

# Järjestäminen

Ari Korhonen

## 8. Järjestäminen (sorting)

### [8.1 Johdanto](#)

### [8.2 Yksinkertaiset menetelmät](#)

### [8.3 Lomitus \(merging\)](#)

### [8.4 Lomitusjärjestäminen \(merge sort\)](#)

### [8.5 Pikajärjestämismenetelmä \(quicksort\)](#)

### [8.6 Valikointi \(selection\)](#)

### [8.7 Jakauman laskeminen \(Distribution counting\)](#)

### [8.8 Kantalukujärjestäminen \(radix sorting\)](#)

### [8.9 Järjestämismenetelmän valinnasta](#)

2/16/05

2

## 8.1 Johdanto

Järjestäminen eli *lajittelu* on tietojenkäsittelyn perusasioita.

**Syöte:** sarja numeroita  $\langle a_1, a_2, \dots, a_n \rangle$

**Tuloste:** sellainen numerosarjan permutaatio  $\langle a_1', a_2', \dots, a_n' \rangle$ , jolle  
 $a_1' \leq a_2' \leq \dots \leq a_n'$

Sovelluksia on lukemattomia:

- henkilörekisterit
- puhelinluettelo
- tilitapahtumat
- sähköpostiohjelma
- graafiset koordinaatit
- ...

2/16/05

3

## Perusajatukset:

- Järjestetään tiedostoja, jotka koostuvat tietueista
- Jokaisessa tietueessa on yksi tai useampi avain, jonka perusteella järjestetään
- Jos järjestetään keskusmuistissa, on kyse *sisäisestä järjestämisestä (internal sorting)*
- Jos järjestetään levyllä tai mg-nauhalla, koska tiedostot eivät mahdu keskusmuistiin, on kyse *ulkoisesta järjestämisestä (external sorting)*

2/16/05

4

- **Järjestämismenetelmän vaatima tietokoneaika on verrannollinen järjestettävien elementtien määrään seuraavasti:**

$O(N^2)$

- perusmenetelmät

$O(N \log N)$

- kehittyneet menetelmät

$\Omega(N \log N)$

- alaraja järjestämisalgoritmeille, jotka perustuvat kokonaisten avainten vertailuihin

$O(N)$

- jos avaimella on erikoisominaisuuksia, joita voidaan käyttää hyväksi bittitasolla

- **Järjestämismenetelmä on *stabiili*, jos se säilyttää samanlaiset avaimet omaavien tietueiden keskinäisen järjestyksen. Perusmenetelmät ovat yleensä stabiileja, kehittyneet menetelmät eivät**
- **Esimerkiksi, jos järjestän sähköpostiviestini ensin ajan suhteen ja sitten lähettäjän suhteen, stabiilissa menetelmässä saman lähettäjän viestit pysyvät aikajärjestyksessä.**
- **Jos lajiteltavat avaimet ovat kaikki erilaisia, ei stabiiliudella ole merkitystä**

**Järjestämismenetelmän valintaan vaikuttavia tekijöitä ovat:**

- tiedon määrä
- tietoalkion koko
- avaimen laatu, esim. integer / string
- järjestysaste
- menetelmän tehokkuus
- menetelmän tilantarve
- menetelmän monimutkaisuus

## 8.2 Yksinkertaiset menetelmät

- **Järjestämisen perusmenetelmät perustuvat *vierekkäisten* alkioiden vertailuun ja niiden vaihtamiseen keskenään. Vaihto voi olla eksplisiittinen kuten esim. kuplajärjestämisessä tai implisiittinen kuten lisäysjärjestämisessä.**
  - 8.2.1 Valintajärjestäminen
  - 8.2.2 Lisäysjärjestäminen
  - (8.2.3 Kuplajärjestäminen)
- **Toteutustavan mukaan eräät menetelmät ovat yksinkertaisia, vaikka ne toimivatkin tehokkaasti.**
  - (8.2.4 Kuorijärjestäminen)
  - 8.7 Jakauman laskeminen

### 8.2.1 Valintajärjestäminen (selection sort)

- Etsitään taulukon pienin alkio ja sijoitetaan se taulukon alkuun vaihtamalla sen paikka 1. alkion kanssa. Seuraavaksi etsitään jäljellä olevasta osuudesta taulukon toiseksi pienin alkio ja vaihdetaan se 2. alkion kanssa, jne.
- Vertailuja tarvitaan  $N^2/2$  kappaletta. Vaihtoja tarvitaan vain  $N$  kappaletta (tällä on merkitystä, jos keskenään vaihdettavat tietueet ovat suuria tai vaihto aiheuttaa jonkin muun työlään operaation).
- Viimeksi mainittu asia voidaan kiertää *epäsuoralla* järjestämisellä, jossa järjestetäänkin hakemisto eikä itse tietueita (vaihdetaankin osoittimia eikä tietueita).

```
for (i = 0; i < N-1; i++)
{
    min = i;
    for (j = i+1; j < N; j++)
        if (a[j] < a[min])
            min = j;
    temp = a[min];
    a[min] = a[i];
    a[i] = temp;
}
```

### 8.2.2 Lisäysjärjestäminen (insertion sort)

- Järjestettävä aineisto jakautuu kahteen osaan, jo järjestettyyn osaan ja sen jälkeiseen järjestämättömään osaan
- Joka askelella otetaan järjestämättömän osan ensimmäinen alkio, verrataan sitä askel askeleelta järjestettyyn osaan, kunnes oikea kohta löytyy. Alkio sijoitetaan tähän väliin, jolloin järjestynyt osa on kasvanut yhdellä

```
for (i = 1; i < N; i++)
{
    temp = a[i];
    j = i;
    while (j > 0 && a[j-1] > temp)
    {
        a[j] = a[j-1];
        j--;
    }
    a[j] = temp;
}
```

- Testi " $j > 0$ " on välttämätön, jotta indeksi ei juokse taulukon ulkopuolelle

- Se voidaan kuitenkin välttää sijoittamalla elementtiin  $a[0]$  *vartiosotilas (sentinel)*, joka on arvoltaan pienempi kuin taulukon muut alkio. Nyt while-silmukka ei juokse taulukon alarajasta läpi. Samalla koodi yksinkertaistuu ja tehostuu hieman.

```

a[0] = INT_MIN;
for (i = 2; i <= N; i++)
{
    temp = a[i];
    j = i;
    while (a[j-1] > temp)
    {
        a[j] = a[j-1];
        j--;
    }
    a[j] = temp;
}

```

2/16/05

13

- Järjestettävä data on alkioiden  $a[1] - a[N]$
- Sisäsilmutta suoritetaan keskimäärin  $N^2/2$  kertaa
- Lisäysjärjestämistä käytetään usein manuaalisessa järjestämisessä, esim. postinjakajilla. Tällöin lisäyskohta haetaan älykkäämmin ja lisäys voidaan toteuttaa yhdellä operaatiolla => algoritmin tehokkuus on noin  $N \log N$  manuaalisessa toteutuksessa

2/16/05

14

### 8.2.3 Kuplajärjestäminen (bubble sort)

- Joissakin ohjelmoinnin oppikirjoissa käytetään tätä esimerkkinä järjestämisalgoritmista. Valitettavasti...
- Perusidea on se, että verrataan pareittain vierekkäisiä taulukon alkioiden ja vaihdetaan tarvittaessa niiden paikkaa. Tällöin suurin alkio "kuplii" taulukon loppuun. Sitten toistetaan sama yhtä pienemmälle joukolle

```

for (i = N-1; i >= 0; i--)
    for (j = 1; j <= i; j++)
        if (a[j-1] > a[j]){
            temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
        }

```

2/16/05

15

**Kuplajärjestäminen on tehottomin perusmenetelmä. Sitä EI KANNATA KÄYTTÄÄ**

#### Teoreema 8.2.

Mikä tahansa algoritmi, joka järjestää alkioiden vaihtamalla vierekkäisiä alkioiden keskenään, vaatii pahimmassa tapauksessa aikaa  $\Omega(N^2)$

#### Todistus:

- Inversioiden lukumäärä on  $N(N-1)/4 = \Omega(N^2)$
  - Jokainen vierekkäisten alkioiden vaihto poistaa vain yhden inversion
- => Tarvitaan  $\Omega(N^2)$  vaihtoa

2/16/05

16

**Analyysiä:**

- Oletetaan järjestämätön lista arvoja, joiden joukossa ei ole duplikaatteja
- *Inversio* on tapaus, jossa kaksi listan alkioita on väärässä järjestyksessä. Esim.  $i < j$ , mutta  $a[i] > a[j]$ .

**Teoreema 8.1.**

Inversioiden lukumäärä satunnaisessa listassa on keskimäärin  $N(N-1)/4$ .

**Todistus:**

- Jokaista listaa  $L$  kohti on olemassa yksikäsitteinen käänteislista  $L_R$
- Oletetaan pari  $(x, y)$ ,  $x > y$  listassa  $L$ . Tällöin pari  $(y, x)$ ,  $x > y$  esiintyy käänteislistassa  $L_R$
- Yo. pari on inversio täsmälleen yhdessä listoista  $L$  ja  $L_R$
- Erilaisten parien lukumäärä kummassakin listassa yhteensä on  $(N-1) + (N-2) + \dots + 1 = N(N-1)/2$
- Keskimääräinen määrä inversioita listaa kohti on siis edellisen puolikas
- $\Rightarrow$  Inversioita on siis keskimäärin  $N(N-1)/4$  kappaletta

**Teoreema 8.2.**

Mikä tahansa algoritmi, joka järjestää alkioita vaihtamalla vierekkäisiä alkioita keskenään, vaatii pahimmassa tapauksessa aikaa  $\Omega(N^2)$

**Todistus:**

- Inversioiden lukumäärä on  $N(N-1)/4 = \Omega(N^2)$
  - Jokainen vierekkäisten alkioiden vaihto poistaa vain yhden inversion
- $\Rightarrow$  Tarvitaan  $\Omega(N^2)$  vaihtoa

Tämä on ns. *alarajatodistus (lower bound proof)*, joka pätee kaikille em. ehdon täyttävälle algoritmeille, riippumatta siitä, onko näitä edes keksitty.

Tehokkaammat järjestämisalgoritmit perustuvat siihen, että ne siirtävät alkioita kerralla kauemmas ja siten poistavat kerralla useampia kuin yhden inversion.

### 8.2.4 Kuorijärjestäminen (shell sort)

- Menetelmä perustuu siihen, että tiedosto *h-järjestetään* eri *h:n* arvoilla
- *H-järjestetyssä* tiedostossa ovat joka *H:n*nesta elementistä muodostuvat alijoukot järjestyksessä
- *H-järjestäminen* tapahtuu lisäsjärjestämisen avulla
- Kuorijärjestämisessä suoritetaan lisäys järjestäminen kaikille osajoukoille käyttäen peräkkäin tiettyjä *h*-arvoja
- Menetelmän tehokkuus perustuu siihen, että alkioita siirretään pitempiä matkoja

- Donald Shellin alkuperäiset inkrementit:

$$h_t = (\text{int})(N / 2), h_k = h_{k+1} / 2$$

$$\Rightarrow O(N^2) \text{ pahin tapaus}$$

- Empiirisesti löydetty hyvä *h*-jono on ns. Hibbardin inkrementit

$$1093, 364, 121, 40, 13, 4, 1 \quad \text{eli } h_n = 3h_{n-1} + 1$$

$$\Rightarrow O(N^{3/2}) \text{ pahin tapaus}$$

- Sedgewickin inkrementit

$$1, 5, 19, 41, 109, \dots$$

$$\Rightarrow \text{pahin tapaus } O(N^{4/3})$$

- Avoimia kysymyksiä:

- Voidaanko löytää parempia sekvenssejä?
- Keskimääräisestä tehokkuudesta ei ole tarkkaa tietoa. On arvioitu eri tapauksille  $O(N^{5/4})$ ,  $O(N^{7/6})$  tai  $O(N (\log N)^2)$ .

### 8.3 Lomitus (merging)

- Kaksi järjestyksessä olevaa tiedostoa yhdistetään uudeksi tiedostoksi, joka on myös järjestyksessä

Tyyppitilanne:

- Rekisteriin, joka on järjestetty, lisätään uusia elementtejä
- Ne kootaan ensin tapahtumatiedostoon (myös järjestyksessä)
- Nämä yhdistetään määräjoihin
- Yhdistely lineaarisessa ajassa, two-way merging:

$$a[0..M], b[0..N] \Rightarrow c[0..M+N]$$

- Valitaan aina pienin alkio a:sta ja b:stä ja kirjoitetaan se c:hen. Taulukkojen lopussa vahtisotilaat.

```

i = 0; j = 0;
a[M] = INT_MAX; b[N] = INT_MAX;

for (k = 0; k < M + N; k++)
    if ( a[i] < b[j] ) {
        c[k] = a[i];
        i = i + 1;
    }
    else {
        c[k] = b[j];
        j = j + 1;
    }

```

- Vaatii aikaa  $O(M + N)$

## 8.4 Lomitusjärjestäminen (merge sort)

- Yhdistetään järjestäminen ja lomitus:

1. Jaa tiedosto kahtia
2. Järjestä puoliskot rekursiivisesti
3. Yhdistele puoliskot yhdeksi

(vertaa Quicksort)

- Divide & Conquer-algoritmi  
 $T(N) = 2T(N/2) + N, T(1) = 1$
- Tehokkuus aina  $N \log N$
- Lyhyt sisin silmukka

- **Stabiili**
- **MUTTA** vaatii ylimääräisen muistitilan
- Mergesort soveltuu hyvin tilanteisiin, joissa tietoon päästään käsiksi vain peräkkäisjärjestyksessä (esim. linkitetty listat, ulkoiset tiedostot).
- Voidaan toteuttaa myös ei-rekursiivisena

```

void mergesort(int left; int right)
{
    int i, j, k, m;

    if (right > left)
    {
        m = (right + left) / 2;
        mergesort(left, m);
        mergesort(m + 1, right);
        for (i = m; i >= left; i--)
            b[i] = a[i];
        for (j = m + 1; j <= right; j++)
            b[right + m + 1 - j] = a[j];
    }
}

```

```

i = left;
j = right;
for (k = left; k <= right; k++)
    if (b[i] < b[j])
        {
            a[k] = b[i];
            i = i + 1;
        }
    else
        {
            a[k] = b[j];
            j = j - 1;
        }
}

```

2/16/05

29

**Mergesort linkitetyille listoille:**

```

link mergesort(link c; int N)
{
    link alkuosa, loppuosa;

    if (c->next == z)
        return c;
    else
        {
            alkuosa = c;
            for (i = 2; i <= (N / 2); i++)
                c = c->next;
        }
}

```

2/16/05

30

```

loppuosa = c->next;
c->next = z;

return merge(
    mergesort(alkuosa, N / 2),
    mergesort(loppuosa, N - (N/2)));
}

link merge(link a; link b){...}

```

- **two-way merge linkitetyille listoille**

2/16/05

31

**8.5 Quicksort****Kehittäjä C.A.R. Hoare 1960**

- Tunnetuin ja yleisin tehokas järjestämismenetelmä
- Keskimääräinen tehokkuus  $N \log N$
- Pahin tapaus  $N^2$ , mutta se voidaan välttää pienillä virityksillä
- Kohtuullisen helppo toteuttaa perusmuodossaan

2/16/05

32

**PERUSIDEA:**

- **Partitioni** muokkaa taulukkoa

**a[left..right]**

ja hakee siitä kohdan *i* siten, että

- 1) alkio **a[i]** on kohdallaan
- 2) alkiot **a[left] .. a[i-1] ≤ a[i]**
- 3) alkiot **a[i+1] ..a[right] ≥ a[i]**

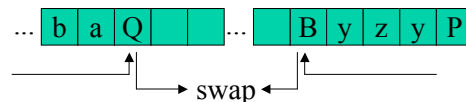
- Tämän jälkeen partitointi suoritetaan molemmille puoliskoille rekursiivisesti

**Quicksort on *divide & conquer* -algoritmi**

```
void quicksort(int left; int right)
{
    int i;
    if (right > left)
    {
        i = partition(left, right);
        quicksort(left, i-1);
        quicksort(i+1, right);
    }
}
```

**Partitiofunktion toiminta:**

1. Valitaan alueen oikeasta reunasta **partitioalkio** (*pivot-alkio*)
2. Verrataan siihen alkioita yhtäaikaan taulukon vasemmasta ja oikeasta reunasta alkaen
3. Vaihdetaan vasemmalla olevat sitä suuremmat alkiot oikealla oleviin sitä pienempiin alkioihin
4. Pivot-alkion kanssa yhtäsuuret alkiot osallistuvat vaihtoon (ei välttämätöntä, mutta tasapainottaa algoritmin toimintaa)



```
int partition(int left; int right)
{
    int i, j, pivot, apu;

    pivot = a[right];
    i = left - 1;
    j = right;
    do
    {
        do
            i++;
        while (a[i] < pivot);
        do
            j--;
        while (a[j] > pivot);
    }
}
```

```

    apu = a[i];
    a[i] = a[j];
    a[j] = apu;
} while (j > i);

a[j] = a[i];
a[i] = a[right];
a[right] = apu;
return i;
}

```

## Pahin tapaus

- Kun partitiointi “epäonnistuu”
- Toiseen puoleen jää  $N-k$  alkioita ja toiseen vain  $k$  (vakio)
- Jos tämä tapahtuu (lähes) jokaisella partitiointikierröksellä ( $\Theta(N)$  operaatio) saadaan rekursioyhtälö (kun  $k=1$ )

$$T(N) = T(N-1) + N, \quad T(1) = 1$$

$$T(N) = N + (N-1) + (N-2) + \dots + 1 = \Theta(N^2)$$

- Epätasapainoinen partitiointi johtaa neliölliseen käyttäytymiseen!
- Tehtävä: Anna esimerkki syötteestä, jolla tämä tapahtuu.

## Ongelmia ja virityksiä

- 1) Jos taulukko on valmiiksi järjestyksessä, partitiot degeneroivat ja kyseessä on  $N^2$  – algoritmi

⇒ Valitaan partitiioelementti *mediaani-kolmesta* -menetelmällä tai satunnaisesti

- 2) Sama tilanne johtaa siihen, että suoritetaan  $N$  rekursiokutsua, jolloin ajonaikaisessa pinossa on  $N$  aktiivatiotietuetta

⇒ Poistetaan rekursio ja työnnetään suuremman alueen rajat itse pinon (pienempi käsitellään suoraan). Pinon kooksi tulee korkeintaan  $\log N$ .

## 3) Rekursio suoritetaan pienillekin partitioidelle

⇒ Vaihetaan lisäysjärjestämiseen, kun  $\text{right} - \text{left} < M$

tai

⇒ Lopetetaan vastaavassa tilanteessa quicksort ja järjestetään lopuksi koko taulukko lisäysjärjestämällä

## Paras tapaus

- Partitiointi jakaa aineiston tasan kahteen osaan

$$T(N) = 2T(N/2) + N, T(2) = 1$$

ratkaistaan esim. sijoituksella  $N=2^k$ , ja jakamalla molemmat puolet  $2^{k-1}$ :llä, kuten opetusmonisteissa, tai induktiolla:

- osoitetaan, että  $T(N) \leq N \log N$ , kun  $N > N_0$  (alkuarvaus)

- pätee pienillä  $N$ :n arvoilla, jostakin  $N_0$  lähtien, tässä  $N_0 = 4$  ja esim.  $T(4) = 2T(2) + 4 = 6 \leq 4 \log(2^2) = 8$

- induktioaskel (induktio-oletus:  $T(N/2) \leq (N/2) \log(N/2)$ )

$$T(N) = 2 * (N/2) * \log(N/2) + N = N \log(N/2) + N = N \log N - N + N = N \log N$$

- Parhaassa tapauksessa QuickSort on  $O(N \log N)$  algoritmi

## 8.6 Valikointi (selection)

- "Hae mielivaltaisessa järjestyksessä olevasta tietoaaineistosta  $k$ :nneksi suurin alkio". Esim. "Hae aineiston mediaani"
- Prioriteettijonon avulla  $O(N + k \log N)$ 
  - Mediaanin haku  $O(N \log N)$
- Täsmällinen järjestäminen vie turhaa aikaa
- Jos  $k$  on pieni, valintajärjestäminen on hyvä menetelmä
- Mielivaltaiselle aineistolla voidaan soveltaa quicksortin idea: Partitoidaan aineisto ja otetaan jatkoon se osuus, johon  $k$  sisältyy.

```
int Select(int left; int k; int right) {
    int i = -1;
    if (left < right) {
        i = partition(left, right);
        if (k < i)
            return Select(left, k, i-1);
        if (k > i)
            return Select(i+1, k, right);
    }
    return i;
}
```

- Keskimääräinen kompleksisuus  $O(N)$

## 8.7 Jakauman laskeminen (Distribution counting)

- $N$  tietuetta, joissa kokonaislukuavaimia välillä  $0..M-1$ .  $N > M$ ; Esim.  $a = \langle x, y, x, y, z, y, z \rangle$ ,  $N = |S| = 7$ ,  $M = 3$
- "Etsitään samojen avainarvojen alueet taulukossa"

1) Aluelaskurien alustus;  $\text{count} = \langle 0, 0, 0 \rangle$

```
for (i=0; i<M; i++)
    count[i] = 0;
```

2) Lasketaan avainten frekvenssit;  $\text{count} = \langle 2, 3, 2 \rangle$

```
for (i=0; i<N; i++)
    count[a[i]]++;
```

**3) Lasketaan kumulatiiviset frekvenssit; count = < 2,5,7 >**

```
for (j=0; j<M; j++)
    count[j] += count[j-1];
```

**4) Sijoitetaan avaimet alueisiinsa; t = < x,x,y,y,y,z,z >  
(a=<x,y,x,y,z,y,z>)**

```
for (i=N-1; i>=0; i--) {
    t[ count[a[i]]-1 ] = a[i]; // t[6]=z,t[4]=y,t[5]=z...
    count[a[i]]--;
}
```

**5) Kopioidaan aputableluko alkuperäiseen taulukkoon; a = t**

```
for (i=0; i<N; i++)
    a[i] = t[i];
```

**8.8 Kantalukujärjestäminen (radix sorting)**

- Järjestäminen perustuu siihen, että käsitellään ja vertaillaan avaimen osia
  - Esimerkki: Avaimet ovat kokonaislukuja välillä 0-999. Järjestetään dekaideittain.
  - Tietokoneessa avaimia käsitellään binäärilukuina eli järjestetään bittipositio/bittiryhmä kerrallaan
- ⇒ Ominaisuuden hyväksikäyttö vaatii tehokasta pääsyä bittitason toimintoihin.
- Jos avaimet ovat lyhyitä, kannattaa käyttää jakauman laskemismenetelmää. Tarvittavan aputablelkon count koko on  $2^b$ .

**Tärkeimmät menetelmät ovat:**

- digitaalinen vaihtojärjestäminen
- suora digitaalinen järjestäminen

**C-simulaatio bittioperaatioille:**

$n \gg k$

- Siirrä (shift) k bittiä oikealle

$n \& (2^k-1)$

- Ota (mask) viimeiset k bittiä

$\text{int bits}(n, k, j) = (n \gg k) \& (2^j-1)$

- Palauttaa sanan osan, jossa on j bittiä alkaen positiosta k oikealta laskien
- Esim:  $\text{bits}(n, 0, 2) = 2$  viimeistä bittiä

### 8.8.1 Digitaalinen vaihtojärjestäminen (radix exchange sort)

- Tiedot järjestetään bittipositio kerrallaan vasemmalta oikealle
- Kukin positio jaetaan kahteen osaan keräämällä nollat ja ykköset omiin ryhmiin
- Osat järjestetään edelleen rekursiivisesti seuraavasta bitistä lähtien
- Jos osan kooksi tulee 1, sitä ei järjestetä eteenpäin

### Esimerkki 1

010011001	↓	010011001
101100101	←	010010000
010010000	←	101100101
101110001	↓	101110001
111000101	↓	111000101

- Tehokkuus  $N \log N$  tai  $N*b$  ( $b$  = bittimäärä)
- Jos avaimet ovat satunnaisesti jakautuneita, ei kaikkia bittejä tarvitse tutkia
- Log N bittiä riittää, jolloin menetelmä on jopa nopeampi kuin Quicksort
- Samoilla avaimilla käydään läpi kaikki bitit, jolloin tehokkuus laskee

```
void radix_exchange(int left; int right; int pos)
{
    if ((right > left) && (pos >=0)) {
        i = left;
        j = right;

        do {
            while (((bits(a[i],pos,1) = 0) && (i < j)))
                i++;
            while (((bits(a[j],pos,1) = 1) && (i < j)))
                j--;
            swap(a[i], a[j]);

        } while (j != i);
    }
}
```

```

if (bits(a[right],b,1) = 0)
    j++;

radix_exchange(left,j-1,pos-1);
radix_exchange(j, right,pos-1);
}
}

```

- Menetelmä muistuttaa Quicksortia!

## 8.8.2 Suora digitaalinen järjestäminen (straight radix sort)

- Järjestetään bittipositio kerrallaan oikealta vasemmalle
- Käyttää jakauman laskemista  $M$ :n bitin osuuksille ( $M \geq 1$ )
- Perustuu osaltaan siihen, että jakauman laskeminen on stabiili menetelmä
- Jos  $M$  on yksi, järjestetään bitti kerrallaan. Tehokkuus paranee, jos  $M$  kasvaa
- Hyvä suositus on  $M = b / 4$ , jossa  $b$  on avainten bittien lukumäärä

### Esimerkki 2

010011001	010010000	111000101	010010000
101100101	010011001	010010000	010011001
010010000	101110001	010011001	101100101
101110001	101100101	101100101	101110001
111000101	111000101	101110001	111000101

- Esim. 32-bittisiä kokonaislukuja järjestetään tavu kerrallaan
- Tehokkuusluku on  $N$ , eli kyseessä on lineaarinen menetelmä

#### Mutta:

- vaatii lisätilaa  $N$  alkiota
- hidas sisin luoppi

## 8.9 Järjestämismenetelmän valinnasta

### Yhteenveto tehokkuusluvuista

	W(N)	A(N)
Selection sort	$N^2 / 2$	$N^2 / 2$
Insertion sort	$N^2 / 2$	$N^2 / 4$
Shellsort	$c_0 N^{4/3}$	$c_0 N^{7/6} *$
Distribution counting	$c_0 N$	$c_0 N$
Quicksort	$N^2 / 2$	$1.38 N \log N$
Radix exchange sort	$c_0 N \log N$	$c_0 N \log N$
Straight radix sort	$c_0 N$	$c_0 N$
Heapsort	$2.0 N \log N$	ei tietoa
Mergesort	$1.0 N \log N$	$1.0 N \log N$

\* Shellsortin h-jono 1, 5, 19, 41, 109, ...

### Huom!

- Kaikki laskettavat operaatiot eivät ole keskenään suoraan verrannollisia
- Kaikille algoritmeille ei löytynyt vakiokertoimen  $c_0$  arvoa

### Muita ominaisuuksia

	Lisämuistin tarve	stabiilius
Selection sort	Ei	On
Insertion sort	Ei	On
Shellsort	Ei	Ei
Distribution count	$N + \text{count}$	On
Quicksort	Ei	Ei
Radix exchange sort	Ei	Ei
Straight radix sort	$N + 2^m$	On
Heapsort	Ei	Ei
Mergesort	N tai osoitt.	On

### Huom!

- Mergesort voidaan toteuttaa ilman lisätilaa (monimutkaista)
- Quicksort ja Radix exchange sort voidaan toteuttaa stabiileina (monimutkaista)

### Tehokkuusluku ei kerro kaikkea! On muitakin näkökulmia:

#### Selection sort

- Yksinkertainen
- Pienelle tiedostolle, jossa suuret tietueet
- Voidaan soveltaa valikointiongelmaan, kun k on pieni

#### Insertion sort

- Yksinkertainen
- Hyvä menetelmä pienille tiedostoille ( $N < 50$ )
- Tehokas (lineaarinen) menetelmä, jos tiedosto 'melkein järjestyksessä'

#### Shellsort

- Yksinkertainen
- Tehokas vielä kohtalaisen suurilla tiedostoilla ( $N < 100000$ )

**Distribution counting**

- Yksinkertainen
- Tilanteisiin, joissa avainavaruus pienempi kuin avainten määrä
- Vaatii lisätilaa

**Quicksort**

- Käytännön valinta useimpiin suuriin sovelluksiin
- Partitioelementtiä ei tule valita triviaalisti laidoista
- Viritykset ohjelmitava huolellisesti
- Ei käy, jos tarvitaan stabiiliutta
- Huolelliset viritykset, jos ehdoton vaatimus  $N \log N$ -käyttäytymisestä
- Voidaan soveltaa valikointiongelmaan lineaarisessa ajassa

**Radix exchange sort**

- Implementaatio edellyttää pääsyä bitti-informaatioon kiinni
- Voidaan toteuttaa myös muissa kantaluvuissa (bucket sort)
- Tehokas, jos aineisto satunnainen
- Tehokkuus kärsii, jos aineistossa runsaasti samoja avainarvoja
- Ei kannata käyttää merkkijonoavaimille

**Straight radix sort**

- Implementaatio edellyttää pääsyä bitti/tavuinformaatioon kiinni
- Tehokas
- Vaatii lisätilan, joka voi olla rasite suurilla tiedostoilla
- Ei kannata käyttää merkkijonoavaimille

**Heapsort**

- Taattu  $N \log N$
- Sopii hyvin ongelmaan, jossa halutaan löytää aineistosta  $k$  pienintä alkioita ( $k \ll N$ )

**Mergesort**

- Taattu  $N \log N$
- Vaatii lisätilan
- Sopii tilanteisiin, joissa dataa käsitellään peräkkäisesti
- Lomitus on ulkoisten järjestämismenetelmien perusalgoritmeja

**Ensi kerraksi...**

- Tutki onko käyttämäsi sähköpostiohjelman lajittelumenetelmä stabiili

Ensi tiistaina (22.2.) suunnittelutehtävän esittely

- Jussi Nikander

Luennot jatkuvat 1.3. (luennoija vaihtuu)

- Lauri Malmi