Sortering 2:

Sammanfattning,
Intressanta algoritmer,
Undre gräns för sortering,
Sortering i linjär tid?

TDDE22, 725G97: DALG

Magnus Nielsen

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Bonusuppgifter tillgängliga

- Varje godkänd uppgift ger 0.5p mot högre betyg på tenta (endast första tentamenstillfälle)
- Bonuspoäng tillgodoräknas när gränsen för G/3 uppnås
- Uppgifter räknas som godkända när de är acceptera av *Kattis*
- Ska vara godkända senast midnatt dagen innan tenta

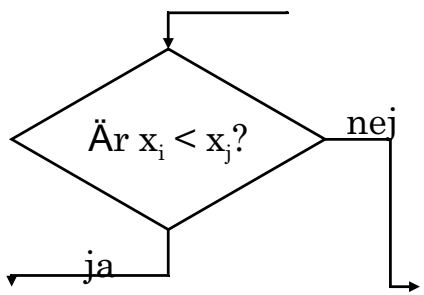
Sammanfattning så långt

Algoritm	Tid	Noteringar
selection-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
insertion-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
quick-sort	$O(n \log n)$ förväntad	• in-place, randomiserad • snabbast (bra för stora indata)
heap-sort	$O(n \log n)$	• in-place • snabb (bra för stora indata)
merge-sort	$O(n \log n)$	• sekvensiell dataaccess • snabb (bra för enorma indata)

LINKÖPING UNIVERSITY

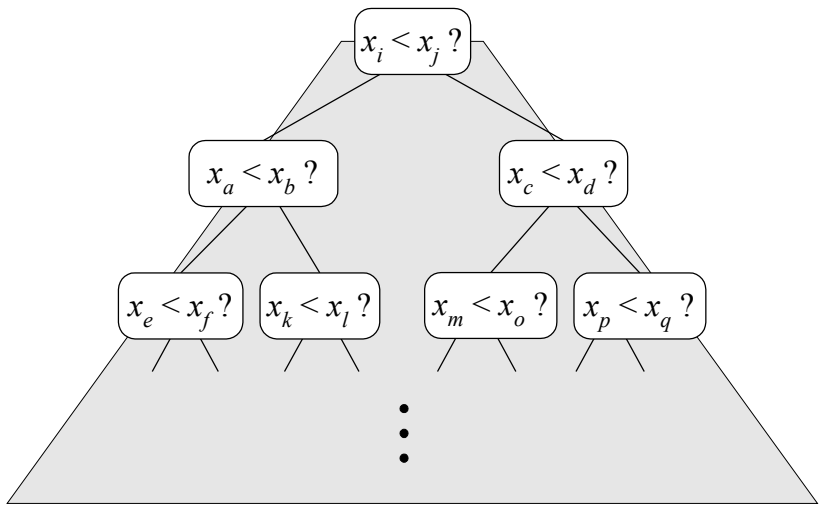
Jämförelsebaserad sortering

- Många sorteringsalgoritmer är *jämförelsebaserade*
  - De sorterar genom att göra jämförelser mellan par av objekt
  - Exempel: insertion-sort, selection-sort, heap-sort, merge-sort, quick-sort, ...
- Låt oss därför försöka härleda en undre gräns för exekveringstiden i värsta fall för varje algoritm som använder jämförelse för att sortera n element $x_1, x_2, \ldots, x_n$



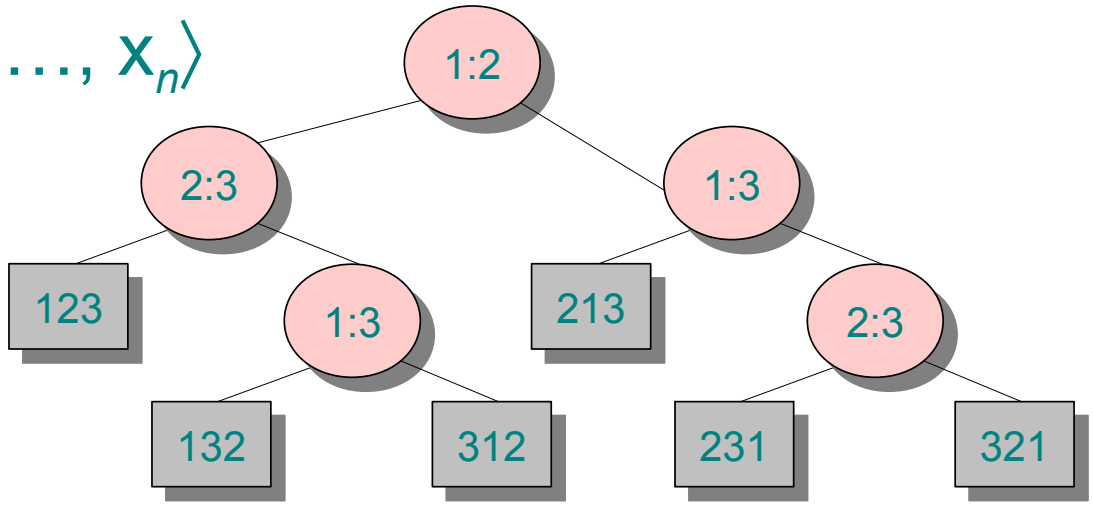
Räkna jämförelser

- Låt oss enbart räkna jämförelser
- Varje möjlig körning av en algoritm motsvaras av en rot-till-lövstig i ett beslutsträd



Exempel: Beslutsträd

Sort $\langle x_1, x_2, \ldots, x_n \rangle$

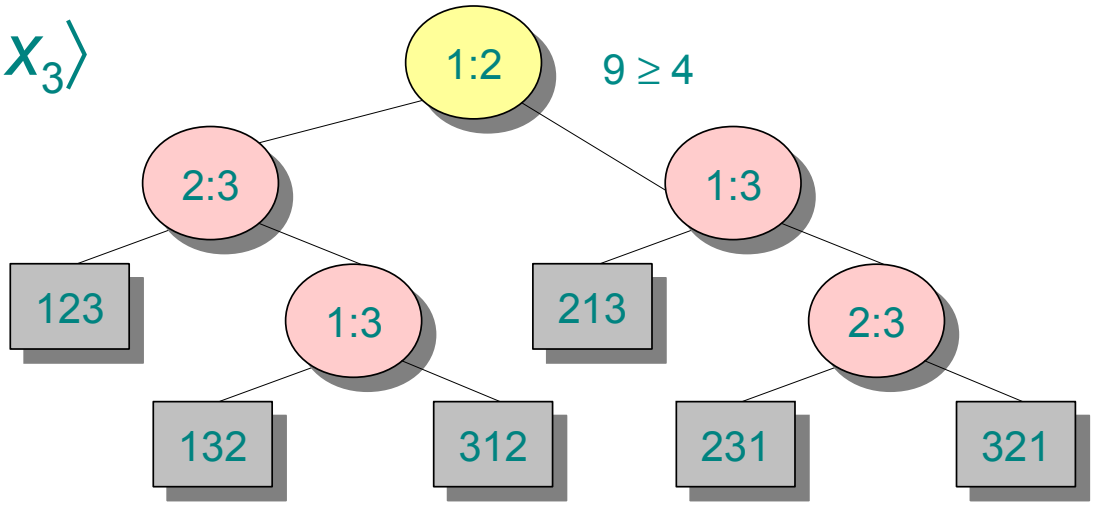


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

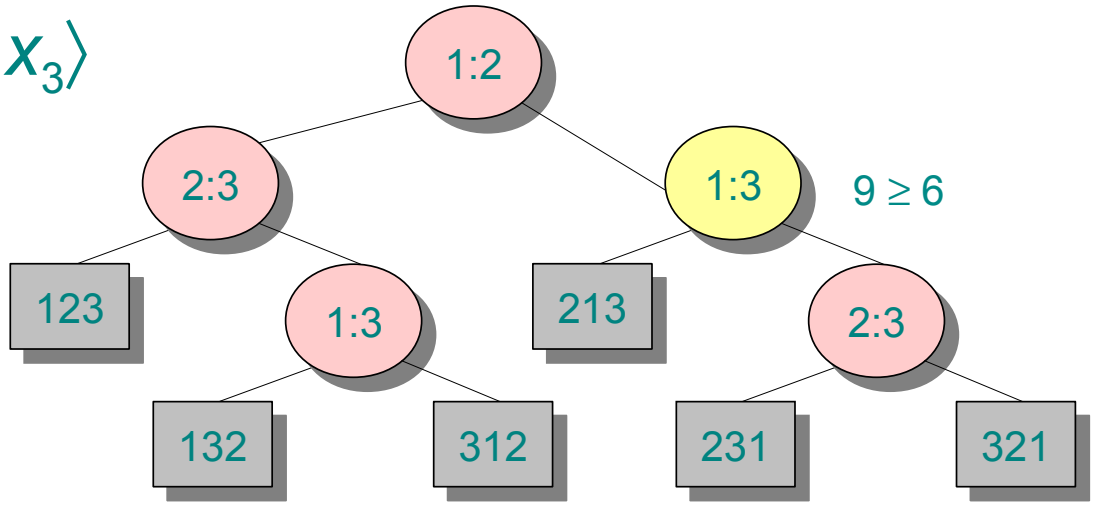


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

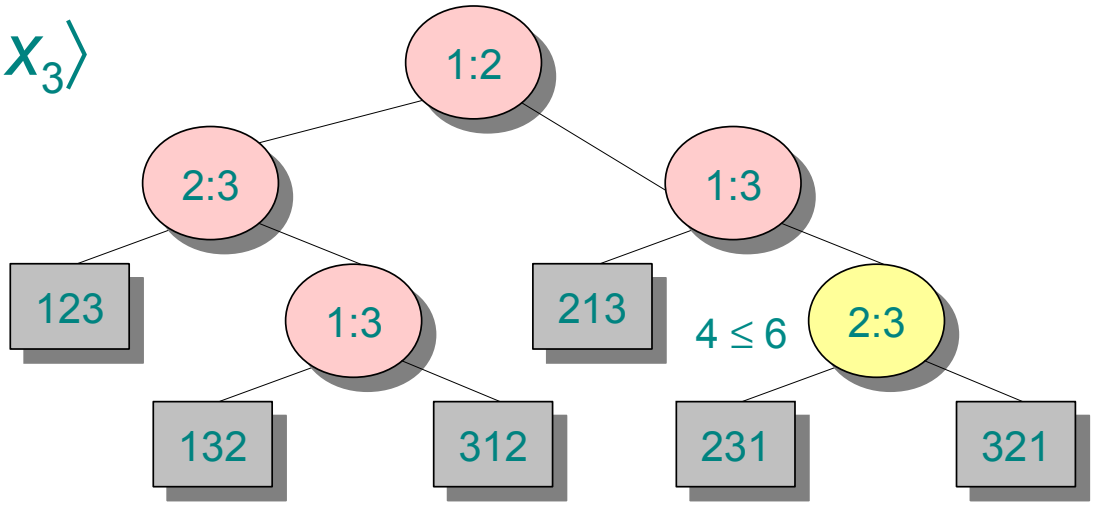


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

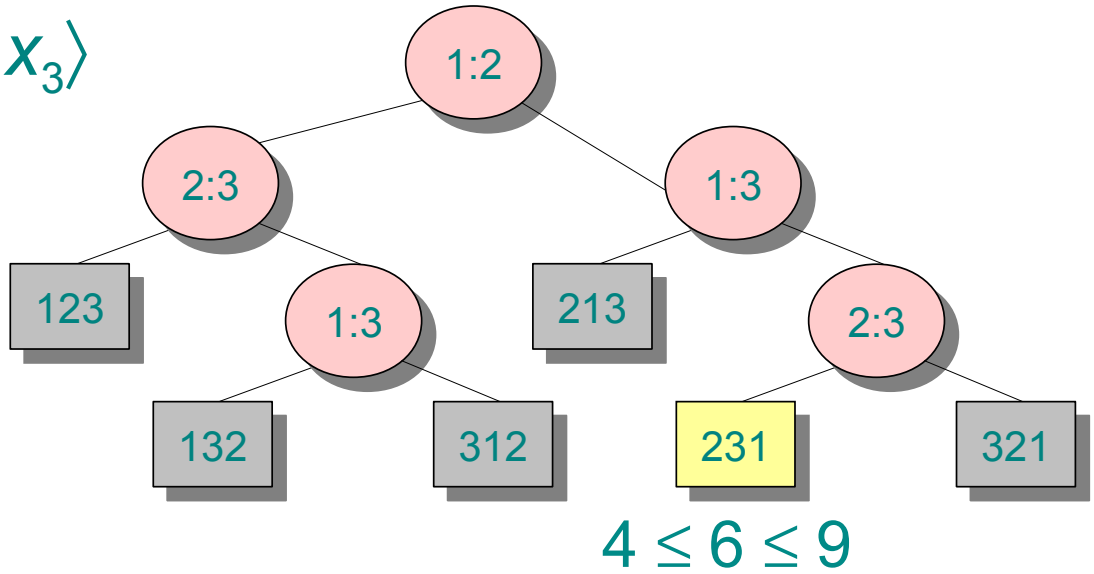


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



Varje löv innehåller en permutation $\langle \pi(1), \pi(2), \pi(i), \ldots, \pi(n) \rangle$ för att indikera att ordningen $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$ har etablerats

Beslutsträdsmodellen

Ett beslutsträd kan modellera exekveringen av vilken jämförelsebaserad sorteringsalgoritm som helst:

- Ett träd för varje storlek på indata
- Betrakta algoritmen som att exekveringen delas närhelst två element jämförs
- Trädet innehåller alla jämförelser längs alla tänkbara följder av instruktioner
- Körtiden för algoritmen $=$ längden av stigen som traverseras
- Körtiden i värsta fall $=$ höjden av trädet

Beslutsträdets höjd

- Höjden av beslutsträdet är en undre gräns för exekveringstiden i värsta fallet
- Varje tänkbar permutation av indata måste leda till ett separat utdatalöv
  - Om det inte vore så skulle något indata $\ldots 4 \ldots 5 \ldots$ ha samma utdataordning som $\ldots 5 \ldots 4 \ldots$ vilket skulle vara fel
- Eftersom det finns $n! = 1 \cdot 2 \cdot \ldots \cdot n$ löv är höjden av trädet minst $\log(n!)$

Den undre gränsen

- Varje jämförelsebaserad sorteringsalgoritm använder minst $\log(n!)$ tid i värsta fallet
- Därför använder en sådan algoritm minst tid

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2)$$

- Alltså är exekveringstiden för alla jämförelsebaserade sorteringsalgoritmer $\Omega(n \log n)$ i värsta fallet

LINKÖPING UNIVERSITY

Några fall då man kan sortera snabbare än $n \log n$

Några fall då man kan sortera snabbare än $n \log n$

- Bara ett konstant antal *olika* element ska sorteras
  - $\Theta(n)$ med räknesortering

Några fall då man kan sortera snabbare än $n \log n$

- Bara ett konstant antal *olika* element ska sorteras
  - $\Theta(n)$ med räknesortering
- Elementen som ska sorteras är tal som är jämnt fördelade i ett visst intervall
  - $\Theta(n)$ med bucket-sort

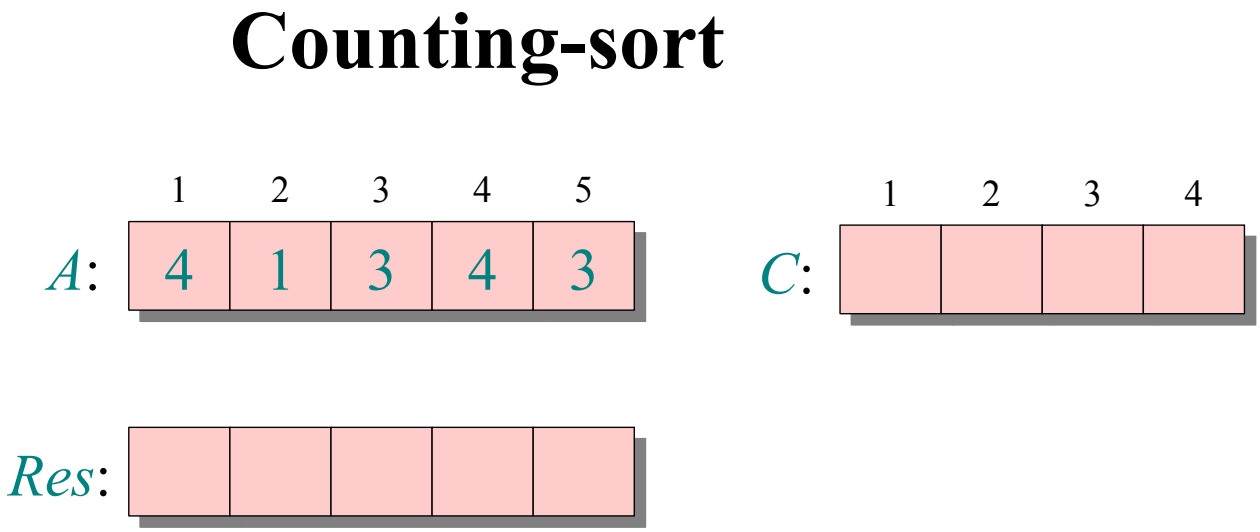
Några fall då man kan sortera snabbare än $n \log n$

- Bara ett konstant antal *olika* element ska sorteras
  - $\Theta(n)$ med räknesortering
- Elementen som ska sorteras är tal som är jämnt fördelade i ett visst intervall
  - $\Theta(n)$ med bucket-sort
- Elementen som ska sorteras är strängar som består av d "siffror" ($S[i] = s_{i,1}s_{i,2}\ldots s_{i,d}$)
  - $\Theta(nd)$ med radix-sort
  - Om d är konstant får vi linjär tidskomplexitet
  - Om vi räknar antalet siffror i indata får vi linjär tidskomplexitet $\Theta(N)$, där $N = nd$

Räknesortering

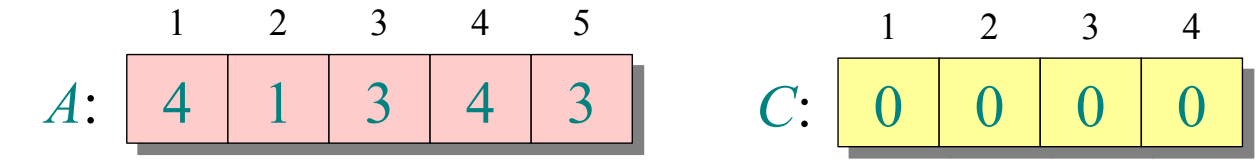
Require: $In[1, \ldots, n]$, där $In[j] \in \{1, 2, \ldots, k\}$
  function CountingSort(In)
    Hjälparray för räkning: $Count[1, \ldots, k]$
    Hjälparray för lagring av resultatet: $Res[1, \ldots, n]$
    for $i \leftarrow 1$ **to** k **do**
      $Count[i] \leftarrow 0$ ▷ Sätt räknaren till 0 för varje tal 1..k
    for $j \leftarrow 1$ **to** n **do**
      $Count[In[j]] \leftarrow Count[In[j]] + 1$ ▷ Beräkna antal förekomster av varje tal
    for $i \leftarrow 2$ **to** k **do**
      $Count[i] \leftarrow Count[i] + Count[i-1]$ ▷ Beräkna startindex för varje tal
    for $j \leftarrow n$ **downto** 1 **do**
      $Res[Count[In[j]]] \leftarrow In[j]$ ▷ Lagra resultatet på rätt index
      $Count[In[j]] \leftarrow Count[In[j]] - 1$ ▷ Räkna ner för att nästa förekomst av talet ska hamna på rätt index

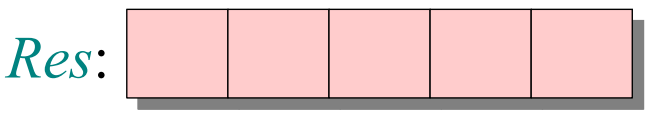
Exempel



Exempel

Loop 1

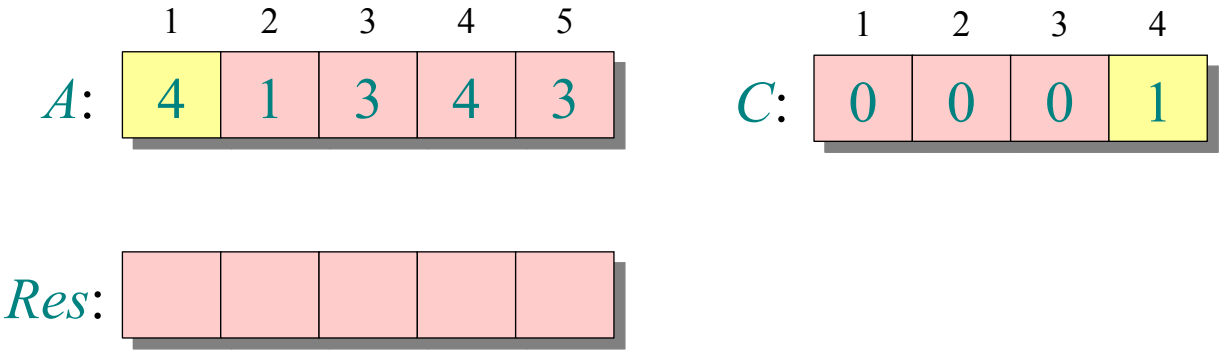




for $i \leftarrow 1$ **to** k **do**
$C[i] \leftarrow 0$

Exempel

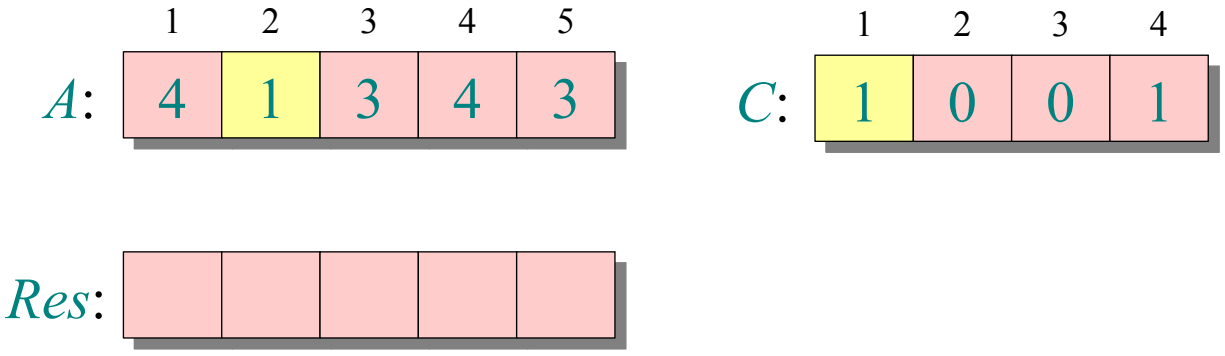
Loop 2



for $j \leftarrow 1$ **to** n **do**
$C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

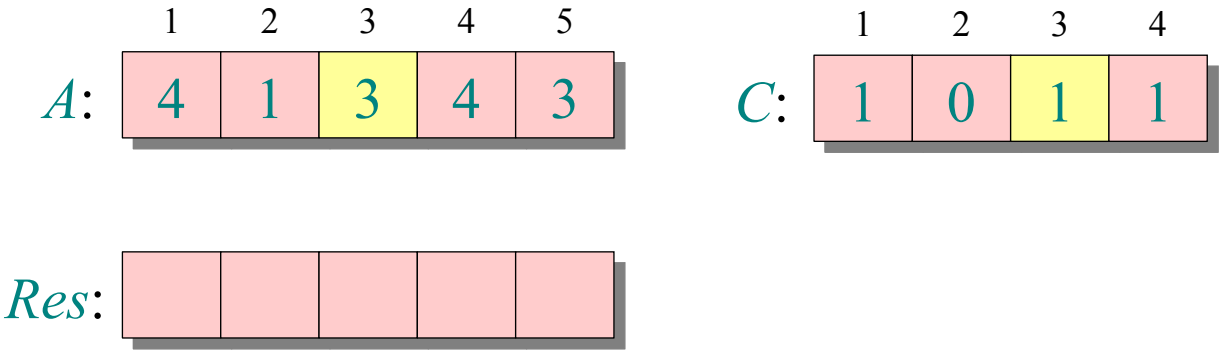
Loop 2



for $j \leftarrow 1$ **to** n **do**
$C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

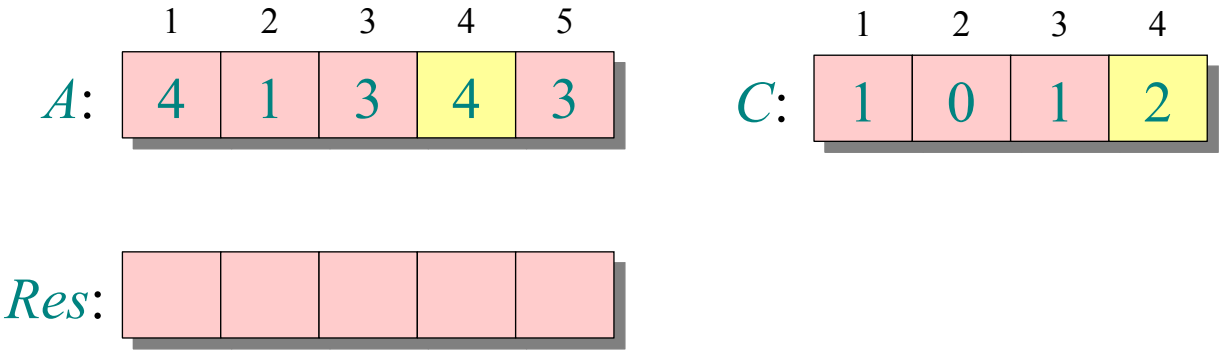
Loop 2



for $j \leftarrow 1$ **to** n **do**
$C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

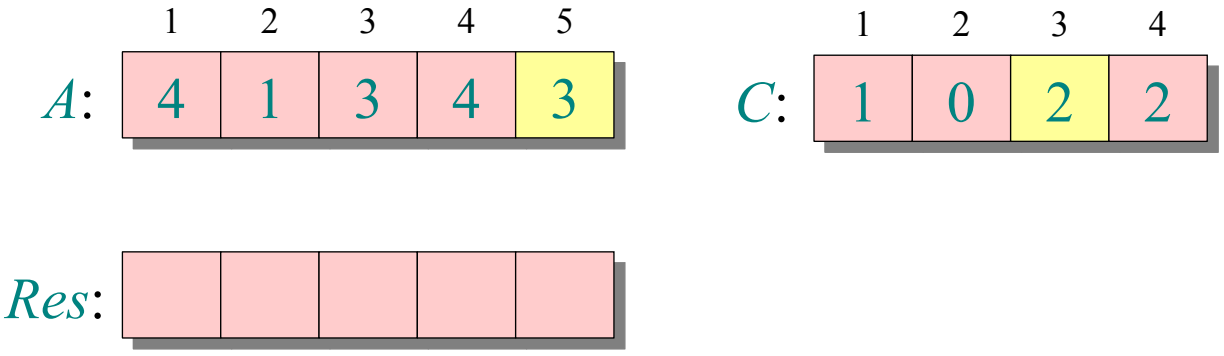
Loop 2



for $j \leftarrow 1$ **to** n **do**
$C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

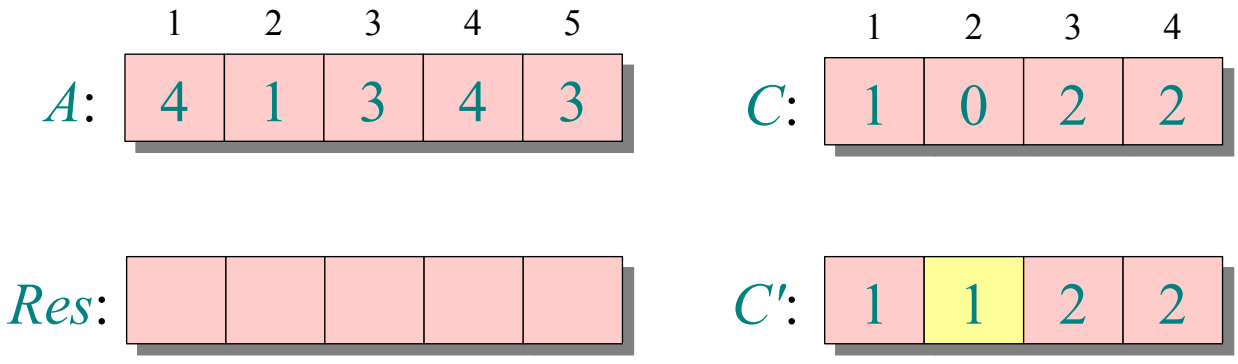
Loop 2



for $j \leftarrow 1$ **to** n **do**
$C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

Loop 3

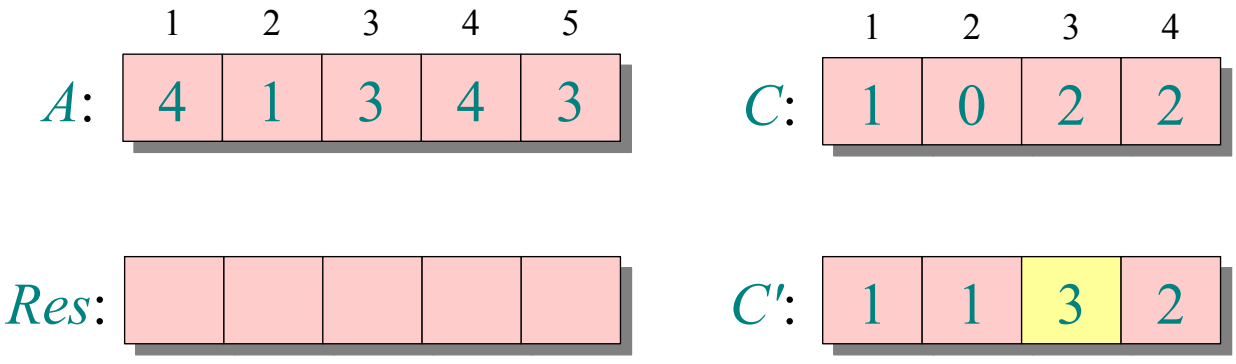


for $i \leftarrow 2$ **to** k **do**

$C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

Loop 3

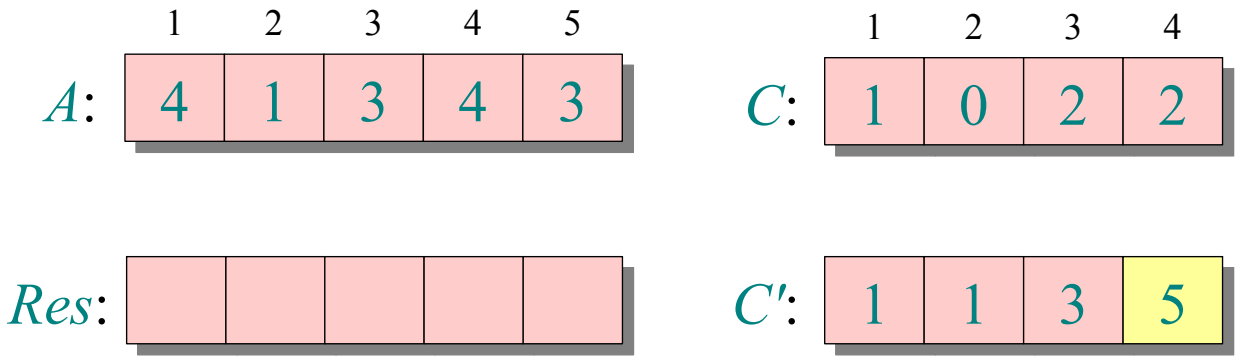


for $i \leftarrow 2$ **to** k **do**

$C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

Loop 3

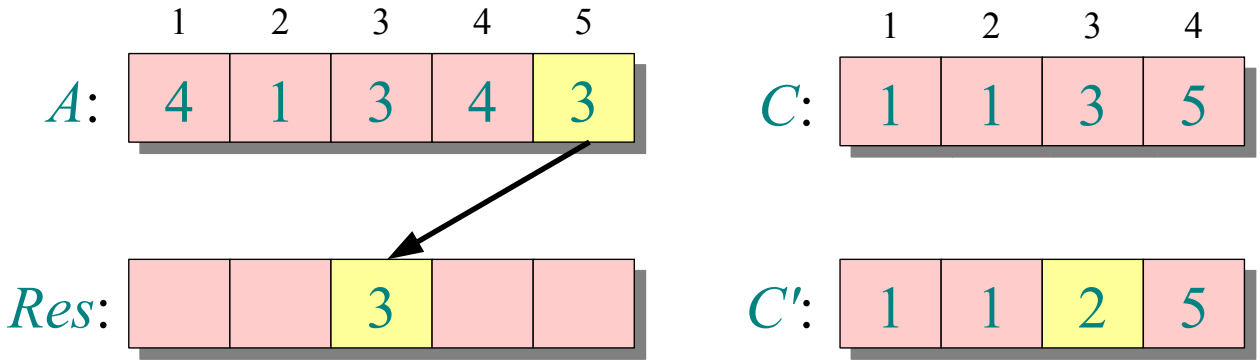


for $i \leftarrow 2$ **to** k **do**

$C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

Loop 4



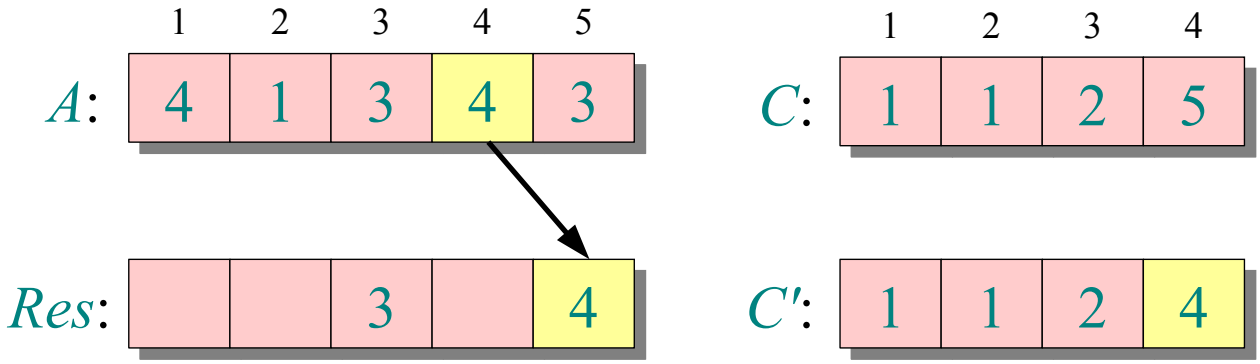
for $j \leftarrow n$ **downto** 1 **do**

$Res[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Exempel

Loop 4



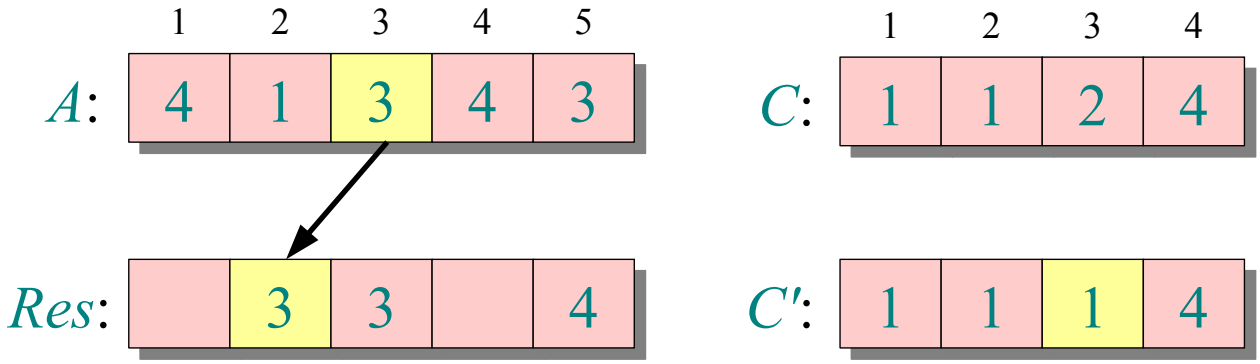
for $j \leftarrow n$ **downto** 1 **do**

$Res[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Exempel

Loop 4



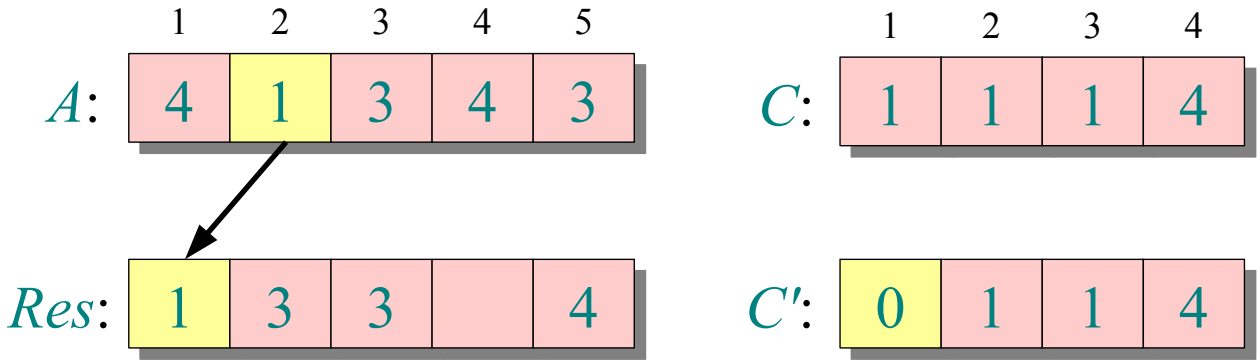
for $j \leftarrow n$ **downto** 1 **do**

$Res[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

Exempel

Loop 4



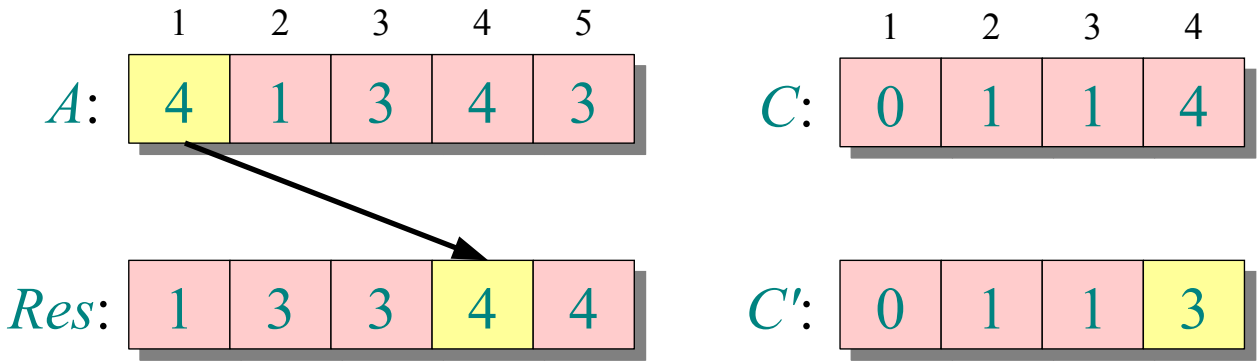
for $j \leftarrow n$ **downto** 1 **do**

$Res[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

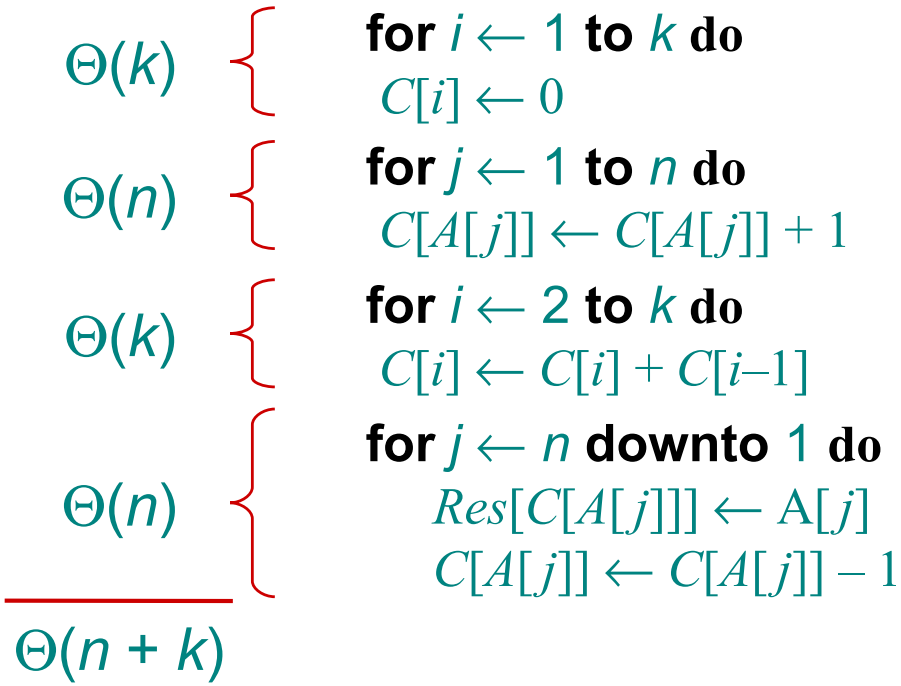
Exempel

Loop 4



for $j \leftarrow n$ **downto** 1 **do**
$Res[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Analys


Θ(k)
for i ← 1 to k do
C[i] ← 0
Θ(n)
for j ← 1 to n do
C[A[j]] ← C[A[j]] + 1
Θ(k)
for i ← 2 to k do
C[i] ← C[i] + C[i–1]
Θ(n)
for j ← n downto 1 do
Res[C[A[j]]] ← A[j]
C[A[j]] ← C[A[j]] – 1
Θ(n + k)

Exekveringstid

Om $k \in O(n)$ tar räknesortering $\Theta(n)$ tid

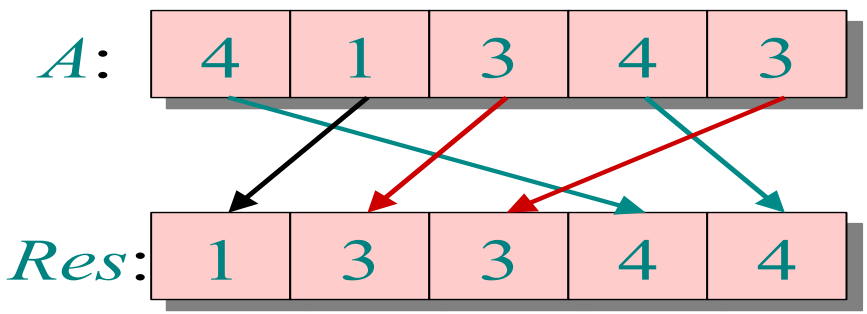
- Men sortering tar ju $\Omega(n \log n)$ tid!
- Vad är det som inte stämmer?

Svar

- Jämförelsebaserad sortering tar $\Omega(n \log n)$ tid
- Counting-sort är inte jämförelsebaserad
- Faktum är att inte en enda jämförelse mellan några element utförs!

Stabil sortering

Counting-sort är en stabil sorteringsmetod: den bevarar indataordningen mellan lika element

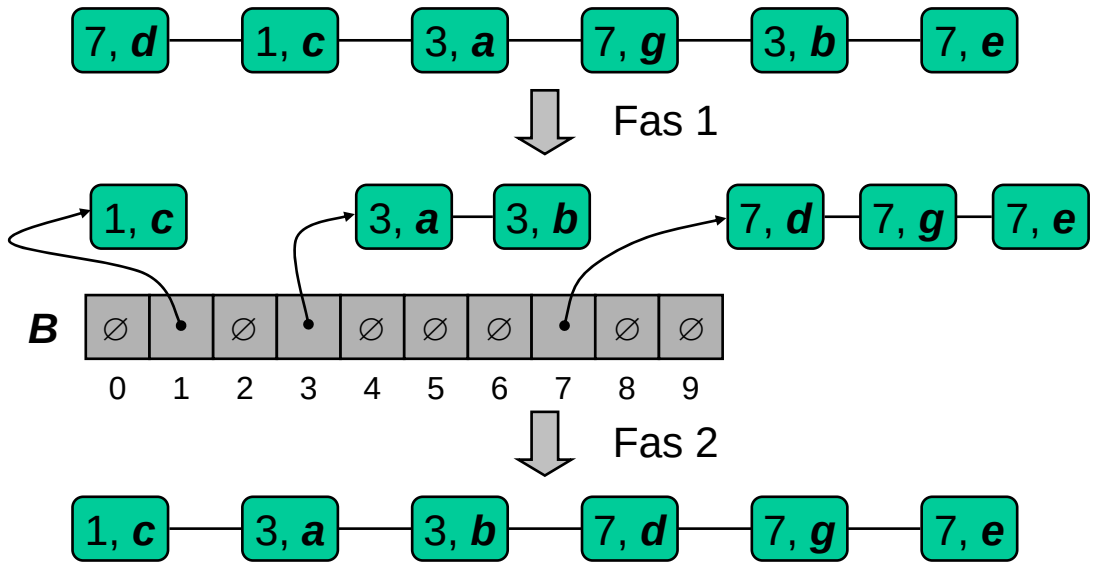


Bucket-sort

- Låt S vara en sekvens av n poster (nyckel, värde) med nycklar från $[0, N-1]$
- Bucket-sort använder nycklarna som index i en hjälparray B av sekvenser
  - Fas 1:
    Töm sekvensen S genom att flytta varje post (k, v) sist i sin bucket $B[k]$
  - Fas 2:
    För $i = 0, \ldots, N-1$ flytta posterna i bucket $B[i]$ till slutet av sekvensen S
- Analys:
  Fas 1 $\in O(n)$, Fas 2 $\in O(n+N)$
  Bucket-sort $\in O(n+N)$

procedure BUCKETSORT(S, N)
  $B \leftarrow$ array med N tomma sekvenser
  while $\neg S$.ISEMPTY() **do**
    $f \leftarrow S$.FIRST()
    $(k, o) \leftarrow S$.REMOVE(f)
    $B[k]$.INSERTLAST((k, o))
  for $i \leftarrow 0$ **to** $N-1$ **do**
    while $\neg B[i]$.ISEMPTY() **do**
      $f \leftarrow B[i]$.FIRST()
      $(k, o) \leftarrow B[i]$.REMOVE(f)
      S.INSERTLAST((k, o))

Exempel: Nycklar från $[0, 9]$



Egenskaper och utökningar

Nycklarnas typ

- Nycklarna används som index i en array och kan inte vara objekt av godtycklig typ (heltal är enklast att hantera)

Stabil sortering

- Den relativa ordningen av alla par av data med samma nycklar är bevarad efter exekveringen av algoritmen

Egenskaper och utökningar

Nycklarnas typ

- Nycklarna används som index i en array och kan inte vara objekt av godtycklig typ (heltal är enklast att hantera)

Stabil sortering

- Den relativa ordningen av alla par av data med samma nycklar är bevarad efter exekveringen av algoritmen

Utökningar

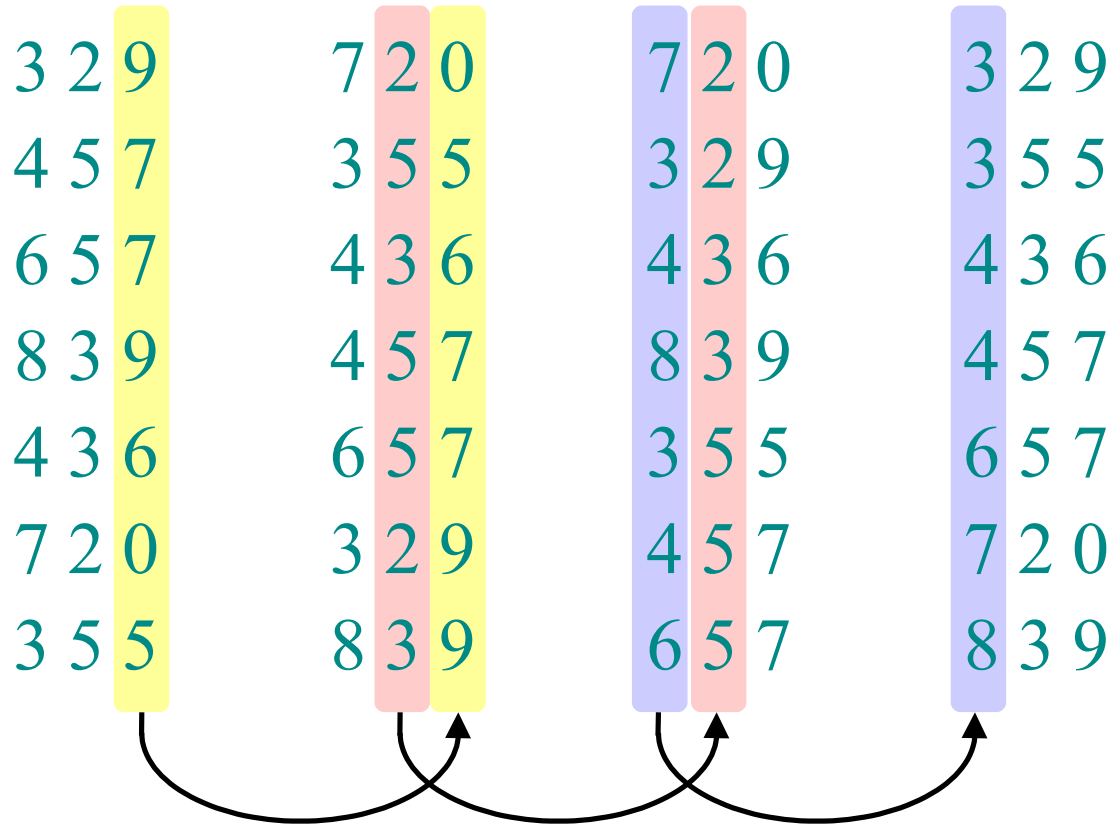
- Heltalsnycklar från $[a, b]$
  - Sätt in (k, v) i bucket $B[k - a]$
- Strängnycklar från en ändlig mängd strängar D
  - Sortera D och beräkna rangen $r(k)$ för varje sträng $k \in D$ i den sorterade sekvensen
  - Sätt in (k, v) i bucket $B[r(k)]$

Radix-sort

- Ursprung: Herman Holleriths kortsorteringsmaskin för folkräkningen 1890 i USA
- Siffra-för-siffrasortering
- Holleriths (dåliga*) ursprungsidé: sortera på mest signifikant siffra först (MSD)
- Bra idé: sortera på minst signifikant siffra först (LSD), med en extern stabil sorteringsrutin

*Han sorterade heltalsnycklar. När kan MSD-sortering vara bättre än LSD?

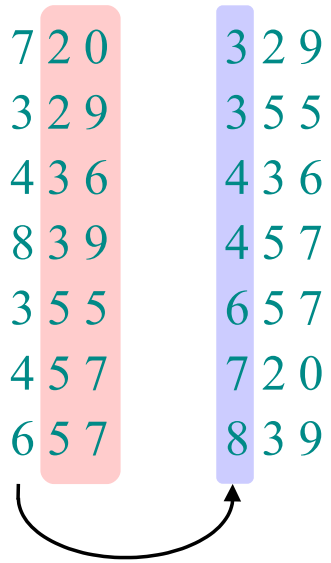
Exempel: Exekvering av radix-sort



Korrekthet för radix-sort

Använd induktion över sifferposition

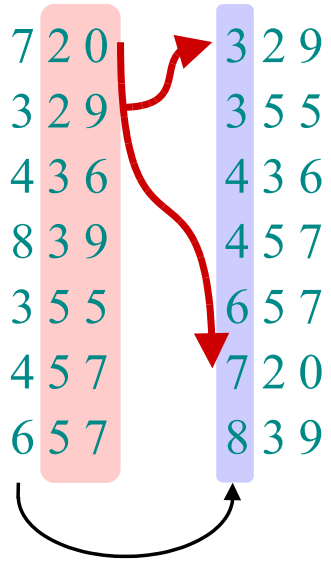
- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t



Korrekthet för radix-sort

Använd induktion över sifferposition

- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t
  - Två tal som skiljer sig i siffra t blir korrekt sorterade

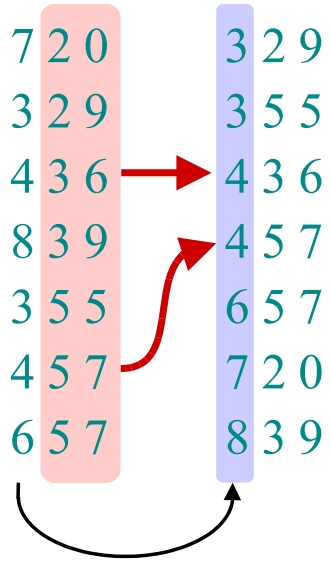


Korrekthet för radix-sort

Använd induktion över sifferposition

- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t
  - Två tal som skiljer sig i siffra t blir korrekt sorterade
  - Två tal som är lika i siffra t får samma ordning som i indata

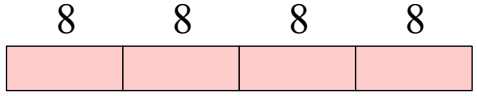
    $\Rightarrow$ rätt ordning



Analys av radix-sort

- Antag att counting-sort används som extern sorteringsrutin
- Sortera n maskinord om b bitar vardera
- Vi kan se det som att varje ord har b/r siffror i bas 2^r

Exempel:

32-bitars ord 

$r = 8 \Rightarrow b/r = 4$ pass av counting-sort på siffror i bas 2^8
eller $r = 16 \Rightarrow b/r = 2$ pass av counting-sort på siffror i bas 2^{16}

Hur många pass bör vi göra?

Analys av radix-sort

Kom ihåg: counting-sort tar tid $\Theta(n+k)$ för att sortera n tal från $[0, k-1]$.
Om varje b-bitars ord bryts upp i r-bitars bitar tar varje pass av counting-sort $\Theta(n+2^r)$ tid.
Eftersom det blir b/r pass får vi

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$$

Välj r för att minimera $T(n,b)$

- Att öka r ger färre pass, men när $r \gg \log n$ ökar tiden exponentiellt.

Att välja r

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

Minimera $T(n,b)$ genom att derivera och sätta till 0.
Eller, observera att vi inte vill ha $2^r \gg n$ och att det inte skadar asymptotiskt att välja r så stort som möjligt givet detta villkor.
Valet $r = \log n$ medför $T(n,b) = \Theta(bn/\log n)$.

- För tal i intervallet från 0 till $n^d - 1$ har vi $b = d \log n$
  $\Rightarrow$ radix-sort kör i tid $\Theta(dn)$.

Slutsatser

I praktiken är radix-sort snabb för stora indata, samt enkel att koda och underhålla.

Exempel: 32-bitars tal

- Som mest 3 pass när man sorterar ≥ 2000 tal.
- Merge-sort och quick-sort använder minst $\lceil \log 2000 \rceil = 11$ pass.

Nackdelar: Det går inte att sortera in-place med räknesortering. Det är också så att radix-sort inte har bra datalokalitet (vilket man kan se till att quick-sort har) vilket gör att en väl trimmad implementation av quick-sort kan vara snabbare på en modern processor med brant minneshierarki.

LINKÖPING UNIVERSITY

Bogosort (ej quantum version)

```
void bogosort(ArrayList<Integer> l) {
    while (!sorted(l)) {
        shuffle(l);
    }
}
bool sorted(ArrayList<Integer> l) {
    for (int i = 1; i < l.size(); i++) {
        if (l.get(i - 1) > l.get(i) {
            return false;
        }
        return true;
    }
}
```

Bogosort (quantum version)

```
void bogosort(ArrayList<Integer> l) {
    shuffle(l);
    if (!sorted(l)) {
        destroyUniverse();
        // Implementationsdetaljer för detta
        // lämnas till programmeraren
    }
}

bool sorted(ArrayList<Integer> l) {
    for (int i = 1; i < l.size(); i++) {
        if (l.get(i - 1) > l.get(i) {
            return false;
        }
        return true;
    }
}
```

Patience sort

Från start har vi inga högar.
Fas 1:

- Vi drar ett "kort"(element)
- "Kortetplaceras i den hög längst till vars översta kort har samma eller högre värde
- Ny hög om ingen existerande passar

Fas 2:
Slå ihop högarna genom att plocka bort det minsta synliga kortet och stoppa in det längst bak i listan med resultatet.

Sleep sort

Vi förutsätter numerisk data.
För varje element x starta en ny tråd som:

- Sover x tid
- Skriver ut x

I teorin ger indatan $[9, 5, 3, 1, 7]$ utsriften 1 3 5 7 9.

- Risk för fel (varför?)
- Vilken tidskomplexitet har vi?

Solar bitflip sort

1. Undersök om arrayen är sorterad
2. Om sorterad: returnera arrayen
3. Om ej sorterad: vänta 10 sekunder och hoppas att strålning från solen har flippat bits i minnet så att den blev sorterad.

Worstsort - slide av Filip

1. Generera alla permutationer av indatat, spara i en lista
2. Sortera listan lexiografiskt med hjälp av bubblesort
3. Ta det första elementet

Komplexitet: $\Omega((n!)^2)$

`https://sites.math.northwestern.edu/~mlerma/papers-and-preprints/inefficient_algorithms.pdf`

Worstsort – even worse - Slide av Filip

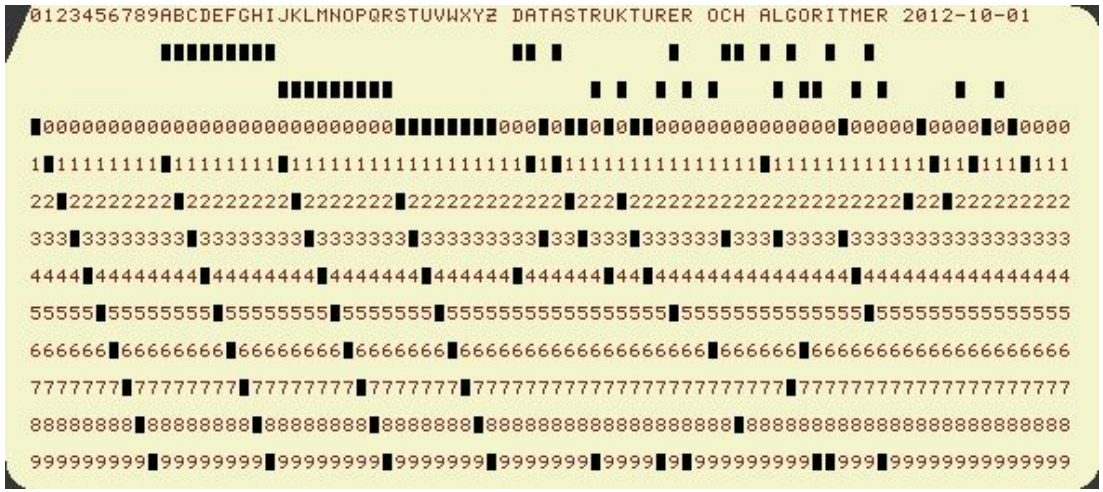
1. Generera alla permutationer av indatat, spara i en lista
2. Sortera listan lexiografiskt med hjälp av **worstsort**, om vi inte har nått ett förutbestämt djup ännu...
3. Ta det första elementet

Komplexitet: $\Omega(((n!)\ldots!)^2)$

`https://sites.math.northwestern.edu/~mlerma/papers-and-preprints/inefficient_algorithms.pdf`

LINKÖPING UNIVERSITY

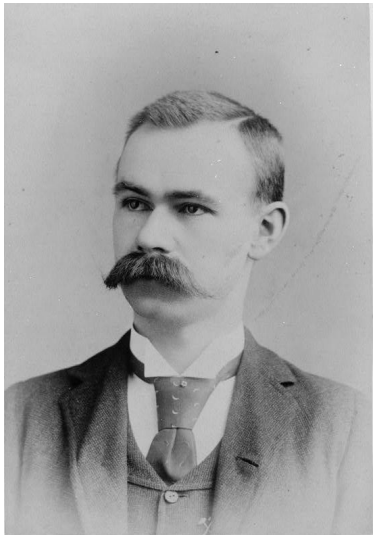
"Modernt" hålkort från IBM



Så det är därför skalfönster har 80 kolumner!

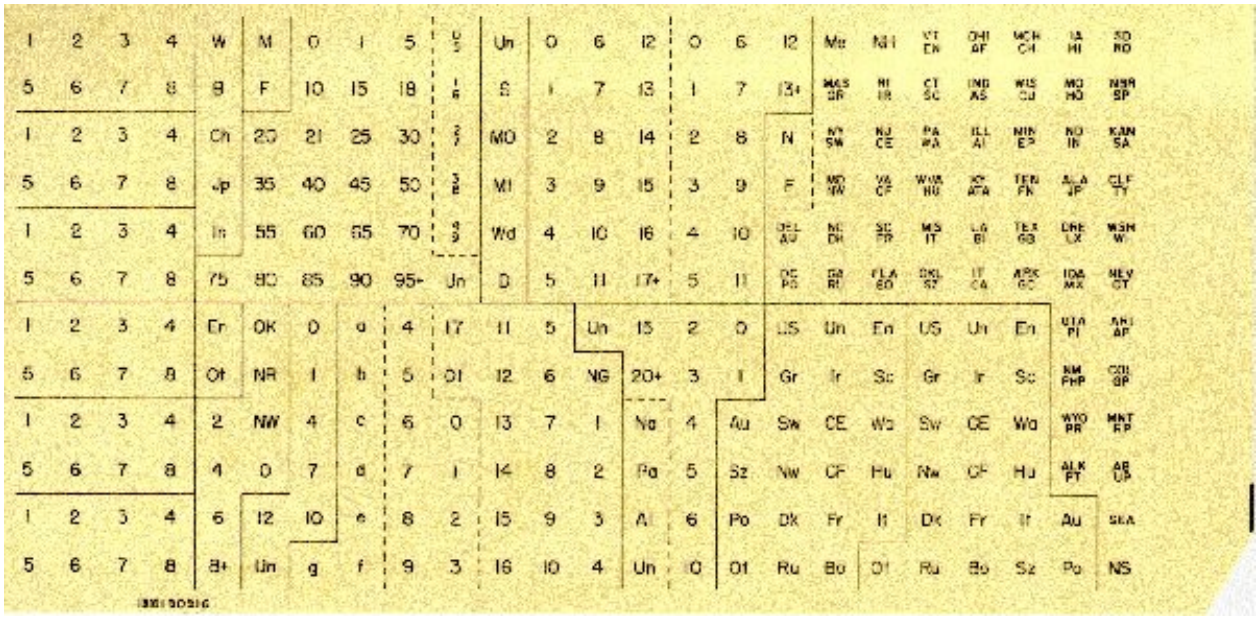
Herman Hollerith (1860-1929)

- 1880 års folkräkning i USA tog nästan 10 år att bearbeta.
- Under tiden arbetade en föreläsare vid MIT fram hålkortsteknologin.
- Hans maskiner, inklusive en ”kortsorterare”, gjorde att folkräkningen 1890 kunde slutföras på 6 veckor.
- Han grundade ”The Tabulating Machine Company” 1911, vilket gick samman med andra företag 1924 för att bilda ”International Business Machines”



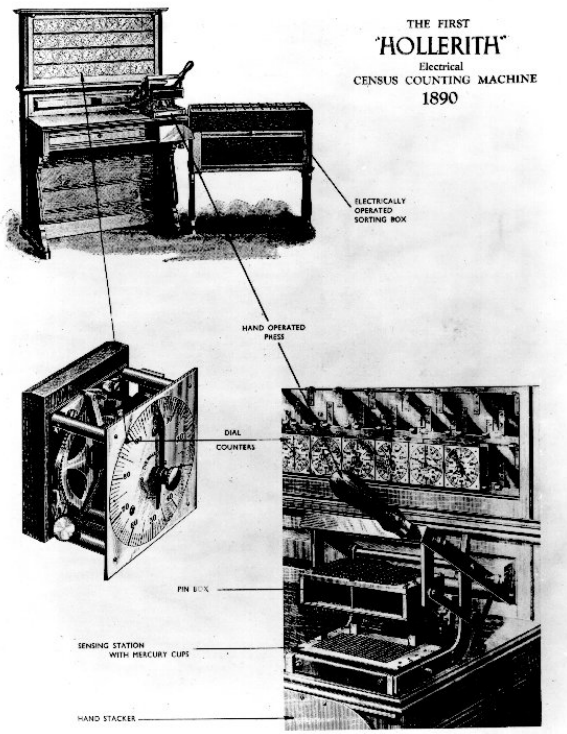
Hålkort

- Hålkort = datapost
- Hål = värde
- Algoritm = maskin + människa



Holleriths tabuleringssytem

- Pantografisk hålmaskin
- Läsare manövrerad med handkraft
- Visare för räkneresultat
- Sorteringlåda



Ursprunget till radix-sort

Holleriths patentansökan från 1899 talar om radix-sort med början från den mest signifikanta siffran:

> *The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards.*

Att köra radix-sort med minst signifikant siffra först verkar vara en uppfinning av maskinoperatörerna.

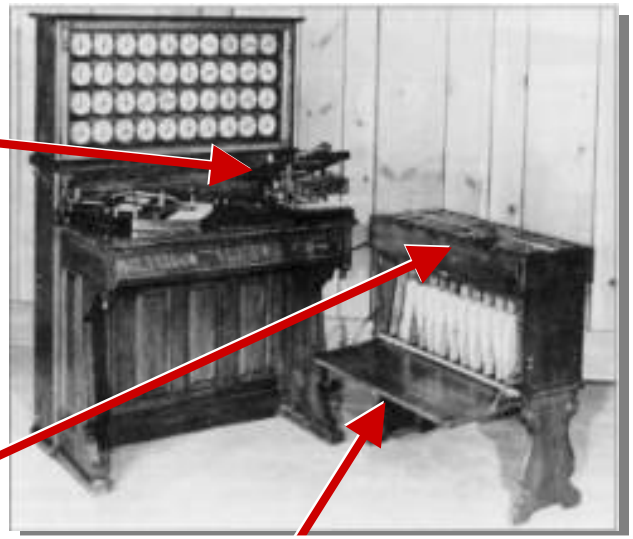
Sorteringsmaskinen

En operatör sätter in ett kort i pressen.

Stift på pressen når genom hålen i hålkortet och får kontakt med kvicksilverfyllda bägare under kortet.

När ett visst siffervärde slås in lyfts locket på motsvarande sorteringsfack.

Operatören lägger kortet i facket och stänger locket.



Holleriths Tabulator, Pantograph, Press och Sorter

När alla kort bearbetats öppnas frontpanelen och korten samlas in i ordning, vilket ger ett pass av en stabil sorteringsrutin.

Nästa gång:
Introduktion till grafer

www.liu.se

LINKÖPING UNIVERSITY