ω-merkinnöissä, koska vakiokertoimilla ei ole niissä väliä, ja erikantaiset logaritmit eroavat toisistaan vain vakiokertoimella (esimerkiksi $\log_8 n = \frac{1}{3} \log_2 n$). Niinpä niillä kerroilla, joilla A:lle varattua muistilohkoa ei kasvateta rivillä 2, on aliohjelman LISÄÄKEKOON suoritusaika $O(\log n)$. Niillä harvoilla kerroilla, joilla muistilohkoa kasvatetaan, on suoritusaika $\Theta(n)$.

Jos aloitetaan tyhjällä keolla ja tehdään useita kekotoimintoja, niin keon kasvatuksen ajan kulutuksena saa käyttää tasattua ajan kulutusta eli $\Theta(1)$. Silloin kunkin yksittäisen kekoon lisäämisen ajan kulutuksena saa käyttää $O(\log n)$.

Toisaalta jos aloitetaan tyhjällä keolla ja lisätään siihen n alkiota, niin ajan kulutusta laskiessa täytyy ottaa huomioon, että ensimmäinen alkio lisätään kekoon jonka koko on 0, toinen alkio lisätään kekoon jonka koko on 1 ja niin edelleen. Siksi ajan kulutus on $O(\log_2 1 + \log_2 2 + \ldots + \log_2 n)$. On kuitenkin selvää, että lisääminen isompaan kekoon vie hitaimmillaan ainakin yhtä paljon aikaa kuin lisääminen pienempään kekoon vie hitaimmillaan. Siksi aikaa kuluu yhteensä $O(\log_2 n + \log_2 n + \ldots + \log_2 n) = O(n \log n)$. Saman tuloksen saa siitä, että $\log_2$ on kasvava funktio, eli $\log_2 a \leq \log_2 b$ aina kun $0 < a < b$. Nimittäin $\log_2 1 + \ldots + \log_2 n \leq n \log_2 n$ kun $n \geq 1$, joten $\log_2 1 + \ldots + \log_2 n = O(n \log n)$.

[4] Kiitos Simo Juvasteelle ehdotuksesta selittää log tällä tavalla.



Kuva 19: Ylä- ja alalikiarvo summalle $\log_2 1 + \log_2 2 + \ldots + \log_2 n$

Tätä on havainnollistettu vaaleansinisellä kuvassa 19.

51. Keossa on jokin suurin ensimmäisenä. Alun perin tyhjään kekoon lisätään n alkiota. Anna syöte, jolla jokainen alkio vuorollaan kiipeää keon ylimmäksi!

Jos alun perin tyhjään kekoon lisätään n alkiota joista jokainen kiipeää keon ylimmäksi, niin ajan kulutus on $\Theta(\log_2 1 + \log_2 2 + \ldots + \log_2 n)$. Summan $\log_2 1 + \ldots + \log_2 n$ tarkka arvo on vaikea laskea. Sille on kuitenkin helppo löytää Θ-merkintöjen kannalta riittävän tarkka likiarvo. Näimme jo, että se on $O(n \log n)$. Käyttämällä keskimmäistä yhteenlaskettavaa alalikiarvona sen jälkeen tuleville yhteenlaskettaville, voidaan osoittaa, että summa on $\Omega(n \log n)$. Jos n on pariton, niin keskimmäinen yhteenlaskettava on $\log_2 \frac{n+1}{2} > \log_2 \frac{n}{2}$ ja yhteenlaskettavia on $\frac{n+1}{2} > \frac{n}{2}$ kappaletta. Jos n on parillinen, niin käytämme keskimmäisenä yhteenlaskettavana $\log_2 \frac{n}{2}$, jolloin yhteenlaskettavia on $\frac{n}{2} + 1 > \frac{n}{2}$ kappaletta. Tätä on havainnollistettu alla.

$$\log_2 1 + \log_2 2 + \underbrace{\log_2 3 + \log_2 4 + \log_2 5} \qquad \log_2 1 + \underbrace{\log_2 2 + \log_2 3 + \log_2 4}$$

Summan loppuosa on vähintään $\frac{n}{2} \log_2 \frac{n}{2} = \frac{1}{2} n \log_2 n - \frac{\log_2 2}{2} n = \Omega(n \log n)$. Summan alkuosa on vähintään 0, koska $\log_2 a \geq 0$ kun $a \geq 1$. Siksi $\log_2 1 + \ldots + \log_2 n = \Omega(n \log n)$ ja $\log_2 1 + \ldots + \log_2 n = \Theta(n \log n)$.

Lukua $\frac{n}{2} \log_2 \frac{n}{2}$ on havainnollistettu vaaleanruskealla kuvassa 19. Kuva ei ole aivan tarkka vastine summalle, koska kuvan x-akseli käyttää reaalilukuja, mutta summan termien indeksit ovat kokonaislukuja. Pienet erot eivät kuitenkaan tavallisesti vaikuta Θ-, O- ja Ω-merkintöihin, koska niissä kiinnitetään huomiota vain isoihin argumenttien arvoihin ja abstrahoidaan vakiokertoimet pois.

Summan $\log_2 1 + \log_2 2 + \ldots + \log_2 n$ tarkemmista likiarvoista kiinnostuneille suositellaan Wikipedian sivua ”Stirling’s approximation”.

52. Miksi ei välttämättä ole totta, että jos aloitetaan tyhjällä keolla ja kasvatetaan sitä n kertaa, niin aikaa kuluu yhteensä $\Theta(n \log n)$?

Myös vastauksessa 47 esitetyn keosta poistamisen silmukkaa kierretään enintään $1 + \log_2 n$ kertaa, kun $n > 0$. C++:n `vector`:ia pienennettäessä ei koskaan siirretä alkioita toiseen muistilohkoon, joten se vie $\Theta(1)$ aikaa (siis $\Theta(1)$ eikä vain

```
HEAPSORT(&A)
1  TEEKEKO(A)
2  for h := A.koko − 1 downto 1 do
3     apu := A[0]
4     MUUTETTUPOISKEOSTA(A,h)
5     A[h] := apu
```

```
TEEKEKO(&A)
1  n := A.koko
2  for k := ⌊n/2⌋ − 1 downto 0 do
3     apu := A[k]; i := k; j := 2i+1
4     while true do
5        if j+1 < n && A[j+1].x ≥ A[j].x then
6           j := j+1
7        if j ≥ n || A[j].x ≤ apu.x then break
8        A[i] := A[j]; i := j; j := 2i+1
9     A[i] := apu
```

Kuva 20: kekojärjestäminen (heapsort)

tasatusti $\Theta(1)$). Siksi POISKEOSTA kuluttaa $O(\log n)$ aikaa ilman sanaa ”tasatusti”, ainakin C++:lla toteutettuna.

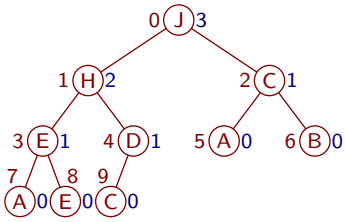
Edellä kuvattu keko ei ole O- ja Θ-merkintöjen tasolla nopein tunnettu. Nimellä ”Fibonaccin keko” tunnettu rakenne on nopeampi, varsinkin jos tarvitaan toimintoa yhdistämään kaksi kekoa. Fibonaccin keot ovat kuitenkin niin monimutkaisia, että niiden ajan kulutuksen vakiokerroin on suuri. Siksi niiden teoreettinen nopeushyöty ei yleensä realisoidu käytännössä ellei syöte ole valtava.

3.6 Kekojärjestäminen

On helppo keksiä, miten taulukko A saadaan kasvavaan suuruusjärjestykseen ajassa $O(n \log n)$ aliohjelmien LISÄÄKEKOON ja POISKEOSTA avulla: ensin kukin alkio lisätään kekoon, ja sitten toistuvasti keon ylin alkio kopioidaan tulostaulukkoon sen lopusta alkaen ja poistetaan keosta. Tätä ratkaisua voi tehostaa kahdella tavalla.

Ensiksi, keon ei tarvitse olla olemassa erikseen, vaan taulukkoa A voidaan käyttää niin, että sen alkuosa on kekona ja loppuosassa alkiot ovat lopullisilla paikoillaan. Siten lisämuistin tarve saadaan pieneksi. POISKEOSTA muutetaan siten, että h:ta ei lasketa A:n koosta vaan se tulee parametrina, ja $A.kooksi(h)$ jätetään pois. Toiseksi, alkioiden lisääminen yksi kerrallaan kekoon korvataan nopeammalla ratkaisulla, jonka esittelemme pian.

Kuvan 20 vasemmalla puolella näytetty HEAPSORT on suoraan edellä annetun kuvauksen mukainen. Siinä h on viimeisen kekoon kuuluvan alkion paikka. Jos käytetään etumerkitöntä kokonaislukutyyppiä ja A on tyhjä, niin $A.koko - 1$ rivillä 2 ei tuota -1 vaan suurimman tyyppiin mahtuvan luvun. Siinä tapauksessa rivien 2, ..., 5 silmukka käsittelee suuren määrän olemattomia A:n alkioita, joka on virhe. C++:n tapauksessa tämä voidaan välttää aloittamalla silmukka näin: `for( unsigned h = A.size(); h-- > 1; )`. Millä tahansa kielellä HEAPSORT:in alkuun voidaan lisätä **if** $A.koko = 0$ **then return**.



Kuva 21: Solmujen korkeuksia keossa

Kuvan 20 oikealla puolella esitetty TEEKEKO käy A:n alkupuolikkaan takaperin läpi. Se etsii kullekin käsiteltävälle alkiolle paikan samaan tapaan kuin POISKEOSTA, ja laittaa sen sinne. Jotta pitkähköä ilmausta $A.koko$ ei tarvitsisi toistaa, rivillä 1 sen arvo kopioidaan muuttujaan n.

Osataulukko $A[k+1\ldots n-1]$ täyttää aina rivin 2 alussa keon ehdon, eli jos $2k+3 \leq h \leq n-1$ niin $A[\lfloor \frac{h-1}{2} \rfloor].x \geq A[h].x$. Kun riville 2 tullaan ensimmäisen kerran se pätee, koska silloin osataulukossa on vain lapsettomia lokeroita. Se säilyy voimassa **for**-silmukan kierroksen aikana, koska $A[k]$ siirretään keko-ominaisuuden mukaiseen paikkaansa siirtämällä sitä suurempia keon alkioita ylöspäin kuten POISKEOSTA tekee. Kun **for**-silmukka on lopettanut, pätee $k=-1$, joten keko-ominaisuus toteutuu koko taulukolle $A[-1+1\ldots n-1]$.

53. Jos A:n koko on pariton, niin kuuluuko A:n keskimmäinen alkio niihin, jotka TEEKEKO käy takaperin läpi? Miksi $\lfloor \frac{n}{2} \rfloor - 1$ on paras aloituspaikka k:lle?

54. Kirjoita TEEKEKO:n rivi 2 C++:lla `unsigned`-tyypillä siten, että se toimii oikein myös jos `A`:ssa on alle kaksi alkiota!

Vaikka kunkin $A[k]$ käsittely voi viedä selvästi enemmän kuin vakion verran aikaa, toimii TEEKEKO silti ajassa $\Theta(n)$. Se johtuu siitä, että suurimmassa osassa tapauksia alkio voi valua alas vain vähän matkaa. Tämän ymmärtämistä auttaa urheilusta peräisin oleva esimerkki. Cup-turnauksessa kunkin ottelun voittaja pääsee jatkoon ja häviäjän pelit ovat loppu, kunnes jäljellä on enää neljä joukkuetta. Jos varsinaisella otteluajalla syntyy tasapeli, ottelua jatketaan tavalla tai toisella kunnes jompikumpi voittaa. Loppujen neljän joukkueen kesken pelataan neljä ottelua.

Jos joukkueita on n kappaletta, tarvitaan joukkueiden määrän pudottamiseksi neljään $n-4$ ottelua, joten otteluitakin on kaikkiaan n kappaletta. Koska kussakin ottelussa pelaa kaksi joukkuetta, on otteluihin osallistumisia kaikkiaan $2n$ kappaletta. Niinpä joukkue osallistuu keskimäärin $\frac{2n}{n}$ eli kahteen otteluun. Joukkueen keskimäärin pelaamien otteluiden määrä ei siis riipu joukkueiden määrästä ja on paljon pienempi kuin joukkueen enimmillään pelaamien otteluiden määrä.

Aliohjelman TEEKEKO ajan käyttöön vaikuttaa samankaltainen ilmiö. Sen pohtimiseksi otamme käyttöön käsitteen solmun ***korkeus*** (***height***). Se on mahdollisimman pitkän solmusta alas lapsettomaan solmuun vievän polun pituus. Ku-

vassa 21 on esimerkkejä. Lapsettoman solmun korkeus on 0. Korkeus on eri asia kuin syvyys eli matka juuresta solmuun, ja taso eli samalla korkeudella olevien solmujen joukko.

Koska keon alin taso täytetään vasemmalta alkaen ja muut tasot ovat täydet, on keossa kunkin solmun i korkeus vähintään sama kuin solmun $i+1$ korkeus, kun $0 \leq i < n-1$. Siksi, korkeutta 0 lukuun ottamatta, kullakin korkeudella olevista solmuista muilla kuin viimeisellä on kaksi lasta, ja ne ovat yhtä alemmalla korkeudella kuin solmu itse. Viimeisenkin solmun vasen lapsi on olemassa ja yhtä alemmalla korkeudella, mutta oikea lapsi voi puuttua (jos solmu itse on korkeudella 1) tai olla yhtä tai kahta alemmalla korkeudella. Kuvan 21 solmut 4 D, 1 H ja 0 J ovat tästä esimerkkejä.

Seuraavaksi perustelemme, että kullakin korkeudella on solmuja vähintään saman verran kuin sitä suuremmilla korkeuksilla yhteensä. Olkoon m korkeudella k olevien solmujen määrä, ja y sitä korkeammalla olevien solmujen määrä. Vähintään korkeudella k on siis $m+y$ solmua. Niistä jokainen muu paitsi kaikkein ylin solmu on vähintään korkeudella $k+1$ sijaitsevan solmun lapsi. Lasten määrä on siis $m+y-1$. Toisaalta kuten hetki sitten todettiin, vähintään korkeudella $k+1$ olevien solmujen lapsista korkeintaan yksi puuttuu tai on alempana kuin korkeudella k. Siksi $m+y-1 = 2y$ tai $m+y-1 = 2y-1$. Niinpä $y = m-1$ tai $y = m$.

Siis ainakin puolet solmuista on korkeudella 0, ainakin puolet muista solmuista on korkeudella 1, ainakin puolet lopuista solmuista on korkeudella 2 ja niin edelleen. Siksi aliohjelman TEEKEKO suorituksen aikana enintään puolet alkioista voi valua alas edes yhden askeleen, niistä enintään puolet voi valua alas toisenkin askeleen, niistä enintään puolet kolmannenkin askeleen ja niin edelleen. Yhteensä valumisia voi tapahtua enintään $\left(\frac{1}{2} + (\frac{1}{2})^2 + \ldots + (\frac{1}{2})^k\right)n < n$ askelta, missä k on ylimmän solmun korkeus. Tästä syystä TEEKEKO toimii ajassa $O(n)$. Toisaalta jo pelkästään rivillä 2 kuluu aikaa $\Omega(n)$.

Siis TEEKEKO vie aikaa $\Theta(n)$. Aliohjelman LISÄÄKEKOON kutsuminen kullekin alkiolle erikseen vie hitaimmillaan aikaa $\Theta(n \log n)$. Se on suurilla taulukoilla huonompi kuin TEEKEKO. HEAPSORT on silti hitaimmillaan $\Theta(n \log n)$ johtuen rivien $2, \ldots, 5$ silmukan kuluttamasta ajasta. Niinpä aliohjelman TEEKEKO käyttöönotto ei paranna HEAPSORT:in Θ-merkinnällä ilmaistua ajan kulutusta, mutta parantaa sekunneissa mitattuna ainakin hitainta tapausta suurilla taulukoilla.

Toisaalta koska suurin osa alkioista päätyy keon alimmille tasoille, on syytä arvella, että tyypillisessä tapauksessa vain pieni osa alkioista nousee korkealle aliohjelman LISÄÄKEKOON kutsuissa. Niinpä vaikutus keskimääräiseen ajan kulutukseen ei ehkä ole suuri.

Asian kokeilemiseksi tehtiin muutamia mittauksia. Suorittimena oli 11th Gen Intel® Core™ i5-1145G7 @ 2.60GHz × 8 ja muistia oli 16,0 GiB. (”Gi” on 2^{30}.) Ohjelmat oli kirjoitettu C++:lla ja käännetty g++ 11.4.0:lla optioilla `-ansi -W`

	$0 \ldots 4\,999$		$10\,000 \ldots 19\,999$		$10^6 \ldots 1\,000\,019$	
TEEKEKO ja suoristus	0,6 s	5,6 s	8,7 s	1 min 25 s	1,8 s	36 s
TEEKEKO, ei suoristus	0,6 s	5,6 s	9,1 s	1 min 25 s	1,8 s	36 s
LISÄÄKEKOON ja suoristus	0,6 s	6,0 s	9,1 s	1 min 28 s	1,9 s	35 s
LISÄÄKEKOON, ei suoristus	0,7 s	6,0 s	9,6 s	1 min 29 s	1,9 s	35 s

Kuva 22: Kekojärjestämisen (heapsort) muunnelmien suoritusaikaesimerkkejä

`-Wall -pedantic -O3`. Käyttöjärjestelmänä oli Ubuntu 22.04.4 LTS.

Mittausten tuloksia on kuvassa 22. Kukin aikamäärä on usean taulukon luomisen ja järjestämisen yhteensä kuluttama aika. Taulukoita oli yksi kutakin kokoa, missä koot on ilmoitettu ylimmällä rivillä. Esimerkiksi kahdessa oikeanpuolimmaisessa sarakkeessa luotiin ja järjestettiin yhteensä 20 taulukkoa kooltaan miljoonasta miljoona yhdeksääntoista. Alkioiden avaimet oli valittu satunnaisesti nollan ja taulukon koon miinus yksi väliltä. Arvonnat olivat toisistaan riippumattomia, joten jokin luku saattoi tulla samaan taulukkoon useasti, jolloin yksi tai useampi muu luku jäi kokonaan pois. Ensimmäisessä, kolmannessa ja viidennessä sarakkeessa alkiossa ei ollut muuta kuin `int`-tyyppinen avain. Muissa sarakkeissa alkiossa oli lisäksi 400 tavua dataa tyyppiä `char`.

Kukin ajo suoritettiin kolmesti samoilla satunnaisluvuilla. Kuvassa 22 ilmoitettu aika on tulosten mediaani. Suoritusajat vaihtelivat niin paljon, että tuloksia ei ole mielekästä ilmoittaa useammalla merkitsevällä numerolla kuin kuvassa. Näin suuret vaihtelut ovat nykyaikaisille tietokoneille tavallisia. Tavallista on myös, että jos ajoja toistetaan puolen vuoden päästä, niin suoritusajat saattavat muuttua. Mittaustulosten vaihtelua mitattavan kohteen ja mittausolosuhteiden säilyessä samoina kutsutaan ***kohinaksi*** (***noise***).

Joillakin taulukoiden ja alkioiden kokojen yhdistelmillä TEEKEKO oli vähän nopeampi kuin LISÄÄKEKOON kullekin alkiolle erikseen. Muilla yhdistelmillä erot olivat niin pienet, että ne ovat käytännössä merkityksettömät. Ne eivät välttämättä ole edes todelliset, vaan saattavat johtua kohinasta.

Suoristus tarkoittaa aliohjelman kutsun korvaamista aliohjelman kopiolla. Se on englanniksi ***inline***, eikä sillä ole vakiintunutta suomennosta. Se voi nopeuttaa ohjelmaa poistamalla aliohjelman käynnistämiseen ja sammuttamiseen liittyvät toiminnot ja antamalla kääntäjälle mahdollisuuden optimoida koodia aliohjelman kutsuympäristön mukaan. Rekursiivisia aliohjelmia ei voi suoristaa. C++:ssa suoristuskomento on vain suositus, jota kääntäjän ei ole pakko noudattaa. Kuvassa 22 suoristus ei juurikaan vaikuttanut suoritusaikaan, paitsi keskisuurilla taulukoilla pienillä alkioilla.

Teoria ennustaa, että toistettu LISÄÄKEKOON kuluttaa eniten aikaa kun taulukko on valmiiksi järjestyksessä. Kuvassa 23 näytetyt mittaustulokset ovat tämän ennusteen mukaiset, ja niidenkin perusteella TEEKEKO vaikuttaa nopeammalta

0...4 999	TEEKEKO				*n* kertaa LISÄÄKEKOON			
	suoristus		ei suoristus		suoristus		ei suoristus	
etuperin	0,4 s	5,3 s	0,3 s	5,4 s	0,5 s	7,6 s	0,5 s	7,7 s
satunnainen	0,6 s	5,6 s	0,6 s	5,6 s	0,6 s	6,0 s	0,7 s	6,0 s
takaperin	0,4 s	5,0 s	0,4 s	5,1 s	0,4 s	5,3 s	0,5 s	5,4 s

Kuva 23: Kekojärjestämisen suoritusaikojen vertailu eri järjestyksille

kuin toistettu LISÄÄKEKOON. Suoristuksen vaikutus oli niin pieni, että sitä ei erota kohinasta. Rivi ”satunnainen” vastaa kuvan 22 kahta ensimmäistä saraketta.

HEAPSORT kuluttaa vain $\Theta(1)$ lisämuistia. HEAPSORT ei ole vakaa.

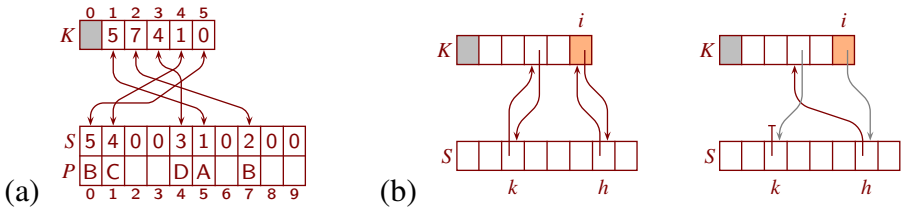
Kuten edellä mainittiin, POISKEOSTA (vastaus 47), MUUTETTUPOISKEOSTA ja TEEKEKO (kuva 20) ovat keskeisiltä osiltaan melkein samat. Usein suositellaan, että melkein samanlaiset toiminnot tulisi esittää yhtenä aliohjelmana, jota kutsutaan sopivilla parametreilla. Tässä tapauksessa se olisi kuitenkin hankalaa, koska (muutettu) POISKEOSTA vertaa tutkittavia alkioita A:ssa olevaan alkioon $A[h]$, mutta TEEKEKO sivuun otettuun alkioon *apu*. Yhdistetyn aliohjelman parametrilistasta tulisi vaikeatajuinen, ja nopeuskin hieman kärsisi ylimääräisestä monimutkaisuudesta.

3.7 Prioriteettijono

Prioriteettijono (***priority queue***) on tietorakenne, johon voidaan milloin tahansa lisätä alkio sekä (kun prioriteettijono ei ole tyhjä) kysyä ja poistaa jokin korkeimman prioriteetin omaava alkio (jos on monta yhtäsuurta, niin prioriteettijono saa palauttaa minkä tahansa niistä). Prioriteettijonoja voidaan käyttää valitsemaan esimerkiksi tehtäviä suoritettavaksi kiireellisin ensin tai shakkipelin siirtoja kokeiltavaksi lupaavimmalta vaikuttava ensin. Luvussa 4 prioriteettijonoa käytetään mahdollisimman lyhyen reitin etsimiseen maantiekartassa.

Jos alkion prioriteettia ei tarvitse muuttaa alkion ollessa prioriteettijonossa, niin keosta saa tehokkaan prioriteettijonon. Jonkin korkeimman prioriteetin alkion saa selville ajassa $\Theta(1)$ lukemalla lokeron 0. LISÄÄKEKOON ja POISKEOSTA toimivat (tasatussa) ajassa $O(\log n)$. Jonkin korkeimman prioriteetin alkion etsiminen mielivaltaisessa järjestyksessä olevasta taulukosta olisi $\Theta(n)$. Lisääminen koko ajan prioriteettijärjestyksessä olevaan taulukkoon olisi tyypillisesti $\Theta(n)$. Ne ovat sekä teoriassa että tyypillisesti myös käytännössä olennaisesti huonommat kuin $\Theta(\log n)$.

Jos prioriteetti voi parantua mutta ei huonontua alkion ollessa keossa ja alkio tarvitsee käsitellä vain kerran, niin alkion voi laittaa uudelleen kekoon aina



Kuva 24: (a) Prioriteettijonon, jossa prioriteetteja voi muuttaa, tietorakenne; (b) poistamisen alku

prioriteetin parantuessa. Silloin sama alkio voi olla keossa useana kappaleena samanaikaisesti. Ylimääräiset kappaleet voi tunnistaa ja hylätä niiden tullessa ulos keosta esimerkiksi sillä perusteella, että alkio on jo käsitelty, tai sillä perusteella, että alkion prioriteetti on tallennettu myös keon ulkopuolelle ja se on korkeampi kuin keosta tullut prioriteetti. Tulemme näkemään jälkimmäisestä esimerkin luvussa 4.4.

Alkioiden oleminen keossa useana kappaleena hidastaa suoritusta varsin vähän. Jos keossa on n eri alkiota ja jokainen niistä on n kappaleena, on keossa n^2 alkiota. Niinpä lisäyksen ja poiston suoritusaika on hitaimmillaan verrannollinen lukuun $\log n^2$. Koska $\log ab = \log a + \log b$ kun $a > 0$ ja $b > 0$, on $\log n^2 = 2\log n$. Siis vaikka alkioiden määrä neliöityi, huonoin suoritusaika vain kaksinkertaistui!

55. Jos prioriteetti voi sekä parantua että huonontua alkion ollessa keossa, niin miksi sen tarkastaminen, että onko alkio jo käsitelty, ei yksinään riitä varmistamaan, että kaikki muut kopiot paitsi yksi oikea hylätään?

56. Jos prioriteetti voi sekä parantua että huonontua alkion ollessa keossa, niin miksi keosta tulleen prioriteetin vertaaminen keon ulkopuolelle tallennettuun prioriteettiin ei yksinään riitä varmistamaan, että kaikki muut kopiot paitsi yksi oikea hylätään?

Edellä esitetyt keinot joutuvat vaikeuksiin, jos tarvitaan jo käsittelyssä olleen alkion laittamista uudelleen prioriteettijonoon. Sellainen tarve voi syntyä esimerkiksi kun jokin työtehtävä keskeytetään ja laitetaan jonottamaan uutta aikaa työn jatkamiseksi. Tällöin on vaikea erottaa edellisen jonotuksen yhteydessä prioriteettijonoon laitettuja ylimääräisiä kopioita uuden jonotuksen kopioista.

Tähänkin on ratkaisu. On mahdollista toteuttaa prioriteettijono keon avulla siten, että prioriteettia voi muuttaa jonotusaikana, kukin alkio on keossa aina enintään yhtenä kappaleena, ja jokaisen toiminnon ajan kulutus on (tasatusti) $O(\log n)$. Sitä varten alkioille annetaan juoksevat numerot nollasta alkaen, ja ylläpidetään taulukkoa S, joka kertoo alkion sijainnin keossa alkion numeron perusteella. Toinen taulukko P kertoo alkion prioriteetin alkion numeron perusteella. Keossa olevassa alkiossa ei ole muuta tietoa kuin alkion numero. Tätä ratkaisua on havainnollistettu kuvassa 24 (a), ja se on esitetty pseudokoodina kuvassa 25. Kuvissa

```
    MUUTAPRIORITEETTI(k, p)
1   if k ≥ S.koko then
2     S.kooksi(k+1,0); P.kooksi(k+1)
3   if S[k] = 0 then
4     i := K.koko; K.kooksi(i+1)
5   else
6     i := S[k]
7   P[k] := p; j := ⌊i/2⌋
8   while j > 0 && P[K[j]] < p do
9     K[i] := K[j]; S[K[i]] := i; i := j; j := ⌊i/2⌋
10  while true do
11    j := 2i
12    if j+1 < K.koko && P[K[j+1]] ≥ P[K[j]] then
13      j := j+1
14    if j ≥ K.koko || P[K[j]] ≤ p then break
15    K[i] := K[j]; S[K[i]] := i; i := j
16  K[i] := k; S[k] := i

    POISPJONOSTA(k)
17  if k ≥ S.koko || S[k] = 0 then return
18  i := K.koko − 1; h := K[i]
19  K.kooksi(i)
20  S[h] := S[k]; S[k] := 0
21  if h ≠ k then
22    MUUTAPRIORITEETTI(h, P[h])

    POISPJONOSTA
23  if K.koko > 1 then
24    POISPJONOSTA(K[1])

    ALUSTAPJONO
25  S.kooksi(0); P.kooksi(0)
26  K.kooksi(1)
```

Kuva 25: Prioriteettijono, jossa prioriteetteja voi muuttaa, pseudokoodina

keon nimi on K.

Sitä, että avaimet eivät ole järjestettävässä taulukossa vaan siellä on vain tieto niiden paikoista toisessa taulukossa, tullaan käyttämään luvussa 5.3 nopeuttamaan taulukoiden järjestämistä silloin, kun alkiot ovat suuria. Kuten Wikipedian sivulla ”Partition refinement” kerrotaan, samankaltaista ratkaisua käytetään myös joissakin graafialgoritmeissa sekä deterministisen äärellisten automaattien minimoinnissa.

Keon alkiot indeksoidaan kuvissa 24 ja 25 ykkösestä alkaen, jotta 0 jää vapaaksi tarkoittamaan, että alkio ei ole keossa. Siksi $K[0]$ on jätetty käyttämättä. Niinpä $K.koko$ on yhden suurempi kuin keon koko.

Prioriteettijonon luomisen yhteydessä suoritetaan ALUSTAPJONO. Sen jäljiltä keko on tyhjä eli K sisältää vain lokeron 0, joten $K.koko = 1$. Taulukot S ja P ovat tyhjät.

MUUTAPRIORITEETTI huolehtii sekä alkion lisäämisestä prioriteettijonoon että prioriteettijonossa jo olevan alkion prioriteetin muuttamisesta. Kummassakin tapauksessa alkion numero on k. Riveillä 1 ja 2 tarvittaessa laajennetaan taulukoita S ja P niin että niissä on tilaa lisättävän alkion tiedoille. Alkioiden numeroiden on tarkoitus olla juoksevia numeroita, mutta tietorakenne ei rajoita niitä muuten kuin että muistia kuluu suurimpaan käytössä olevaan numeroon verrannollisesti. Niinpä jos k on hyvin suuri, niin muisti voi loppua kesken.

Taulukon S uudet alkiot alustetaan arvoon 0. Jos alkio ei vielä ole keossa, niin sille varataan tilaa rivillä 4. Alkion juuri luotu tai olemassa ollut paikka keossa tallennetaan i:hin. Uusi prioriteetti tallennetaan P:hen rivillä 7.

Jos alkio tai sille varattu uusi paikka on alempana keossa kuin alkion (uusi) prioriteetti edellyttää, niin riveillä 7, ..., 9 paikkaa siirretään ylöspäin eli alkion (esi)vanhempia siirretään alaspäin kunnes alkion paikka on oikea. Jos paikka on ylempänä keossa kuin alkion (uusi) prioriteetti edellyttää, niin riveillä 10, ..., 15 paikkaa siirretään alaspäin eli alkion (lapsen)lapsia siirretään ylöspäin kunnes alkion paikka on oikea. Rivillä 16 alkio laitetaan paikalleen. Aina kun jonkin alkion paikkaa vaihdetaan eli sijoitetaan K:hon, tallennetaan alkion uusi paikka S:ään.

POISPJONOSTA(k) ottaa alkion k pois prioriteettijonosta. Jos se ei ollut siellä alun perinkään, niin lopetetaan rivillä 17. Muussa tapauksessa riveillä 18 ja 19 keon viimeisen alkion numero otetaan talteen h:hon ja kekoa pienennetään. Sitten keon viimeisen alkion h paikaksi keossa asetetaan poistettavan alkion k paikka ja poistettava alkio merkitään poistetuksi. Jos poistettava alkio on viimeisenä ollut alkio eli $k = h$, niin lopetetaan. Muussa tapauksessa tilanne on kuten kuvassa 24 (b). Viimeisenä ollut alkio (jolle on nyt varattu poistetun alkion paikka) siirretään keossa prioriteettinsa mukaiselle paikalle. Kuvan vasemmanpuoleinen harmaa nuoli korjaantuu siinä yhteydessä, ja oikeanpuoleista ei tarvitse korjata.

POISPJONOSTA poistaa epätyhjästä prioriteettijonosta ensimmäisen alkion, eikä tee tyhjälle prioriteettijonolle mitään. Rivi 23 varmistaa, että $K[1]$:stä indeksoidaan vain jos se on olemassa.

57. Onko kuvan 25 prioriteettijonossa jokin suurin ensin vai jokin pienin ensin? Mistä sen voi päätellä?

58. Jos keosta poistetaan sen viimeinen alkio, niin $S[h] := S[k]$ ei muuta mitään rivillä 20. Kannattaisiko se siirtää ehdon $h \neq k$ alle rivien 21 ja 22 väliin?

Prioriteettijonollamme on seuraava ***luokkainvariantti*** (***class invariant***), eli väittämä, joka on voimassa aina kun minkään aliohjelman suoritus ei ole kesken. Sen osa 1 sanoo, että taulukot S ja P ovat samankokoiset. Osa 2 ilmaisee, että K:n alkiot ovat laillisia indeksejä S:lle, ja että keossa kohdassa i oleva alkio on S:nkin mukaan siinä. Osa 3 ilmaisee, että S:n nollasta poikkeavat alkiot ovat laillisia indeksejä K:lle, ja että jos $S[i] \neq 0$, niin keossa kohdassa $S[i]$ todella on alkio i. Osa 4 ilmaisee keko-ominaisuuden ottaen huomioon, että lokero 0 ei ole mukana ja että prioriteetti katsotaan P:stä alkion numeron perusteella.

1. $P.koko = S.koko$
2. Jokaisella $1 \leq i < K.koko$ pätee: $0 \leq K[i] < S.koko$ ja $S[K[i]] = i$.
3. Jokaisella $0 \leq i < S.koko$ pätee: jos $S[i] \neq 0$, niin $1 \leq S[i] < K.koko$ ja $K[S[i]] = i$.
4. Jokaisella $2 \leq i < K.koko$ pätee: $P[K[\lfloor \frac{i}{2} \rfloor]] \geq P[K[i]]$.

ALUSTAPJONO asettaa invariantin kaikki osat voimaan. Invariantin osa 1 säilyy voimassa, koska S:n tai P:n kokoa muutetaan vain rivillä 2, ja ne asetetaan keskenään samankokoisiksi. Osoitamme seuraavaksi, että jos parametrina tuleva k on alkion numero, niin invariantin osat 2 ja 3 säilyvät voimassa.

Aina kun S:ään lisätään alkioita, osa 3 pätee niille, koska ne alustetaan nollaksi. Rivillä 4 luotuun tai rivillä 6 löydettyyn K:n lokeroon sijoitetaan jonkin alkion numero rivillä 9, 15 tai 16, ja sitä ennen sen sisältöä ei käytetä. Rivien 9 ja 15 sijoitukset eivät vaaranna invariantin osan 2 alkuosaa, koska arvot kopioidaan toisista K:n alkioista. Rivillä 16 K:hon sijoitettava alkion numero on rivillä 2 varmistettu riittävän pieneksi. Samoilla riveillä S:ään sijoitettavat luvut ovat laillisia keon indeksejä, ja valittu säilyttämään invariantin osien 2 ja 3 loppuosat voimassa.

Riveillä 18 ja 19 keon viimeinen lokero poistuu. Jos sen sisältämä numero kuuluu alkiolle jonka pitikin poistua, eli jos $K[K.koko-1] = k$ rivillä 18, niin viittaus siihen poistuu S:stä rivillä 20. Muussa tapauksessa rivin 22 kutsu käyttäytyy ikään kuin viimeisenä ollut alkio h olisi poistetun alkion k kohdalla keossa, ja sen prioriteettia muutettaisiin. Tässä vaiheessa $K[S[h]] = k \neq h$, mutta sitä ei tarvitse korjata, koska MUUTAPRIORITEETTI ei käytä sen arvoa.

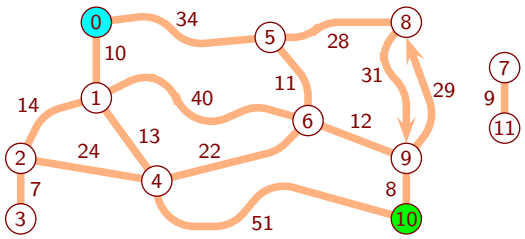
Näistä syistä invariantin osat 2 ja 3 säilyvät voimassa. Invariantin osan 4 voi tarkastaa samaan tapaan kuin aikaisemminkin.

4 Sovellusesimerkki: reitin etsintä

Tässä luvussa sovelletaan kekoon lisäämistä ja keosta poistamista jonkin lyhimmän reitin etsimiseen tiekartassa (jos yhtä pitkiä on monta, ohjelma valitsee jonkin niistä). Esimerkki esitetään siinä mielessä täydellisenä C++-ohjelmana, että sen voi kääntää ja suorittaa ilman että siihen lisätään mitään. Se ei ole täydellinen ohjelmointityyliltään, muun muassa siksi että se ei veisi tarpeettoman paljon tilaa, sekä siksi, että ei vallitse yksimielisyyttä siitä, mikä on ja ei ole hyvää ohjelmointityyliä. Ohjelma perustuu Dijkstran algoritmina tunnettuun algoritmiin. Luvussa myös esitellään sen muunnelma A* ja kaksi muunlaista reitinetsintäalgoritmia.

4.1 Ohjelman alkuosat

Kuvassa 26 on esimerkki tiekartasta. Suurin osa tienpätkistä on kaksisuuntaisia, mutta paikasta 8 on 31 km pitkä yksisuuntainen tienpätkä paikkaan 9, ja paikasta 9 on 29 km pitkä yksisuuntainen tienpätkä paikkaan 8. Tienpätkät esitetään tietokoneelle yksisuuntaisina. Kaksisuuntainen tienpätkä esitetään antamalla yksisuuntainen tienpätkä kumpaankin suuntaan. Niinpä esimerkiksi paikkojen 0 ja 5 välinen 34 km pitkä tienpätkä esitetään tietokoneelle kahtena tienpätkänä joista kummankin pituus on 34 km, yksi vie paikasta 0 paikkaan 5 ja toinen vie paikasta 5 paikkaan 0.



Kuva 26: Esimerkki tiekartasta

Kuvassa on useita reittejä paikasta 0 paikkaan 10, kuten paikkojen 1 ja 4 kautta kulkeva $(10+13+51)$ km eli 74 km pitkä ja paikkojen 5, 8 ja 9 kautta kulkeva 101 km pitkä. Lyhimpien pituus on 65 km. Yksi lyhin reitti kulkee paikkojen 1, 4, 6 ja 9 kautta, ja toinen paikkojen 5, 6 ja 9 kautta. Paikasta 0 ei ole yhtään reittiä paikkaan 7.

Ohjelma alkaa ottamalla käyttöön C++-kirjaston, jossa on syötteen lukemis- ja tulostamistoimintoja. Käyttöön otetaan myös taulukot, joiden kokoa voi muuttaa kesken suorituksen.

```
1  #include <iostream>
2  #include <vector>
```

Jokaiseen paikkaan liittyy muuttuja, jossa pidetään kirjaa jonkin lyhimmän jo löydetyn lähdöstä siihen paikkaan vievän reitin pituudesta. Siihen kannattaa aluksi sijoittaa suurin sen tyyppiin mahtuva arvo, koska siten vältetään tarve käsitellä erikoistapauksena tilanne, jossa paikkaan ei ole vielä löydetty yhtään reittiä. Suurin tavallisten nykyaikaisten tietokoneiden liukulukutyyppeihin `float` ja `double` mahtuva arvo on ääretön. C++:ssa sen voi hakea kirjastosta `limits` tai tuottaa lausekkeella `1/.0`. Desimaalipiste nollan edessä on tärkeä, koska sen ansiosta lauseke lasketaan liukuluvuilla. Lauseke `1/0` laskettaisiin kokonaisluvuilla. Se ei tuottaisi ääretöntä, vaan kaataisi ohjelman.

```
3  const double aareton = 1/.0;
```

Reitin etsimistehtävän yhteydessä paikkoja on tapana kutsua ***solmuiksi*** (***vertex*** tai ***node***) ja niiden välisiä tienpätkiä ***kaariksi*** (***edge*** tai ***arc***). Tämän ohjelman tarpeisiin sopiva tapa esittää kaaret on taulukko, jossa on ensin solmusta 0 lähtevät kaaret, sitten solmusta 1 lähtevät ja niin edelleen. Kustakin kaaresta tallennetaan sen edustaman tienpätkän pituus ja sen solmun numero, joka edustaa paikkaa, jonne tienpätkä vie.

```
4  struct kaarityyppi{ unsigned minne; double pituus; };
5  std::vector< kaarityyppi > kaaret;
```

Koska ohjelmassa on vain yksi kaaritaulukko, ei sitä välitetä aliohjelmien parametrina vaan siitä on tehty globaali muuttuja. Globaalien muuttujien käyttöä pidetään usein hyvän ohjelmointityylin vastaisena. Tässä esimerkissä siihen kuitenkin päädyttiin, koska kaiken välittäminen parametreina olisi tehnyt aliohjelmien parametrilistoista pitkiä ja sekavia. Yksi mahdollisuus olisi kapseloida tietorakenteet ja niitä käyttävät aliohjelmat luokaksi tai nimiavaruudeksi. Sekin jätettiin tekemättä, koska se olisi pidentänyt ja monimutkaistanut tätä tekstiä eikä olisi tuonut lisäarvoa tietorakenteiden ja algoritmien näkökulmasta.

Myös solmut on kätevä esittää taulukkona. Jos $i > 0$, niin solmusta i lähtevät kaaret ovat taulukon `kaaret` lokeroissa `solmut[`$i-1$`].kaarten_loppu`, ...,

`solmut[`i`].kaarten_loppu` -1. Solmusta 0 lähtevät kaaret alkavat taulukon `kaaret` lokerosta 0. Muuttuja `etaisyys` sisältää jonkin lyhimmän löydetyn lähdöstä solmuun vievän reitin pituuden. Muuttuja `reitti` sisältää edellisen solmun numeron tällä reitillä, paitsi lähtösolmun tapauksessa.

```
6  struct solmutyyppi{
7    unsigned kaarten_loppu, reitti; double etaisyys;
8    solmutyyppi( unsigned kl ):
9      kaarten_loppu( kl ), reitti( ~0u ), etaisyys( aareton ) {}
10 };
11 std::vector< solmutyyppi > solmut;
```

Aina kun luodaan uusi solmu, rivit 8 ja 9 alustavat sen. Kaarten loppukohta asetetaan luonnin yhteydessä annetun tiedon mukaisesti. Koska solmuun ei vielä ole löydetty reittiä, asetetaan etäisyydeksi ääretön ja edellisen solmun numeroksi suurin `unsigned`-tyyppiin mahtuva luku eli `~0u`. Ohjelman kannalta ei ole tärkeää, mikä asetetaan muuttujan `reitti` alkuarvoksi. Virheiden löytämisen kannalta on kuitenkin eduksi asettaa alkuarvo, joka kuvastaa sitä, että reittiä ei vielä ole. Siihen `~0u` on paras, koska sillä on pienin mahdollisuus olla jonkin solmun numero.

Seuraavaksi luodaan muuttujat etsittävän reitin alku- ja loppupaikoille.

```
12 unsigned lahto = ~0u, maali = ~0u;
```

4.2 Syötteen lukeminen

Kuvassa 27 on yksinkertainen aliohjelma syötteen lukemiseksi. Syötteessä on oltava ensin lähtösolmun numero, sitten maalisolmun numero ja lopuksi kaaret. Kaaret ovat kolmikoita, joissa on ensin häntäpään solmun numero, sitten kärkipään solmun numero ja lopuksi pituus. Ne täytyy antaa häntäpään solmun numeron mukaisessa kasvavassa järjestyksessä. (Luvussa 8.1 syöte suunnitellaan uusiksi niin että tämä rajoitus poistuu.) Pituus ei saa olla negatiivinen eikä ääretön. Eri tiedot täytyy erottaa toisistaan välilyönneillä, rivinsiirroilla ja/tai muulla niin sanotulla valkoisella tilalla (***whitespace***). Syötteen alussa, lopussa ja eri tietojen välissä saa olla ylimääräisiä välilyöntejä, rivinsiirtoja ja muuta valkoista tilaa miten paljon tahansa. Aliohjelma `lue_syote` palauttaa tiedon, onnistuiko syötteen lukeminen.

Rivillä 14 luetaan lähdön ja maalin numerot. Jos se ei onnistunut, niin rivillä 16 annetaan virheilmoitus ja lopetetaan.

Rivien 19, . . . , 34 silmukassa luetaan ja tallennetaan kaaria kunnes syöte loppuu tai lukeminen epäonnistuu. Muuttujassa `solmu` on aluksi pienin mahdollinen solmun numero eli 0, ja myöhemmin viimeksi luetun kaaren häntäpään solmun

```
13  bool lue_syote(){
14    std::cin >> lahto >> maali;
15    if( !std::cin ){
16      std::cout << "!!! Lähtö tai maali puuttuu\n"; return false;
17    }
18    unsigned solmu = 0, max_muu = lahto > maali ? lahto : maali;
19    while( true ){
20      kaarityyppi kaari; unsigned vanha = solmu;
21      std::cin >> solmu >> kaari.minne >> kaari.pituus;
22      if( !std::cin ){ break; }
23      if( solmu < vanha ){
24        std::cout << "!!! Väärä mistä-solmu " << solmu << "\n";
25        return false;
26      }
27      if( !( 0. <= kaari.pituus && kaari.pituus < aareton ) ){
28        std::cout << "!!! Laiton pituus " << kaari.pituus << "\n";
29        return false;
30      }
31      solmut.resize( solmu + 1, solmutyyppi( kaaret.size() ) );
32      kaaret.push_back( kaari ); ++solmut[ solmu ].kaarten_loppu;
33      if( max_muu < kaari.minne ){ max_muu = kaari.minne; }
34    }
35    if( max_muu >= solmut.size() ){
36      solmut.resize( max_muu + 1, solmutyyppi( kaaret.size() ) );
37    }
38    return true;
39  }
```

Kuva 27: Reitin etsinnän syötteen lukeminen

numero. Muuttujan `vanha` avulla tarkastetaan, että kaaret saapuvat häntäpään solmun mukaisessa kasvavassa järjestyksessä. Rivillä 27 käytetty tapa tarkastaa kaaren pituus toimii myös jos pituutena on `double`-tyyppiin kuuluva erikoisarvo ***epäluku*** (***not a number***). Jotta kunkin solmun kaarten alkukohta todella olisi edellisen solmun `kaarten_loppu`, luodaan solmuja rivillä 31 sitä mukaa kuin syötteessä tulevien kaarten häntäpäiden solmujen numerot kasvavat.

Syötteessä saattaa esiintyä myös solmuja, joiden numero on suurempi kuin suurin kaaren häntäpään numero. Siksi muuttujassa `max_muu` pidetään kirjaa suurimmasta solmun numerosta, joka esiintyy lähtönä, maalina tai kaaren kärkipäänä. Sen avulla luodaan tarvittaessa lisää solmuja rivillä 36. Jos C++:n `resize` saa ensimmäiseksi parametrikseen pienemmän luvun kuin taulukon senhetkinen koko, niin se pienentää taulukkoa. Siksi rivillä 35 varmistetaan, että sitä kutsutaan vain jos solmuja tarvitsee luoda lisää. Rivi 31 ei tarvitse samanlaista suojausta seuraavasta syystä.

Merkitsemme $n =$ `solmut.size()`. Osoitamme, että aina rivin 31 alussa pätee `solmu` $\geq n-1$. Todistuksemme on invarianttitodistus sillä muutoksella, että nyt invariantin ei väitetä olevan voimassa silmukan alkukohdassa vaan silmukan rungon sisällä olevassa kohdassa. Kun riville 31 tullaan ensimmäisen kerran, on `solmu` $= 0$ rivin 18 vuoksi, ja $n = 0$ koska yhtään solmua ei vielä ole luotu. Rivi 31 asettaa voimaan $n =$ `solmu` $+1$ eli `solmu` $= n-1$. Kun riville 31 tullaan uudelleen, on rivi 20 asettanut `vanha` $= n-1$. Aina rivillä 31 pätee `solmu` $\geq$ `vanha`, koska muuten olisi menty riville 25 ja lopetettu. Niinpä `solmu` $\geq n-1$.

Toiminto `resize` alustaa luodut solmut jälkimmäisen parametrinsa mukaisesti. Niinpä sekä rivillä 31 että rivillä 36 kunkin niistä `kaarten_loppu` saa ensimmäiseksi arvokseen senhetkisen kaarten määrän. Aina kun kaarten määrä kasvaa rivillä 32, kasvaa myös `solmut[ solmu ].kaarten_loppu`. Siksi koko ajan `solmut[ solmu ].kaarten_loppu` = `kaaret.size()`. Rivin 21 vuoksi `solmu` on lisätyn kaaren häntäpään solmun numero, joten kaaret lisätään sen solmun nimiin, jolle ne kuuluvatkin.

Ainoa taulukon indeksointi on rivillä 32. Se on laillinen, koska edellisellä rivillä varmistettiin, että taulukko on riittävän suuri.

Jos syöte on hyvin suuri, saattaa muisti loppua kesken rivillä 31, 32 tai 36, joissa varataan tilaa uusille solmuille tai kaarille. Hyvin suurilla syötteillä vaarana on myös `unsigned`-tyypin lukualueen loppuminen kesken. Kuten sivulla 17 kerrottiin, sen sijaan voi käyttää tyyppiä `std::size_t`, joka riittää taulukoille aina. Jos siirrytään siihen, niin `~0u`:n tilalle pitää laittaa `~0uz`.

Kunkin kaaren lukemiseen ja tallentamiseen menee tasatusti $\Theta(1)$ aikaa. (”Tasattu” ajan kulutus on selostettu luvussa 3.3 ja ”$\Theta(1)$” on selostettu sivulla 27.) Lisäksi jokaista solmua kohti kuluu tasatusti $\Theta(1)$ aikaa vaikka solmua ei olisi mainittu syötteessä, koska se luodaan ja alustetaan rivillä 31 tai 36. (Syötteessä mainitsematon numero on solmun numero, jos ja vain jos se on vähintään nolla ja pienempi kuin suurin syötteessä mainittu solmun numero.) Niinpä jos solmujen määrä on n ja kaarten määrä m, niin ajan kulutus on $\Theta(n+m)$.

4.3 Kekotoiminnot

Reitin etsinnän käyttämät kekotoiminnot ovat kuvassa 28. Aliohjelman `lisaa_kekoon` ainoat sisällölliset erot kuvaan 16 ovat pituusvertailun suunta rivillä 44 sekä erot tietojen välityksessä. Keko on globaali muuttuja, ja parametreina ei anneta kekotietuetta vaan sen kumpikin osa erikseen. Aliohjelma `poista_keosta` eroaa sisällöllisesti vastauksesta 47 vain pituusvertailun suunnan ja muuttujan keko välityksen osalta.

Kekoon tallennettavat pituudet ovat lähdöstä alkavien reittien pituuksia. Muualla ohjelmassa lähdöstä alkavien reittien pituuksia kutsutaan etäisyyksiksi. Kut-

```
40 std::vector< kaarityyppi > keko;

41 inline void lisaa_kekoon( unsigned solmu, double pituus ){
42   unsigned i = keko.size(), j = (i-1) / 2;
43   keko.resize(i+1);
44   while( i > 0 && keko[j].pituus > pituus ){
45     keko[i] = keko[j]; i = j; j = (i-1) / 2;
46   }
47   keko[i].minne = solmu; keko[i].pituus = pituus;
48 }

49 inline void poista_keosta(){
50   unsigned h = keko.size() - 1, i = 0, j = 1;
51   while( true ){
52     if( j+1 < h && keko[j+1].pituus <= keko[j].pituus ){ ++j; }
53     if( j >= h || keko[j].pituus >= keko[h].pituus ){ break; }
54     keko[i] = keko[j]; i = j; j = 2*i + 1;
55   }
56   keko[i] = keko[h]; keko.resize(h);
57 }
```

Kuva 28: Reitin etsinnän kekotoiminnot

sumalla niitä keossa pituuksiksi vältettiin sellaisen uuden tyypin luominen keon alkioita varten, joka eroaa jo olemassa olevasta vain yhden nimen osalta.

4.4 Reitin etsintä

Nyt päästään reitin etsinnän ytimeen! Kuvassa 29 on toteutettu Dijkstran algoritmi eli Edsger Dijkstran 1950-luvulla esittämä algoritmi jonkin lyhimmän reitin etsimiseen suunnatussa graafissa, kuten maantiekartassa. Se käy reitin pituuden mukaisessa kasvavassa järjestyksessä läpi niitä solmuja, joihin on olemassa reitti solmusta `lahto`, kunnes käsittelyvuoroon tulee `maali` tai käsiteltävät solmut loppuvat. Käsiteltäväksi valitut solmut odottavat tässä toteutuksessa käsittelyvuoroaan keossa. Keon alkiossa on myös solmun etäisyys silloin, kun solmu laitettiin kekoon. ***Etäisyys*** tarkoittaa pienintä jo löydetyn lähdöstä solmuun vievän reitin pituutta. Solmu laitetaan kekoon uudelleen joka kerta kun löydetään aikaisempaa lyhyempi reitti. Siksi sama solmu voi olla keossa samanaikaisesti monta kertaa eri etäisyyksillä.

Rivillä 59 merkitään, että solmuun `lahto` johtaa lähdöstä reitti, jonka pituus on 0. Sitten ensimmäiseksi käsittelyvuoroa odottavaksi solmuksi laitetaan `lahto` etäisyydellä 0.

Rivillä 61 annetaan keon ylimmälle solmulle nimi `solmu1` ja otetaan sen keosta saatu etäisyys talteen. Sitten solmu poistetaan keosta. Sama solmu saattaa jäädä

```
58  bool etsi_reitti(){
59    solmut[ lahto ].etaisyys = 0.; lisaa_kekoon( lahto, 0. );
60    while( !keko.empty() ){
61      unsigned solmu1 = keko[0].minne; double etaisyys = keko[0].pituus;
62      poista_keosta();
63      if( solmu1 == maali ){ return true; }
64      if( solmut[ solmu1 ].etaisyys < etaisyys ){ continue; }
65      unsigned i = solmu1 ? solmut[ solmu1 - 1 ].kaarten_loppu : 0;
66      for( ; i < solmut[ solmu1 ].kaarten_loppu; ++i ){
67        unsigned solmu2 = kaaret[i].minne;
68        etaisyys = kaaret[i].pituus + solmut[ solmu1 ].etaisyys;
69        if( etaisyys < solmut[ solmu2 ].etaisyys ){
70          solmut[ solmu2 ].etaisyys = etaisyys;
71          solmut[ solmu2 ].reitti = solmu1;
72          lisaa_kekoon( solmu2, etaisyys );
73        }
74      }
75    }
76    return false;
77  }
```

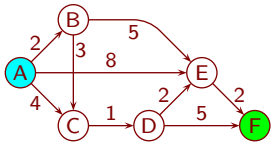
Kuva 29: Dijkstran reitinetsintäalgoritmi C++:lla

kekoon muina kappaleina, joilla on suurempi etäisyys.

Jos keosta tullut solmu on `maali`, niin lopetetaan paluuarvolla `true`. Se kertoo, että reitti löytyi. Muussa tapauksessa testataan rivillä 64, onko keosta tullut etäisyys suurempi kuin solmun tietoihin kirjattu etäisyys. Jos se on, niin solmu on käsitelty jo aiemmin, nimittäin silloin kun keosta tuli solmun tietoihin kirjattu etäisyys. Siksi palataan riville 60 hakemaan keosta seuraavaa solmua tai huomaamaan, että keko on tyhjä.

Muussa tapauksessa solmusta lähtevät kaaret käydään läpi riveillä 65, ..., 74. Jokaisesta selvitetään, mihin solmuun `solmu2` se vie ja kuinka pitkä on reitti lähdöstä sinne `solmu1`:n ja käsiteltävän kaaren kautta. Jos se on lyhyempi kuin mikään aiemmin löydetty reitti, niin merkitään `solmu2`:n tietoihin että siihen pääsee `solmu1`:n kautta juuri lasketulla etäisyydellä, sekä lisätään `solmu2` käsittelyä odottaviin solmuihin samalla etäisyydellä.

59. Algoritmi suoritetaan alla näytetylle kartalle siten, että lähtö on A ja maali on F. Kerro, mitkä parit ovat keossa, kun tullaan ensimmäisen, toisen ja niin edelleen kerran riville 60. Riittää, että luettelet kulloinkin keossa olevat parit tyyliin ”X8 Y4 Z7”, ja saat luetella ne missä tahansa järjestyksessä!



60. Aliohjelmaa `lisaa_kekoon` kutsutaan sekä rivillä 59 että rivillä 72. Jos kumpikin kutsu suoristetaan (sivu 48), niin aliohjelman koodi tulee ohjelmaan kahteen kertaan. Miten tämän voi välttää ja silti saada suoristamisen nopeushyöty?

Aliohjelman `etsi_reitti` osoittaminen oikeaksi perustuu kolmeen invarianttiin, jotka pätevät aina rivin 60 alussa.

1. Jokainen keossa oleva tieto ja jokainen alustuksen jälkeen mihinkään solmuun merkitty tieto on jonkin reitin mukainen.

Kun riville 60 tullaan ensimmäisen kerran, on keossa vain tieto, että lähtösolmusta pääsee itseensä kulkemalla reitin, jonka pituus on nolla. Se on totta. Sama tieto on myös lähtösolmussa. Minkään muun solmun tietoja ei ole vielä muutettu. Sen jälkeen kekoon lisätään tietoja ja solmujen tietoja muutetaan vain riveillä 70, ..., 72. Uudet tiedot perustuvat `solmu1`:n tietoihin jotka ovat invariantin vuoksi jonkin reitin mukaiset, sekä `solmu1`:n jonkin lähtökaaren tietoihin. Siksi uudet tiedot ovat jonkin reitin mukaiset. Invariantin 1 perustelu päättyy tähän.

Jatkossa solmun ***minimietäisyydellä*** tarkoitetaan jonkin lyhimmän lähdöstä solmuun vievän reitin pituutta, vaikka reittiä ei olisi vielä löydetty. Riveistä 59 ja 69 seuraa, että alustuksen jälkeen minkään solmun `etaisyys` ei kasva. Siitä ja invariantista 1 seuraa toinen invariantti:

2. Jos minkään solmun `etaisyys` on koskaan solmun minimietäisyys, niin se on siitä eteenpäin `etsi_reitti`:n loppuun saakka solmun minimietäisyys.

Tarkoittakoon lähdöstä mihin tahansa solmuun vievän minimipituisen ***reitin kärki*** solmua, joka on keossa minimietäisyytensä kanssa, ja jonka jälkeen tällä reitillä minkään solmun `etaisyys` ei ole minimietäisyys.

61. Perustele, että `solmut[` *kärki* `].etaisyys` on minimietäisyys!

62. Perustele, että minimipituisella reitillä on enintään yksi kärki!

Osoitamme, että myös seuraava on invariantti.

3. Jokaiselle solmulle pätee: solmun `etaisyys` on minimietäisyys tai solmuun johtaa minimipituinen reitti, jolla on kärki.

Jos solmuun ei ole reittiä, niin solmun minimietäisyys on ääretön. Solmun `etaisyys` on sama, sillä se alustettiin äärettömäksi, eikä invariantin 2 mukaan muutu.

Muussa tapauksessa solmuun johtaa minimipituinen reitti, ja sen jokaisen solmun minimietäisyys on äärellinen. Invariantti 3 astuu voimaan rivillä 59 siten, että

`lahto` on keossa minimietäisyytensä kanssa, ja reitin muiden solmujen `etaisyys` on ääretön. Niinpä solmusta `lahto` tuli reitin kärki. Sen jälkeen invariantti 3 säilyy voimassa seuraavista syistä.

Invariantti 3 on vaarassa rikkoutua, kun minimipituisen reitin kärki poistetaan keosta rivillä 62. Se sai nimen `solmu1` rivillä 61.

Jos se on reitin viimeinen solmu, niin invariantin 3 alkuosa oli voimassa kysymyksen 61 vuoksi ja pysyi voimassa invariantin 2 vuoksi.

Jos `solmu1` on `maali`, niin riville 60 ei enää tulla rivin 63 vuoksi.

Muussa tapauksessa jatketaan riville 64. Sen ehto ei toteudu, koska muutoin `solmu1`:n `etaisyys` olisi alle minimietäisyyden, joka olisi vastoin invarianttia 1. Koska puheena oleva reitti on minimipituinen, niin kun sen seuraava kaari tulee käsittelyvuoroon rivillä 68, saa `etaisyys` arvokseen seuraavan solmun minimietäisyyden. Tätä kaarta ei hylätä rivillä 69, koska reitin kärjen määritelmä sanoo, että seuraavan solmun `etaisyys` ei ole minimietäisyys, ja invariantti 1 estää sitä olemasta vielä pienempi. Siksi seuraava solmu laitetaan kekoon ja siitä tulee uusi reitin kärki.

Invariantti 3 on vaarassa rikkoutua myös, kun reitin kärki lakkaa olemasta reitin kärki siksi, että reitin jonkin myöhäisemmän solmun `etaisyys` muuttuu minimietäisyydeksi. Jos niin käy, niin se tapahtuu rivillä 70. Rivillä 72 tämä solmu laitetaan kekoon minimietäisyytensä kanssa. Silloin siitä tulee uusi reitin kärki.

Invariantin 3 perustelu päättyy tähän.

63. Esitä esimerkki, jossa minimipituisen reitin solmun `etaisyys` muuttuu minimietäisyydeksi, mutta solmusta ei tule reitin kärkeä.

64. Voiko solmuun olla samanaikaisesti sekä minimipituinen reitti, jolla on kärki, että minimipituinen reitti, jolla ei ole kärkeä?

Osoitamme seuraavaksi, että aina kun solmu tulee ulos keosta, on sen `etaisyys` minimietäisyys. Jos `etaisyys` ei ole minimietäisyys, niin invariantin 3 vuoksi solmuun vie minimipituinen reitti, jolla on kärki. Kärki on määritelmänsä mukaan keossa minimietäisyytensä kanssa. Solmun itsensä minimietäisyys on vähintään yhtäsuuri, koska kaarten pituudet eivät ole negatiivisia. Solmun `etaisyys` on vielä suurempi, koska tarkastelemme tapausta, jossa se ei ole minimietäisyys. Jokainen etäisyys, jonka kanssa solmu on keossa, on ainakin yhtäsuuri. Solmut tulevat keosta niiden keossa olevan etäisyyden mukaisessa järjestyksessä. Siksi reitin kärki tulee keosta ennen solmua itseään. Niinpä solmu ei voi olla ulostulovuorossa, ellei sen `etaisyys` ole minimietäisyys.

Tästä seuraa välittömästi, että jos `etsi_reitti` lopettaa rivillä 63, niin löydetty reitti on minimipituinen. Toisaalta jos `etsi_reitti` lopettaa rivillä 76, niin rivin 60 vuoksi keko on tyhjä. Silloin invariantin 3 vuoksi jokaisen solmun `etaisyys` on minimietäisyys. Koska ei lopetettu rivillä 63, ei `maali` käynyt keossa, joten sen `etaisyys` on yhä ääretön. Niinpä paluuarvo `false` on oikea.

Seuraavaksi osoitamme, että `etsi_reitti` lopettaa ja laskemme ajan kulutukselle ylärajan. Rivien 66, ..., 74 silmukka ei tarvitse muuta kuin toteamuksen, että se on aito `for`-silmukka (sivu 20). Jäljellä on rivien 60, ..., 75 `while`-silmukka.

Osoitimme edellä, että aina kun solmu tulee ulos keosta, on sen `etaisyys` minimietäisyys. Siksi rivin 69 ehto estää lisäämästä solmua uudelleen kekoon sen jälkeen kun se on ensimmäisen kerran tullut sieltä. Solmu voi edelleen olla keossa muina kopioina, mutta niissä on suurempi etäisyys, koska kekoon lisätään uusia kopioita vain kun `etaisyys` pienenee, ja ne tulevat ulos jokin pienin ensin. Niinpä muut kopiot hylätään rivillä 64. Näin ollen rivi 65 suoritetaan kullekin solmulle enintään kerran. Siitä seuraa, että rivit 67, ..., 73 suoritetaan kullekin kaarelle enintään kerran. Koska keosta voidaan poistaa korkeintaan yhtä monta kertaa kuin siihen lisätään, suoritetaan rivit 61, ..., 64 korkeintaan kerran kutakin kaarta kohti ja lisäksi korkeintaan kerran lähtösolmulle. Siksi `while`-silmukka lopettaa.

Kuvan 29 muut toiminnot vievät $\Theta(1)$ aikaa, paitsi `lisaa_kekoon` ja `poista_keosta`, jotka vievät tasatusti $O(\log k)$ aikaa, missä k on keon koko. Edellä olevasta seuraa, että keon koko voi olla enintään kaarten määrä. Olemme merkinneet kaarten määrää m:llä. Niinpä `etsi_reitti` vie $O(m \log m)$ aikaa.

65. Miksi ei välttämättä ole totta, että rivit 61, ..., 64 suoritetaan lähtösolmulle kerran?

66. Kuinka paljon O- tai Θ-merkinnällä ilmaistuna ohjelman tähän mennessä esitetyt osat tarvitsevat yhteensä muistia?

67. Miksi vastauksessa 66 tarvitaan ”plus yksi”, vaikka `lahto` ei voi olla keossa yhtäaikaa minkään muun solmun kanssa?

68. Anna syöte, jolla `etsi_reitti` vie $\Theta(m \log m)$ aikaa.

Indeksointien laillisuus on sen varassa, että `lahto` ja kunkin kaaren `minne` ovat välillä 0, ..., `solmut.size()` $-$ 1, ja kukin `kaarten_loppu` on välillä 0, ..., `kaaret.size()`. Ne on varmistettu `lue_syote`:ssä. Rivi 60 takaa, että `poista_keosta` ei yritä poistaa tyhjästä keosta. Jollei syöte ole niin suuri, että muisti tai `unsigned`-tyypin lukualue loppuu kesken, niin muuta laitonta ei `etsi_reitti`:ssä voi tapahtua.

Koska liukuluvuilla voi tapahtua pyöristysvirheitä, voi `etsi_reitti` valita lyhimmän reitin sijaan jonkin toisen, hiukan pitemmän reitin. Tyypillä `float` ero voi olla 1000 km matkalla suuruusluokkaa 1 mm ja `double`:lla vielä paljon vähemmän, joten sillä on normaalitapauksessa hyvin vähän merkitystä. Rivin 68 laskutoimitus voi tuottaa äärettömän silloin, kun tarkka lopputulos olisi äärellinen mutta hyvin suuri. Jos pituudet ilmaistaan millimetreissä, niin koko näkyvään maailmankaikkeuteen ei mahdu niin pitkää suoraa matkaa, että tästä tulisi ongelma.

Pituuksille ja etäisyyksille voi käyttää myös kokonaislukutyyppiä. Silloin pyöristysvirheitä ei tapahdu, mutta rivillä 68 voi tapahtua ylivuoto, jolloin `etaisyys`

```
78  void tulosta_reitti( unsigned solmu ){
79    if( solmu == lahto ){ std::cout << lahto; return; }
80    unsigned edell = solmut[ solmu ].reitti;
81    tulosta_reitti( edell );
82    std::cout << " "
83      << solmut[ solmu ].etaisyys - solmut[ edell ].etaisyys
84      << "km " << solmu;
85  }
```

Kuva 30: Lyhimmän reitin tulostaminen

saa `int`-tyypillä negatiivisen ja `unsigned`-tyypillä muuten vaan liian pienen arvon. Se sekoittaa algoritmin toiminnan.

4.5 Ohjelman loppuosat

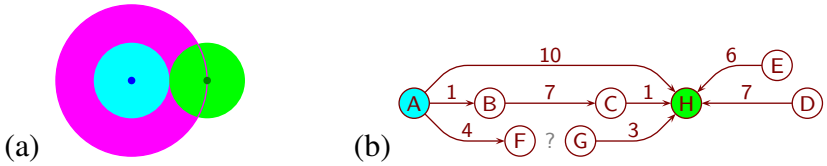
Lyhimmän reitin tulostamista hankaloittaa se, että `reitti`-kentät esittävät sen lopusta alkuun. Sen kääntämiseksi etuperin on monta keinoa. Kuvassa 30 on algoritmi, joka ensin tulostaa reitin alkuosan kutsumalla itseään rekursiivisesti ja sitten tulostaa yhden askeleen. Sen pysähtyminen ja indeksointien laillisuus seuraavat siitä, että `etsi_reitti` on tallentanut `reitti`-kenttiin oikeat tiedot. Se kuluttaa $O(n)$ aikaa, koska kukin solmu voi olla reitissä enintään kerran. Rekursion vuoksi se kuluttaa $O(n)$ lisämuistia. Tehtävässä 108 selvitetään, miten lisämuistin tarve voidaan pudottaa $\Theta(1)$:ksi ajan käytön säilyessä $O(n)$:nä. Tällä tehostuksella on käytännön merkitystä vain poikkeustapauksissa.

Enää ei tarvita muuta kuin pääohjelma. C++:ssa pääohjelma voi ilmoittaa lopettaneensa epätavallisesti palauttamalla jonkin muun arvon kuin 0. Tämä ohjelma ilmoittaa sillä tavalla syötteen lukemisen epäonnistumisen ja sen, että reittiä ei löytynyt. Jos reitti löytyi, niin ohjelma tulostaa ensin sen pituuden, sitten reitin ja lopuksi rivinsiirron.

```
86  int main(){
87    if( !lue_syote() ){ return 1; }
88    if( !etsi_reitti() ){ std::cout << "Ei reittiä\n"; return 2; }
89    std::cout << solmut[ maali ].etaisyys << "km: ";
90    tulosta_reitti( maali ); std::cout << "\n";
91  }
```

Pääohjelma ei tuo olennaista uutta toimintojen laillisuuden tarkasteluun eikä ajan ja muistin kulutukseen. Laskemalla yhteen ohjelman eri osien ajan kulutukset saadaan $O(n+m\log m)$, missä n on solmujen ja m on kaarten määrä. Muistin kulutus on $\Theta(n+m)$.

Dijkstran algoritmi tutkii kartasta vain sen osan, joka koostuu lähdöstä ja sen ympäristöstä joka suuntaan maalin etäisyydelle saakka (tarkemmin sanoen osaan



Kuva 31: (a) yksi- ja kaksisuuntaisen etsinnän vertailu; (b) lopettamisehto ei ole itsestään selvä

niistä solmuista saakka, joihin on kaari, jonka häntäpää on enintään maalin etäisyydellä lähdöstä). Jos kartta on massamuistissa ja sitä haetaan sieltä vain tarpeen mukaan, niin suoritusajan n ja m eivät ole koko kartan vaan tutkitun osan solmujen ja kaarten määrät. Jos kartta kattaa koko Suomen mutta etsitään reittiä yhden kaupunkialueen sisällä kotoa yliopistolle, niin tutkitun osan n ja m ovat huomattavasti pienemmät kuin koko kartan n ja m. Tästä voi seurata merkittävä ajan säästö.

69. Oletetaan, että samasta kartasta kysytään useita reittejä. Miten edellä esitetyt tietorakenteet pitää alustaa kyselyjen välillä?

70. Jos kartta kattaa koko Suomen, mutta suurin osa reittien etsinnöistä tapahtuu pienellä alueella, niin jokaisen solmun alustamiseen etsintöjen välillä kuluu paljon enemmän aikaa kuin itse etsintöihin. Miten kahden etsinnän välistä alustamista voidaan tehostaa niin että siihen kuluu vain jommankumman etsintäalueen kokoon verrannollinen aika, eikä koko kartan kokoon verrannollinen aika?

71. Onko tarpeen ottaa `maali` huomioon, kun `max_muu` alustetaan rivillä 18?

72. Osoita, että rivin 63 testin muuttaminen muotoon `etaisyys == solmut[ maali ].etaisyys` ei rikkoisi algoritmia!

73. Nopeuttaisiko edellisen tehtävän muutos algoritmia niin paljon, että se kannattaisi tehdä?

4.6 Etsintä yhtäaikaa kahteen suuntaan

Yksi ehdotus etsinnän tehostamiseksi on etsiä reittiä samanaikaisesti etuperin lähdöstä alkaen ja takaperin maalista alkaen. Kun etsinnät kohtaavat (tietyssä mielessä), on jokin lyhin reitti löytynyt. Kuvassa 31 (a) vaaleanvioletti alue edustaa yksisuuntaisen etsinnän tutkimaa osaa kartasta, vaaleansininen kaksisuuntaisen etsinnän lähdöstä alkavan etsinnän tutkimaa osaa, ja vaaleanvihreä kaksisuuntaisen etsinnän maalista alkavan etsinnän tutkimaa osaa. Kuvan perusteella voidaan arvioida, että jos solmuja ja kaaria on suurin piirtein tasaisesti kartan alueella, niin kaksisuuntaisuus vähentää tutkittavien solmujen ja kaarten määrän suunnilleen puoleen.

Kuva 31 (b) havainnollistaa, että etsintää ei voida lopettaa jo silloin, kun molemmat etsinnät ovat kohdanneet saman solmun eikä edes silloin, kun toinen niistä

on saanut sen käsiteltyä. Kun kumpikin etsintä on käsitellyt ensimmäisen solmunsa, on H etuperin etsinnän keossa etäisyydellä 10 ja käsitelty takaperin etsinnässä. Silti niiden välille löydetty reitti ei ole lyhin, vaan lyhyempi löytyy sitten kun B käsitellään etuperin etsinnässä tai C käsitellään takaperin etsinnässä. Toisaalta jatkaminen niin kauan että sama solmu on tullut ulos kummastakin keosta teettäisi tarpeetonta työtä. Esimerkin tapauksessa (ilman kaarta F:stä G:hen) se tapahtuisi vasta kun etsinnät ovat saavuttaneet etäisyyden 8, joten E ja D olisi käsiteltävä takaperin etsinnässä.

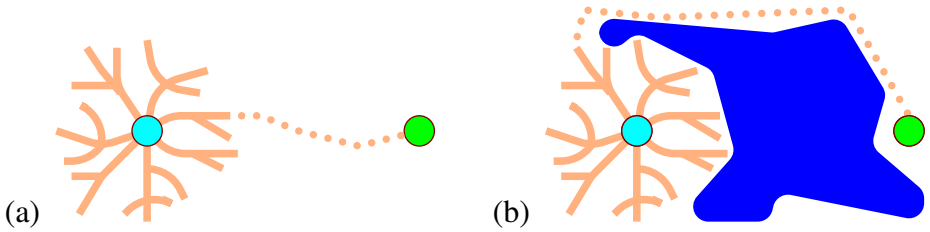
Oikea hetki lopettaa on kun lyhin löydetyn reitin pituus on enintään luku joka saadaan katsomalla kummastakin keosta ylin etäisyys ja laskemalla ne yhteen. Jos jokin kaari on tutkimatta kun tämä toteutuu, niin sen häntäpää on vähintään yhtä kaukana lähdöstä kuin etuperin etsinnän keon ylin etäisyys, ja sen kärkipäästä on ainakin yhtä pitkä matka maaliin kuin takaperin etsinnän keon ylin etäisyys, joten se ei voi olla osa lyhyempää reittiä. Ennen kuin tämä toteutuu, on mahdollista, että keoissa ylimpien kautta (tai jotain muuta kautta) kulkee vielä lyhyempi reitti. Esimerkissä on tämä tilanne, kun F ja G ovat kumpikin kekonsa ylimpänä.

4.7 A* reitinetsintäalgoritmi

Dijkstran algoritmi on hyvä reitinetsintäalgoritmi kun kaarten pituuksista ei algoritmin valintavaiheessa tiedetä muuta kuin että ne ovat positiivisia lukuja tai nollia. Jos muutakin tietoa on käytettävissä, niin sitä voidaan yrittää hyödyntää algoritmin tehostamiseksi. Tällainen muu tieto voi olla esimerkiksi, että reittiä etsitään maantiekartasta, joka noudattaa todellisen maailman lainalaisuuksia eikä niin ollen sisällä tieteiselokuvien madonreikiä, joiden kautta voi oikaista kauas.

Kuvassa 32 (a) on havainnollistettu ilmiötä, joka huonontaa Dijkstran algoritmin tehoa todellisten maantiekarttojen tapauksessa. Dijkstran algoritmi ei hyödynnä sitä, että kaikki samanpituiset jo löydetyt minimipituiset reitit eivät ole samanarvoisia, vaan toiset niistä johtavat maalia kohti ja toiset poispäin. Se ottaa huomioon mahdollisuuden, että vasemmalla saattaa olla madonreikä suoraan maaliin. Jos luotetaan siihen, että madonreikiä ei ole, niin algoritmia voi nopeuttaa. Maantiekartan tapauksessa siihen voi luottaa, mutta tietokonepelien tapauksessa ei voi.

A* on muuten samanlainen kuin Dijkstran algoritmi, mutta se ei käsittele solmuja niiden lähdöstä etäisyyden mukaisessa järjestyksessä, vaan ottaa huomioon myös arvion jäljellä olevasta matkasta. Arvioiden muodostaminen riippuu etsintätehtävän luonteesta. Maantiekartassa arviona voidaan käyttää etäisyyttä käsiteltävästä solmusta maaliin linnuntietä pitkin. Kahden paikan välinen linnuntie-etäisyys on niiden välisen suoran viivan pituus. Se saadaan Pythagoraan lauseesta. Jos paikkojen koordinaatit ovat (x_1, y_1) ja (x_2, x_2), niin se on $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.



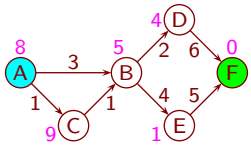
Kuva 32: (a) Dijkstran algoritmi ei etene kohti maalia vaan tasapuolisesti kaikkiin suuntiin; (b) kohti maalia ei välttämättä ole paras jos pitää kiertää este

Solmun ***kokonaismatkan arvio*** on lähdöstä solmuun jo löydetyn reitin pituuden ja solmusta linnuntietä maaliin etäisyyden summa. Seuraavaksi käsiteltäväksi solmuksi valitaan aina sellainen, jolle kokonaismatkan arvio on mahdollisimman pieni. Kuvan 32 (a) tapauksessa se saa algoritmin suosimaan oikeanpuoleisia solmuja. Tyypillisesti se nopeuttaa algoritmia, mutta se voi myös hidastaa. Se hidastaa esimerkiksi kun lähdön ja maalin välissä on järvi joka joudutaan kiertämään, kuten kuvassa 32 (b).

Jos madonreikiä ei ole, niin etäisyys linnuntietä on lyhin mahdollinen matka kahden paikan välillä. Siitä seuraa, että loppumatkan arvio maalissa on nolla. Siitä seuraa myös, että jos solmusta on kaari toiseen, niin ensimmäisen solmun linnuntie-etäisyys maaliin on enintään yhtä suuri kuin kaaren pituus plus jälkimmäisen solmun linnuntie-etäisyys maaliin. Siis aina kun kuljetaan kaari, niin loppumatkan arvion muutoksen ja kaaren pituuden summa on nolla tai positiivinen. Arvio, jolla on nämä kaksi ominaisuutta, tunnetaan adjektiivilla ***consistent*** tai ***monotone***.

Tämä ominaisuus on tärkeä. Siitä seuraa, että A*:n suoritus linnuntie-etäisyydellä voidaan tulkita Dijkstran algoritmin suoritukseksi siten, että kaaren pituutena on kaaren todellisen pituuden ja linnuntie-etäisyyden muutoksen summa. Siitä puolestaan seuraa, että A*:ä linnuntie-etäisyydellä ei tarvitse erikseen todistaa oikeaksi eikä analysoida sen suoritusaikaa, vaan molemmat saadaan ilmaiseksi vastaavista tuloksista Dijkstran algoritmille. Tällä tulkinnalla Dijkstran algoritmi etsii reittiä, jonka pituus miinus lähdön linnuntie-etäisyys maaliin on mahdollisimman pieni, mutta se on sama kuin oikeasti mahdollisimman lyhyt reitti. Kaaren todellisen pituuden ja linnuntie-etäisyyden muutoksen summa piti osoittaa aina ei-negatiiviseksi, koska Dijkstran algoritmi edellyttää, että kaarten pituudet eivät ole negatiivisia.

Kaarten pituuksia ei oikeasti kannata muuttaa, koska siitä aiheutuisi tarpeetonta työtä. Todellisen pituuden ja linnuntie-etäisyyden muutoksen summa riippuu maalista, joten pituudet jouduttaisiin muuttamaan aina kun maali vaihtuu. Tällöin menetettäisiin se etu, että jos lähtö ja maali ovat lähekkäin, niin jonkin lyhimmän reitin löytämiseksi riittää käsitellä pientä osaa graafista. Sen sijaan kannattaa



Kuva 33: ”Admissible”-arvioilla A* voi joutua käsittelemään solmuja uudelleen

lisätä solmun tietoihin sen linnuntie-etäisyys maaliin, ja laskea se samalla kun solmu löydetään kuvan 29 rivillä 59 tai 70. Solmussa on siis tallennettuna erikseen lähdöstä solmuun vievän jo löydetyn reitin pituus ja erikseen etäisyys maaliin linnuntietä. Kekoon pannaan näiden summa.

Matemaattisesti olisi yhtäpitävää tallentaa solmuun linnuntie-etäisyys ja summa. Ohjelmoinnin näkökulmasta siihen kuitenkin liittyy se ongelma, että rivillä 68 pitäisi sekä vähentää kaaren häntäpään linnuntie-etäisyys että lisätä kaaren kärkipään linnuntie-etäisyys. Reitin seuraavan kaaren kohdalla vähennettäisiin se mikä edellisen kaaren kohdalla lisättiin, ja niin edelleen koko reitin matkalta. Liukuluvuilla laskeminen ei ole täysin tarkkaa, joten toistuva saman luvun lisääminen ja vähentäminen voi turhaan aiheuttaa kasautuvaa pyöristysvirhettä. Tässä sovelluksessa se tuskin on ongelma, koska pyöristysvirheitä ehtii kertyä vähän ja nykyaikaisten tietokoneiden liukulukuyksiköt ovat laadukkaita. On kuitenkin hyvä ottaa tavaksi miettiä pyöristysvirheiden riskiä aina kun käyttää liukulukuja.

A* sallii myös muunlaisia arvioita loppumatkalle. Silloin toteutus Dijkstran algoritmina ei välttämättä ole pätevä.

Adjektiivilla ***admissible*** tunnettu ehto vaatii, että jäljellä olevan matkan arvio on aina enintään todellinen lyhin etäisyys maaliin. Silloinkin A* löytää jonkin lyhimmän reitin, mutta samoja solmuja voidaan joutua käsittelemään monta kertaa. Kuvassa 33 on siitä esimerkki. Solmujen arvioidut etäisyydet maaliin on merkitty solmujen viereen. Kun solmu A on käsitelty, odottaa B käsittelyvuoroa kokonaismatkan arviolla $(3)+5=8$ ja C kokonaismatkan arviolla $(1)+9=10$ (sulkeissa on reitin pituus ja ilman sulkeita loppumatkan arvio). Seuraavaksi käsitellään B, jolloin jonottamaan tulevat D arviolla $(3+2)+4=9$ ja E arviolla $(3+4)+1=8$. Kun ne on käsitelty, on maali löytynyt matkoina 12 ja 11. C tulee käsittelyyn vasta nyt. Silloin B saa uuden, aikaisempaa pienemmän arvion $(1+1)+5=7$. Nyt kaikki kuvassa B:n oikealla puolella olevat solmut on käsiteltävä uudelleen.

4.8 Floyd–Warshall reitinetsintäalgoritmi

Reitinetsintäalgoritmit eroavat toisistaan toimintaperiaatteiltaan, suorituskyvyiltään ja syötteeltä edellyttämiltään ominaisuuksilta. Esimerkiksi A* vaatii ”admissible”-ominaisuuden ollakseen nopea. Osa algoritmeista sallii ja osa ei salli nega-

tiivisia kaarten pituuksia, ja osa ensin mainituista sallii ja osa ei salli silmukan kokonaispituuden olevan negatiivinen. Algoritmit eroavat toisistaan myös lähtöjen ja maalien määriltään. Dijkstran algoritmia käytettiin kuvassa 29 etsimään jokin lyhin reitti lähdöstä maaliin, mutta poistamalla rivi 63 se saadaan etsimään jotkin lyhimmät reitit jokaiseen solmuun. Sen sijaan A*:ä ei saa yhtä näppärästi muutettua etsimään joitakin lyhimpiä reittejä jokaiseen solmuun, koska siinä käytettävä arvio loppumatkan pituudesta riippuu siitä, mikä solmu on maali.

Tietenkin minkä tahansa algoritmin, joka löytää jonkin lyhimmän reitin yhteen solmuun, saa löytämään jotkin lyhimmät reitit jokaiseen solmuun kutsumalla sitä erikseen jokaiselle niistä. Mutta sillä tavalla suoritusaika huononee olennaisesti. Pahimmillaan se tulee kerrotuksi solmujen määrällä. Dijkstran algoritmi löytää jotkin lyhimmät reitit jokaiseen solmuun käytännöllisesti katsoen samassa ajassa kuin se löytää jonkin lyhimmän reitin kauimmaiseen solmuun. Se on tyypillisesti olennaisesti parempi kuin olisi kutsua A* jokaiselle solmulle erikseen.

74. Miten algoritmin, joka löytää jotkin lyhimmät reitit yhdestä lähdöstä moneen maaliin, saa etsimään jotkin lyhimmät reitit monesta lähdöstä yhteen maaliin?

Tehtävän luonne muuttuu olennaisesti, jos on löydettävä jotkin lyhimmät reitit jokaisesta solmusta jokaiseen solmuun, tai edes niiden pituudet. Kuten kohta näemme, siitä seuraa, että kaaret kannattaa esittää eri tavalla kuin tähän asti tässä luvussa. Tähänastinen tapa on nimeltään ***kytkentälista*** (***adjacency list***), tarkemmin sanottuna sen alalaji ***forward star***. Kytkentälistassa jokaiseen solmuun on liitetty luettelo siitä lähtevistä kaarista. Alalajissa ”forward star” nämä luettelot ovat peräkkäin samassa taulukossa. Kytkentälistan muistin tarve on $\Theta(m)$, missä m on kaarten määrä.

Jos samojen solmujen välillä on monta kaarta samaan suuntaan, niin reitin etsinnän kannalta on eduksi poistaa etukäteen muut paitsi jokin lyhin. Siksi tavallisesti kaarten määrä on enintään solmujen määrän n neliö. Jos lisäksi ei sallita kaaria solmuista itseensä, niin kaaria on enintään $n^2 - n$. Kytkentälista on tyypillisesti edullinen silloin, kun solmusta keskimäärin lähtevien kaarten määrä on paljon pienempi kuin solmujen määrä. Maantiekartassa saattaa olla tuhansittain risteyksiä eli solmuja, mutta risteyksestä eri suuntiin lähtevien teiden määrä on yleensä kolme tai neljä, ja hyvin harvoin jos koskaan yli kymmenen. Silloin m on paljon vähemmän kuin n^2, joten muistia tarvitaan paljon vähemmän kuin $\Theta(n^2)$. Kytkentälistan etuna on myös, että solmusta lähtevien kaarten selaaminen vie aikaa vain verrannollisesti niiden määrään.

Jos on löydettävä reittien pituudet jokaisesta solmusta jokaiseen solmuun, niin vastaus koostuu n^2 luvusta. Niinpä jo pelkästään vastauksen esittäminen vaatii $\Theta(n^2)$ muistia, ja kytkentälistan kyky säästää muistia menettää merkityksensä. Silloin kaaret kannattaa esittää ***kytkentämatriisina*** (***adjacency matrix***). Se tarkoittaa kaksiulotteista taulukkoa, jonka rivin i ja sarakkeen j leikkauskohdassa on sol-

```
   FLOYDWARSHALL(&M)
1  for k := 0 to n − 1 do
2    for i := 0 to n − 1 do
3      for j := 0 to n − 1 do
4        d := M[i,k] + M[k, j]
5        if M[i, j] > d then M[i, j] := d
```

Kuva 34: Floyd–Warshall-algoritmi

musta i solmuun j vievän kaaren pituus. Jos solmusta i ei ole kaarta solmuun j, niin leikkauskohdassa on ääretön.

Floyd–Warshall-algoritmi (kuva 34) aloittaa tällaisella matriisilla. Kaarten pituudet saavat olla negatiivisia, mutta kaarista ei saa muodostua silmukoita, joiden pituudet ovat negatiivisia. Kukin $M[i,i]$ voidaan alustaa nollaksi merkiksi siitä, että solmusta pääsee itseensä nollalla askeleella, tai sen mukaan onko solmusta i kaarta itseensä ja jos on, niin mikä on sen pituus. Jälkimmäisessä tapauksessa algoritmi selvittää kullekin solmulle mahdollisimman lyhyen sellaisen solmusta itseensä vievän reitin pituuden, joka kulkee ainakin yhtä kaarta pitkin. Voimme rajoittua reitteihin joissa mikään solmu ei toistu (muuten kuin että lähtö saa olla maali), koska solmun toistuminen ei voi lyhentää reittiä, koska silmukan pituus ei voi olla negatiivinen.

Ensiksi algoritmi laskee jokaisesta solmusta jokaiseen solmuun jonkin sellaisen lyhimmän reitin pituuden, joka joko koostuu yhdestä kaaresta tai kulkee ensin lähdöstä (joka tässä tapauksessa ei ole 0) yhtä kaarta pitkin solmuun 0 ja jatkaa sieltä yhtä kaarta pitkin maaliin (joka tässä tapauksessa ei ole 0). Se tekee sen vertaamalla näiden kahden vaihtoehdon pituuksia. Solmu 0 ei voi tulla reittiin sekä lähtönä tai maalina että keskimmäisenä solmuna, koska sama reitti ilman silmukkaa solmusta 0 itseensä on ainakin yhtä lyhyt ja, koska siinä on täsmälleen yksi kaari, on sen pituus jo M:ssä. Toisin sanoen, vaikka $M[i,0]$ tai $M[0, j]$ olisi jo päivitetty kierroksella $k = 0$ kun sitä käytetään rivillä 4, niin se ei haittaa, koska päivitys ei muuttanut sen arvoa.

Sitten algoritmi laskee minimipituudet, jos matkan varrella saa käyttää vain solmuja 0 ja 1, kumpaakin enintään kerran, eivätkä lähtö ja maali saa esiintyä matkan varrella. Algoritmilla on jo minimipituudet jos matkan varrella saa käyttää vain solmua 0 enintään kerran. Se vertaa tällaista minimipituutta lähdöstä maaliin siihen, että mennään tällaista reittiä solmuun 1 ja jatketaan sieltä tällaista reittiä maaliin. Solmu 0 tulee solmun 1 kautta kulkevaan reittiin enintään kerran, koska jos se olisi sekä osuudessa ennen solmua 1 että solmun 1 jälkeen, niin ainakin yhtä lyhyt reitti saataisiin jättämällä solmun 0 ensimmäisen ja toisen esiintymän välinen osuus pois. Sellainen reitti olisi löydetty edellisellä kierroksella. Myös solmu 1 tulee reittiin enintään kerran, koska reitti ei lyhenisi siitä, että siihen lisättäisiin

kaari solmusta 1 itseensä.

Vaiheessa k algoritmi laskee jokaisesta solmusta jokaiseen solmuun minimipituuden, jos reitillä voi esiintyä lähdön ja maalin lisäksi vain solmuja $0, \ldots, k$, eikä mikään solmu toistu paitsi että lähtö saa olla maali. Jos sellainen reitti ei kulje solmun k kautta, niin se käyttää lähdön ja maalin lisäksi vain solmuja $0, \ldots, k-1$ eikä sisällä kiellettyä toistoa, joten algoritmilla on jo sen minimipituus.

Muussa tapauksessa reitti kulkee ensin lähdöstä solmuun k käyttäen vain solmuja $0, \ldots, k-1$, ja jatkaa maaliin käyttäen vain solmuja $0, \ldots, k-1$. Algoritmilla on tiedossa kummankin osan minimipituus. Nytkään reitti ei käytä mitään solmua sekä ennen solmua k että solmun k jälkeen (paitsi lähtö saa olla maali), koska se olisi ainakin yhtä pitkä kuin sama reitti ilman solmun ja sen toiston välistä osuutta, ja sellainen reitti on löydetty jo aikaisemmilla k:n arvoilla. Nytkään lähtö ja maali eivät voi olla k, koska osuus k:sta itseensä ei lyhennä matkaa, joten rivin 5 ehto ei voi toteutua. Toisin sanoen, vaikka $M[i,k]$ tai $M[k,j]$ olisi jo päivitetty kierroksella k kun sitä käytetään rivillä 4, niin se ei haittaa, koska päivitys ei muuttanut sen arvoa.

Löydetyn reitin tulostamista varten täytyy tavalla tai toisella tallentaa tieto siitä, mikä vaihtoehto solmujen i ja j väliseksi reitiksi jäi voimaan, kun rivi 5 toistettiin k:n eri arvoilla. Yksi mahdollisuus olisi tallentaa k:n arvo rivillä 5 muuttujaan $R[i,j]$, joka on alustettu arvoon, joka ei voi olla minkään solmun numero. Sekin toimii, mutta näppärämpää on tallentaa i:n jälkeen seuraavan solmun numero reitillä i:stä k:hon. Aluksi asetetaan $R[i,j] := j$, jos i:stä on kaari j:hin. Ei ole väliä miten $R[i,j]$ alustetaan, jos i:stä ei ole kaarta j:hin. Kun sijoitetaan $M[i,j] := d$ rivillä 5, niin samalla sijoitetaan $R[i,j] := R[i,k]$.

75. Kirjoita algoritmi, joka tulostaa löydetyn reitin solmujen numerot ja kaarten pituudet tyyliin `5 7km 2 3km 4`!

76. Mikä menee rikki Floyd–Warshall-algoritmin oikeellisuuden perustelussa, jos $>$:n tilalle rivillä 5 kirjoitetaan $\geq$? Anna esimerkki virhetoiminnosta, joka siitä saattaa aiheutua!

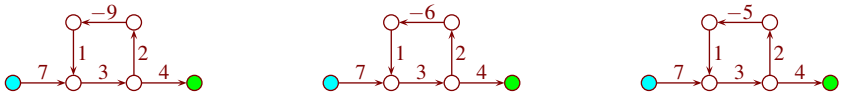
Floyd–Warshall-algoritmin suoritusaika on $\Theta(n^3)$ sekä ilman R:n ylläpitoa että kun R:n ylläpito on mukana ja lopuksi tulostetaan kaikki reitit. Reittejä on enintään n^2 kappaletta, ja vastaus 75 kuluttaa $O(n)$ aikaa. Koska tavallisesti $m \leq n^2$, on $\Theta(n^3)$ tavallisesti selvästi huonompi kuin Dijkstran algoritmin $O(n + m \log m)$ reiteille yhdestä solmusta jokaiseen solmuun. Toisaalta Dijkstran algoritmin ajan kulutus reiteille jokaisesta solmusta jokaiseen solmuun on $O(n^2 + nm \log m)$. Jos $m \approx n^2$, niin se sievenee muotoon $O(n^3 \log n)$, joka on hieman huonompi kuin $\Theta(n^3)$. Ero ei ole O- ja Θ-merkintöjen tasolla iso, mutta käytännön eroa kasvattaa se, että Floyd–Warshall on yksinkertainen, joten sillä on pieni vakiokerroin.

Floyd–Warshall-algoritmin muistin kulutus on $\Theta(n^2)$ riippumatta siitä, ovatko R ja reittien tulostus mukana.

4.9 Bellman–Ford reitinetsintäalgoritmi

Bellman–Ford-algoritmi on yksinkertainen hitaanpuoleinen algoritmi, joka ilmoittaa, onko graafin lähdöstä saavutettavassa osassa silmukkaa, jonka kaarten pituuksien summa on negatiivinen. Jollei sellaista silmukkaa ole, se löytää lähdöstä kuhunkin solmuun jonkin lyhimmän reitin.

Alla olevassa kuvassa on lähdöstä maaliin vievällä reitillä silmukka, jonka kaarten pituuksien summa on vasemmalla negatiivinen, keskellä nolla ja oikealla positiivinen. Jos kuvassa vasemmalla silmukka kierretään nolla, yksi, kaksi, ... kertaa, niin matkan pituus lähdöstä maaliin on 14, 11, 8, ... Jos silmukka kierretään viisi kertaa, on reitin pituus -1. Mitä useammin silmukka kierretään, sitä enemmän negatiivinen on reitin pituus. Tämä havainnollistaa sitä, että jos reitillä lähdöstä maaliin on silmukka, jonka kaarten pituuksien summa on negatiivinen, niin mahdollisimman lyhyttä reittiä ei ole olemassa.



Mahdollisimman lyhyt reitti ei voi sisältää sellaisen silmukan kierroksia, jonka kaarten pituuksien summa on positiivinen, sillä sellainen reitti lyhenisi jättämällä silmukan kierrokset pois. Tästä on esimerkki kuvassa oikealla. Mahdollisimman lyhyt reitti voi sisältää silmukan, jonka kaarten pituuksien summa on nolla. Sellaisen reitin pituus ei muutu vaikka silmukka kierrettäisiin nolla tai jokin muu määrä kertoja. Siksi mahdollisimman lyhyen reitin etsintä voidaan rajoittaa reitteihin, joissa mikään solmu ei toistu, paitsi että lähtö saa olla sama solmu kuin maali. Teimme tämän rajoituksen edellistenkin algoritmien kohdalla.

Algoritmi on näytetty kuvassa 35. Merkitsemme solmujen määrää n:llä ja kaarten määrää m:llä. Solmut ovat taulukossa $S[0 \ldots n-1]$ siten, että $S[i].e$ kertoo solmun etäisyyden lähdöstä jotakin lyhintä jo löydettyä reittiä pitkin, ja $S[i].r$ kertoo edellisen solmun tällä reitillä. Lähdön etäisyydeksi alustetaan 0 ja jokaisen muun solmun etäisyydeksi ääretön. Ei ole väliä, miten $.r$-kentät alustetaan. Kaaret ovat taulukossa $K[0 \ldots m-1]$. Kaari j alkaa solmusta $K[j].h$ ja päättyy solmuun $K[j].k$. Kenttien nimet ”h” ja ”k” voidaan lukea ”häntä” ja ”kärki”. Kaaren pituus on $K[j].p$.

Algoritmi tekee enintään n kierrosta. Jokaisella kierroksella tutkitaan jokaisesta kaaresta, saataisiinko sen kautta kaaren kärkipään solmuun aikaisemmin löydettyjä lyhyempi reitti. Jos saataisiin, niin solmun tiedot päivitetään tämän reitin mukaisiksi rivillä 9. Algoritmi lopettaa kun minkään solmun tiedot eivät muutu

```
1   for i := 0 to n − 1 do S[i].e := ääretön
2   S[lähtö].e := 0
3   kierros := 1; muuttui := true
4   while kierros ≤ n && muuttui
5     kierros := kierros + 1; muuttui := false
6     for j := 0 to m − 1 do
7       d := S[K[j].h].e + K[j].p; i := K[j].k
8       if d < S[i].e then
9         S[i].e := d; S[i].r := K[j].h
10        muuttui := true
```

Kuva 35: Bellman–Ford-algoritmi

kierroksen aikana, mutta viimeistään n kierroksen jälkeen. Algoritmin ajan kulutus on $\Theta(nm)$.

Algoritmin solmuihin merkitsemät tiedot ovat aina jonkin todellisen reitin mukaisia. Osoitamme seuraavaksi, että algoritmi löytää kullekin solmulle jonkin lyhimmän reitin, jossa mikään solmu ei toistu, mikäli sellainen on olemassa.

Tarkastelkaamme jotakin sellaista reittiä. Koska ensimmäisellä kierroksella käsitellään kaikki kaaret, sillä käsitellään myös tämän reitin ensimmäinen kaari. Sen jälkeen reitin lähtöä seuraavan solmun tiedot ovat tämän kaaren mukaiset. Toisella kierroksella käsitellään reittimme toinen kaari. Nyt lähdön jälkeen toisen solmun tiedot ovat joko tämän kaaren mukaiset, tai jonkin muun yhtä pitkän lähdöstä tähän solmuun vievän reitin mukaiset. Kolmannen kierroksen jälkeen reittimme lähdön jälkeen kolmannen solmun $.e$ on reittimme mukainen, ja $.r$ on joko reittimme tai jonkin muun yhtä pitkän vaihtoehdon mukainen. Koska mikään solmu ei toistu reitillä, on siinä lähtö mukaan lukien korkeintaan n solmua, joten viimeistään $n-1$ kierroksen jälkeen on koko reitti käsitelty. Silloin maalin $.e$ on jonkin minimipituisen reitin mukainen, jonka edellinen solmu on kerrottu maalin $.r$-kentässä.

77. Anna esimerkki, jossa kahdesta kaaresta koostuva reitti solmusta A solmun B kautta solmuun C on mahdollisimman lyhyt, mutta silti kahden kierroksen jälkeen solmun C tiedot eivät ole tämän reitin mukaiset!

Jos kierroksen aikana ei suoriteta rivejä 9 ja 10, niin minkään solmun $.e$ ei muutu sillä kierroksella. Niinpä seuraavalla kierroksella $S[K[j].h].e$, d ja $S[i].e$ saisivat kullekin kaarelle j samat arvot kuin edellisellä kierroksella, joten silläkään kierroksella ei rivejä 9 ja 10 suoritettaisi kertaakaan. Siksi siinä vaiheessa voidaan lopettaa, eikä algoritmin lopputulos siitä muutu.

Jos lähdöstä ei ole reittiä silmukkaan, jonka pituus on negatiivinen, niin *muuttui* jää arvoon false viimeistään kierroksella n. Osoitamme seuraavaksi, että muussa tapauksessa rivin 8 ehto toteutuu aina ainakin yhdelle silmukan kaarelle, joten algoritmin lopetettua *muuttui* = true.

Olkoot s_0, s_1, s_2 ja niin edelleen mitä tahansa solmuja. Jos s_0:sta on s_1:een kaari, jolle rivin 8 ehto ei toteudu, niin $S[s_0].e + p_{0,1} \geq S[s_1].e$, missä $p_{0,1}$ on tämän kaaren pituus. Jos s_1:stä on s_2:een kaari, jolle rivin 8 ehto ei toteudu, niin $S[s_1].e + p_{1,2} \geq S[s_2].e$. Niinpä $S[s_0].e + p_{0,1} + p_{1,2} \geq S[s_1].e + p_{1,2} \geq S[s_2].e$. Jatkamalla näin ℓ askelta saadaan $S[s_0].e + p_{0,1} + \ldots + p_{\ell-1,\ell} \geq S[s_\ell].e$. Jos tämä päättely tehdään silmukalle, jonka millekään kaarelle ei rivin 8 ehto toteudu, niin s_ℓ on sama solmu kuin s_0, joten $S[s_0].e = S[s_\ell].e$. Jos lähdöstä on reitti silmukkaan, niin $S[s_0].e$ on äärellinen, joten vähentämällä se molemmilta puolilta saadaan $p_{0,1} + \ldots + p_{\ell-1,\ell} \geq 0$. Niinpä jos silmukan pituus on negatiivinen ja lähdöstä on reitti siihen, niin siinä on ainakin yksi kaari, jolle rivin 8 ehto toteutuu.

Toisin kuin Dijkstran algoritmi, Bellman–Ford saattaa laskea solmun etäisyyden ja solmusta lähtevän kaaren pituuden summan myös silloin, kun solmun etäisyys on ääretön. Siksi rivi 7 täytyy toteuttaa siten, että jos $S[K[j].h].e$ on ääretön, niin myös d on ääretön. Tyyppien `float` ja `double` äärettömiä käytettäessä tämä toteutuu automaattisesti, mutta tavallisilla kokonaislukutyypeillä asiasta joudutaan huolehtimaan erikseen.

Jos algoritmin lopetettua *muuttui* = false, niin kussakin solmussa on jonkin siihen johtavan lyhimmän reitin tiedot. Muussa tapauksessa lähdöstä on reitti silmukkaan, jonka pituus on negatiivinen. Tällöin solmujen *e*- ja *r*-kenttiin tallennetut tiedot ovat pätevät vain niiden solmujen osalta, joihin ei ole reittiä sellaisesta silmukasta. Bellman–Ford-algoritmin tuottamista tiedoista ei näe sellaisinaan, mitkä solmut ovat ja mitkä eivät ole tällaisia solmuja, mutta se voidaan selvittää jälkikäteen. Voidaan esimerkiksi suorittaa mikä tahansa luvussa 10 esitellyistä hakualgoritmeista käyttäen lähtöinä jokaista sellaisen kaaren kärkipäätä, jolle rivin 8 ehto toteutuu. Nämä solmut voidaan löytää suorittamalla kuvan 35 kierroksen n sijaan kierros, joka on muuten samanlainen, mutta löydetyt solmut lisätään hakualgoritmin lähtötietoihin.

Solmut, joiden *e*- ja *r*-tiedot eivät ole pätevät, voi löytää myös Dijkstran algoritmilla käyttämällä kaarten pituuksina nollaa. Rivi 63 pitää tällöin tietenkin jättää pois. Dijkstran algoritmi on kuitenkin tähän tehtävään tarpeettoman monimutkainen ja hidas. Keon voi korvata järjestämättömällä taulukolla, jonka alkiossa on vain solmun numero. Silloin nopeus paranee kertaluokkaan $O(n+m)$. Muuttujien `etaisyys` ja `reitti` tilalla voi olla yksi bitti, joka kertoo, onko solmu kohdattu haussa. Näiden muutosten jälkeen algoritmi onkin melkein sama kuin eräs luvun 10 algoritmi.

78. Olkoot $c_{i,j}$ valuuttojen muunnoskurssit siten, että yksi yksikkö valuuttaa j maksaa $c_{i,j}$ yksikköä valuuttaa i, eli yhdellä yksiköllä valuuttaa i voi ostaa $\frac{1}{c_{i,j}}$ yksikköä valuuttaa j. Esimerkiksi yksi Laihian lantti maksaa 0,87 Hollolan dollaria ja yksi Kouvolan kolikko maksaa 2,88 Punkaharjun puntaa. Miten voidaan selvittää, onko valuuttakursseissa sellaista silmukkaa, että ostamalla valuuttaa kierros sen ympäri,

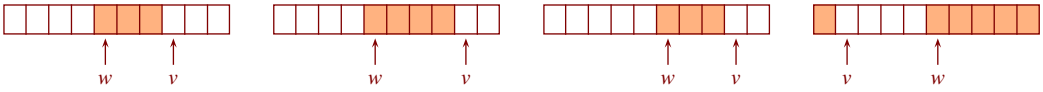
on lopulta alkuperäistä valuuttaa enemmän kuin alussa? Tällainen silmukka olisi esimerkiksi, jos edellä annettujen kurssien lisäksi yksi Hollolan dollari maksaisi 0,73 Kouvolan kolikkoa ja yksi Punkaharjun punta maksaisi 0,54 Laihian lanttia, jolloin jos alussa olisi $0,54 \cdot 2,88 \cdot 0,73 \cdot 0,87 \approx 0,988$ Laihian lanttia, niin sillä saisi ostettua $2,88 \cdot 0,73 \cdot 0,87$ Punkaharjun puntaa, sillä saisi ostettua $0,73 \cdot 0,87$ Kouvolan kolikkoa, sillä saisi ostettua 0,87 Hollolan dollaria, ja sillä saisi ostettua yhden Laihian lantin. (Joka uskoo että mitään tämänkaltaista ei voi tapahtua, ottakoon selvää mitä Suomen sähköpörssissä tapahtui 24.11.2023.)

4.10 Bellman–Fordin nopeuttaminen rengaspuskurilla

Kun kaari on käsitelty Bellman–Fordin algoritmin jollain kierroksella, niin samaa kaarta ei kannata käsitellä uudelleen ennen kuin sen häntäpään solmun $.e$ on muuttunut. Tässä luvussa kehitämme Bellman–Fordin algoritmin muunnoksen, joka hyödyntää tätä.

Ensin suunnittelemme tietorakenteen pitämään kirjaa niistä solmuista, joiden $.e$ on muuttunut. Koska ne halutaan selata tehokkaasti, ne kannattaa tallentaa taulukkoon peräkkäin. Solmua, joka on jo käsitelty käynnissä olevalla Bellman–Fordin kierroksella ja jonka $.e$ muuttui sen jälkeen, ei saa ottaa käsiteltäväksi uudelleen samalla kierroksella, vaan vasta seuraavalla kierroksella. Tämä onnistuu näppärästi käyttämällä ***jonoa*** (***queue***), eli tietorakennetta, johon voi milloin tahansa lisätä alkion ja josta (aina kun se ei ole tyhjä) voi poistaa siinä eniten aikaa olleen alkion, jota ei ole vielä poistettu.

Jonon voi toteuttaa tehokkaasti ***rengaspuskurina*** (***circular buffer***). Rengaspuskuri on taulukko, jossa jonossa olevat alkiot ovat joko peräkkäin tai siten, että osa niistä on taulukon lopussa ja loput taulukon alussa. Taulukossa kauimmin olleen eli wanhimman alkion paikkaa osoittaa kokonaislukumuuttuja w, ja v osoittaa seuraavana vuorossa olevaa vapaata paikkaa, eli sitä, johon seuraavaksi jonoon lisättävä alkio laitetaan. Kun alkio on lisätty, v:tä kasvatetaan yhdellä tai, jos taulukon oikea reuna tuli vastaan, niin v asetetaan osoittamaan taulukon vasempaan reunaan. Tämän voi toteuttaa kätevästi modulo-operaatiolla, jota monissa ohjelmointikielissä merkitään symbolilla `%`. Vastaavasti poistettava alkio otetaan kohdasta w, jonka jälkeen w kasvatetaan tai laitetaan osoittamaan taulukon vasempaan reunaan.



Rengaspuskuri on tyhjä jos ja vain jos $w = v$. Täyden rengaspuskurin voi erottaa tyhjästä esimerkiksi siten, että taulukolle varataan yksi lokero enemmän kuin mitä rengaspuskurissa voi olla yhtäaikaa alkioita.

Bellman–Ford-algoritmin tapauksessa rengaspuskuriin tallennetaan solmujen numeroita, jotka ovat kokonaislukuja. Koska solmun uudesta käsittelystä ei ole hyötyä ellei solmun *.e* ole muuttunut edellisen käsittelyn jälkeen, ei solmua kannata lisätä jonoon jos se on jo siellä, ei vaikka uusi esiintymä kuuluisi eri kierrokselle kuin vanha. Tämä voidaan toteuttaa lisäämällä solmun tietoihin bitti *.m*, joka on true täsmälleen silloin kun solmu on jonossa. Rengaspuskurissamme voi siis olla yhtäaikaa enintään n alkiota, joten taulukon kooksi riittää $n+1$. Käytämme taulukolle nimeä J.

79. Kirjoita jonoon lisääminen ja siitä poistaminen pseudokoodina! Sisällytä toimintoihin myös *.m*-kenttien ylläpito. Kaikki muut tarvittavat muuttujat ovat käytettävissä globaaleina muuttujina, paitsi jonoon lisättävä alkio tulee parametrina ja jonosta poistettu alkio palautetaan **return**-lauseella. Lisäämistä ei tarvitse suojata liian monen alkion lisäämistä vastaan, eikä poistamista tarvitse suojata tyhjästä jonosta poistamista vastaan.

Kullakin kierroksella käsiteltäväksi kuuluvat siis ne kaaret, joiden häntäpäiden solmujen numerot lisättiin jonoon edellisen kierroksen aikana. Nämä solmut saadaan selattua tehokkaasti jonosta, ja niistä lähtevät kaaret saadaan selattua tehokkaasti jos kaaret esitetään samalla tavalla kuin Dijkstran algoritmissa. Solmusta i lähtevien kaarten loppu eli viimeisen kaaren numero plus yksi tallennetaan kenttään $S[i].\ell$, missä "ℓ" luetaan "ell". Se on l-kirjaimen korvike, joka ei helposti sekaannu numeroon 1 eikä isoon i-kirjaimeen. Kullekin kierrokselle kuuluvien solmujen loppuminen voidaan tunnistaa siitä, että rengaspuskurin vanhimman solmun paikka saavutti paikan, jossa rengaspuskurin seuraavana vuorossa oleva vapaa paikka oli kierroksen alkaessa.

80. Kirjoita näin muutettu Bellman–Ford-algoritmi pseudokoodina!

5 Pikajärjestäminen ja mediaanin etsiminen

Pikajärjestäminen (***quicksort***) lienee maailman eniten käytetty isojen taulukoiden järjestämisalgoritmi. Sen suosio perustuu siihen, että se on käytännössä yleensä kilpailijoitaan nopeampi. Se ei ole vakaa, ja se on poikkeuksellisen vaikea ohjelmoida kunnolla. Sen muistin tarve on huonompi kuin kekojärjestämisen (luku 3.6), ja hitaimmillaan se on erittäin hidas. Pikajärjestämisen hitain tapaus saadaan kuitenkin hyvällä toteutuksella niin harvinaiseksi, että sillä on vain vähän merkitystä. (Lomitusjärjestäminen, jota käsitellään luvussa 7.1, on viime aikoina noussut vakavaksi kilpailijaksi.) Tässä luvussa käsitellään myös miten pikajärjestämisen perusajatuksen avulla voidaan keskimäärin tehokkaasti etsiä taulukosta mediaani tai mille tahansa i suuruusjärjestyksessä i:s alkio.

5.1 Lippmanin versio

Pikajärjestäminen toimii rekursiivisesti. Taulukoille, joissa on enintään yksi alkio, sen perusversio ei tee mitään. Muille se vaihtaa alkioiden paikkoja siten, että taulukko jakautuu alkuosaan ja loppuosaan siten, että jokainen alkuosan alkio on enintään yhtäsuuri kuin mikään loppuosan alkio. Tätä kutsutaan ***osittamiseksi*** (***partitioning***). Sitten se järjestää alkuosan samalla tavalla ja loppuosan samalla tavalla. Kuten tulemme näkemään, pikajärjestämisestä on muunnelmia, joissa yksityiskohdat voivat olla hieman toisin.

Osittamiseenkin on toisistaan eroavia menetelmiä. Yleisintä lienee valita järjestettävän osataulukon alkioista jokin ***jakoalkioksi*** (***pivot***). Sitten osataulukkoa selataan alkupäästä alkaen kunnes kohdataan suuri alkio ja loppupäästä alkaen kunnes kohdataan pieni alkio. Suuria alkioita ovat ainakin kaikki jakoalkiota suuremmat alkiot, ja pieniä alkioita ovat ainakin kaikki jakoalkiota pienemmät alkiot. Jakoalkion suuruinen alkio voi tilanteesta riippuen olla suuri tai pieni. Löydetyt suuri ja pieni alkio vaihdetaan keskenään. Tätä toistetaan kunnes alusta ja lopusta alkaneet selaukset kohtaavat.

Kuvassa 36 on eräässä aikanaan suositussa C++-oppikirjassa esitetty pikajärjestäminen. Ladontaa on muutettu niin että ohjelman voi kätevästi näyttää luentosalin valkokankaalla. Muuten kuva 36 on suora kopio kirjasta. Siinä käytetään C:stä peräisin olevia taulukoita eikä `vector`:eita. Tämä vaikuttaa vain riveihin 1 ja 3.

Lippmanin `qsort` järjestää taulukon `ia` osan, jonka ensimmäinen lokero on kohdassa `low` ja viimeinen kohdassa `high`. Sen suorituksen aikana `low` ja `high`

```
1  static void swap( int *ia, int i, int j )
2    { int tmp = ia[i]; ia[i] = ia[j]; ia[j] = tmp; }
3  void qsort( int *ia, int low, int high ){
4    if( low < high ){
5      int lo = low; int hi = high + 1; int elem = ia[ low ];
6      for (;;){
7        while ( ia[ ++lo ] <= elem ) ;
8        while ( ia[ --hi ] > elem ) ;
9        if( lo < hi ) swap( ia, lo, hi );
10         else break;
11     } // end, for(;;)
12     swap( ia, low, hi );
13     qsort( ia, low, hi - 1 ); qsort( ia, hi + 1, high );
14   } // end, if ( low < high )
15 }
```

Kuva 36: Lippmanin `qsort` (S.B. Lippman, C++ Primer, s. 128, 1989)

eivät muutu. Rivien 1 ja 2 ansiosta `swap`$(\texttt{ia}, x, y)$ vaihtaa lokeroiden x ja y sisällöt keskenään. Sana `static` ei ole tässä luvussa tärkeä (se on osa vanhanaikaista keinoa estää funktiota `swap` sekoittumasta muihin samannimisiin funktioihin). Jos osassa $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ on enintään yksi alkio, niin `qsort` suorittaa vain rivit 3 ja 4 ja lopettaa. Muussa tapauksessa se valitsee osan $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ ensimmäisen alkion jakoalkioksi `elem`. Se valmistautuu selaamaan osan $\texttt{ia}[\texttt{low}+1\ldots\texttt{high}]$ asettamalla `lo`:n ja `hi`:n yhden lokeron verran osan alun ja lopun ohi.

Rivillä 7 `qsort` selaa osaa $\texttt{ia}[\texttt{low}+1\ldots\texttt{high}]$ alusta alkaen kunnes se löytää suuren alkion. Rivillä 8 se selaa lopusta alkaen kunnes se löytää pienen alkion. Jos selaukset eivät vielä kohdanneet, se vaihtaa löydetyt alkiot keskenään ja jatkaa selauksia. Muussa tapauksessa se siirtyy riville 12, jossa se vaihtaa jakoalkion alkuosan viimeisen alkion kanssa. Nyt osassa $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ on ensin pienet alkiot kohdissa $\texttt{low}, \ldots, \texttt{hi}-1$, sitten jakoalkio kohdassa `hi` ja lopuksi suuret alkiot kohdissa $\texttt{hi}+1, \ldots, \texttt{high}$. Sitten `qsort` kutsuu itseään rekursiivisesti järjestääkseen alkuosan ja uudelleen järjestääkseen loppuosan.

Tietokoneella, jolla tämä teksti kirjoitettiin, `qsort` järjesti silmänräpäyksessä taulukon, jossa oli alkiot 0, 1, ..., 199 999 sekalaisessa järjestyksessä. Kun samat alkiot olivat kasvavassa järjestyksessä, aikaa meni 6,7 sekuntia. Taulukon, jossa oli 200 000 kappaletta keskenään yhtäsuuria alkioita, järjestämiseen kului 9,8 sekuntia. Eräällä pienellä taulukolla ohjelma kaatui ilmoittaen ”Muistialueen ylitys”.

Pian tekstissä tulee vastaan kohta (a), sitten (b) ja myöhemmin (c). Lukijaa pyydetään lopettamaan lukeminen niissä ja yrittämään `qsort`:in vian tai vikojen löytämistä. Kohtien (a) ja (b) välissä annetaan tietoa, joka saattaa auttaa vian tai vikojen löytämisessä. Kohtien (b) ja (c) välissä annetaan lisää tietoa. Kohdan (c)

jälkeen vika tai viat kerrotaan ja näytetään miten ne voi löytää. Lippmanin `qsort` ei itsessään ansaitse pitkää pohdintaa, koska se ei toimi kunnolla. Kohdassa (c) alkava pohdinta kuitenkin tarjoaa mainion mahdollisuuden harjoitella todellisesta ohjelmakoodista päättelemistä ja kohdata pikajärjestämisen kannalta olennaisia asioita. Siksi se on pitkä ja perusteellinen.

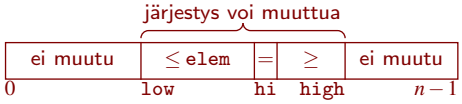
(a) Aiemmissa luvuissa moni algoritmi osoitettiin oikein toimivaksi käyttäen kolmea eri keinoa. Kukin silmukka tarkastettiin esittämällä sen perusidea silmukkainvariantin muodossa ja tarkastamalla, että silmukkainvariantilta sivulla 18 vaaditut kolme asiaa toteutuvat. Erikseen tarkastettiin, että jokainen silmukka pysähtyy ja että ohjelma ei suorita laittomia toimintoja. Nämä työvaiheet tehtiin osittain lomittain, muun muassa siksi että toimintojen laillisuuden tarkastaminen edellytti, että silmukoiden toiminnasta tiedettiin jo jotakin, ja silmukoiden tarkastamisen vieminen loppuun hyötyi siitä, että toiminnot oli jo tarkastettu laillisiksi.

Näitä keinoja voi soveltaa myös `qsort`:in vikojen etsimiseen. Silmukkainvariantit saadaan miettimällä, mikä on kunkin silmukan tehtävä tai tavoite. Koska `qsort` on rekursiivinen, tarvitsee myös tarkastaa, että rekursio päättyy.

(b) Lippmanin kirjan toisessa painoksessa (2nd Edition) sivulla 138 oli rivi 7 muutettu muotoon `while ( ia[ ++lo ] < elem ) ;`. Keskenään yhtäsuurista alkioista muodostuvan taulukon järjestämisaika muuttui silmänräpäykseksi, mutta muut viat säilyivät.

(c) Jotta `ia` olisi oikeassa järjestyksessä rivin 13 lopussa, täytyy kahden asian päteä rivin 13 alussa:

- Osassa $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ on samat alkiot kuin rivin 3 lopussa. Osissa $\texttt{ia}[0\ldots\texttt{low}-1]$ ja $\texttt{ia}[\texttt{high}+1\ldots n-1]$ on samat alkiot samassa järjestyksessä kuin rivin 3 lopussa, missä n on koko taulukon `ia` koko.
- Jos $\texttt{low} \leq x < \texttt{hi}$ niin $\texttt{ia}[x] \leq \texttt{elem}$, $\texttt{ia}[\texttt{hi}] = \texttt{elem}$ ja jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] \geq \texttt{elem}$.



Koska `ia`:n sisältöä muutetaan vain vaihtamalla kahden alkion paikat keskenään `swap`:illa, riittää ylemmän väitteen tarkastamiseksi tarkastaa, että kukin `swap` käsittelee vain osan $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ alkioita. Koodissa on kaksi `swap`:ia, nimittäin riveillä 9 ja 12.

Rivillä 9 on `swap( ia, lo, hi )`. Sille pätee $\texttt{lo} < \texttt{hi}$, koska muutoin sitä ei suoriteta. Rivillä 9 $\texttt{lo} > \texttt{low}$ rivien 5 ja 7 ansiosta ja $\texttt{hi} \leq \texttt{high}$ rivien 5 ja 8 ansiosta.

Niinpä rivin 9 `swap`:in kohdalla $\texttt{low}+1 \leq \texttt{lo} < \texttt{hi} \leq \texttt{high}$. Siksi rivin 9 `swap` käsittelee vain sitä mitä saa käsitellä. Lisäksi huomaamme, että $\texttt{ia}[\texttt{low}]$ ei muutu, joten $\texttt{ia}[\texttt{low}] = \texttt{elem}$ riveillä 6, ..., 11.

Rivillä 12 on `swap( ia, low, hi )`. Siis tarvitsee tarkastaa, että $\texttt{low} \leq \texttt{hi} \leq \texttt{high}$. Rivi 5 asettaa $\texttt{hi} = \texttt{high}+1$, mutta sieltä ei pääse riville 12 muuten kuin rivin 8 kautta, joten rivillä 12 pätee $\texttt{hi} \leq \texttt{high}$. Rivin 4 vuoksi $\texttt{low} < \texttt{high}+1$, ja äsken huomasimme että ennen riviä 12 koko ajan $\texttt{ia}[\texttt{low}] = \texttt{elem}$. Siksi rivi 8 pysähtyy viimeistään arvoon $\texttt{hi} = \texttt{low}$, joten $\texttt{low} \leq \texttt{hi} \leq \texttt{high}$ pätee rivillä 12.

81. Voiko `qsort` koskaan tehdä niin, että se ensin vaihtaa kaksi keskenään oikeassa järjestyksessä olevaa erisuurta alkiota väärään järjestykseen, ja myöhemmin vaihtaa ne uudelleen oikeaan järjestykseen? Perustele vastauksesi.

Sivun 79 alemman riviä 13 koskevan väitteen tarkastamiseksi perustelemme seuraavat kaksi väitettä:

(a) Rivin 6 alussa pätee: Jos $\texttt{low}+1 \leq x \leq \texttt{lo}$ niin $\texttt{ia}[x] \leq \texttt{elem}$, ja jos $\texttt{hi} \leq x \leq \texttt{high}$ niin $\texttt{ia}[x] > \texttt{elem}$.

(b) Rivin 9 alussa pätee: $\texttt{ia}[\texttt{lo}] > \texttt{elem}$, $\texttt{ia}[\texttt{hi}] \leq \texttt{elem}$, jos $\texttt{low}+1 \leq x < \texttt{lo}$ niin $\texttt{ia}[x] \leq \texttt{elem}$, ja jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] > \texttt{elem}$.

Rivi 5 saattaa (a):n voimaan asettamalla muuttujiin `lo` ja `hi` sellaiset arvot, että jos-osuudet eivät toteudu millään x. Rivillä 7 `lo`:ta kasvatetaan kunnes sen osuus (a):sta lakkaa olemasta voimassa, ja rivillä 8 tehdään vastaava `hi`:lle. Siksi rivin 9 alussa pätee (b). Jos palataan riville 6, niin $\texttt{ia}[\texttt{lo}]$ ja $\texttt{ia}[\texttt{hi}]$ vaihdettiin keskenään, jolloin (a) palasi voimaan.

Jollei palata riville 6, niin $\texttt{lo} \geq \texttt{hi}$ ja hypätään riville 12. Nyt $\texttt{ia}[x] \leq \texttt{elem}$ pätee jos $\texttt{low}+1 \leq x < \texttt{hi}$, koska (b) lupasi sen jos $\texttt{low}+1 \leq x < \texttt{lo}$, ja nyt $\texttt{hi} \leq \texttt{lo}$. Se pätee myös jos $x = \texttt{hi}$, koska (b) lupasi myös $\texttt{ia}[\texttt{hi}] \leq \texttt{elem}$. Sivulla 80 todettiin, että $\texttt{ia}[\texttt{low}] = \texttt{elem}$. Niinpä rivillä 12 suoritettava `swap( ia, low, hi )` tuottaa alemman sivulla 79 riviä 13 koskevan väitteen alkuosan "jos $\texttt{low} \leq x < \texttt{hi}$ niin $\texttt{ia}[x] \leq \texttt{elem}$" ja keskiosan "$\texttt{ia}[\texttt{hi}] = \texttt{elem}$". Sen loppuosa "jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] \geq \texttt{elem}$" on voimassa rivin 12 lopussa, koska se oli (b):n vuoksi voimassa rivin 12 alussa eikä rivin 12 `swap` vaikuta siihen, koska sivulla 80 todettiin että $\texttt{low} \leq \texttt{hi}$.

Vielä pitää tarkastaa indeksointien laillisuudet ja että jokainen silmukka ja rekursio lopettavat. Sivulla 80 todettiin, että $\texttt{low} \leq \texttt{hi} \leq \texttt{high}$ rivillä 12. Siksi kumpikin rivin 13 rekursiivinen kutsu käsittelee pienempää taulukon osaa kuin mikä

rivillä 3 saatiin käsiteltäväksi. Taulukon osa voi pienentyä korkeintaan niin monta kertaa kuin siinä on alkioita, joten rekursio lopettaa ennemmin tai myöhemmin.

Totesimme sivulla 80, että rivi 8 pysähtyy viimeistään arvoon `hi` = `low`. Mutta mikä pysäyttää rivin 7 silmukan?

Rivin 7 silmukka etenee kunnes se kohtaa alkion, joka on suurempi kuin `elem`. Osassa `ia`[`low...high`] ei sellaista välttämättä ole. Voi olla, että sellaista ei ole edes koko taulukossa. Siinä tapauksessa rivin 7 silmukka jatkaa `ia`:n ulkopuolelle. Rivin 7 silmukka voi siis indeksoida `ia`:ta laittomasti.

C++:ssa on indeksointi, joka heittää poikkeuksen indeksin ollessa laiton, mutta se ei ole `[...]` vaan `.at(...)` (ja sitä voi käyttää `vector`:eille, eikä C:stä peräisin oleville taulukoille). Siksi rivin 7 silmukasta tulee virheilmoitus vasta kun se siirtyy käyttöjärjestelmän ohjelman suoritusta varten varaaman muistialueen ulkopuolelle. On kuitenkin mahdollista, usein jopa hyvin todennäköistä, että ennen sitä kohdataan muistipaikkojen ryhmä, jonka sisältö `int`-tyyppiseksi tulkittuna on suurempi kuin `elem`. Siinä tapauksessa silmukka pysähtyy ilman virheilmoitusta ja ohjelman suoritus jatkuu normaalisti rivillä 8.

Niinpä virhe jää todennäköisesti havaitsematta, ellei taulukossa ole yhtään lukua, joka on lähellä suurinta lukua, jonka `int`-tyyppinen muuttuja voi sisältää. Tätä kirjoitusta varten tehdyissä koeajoissa virhe toisinaan ilmeni ja toisinaan ei ilmennyt syötteellä $[2\,070\,000\,000, 0]$. Virhe ilmeni ohjelman kaatumisena ilmoituksella ”Muistialueen ylitys”.

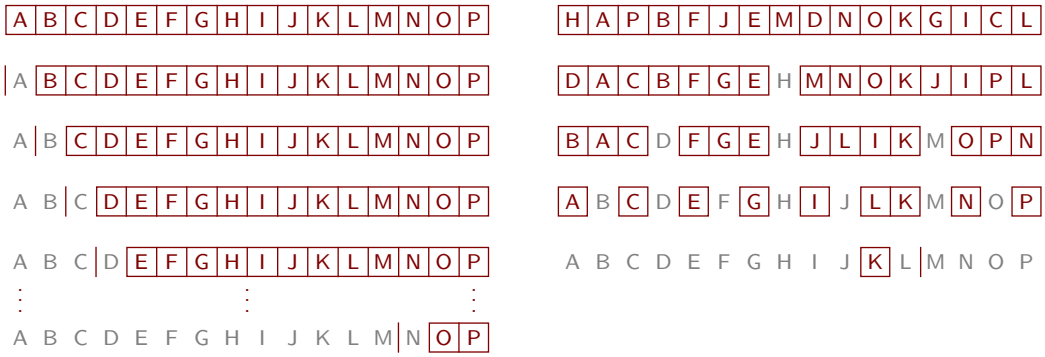
82. Totesimme edellä, että jos mikään osan `ia`[`low...high`] alkio ei ole suurempi kuin `elem`, niin rivin 7 silmukka selaa kohdan `high` ohi. Jos niin käy, niin se tapahtuu rivien 6, ..., 11 `for`-silmukan ensimmäisellä kierroksella. Voiko rivin 7 silmukka selata kohdan `high` ohi `for`-silmukan millään muulla kuin ensimmäisellä kierroksella? Perustele vastauksesi.

83. Päättele kysymyksen 82 vastauksesta, että jos ainakin yksi osan `ia`[`low...high`] alkio on suurempi kuin `elem`, niin rivin 7 silmukka ei selaa kohdan `high` ohi.

Jatkossa `qsort`:ista puhuessamme tarkoitamme ***ohi-indeksoinnilla*** `ia`:n indeksointia, joka ei osu osataulukkoon `ia`[`low...high`]. Lailliset ohi-indeksoinnit osuvat muualle `ia`:han, ja laittomat ohi-indeksoinnit osuvat `ia`:n ulkopuolelle.

Käymme vielä loput indeksoinnit läpi nähdäksemme, onko niissäkin virheitä. Indeksoinnit `swap`:in kutsuissa sekä rivi 8 on jo tarkastettu. Rivin 5 `ia[ low ]` on laillinen, koska rivi 4 takaa `low` $\leq$ `high`. Niinpä rivin 7 indeksointivirhe on ainoa.

Rivin 7 virhe ei selitä sitä, että `qsort` käytti taulukolle $[0, 1, \ldots, 199\,999]$ huomattavasti enemmän aikaa kuin samoista alkioista muodostetulle sekalaisessa järjestyksessä olevalle taulukolle. Nimittäin taulukolla $[0, 1, \ldots, 199\,999]$ rivi 7 pysähtyy aina heti, eikä ohi-indeksointeja tapahdu. Sitäpaitsi vastauksesta 81 seuraa,



Kuva 37: `qsort`:in käyttäytyminen, kun ositukset menevät mahdollisimman huonosti ja mahdollisimman hyvin tasan

että sen järjestämisen aikana alkioita ei lainkaan siirrellä. Senhän luulisi nopeuttavan järjestämistä!

Taulukon $[0,1,\ldots,199\,999]$ järjestämisen hitauden syy on seuraava. Se on ***aidosti kasvava*** (***strictly increasing***), mikä tarkoittaa, että se on kasvavassa järjestyksessä ja koostuu keskenään erisuurista alkioista. Rivi 8 lopettaa aidosti kasvavassa järjestyksessä olevalle taulukolle vasta kun $\texttt{hi} = \texttt{low}$. Tällöin ensimmäinen rivin 13 rekursiivisista kutsuista saa tyhjän osataulukon $\texttt{ia}[\texttt{low}\ldots\texttt{low}-1]$ ja jälkimmäinen yhtä vaille kaikista rivillä 3 saaduista alkioista muodostuvan osataulukon $\texttt{ia}[\texttt{low}+1\ldots\texttt{high}]$. Siksi `qsort`:in alkuperäinen ja rekursiiviset kutsut selaavat rivillä 8 yhteensä $n+(n-1)+\ldots+2$ eli $\Theta(n^2)$ alkiota (suunnilleen $\frac{1}{2}n^2$).

Kuvan 37 vasemmalla puolella on havainnollistettu tätä taulukolle $[\mathsf{A},\mathsf{B},\ldots,\mathsf{P}]$. Kunkin alkuperäisen tai rekursiivisen `qsort`:in kutsun saama osataulukko on esitetty ruskeana vaakasuorana ruudukkona tai pystyviivana (tyhjä osataulukko). Harmaat alkiot ovat olleet rivillä 13 lokerossa `hi`, joten niitä ei enää käsitellä.

Aliohjelmakutsun ***rekursiotaso*** (***level of recursion***) on yksi plus niiden aikaisempien aliohjelmakutsujen määrä, joiden suoritus oli kutsuhetkellä kesken. Samalla rekursiotasolla olevien kutsujen saamat osataulukot ovat kuvassa 37 vierekkäin. Kuvan oikealla puolella jokainen ositus jakaa osataulukon mahdollisimman tarkasti keskeltä. Silloin rekursiotasojen määrä on melko tarkasti $\log_2 n$. Kullakin rekursiotasolla käsitellään enintään n alkiota, ja melkein kaikilla ainakin $\frac{1}{2}n$ alkiota. Ositus käyttää aikaa selaamiensa alkioiden määrään verrannollisesti. Jos ohi-indeksointeja tapahtuu vain vähän tai ei lainkaan, niin ajan kulutus on kaiken kaikkiaan $\Theta(n\log n)$.

Sekalaisessa järjestyksessä olevan, alkioista 0, 1, …, 199 999 muodostetun taulukon järjestämisajan huolellinen analyysi olisi vaikeaa. Voidaan kuitenkin arvioida, että melko moni rekursiokutsu jakaa saamansa osataulukon melko läheltä puoliväliä. Siitä seuraa, että `qsort`:in käyttäytyminen muistuttaa enemmän ku-

van 37 oikeaa kuin vasenta puolta. (Voidaan osoittaa, että rekursiotasojen määrä on keskimäärin alle $5\log_2 n$.) Kumpikin riveistä 7 ja 8 selaa saadun osataulukon enintään kerran. Kysymysten 83 ja 84 vastaukset antavat vahvan syyn uskoa, että ohi-indeksointien määrä on normaalisti niin pieni, että sitä ei tarvitse ottaa huomioon. Siksi rekursiotason työmäärää voidaan kuvata $O(n)$:llä ja kokonaistyömäärää $O(n\log n)$:llä.

Kun $n = 200\,000$, on $\frac{1}{2}n^2$ paljon suurempi kuin $5n\log_2 n$. Tämä riittää selittämään, miksi taulukon $[0, 1, \ldots, 199\,999]$ järjestäminen kesti paljon kauemmin kuin samoista alkioista muodostetun sekalaisessa järjestyksessä olevan taulukon järjestäminen.

84. Perustele, että jos kaikki alkiot ovat keskenään erisuuret ja kutsu ei ole rekursiotasonsa viimeinen, niin rivi 7 pysähtyy viimeistään kun `lo` $=$ `high` $+ 1$!

On helppo selittää, miksi kaikkien alkioiden ollessa yhtäsuuret aikaa kului vielä selvästi enemmän kuin aidosti kasvavalla taulukolla. Kun kaikki alkiot ovat yhtäsuuret, pysähtyy rivin 8 silmukka heti. Niinpä rivin 13 ensimmäinen kutsu saa osan `ia`$[$`low`$\ldots$`high`$-1]$ ja jälkimmäinen tyhjän osan `ia`$[$`high`$+1\ldots$`high`$]$. Nytkin rekursiotasoja tulee n kappaletta (paitsi kun $n = 0$). Jokaisella kutsulla, jolla `low` $<$ `high`, `lo` selaa koko alkuperäisen taulukon loppuun ja jatkaa kunnes kohtaa muistipaikkojen ryhmän, jonka sisältö tulkittuna `int`-tyyppiseksi on suurempi kuin `elem`. Niinpä tehdään paljon ohi-indeksointeja. Kysymyksestä 83 seuraa, että aidosti kasvavalla taulukolla ohi-indeksointeja ei tapahdu.

Pohdimme seuraavaksi, miten käyttäytyminen muuttui, kun rivi 7 muutettiin muotoon `while ( ia[ ++lo ] < elem ) ;`. Jos kaikki alkiot ovat yhtäsuuret, niin sekä rivi 8 että muutettu rivi 7 pysähtyvät aina heti. Jos silloin `lo` $<$ `hi`, niin riviltä 8 mennään rivien 9 ja 6 kautta uudelleen riville 7. Niinpä `lo` ja `hi` etenevät tasatahtia kohti toisiaan. Ne kohtaavat osataulukon puolivälissä, joten osataulukko jaetaan yhtäsuuriin tai yhtä vaille yhtäsuuriin osiin. Rekursiotasoja tulee likimain $\log_2 n$ ja suoritusaika 200 000 alkiolla muuttuu silmänräykseksi.

Keskenään erisuurten alkioiden tapauksessa on yhä mahdollista, että `lo` karkaa taulukon oikean reunan ohi ja jatkaa ohjelmalle varatun muistialueen ulkopuolelle. Se on nyt hiukan epätodennäköisempää kuin aiemmin, koska nyt eteneminen pysähtyy myös jos kohdataan muistipaikkojen ryhmä, jonka sisältö tulkittuna `int`-tyyppiseksi on täsmälleen `elem`:in suuruinen. Tällä erolla ei yleensä liene suurta merkitystä. Muuta vaikutusta keskenään erisuurten alkioiden tapaukseen ei muutoksella ole, koska `<=` ja `<` tuottavat saman tuloksen kun verrataan kahta erisuurta alkiota.

Lippman oli tarkoittanut `qsort`:in havainnollistamaan C++:n rekursiivisia funktioita eikä sitä, miten pikajärjestäminen kannattaa toteuttaa. Silti siinä esiintyneet

```
    QUICKSORT(&A, a, y)
1   if a ≥ y then return
2   x := A[RANDOM(a, y)].x
3   i := a; j := y
4   while true do
5     while A[i].x < x do i := i + 1
6     while A[j].x > x do j := j − 1
7     if i ≥ j then break
8     SWAP(A, i, j)
9     i := i + 1; j := j − 1
10  QUICKSORT(A, a, i − 1); QUICKSORT(A, j + 1, y)
```

Kuva 38: Pikajärjestämisen perusversio

viat ovat esimerkki siitä, että pikajärjestäminen on vaikea toteuttaa kunnolla. Tapaus on esimerkki myös siitä, että joitakin virheitä on melkein mahdoton löytää testaamalla.

5.2 Perusversio

Kuvassa 38 on pikajärjestämisestä laadukas versio, jossa ei kuitenkaan ole mukana luvussa 5.3 esitettäviä merkittäviä tehostuskeinoja. Se järjestää taulukon osan $A[a \ldots y]$. Jos siinä on enintään yksi alkio, QUICKSORT lopettaa heti. Muussa tapauksessa se valitsee satunnaisesti yhden osan $A[a \ldots y]$ alkioista jakoalkioksi, ja ottaa sen avaimen arvon talteen muuttujaan x. Sitten se selaa osaa $A[a \ldots y]$ molemmista päistä alkaen kunnes selaukset kohtaavat. Aina kun se on löytänyt alkuosasta jakoalkion suuruisen tai suuremman alkion ja loppuosasta jakoalkion suuruisen tai pienemmän alkion, se vaihtaa ne keskenään. (Kun sanomme, että alkio on suurempi kuin jokin, niin tarkoitamme, että alkion avain on suurempi kuin jokin, ja samoin ilmauksille pienempi ja yhtäsuuri.) Kun selaukset ovat kohdanneet, QUICKSORT kutsuu itseään rekursiivisesti alku- ja loppuosille.

On osoitettava, että rivillä 10 osan $A[a \ldots i-1]$ alkiot ovat enintään yhtäsuuria kuin osan $A[i \ldots j]$ alkiot, ja molempien näiden osien alkiot ovat enintään yhtäsuuria kuin osan $A[j+1 \ldots y]$ alkiot. Koska keskimmäistä osaa ei jatkokäsitellä, on osoitettava, että siinä on enintään yksi alkio, eli $i \geq j$. Ajan kulutuksen analyysi olettaa, että alku- ja loppuosa eivät mene yhtään toistensa päälle. Siksi osoitamme, että $i-1 < j+1$ rivillä 10. Samalla varmistuu, että keskimmäinen osa on hyvin määritelty, eli $j \geq i-1$. Jotta olisi varmaa että QUICKSORT ei jää ikuiseen rekursioon, osoitamme, että $A[a \ldots i-1]$ ja $A[j+1 \ldots y]$ ovat pienemmät kuin $A[a \ldots y]$. Myös on osoitettava, että jokainen silmukka pysähtyy, A:ta ei indeksoida laittomasti, eikä haittaa aiheuttavia aritmeettisia ylivuotoja tapahdu. Koska ainoa tapa, jolla QUICKSORT muuttaa A:n sisältöä on kahden alkion vaihtaminen keskenään,

on selvää, että A:ssa on lopussa samat alkiot kuin alussa. Se ei tarvitse tämän enempää todistusta.

85. Nyt pääset yrittämään porsaanreiän etsimistä. Miksi ei riitä vaatia, että osan $A[a \ldots i-1]$ alkiot ovat enintään yhtäsuuria kuin osan $A[i \ldots j]$ alkiot, ja osan $A[i \ldots j]$ alkiot ovat enintään yhtäsuuria kuin osan $A[j+1 \ldots y]$ alkiot?

Riveillä $2, \ldots, 10$ pätee $a < y$ rivin 1 vuoksi. Koska x on peräisin jostakin osan $A[a \ldots y]$ alkiosta, näyttää tilanne rivin 3 lopussa tältä:

Rivien $4, \ldots, 9$ silmukalla on seuraava invariantti:

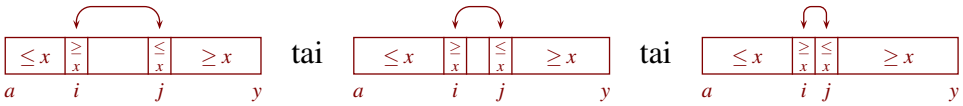
1. $a \le i \le y$ ja $a \le j \le y$ ja $i \le j+1$.
2. Kukin osan $A[a \ldots i-1]$ alkio on enintään x ja kukin osan $A[j+1 \ldots y]$ alkio on vähintään x.
3. Osassa $A[i \ldots y]$ on ainakin yksi alkio, joka on vähintään x. Osassa $A[a \ldots j]$ on ainakin yksi alkio, joka on enintään x.

Kun riville 4 tullaan ensimmäisen kerran on invariantin osa 1 voimassa, koska rivien 1 ja 3 vuoksi $a = i < y$ ja $a < j = y$. Silloin $A[a \ldots i-1]$ ja $A[j+1 \ldots y]$ ovat tyhjät, joten myös osa 2 on voimassa. Osa 3 seuraa siitä, että x poimittiin rivillä 2 jostakin osan $A[a \ldots y]$ alkiosta.

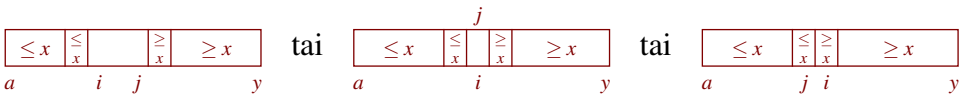
Rivin 3 jälkeen i voi muuttua vain kasvamalla rivillä 5 tai 9 ja j vähenemällä rivillä 6 tai 9. Siksi riveillä $4, \ldots, 9$ pätee $a \le i$ ja $j \le y$. Koska riville 9 ei tulla ellei $i < j$, pätee rivin 9 alussa $a \le i < j \le y$ eli $i+1 \le j \le y$ ja $a \le i \le j-1$, joten myös rivin 9 lopussa $i \le y$ ja $a \le j$. Tiedosta $i < j$ seuraa myös $i+1 \le (j-1)+1$, joten rivin 9 lopussa $i \le j+1$. Niinpä osa 1 pätee riveillä $4, \ldots, 9$ koko ajan. (Invariantin osa 3 takaa, että rivin 5 lopussa $i \le y$ ja rivin 6 lopussa $a \le j$, mutta tätä tietoa tarvitaan vasta myöhemmin.)

Rivit 5 ja 6 säilyttävät invariantin osan 2 voimassa, ja lisäksi saattavat voimaan $A[i].x \ge x$ ja $A[j].x \le x$.

Jos rivin 7 alussa $i < j$, niin tilanne näyttää tältä:

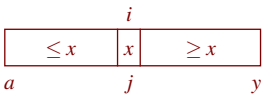


Rivin 8 SWAP vaihtaa kaksipäisen nuolen osoittamat alkiot keskenään. Siksi invariantin osa 2 säilyy voimassa rivillä 9. Kohdasta i kohtaan j siirtyvä alkio saattaa osan 3 alkuosan voimaan, ja kohdasta j kohtaan i siirtyvä alkio tekee saman osan 3 loppuosalle. Rivin 9 lopussa tilanne näyttää tältä:



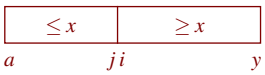
Olemme osoittaneet, että koko invariantti säilyy voimassa, kun rivien 4, ..., 9 silmukan runko suoritetaan.

Jos rivin 7 alussa $i < j$ ei päde, niin on kaksi mahdollisuutta. Jos $i = j$, niin $A[i].x = x$, koska äsken todettiin että rivin 7 alussa $A[i].x \geq x$ ja $A[j].x \leq x$.



Osassa $A[i \ldots j]$ on nyt täsmälleen yksi alkio, ja se on x:n suuruinen. Koska invariantin osa 2 on voimassa rivillä 7, on se voimassa myös rivillä 10. Niinpä osan $A[a \ldots i-1]$ alkiot ovat enintään yhtäsuuria kuin osan $A[i \ldots j]$ ainoa alkio, ja se on enintään yhtäsuuri kuin osan $A[j+1 \ldots y]$ alkiot. Koska $i = j$, pätee $i - 1 < j + 1$. Ensimmäinen ja viimeinen osa ovat pienempiä kuin $A[a \ldots y]$, koska ainakin $A[i]$ puuttuu niistä.

Muussa tapauksessa rivin 7 alussa $j < i$. Osoitamme, että $i = j + 1$. Invariantin osan 1 mukaan rivin 4 alussa päti $i \leq j + 1$. Tarvitsee osoittaa, että i ei kasvanut eikä j vähentynyt sen jälkeen kun $j < i$ astui voimaan, eli $i = j+1$ päti. Invariantin osa 3 takaa, että rivillä 5 päti $i \leq y$. Siksi kun $i = j + 1$ päti, päti $j + 1 \leq i \leq y$, joten invariantin osan 2 vuoksi rivin 5 silmukka ei jatkanut. Vastaavasti rivilllä 6 päti $a \leq j$, joten kun $i = j+1$ päti, päti $a \leq j \leq i-1$ ja rivin 6 silmukka ei jatkanut.



Koska $j + 1 = i$, on osa $A[i \ldots j]$ tyhjä ja $i - 1 < j + 1$ pätee. Invariantin osan 2 vuoksi ovat osan $A[a \ldots i-1]$ alkiot enintään yhtäsuuria kuin osan $A[j+1 \ldots y]$ alkiot. Koska $i \leq y$ ja $a \leq j$, ovat $A[a \ldots i-1]$ ja $A[j+1 \ldots y]$ pienempiä kuin $A[a \ldots y]$.

Olemme osoittaneet, että rivillä 10 alkuosan alkiot ovat enintään keskiosan ja loppuosan alkioiden suuruisia, ja keskiosan alkiot ovat enintään loppuosan alkioiden suuruisia. Olemme myös osoittaneet, että $A[a \ldots i-1]$ ja $A[j+1 \ldots y]$ ovat pienemmät kuin $A[a \ldots y]$, ja että $i - 1 < j + 1$.

Rivien 5 ja 6 silmukat pysähtyvät invariantin osan 3 vuoksi. Rivien 4, ..., 9 silmukka pysähtyy, koska jokaisella paitsi viimeisellä kierroksella i kasvaa ja j vähenee rivillä 9 eivätkä i ja j koskaan muutu päinvastaiseen suuntaan, joten lopulta $i \geq j$. Taulukkoa A ei indeksoida laittomasti rivillä 2, koska RANDOM(a, y) tuottaa satunnaisluvun väliltä $a, \ldots, y$. Muut indeksoinnit ovat laillisia, koska $a \leq i \leq y$ ja $a \leq j \leq y$ riveillä $4, \ldots, 9$. Rivin 10 kutsuissa yläraja voi olla alaraja miinus

yksi mutta ei vähemmän, ja muussa tapauksessa molemmat rajat ovat osataulukon $A[a \ldots y]$ alueella.

Riveillä 4, ..., 9 ei tapahdu aritmeettisia ylivuotoja, koska $a \leq i \leq y$ ja $a \leq j \leq y$. Riveillä 1, ..., 3 ei ole aritmeettisia laskutoimituksia. Rivillä 10 on vaarana, että $i-1$ tai $j+1$ aiheuttaa ylivuodon. Niin käy ainakin silloin, kun käytettävä tyyppi on etumerkitön ja $a = i = 0$. Jos ei ole varmaa, että käytetty tyyppi kykenee esittämään luvut $a-1$ ja $y+1$, voi rivin 10 sijaan kirjoittaa **if** $i > a+1$ **then** QUICKSORT$(A, a, i-1)$ ja **if** $j < y-1$ **then** QUICKSORT$(A, j+1, y)$, ja alkuperäisen kutsun sijaan **if** $a < y$ **then** QUICKSORT(A, a, y). Tällöin jokaisessa kutsussa $a < y$, joten $a+1$ ja $y-1$ eivät vuoda yli. Samasta syystä rivin 1 voi jättää pois. Tämä muutos saattaa jopa hieman nopeuttaa ohjelmaa, koska se vähentää QUICKSORT:in käynnistysten määrää. Vaihtoehtoisesti koodin voi muuttaa siten että osataulukon viimeisen indeksin sijaan ilmoitetaan se plus yksi, mutta silloin menetetään koodin symmetria ja rivi 1 täytyy korvata monimutkaisemmalla.

Jos osataulukon kaikki alkiot ovat keskenään samansuuruisia, niin QUICKSORT jakaa sen täsmälleen keskeltä. Jos osataulukossa on paljon jakoalkion suuruisia mutta myös muita alkioita, ei QUICKSORT välttämättä jaa sitä ihanteellisesti, mutta ei myöskään kovin huonosti. Tätä analysoidaan kysymyksissä 86, ..., 88.

86. Oletetaan, että osataulukossa on jakoalkion suuruisia alkioita enintään yhtä paljon kuin jakoalkiota pienempiä alkioita. Jos jakoalkion suuruiset alkiot ovat aluksi, sitten jakoalkiota suuremmat ja lopuksi jakoalkiota pienemmät, niin kuinka moni jakoalkion suuruinen alkio päätyy alkuosaan ja kuinka moni loppuosaan?

87. Olkoon jakoalkion suuruisia alkioita enintään yhtä paljon kuin jakoalkiota suurempia alkioita. Enintään kuinka moni jakoalkion suuruinen alkio voi päätyä alkuosaan? Anna esimerkki osataulukon järjestyksestä, jolla mahdollisimman moni jakoalkion suuruinen alkio päätyy alkuosaan!

88. Olkoon s jakoalkiota suurempien ja k jakoalkion suuruisten alkioiden määrä. Jos $k > s$, niin enintään kuinka moni jakoalkion suuruinen alkio voi päätyä alkuosaan? Anna esimerkki osataulukon järjestyksestä, jolla mahdollisimman moni jakoalkion suuruinen alkio päätyy alkuosaan!

Toisin kuin Lippmanin `qsort`, kuvan 38 QUICKSORT ei osita aidosti kasvavia eikä minkään muunkaanlaisia taulukoita systemaattisesti huonosti. Jos osataulukon kaikki alkiot ovat keskenään erisuuret, niin QUICKSORT jakaa osataulukon keskimäärin joka toinen kerta melko hyvin ja keskimäärin joka toinen kerta melko huonosti sen mukaan, kuinka hyvä arpaonni rivillä 2 kulloinkin oli. Huonot jaot kasvattavat rekursiotasojen määrää, mutta ei kovin moninkertaiseksi verrattuna ihanteellisiin jakoihin. Siksi keskimääräinen ajan kulutus on $\Theta(n \log n)$. Huonoimman tapauksen ajan kulutus on $\Theta(n^2)$, mutta huono tapaus toteutuu vain jos rivin 2 arvonnoissa on uskomattoman huono onni.

Pikajärjestämisen nopeus perustuu suurelta osalta siihen, että toisin kuin keko-

järjestäminen, se pysyttelee yhden osataulukon alueella kunnes on saanut sen valmiiksi, ja vasta sitten siirtyy seuraavaan osataulukkoon. Kekojärjestäminen siirtyy vähän väliä taulukon alkiosta kauas toiseen alkioon siirtymällä kohdasta i kohtiin $2i+1$ ja $2i+2$. Tähän ominaisuuteen, joka pikajärjestämisellä on mutta kekojärjestämisellä ei ole, viitataan käsitteellä ***paikallisuus*** (***locality***). Paikallisuus tarkoittaa, että ohjelma käsittelee pitkiä aikoja enimmäkseen yhdellä tai muutamalla pienellä muistialueella sijaitsevaa tietoa, sen sijaan että käsittelisi toistuvasti hujan hajan siellä täällä isossa muistissa sijaitsevaa tietoa. Syy siihen, että paikallisuus nopeuttaa ohjelmaa, on seuraava.

Tietokoneissa on varsinaisen suuren muistin lisäksi paljon pienempi mutta nopeampi muisti, jota kutsutaan ***välimuistiksi*** (***cache***). Välimuisteja saattaa olla useita kerroksia. Siltä osin kuin ohjelman käsittelemä tieto mahtuu välimuistiin, toimii ohjelma nopeammin kuin silloin kun joudutaan käyttämään muutakin muistia. Pikajärjestäminen hyötyy tästä niin kauan kun osataulukko mahtuu välimuistiin — sillä varauksella, että avaimet sijaitsevat alkioissa sellaisinaan, eivätkä osoittimien tai viitteiden takana. Uusissa oliokielissä on tavallista, että tämä oletus ei toteudu. Siksi C++:n `sort` perustuu tyypillisesti pikajärjestämiseen, mutta Javan `sort` ei perustu. Tähän palataan sivulla 101.

Järjestettävä taulukko on rekursiivisille kutsuille yhteinen. Kukin rekursiivinen kutsu tarvitsee $\Theta(1)$ omaa muistia. Samalla rekursiotasolla olevat kutsut eivät tarvitse omaa muistia yhtäaikaa, koska ne eivät ole suorituksessa yhtäaikaa. Niinpä rekursiiviset kutsut tarvitsevat omaa muistia yhteensä rekursiotasojen määrään verrannollisesti. Siksi QUICKSORT tarvitsee muistia järjestettävän taulukon lisäksi keskimäärin $\Theta(\log n)$ ja huonoimmillaan $\Theta(n)$.

89. QUICKSORT ei ole vakaa. Anna esimerkki taulukosta, jolla QUICKSORT vaihtaa kaksi yhtäsuurta alkiota keskenään!

90. Muuttuisiko QUICKSORT vakaaksi, jos SWAP muutettaisiin siten, että se ei vaihda yhtäsuuria alkioita keskenään? Perustele.

91. Kirjoita pseudo- tai ohjelmakoodina muunnettu pikajärjestäminen QS3, joka osittaessaan selaa osataulukon $A[a \ldots y]$ vasemmalta alkaen ja jakaa sen kolmeen osaan siten että alkuosan alkiot ovat pienempiä kuin jakoalkio $A[\,\textsc{Random}(a,y)\,]$, keskiosan alkiot yhtäsuuria ja loppuosan alkiot suurempia! Osituksen ajan kulutuksen pitää olla $O(n)$ ja lisämuistin tarpeen $O(1)$. Piirä kuva, joka näyttää taulukon osat ja niiden rajat osituksen suorituksen aikana!

92. Kuvassa 39 on pikajärjestäminen toteutettu funktionaalisella ohjelmointikielellä nimeltä Haskell. Jos syötteeksi saatiin tyhjä lista, niin rivi 2 palauttaa tyhjän listan. Rivi 5 poimii listasta `xs` ne alkiot, jotka ovat pienemmät kuin `p`, ja palauttaa tuloksen listana `lesser`. Rivi 6 tuottaa vastaavasti `xs`:stä vähintään `p`:n suuruisten alkioiden listan `greater`. Jos syötteeksi saatiin epätyhjä lista, niin rivi 3 jakaa sen ensimmäiseen alkioon `p` ja loppuosaan `xs`, ja palauttaa listan, jossa on aluksi

```
1  quicksort :: Ord a => [a] -> [a]
2  quicksort []     = []
3  quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
4      where
5          lesser  = filter (< p) xs
6          greater = filter (>= p) xs
```

Kuva 39: Elegantti mutta ei kovin laadukas pikajärjestäminen Haskellilla, haettu 2.2.2024 `https://wiki.haskell.org/Introduction#Quicksort_in_Haskell`

`lesser` järjestettynä `quicksort`:illa, sitten alkio `p` ja lopuksi `greater` järjestettynä `quicksort`:illa. Mitä heikkouksia tällä toteutuksella on?

5.3 Tehostuskeinoja

Pienillä taulukoilla INSERTIONSORT on yksinkertaisuutensa ansiosta nopeampi kuin QUICKSORT. Tätä voidaan käyttää hyväksi QUICKSORT:in nopeuttamiseen siten, että jätetään enintään jonkin kokoiset osataulukot järjestämättä ja suoritetaan lopuksi INSERTIONSORT. Tämä on näytetty kuvassa 40. Kuvassa rajana on 50.

Kuvan oikealla puolella on ***kääre*** (***wrapper function***), eli aliohjelma, joka tarjoaa halutun rajapinnan ja jonka kautta pääosan työstä tekeviä aliohjelmia kutsutaan. Kääre voi myös muun muassa huolehtia mahdollisista erikoistapauksista sekä varata ja vapauttaa muistia pääosan työstä tekeviä aliohjelmia varten. Tässä tapauksessa kääre ensin käsittelee pienimmät taulukot erikoistapauksena, jotta rivillä 2 oleva $A.koko - 1$ ei aiheuttaisi ylivuotoa tyhjällä taulukolla etumerkittömillä kokonaisluvuilla. Ylivuodon estämiseksi riittäisi testata $koko < 1$ tai C++:n `vector`:eilla `A.empty()`, mutta samalla kertaa saa helposti käsiteltyä erikoistapauksena myös yhden alkion taulukot. Näiden kahden vaihtoehdon erolla on käytännössä erittäin vähän merkitystä.

Sitten kääre kutsuu pääosan työstä tekevää QUICKSORT:ia niin että rajoina

```
   QUICKSORT(&A, a, y)
1  while y − a ≥ 50 do
     ... kuvan 38 rivit 2, ..., 9
10   if i − a ≤ y − j then
11     if i > a then QUICKSORT(A, a, i − 1)
12     a := j + 1
13   else
14     if j < y then QUICKSORT(A, j + 1, y)
15     y := i − 1
```

```
   QUICKSORT(&A)
1  if A.koko < 2 then return
2  QUICKSORT(A, 0, A.koko − 1)
3  INSERTIONSORT(A)
```

Kuva 40: Kaksi tehostusta pikajärjestämiselle ja vastaava alkuperäinen kutsu

ovat alkuperäisen taulukon rajat, ja kutsuu lopuksi INSERTIONSORT:ia. INSERTIONSORT:in kutsuminen vain kerran lopuksi eikä kullekin QUICKSORT:in järjestämättä jättämälle osataulukolle erikseen nopeuttaa ohjelmaa, koska siten vältetään paljon aliohjelmien käynnistämisiä. Se myös selkeyttää ohjelmakoodia. On hyvin helppo nähdä että lopussa A on järjestyksessä, sillä siihen riittää, että INSERTIONSORT toimii itse oikein ja että rivin 3 alkuun päästään ja silloin A:ssa on oikeat alkiot missä tahansa järjestyksessä.

Isompi kysymys on, miksi INSERTIONSORT:in kutsu ei pilaa ohjelman nopeutta, sillä INSERTIONSORT käyttää keskimäärin $\Theta(n^2)$ aikaa. Syy on siinä, että QUICKSORT:in ansiosta jokaisessa järjestämättä jätetyssä osataulukossa on siihen lopullisesti kuuluvat alkiot, ne vain eivät välttämättä ole keskenään oikeassa järjestyksessä. Siksi INSERTIONSORT siirtää kutakin alkiota korkeintaan osataulukon koon verran miinus yksi. Koolla on kiinteä yläraja (vaikka 50), joten tässä tapauksessa INSERTIONSORT:in suoritusaika on $\Theta(n)$.

Toinen parannus kuvassa 40 on toisen rekursiokutsun korvaaminen silmukalla. Osituksen tuottamista osataulukoista pienempi järjestetään rekursiivisella kutsulla, ja isompi palaamalla silmukassa takaisin QUICKSORT:in alkuun. Siksi kullakin rekursiotasolla osataulukon koko on enintään puolet edellisestä, joten rekursiotasoja on enintään noin $\log_2 n$. Ajan kulutukseen tämä ei paljoa vaikuta, koska poistetun rekursiivisen kutsun tekemä työ ei ole kadonnut minnekään vaan on siirtynyt nykyisen kutsun tehtäväksi. Mutta huonoimman tapauksen muistin käyttöön sillä on suuri vaikutus. Ilman tätä muutosta huonoimman tapauksen lisämuistin käyttö on $\Theta(n)$, koska jokainen rekursiotaso tarvitsee $\Theta(1)$ omaa muistia, ja tasoja voi olla melkein n kappaletta. Muutoksen ansiosta huonoimman tapauksen muistin lisäkäyttö on $\Theta(\log n)$.

Kuvassa 41 on suoritusaikoja muutamalle pikajärjestämisen versiolle ja muutamalle muulle järjestämisalgoritmille. Mittausolosuhteet olivat samat kuin sivulla 47, paitsi että yli 3 minuuttia vieneitä ajoja ei toistettu. Vakaat algoritmit on merkitty symbolilla *. Koska vakaus on lisävaatimus ja lisävaatimuksilla on tapana nostaa hintaa, kannattaa hyväksyä, että kaikkein nopeimmat algoritmit eivät ehkä ole vakaita.

INSERTIONSORT on kuvan 41 algoritmeista ainoa, joka käyttää aikaa keskimäärin $\Theta(n^2)$. Kuvasta näkyy, että se on selvästi muita hitaampi. Kaksi mittausta sille jätettiin tekemättä, koska ne olisivat kestäneet kovin kauan.

HEAPSORT esiteltiin luvussa 3.6 ja MERGESORT esitellään luvussa 7.1. Pienet erot HEAPSORT:in ajan kulutuksessa kuvaan 22 verrattuna johtuvat luultavasti kohinasta (sivu 48). C++ `stable_sort` käyttää vakaata nopeaa algoritmia, josta standardi ei kerro mikä se on. Standardi lupaa ajan kulutukseksi $O(n \log n)$ jos muistia on paljon, ja $O(n(\log n)^2)$ jos muistia on niukasti. Siksi on syytä uskoa, että useimmissa toteutuksissa `stable_sort` pohjautuu kahteen MERGE-

	0...4 999		10 000...19 999		10^6...1 000 019	
INSERTIONSORT *	6,1 s	5 min 35 s	5 min 36 s	-	48 min	-
HEAPSORT	0,6 s	5,5 s	8,9 s	1 min 23 s	1,8 s	35 s
rekursiivinen QUICKSORT	0,6 s	4,1 s	8,6 s	59 s	1,5 s	16 s
silmukka-QUICKSORT	0,6 s	4,1 s	8,5 s	59 s	1,5 s	16 s
MERGESORT *	0,5 s	7,2 s	7,5 s	2 min 18 s	1,5 s	27 s
C++ `stable_sort` *	0,5 s	5,2 s	6,8 s	1 min 25 s	1,4 s	27 s
QUICKSORT-16	0,5 s	4,6 s	6,6 s	1 min 06 s	1,2 s	17 s
QUICKSORT-50	0,4 s	6,4 s	6,0 s	1 min 27 s	1,1 s	20 s
C++ `sort`	0,4 s	4,7 s	6,0 s	1 min 06 s	1,1 s	17 s
RADIXSORT *	0,1 s	4,6 s	1,6 s	1 min 16 s	0,3 s	12 s
COUNTINGSORT *	0,04 s	3,6 s	0,9 s	54 s	0,4 s	10 s
aputaul. vakautettu `sort` *	0,7 s	2,3 s	9,7 s	34 s	2,4 s	13 s
aputaul. `stable_sort` *	0,6 s	2,2 s	8,7 s	32 s	2,2 s	14 s
aputaulukko-QUICK-50	0,5 s	2,1 s	7,6 s	31 s	2,1 s	13 s
aputaulukko C++ `sort`	0,5 s	2,1 s	7,7 s	30 s	2,0 s	12 s

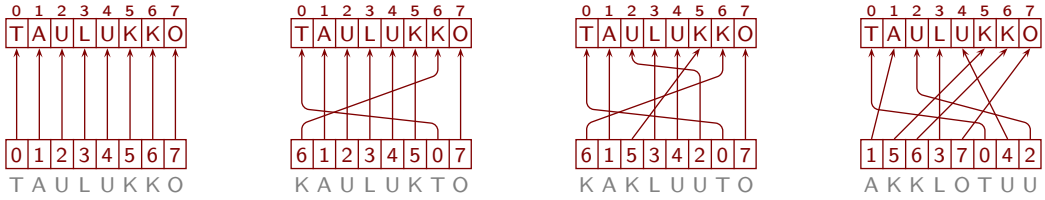
Kuva 41: Järjestämisalgoritmien suoritusaikaesimerkkejä

SORT:in muunnelmaan, joista se valitsee jommankumman sen mukaan, kuinka paljon muistia on käytettävissä.

Rekursiivinen QUICKSORT vastasi kuvaa 38 sovitettuna `unsigned`-tyypille, ja silmukka-QUICKSORT:issa toinen rekursiivisista kutsuista oli korvattu silmukalla kuten kuvassa 40. Näiden kahden vaihtoehdon nopeuserot olivat pienet. Tämä viittaa siihen, että rekursiivisessa QUICKSORT:issa rekursiotasojen määrä pysyi melkein aina pienenä.

QUICKSORT-50 oli silmukka-QUICKSORT:in muunnos, joka jättää enintään 50 alkion kokoiset osataulukot järjestämättä ja lopuksi suorittaa INSERTIONSORT:in. Raja 50 valittiin kokeilemalla siten, että suoritusaika tapauksessa 10 000 ... 19 999 olisi mahdollisimman pieni kun kussakin alkiossa on vain avain. Rajaksi valittiin pyöreä luku, koska suoritusaika vaihteli sen lähiympäristössä niin vähän, että minimikohdan tarkempi määrittäminen olisi ollut kohinan vuoksi työlästä. Samaa algoritmia kokeiltiin myös niin, että rajana oli 16. QUICKSORT-50 ja QUICKSORT-16 olivat silmukka-QUICKSORT:ia nopeammat kun alkiossa oli vain avain, mutta hitaammat kun alkiossa oli 400 tavua muutakin dataa.

C++-standardi ei määrittele, minkä algoritmin `sort` toteuttaa. Wikipedian sivun ”Introsort” mukaan g++ käyttää yhdistelmäalgoritmia, joka toimii melkein, mutta ei täysin samoin kuin QUICKSORT-15. Tärkein ero on, että rekursiotasojen määrän ylittäessä suunnilleen $2\log_2 n$ siirtyy C++:n `sort` käyttämään kekojärjestämistä (heapsort). Siten se varmistaa, että suoritusaika ja lisämuistin kulutus ovat huonoimmassakin tapauksessa $\Theta(n \log n)$. Todellinen toteutus saatiin näkyville komennolla `g++ -E`. Sekin viittaa introsort-algoritmiin, mutta on monimut-



Kuva 42: Aputaulukon järjestäminen

kaisempi kuin Wikipedian kuvaus, ja rajana on 16 eikä 15. (QUICKSORT-16:n rajaksi valittiin 16 siksi, että se on sama kuin C++:n `sort`:issa.)

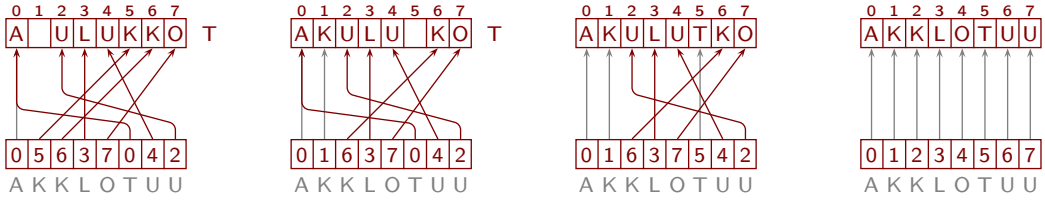
Kun alkiossa oli vain avain, nopeimmat yleiskäyttöiset algoritmit olivat C++:n `sort` ja QUICKSORT-50. Mitä suurempia alkiot ovat, sitä enemmän alkion kopioimiseen kuluu aikaa. Kaikkein pienimpiä taulukoita lukuun ottamatta QUICKSORT kopioi vähemmän alkioita mutta käyttää enemmän aikaa kunkin kopioitavan alkion valitsemiseen kuin INSERTIONSORT. Siksi mitä suurempia alkiot ovat, sitä pienempi on se raja, jota pienemmät osataulukot kannattaa jättää INSERTIONSORT:in järjestettäviksi. Raja 50 oli sopiva kun alkiossa oli vain avain, mutta aivan liian suuri kun alkiossa oli lisäksi 400 tavua muuta dataa. Niillä raja 16 oli parempi, mutta silmukka-QUICKSORT vielä parempi.

Kuva 41 havainnollistaa, että muutos, joka nopeuttaa algoritmia yhdenlaisilla syötteillä saattaa hidastaa sitä toisenlaisilla. Tästä syystä nopeuden maksimointi on vaikeaa. (Muitakin syitä on, kuten riippuvuus tietokoneesta, jossa ohjelma ajetaan.) Toisaalta nopeuden kovin pitkälle menevä maksimointi on harvoin tarpeen. Kuvassa 41 nopeusero INSERTIONSORT:in ja muiden algoritmien välillä on valtava ja usein käytännön kannalta olennainen, mutta muiden algoritmien väliset nopeuserot ovat niin pienet, että niillä on käytännön merkitystä vain kun nopeus on poikkeuksellisen tärkeää. Tämä on esimerkki siitä, että riittävän nopean algoritmin valitsemisessa kannattaa tyypillisesti (mutta ei aina!) ottaa huomioon Θ-merkinnällä ilmaistu ajan kulutus, ja vain se.

Kun alkiot ovat isoja, voidaan järjestämistä nopeuttaa ottamalla käyttöön aputaulukko $N[0 \ldots n-1]$, joka alustetaan **for** $i := 0$ **to** $n-1$ **do** $N[i] := i$ ja järjestetään siten, että vertailujen $N[i] < N[j]$, $N[i] \leq N[j]$ ja niin edelleen tilalla on $A[N[i]].x < A[N[j]].x$, $A[N[i]].x \leq A[N[j]].x$ ja niin edelleen. (Vaihtoehtoisesti N:ssä voi olla osoittimia A:n alkioihin.) Tätä on havainnollistettu kuvassa 42. Nyt $A[N[0]].x \leq A[N[1]].x \leq \ldots \leq A[N[n-1]].x$, joten paikkaan 0 kuuluu alkio $A[N[0]]$, paikkaan 1 kuuluu $A[N[1]]$ ja niin edelleen. Lopuksi A:n alkiot siirretään N:n perusteella suoraan oikeisiin paikkoihinsa mahdollisimman vähillä kopioinneilla pian kerrottavalla tavalla.

93. Jos $A = [\mathsf{C}, \mathsf{B}, \mathsf{E}, \mathsf{F}, \mathsf{A}, \mathsf{D}]$, niin mikä on N?

Taulukon A alkioiden kopiointien määrä saadaan näin minimoitua. Hintana



Kuva 43: Alkioiden siirto lopullisille paikoilleen aputaulukon avulla

on, että vertailut muuttuvat hitaammiksi, koska vertaamista varten joudutaan katsomaan indeksin tai osoittimen taakse. Lisäksi tyypillisesti menetetään paikallisuuden tuoma hyöty, koska aputaulukossa lähekkäin toisiaan olevat alkiot ovat varsinaisessa taulukossa kaukana toisistaan. (Aputaulukon alustaminen luvuilla $0, \ldots, n-1$ ei lisää ajan kulutusta olennaisesti.) Koska aputaulukon alkiot ovat pelkkiä kokonaislukuja tai osoittimia, kuluttaa aputaulukko vähän muistia varsinaiseen taulukkoon verrattuna. Niinpä aputaulukon tarvitsema muisti on harvoin ongelma, vaikka se onkin $\Theta(n)$.

Kuvassa 41 tätä keinoa käytettiin niin että aputaulukko järjestettiin kolmella edellä mainitulla algoritmilla sekä C++:n `sort`:illa, joka oli muutettu vakaaksi kysymyksen 94 vastauksessa esitetyllä keinolla. Niin saadut neljä ohjelmaa olivat isoilla alkioilla melkein aina kaikkia muita kuvan ohjelmia selvästi nopeammat. Vakaat kaksi ohjelmaa olivat niiden keskuudessa hitaammat kuin muut kaksi.

94. Miten aputaulukkoa käyttävä järjestäminen saadaan vakaaksi, vaikka varsinainen järjestämisalgoritmi ei olisi vakaa?

Tarkastelemme vielä taulukon alkioiden siirtämistä järjestetyn aputaulukon N avulla oikeille paikoilleen.

95. Mikä on lopputulos, jos taulukosta $A = [\mathsf{C}, \mathsf{B}, \mathsf{D}, \mathsf{A}]$ vaihdetaan $A[0]$ ja $A[N[0]]$ päikseen, sitten $A[1]$ ja $A[N[1]]$ ja niin edelleen? Onko se oikein?

Jotta alkioita kopioitaisiin mahdollisimman vähän, kannattaa toimia tavalla, jota on havainnollistettu kuvassa 43. Otetaan jokin väärässä paikassa oleva alkio sivuun. Siirretään niin syntyneeseen aukkoon se alkio, joka kuuluu siihen paikkaan. Samalla aukko siirtyy siirretyn alkion aikaisempaan paikkaan. Sitten toistuvasti siirretään aukon uuteen paikkaan alkio, joka kuuluu siihen paikkaan, kunnes aukko on siinä paikassa, johon sivuun otettu alkio kuuluu. Lopuksi sivuun otettu alkio laitetaan aukkoon.

96. Kirjoita välivaiheet, kun tämä tehdään taulukolle $[\mathsf{C}, \mathsf{B}, \mathsf{E}, \mathsf{F}, \mathsf{A}, \mathsf{D}]$ aloittaen paikasta 0!

97. Miksi aukko päätyy lopulta siihen paikkaan, johon sivuun otettu alkio kuuluu?

98. Edellä kuvattu kierros ei välttämättä siirrä jokaista alkiota oikeaan paikkaansa. Tämä on tietenkin helppo ratkaista lisäämällä silmukka, joka käy jokaisen alkion läpi ja aloittaa uuden kierroksen niille, jotka eivät jo ole oikeissa paikoissaan.

Miten voidaan helposti varmistaa, että uutta kierrosta ei aloiteta mistään sellaisesta paikasta, jonne jokin aikaisempi kierros on jo siirtänyt oikean alkion?

99. Kirjoita pseudo- tai ohjelmakoodi, joka siirtää A:n alkiot oikeisiin paikkoihinsa edellä kuvatulla tavalla!

100. Vastauksen 99 ulompi silmukka voidaan perustella oikein toimivaksi seuraavan invariantin avulla: kuhunkin paikkaan h, missä $0 \leq h < n$, kuuluu $A[N[h]]$; ja $A[0], \ldots, A[i-1]$ ovat oikeissa paikoissa. Perustele, että tämä invariantti on voimassa, kun riville 1 tullaan ensimmäisen kerran.

101. Perustele, että kun **for**-silmukka lopettaa, on jokainen alkio siellä missä sen kuuluukin olla.

5.4 Nopea etsiminen järjestysluvun perusteella

QUICKSORT:in pohjalta on helppo suunnitella algoritmi QUICKSELECT$(\&A, k)$, joka toimii keskimäärin ajassa $\Theta(n)$ ja palauttaa A:n sen alkion, joka on suuruusjärjestyksessä $(k+1)$:s. Jos samansuuruisia alkioita on monta, niin QUICKSELECT saa palauttaa niistä minkä tahansa. Jos $k = 0$, niin se palauttaa A:n jonkin pienimmän alkion. Jos A:ssa on kaksi yhtäsuurta, muita pienempää alkiota, niin se palauttaa jommankumman niistä myös kun $k = 1$. Jos $k = n-1$, niin se palauttaa A:n jonkin suurimman alkion. Jos $k = \lfloor \frac{n-1}{2} \rfloor$, niin se palauttaa mediaanin. Jos k ei ole väliltä $0, \ldots, n-1$, niin se palauttaa virhe. Sen huonoimman tapauksen ajan kulutus on $\Theta(n^2)$, mutta huono tapaus esiintyy hyvin harvoin, jos QUICKSELECT on laadukkaasti toteutettu.

QUICKSELECT$(\&A, k)$ tarvitsee A:n lisäksi $\Theta(1)$ muistia, ja voi muuttaa kutsujan taulukon järjestystä. Jos kutsujan taulukon järjestystä ei saa muuttaa, niin riittää vaihtaa A:n välitysmekanismi. QUICKSELECT(A, k) saa A:sta kopion, joten se ei muuta kutsujan taulukon järjestystä. Samasta syystä sen lisämuistin tarve on $\Theta(n)$.

102. Kirjoita QUICKSELECT pseudo- tai ohjelmakoodina!

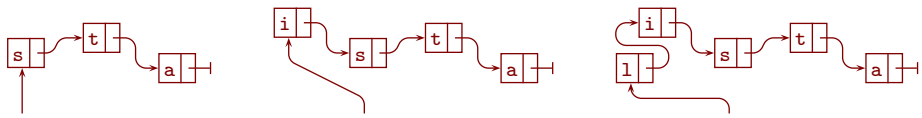
Tunnetaan algoritmi, joka tarjoaa saman palvelun kuin QUICKSELECT, mutta toimii (hitaimmillaankin) ajassa $O(n)$. Sitä ei kuitenkaan juurikaan käytetä, koska se on monimutkainen ja melkein aina käytännössä hitaampi kuin QUICKSELECT.

103. Suunnittele algoritmi, joka palauttaa suuruusjärjestyksessä k:nnen alkion keskimäärin ajassa $O(n)$ siten, että toimintansa nopeuttamiseksi se hyödyntää aikaisemmin tekemäänsä järjestämistyötä. Sen käyttöön annetaan taulukko $A[0 \ldots n-1]$, joka sisältää alkiot tuntemattomassa järjestyksessä. Vain algoritmi itse saa muuttaa A:n sisältöä. Se saa käyttää lisämuistia yhden bitin alkiota kohti plus $\Theta(1)$. Ei tarvitse antaa pseudo- eikä ohjelmakoodia, vaan riittää kuvailla algoritmi sanallisesti.

6 Hieman linkitetyistä listoista

Yksisuuntainen ***linkitetty lista*** (***linked list***) tarkoittaa tietorakennetta, jossa jokainen alkio sisältää tiedon, missä seuraava alkio sijaitsee. Tämä tieto eli linkki voi olla esitetty osoittimena, ei-negatiivisena kokonaislukuna, iteraattorina ja ehkä muullakin tavalla. Olemme jo nähneet yksisuuntaisia linkitettyjä listoja kuvassa 18 (a). Kaksisuuntaisessa linkitetyssä listassa jokainen alkio sisältää myös tiedon, missä edellinen alkio sijaitsee.

Perusmuotoisen linkitetyn listan alkuun voi milloin tahansa lisätä alkion tehokkaasti. Jos lista on epätyhjä, niin siitä voi ottaa ensimmäisen alkion pois tehokkaasti. Jos listaa käsitellään vain sen alusta käsin, niin ensimmäinen alkio on jäljellä olevista alkioista se, joka on ollut listassa vähiten aikaa. ***Pino*** (***stack***) on tietorakenne, joka toteuttaa nämä palvelut. Siksi linkitetyllä listalla on helppo toteuttaa pino tehokkaasti. Alla on näytetty kaksi lisäämistä linkitetyn listan alkuun.



Alkioille sopivan tietotyypin voi C++:lla määritellä esimerkiksi seuraavasti:

```
1  struct alkio{
2    char mrk; alkio *seur;
3    alkio( char mrk ): mrk(mrk), seur(0) {}
4  };
```

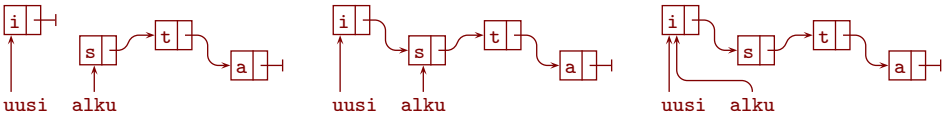
Kentän `seur` nimen edessä oleva `*` tekee kentästä ***osoittimen*** (***pointer***) alkioon, eli muuttujan, joka ei sisällä alkiota vaan tiedon, missä alkio sijaitsee. Rivillä 3 on niin sanottu ***rakentaja*** (***constructor***). Se suoritetaan aina automaattisesti alkion luonnin yhteydessä. Tässä esimerkissä se sijoittaa saamansa parametrin alkion `mrk`-kenttään, ja `seur`-kenttään osoittimen ei minnekään (***null pointer***). Rakentajan parametrilistan ei ole pakko olla esimerkin mukainen, vaan sen saa suunnitella kuten minkä tahansa aliohjelman parametrilistan. Rivin lopussa olevien `{` ja `}` väliin voi sijoittaa suoritettavaa koodia kuten mihin tahansa C++:n aliohjelmaan. Kaksoispisteen ja `{`:n välissä oleva osuus asettaa kentille alkuarvoja. Sen saa jättää kokonaan tai osittain pois. Jos sen jättää kokonaan pois, niin myös kaksoispiste pitää jättää pois.

Jos alkiot on määritelty kuten edellä, niin alkion voi lisätä listan alkuun näin:

```
5  alkio *uusi = new alkio( 'i' );
6  uusi->seur = alku; alku = uusi;
```

Rivillä 5 luodaan uusi alkio ja varataan sille muistia. Argumentti `'i'` menee alkion rakentajalle. Tässä tapauksessa rakentaja sijoittaa sen uuden alkion `mrk`-kenttään.

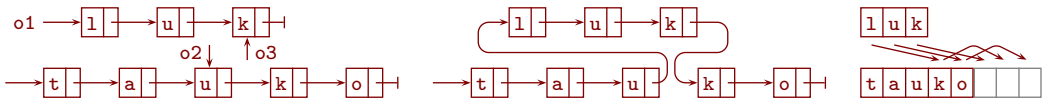
Jos aloitetaan edellä vasemmalla olleesta tilanteesta, niin rivin 5 lopussa tilanne on kuten alla vasemmalla, rivin 6 keskellä kuten alla keskellä ja rivin 6 lopussa kuten alla oikealla:



Alla oleva koodinpätkä tulostaa ensimmäisen alkion merkin ja poistaa alkion listasta sekä palauttaa alkiolle varatun muistin käyttöjärjestelmälle, tai tulostaa tiedon, että listassa ei ollut yhtään alkiota:

```
if( alku ){
  std::cout << alku->mrk << "\n";
  alkio *poistettava = alku; alku = alku->seur; delete poistettava;
}else{
  std::cout << "Lista oli tyhjä.\n";
}
```

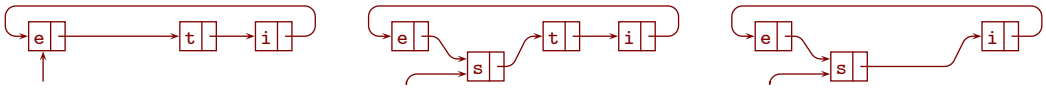
Linkitettyyn listaan voi lisätä alkion vakioajassa minne tahansa, kunhan ensin on löydetty kohta jonka perään lisäys tehdään. Vakioajassa voi lisätä jopa toisen listan, jos käytettävissä on myös osoitin sen loppuun. Alla olevan kuvan havainnollistaman lisäyksen toteuttamiseksi riittää suorittaa `o3->seur = o2->seur;` `o2->seur = o1;`. Tämä on suuri etu verrattuna taulukoihin. Kuvan oikea puoli havainnollistaa, että kun lisätään taulukon sisältö keskelle toista taulukkoa, joudutaan jokainen lisättävä alkio kopioimaan määränpäätaulukkoon, ja lisäyskohdasta alkaen jokainen määränpäätaulukon alkio joudutaan kopioimaan kohti määränpäätaulukon loppua.



Myös poistaminen mistä tahansa kohdasta onnistuu vakioajassa, mutta yksisuuntaisen linkitetyn listan tapauksessa siihen ei riitä, että käytettävissä on osoitin poistettavaan alkioon. Sen sijaan tarvitaan osoitin poistettavan alkion eteen. Se voi olla vaikea löytää tehokkaasti, koska edessä oleva alkio on voinut vaihtua sen jälkeen kun poistettavaa alkiota käsiteltiin edellisen kerran. Kaksisuuntaisesta linkitetystä listasta voi poistaa vakioajassa siinäkin tapauksessa, että osoitin on poistettavaan alkioon. Myös osalistan voi poistaa vakioajassa, jos lisäksi on osoitin viimeiseen poistettavaan alkioon.

Jonon voi toteuttaa tehokkaasti ***rengaslistalla*** (***circular linked list***) eli linkitetyllä listalla, jonka viimeisen alkion osoitin osoittaa ensimmäiseen alkioon, ja ulkoapäin osoittava osoitin osoittaa viimeiseen alkioon. Alla on näytetty lisääminen

epätyhjään rengaslistalla toteutettuun jonoon, ja sen jälkeen poistaminen siitä. Esimerkin jonosta poistuu ensin `t`, sitten `i`, sitten `e` ja sitten `s` (kuvassa on näytetty vain `t`:n poistaminen).



Kun uusi alkio on luotu, se voidaan lisätä jonoon C++:lla näin:

```
void lisaa( alkio *uusi ){
  if( !viim ){ uusi->seur = uusi; viim = uusi; }
  else{ uusi->seur = viim->seur; viim->seur = uusi; viim = uusi; }
}
```

104. Kirjoita rengaslistana toteutetusta jonosta poistaminen C++:lla aliohjelmana, joka palauttaa osoittimen poistettuun alkioon! Jos jono oli tyhjä, se palauttaa osoittimen ei minnekään.

105. ***Pakka*** (***deque***) on pinon ja jonon yleistys, johon voi lisätä kumpaan päähän tahansa ja poistaa kummasta päästä tahansa. Miten rengaslistalla voidaan toteuttaa kolme näistä neljästä toiminnosta ajassa $\Theta(1)$?

Kaikki neljä pakan toimintoa saadaan kaksisuuntaisella linkitetyllä listalla ajassa $\Theta(1)$. Rengaspuskurilla ne saadaan tasatussa ajassa $\Theta(1)$. Rengaspuskurille varatun taulukon täyttyessä saatetaan joutua varaamaan isompi taulukko ja siirtämään alkiot sinne.

106. Mihin alkioihin rengaslistojen tavanomaisten osoittimien lisäksi tarvitsee olla osoittimet, jotta rengaslistaan voisi lisätä haluttuun kohtaan toisen rengaslistan ajassa $\Theta(1)$? Käytä esimerkkinä rengaslistan, jonka sisältö on D E F, lisäämistä rengaslistaan A B C G H siten, että lopputulos on A B C D E F G H. Esimerkissä alkion A osoitin osoittaa alkioon B ja niin edelleen, ja ulkoa päin osoittavat osoittimet osoittavat alkioita F ja H.

107. Mihin alkioihin rengaslistojen tavanomaisten osoittimien lisäksi tarvitsee olla osoittimet, jotta rengaslistasta voisi poistaa halutun yhtenäisen osan ajassa $\Theta(1)$? Poistettu osa pitää palauttaa rengaslistana. Poistotoiminto saa luottaa siihen, että se alkio ei kuulu poistettaviin, johon alkuperäisen rengaslistan ulkoapäin osoittava osoitin osoittaa. Käytä esimerkkinä osan B C D E poistamista rengaslistasta A B C D E F G H.

Linkitettyjen listojen suurin heikkous on, että alkioihin pääsee käsiksi vain selaamalla listaa tai käyttämällä osoitinta tai muuta sellaista, johon alkion sijainti on tallennettu. Tämä vähentää hyötyä, joka saadaan siitä, että sen jälkeen kun oikeat alkiot on löydetty, voi linkitettyihin listoihin lisätä keskelle ja poistaa keskeltä tehokkaasti. Esimerkiksi kekojärjestäminen olisi linkitetyille listoille erittäin hidas,

koska se vaatii kykyä siirtyä kohdasta i kohtiin $2i+1$ ja $2i+2$. Pinon ja jonon voi toteuttaa taulukoillakin. Siksi linkitetyille listoille itsenäisenä tietorakenteena on niukasti käyttöä.

Toisaalta linkkejä voi käyttää muillakin tavoilla kuin rakentamaan listoja. Luvussa 9 niillä toteutetaan puita, joilla voi tehdä monenlaisia asioita tehokkaasti. Linkkejä voi käyttää myös apuvälineenä monimutkaisemmissa kokonaisuuksissa, joissa käytetään myös taulukoita. Kun linkitettyyn rakenteeseen lisätään tai poistetaan alkioita tai alkioiden järjestystä muutetaan, ei alkioiden sijaintia muistissa tarvitse muuttaa. Taulukoilla ei ole tätä etua. Siksi kun luvussa 3.7 haluttiin mahdollistaa alkion prioriteetin muuttaminen alkion ollessa kekoon perustuvassa prioriteettijonossa, ongelmaksi tuli alkion löytäminen keosta. Ratkaisuna oli jossain määrin monimutkainen rakenne, jossa keossa oli vain linkkejä alkioihin, ja alkion tiedoissa oli linkki alkion paikkaan keossa (kuva 24).

Tällaisissa sovelluksissa linkkeinä voi olla helpompi käyttää ei-negatiivisia kokonaislukuja kuin osoittimia. Luvussa 4 kukin maantieverkon solmu esitettiin tietueella, jossa oli muun ohessa `unsigned`-tyyppinen muuttuja `reitti` kertomassa edellisen solmun jossakin lyhimmässä jo löydetyssä lähdöstä solmuun vievässä reitissä. Vaikka `reitti`-muuttujat eivät olleet osoitintyyppisiä, ne voi silti ajatella linkeiksi, joiden avulla reitti on esitetty linkitettynä listana takaperin.

108. Kirjoita tehokas $\Theta(1)$ lisämuistia käyttävä ohjelmanpätkä, joka kääntää kuvan 29 algoritmin löytämän reitin etuperin!

Linkitetyn listan alkioiden maksimimäärää ei tarvitse tietää etukäteen. Aikoinaan se oli suuri etu taulukoihin nähden, mutta sen jälkeen kun kasvavat taulukot tulivat ohjelmointikieliin, ei taulukonkaan alkioiden maksimimäärää ole tarvinnut tietää etukäteen.

Jos linkkeinä käytetään osoittimia, niin luonteva tapa ilmaista listan loppuminen on osoitin ei minnekään. Jos linkkeinä käytetään lukuja ja jos nolla ei voi olla minkään alkion numero, niin se sopii tarkoittamaan listan loppumista. Kuvassa 24 jätettiin keon lokero 0 tarkoituksella käyttämättä juuri siksi, että linkki 0 voisi tarkoittaa, että linkin häntäpään alkio ei ole keossa. Jos nolla voi olla alkion numero, niin loppumerkkinä voi käyttää `int`-tyypillä lukua -1 ja `unsigned`-tyypillä suurinta tyyppiin mahtuvaa kokonaislukua. Kuten sivulla 41 todettiin, jälkimmäisen voi kirjoittaa C++:ssa `~0u`. Jos parempaa vaihtoehtoa ei ole käytettävissä, niin sekä osoittimien että lukujen tapauksessa voi laittaa listan viimeisen alkion linkin osoittamaan alkioon itseensä.

Linkitetyistä listoista on olemassa liian monta muunnelmaa tässä järjestelmällisesti läpikäytäväksi. Tyydymme esittelemään enää yhden hyödyllisen niksin. Kuva 44 havainnollistaa ongelmaa, jota niksi lievittää (ja samalla hieman johdattelee lukuun 7.1). Siinä näytetty aliohjelma yhdistää kaksi kasvavassa järjestyksessä olevaa, mahdollisesti tyhjää linkitettyä listaa yhdeksi kasvavassa järjestyksessä

```
1  alkio *lomita( alkio *eka, alkio *toka ){
2    if( !eka ){ return toka; }
3    if( !toka ){ return eka; }
4    alkio *tulos = 0;
5    if( eka->x <= toka->x ){ tulos = eka; eka = eka->seur; }
6    else{ tulos = toka; toka = toka->seur; }
7    alkio *os = tulos;
8    while( eka && toka ){
9      if( eka->x <= toka->x ){ os->seur = eka; eka = eka->seur; }
10     else{ os->seur = toka; toka = toka->seur; }
11     os = os->seur;
12   }
13   os->seur = eka ? eka : toka;
14   return tulos;
15 }
```

Kuva 44: Kahden järjestetyn linkitetyn listan lomittaminen yhdeksi

olevaksi linkitetyksi listaksi. Yhtäsuurten alkioiden tapauksessa se suosii listassa `eka` olevaa alkiota. Aliohjelma luottaa siihen, että sille annettavat listat ovat kasvavassa järjestyksessä.

Jos jompikumpi (tai molemmat) syötelistoista on tyhjä, niin `lomita` palauttaa vastakkaisen syötelistan rivillä 2 tai 3. Muussa tapauksessa se valitsee syötelistojen ensimmäisistä alkioista pienemmän ja siirtää sen tuloslistaan riveillä 4, . . . , 6. Riveillä 7, . . . , 12 se uudelleen ja uudelleen valitsee syötelistojen jäljellä olevista alkioista pienemmän ja siirtää sen tuloslistaan, kunnes jompikumpi syötelista loppuu. Sen syötelistan, joka ei vielä loppunut, loppuosa linkitetään tuloslistan jatkoksi rivillä 13.

Kuvan 44 aliohjelma käsittelee erikoistapauksina sekä tyhjät syötelistat että tuloslistan ensimmäisen alkion valitsemisen. Linkitettyjä rakenteita käsittelevissä algoritmeissa on melko tavallista, että listojen päitä ja tyhjiä rakenteita joudutaan käsittelemään erikoistapauksina.

Erikoistapausten määrää voidaan toisinaan vähentää ***päätemerkeillä*** (***sentinel***). Päätemerkki on tietue, joka on muuten samanlainen kuin rakenteen alkio, mutta siinä ei ole hyötykuormaa. Se voi olla varattu `new`:lla kuten rakenteen alkiot, tai, jos ohjelmointikieli sallii, se voi olla kiinteä kuten tavalliset muuttujat. Hyötykuormalle varattu muisti joko jätetään käyttämättä tai ohjelmointikielen sallimin keinoin kikkaillaan niin, että sitä ei koskaan varatakaan.

Päätemerkkejä käyttämällä järjestettyjen linkitettyjen listojen lomittaminen saadaan tällaiseksi:

```
1  void pm_lomita( alkio *eka, alkio *toka ){
2    alkio *os = toka; toka = toka->seur; os->seur = 0;
3    os = eka; eka = eka->seur;
     kuvan 44 rivit 8, . . . , 13
10 }
```

Päätemerkit voidaan nyt luoda komennolla `alkio Eka('\0'), Toka('\0');`. Koska niiden nimien edestä puuttuu `*`, tulee niistä tavallisia `alkio`-tyyppisiä muuttujia, eikä sellaisia, joille varataan muistia `new`:lla. Tässä tapauksessa päätemerkeissä on muistia myös hyötykuormalle, mutta se jätetään käyttämättä. Kuitenkin johtuen siitä miten `alkio`:n rakentaja määriteltiin edellä, on luonnin yhteydessä annettava jokin merkki, joka menee hyötykuormalle varattuun kohtaan muistia. Ei ole väliä, mikä merkki siihen laitetaan. Tässä käytetty `'\0'` tarkoittaa tyhjää merkkiä.

Komento `pm_lomita( &Eka, &Toka );` lomittaa päätemerkeistä `Eka` ja `Toka` alkavat listat siten, että `Eka` muuttuu osoittamaan lopputulosta ja `Toka` muuttuu tyhjäksi listaksi. Tässä `&` muodostaa osoittimen perässään olevaan alkioon. Kuvaan 44 verrattuna säästettiin viisi riviä koodia. Lisäksi ohjelma saatiin sikäli vikasietoisemmaksi, että `Toka`:n kautta ei voi jälkikäteen päästä sotkemaan lomituksen lopputulosta.

7 Lisää järjestämisalgoritmeista

7.1 Lomitusjärjestäminen

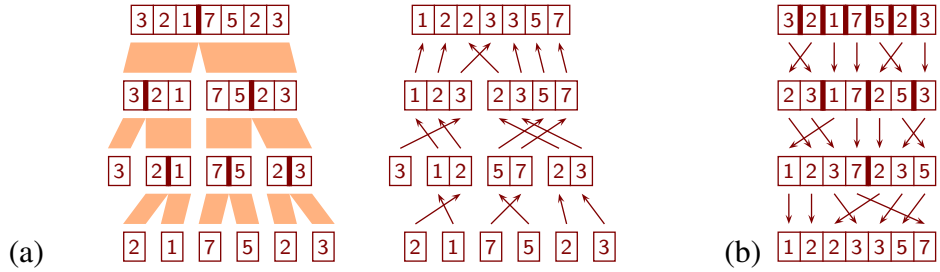
Linkitetyistä listoista voidaan hakea alkioita vain osoittimen tai vastaavan avulla, sekä siirtymällä alkiosta seuraavaan tai edelliseen. Kekojärjestäminen (heapsort) edellyttää kykyä hakea alkioita niiden indeksien perusteella sekalaisessa järjestyksessä, joten se ei sovellu linkitettyjen listojen järjestämiseen. Pikajärjestäminen (quicksort) saadaan toimimaan linkitetyille listoille. On kuitenkin olemassa toinen, linkitetyille listoille erityisen hyvä järjestämisalgoritmi: ***lomitusjärjestäminen*** (***merge sort***).

Lomitusjärjestäminen toimii hyvin myös taulukoille. Siinä käytössä sen ainoa selkeästi huono ominaisuus on, että se käyttää $\Theta(n)$ lisämuistia. Toisin kuin keko- ja pikajärjestäminen, se on vakaa. Kaikki laajalti tunnetut vakaat $O(n \log n)$ järjestämisalgoritmit perustuvat lomitusjärjestämiseen tai alkuperäisen järjestyksen vertaamiseen silloin, kun avaimet ovat yhtäsuuret.

Pikajärjestäminen menettää nykyaikaisissa oliokielissä paikallisuuteen (sivu 88) perustuvan osan nopeusedustaan, koska niissä alkion sisältö on osoittimen tai viitteen takana, joten vierekkäisten alkioiden avaimet eivät välttämättä ole muistissa lähellä toisiaan. Siksi ja vakauden vuoksi esimerkiksi Javan ja Pythonin `sort` ovat tyypillisesti lomitusjärjestämisen versioita (katso Wikipedia ”Timsort” ja etsi muualta ”powersort”). Tätä ei kuitenkaan voi luvata varmana totuutena, koska standardit sallivat niiden perustuvan muihinkin vakaisiin nopeisiin algoritmeihin.

C++-standardi lupaa vakaan järjestämisalgoritmin nimellä `stable_sort`. Sen suoritusajaksi luvataan $O(n \log n)$ jos riittävästi lisämuistia on käytettävissä, ja $O(n(\log n)^2)$ muutoin. Komennolla `g++ -E` esiin tuli ohjelma, joka aloittaa yrittämällä varata tarpeeksi ison työmuistialueen. Jos se onnistuu, niin ohjelma jatkaa yhdellä monimutkaisella lomitusjärjestämisen muunnelmalla, ja muussa tapauksessa toisella monimutkaisella lomitusjärjestämisen muunnelmalla.

Lomitusjärjestämisen toimintaperiaatteen ymmärtämiseksi tarkastelemme ensin helppotajuista toteutusta, jonka toimintaa on havainnollistettu kuvassa 45 (a). Jos järjestettävässä taulukon tai linkitetyn listan osassa on enintään yksi alkio, niin sille ei tehdä mitään. Muussa tapauksessa osa jaetaan keskeltä alkuosaan ja loppuosaan. Jos jako ei mene tasan, niin ei ole väliä kumpaan osaan keskimmäinen



Kuva 45: (a) rekursiivinen ja (b) iteratiivinen lomitusjärjestäminen

alkio otetaan mukaan. Kumpikin osa järjestetään lomitusjärjestämisellä. Lopuksi osat yhdistetään ***lomittamalla*** (***merge***) ne. Kuten kohta nähdään, lomittamisessa on paljon yksityiskohtia, mutta muuten algoritmi on yksinkertainen.

Selkeyden vuoksi kuvassa näytetään kulloinkin taulukosta vain se osa, jota käsitellään sillä hetkellä. Esimerkiksi siinä vaiheessa kun alkiot 7 ja 5 ovat juuri vaihtaneet paikkoja keskenään, on taulukon koko sisältö 1 2 3 5 7 2 3.

Lomittaminen vertaa osataulukoiden ensimmäisiä alkioita keskenään ja kopioi niistä pienemmän tulostaulukon loppuun. Jotta algoritmi olisi vakaa, niin kun osataulukoiden ensimmäiset alkiot ovat yhtäsuuret, se valitsee ensimmäisen osataulukon ensimmäisen alkion. Sitten se tekee saman jäljellä olevista alkioista ensimmäisille ja niin edelleen. Kun jompikumpi osataulukko loppuu, se kopioi toisen osataulukon jäljellä olevan sisällön tulostaulukon jatkeeksi.

Tämä algoritmi on toteutettu kuvassa 46. (Tämän luvun toteutuksia ei ole suo-

```
1  void mergerec( taulukko & A, taulukko & B, unsigned aa, unsigned ll ){
2    if( ll - aa <= 1 ){ return; }
3    unsigned vv = (aa + ll) / 2;
4    mergerec( A, B, aa, vv ); mergerec( A, B, vv, ll );
5    unsigned ii = aa, jj = aa, kk = vv;
6    while( jj < vv && kk < ll ){
7      if( A[ jj ].x <= A[ kk ].x ){ B[ ii ] = A[ jj ]; ++ii; ++jj; }
8      else                        { B[ ii ] = A[ kk ]; ++ii; ++kk; }
9    }
10   while( jj < vv ){ B[ ii ] = A[ jj ]; ++ii; ++jj; }
11   while( kk < ll ){ B[ ii ] = A[ kk ]; ++ii; ++kk; }
12   for( unsigned ii = aa; ii < ll; ++ii ){ A[ii] = B[ii]; }
13 }

14 void mergerec( taulukko & A ){
15   taulukko B( A.size() ); mergerec( A, B, 0, A.size() );
16 }
```

Kuva 46: Rekursiivinen lomitusjärjestäminen taulukoille

	0...4 999		10 000...19 999		10^6...1 000 019	
muistia toistuvasti varaava	0,8 s	14 s	11 s	3 min 45 s	2,0 s	52 s
kuvan 46 `mergerec`	0,6 s	9,2 s	8,6 s	2 min 26 s	1,7 s	32 s
kuvan 48 `mergesort`	0,5 s	7,2 s	7,7 s	2 min 18 s	1,5 s	28 s
`mergesort-8`	0,5 s	7,4 s	7,5 s	2 min 14 s	1,5 s	26 s
`mergesort-16`	0,5 s	7,9 s	7,1 s	2 min 19 s	1,4 s	26 s
`mergesort-32`	0,5 s	9,1 s	6,7 s	2 min 30 s	1,4 s	27 s
`mergesort-64`	0,5 s	12 s	6,8 s	3 min 00 s	1,4 s	31 s
listojen	1,0 s	3,2 s	19 s	1 min 17 s	36 s	1 min 2 s

Kuva 47: Lomitusjärjestämisen suoritusaikaesimerkkejä

jattu Blochin ylivuotoa vastaan, joten ne eivät sovellu valtavan suurille taulukoille.) Enintään yhden alkion kokoiset taulukot käsitellään rivillä 2. Taulukko halkaistaan ja osat käsitellään rekursiivisesti riveillä 3 ja 4. Rivillä 6 ensimmäisen osataulukon jäljellä oleva osa alkaa kohdasta `jj` ja päättyy juuri ennen kohtaa `vv`. Toisen osataulukon jäljellä oleva osa alkaa kohdasta `kk` ja päättyy juuri ennen kohtaa `ll`. Riveillä 6, ..., 9 kopioidaan alkioita tulostaulukkoon kunnes jompikumpi osataulukko loppuu. Riveillä 10 ja 11 kopioidaan loput alkiot siitä osataulukosta, joka ei vielä loppunut.

Taulukoiden tapauksessa tulosta ei ole helppo kirjoittaa samaan taulukkomuuttujaan kuin missä lomitettavat osataulukot sijaitsevat. Siksi tyypillisesti tulos kirjoitetaan toiseen taulukkomuuttujaan. Siitä aiheutuu $\Theta(n)$ lisämuistin tarve. (Wikipedian sivu "Merge sort" mainitsee keinoja, joilla tätä voi lievittää.) Kuvan 46 rivillä 12 tulos kopioidaan alkuperäiseen taulukkoon.

Kuvassa 46 käytetään samaa aputaulukkoa koko suorituksen ajan. Sille varataan muistia suorituksen alussa rivillä 15. Kuvan 47 ensimmäinen ja toinen rivi vaakaviivan jälkeen kertovat, miksi kannattaa tehdä niin. Ensimmäisellä rivillä käytettiin rekursiivista versiota, joka varaa ja vapauttaa tarvitsemansa kokoisen aputaulukon erikseen jokaista lomittamista varten. Toisella rivillä käytettiin kuvan 46 aliohjelmia. Ensimmäisen rivin vertaaminen toiseen kertoo, että on hyödyllistä välttää toistuvaa muistin varaamista ja vapauttamista.

Linkitettyjen listojen tapauksessa ei tarvita aputaulukkoa, joten niillä lomittamisen tarvitseman lisämuistin määrä on $\Theta(1)$ ja muutenkin pieni. Rekursiivinen algoritmi käyttää niille kaiken kaikkiaan $\Theta(\log n)$ lisämuistia, koska ohjelmointikielen suoritusympäristö tarvitsee sen verran rekursion toteuttamiseen.

Koko algoritmin lisämuistin tarve linkitetyillä listoilla saadaan pienennettyä $\Theta(1)$:ksi aloittamalla kokoa 1 olevilla osalistoilla, lomittamalla aina kaksi peräkkäistä kokoa 1 olevaa osalistaa kokoa 2 olevaksi osalistaksi, sitten lomittamalla aina kaksi peräkkäistä kokoa 2 olevaa osalistaa kokoa 4 olevaksi osalistaksi ja niin edelleen. Koska yhden aliohjelmakutsun toteuttamiseen ei tarvita paljoa muistia ja

```
1  inline void merge( const taulukko & A, taulukko & B, unsigned dd ){
2    unsigned ii = 0;
3    while( ii < A.size() ){
4      unsigned jj = ii, kk = ii + dd, vv = kk, ll = kk + dd;
5      if( A.size() < vv ){ vv = A.size(); }
6      if( A.size() < ll ){ ll = A.size(); }
       ... kuvan 46 rivit 6, ..., 11
13   }
14 }

15 void mergesort( taulukko & A ){
16   taulukko B( A.size() );
17   for( unsigned dd = 1; dd < A.size(); dd *= 2 ){
18     merge( A, B, dd ); A.swap( B );
19   }
20 }
```

Kuva 48: Lomitusjärjestäminen (merge sort)

koska $\log n$ kasvaa hitaasti, ei rekursion poistamisella saavutettava muistin säästö ole käytännössä kovin merkittävä.

Saman voi toteuttaa myös taulukoiden tapauksessa, ja siellä siitä on hyötyä. Sen toimintaa on havainnollistettu kuvassa 45 (b), ja se on toteutettu kuvassa 48. Viimeinen osataulukko voi olla (ja usein onkin) muita pienempi. Kuten kuvan 47 toinen ja kolmas vaakaviivan alapuolinen rivi näyttävät, sen ansiosta voi säästää aikaa. Hyöty ei tule rekursion poistosta sellaisenaan, vaan välillisesti: kuten kohta näemme, rekursion poisto tekee helpoksi poistaa myös tuloksen kopioimisen aputaulukosta alkuperäiseen taulukkoon.

Kuvassa 48 muuttujassa `dd` on vuorossa oleva kahden potenssi. Rivin 18 lopussa vaihdetaan `A`:n ja `B`:n sisällöt keskenään. Jos taulukko on C++:n `vector`, niin vaihto on nopea vakioaikainen toimenpide. Vakioaikaisuuden mahdollistaa se, että kuten luvussa 3.3 kerrottiin, `vector`:in alkiot ovat erillisessä muistilohkossa, joka löytyy osoittimen avulla. Niiden sisällöille ei vaihdossa tarvitse tehdä mitään, vaan riittää vaihtaa niihin osoittavat osoittimet keskenään. Jos ohjelmointikieli ei tarjoa vastinetta C++:n `swap`:lle, niin tarvitaan muu ratkaisu. Eräs sellainen on muuttaa rivi 18 muotoon `merge( A, B, dd ); dd *= 2; merge( B, A, dd );`.

Kuten pikajärjestämisen tapauksessa, algoritmia voi nopeuttaa järjestämällä jotakin rajaa pienemmät osataulukot lisäysjärjestämisellä. Toisin kuin pikajärjestämisessä, nyt se tehdään heti aluksi. Lomittaminen aloitetaan tilanteessa, jossa taulukko koostuu esimerkiksi 8 alkion kokoisista osataulukoista, joista jokainen on sisäisesti järjestyksessä (ja viimeinen voi olla muita pienempi). Kuvassa 47 tästä saatiin hyötyä kun alkiossa on vain avain, mutta enimmäkseen haittaa kun alkiossa on lisäksi 400 tavua muuta dataa.

7.2 Satunnaisen järjestyksen tuottaminen

Taulukon alkioiden saattamista satunnaiseen järjestykseen tarvitaan esimerkiksi kun halutaan arpoa lukuja siten, että mikään luku ei toistu. Se onnistuu arpomalla ensin alkio koko taulukosta ja vaihtamalla se ensimmäiseksi. Sitten arvotaan alkio muista kuin ensimmäinen ja vaihdetaan se toiseksi, ja niin edelleen. Alla $\textsc{Random}(i, n-1)$ tuottaa satunnaisen kokonaisluvun väliltä $i, \ldots, n-1$ tasajakaumalla.

```
1  for i := 0 to n − 1 do A[i] := i
2  for i := 0 to n − 2 do
3      Swap(A, i, Random(i, n − 1))
```

On tärkeää tiedostaa, että tasajakaumaa ei saavuteta esimerkiksi siten, että arvotaan paikka koko taulukosta ja vaihdetaan sen sisältö ensimmäisen paikan kanssa, sitten arvotaan paikka koko taulukosta ja vaihdetaan sen sisältö toisen paikan kanssa ja niin edelleen. Esimerkiksi kolmen alkion taulukossa tällainen arvonta voi tuottaa kaikkiaan $3^3 = 27$ erilaista arvontatulosten yhdistelmää. Mutta kolmella erisuurella alkiolla on kuusi eri järjestystä, ja 27 ei ole tasan jaollinen kuudella. Siksi tällainen arvonta tuottaa jotkin järjestykset useammin kuin toiset. Tarkempi analyysi osoittaa, että osa järjestyksistä tulee todennäköisyydellä $\frac{4}{27}$ ja loput todennäköisyydellä $\frac{5}{27}$. Pienellä ohjelmalla tehdyn kokeen mukaan neljällä erisuurella alkiolla tulee niinkin suuria eroja, että järjestys 1 0 3 2 tulee todennäköisyydellä $\frac{15}{256}$ mutta 3 0 1 2 todennäköisyydellä $\frac{8}{256}$.

Jos satunnaisluvun joutuu itse rajaamaan halutulle välille, niin on hyvä tiedostaa myös, että modulo ei tuota tasajakaumaa paitsi siinä tapauksessa, että lähtökohtana oleva erisuuruisten arvojen määrä on jaollinen tavoitteena olevalla erisuuruisten arvojen määrällä. Usein lähtökohtana on 2^{31} tai 2^{32} erisuuruista arvoa. Yllä olevassa algoritmissa tavoitteena on vuorotellen kaikki määrät n:stä kahteen, joten jos $n \geq 5$, niin jaollisuus jää monta kertaa toteutumatta (ja yhden kerran toteutumatta, jos n on 3 tai 4). Toisaalta jos tavoitteena oleva määrä on paljon pienempi kuin lähtökohtana oleva määrä, niin modulon tuottama virhe on niin pieni, että siitä ei ehkä tarvitse välittää.

7.3 Vertailuun perustuvan järjestämisen ajan alaraja

Taulukon järjestämiseksi tarvittavat alkioiden siirrot riippuvat siitä, missä järjestyksessä taulukko on alun perin. Jos kaikki taulukon alkiot ovat keskenään erisuuret, niin yksi nimenomainen alkio kuuluu ensimmäiseksi (nimittäin pienin). Se voi alussa olla missä tahansa kohdassa taulukkoa. Niinpä jos alkioita on n kappaletta, niin pienimmän siirtäminen ensimmäiseksi edellyttää sen löytämistä n mahdollisesta paikasta. Se tarkoittaa yhden vaihtoehdon valitsemista n:stä. Toiseksi pienin

voi olla missä tahansa $n-1$ paikasta, koska se ei voi olla siellä missä pienin on, mutta voi olla missä tahansa muualla. Sen löytäminen edellyttää yhden vaihtoehdon valitsemista $(n-1)$:stä. Sitä seuraava alkio on löydettävä $n-2$ vaihtoehdosta ja niin edelleen, kunnes viimeinen alkio on ainoa joka on jäljellä.

Siksi jos kaikki taulukon alkiot ovat keskenään erisuuret, niin taulukon järjestäminen edellyttää yhden vaihtoehdon valitsemista $n \cdot (n-1) \cdot \ldots \cdot 1$ vaihtoehdosta. Lukua $n \cdot (n-1) \cdot \ldots \cdot 1$ eli $1 \cdot 2 \cdot \ldots \cdot n$ kutsutaan nimellä n ***kertoma*** (***factorial***) ja merkitään $n!$.

Jokainen kahden alkion vertailu jakaa järjestykset kahtia: niihin, joissa tulos on true ja niihin, joissa se on false. Kasat eivät välttämättä ole edes likimain yhtäsuuret. Esimerkiksi lisäysjärjestämisen ulomman silmukan viimeisen kierroksen alussa taulukko on muuten järjestyksessä, mutta viimeinen alkio on vielä siirtämättä oikealle paikalleen suhteessa muihin. Silloin on jäljellä n vaihtoehtoa: viimeinen alkio kuuluu ensimmäiseksi, toiseksi, ..., toiseksi viimeiseksi tai viimeiseksi. Jos ulomman silmukan viimeisen kierroksen ensimmäinen vertailu tuottaa false, niin viimeinen alkio kuuluu viimeiseksi, ja muussa tapauksessa se kuuluu johonkin paikoista $1, \ldots, n-1$. Siis tässä esimerkissä vertailu jakaa vaihtoehdot siten, että jäljelle jää joko yksi tai $n-1$ vaihtoehtoa.

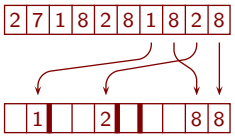
Huonoimmassa tapauksessa aina kun jako ei mene tasan, päätyy isompi kasa jatkoon. Jotta siinä tapauksessa saataisiin valittua yksi vaihtoehto $n!$ vaihtoehdosta, tarvitaan ainakin $\log_2(n!)$ eli $\log_2 1 + \log_2 2 + \ldots + \log_2 n$ vertailua. Laskimme sivulla 44, että $\log_2 1 + \log_2 2 + \ldots + \log_2 n$ on $\Theta(n \log n)$. Tästä seuraa, että jos kaikki alkiot ovat keskenään erisuuret, niin jokainen järjestämisalgoritmi, joka hankkii tiedon oikeasta järjestyksestä vertaamalla alkioita (eikä muilla keinoin), tekee hitaimmillaan $\Omega(n \log n)$ vertailua ja siksi käyttää hitaimmillaan $\Omega(n \log n)$ aikaa.

Keskimääräisessä tapauksessa, jos jako ei mene tasan, niin jatkoon päätyy isompi tai pienempi kasa niiden kokoihin verrannollisilla todennäköisyyksillä. Isomman kasan jatkokäsittely vaatii keskimäärin enemmän vertailuja kuin pienemmän kasan. Tarkka lasku on liian monimutkainen tässä läpi käytäväksi, mutta lopputuloksen voi kertoa: keskimäärin tarvittavien vertailujen määrä on sitä suurempi mitä epätasaisempia jaot ovat. Siksi jos kaikki alkiot ovat keskenään erisuuret, niin jokainen järjestämisalgoritmi, joka hankkii tiedon oikeasta järjestyksestä vertaamalla alkioita (eikä muilla keinoin), käyttää keskimäärin $\Omega(n \log n)$ aikaa.

Tämä laskelma perustuu oletukseen, että kaikki alkiot ovat keskenään erisuuret. Jos alkioissa voi olla yhtäsuuria, niin pienempi työmäärä saattaa riittää. Esimerkiksi vastauksessa 91 esitetty pikajärjestämisen muunnelma ei lähetä jakoalkion suuruisia alkioita alemmille rekursiotasoille. Siksi jos avaimissa esiintyy vain kahta eri arvoa, niin rekursiotasoja on enintään kaksi, joten työmäärä on $O(n)$.

Laskelma olettaa myös, että järjestämisalgoritmi hankkii tietoa oikeasta järjestyksestä vain vertaamalla alkioita toisiinsa. Seuraavassa kahdessa alaluvussa

```
1  CountingSort(&A, k)
2  luo taulukko B(A.koko); luo taulukko L(k)
3  for j := 0 to k − 1 do L[j] := 0
4  for i := 0 to A.koko − 1 do L[A[i].x] := L[A[i].x] + 1
5  for j := 1 to k − 1 do L[j] := L[j] + L[j − 1]
6  for i := A.koko − 1 downto 0 do
7     L[A[i].x] := L[A[i].x] − 1; B[L[A[i].x]] := A[i]
8  A.vaihda(B)
```



Kuva 49: Laskentajärjestäminen

käsitellään järjestämisalgoritmeja, joille tämä oletus ei päde.

7.4 Laskentajärjestäminen

Laskentajärjestäminen (***counting sort***) on erinomainen silloin, kun avaimet ovat peräisin pienehköstä peräkkäisten kokonaislukujen joukosta, ja muistia ei tarvitse säästää. Siinä järjestettävä taulukko A selataan läpi ja lasketaan, kuinka monta kertaa kukin avain esiintyy. Tulosten perusteella kukin alkio kopioidaan tulostaulukkoon suoraan oikealle paikalleen.

Laskentajärjestäminen on esitetty pseudokoodina kuvassa 49 vasemmalla ja sen toimintaa on havainnollistettu kuvassa oikealla. Siinä A on järjestettävä taulukko, tulos muodostetaan taulukkoon B, ja L sisältää laskurit, joilla lasketaan kunkin avaimen esiintymien määrä. Kuvassa oletetaan, että avainten arvot ovat väliltä $0, \ldots, k-1$. L:n alkiot ovat ei-negatiivisia kokonaislukuja.

Rivillä 2 B luodaan samankokoiseksi kuin A, eikä sitä tarvitse alustaa. L luodaan kokoon k ja alustetaan täyteen nollia. Vaikka alustaminen voidaan monissa ohjelmointikielissä määrätä luontikomennossa, on se tässä näytetty erikseen rivillä 3, koska se muodostaa merkittävän osan kokonaistyömäärästä.

Rivillä 4 lasketaan avaimille mahdollisten arvojen esiintymien määrät. Rivillä 5 L muutetaan sellaiseksi, että $L[i]$ kertoo niiden alkioiden määrän, joiden avain on enintään i.

Jos avain i esiintyy A:ssa, niin tässä vaiheessa $L[i]$ on yhtä suurempi kuin sen paikan numero, johon kuuluu viimeinen niistä alkioista, joiden avain on i. Alkiot kopioidaan tämän tiedon perusteella suoraan oikeille paikoilleen riveillä 6 ja 7. Aina kun alkio kopioidaan, vastaavaa laskuria pienennetään, jotta seuraavaksi kopioitava samansuuruinen alkio tulee kopioitua viereiseen paikkaan.

Lopuksi tulos siirretään A:han hyödyntämällä ainakin C++:ssa tarjolla olevaa toimintoa, joka vaihtaa kahden taulukon sisällöt keskenään ajassa $\Theta(1)$.

Jos merkitään k:lla sen joukon kokoa, josta avaimet poimitaan, niin laskentajärjestämisen sekä ajan että muistin kulutus on $\Theta(n+k)$. Koska laskentajärjestäminen siirtää kunkin alkion vain kerran, katsoo sen avainta vain muutaman kerran

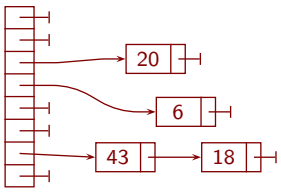
ja tekee vain vähän työtä kutakin laskuritaulukon alkiota kohti, on se myös käytännössä hyvin nopea. Kuvassa 41 se oli pienillä alkioilla ylivoimainen. Suurilla alkioilla se hävisi vain aputaulukkoa käyttäville algoritmeille, ja kaikkein isoimmilla taulukoilla se voitti nekin. Ei ole tiedossa, miksi se hävisi aputaulukkoa käyttäville algoritmeille suurilla alkioilla pienillä ja keskisuurilla taulukoilla.

109. Kirjoita laskentajärjestäminen C++:lla olettaen, että avaimet ovat väliltä $0, \ldots, 255$!

110. Etukäteen on luvattu, että avaimet ovat väliltä $-35\,000, \ldots, -30\,000$. Taulukot indeksoidaan nollasta alkaen. Miten laskentajärjestäminen saadaan toimimaan ilman suurta turhaa muistin kulutusta?

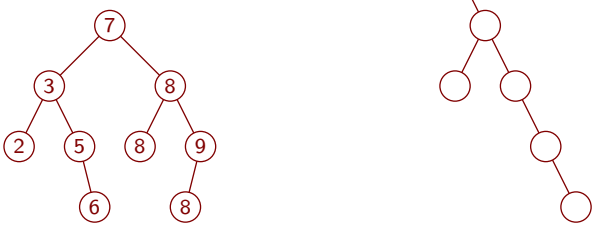
7.5 Kantalukujärjestäminen

8 Hajautustaulut

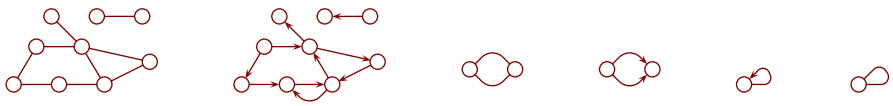


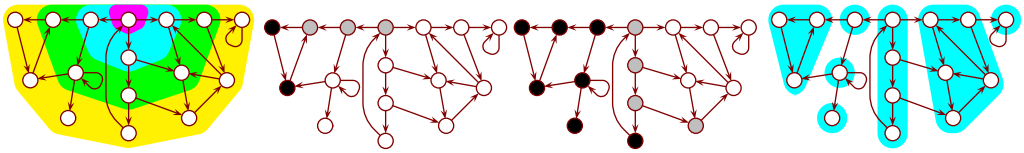
8.1 Sovellusesimerkki: reitin etsinnän syötteen uudistaminen

9 Binääripuut



10 Graafialgoritmeja





11 Välitulosten muistaminen

Kysymysten vastauksia

1. Löysit kysymyksen 1 mallivastauksen! Jos tulit tänne klikkaamalla, niin nyt klikkaa tuosta ↰, niin pääset takaisin kysymyksen 1 kohdalle.

2. `pisteraja`$[2] \leq$ `pisteet` $<$ `pisteraja`$[3]$

3. Pisterajojen täytyy olla kasvavassa järjestyksessä.

4. Jos ohjelmanpätkä antaa arvosanaksi 5, niin suurempia arvosanoja ei ole olemassa. Muussa tapauksessa silmukan lopetusehdon vuoksi rivin 4 jälkeen `pisteet` $<$ `pisteraja`$[$`arvosana`$]$, joten pisteet eivät riitä lähinnä suurempaan arvosanaan. Ne eivät riitä suurempiinkaan arvosanoihin, koska pisterajat ovat kasvavassa järjestyksessä.

5. Jos arvosanaksi tuli 0, niin pisteet riittävät siihen, sillä mikä tahansa pistemäärä riittää nollaan. Muussa tapauksessa `while`-silmukan runko on suoritettu ainakin kerran. Sen viimeisen kierroksen vaikutuksesta `pisteet` $\geq$ `pisteraja`$[$`arvosana` $-1]$, joten pisteet riittävät annettuun arvosanaan.

6. Otetaan käyttöön lokero `pisteraja`$[5]$ ja sijoitetaan sinne luku, joka on suurempi kuin tentin maksimipistemäärä.

7. RANDOM$(y-a+1)+a$

8. ```
   26 0
   27 0
   28 0
   29 0
   30 0
   ```

9. Kun arvosanan tulisi olla 5, se palauttaa 0.

10. Se indeksoi taulukkoa laittomasti kohdassa `rajat[i+1]`, kun pisteitä on ainakin 26.

11. Lisätään rivien 14 ja 15 väliin `if( pist >= rajat[4] ) return 5;`.

12. $[14, 17, 21, 23, 26]$

    Lisätietoa. Niitä on melko monta muutakin kuin mallivastaus.

13. $[14, 17, 20, 23, 25]$

    Lisätietoa. Niitäkin on monta erilaista.
   ```

14. Avain saattaa olla taulukossa useasti, ja testikohde ja luotettava etsintä saattavat löytää eri kohdat.

15. 

16. Rivillä 3 on ainakin kerran $j = 0$ jos ja vain jos jokin A:n alkio on pienempi kuin A:n ensimmäinen alkio. Taulukossa A on tällöin ainakin kaksi alkiota.

17. Kun $n = 1$, ei **for**-silmukka kierrä yhtään kierrosta. Niinpä lopputuloksena on sama taulukko kuin alussa. Se on oikein, koska kun $n = 1$, on A:ssa täsmälleen yksi alkio, joten A on valmiiksi järjestyksessä.

18. $y - a + 1$

19. $A[0 \ldots i-1]$ ja $A[i \ldots n-1]$

20. $0, \ldots, n$. Niillä saadaan kaikki vaihtoehdot alkaen tyhjästä alkuosasta siihen saakka, että kaikki alkiot ovat alkuosassa. Jos $i < 0$, niin $A[0 \ldots i-1]$:n yläraja on liian pieni alarajaan verrattuna. Jos $i > n$, niin $A[i \ldots n-1]$:n yläraja on liian pieni alarajaan verrattuna.

21. Rivin 5 sijoituksen tarkastamiseksi pitää ottaa huomioon myös se mahdollisuus, että sisempi silmukka ohitetaan. Silloin ei voida vedota siihen mikä arvo j:llä on sisemmän silmukan lopettaessa, vaan täytyy vedota j:n arvoon juuri ennen sisempää silmukkaa. Myös sille pätee $0 \leq j \leq i$, koska sille pätee $j = i$ ja $1 \leq i$.

22. Koska osa ei voi olla pienempi kuin tyhjä, täytyy päteä $-1 \leq j-2$. Se pätee rivin 4 alussa, koska rivin 3 ehdon vuoksi $j > 0$. Jotta osa ei ulottuisi liian pitkälle oikealle, täytyy päteä $j - 2 \leq i - 1$. Se pätee, koska $j \leq i$.

23. Ensin aikajärjestykseen vanhin ensin, sitten lähettäjän mukaiseen aakkosjärjestykseen ja lopuksi, jotta otsikko ”Ääkköset eivät toimi” olisi lähellä alkua, aiheen mukaiseen takaperoiseen aakkosjärjestykseen.

24. Rivillä 3 $A[j-1].x > apu.x$ muutetaan muotoon $A[j-1].x \geq apu.x$.

25. Se on $O(n^3)$. Se ei ole hitaimmillaan $\Theta(n^3)$.

    Lisätietoa. INSERTIONSORT:in suoritusaika voi syötteen lajista riippuen olla muotoa n, n^2 tai siltä väliltä. Järjestyksessä oleville taulukoille se on muotoa n ja takaperoisessa järjestyksessä oleville muotoa n^2. Jokainen niistä kasvaa hitaammin kuin n^3, joten jokainen niistä kasvaa enintään yhtä nopeasti kuin n^3. O-merkintä tarkoittaa ”yhtä nopeasti tai hitaammin”. Siksi syötteen lajista riippumatta suoritusaika on $O(n^3)$ (ja $O(n^7)$ ja $O(n^2\sqrt{n})$ ja

...). Θ-merkintä tarkoittaa "yhtä nopeasti". Suoritusaika on hitaimmillaan muotoa n^2. Se kasvaa aidosti hitaammin kuin n^3, joten se ei ole $\Theta(n^3)$.

26. Se on $O(\frac{n^2}{100})$ ja hitaimmillaan $\Theta(\frac{n^2}{100})$.

    Lisätietoa. Jakaminen sadalla on kertomista vakiolla $\frac{1}{100}$. Vakiokertoimia ei oteta huomioon O-, Θ- ja Ω-merkinnöissä. Niinpä $O(\frac{n^2}{100})$ tarkoittaa samaa kuin $O(n^2)$ ja $\Theta(\frac{n^2}{100})$ tarkoittaa samaa kuin $\Theta(n^2)$.

27. INSERTIONSORT:in suoritusaikaa ei voi ilmaista Θ-merkinnällä.

    Lisätietoa. Koska mukana ei ole rajausta tyyliin "hitaimmillaan", "nopeimmillaan", "keskimäärin" tai "tällaisille taulukoille", on ilmauksen oltava pätevä syötteen lajista riippumatta. Koska ainakin yhdellä syötteiden lajilla suoritusaika on $\Theta(n^2)$, ja koska $\Theta(n)$ ei tarkoita samaa kuin $\Theta(n^2)$, ei $\Theta(n)$ ole pätevä syötteen lajista riippumatta. Koska ainakin yhdellä syötteiden lajilla suoritusaika on $\Theta(n)$, ei $\Theta(n^2)$ ole pätevä syötteen lajista riippumatta.

28. $\Theta(n^2)$. Kukin nolla joudutaan siirtämään vasemmalle ykkösten ohi, mistä tulee $\frac{n}{3}$ rivin 4 suoritusta. Koska nollia on $\frac{2n}{3}$ kappaletta, tulee rivin 4 suorituksia yhteensä $\frac{2n}{3} \cdot \frac{n}{3} = \frac{2n^2}{9}$. Tästä tulee suoritusajaksi $\Omega(n^2)$. Koska tekstin mukaan se on aina $O(n^2)$, on se tällaisten taulukoiden tapauksessa $\Theta(n^2)$.

29. $\Theta(n\sqrt{n})$. Kukin nolla joudutaan siirtämään vasemmalle ykkösten ohi, mistä tulee $\sqrt{n}$ rivin 4 suoritusta. Nollia on $n - \sqrt{n}$ kappaletta, joten rivin 4 suorituksia tulee yhteensä $(n - \sqrt{n})\sqrt{n} = n\sqrt{n} - (\sqrt{n})^2 = n\sqrt{n} - n$. Rivi 3 suoritetaan $n-1$ kertaa enemmän eli $n\sqrt{n}-1$ kertaa (paitsi jos $n = 0$). Muut rivit suoritetaan $O(n)$ kertaa. Niinpä $n\sqrt{n}$ hallitsee, joten suoritusaika on $\Theta(n\sqrt{n})$.

30. $\Theta(n)$. Kun kaikki alkiot ovat yhtäsuuret, tuottaa rivin 3 testi $A[j-1].x > apu.x$ aina epätoden, joten riville 4 ei koskaan mennä.

    Lisätietoa. Tämän voi perustella myös toteamalla, että kun kaikki alkiot ovat yhtäsuuret, on taulukko kasvavassa järjestyksessä, ja INSERTIONSORT:in suoritusaika kasvavassa järjestyksessä olevalle taulukolle on $\Theta(n)$.

31. Toiminto `erase` siirtää `A`:n loppuosaa taaksepäin, joten se tekee kaiken kaikkiaan $(n-1)+(n-2)+\ldots+0 = \Theta(n^2)$ siirtoaskelta. Toiminto `insert` tekee ainakin yhtä monta siirtoaskelta mutta voi tehdä enemmänkin, koska lisäys tapahtuu samaan tai enemmän vasemmalla olevaan paikkaan kuin poisto. Enimmillään se tekee $n(n-1)$ siirtoaskelta. Nimittäin takaperin olevalle taulukolle se sijoittaa kunkin alkion vuorollaan ensimmäiseksi, jolloin jokainen muista $n-1$ alkiosta siirtyy yhden alkion verran taaksepäin. Niinpä

kysymyksen ohjelma käyttää sekä hitaimmillaan että nopeimmillaan $\Theta(n^2)$ aikaa.

INSERTIONSORT:in nopeimman tapauksen ajan kulutus $\Theta(n)$ on paljon parempi. Hitaimmassa ja keskimääräisessä tapauksessa molemmat ovat $\Theta(n^2)$, mutta kysymyksen ohjelma on vakiokertoimen verran hitaampi, koska se siirtää `for`-silmukan jokaisella kierroksella `A`:n loppuosan turhaan yhden askeleen verran edestakaisin.

32. Koska `size()` on etumerkitöntä tyyppiä, ei `A.size()-1` tuota tyhjällä taulukolla -1 vaan suurimman luvun, jonka `size()`:n tyyppi pystyy esittämään. Siksi `i < A.size()-1` toteutuisi kunnes `i` on hyvin suuri, joten ohjelma käsittelisi `A`:n olemattomia alkioita kunnes se kaatuu.

33. Rivin 2 alussa pätee $0 \leq i < n-1$. Rivin 4 alussa pätee $0 \leq i \leq p < j < n$. Rivin 6 alussa pätee $0 \leq i \leq p < n$.

34. Taulukossa on alkuperäiset alkiot. Osa $A[0 \ldots i-1]$ on kasvavassa järjestyksessä. Joko $i = 0$ tai jokainen osan $A[i \ldots n-1]$ alkio on vähintään yhtäsuuri kuin $A[i-1]$.

    Lisätietoa. Tämän vaatimuksen voi ilmaista myös muodossa ”jokainen osan $A[i \ldots n-1]$ alkio on vähintään yhtäsuuri kuin jokainen osan $A[0 \ldots i-1]$ alkio.” Jollei sitä vaadita missään muodossa, ei voida osoittaa, että $A[0 \ldots i-1]$ on kasvavassa järjestyksessä sen jälkeenkin kun i:tä on kasvatettu.

35. Jokainen osan $A[i \ldots j-1]$ alkio on vähintään yhtäsuuri kuin $A[p]$.

36. Ei ole. $[2,2,1]$

37. $[1,1]$

38. Sekä hitaimman että nopeimman tapauksen ajan kulutus on $\Theta(n^2)$. Kun $n > 1$, rivi 3 suoritetaan $n + (n-1) + \ldots + 1 = \frac{1}{2}n^2 + \frac{1}{2}n$ kertaa, eikä mitään riviä suoriteta useammin.

39. $tulos \cdot x^{i+1} + a_i x^i + \ldots + a_0 = a_n x^n + \ldots + a_0$

40. Iso-Syöte

41. Täsmälleen yksi lapsi on yhdellä tai ei yhdelläkään solmulla. Jos jollakin solmulla on täsmälleen yksi lapsi, niin solmujen määrä on parillinen.

Lisätietoa. Koska alin rivi täytetään vasemmalta alkaen, on yksilapsisia solmuja enintään yksi, nimittäin se jonka vasen lapsi on alimman rivin viimeinen solmu. Jos yksilapsisia solmuja ei ole lainkaan, on lasten kokonaismäärä parillinen, ja jos täsmälleen yhdellä solmulla on yksi lapsi, on lasten kokonaismäärä pariton. Olkoon n solmujen määrä. Koska ylin solmu ja vain se ei ole minkään solmun lapsi, on lapsia kaikkiaan $n-1$. Niinpä jos jollakin solmulla on täsmälleen yksi lapsi, on $n-1$ pariton, joten n on parillinen.

42. Olkoon n solmujen määrä. Koska jokaisella solmulla on paikka vasemmalle ja oikealle lapselle, on lapsille paikkoja kaikkiaan $2n$. Koska solmuista $n-1$ on lapsia, on käyttämättömiä lasten paikkoja $n+1$. Jos n on parillinen, niin yksi solmu on yksilapsinen. Silloin yksi käyttämätön lapsen paikka on sen, ja muut käyttämättömät lasten paikat ovat lapsettomien solmujen. Niinpä $\frac{n}{2}$ solmua on lapsettomia. Jos n on pariton, niin yksilapsisia solmuja ei ole, joten $\frac{n+1}{2}$ solmua on lapsettomia.

43. Jos $1 \leq i \leq \frac{n}{2}$ niin lokeron i vasen lapsi on lokerossa $2i$, ja jos $1 \leq i \leq \frac{n-1}{2}$ niin oikea lapsi on lokerossa $2i+1$.

44. Jos $1 \leq i \leq n-1$, niin lokeron i vanhempi on lokerossa $\lfloor \frac{i-1}{3} \rfloor$. Jos $0 \leq i \leq \frac{n-2}{3}$ niin lokeron i ensimmäinen lapsi on lokerossa $3i+1$, ja jos $0 \leq i \leq \frac{n-4}{3}$ niin viimeinen lapsi on lokerossa $3i+3$.

45. Jos $1 \leq i \leq n-1$, niin lokeron i vanhempi on lokerossa $\lfloor \frac{i-1}{d} \rfloor$. Jos $0 \leq i \leq \frac{n-2}{d}$ niin lokeron i ensimmäinen lapsi on lokerossa $di+1$, ja jos $0 \leq i \leq \frac{n-d-1}{d}$ niin viimeinen lapsi on lokerossa $di+d$.

46. Suurempi, jotta nostettu lapsi ja toinen lapsi olisivat noston jälkeen oikeassa suuruusjärjestyksessä.

    Lisätietoa. Voi ajatella, että jos lapset ovat keskenään yhtäsuuret, niin kannattaa nostaa oikeanpuolimmainen, sillä siitä ei ole pitempi mutta saattaa olla lyhyempi matka lapsettomaan alkioon. Mutta nykyaikaisissa tietokoneissa tällainen päättely ei välttämättä päde, koska muutoksella saattaa olla vaikutusta myös muun muassa siihen, kuinka hyvin ohjelma kykenee hyödyntämään muistin nopeimpia osia.

47.
```
    POISKEOSTA(&A)
1   h := A.koko − 1; i := 0; j := 1
2   while true do
3     if j + 1 < h && A[j + 1].x ≥ A[j].x then j := j + 1
4     if j ≥ h || A[j].x ≤ A[h].x then break
5     A[i] := A[j]; i := j; j := 2i + 1
6   A[i] := A[h]; A.kooksi(h)
```

48. Ajan kulutus on $\Theta(n)$, jos ja vain jos se on sekä $O(n)$ että $\Omega(n)$. Puuttuu todistus, että se on $\Omega(n)$, eli että aikaa kuluu ainakin lukuun n verrannollisesti. Niin paljon aikaa kuluu jo siihen, että kukin n:stä alkiosta sijoitetaan taulukkoon.

49.
```
1  unsigned tuplaa( unsigned alue, unsigned eksp ){
2    if( eksp >= maks_eksp ){ return ~0u; }
3    unsigned uusi = vapaat[ eksp+1 ];
4    if( uusi != ~0u ){ vapaat[ eksp+1 ] = muisti[ uusi ]; }
5    else if( (1u << (eksp+1)) > kaikki - loppu ){ return ~0u; }
6    else{ uusi = loppu; loppu += (1u << (eksp+1)); }
7    for( unsigned i = 0; i < (1u << eksp); ++i ){
8      muisti[ uusi + i ] = muisti[ alue + i ];
9    }
10   muisti[ alue ] = vapaat[ eksp ]; vapaat[ eksp ] = alue;
11   return uusi;
12 }
```

50. 40.

51. $1, 2, \ldots, n$

52. Jos kaikki alkiot ovat keskenään yhtäsuuret, niin riville 4 ei mennä kertaakaan. Silloin kukin aliohjelman LISÄÄKEKOON kutsu vie aikaa $\Theta(1)$ plus A:n kasvattamiseen kuluva aika, ja kaikkiaan aikaa kuluu $\Theta(n)$.

53. Ei kuulu. Se on lokerossa $\frac{n-1}{2}$ eli $\lfloor \frac{n}{2} \rfloor$, ja k aloittaa sitä edeltävästä lokerosta. Paras aloituspaikka k:lle on viimeisen lapsellisen alkion paikka eli viimeisen alkion vanhemman paikka. Se on $\lfloor \frac{(n-1)-1}{2} \rfloor$ eli $\lfloor \frac{n}{2} \rfloor - 1$.

54. `for( unsigned k = n/2; k--; )`

55. Jos alkion prioriteetti huonontuu alkion ollessa keossa, niin alkio tulee keosta ulos ensin aikaisemmalla, liian korkealla prioriteetillaan. Siksi se voi tulla käsitellyksi liian aikaisin, eli ennen jotain toista alkiota, jonka prioriteetti on korkeampi kuin alkion voimassa oleva mutta matalampi kuin alkion aikaisempi prioriteetti.

56. Jos alkion prioriteetti ensin huonontuu ja sitten parantuu ennalleen alkion ollessa keossa, niin alkio saattaa tulla käsitellyksi sekä silloin kun alkuperäinen että silloin kun kolmas kappale tulevat keosta.

57. Jokin suurin ensin. Sen näkee vertailuista riveillä 8, 12 ja 14, joissa valitaan pienempi siirrettäväksi alaspäin tai suurempi siirrettäväksi ylöspäin.

58. Ei. Siirto rikkoisi koodin, koska vanhan ja uuden paikan välissä on suoritettu $S[k] := 0$.

59. A0 B2 C4 E8 C4 E7 E8 D5 E7 E8 E7 E8 F10 E8 F9 F10 F9 F10

60. Korvataan ensimmäinen kutsu koodilla, joka asettaa keon samaan tilaan kuin ensimmäinen kutsu, eli alustaa sen sisältämään vain lähdön etäisyydellä 0: `keko.resize(1); keko[0].minne = lahto; keko[0].pituus = 0.;`

61. Koska kärki on keossa minimietäisyytensä kanssa, sille on suoritettu rivi 59 tai 72 minimietäisyydellä. Rivin 59 tai 70 vuoksi myös `solmut[` *kärki* `].etaisyys` oli silloin minimietäisyys. Invariantin 2 vuoksi se on yhä minimietäisyys.

62. Jos minimipituisella reitillä olisi kaksi kärkeä, niin niistä jälkimmäinen rikkoisi ensimmäisen osalta ehtoa, että kärjen jälkeen reitillä minkään solmun `etaisyys` ei ole minimietäisyys.

63. Tehtävän 59 kartassa minimipituisen reitin A C D E solmun E `etaisyys` oli minimietäisyys silloin, kun D:n `etaisyys` muuttui minimietäisyydeksi. Siksi E säilyi reitin kärkenä eikä D:stä tullut reitin kärkeä.

64. Ei voi. Invariantti 3 ja sen perustelu pätevät ei pelkästään yhdelle, vaan jokaiselle solmuun vievälle minimipituiselle reitille.

65. Jos lähtö ja maali ovat sama solmu, niin riviä 64 ei suoriteta kertaakaan.

66. Taulukko `solmut` käyttää $\Theta(n)$ ja `kaaret` käyttää $\Theta(m)$ muistia. Keossa voi olla alkioita kerrallaan enintään kaarten määrän verran plus yksi, joten keko tarvitsee $O(m)$ muistia. Yhteensä niistä tulee $\Theta(n+m)$.

67. Jos kaaria ei ole lainkaan, niin ”enintään kaarten määrän verran” on nolla, mutta keossa on sitä enemmän alkioita kun `lahto` on (yksinään) siellä.

68. Kaikki kaaret alkavat lähtösolmusta ja ne tulevat syötteessä aidosti lyhenevässä järjestyksessä. Silloin kukin kaari vuorollaan tuottaa solmun ja etäisyyden parin, joka kiipeää keon ylimmäksi. Aikaa kuluu lukuun $\log_2 1 + \log_2 2 + \ldots + \log_2 n$ verrannollisesti. Sivulla 44 nähtiin, että se on $\Theta(n \log n)$.

69. Keko pitää tyhjentää (ellei käytetä vastausta 60) ja etäisyydet asettaa ääarettömäksi. Muuttujat `lahto` ja `maali` pitää asettaa uuden tehtävän mukaisiksi. Kaaria ja `kaarten_loppu`:ja ei pidä muuttaa, koska kartta säilyy entisenä. Muuttujille `reitti` ei tarvitse tehdä mitään, mutta ne saa alustaa jos haluaa.

70. Otetaan käyttöön taulukko, jonka perään lisätään solmun numero aina kun solmu käsitellään rivillä 59 tai 70. Alustuksessa alustetaan vain ne solmut, joiden numero on tässä taulukossa.

    Lisätietoa. Tällainen taulukko voi kasvaa kaarten määrän plus yksi kokoiseksi. Muistin tarve voidaan vähentää verrannolliseksi solmujen määrään lisäämällä solmu taulukkoon vain jos se ei ole siellä jo. Se onnistuu tehokkaasti testaamalla rivillä 70 ennen lisäystä, onko solmun `etaisyys` ääretön.

71. Ei ole. Jos `maali` ei esiinny minkään kaaren kärkipäänä eikä ole `lahto`, niin siihen ei ole reittiä. Siksi se ei voi olla `etsi_reitti`:n `solmu1` eikä `solmu2`, ja pääohjelma lopettaa rivillä 88 ennen kuin se yrittää tulostaa sen etäisyyttä ja reittiä siihen.

    Lisätietoa. On helpompaa ottaa maali huomioon kuin dokumentoida ohjelmaan, että sitä ei tarvitse ottaa huomioon, ja se lisää resurssien kulutusta olennaisesti vain poikkeustapauksissa (maalin ei ole reittiä ja sen numero on paljon suurempi kuin minkään muun syötteessä esiintyvän solmun numero).

72. Jos rivillä 63 `solmu1` on maali, niin alkuperäinen algoritmi lopettaa. Siinä tapauksessa `solmu1` ei ole aikaisemmin ollut maali, joten keosta tullut etäisyys on maalin `etaisyys`. Siksi myös muutettu testi tuottaa `true`, joten myös muutettu algoritmi lopettaa.

    Jos `solmu1` ei ole maali ja muutettu testi tuottaa `false`, niin molemmat algoritmit käyttäytyvät samalla tavalla.

    Jäljellä on tapaus, jossa `solmu1` ei ole maali mutta muutettu testi tuottaa `true`. Siinä tapauksessa keosta tullut etäisyys on maalin etäisyys. Se on maalin minimietäisyys, koska muuten invariantin 3 mukaan keossa olisi solmu, jonka etäisyys keossa on pienempi kuin keosta juuri tullut etäisyys, mikä on ristiriidassa keon järjestyksen kanssa. Niinpä maalin tiedot on jo asetettu minimietäisyyden mukaan, joten lopettaminen on oikein.

73. Vaikea sanoa. Muutettu testi voi olla hiukan hitaampi kuin alkuperäinen, mutta ero on todennäköisesti merkityksetön. Toisaalta muutettu testi tuo merkittävää hyötyä vain jos usealla solmulla on täsmälleen sama minimietäisyys kuin maalilla, mikä lienee oikeiden maantiekarttojen tapauksessa harvinaista. Lisäksi testin muuttaminen monimutkaistaa algoritmin perustelemista oikeaksi, jolloin se riski kasvaa, että perustelussa on tehty virhe ja algoritmi meni sittenkin rikki.

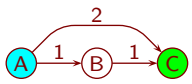
74. Käännetään kaaret takaperin ennen algoritmin suorituksen alkua.

Lisätietoa. Lisähyötynä saadaan, että muuttujiin `reitti` kerätyt tiedot esittävät reitit valmiiksi oikein päin.

75.
```
    TULOSTAREITTI(i,maali)
1   tulosta i
2   while i ≠ maali do
3     j := R[i,maali]
4     tulosta " "M[i, j] "km "j
5     i := j
```

76. Perustelu, jonka mukaan sama solmu ei voi tulla reittiin kahteen kertaan, lakkaa olemasta pätevä. Siitä voi seurata, että luvut $R[i,j]$ eivät esitä reittiä oikein. Esimerkiksi kun $i = k$ rivillä 5, sijoitetaan $R[i,j]$:hin alustamaton arvo $R[i,i]$.

77. Kuvassa ensimmäisellä kierroksella asetetaan $S[\mathrm{C}].r$:ksi A. Se ei muutu toisella kierroksella, koska muutos ei lyhentäisi reittiä.



78. Ajetaan Bellman–Ford graafille, jonka solmut ovat valuuttoja ja kaarten painoina on muunnoskurssien logaritmit, eli kullekin valuutoille i ja j on graafissa kaari pituudella $\log c_{i,j}$. Se selvittää, onko graafissa silmukkaa, jolle kurssien logaritmien summa on negatiivinen, eli kurssien tulo on alle 1.

79.
```
1   jonoon(i)
2   if S[i].m = false then
3     J[v] := i; v := (v+1) mod (n+1); S[i].m := true

1   jonosta()
2   i := J[w]; w := (w+1) mod (n+1); S[i].m := false; return i
```

80.
```
1   for i := 0 to n − 1 do
2     S[i].e := ääretön; J[i] := i; S[i].m := true
3   S[lähtö].e := 0; w := 0; v := n; kierros := 1
4   while kierros ≤ n && w ≠ v do
5     kierros := kierros + 1; u := v
6     while w ≠ u do
7       h := jonosta()
8       if h = 0 then j := 0 else j := S[h − 1].ℓ
9       while j < S[h].ℓ do
10        d := S[h].e + K[j].p; k := K[j].k; j := j + 1
11        if d < S[k].e then
12          S[k].e := d; S[k].r := h; jonoon(k)
```

81. Ei voi. Rivin 9 alussa pätee `ia[lo]` $>$ `elem` ja `ia[hi]` $\leq$ `elem`, koska rivien 7 ja 8 silmukat ovat lopettaneet. Niinpä `ia[lo]` $>$ `ia[hi]`. Rivin 9 `swap` suoritetaan vain jos `lo` $<$ `hi`, joten rivin 9 vaihtamat alkiot olivat väärässä järjestyksessä.

Myös rivin 12 alussa $\texttt{ia}[\texttt{hi}] \leq \texttt{elem}$, ja lisäksi $\texttt{ia}[\texttt{low}] = \texttt{elem}$ ja $\texttt{low} \leq \texttt{hi}$. Siksi rivin 12 vaihtamat alkiot olivat yhtäsuuret tai väärässä järjestyksessä.

Lisätietoa. Ei riitä todeta, että kun kaksi alkiota on vaihdettu keskenään SWAP:illa, ei niitä enää uudelleen vaihdeta keskenään eikä minkään muidenkaan alkioiden kanssa. Voihan olla, että ensin jompikumpi vaihtaa paikkaa jonkin kolmannen kanssa, ja myöhemmin toinen vaihtaa paikkaa jonkin neljännen kanssa. Todistuksen pitää tavalla tai toisella sulkea pois se mahdollisuus, että tässä skenaariossa kahden alkion järjestys vaihtuu ensin vääräksi ja sitten oikeaksi.

82. Ei voi. Jos $\texttt{lo} \geq \texttt{hi}$ rivillä 9, niin `for`-silmukka lopettaa, joten muita kierroksia ei tule. Muussa tapauksessa $\texttt{lo} < \texttt{hi}$ ja `swap` siirtää kohdassa `lo` olevan alkion kohtaan `hi`. Tämä alkio on suurempi kuin `elem` rivin 7 vuoksi. Niinpä `for`-silmukan seuraava kierros pysähtyy viimeistään kohdassa `hi`. Sivun 80 mukaan se on osassa $\texttt{ia}[\texttt{low} \ldots \texttt{high}]$.

83. Jos jokin osan `ia` alkio on suurempi kuin `elem`, niin rivin 7 silmukka pysähtyy siihen `for`-silmukan ensimmäisellä kierroksella. Vastauksessa 82 todettiin, että muilla `for`-silmukan kierroksilla rivin 7 silmukka ei voi ohittaa kohtaa `high`.

84. Jos kutsu ei ole rekursiotasonsa viimeinen, niin kohdassa $\texttt{high}+1$ on `ia`:n lokero. Tällöin $\texttt{ia}[\texttt{high}+1]$ sisältää jonkin aiemman osituksen jakoalkion. Koska `elem` sijoitettiin sen vasemmalle puolelle ja kaikki alkiot ovat keskenään erisuuret, pätee $\texttt{elem} < \texttt{ia}[\texttt{high}+1]$. Niinpä rivin 7 silmukan ehto ei päde, kun $\texttt{lo} = \texttt{high}+1$.

85. Kysymyksessä esitetty vaatimus pätee mutta oikeasti tarvittava asia ei päde esimerkiksi kun $A = [2,1]$, $i = 1$ ja $j = 0$. Nimittäin väite muotoa "X:n alkiot ovat enintään yhtäsuuria kuin Y:n alkiot" on epätosi jos ja vain jos jokin X:n alkio on suurempi kuin jokin Y:n alkio. Jos Y:ssä ei ole lainkaan alkioita, ei mikään X:n alkio voi olla suurempi kuin jokin Y:n alkio, joten väite ei ole epätosi vaan tosi. Tästä syystä kysymyksessä esitetty vaatimus pätee aina kun osa $A[i \ldots j]$ on tyhjä, riippumatta muiden osien sisällöistä. Oikeasti kuitenkin silloinkin tarvitaan myös, että ensimmäisen osan alkiot ovat enintään yhtäsuuria kuin kolmannen osan alkiot.

86. Kaikki päätyvät loppuosaan.

Lisätietoa. Ositus aloittaa vaihtamalla alussa olevia jakoalkion suuruisia alkioita ja lopussa olevia jakoalkiota pienempiä alkioita päikseen kunnes jommatkumpi loppuvat. Tehtävän alussa sanotun vuoksi yhtäsuuret loppuvat ensin tai molemmat loppuvat yhtäaikaa.

87. Kaikki voidaan saada alkuosaan. Niin käy jos aluksi on jakoalkiota pienemmät, sitten jakoalkiota suuremmat ja lopuksi jakoalkion suuruiset.

88. $s + \frac{k-s}{2}$ alaspäin pyöristettynä eli $\lfloor \frac{k+s}{2} \rfloor$. Nytkin aluksi on jakoalkiota pienemmät, sitten jakoalkiota suuremmat ja lopuksi jakoalkion suuruiset.

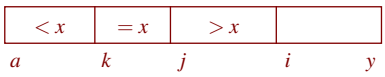
    Lisätietoa. Merkintä $\lfloor x \rfloor$ tarkoittaa suurinta kokonaislukua, joka on enintään yhtäsuuri kuin x. Ensin ohitetaan jakoalkiota pienemmät. Sitten vaihdetaan jakoalkiota suurempia päikseen jakoalkion suuruisten kanssa kunnes suuremmat loppuvat. Lopuksi vaihdetaan jakoalkion suuruisia pareittan päikseen kunnes selaukset kohtaavat.

    Niinpä QUICKSORT voi sijoittaa alkuosaan enemmän jakoalkion suuruisia alkioita kuin loppuosaan, mutta tämä ylimäärä ei voi olla isompi kuin jakoalkiota suurempien alkioiden määrä. Jos jakoalkion suuruisia alkioita on tätä enemmän, niin QUICKSORT jakaa loput tasan alku- ja loppuosaan. Samalla tavalla voidaan huomata, että QUICKSORT voi sijoittaa loppuosaan enemmän jakoalkion suuruisia alkioita kuin alkuosaan, mutta tämä ylimäärä ei voi olla isompi kuin jakoalkiota pienempien alkioiden määrä.

89. $[1,1]$

90. Ei. Vastaesimerkki: $[2,1,1]$.

91.
```
1  QS3(&A,a,y)
2  if a ≥ y then return
3  x := A[RANDOM(a,y)].x
4  j := a; k := a
5  for i := a to y do
6     if A[i].x ≤ x then
7        apu := A[i]; A[i] := A[j]
8        if apu.x = x then A[j] := apu
9        else A[j] := A[k]; A[k] := apu; k := k+1
10       j := j+1
11 QS3(A,a,k−1); QS3(A,j,y)
```



92. Jos alkuperäinen lista on kasvavassa järjestyksessä, niin suoritusaika on $\Theta(n^2)$ riippumatta siitä, onko siinä keskenään yhtäsuuria alkioita. Tämä johtuu siitä, että silloin yhtä vaille kaikki alkiot päätyvät suurten listaan, koska jakoalkioksi otetaan aina listan ensimmäinen alkio ja koska jakoalkion kanssa yhtäsuuret alkiot sijoitetaan suurten listaan. Lista, jonka kaikki alkiot ovat keskenään yhtäsuuret on kasvavassa järjestyksessä, joten sillekin suoritusaika on $\Theta(n^2)$. Se on $\Theta(n^2)$ myös kun alkuperäinen lista on aidosti vähenevässä järjestyksessä. Vähäisempi heikkous on, että muistia tarvitaan alkuperäisen listan lisäksi $\Theta(n)$ listoille `lesser` ja `greater`.

Lisätietoa. Kuvassa 39 käytetty ohjelmointityyli sopii hyvin listoille mutta ei kovin hyvin taulukoille, ja pikajärjestäminen on taulukoille eikä listoille suunniteltu algoritmi. Siksi ei pohjimmiltaan ole yllätys, että yhdistelmä ei ole kovin laadukas. Listoille paremmin toimiva algoritmi on lomitusjärjestäminen. Sitä käsitellään luvussa 7.1.

93. $N = [4,1,0,5,2,3]$

94. Jos $A[N[i]].x \neq A[N[j]].x$, niin vertailun tuloksena palautetaan se mitä saadaan kun verrataan $A[N[i]].x$ ja $A[N[j]].x$. Muussa tapauksessa vertailun tuloksena palautetaan se mitä saadaan kun verrataan $N[i]$ ja $N[j]$.

95. Nyt $N = [3,1,0,2]$. Vaihdot tuottavat [A, B, D, C], [A, B, D, C], [D, B, A, C] ja [D, B, C, A]. Lopputulos on $A = [\mathsf{D},\mathsf{B},\mathsf{C},\mathsf{A}]$. Se ei ole oikein.

96. [C, B, E, F, A, D] C [_, B, E, F, A, D] C [A, B, E, F, _, D] C [A, B, _, F, E, D] [A, B, C, F, E, D]

97. Aina kun alkio siirretään aukkoon, kasvaa oikeassa paikassa olevien alkioiden määrä yhdellä. Se ei kuitenkaan voi kasvaa suuremmaksi kuin alkioiden kokonaismäärä. Niinpä siirtelyn on pakko loppua joskus. Taulukossa olevan alkion siirtäminen aukkoon ei lopeta siirtelyä, koska silloin aukko vain siirtyy toiseen paikkaan taulukkoa. Siirtely loppuu ainoastaan sivuun otetun alkion siirtämisellä aukkoon. Menetelmän kuvauksen mukaan aukko on tällöin siinä paikassa, johon sivuun otettu alkio kuuluu.

98. Kun alkio siirretään oikeaan paikkaansa, merkitään, että siinä paikassa on se alkio joka siihen kuuluu. Se tapahtuu sijoittamalla $N[j] := j$. Tätä ei tarvitse tehdä kierroksen aloituspaikalle, koska lisätty silmukka on jo mennyt sen ohi.

99.
```
1  for i := 0 to A.koko − 2 do
2    if N[i] ≠ i then
3      apu := A[i]; j := i; k := N[j]
4      while k ≠ i do
5        A[j] := A[k]; j := k; k := N[j]; N[j] := j
6      A[j] := apu
```

100. Alussa $i = 0$. Jos $0 \leq h < n$ niin paikkaan h kuuluu $A[N[h]]$, koska N järjestettiin sellaiseksi, eikä A:ta eikä N:ää ole vielä muutettu. Osan $A[0 \ldots i-1]$ alkiot ovat oikeissa paikoissa, koska osa on tyhjä.

101. Lopussa $i = n-1$. Invariantti lupaa, että $A[0]$, …, $A[n-2]$ ovat oikeissa paikoissa. Koska paikoissa $0, \ldots, n-2$ on niihin kuuluvat alkiot, ei $A[n-1]$

voi kuulua mihinkään niistä. Ainoa muu paikka on $n-1$, joten $A[n-1]$ kuuluu siihen. Siksi myös $A[n-1]$ on oikeassa paikassa.

102.
```
    QUICKSELECT(&A,k)
 1  if k < 0 || k ≥ A.koko return virhe
 2  while true do
      ... kuvan 38 rivit 2, ..., 9
11    if      k < i then y := i − 1
12    else if k > j then a := j + 1
13    else return A[i]
```

103. Se toimii muuten kuten QUICKSELECT, mutta lisäksi se ylläpitää ja hyödyntää taulukkoa $B[0\ldots n]$, jonka alkiot ovat bittejä. Aluksi $B[0]$ ja $B[n]$ ovat true ja muut B:n alkiot ovat false. Aina kun muutettu QUICKSELECT osittaa osataulukon, se asettaa jälkimmäisen osan ensimmäisen alkion kohdalla B:ssä olevan bitin true:ksi. Jos syntyy myös keskimmäinen osa, niin senkin ensimmäisen alkion kohdalle B:hen sijoitetaan true. Muutettu QUICKSELECT valitsee ensimmäiset i:n ja j:n selaamalla B:tä kohdasta k taaksepäin ja kohdasta $k+1$ eteenpäin kunnes se löytää true:n, ja peruuttamalla jälkimmäistä yhden askeleen.

Lisätietoa. Jos tarvitaan vain suurimpia ja pienimpiä alkioita, niin luvussa 3.6 esitetty ratkaisu on parempi.

104.
```
1  alkio *poista(){
2    if( !viim ){ return 0; }
3    alkio *vanhin = viim->seur;
4    if( vanhin == viim ){ viim = 0; }
5    else{ viim->seur = vanhin->seur; }
6    return vanhin;
7  }
```

105. Edellä näytettiin, miten rengaslistaan voi lisätä tehokkaasti yhteen päähän ja poistaa tehokkaasti toisesta päästä. Kun lisäystoiminnosta jätetään pois jälkimmäinen lause `viim = uusi;`, saadaan tehokas lisääminen myös siihen päähän, josta voidaan poistaa tehokkaasti.

106. Tarvitaan vain yksi osoitin tavanomaisten osoittimien lisäksi: siihen alkioon, jonka perään lisätään. Esimerkissä se alkio on C.

107. Tarvitaan kaksi osoitinta tavanomaisten osoittimien lisäksi: siihen alkioon, jonka perästä poistetaan (esimerkissä A), sekä viimeiseen poistettavaan (esimerkissä E).

108.
```
unsigned i1 = maali, i2 = ~0u;
while( i2 != lahto ){
  unsigned i3 = i2;
  i2 = i1; i1 = solmut[ i1 ].reitti; solmut[ i2 ].reitti = i3;
}
```

109.
```
void countingsort( taulukko & A ){
  const unsigned k = 256;
  taulukko B( A.size() ); unsigned L[k] = {};
  for( unsigned i = 0; i < A.size(); ++i ){ ++L[ A[i].x ]; }
  for( unsigned j = 1; j < k; ++j ){ L[j] += L[ j-1 ]; }
  for( unsigned i = A.size(); i--; ){ B[ --L[ A[i].x ] ] = A[i]; }
  A.swap(B);
}
```

110. Taulukon L kooksi varataan 5 001. Kohtien $L[A[i].x]$ sijaan kirjoitetaan $L[A[i].x + 35\,000]$.