## 6.2 Lomitusjärjestäminen

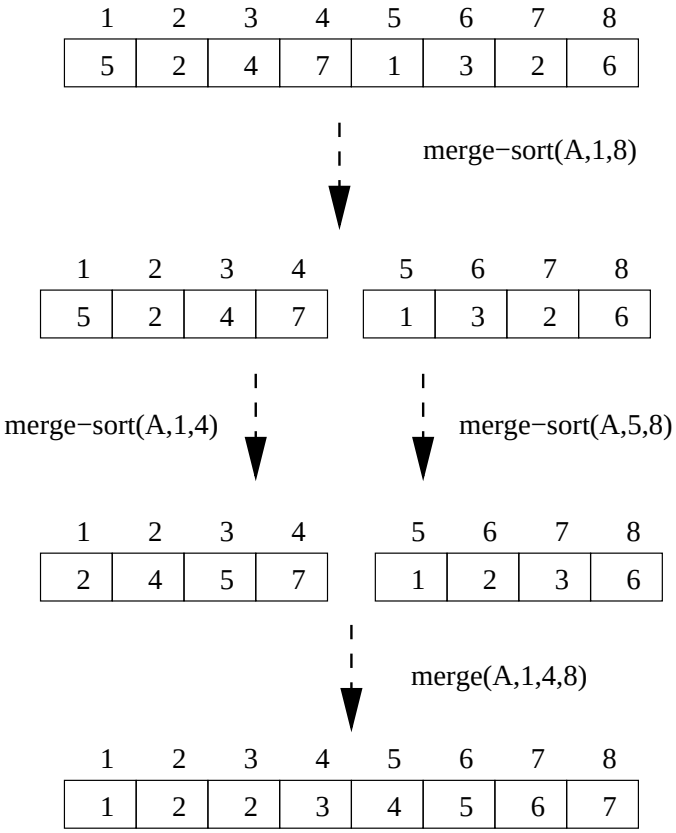
- lomitusjärjestäminen perustuu *hajoita-ja-hallitse* (engl. divide-and-conquer) tekniikkaan:
  - *hajoitetaan* ongelma pienempiin osaongelmiin
  - *hallitaan*, eli ratkaistaan osaongelmat rekursiivisesti
  - *yhdistetään* osaratkaisut siten että saadaan ratkaisu koko ongelmalle
- taulukko A[1,n] järjestetään kutsumalla merge-sort(A,1,n):

```
merge-sort(A,p,r)
1   if p<r then
2       q ←⌊(p + r)/2⌋
3       merge-sort(A,p,q)
4       merge-sort(A,q+1,r)
5       merge(A,p,q,r)
```

- toimintaidea
  - operaation tehtävä on järjestää taulukon A osa A[p,q]
  - jos järjestettävän osan pituus on korkeintaan 1 (eli p $\geq$ q), ei tehdä mitään, sillä tällöin haluttu taulukon osa on valmiiksi järjestyksessä (rivin 1 if-ehto)
  - rivillä 2 asetetaan q käsiteltävän taulukon osan keskikohtaan
  - taulukon osat A[p,q] ja A[q+1,r] järjestetään kutsumalla niille merge-sort operaatiota rekursiivisesti
  - riville 5 tultaessa siis taulukon osat A[p,q] ja A[q+1,r] ovat järjestyksessä

– rivillä 5 kutsutaan operaatiota merge joka "lomittaa" osaratkaisut siten että A[p,q] saadaan järjestykseen

- esimerkki:




- järjestyksessä olevien taulukon osien A[p,q] ja A[q+1,r] lomittaminen järjestykseen on helppo tehdä:
  - kopioidaan A[p,q] aputaulukkoon L[1,$n_1$]
  - ja A[q+1,r] aputaulukkoon R[1,$n_2$]
  - laitetaan paikkaan A[p] pienin alkioista L[1] ja R[1],
  - jos L[1] laitettiin taulukkoon, niin paikkaan A[p+1] laitetaan pienin alkioista L[2] ja R[1]

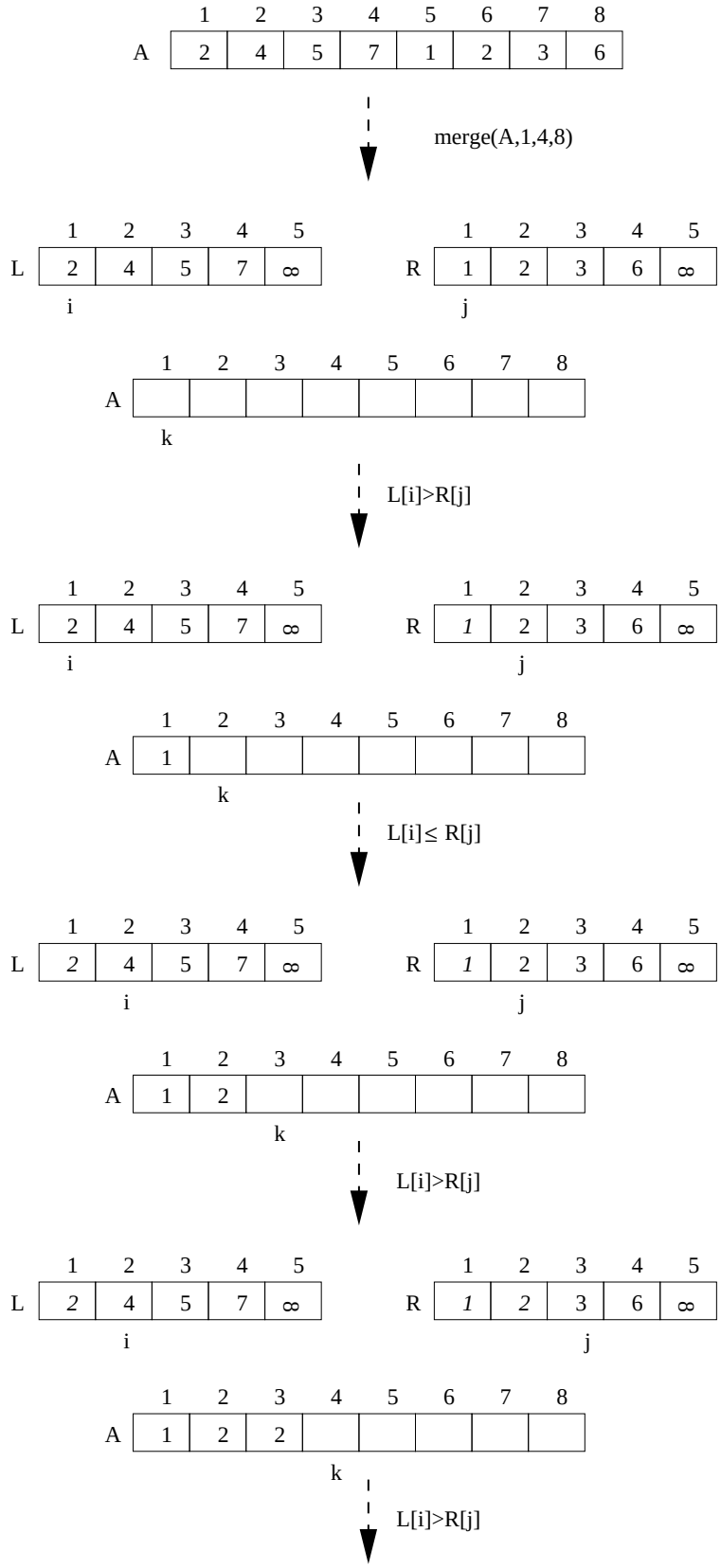
    jos taas R[1] laitettiin taulukkoon, niin paikkaan A[p+1] laitetaan pienin alkioista L[1] ja R[2]

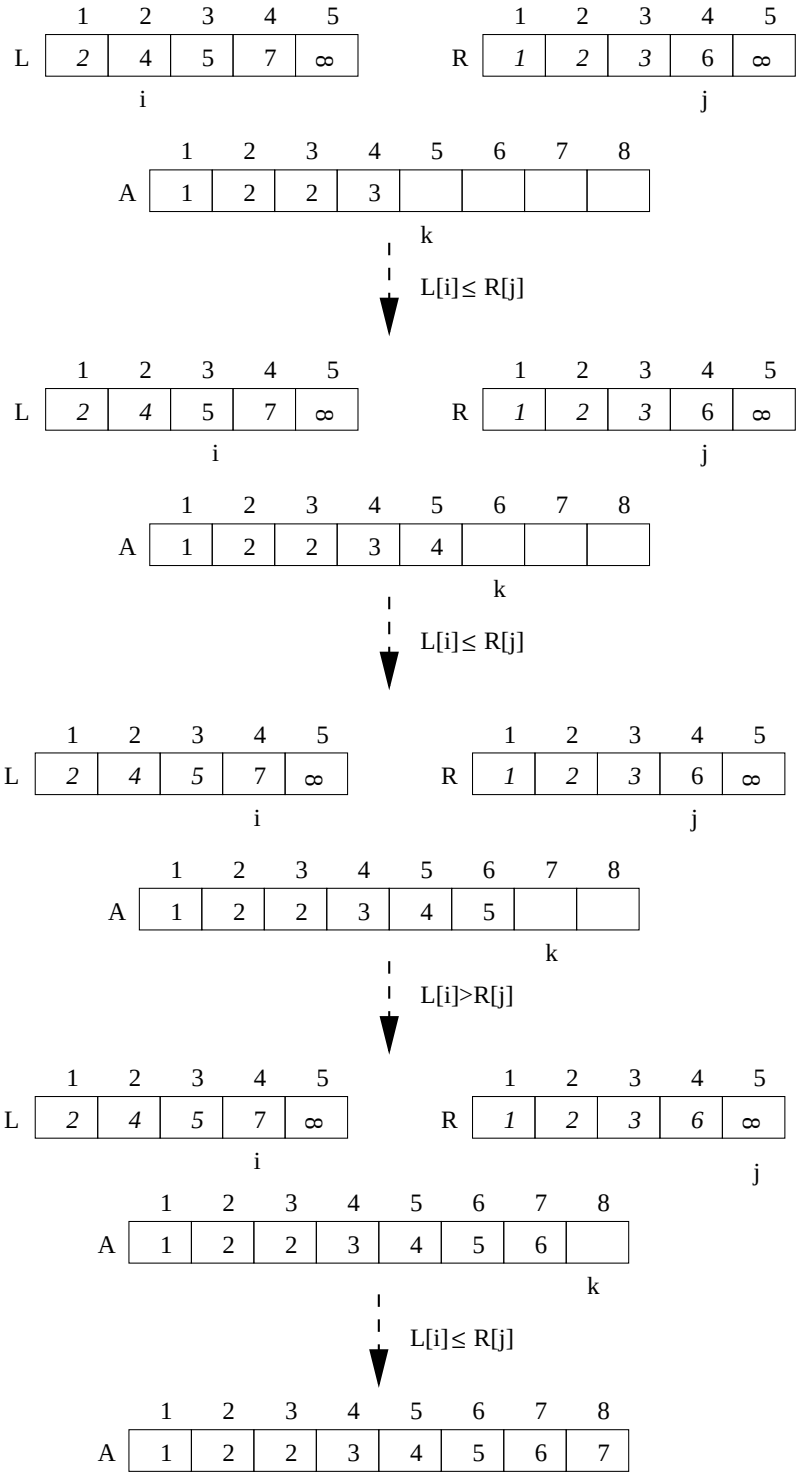
– näin jatketaan kunnes kaikki aputaulukoiden alkiot on siirretty taulukkoon A

- algoritmina

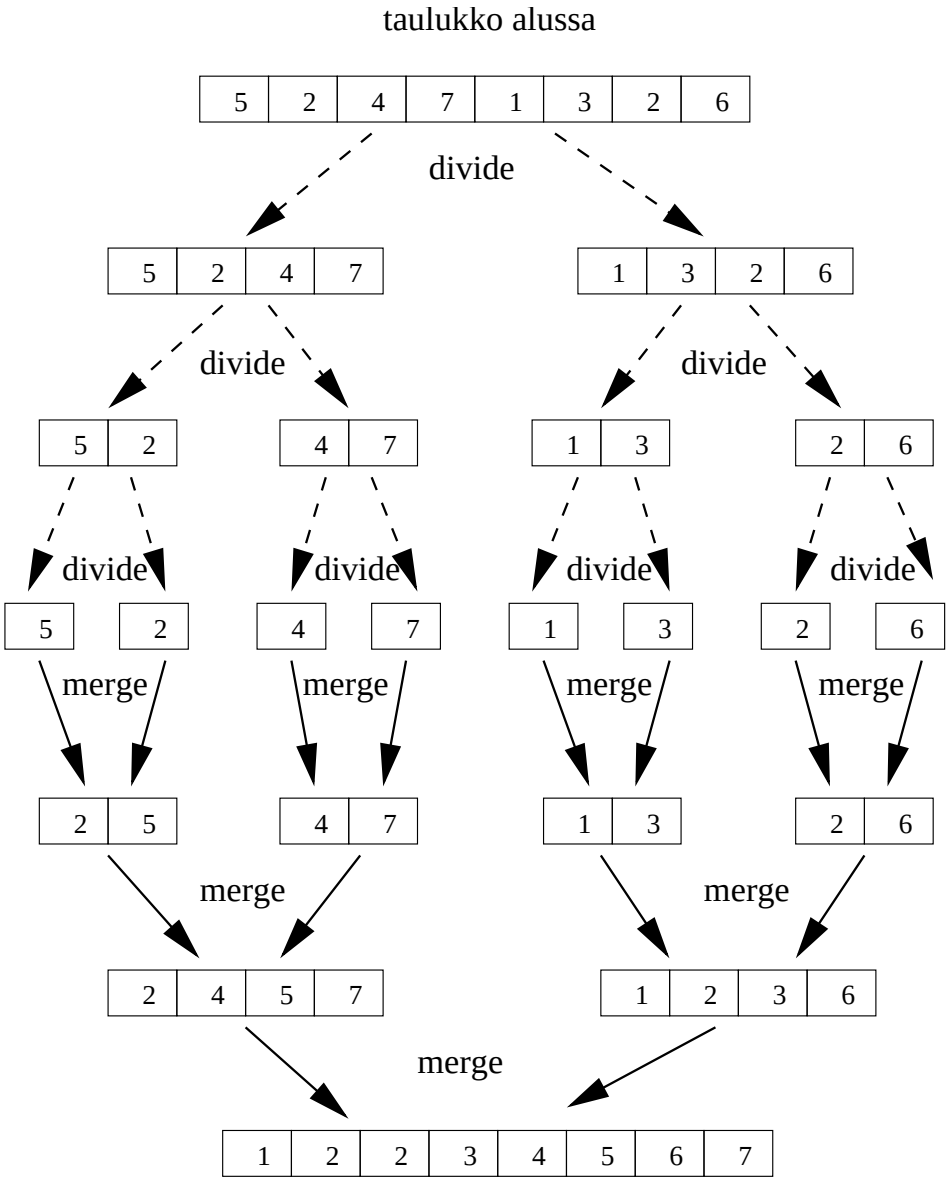
```
merge(A,p,q,r)
1   n1 ←q−p+1
2   n2 ←r−q
3   ▷luodaan taulukot L[1,..., n1+1] ja R[1,...,n2+1]
4   for i ←1 to n1 do
5       L[i] ←A[p+i−1]
6   L[n1+1] ←∞
7   for j ←1 to n2 do
8       R[j] ←A[q+j]
9   R[n2+1] ←∞
10  i ←1
11  j ←1
12  for k ←p to r do
13      if L[i] ≤ R[j] then
14          A[k] ←L[i]
15          i←i+1
16      else
17          A[k] ←R[j]
18          j←j+1
```

- esimerkki merge-operaation toiminnasta:


1 2 3 4 5 6 7 8
A 2 4 5 7 1 2 3 6
merge(A,1,4,8)
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A
k
L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1
k
L[i]≤ R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2
k
L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2
k
L[i]>R[j]


1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3
k
L[i]≤ R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3 4
k
L[i]≤ R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3 4 5
k
L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3 4 5 6
k
L[i]≤ R[j]
1 2 3 4 5 6 7 8
A 1 2 2 3 4 5 6 7

- esimerkki koko algoritmin toiminnasta:



- ennen kuin analysoimme koko lomitusjärjestämisen aikavaativuutta, tarkastellaan mikä on merge-operaation vaativuus
  - olkoon $n$ lomitettavan taulukonosan pituus
  - operaatio luo kaksi aputaulukko L:n ja R:n kooltaan yhteensä $n_1 + 1 + n_2 + 1 = n + 2 = \mathcal{O}(n)$
  - rivien 4 ja 7 for-lauseiden vaativuus yhteensä $\mathcal{O}(n_1 + n_2) = \mathcal{O}(n)$
  - rivin 12 for toistetaan n kertaa tehden toisto-osassa vakiomäärä operaatioita, myöhempi for-lause siis myös $\mathcal{O}(n)$
- merge-operaation aikavaativuus sekä tilavaativuus siis $\mathcal{O}(n)$, missä $n$ lomitettavan taulukonosan pituus $r - p + 1$

- entä koko lomitusjärjestämisen aikavaativuus?
- tehdään yksinkertaistava oletus: järjestettävän taulukon koko on jokin kahden potenssi, jokainen jako siis puolittaa taulukon kahteen yhtäsuureen osaan
- Käytetään $k$:n kokoisen taulukon lomitusjärjestämisen vaativuudesta merkintää $T(k)$
- $k$:n kokoisen taulukon lomitusjärjestämisen vaativuus on nyt sama kuin kahden $k/2$ kokoisen taulukon järjestämisen vaativuus + taulukon osien lomittaminen
- yhden kokoisen taulukon lomitusjärjestämiseksi ei tarvitse tehdä mitään
- eli, voimme määritellä $T$:n *rekursioyhtälönä*:

$$
\begin{aligned}
T(1) &= \mathcal{O}(1) \\
T(k) &= T(k/2) + T(k/2) + \mathcal{O}(k), \quad \text{kun } k > 1
\end{aligned}
$$

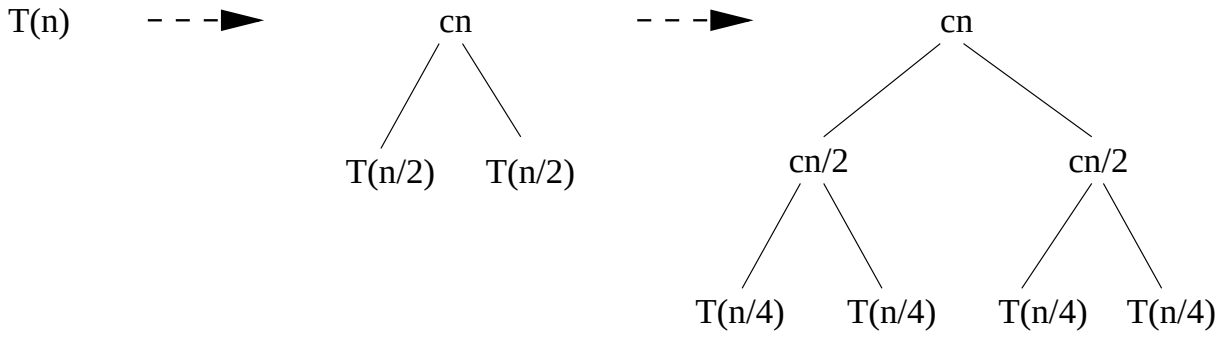
- Taulukon A[1,n] lomitusjärjestämisen aikavaativuus saadaan selville ratkaisemalla rekursioyhtälö $T(n)$
- kirjoitetaan rekursioyhtälö hieman toisin, käyttämättä $\mathcal{O}$-termejä

$$
\begin{aligned}
T(1) &= c \\
T(k) &= T(k/2) + T(k/2) + ck, \quad \text{kun } k > 1
\end{aligned}
$$

- missä $c$ on vakio
- aletaan laskemaan auki rekursioyhtälöä:

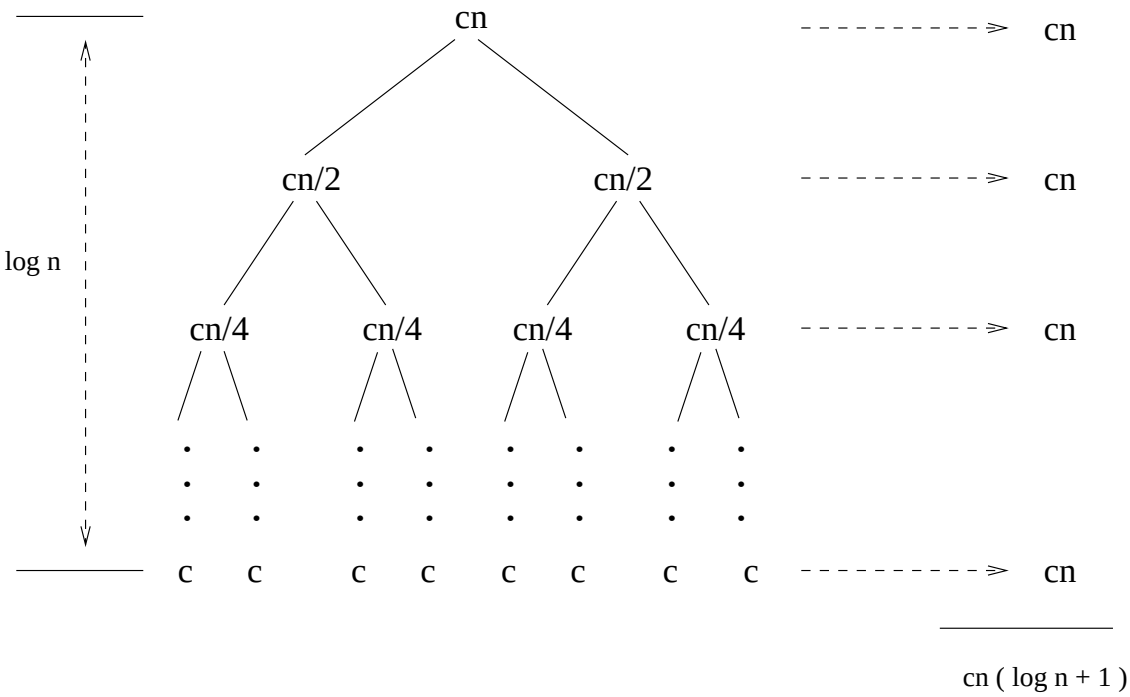
$$
\begin{aligned}
T(n) &= T(n/2) + T(n/2) + cn \\
&= T(n/4) + T(n/4) + cn/2 + T(n/4) + T(n/4) + c/2 + cn \\
&\quad \ldots
\end{aligned}
$$

- havainnollisempaa on muodostaa *rekursio-puu*:




- merkataan rekursiopuun solmuihin kyseisen rekursio-instanssin vaativuus

- jatketaan rekursiopuun aukipiirtämistä niin kauan että tullaan yhden kokoisen taulukon järjestämistä vastaaviin lehtisolmuihin



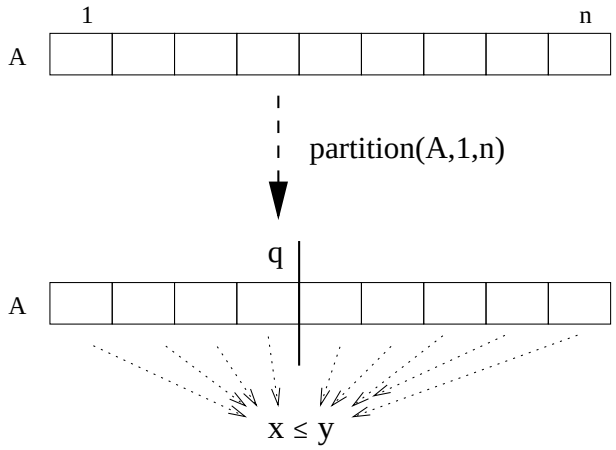

- huomaamme että rekursio-puun jokaisen tason vaativuus on $cn$
- rekursio-puu on täydellinen binääripuu jolla $n$ lehteä, sivulla 49 todistetusta Lauseesta 1 seuraa nyt suoraan että rekursiopuun korkeus on $log\ n$ eli puulla on $log\ n + 1$ tasoa
- lomitusjärjestämisen vaativuus saadaan siis laskemalla yhteen kaikkien rekursiotasojen vaativuus, joka on $cn(log\ n + 1)$, eli
- lomitusjärjestämisen aikavaativuus $\mathcal{O}(n\ log\ n)$
- algoritmi voidaan toteuttaa myös ilman rekursiota, tällöin tilavaativuus on $\mathcal{O}(n)$, sama kuin ylimmän tason merge-operaatiolla

## 6.3 Pikajärjestäminen

- sovelletaan edelleen hajoita-ja-hallitse -periaatetta
- taulukko A[1,n] järjestetään kutsumalla quick-sort(A,1,n):

```
quick-sort(A,p,r)
1   if p<r then
2       q ←partition(A,p,r)
3       quick-sort(A,p,q)
4       quick-sort(A,q+1,r)
```

- toimintaidea
  - operaation tehtävä on järjestää taulukon A osa A[p,q]
  - jos järjestettävän osan pituus on korkeintaan 1 (eli p $\geq$ q), ei tehdä mitään, sillä tällöin haluttu taulukon osa on valmiiksi järjestyksessä (rivin 1 if-ehto)
  - rivillä 2 kutsutaan partition-operaatiota joka jakaa taulukon kahteen osaan:




  - jaon jälkeen kaikki alkuosan A[p,q] alkiot ovat *korkeintaan yhtä suuria* kuin loppuosan A[q+1,r] alkiot

- huom: toisin kuin lomitusjärjestämisessä, taulukon osat A[p,q] ja A[q+1,r] eivät ole välttämättä saman kokoisia!
- taulukon osat A[p,q] ja A[q+1,r] järjestetään kutsumalla niille rekursiivisesti quick-sort operaatiota
- tulosten kokoamisvaihetta ei tarvita, sillä alkuosan ja loppuosan alkiot ovat jo partition-operaation jäljiltä keskenään oikeassa järjestyksessä

- pikajärjestämisen tehokkuuden kannalta on oleellista että partition-operaatio toimii nopeasti (lineaarisessa ajassa jaettavaan taulukon-osan koon suhteen) ja tuottaa mahdollisimman tasaisia jakoja

- käytetään seuraavaa partition-operaatiota

```
partition(A,p,r)
1  a ←A[p]
2  i ←p-1
3  j ←r+1
4  while true do
5      repeat i ←i+1 until A[i] ≥ a
6      repeat j ←j−1 until A[j]≤ a
7      if i<j then vaihda A[i] ja A[j]
8          else return j
```

- toimintaidea
  - valitaan alussa *jakoalkioksi* vasemmanpuoleisimman alkion A[p] arvo $a$
  - ideana on nyt että jaon jälkeen kaikki vasemman puolen alkiot ovat suuruudeltaan korkeintaan $a$ ja oikean puolen alkiot ovat suuruudeltaan vähintään $a$

– lähdetään kulkemaan taulukkoa kummastakin päästä läpi, indeksi $i$ alusta loppuun ja indeksi $j$ lopusta alkuun

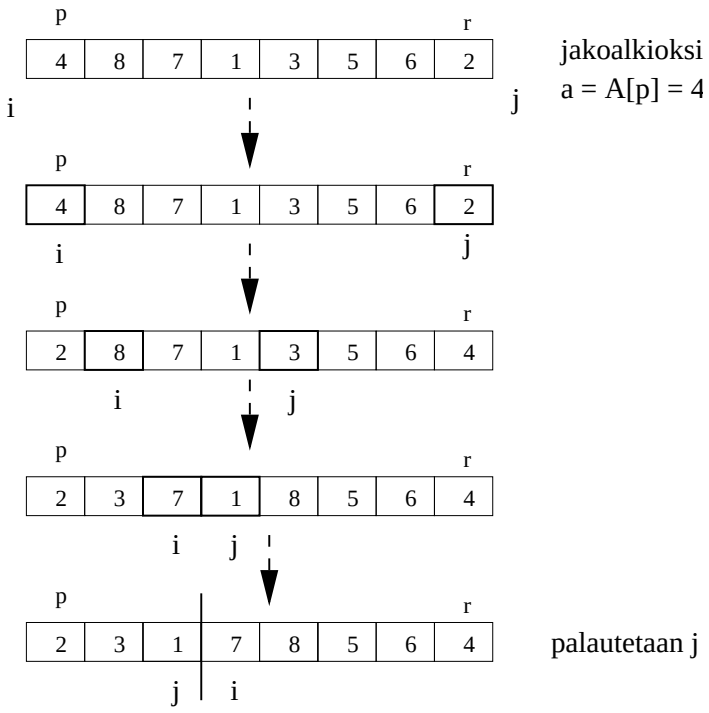
– rivillä 5 viedään $i$ osoittamaan alustapäin ensimmäistä alkiota jonka arvo on suurempi tai yhtä suuri kuin jakoalkio A[i]$\geq$ a

– rivillä 6 viedään $j$ osoittamaan lopustapäin ensimmäistä alkiota jonka arvo on korkeintaan jakoalkio A[j]$\leq$a

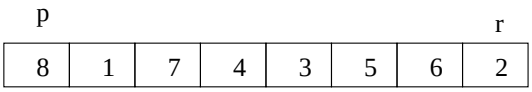
– jos indeksi $i$ on vielä indeksin $j$ vasemmalla puolella, vaihdetaan alkioiden A[i] ja A[j] arvot

– jos $i \geq j$, on tilanne se että kaikki $j$:n oikealla puolella olevat taulukon alkiot ovat vähintään $a$:n suuruisia ja $j$:n vasemmalla puolella olevat taas korkeintaan $a$:n suuruisia, eli voimme lopettaa ja palautetaan $j$, sillä se osoittaa oikeaa jakokohtaa

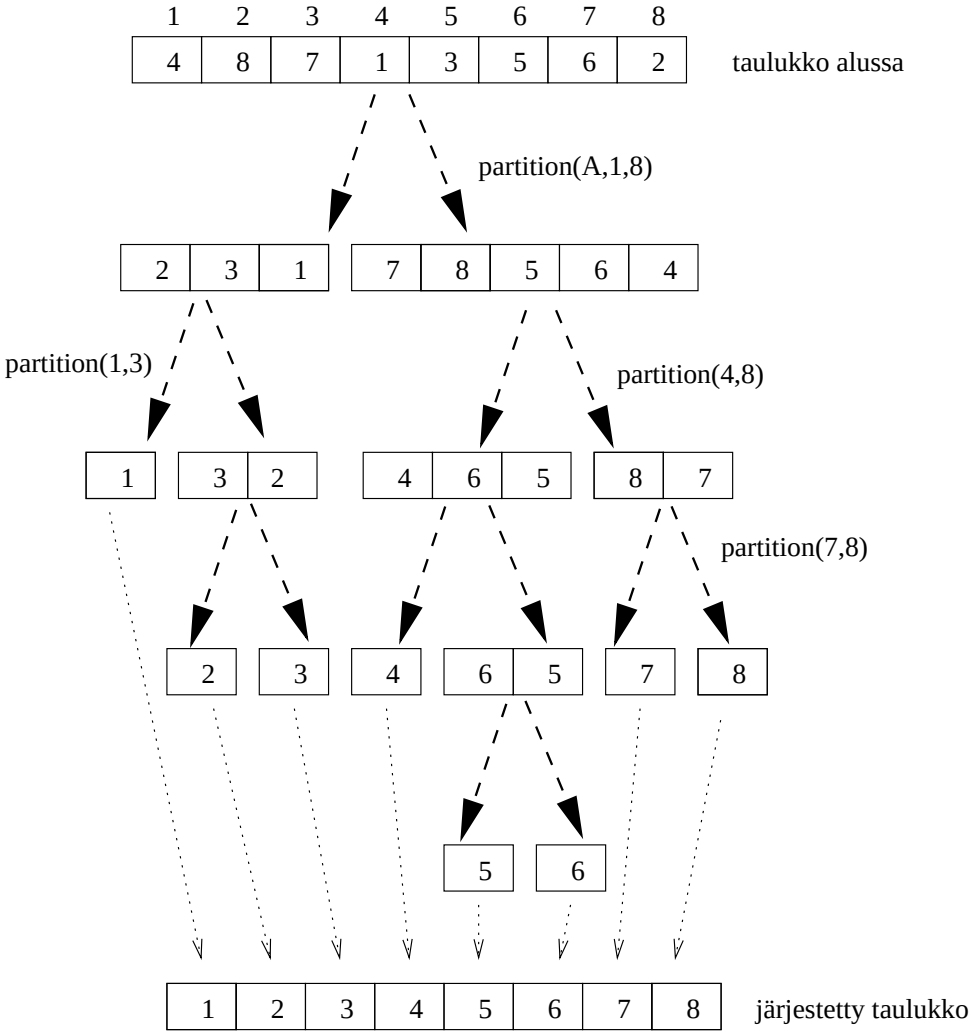
• esimerkki partition-operaation toiminnasta:



- selvästi partition-operaation aikavaativuus $\mathcal{O}(n)$ suhteessa käsiteltävän taulukon pituuteen $n$, aputilaa ei tarvita kuin parin muuttujan verran eli tilavaativuus $\mathcal{O}(1)$
- esimerkissämme meillä oli melko hyvä tuuri, taulukko jakautui kohtuullisen tasan kahtia, entä jos kyseessä olisi ollut seuraava taulukko?




- esimerkki pikajärjestämisen toiminnasta



- mikä on pikajärjestämisen aikavaativuus?
- pahimmassa tapauksessa partition jakaa $m$:n pituisen taulukon kahtia siten että toisessa osassa on vain 1 alkio
- pahimman tapauksen vaativuus $T_w(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$\begin{aligned} T_w(1) &= c \\ T_w(k) &= T_w(1) + T_w(k-1) + ck, \quad \text{kun } k > 1 \end{aligned}$$

- missä $c$ on vakio eli termi $ck$ kuvaa partition-operaation vaativuutta
- lasketaan auki rekursioyhtälöä:

$$\begin{aligned} T_w(n) &= T_w(1) + T_w(n-1) + cn \\ &= c + T_w(1) - T_w(n-2) + c(n-1) + cn \\ &= c + c + T_w(1) + T_w(n-3) + c(n-2) + c(n-1) + cn \\ &\quad \ldots \\ &= cn + c\sum_{i=1}^{n} i = cn + \frac{cn(n+1)}{2} = \mathcal{O}(n^2) \end{aligned}$$

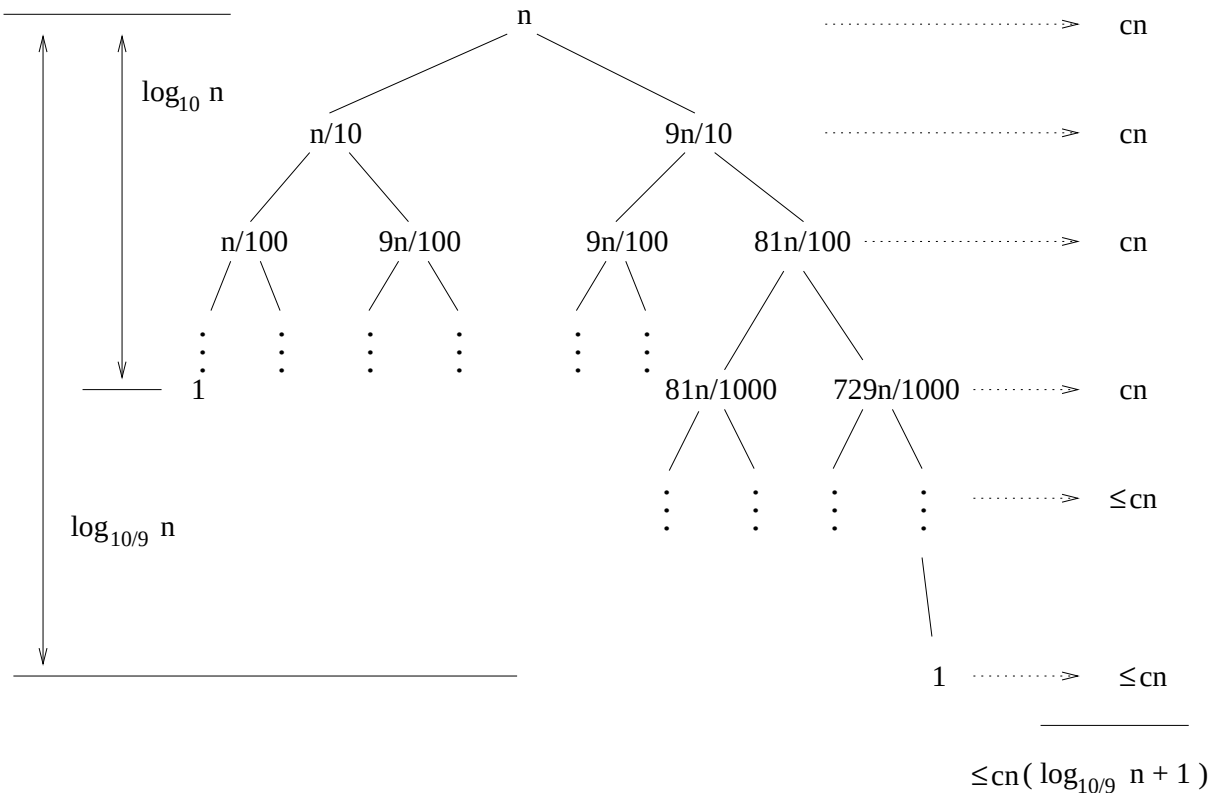
- pahimmassa tapauksessa pikajärjestäminen toimii neliöisesti, eli on selvästi huonompi kuin lomitus- ja kekojärjestäminen
- entä parhaassa tapauksessa, missä partition-operaation sattuu jakamaan taulukon aina kahteen yhtäsuureen osaan?
- parhaan tapauksen vaativuus $T_b(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$
\begin{aligned}
T_b(1) &= c \\
T_b(k) &= T_b(k/2) + T_b(k/2) + ck, \quad \text{kun } k > 1
\end{aligned}
$$

- yhtälö on täsmälleen sama kuin lomitusjärjestelmän vaativuutta kuvannut yhtälö, eli parhaan tapauksen vaativuus on $\mathcal{O}(n\ log\ n)$
- pikajärjestämisen pahin tapaus on kuitenkin erittäin harvinainen, ja käytännössä pikajärjestys toimii yleensä paremmin kuin lomitus- tai kekojärjestäminen
- *keskimääräisen tapauksen vaativuus* pikajärjestämisellä onkin $\mathcal{O}(n\ log\ n)$
- analysoidaan vielä tilannetta missä partition jakaisi alkioita melko huonosti, eli oletetaan että $m$:n alkion taulukko jakautuisi pahimmillaan siten että pienemmässä osassa on vain $m/10$ alkiota ja suuremmassa $9m/10$
- näytetään että jopa tässäkin tapauksessa pikajärjestämisen vaativuus olisi $\mathcal{O}(n\ log\ n)$
- "epätasapainoisten jakojen" tapauksen vaativuus $T_u(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$
\begin{aligned}
T_u(1) &= c \\
T_u(k) &= T_u(k/10) + T_u(9k/10) + ck, \quad \text{kun } k > 1
\end{aligned}
$$

- kirjoitetaan rekursioyhtälöä auki rekursiopuu-muodossa, solmuihin merkitty nyt rekursiokutsua vastaavan taulukonosan pituus




- Saamme siis epätasapainoisten jakojen tapauksen aikavaativuudeksi $T_u(n) \leq cn(log_{10/9}\ n + 1) = cn(\frac{log\ n}{log 10/9} + 1) \leq cn(22 \times log\ n\ + 1) = 22 \times cn(\ log\ n\ + \frac{1}{22}) = \mathcal{O}(n\ log\ n)$
- näinkin epätasapainoinen partitiointi johtaa vielä $\mathcal{O}(n\ log\ n)$ aikavaativuuteen, tosin rekursiotasoja tulee noin 22 kertaa enemmän kuin täysin epätasapainoisissa jaoissa

- pikajärjestämisen tilavaativuus on sama kuin rekursion syvyys pahimmillaan, eli keskimääräisessä tapauksessa $\mathcal{O}(log\ n)$ ja pahimmassa tapauksessa $\mathcal{O}(n)$
- pikajärjestäminen voidaan toteuttaa myös ilman rekursiota jolloin päästään tilavaativuuteen $\mathcal{O}(1)$

- muutama käytännön huomautus:
- lisäysjärjestäminen on lyömätön pienillä aineistoilla
- pikajärjestämistä kannattaakin viritellä siten että jos järjesteltävän taulukonosan pituus on enää esim. alle 20, järjestetäänkin tämä osa käyttäen lisäysjärjestämistä, muuten toimitaan kuten pikajärjestämisessä normaalistikin

```
quick-sort(A,p,r)
1  if r−p<20 then insertion-sort(A,p,q)
2  else
3       q ←partition(A,p,r)
4       quick-sort(A,p,q)
5       quick-sort(A,q+1,r)
```

- partition-operaatiossa paras valinta jakoalkioksi olisi jaettavan aineiston mediaani, tällöin jako olisi mahdollisimman tasainen
- mediaanin selvittäminen ei kuitenkaan onnistu lineaarisessa ajassa
- melko hyvä keino on valita jako-alkioksi mediaani arvoista A[p], A[$\lfloor (p+q)/2 \rfloor$] ja A[q]
- nyt esim. valmiina järjestyksessä tai käänteisessä järjestyksessä olevan aineiston pikajärjestäminen onnistuukin ajassa $\mathcal{O}(n\ log\ n)$
- pahinta tapausta pikajärjestämisessä ei kuitenkaan voida välttää, mutta pahin tapaus voidaan tehdä niin harvinaiseksi ettei se esiinny käytännössä juuri koskaan

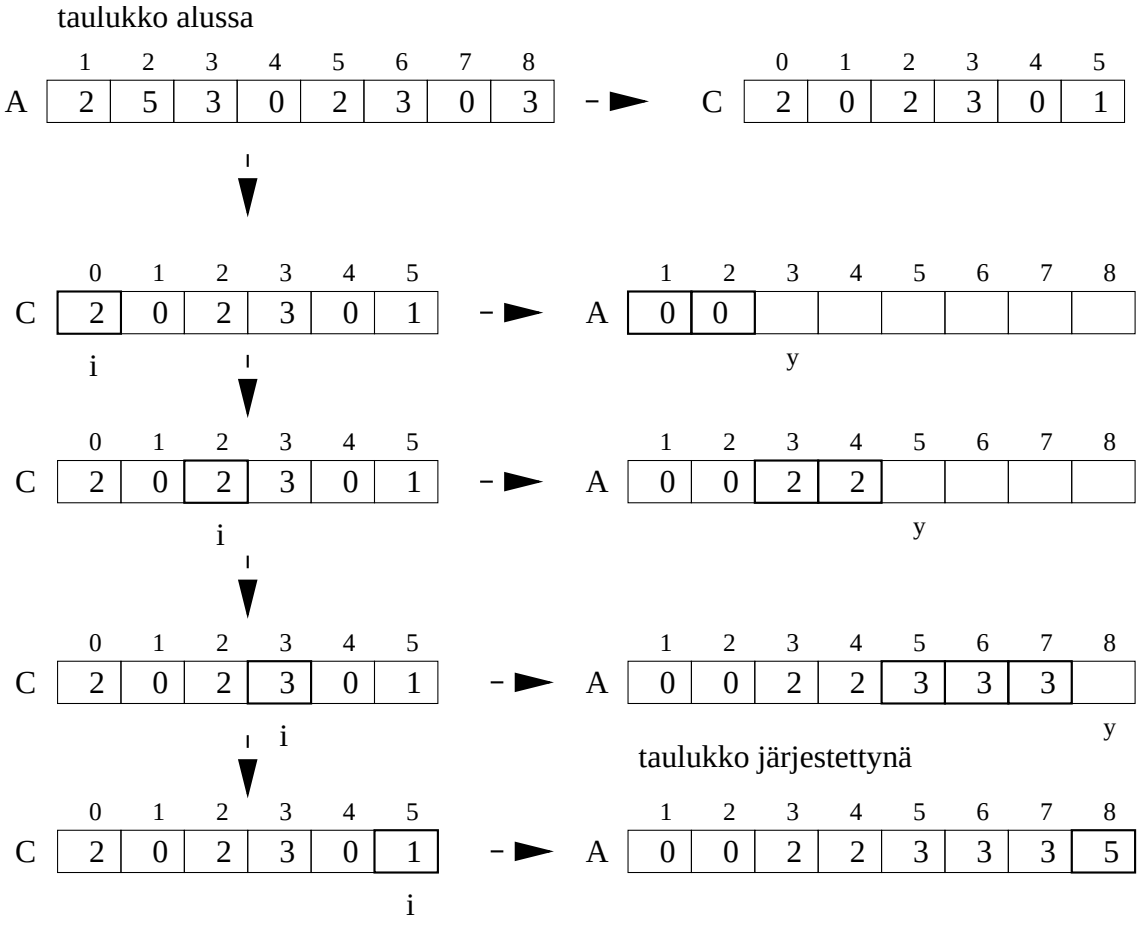
## 6.4 Järjestäminen lineaarisessa ajassa

- kaikki tähän asti tuntemamme järjestämisalgoritmit perustuvat järjestettävän aineiston lukujen keskinäiseen vertailuun
- voidaan todistaa että vertailuihin perustuva järjestämisalgoritmi ei voi toimia pahimmassa tapauksessa nopeammin kuin ajassa $\mathcal{O}(n\ log\ n)$, ks. Cormen luku 8.1 tai Karvi 6.5
- koska lomitus- ja kekojärjestäminen toimivat pahimmassa tapauksessa ajassa $\mathcal{O}(n\ log\ n)$, ovat ne optimaalisen tehokkaita

- tehokkuusraja $\mathcal{O}(n\ log\ n)$ voidaan kuitenkin rikkoa jos järjestäminen perustuu johonkin muuhun kuin alkioiden keskinäiseen vertailuun
- oletetaan että järjestettävä aineisto A[1,n] koostuu luvuista joiden arvo on väliltä $0, \ldots, k$
- yksinkertainen ja tehokas järjestämismenetelmä saadaan aikaan seuraavasti:
  - otetaan käyttöön aputaulukko C[0,k]
  - käydään A läpi ja lasketaan kuinka monta kertaa kukin luku esiintyy, luvun i esiintymien määrä talletetaan paikkaan C[i]
  - tulostetaan sitten taulukkoon A ensin C[0] kertaa luku 0, C[1] kertaa luku 1 jne
  - näin taulukossa on samat luvut kuin alussa ja luvut ovat suuruusjärjestyksessä!
- algoritmina:

```
counting-sort1(A,k,n)
1   for i ←0 to k do C[i]←0
2   for j ←1 to n do
3       x ←A[j]
4       C[x] ←C[x]+1
5   y ←1
6   for i ←0 to k do
7       for j ←1 to C[i] do
8           A[y] ←i
9           y ←y+1
```

- esimerkki:



- algoritmi käy kerran läpi taulukon A, kahteen kertaan taulukon C ja tulostaa luvun jokaiseen A:n paikkaan, aikavaativuus siis $\mathcal{O}(n+k)$ ja jos $k = \mathcal{O}(n)$ niin aikavaativuus on lineaarinen järjestettävän aineiston koon suhteen
- tilavaativuus luonnollisesti $\mathcal{O}(k)$
- kutsutaan algoritmia *laskemisjärjestämiseksi* (counting sort), kyseessä ei kuitenkaan ole sama versio laskemisjärjestämisestä mikä löytyy Cormenista ja Karvista
- jos järjestettäviin alkioihin liittyy muita datakenttiä ei juuri esitetty versio laskentajärjestämisestä toimi

- esitetään vielä laskemisjärjestämisestä Cormenin versio joka välttää yllä mainitun ongelman

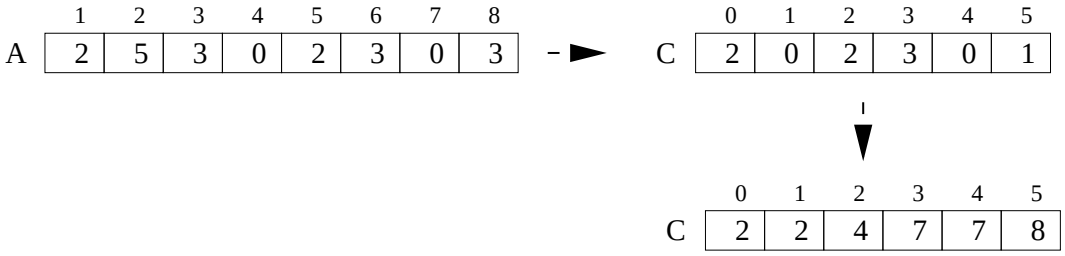
```
counting-sort2(A,k,n)
1   for i ←0 to k do C[i]←0
2   for j ←1 to n do
3       x ←A[j]
4       C[x] ←C[x]+1
5   for i ←1 to n do
6       C[i] = C[i]+C[i-1]
7   for j ←downto 1 do
8       x ←A[j]
9       B[C[x]]←x
10      C[x] ←C[x]-1
11  for i ←1 to n do A[i] ←B[i]
```

- toimintaidea:

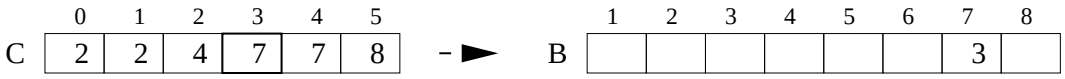
  - algoritmi käyttää aputaulukkoa B[1,n]
  - rivien 3-4 for-lauseen jälkeen C[i] sisältää tiedon kuinka monta lukua $i$ taulukossa A on
  - rivien 5-6 for-lauseen jälkeen C[i] sisältää tiedon kuinka monta *korkeintaan yhtä suurta* lukua kuin $i$ taulukossa A on
  - rivien 7-10 for-lauseke järjestää A:n alkiot taulukkoon B
  - laitetaan ensin paikalleen paikan A[n] alkio x
  - C[x] kertoo kuinka monta korkeintaan x:n suuruista alkiota taulukossa A on
  - x on siis C[x]:nneksi suurin A:n alkioista, eli laitetaan x paikkaan B[C[x]]
  - vähennetään vielä arvoa C[x] yhdellä jotta seuraava paikalleen laitettava saman suuruinen alkio menee oikealle paikalleen
  - jatketaan laittamalla paikoilleen alkio A[n-1]
  - lopuksi kopioidaan järjestetyt alkiot taulukosta B takaisin taulukkoon A

- algoritmi käy kahteen kertaan läpi molemmat taulukot A ja C sekä kertaalleen läpi taulukon B, aikavaativuus siis $\mathcal{O}(n+k)$
- aputaulukon B koko on n ja aputaulukon C koko on k eli tilavaativuus $\mathcal{O}(n+k)$
- edelleen, jos $k = \mathcal{O}(n)$, on molempina vaativuuksina $\mathcal{O}(n)$
- esimerkki Cormenin laskemisjärjestyksen toiminnasta seuraavalla sivulla:

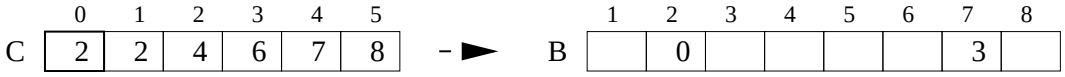
taulukko alussa



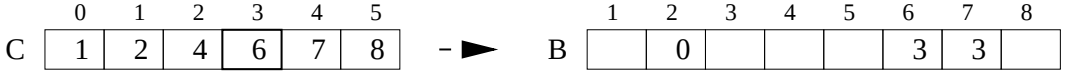

A[8]=3



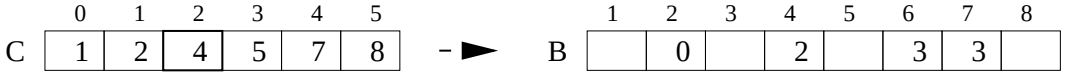

A[7]=0



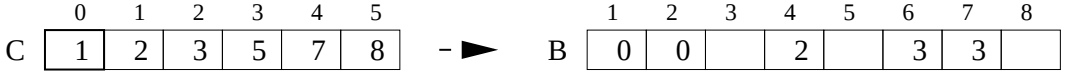

A[6]=3



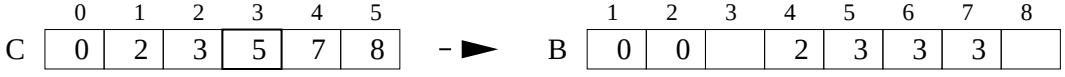

A[5]=2

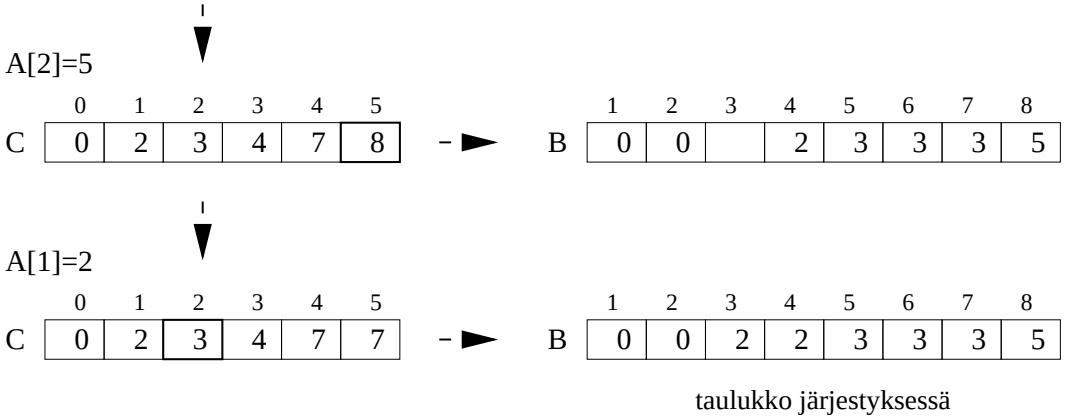



A[4]=0




A[3]=3






taulukko järjestyksessä

## 6.5 Yhteenveto järjestämisalgoritmeista

- aikavaativuus

| | pahin tapaus | keskim. tapaus | paras tapaus |
|---|---|---|---|
| kupla | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |
| lisäys | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ |
| lomitus | $\mathcal{O}(n\ log\ n)$ | $\mathcal{O}(n\ log\ n)$ | $\mathcal{O}(n\ log\ n)$ |
| keko | $\mathcal{O}(n\ log\ n)$ | $\mathcal{O}(n\ log\ n)$ | $\mathcal{O}(n\ log\ n)$ |
| pika | $\mathcal{O}(n^2)$ | $\mathcal{O}(n\ log\ n)$ | $\mathcal{O}(n\ log\ n)$ |
| laskemis | $\mathcal{O}(n+k)$ | $\mathcal{O}(n+k)$ | $\mathcal{O}(n+k)$ |

- tilavaativuus
  - lomitusjärjestämisellä $\mathcal{O}(n)$
  - laskemisjärjestämisellä $\mathcal{O}(n+k)$
  - muilla $\mathcal{O}(1)$
- entä käytännössä? mitä kannattaa käyttää?