Sortering:
TDDE22, 725G97: DALG

Magnus Nielsen

LINKÖPING UNIVERSITY

1 Sortering

LINKÖPING UNIVERSITY

Sorteringsproblemet

Indata:

- En lista L innehållande data med *nycklar* från en linjärt ordnad mängd K

Utdata:

- En lista L' innehållande samma data sorterat i stigande ordning m.a.p. nycklarna

Exempel

$[8, 2, 9, 4, 6, 10, 1, 4] \rightarrow [1, 2, 4, 4, 6, 8, 9, 10]$

Varför sortera?

Varför vill vi sortera data?

- Underlätta sökning i stora datamängder
- Presentera data i ett lätthanterligt format
- ...

Varför sortera?

Varför vill vi sortera data?

- Underlätta sökning i stora datamängder
- Presentera data i ett lätthanterligt format
- ...
- Underlätta problemlösning.

Exempelproblem

Du och en grupp vänner har startat en onlinebutik. I er webbserver har ni en (osorterad) länkad lista med alla era 10 miljoner artiklar. På ett styrelsemöte kommer ni fram till att ni vill se över era 1000 dyraste artiklar. Ni har två val:

1. Sök igenom listan 1000 gånger (tidskomplexitet O(n)).
2. Konvertera den länkade listan till en array (tidskomplexitet O(n)), sortera arrayen (tidskomplexitet O(n log n)) och sedan hämta ut de 1000 dyraste artiklarna?

Vilket val gör ni, och varför? Vi kan avfärda tiden det tar att plocka fram ett data ur arrayen.

Svar

Vi räknar lite:

1. Vi har 10^7 element, och ska göra 1000 fullständiga traverseringar av listan. Det ger oss ungefär $1000 \times 10^7 = 10^{10} = 10 miljarder$ operationer
2. Konverteringen kostar oss ca 10^7 operationer. I datavetenskapen refererar logaritmuttrycket till bas 2 logaritmer. Vi har då: $10^7 \times \log_2(10^7) \approx 2.3 \times 10^8$ operationer för att sortera arrayen. Uppskattat antal operationer blir ungefär: $2.3 \times 10^8 + 10^7 = 240 miljoner$. Anledningen till att vi kan bortse från kostnaden för att plocka fram de 1000 största (dyraste) elementen i arrayen är att de kommer vara från slutet av arrayen, och således behöver vi göra 1000 triviala operationer för att plocka fram dem. 1000 triviala operationer är försumbara i sammanhanget.

Vad har vi lärt oss?

Svar

Vi räknar lite:

1. Vi har 10^7 element, och ska göra 1000 fullständiga traverseringar av listan. Det ger oss ungefär $1000 \times 10^7 = 10^{10} = 10 miljarder$ operationer
2. Konverteringen kostar oss ca 10^7 operationer. I datavetenskapen refererar logaritmuttrycket till bas 2 logaritmer. Vi har då: $10^7 \times \log_2(10^7) \approx 2.3 \times 10^8$ operationer för att sortera arrayen. Uppskattat antal operationer blir ungefär: $2.3 \times 10^8 + 10^7 = 240 miljoner$. Anledningen till att vi kan bortse från kostnaden för att plocka fram de 1000 största (dyraste) elementen i arrayen är att de kommer vara från slutet av arrayen, och således behöver vi göra 1000 triviala operationer för att plocka fram dem. 1000 triviala operationer är försumbara i sammanhanget.

Vad har vi lärt oss? Att inte bara stirra oss blinda på tidskomplexitet. Vi måste även fundera över hur vi ämnar använda oss av det vi undersöker. Dessutom har vi lärt oss att sortering är superhäftigt.

Varför så många algoritmer?

- Olika typer av indata som ska sorteras
- Olika grader av sortering redan gjord
- Olika storlekar på indata
- Olika förutsättningar / syfte för programmet som utför sorteringen

Aspekter på sortering

- In-place vs använda extra minne
- Internt vs externt minne
- Stabil vs icke stabil
- Jämförelsebaserad vs digital

Strategier

Sortering genom insättning

Titta efter rätt plats att sätta in varje nytt element som ska läggas till i den sorterade sekvensen...
linjär insättning, Shell-sort, ...

Sortering genom urval

Sök i varje iteration i den osorterade sekvensen efter det minsta kvarvarande datat och lägg det till slutet av den sorterade sekvensen...
rakt urval, Heap-sort, ...

Sortering genom platsbyten

Sök fram och tillbaka i något mönster och byt plats på par i fel inbördes ordning så fort ett sådant upptäcks...
Quick-sort, Merge-sort, Bubble-sort ...

Bubble sort

- Börja från första index, ”bubbla” fram till sista index, eller vice versa.
- Upprepa lika många gånger som det finns index,

I varje varv bubblas största (eller sjunker minsta, beroende på implementation) kvarvarande elementet till sin rätta plats.

Kan effektiviseras lite: exempelvis genom att ”bubbla” ett steg kortare varje varv.

(Linjär) insättningssortering

- Algoritmen är in-place!
- Dela arrayen som ska sorteras $A[0, \ldots, n-1]$ i två delar
  - $A[0, \ldots, i-1]$ som är sorterad
  - $A[i, \ldots, n-1]$ ej ännu sorterad

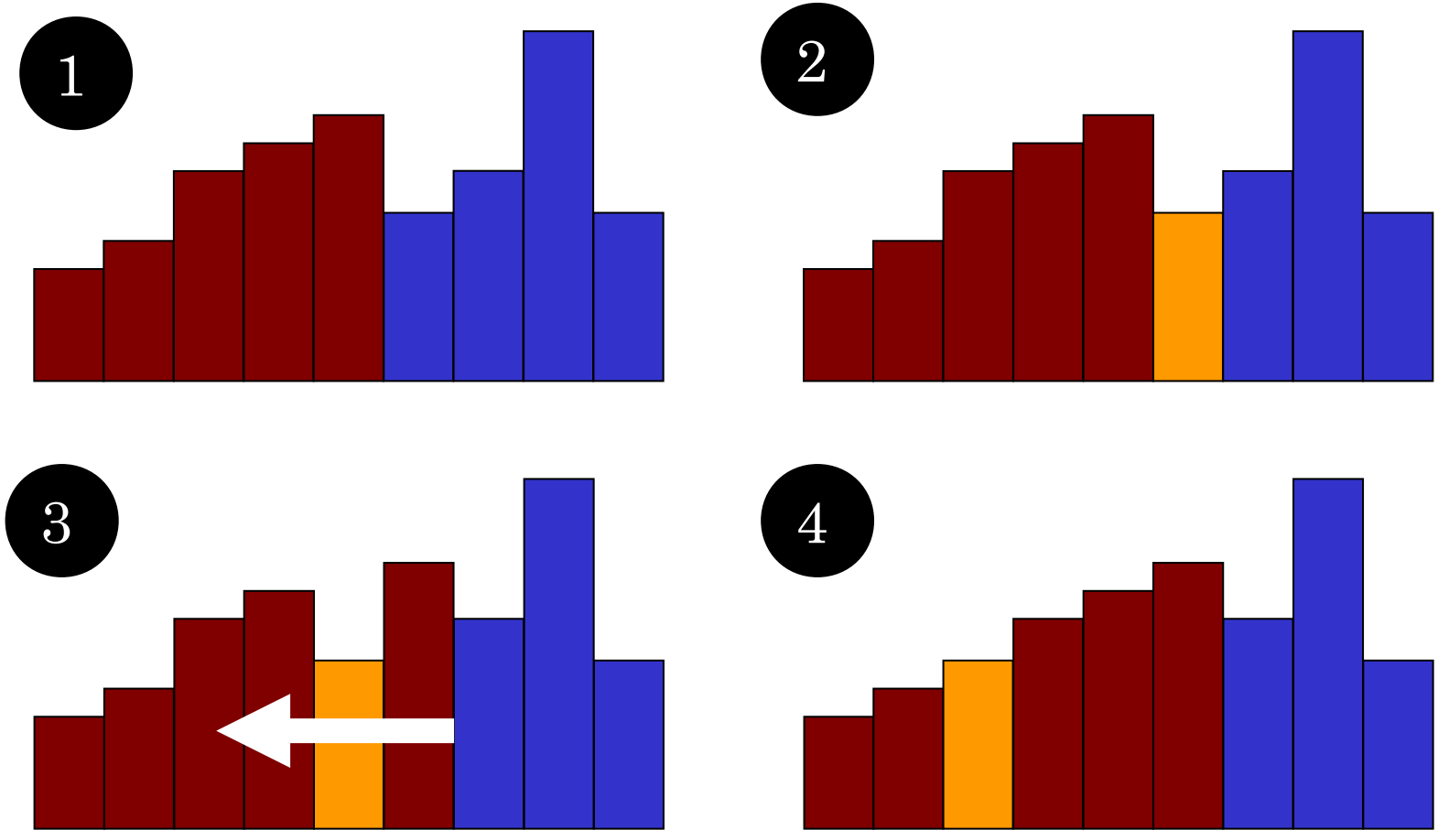
  Initialt är $i = 1$, i vilket fall $A[0, \ldots, 0]$ (trivialt) är sorterad

procedure InsertionSort($A[0, \ldots, n-1]$)
  for $i = 1$ **to** $n-1$ **do**
    sätt in $A[i]$ i rätt (=sorterad) position i $A[0, \ldots, i-1]$

Insertion sort

- Utgå från andra positionen (första positionen är redan “sorterad”)
- Är nyckeln på denna position mindre än nyckeln på föregående position?
  - Om så är fallet: byt plats på dem (swap) och fortsätt mot början av den sorterade delen
  - Om så ej är fallet: den sorterade delen är sorterad
- Utgå från nästa position
- Upprepa för resten av den osorterade mängden

Exempel: Visualisering av Insertion-sort



Värstafallsanalys av Insertion-sort

```
1: procedure InsertionSort(A[0, ..., n − 1])
2:     for i = 1 to n − 1 do
3:         inner ← i; x ← A[i]
4:         while inner ≥ 1 and A[inner − 1] > x do
5:             A[inner] ← A[inner − 1]; inner ← inner − 1
6:         A[inner] ← x
```

Värstafallsanalys av Insertion-sort

1: **procedure** INSERTIONSORT($A[0, \ldots, n-1]$)
2: **for** $i = 1$ **to** $n-1$ **do**
3: $inner \leftarrow i$; $x \leftarrow A[i]$
4: **while** $inner \geq 1$ **and** $A[inner-1] > x$ **do**
5: $A[inner] \leftarrow A[inner-1]$; $inner \leftarrow inner - 1$
6: $A[inner] \leftarrow x$

- rad_2: $n-1$ pass
- rad_3: $n-1$ pass
- rad_4: Låt I vara antalet iterationer i värsta fallet av innerloopen:

$$I = 1 + 2 + \ldots + (n-1) = n(n-1)/2 = (n^2 - n)/2$$

- rad_5: I pass
- rad_6: $n-1$ pass
- Totalt:
  $rad_2 + rad_3 + rad_4 + rad_5 + rad_6 = 3(n-1) + (n^2 - n) = n^2 + 2n - 3$
  Alltså $O(n^2)$ i värsta fallet...
  men bra om sekvensen nästan sorterad

(Rak) urvalssortering

- Algoritmen är in-place!
- Dela arrayen som ska sorteras $A[0, \ldots, n-1]$ i två delar
  - $A[0, \ldots, i-1]$ som är sorterad (alla element mindre än eller lika med $A[i, \ldots, n-1]$)
  - $A[i, \ldots, n-1]$ ej ännu sorterad

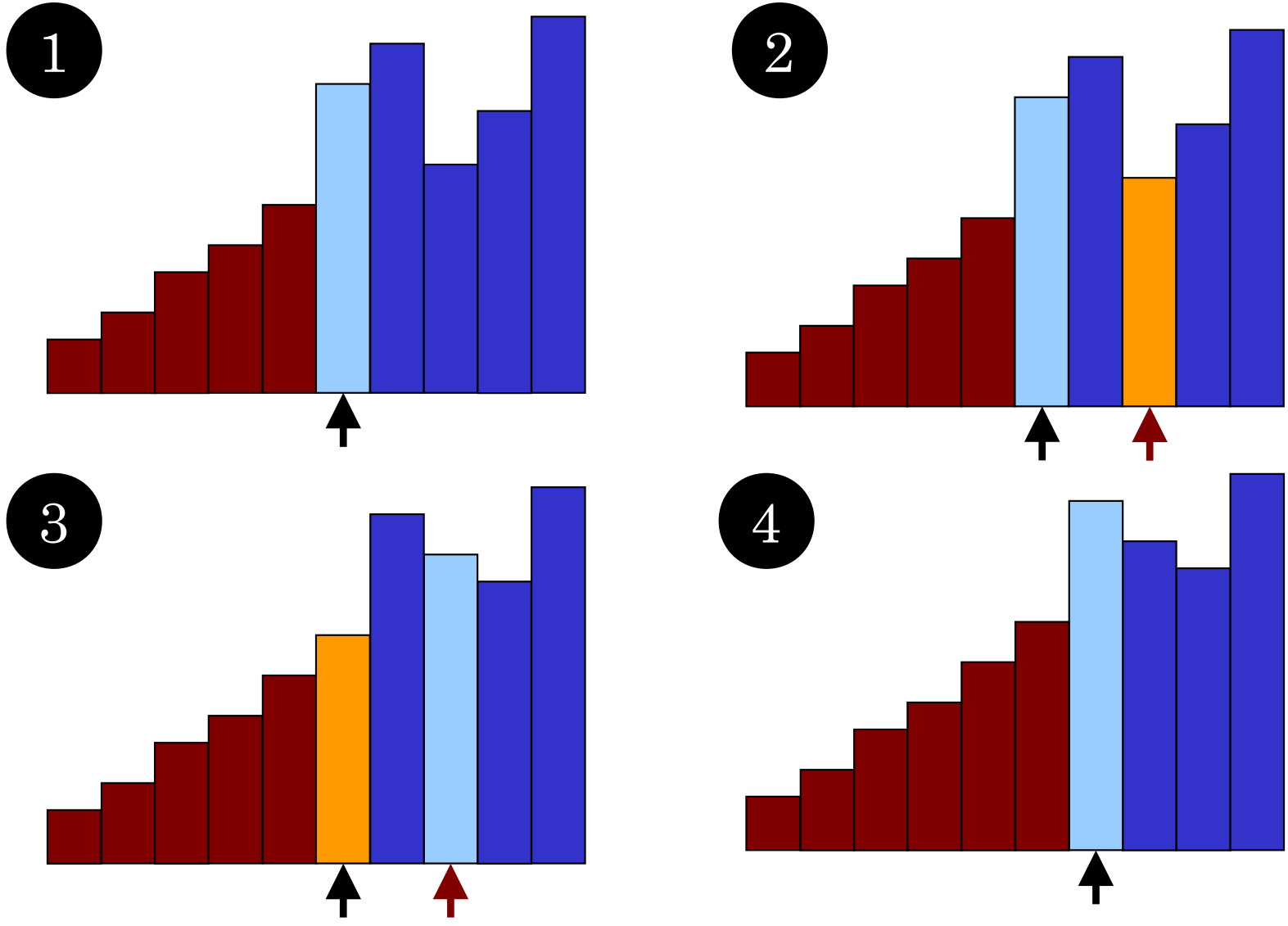
  Initialt är $i = 0$, d.v.s. den sorterade delen är tom (och trivialt sorterad)

procedure SelectionSort($A[0, \ldots, n-1]$)
  for $i = 0$ **to** $n-2$ **do**
    hitta ett minimalt element $A[j]$ i $A[i, \ldots, n-1]$
    byt plats på $A[i]$ och $A[j]$

Selection sort

- Utgå från första positionen
- Hitta minsta elementet i mängden, och swappa till den position vi utgår ifrån
- Utgå från nästa plats
- Upprepa för resten av mängden, lika många gånger som det finns platser

Exempel: Visualisering av Selection-sort



Värstafallsanalys av Selection-sort

```
1: procedure SelectionSort(A[0, ..., n − 1])
2:     for i = 0 to n − 2 do
3:         s ← i
4:         for j ≥ i + 1 to n − 1 do
5:             if A[j] < A[s] then  s ← j
6:         swap(A[i], A[s])
```

Värstafallsanalys av Selection-sort

1: **procedure** SELECTIONSORT($A[0, \ldots, n-1]$)
2: **for** $i = 0$ **to** $n-2$ **do**
3: $s \leftarrow i$
4: **for** $j \geq i+1$ **to** $n-1$ **do**
5: **if** $A[j] < A[s]$ **then** $s \leftarrow j$
6: SWAP($A[i], A[s]$)

- rad_2: $n-1$ pass
- rad_3: $n-1$ pass
- rad_4: Låt I vara antalet iterationer i värsta fallet av innerloopen:

  $$I = (n-2) + (n-3) + \ldots + 1 = (n-1)(n-2)/2 = (n^2 - 3n + 2)/2$$

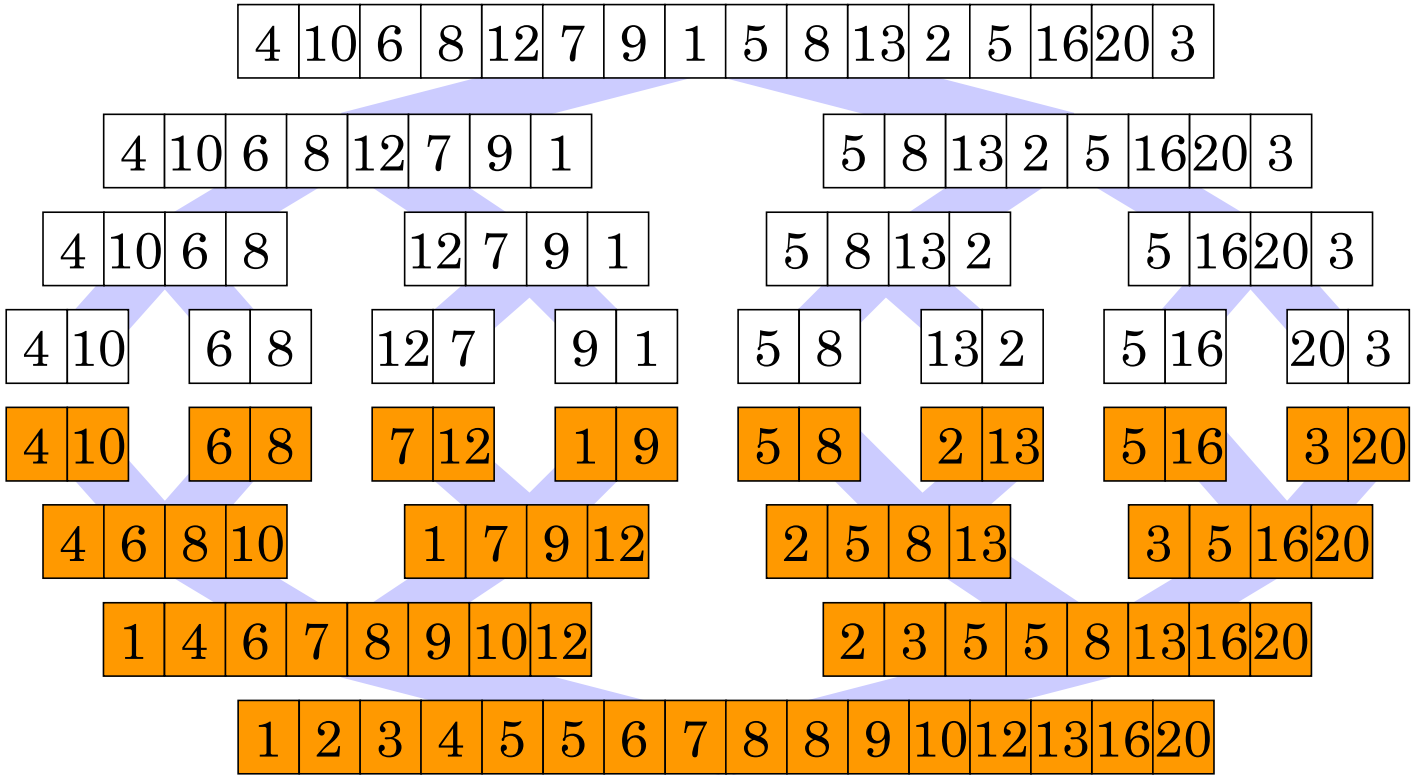
- rad_5: I pass
- rad_6: $n-1$ pass
- Totalt: $rad_2 + rad_3 + rad_4 + rad_5 + rad_6 =$
  $3(n-1) + (n^2 - 3n + 2) = n^2 - 1 \in O(n^2)$

Principen söndra-och-härska för algoritmkonstruktion

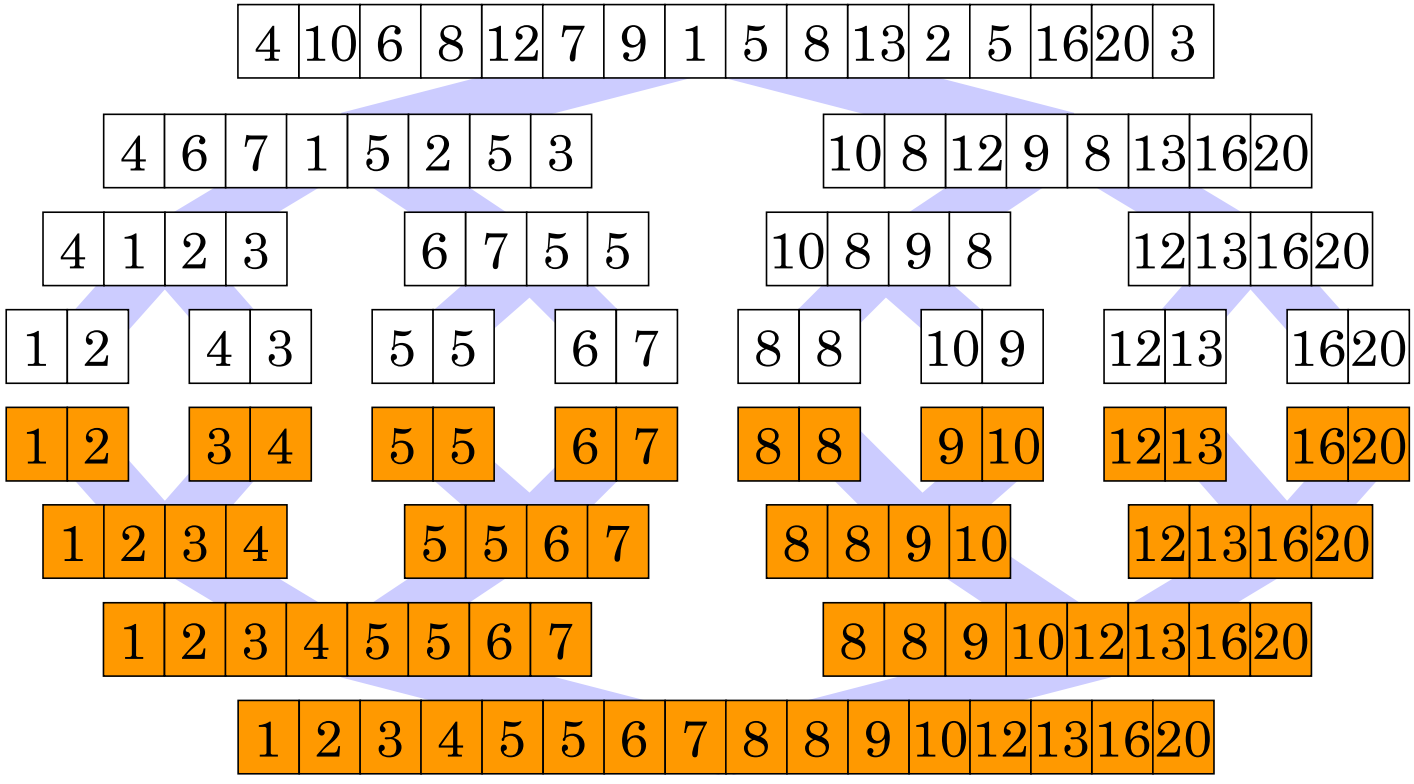
- söndra: dela upp problemet i mindre, oberoende, delproblem
- härska: lös delproblemen rekursivt (eller direkt om trivialt)
- kombinera lösningarna till delproblemen till en lösning till ursprungsproblemet
- Typexempel: Merge-Sort

Eng. *divide-and-conquer*

Exempel: Söndra-och-härska



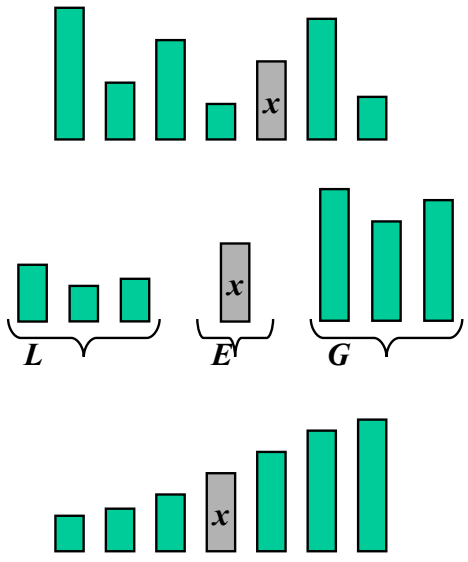
Exempel: Söndra-och-härska



Quick-sort

Quick-sort är en *randomiserad* sorteringsalgoritm baserad på paradigmet söndra-och-härska

- söndra: välj slumpvist (eller enligt en algoritm) ett element x (kallat pivot) och partitionera Seq till
  - $Less$ element mindre än x
  - Eq element lika med x
  - Grt element större än x
- härska: sortera $Less$ och Grt
- kombinera $Less$, Eq och Grt



Val av pivot-element

Många olika sätt att välja pivot-element:

- Slumpa ett pivot-element
- Elementet längst till höger eller vänster
- Medianen av tre (ex: vänster, mitten och höger)
- etc

Partitionering

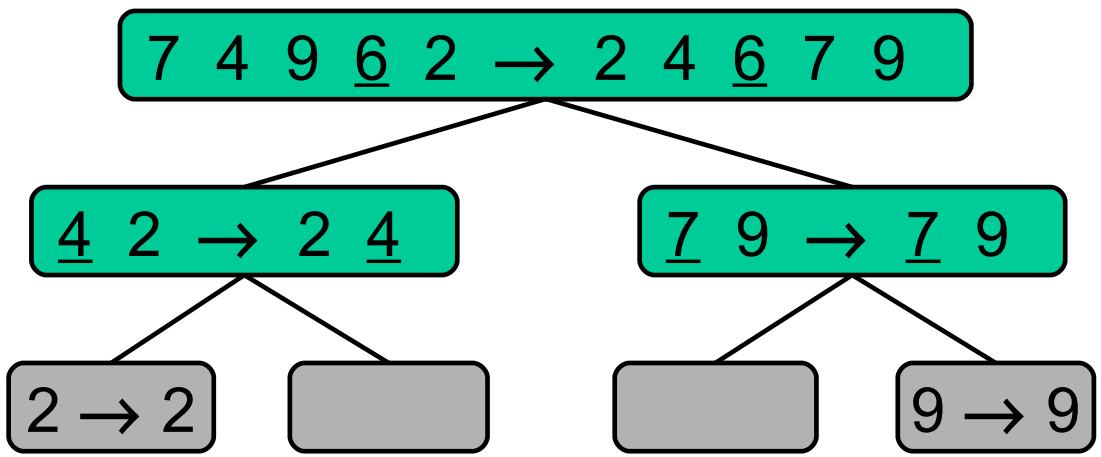
- Vi partitionerar indatasekvensen på följande vis:
  - Vi tar bort, i tur och ordning, varje element y från Seq och
  - Vi sätter in y i $Less$, Eq eller Grt beroende på resultatet av jämförelsen med pivot-elementet x
- Varje insättning och borttagning är i början eller slutet av en sekvens och tar alltså $O(1)$ tid
- Alltså tar partitioneringssteget i quick-sort $O(n)$ tid

Partitionering

```
function PARTITION(Seq, pivot)
    Less, Eq, Grt ← tomma sekvenser
    x ← Seq.REMOVE(pivot)
    while ¬Seq.ISEMPTY() do
        y ← Seq.REMOVE(Seq.FIRST())
        if y < x then
            Less.INSERTLAST(y)
        else if y = x then
            Eq.INSERTLAST(y)
        else
            Grt.INSERTLAST(y)
    return Less, Eq, Grt
```

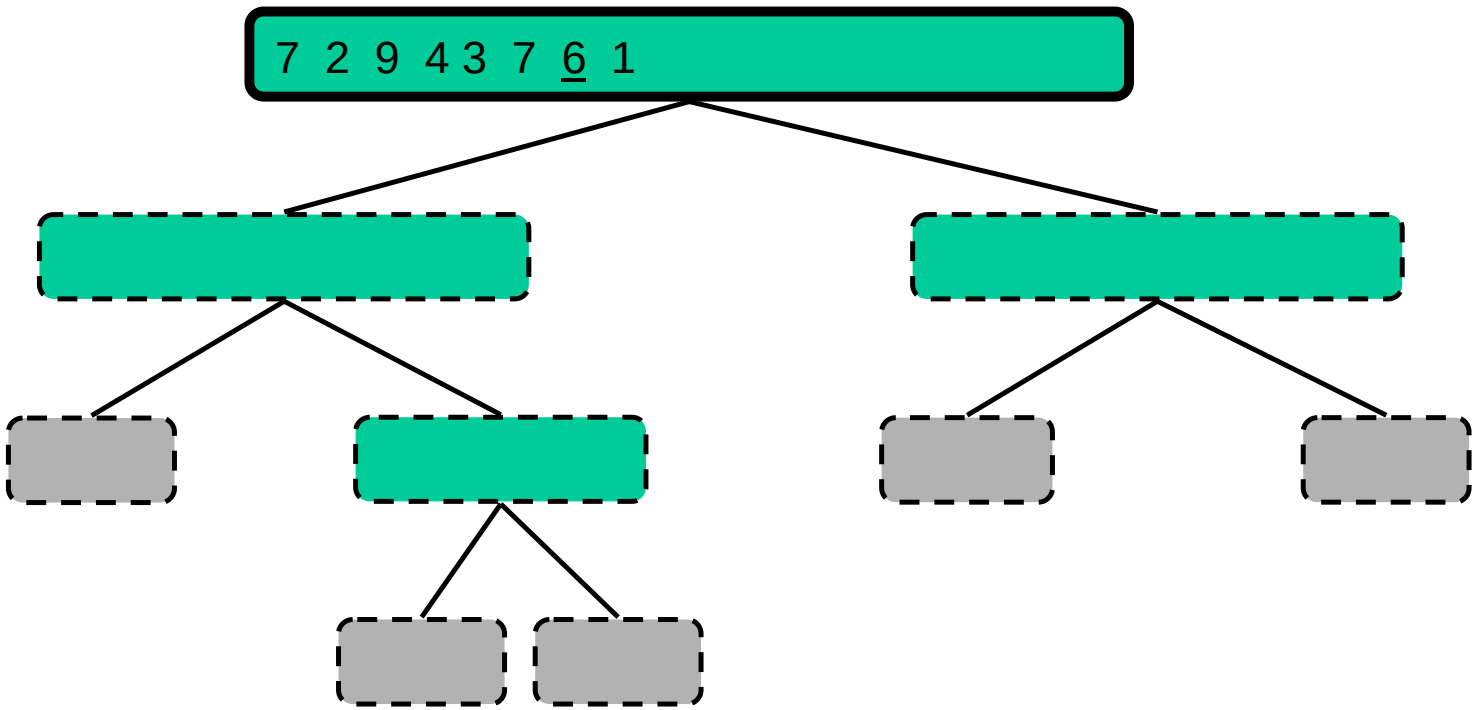
Quick-sortträdet

- Exekveringen av quick-sort kan visualiseras som ett binärt träd
  - Varje nod representerar ett rekursivt anrop till quick-sort och lagrar
    - Osorterad sekvens före exekveringen och dess pivot
    - Sorterad sekvens efter exekveringen
  - Roten är ursprungsanropet
  - Löven är anrop på delsekvenser av storlek 0 eller 1



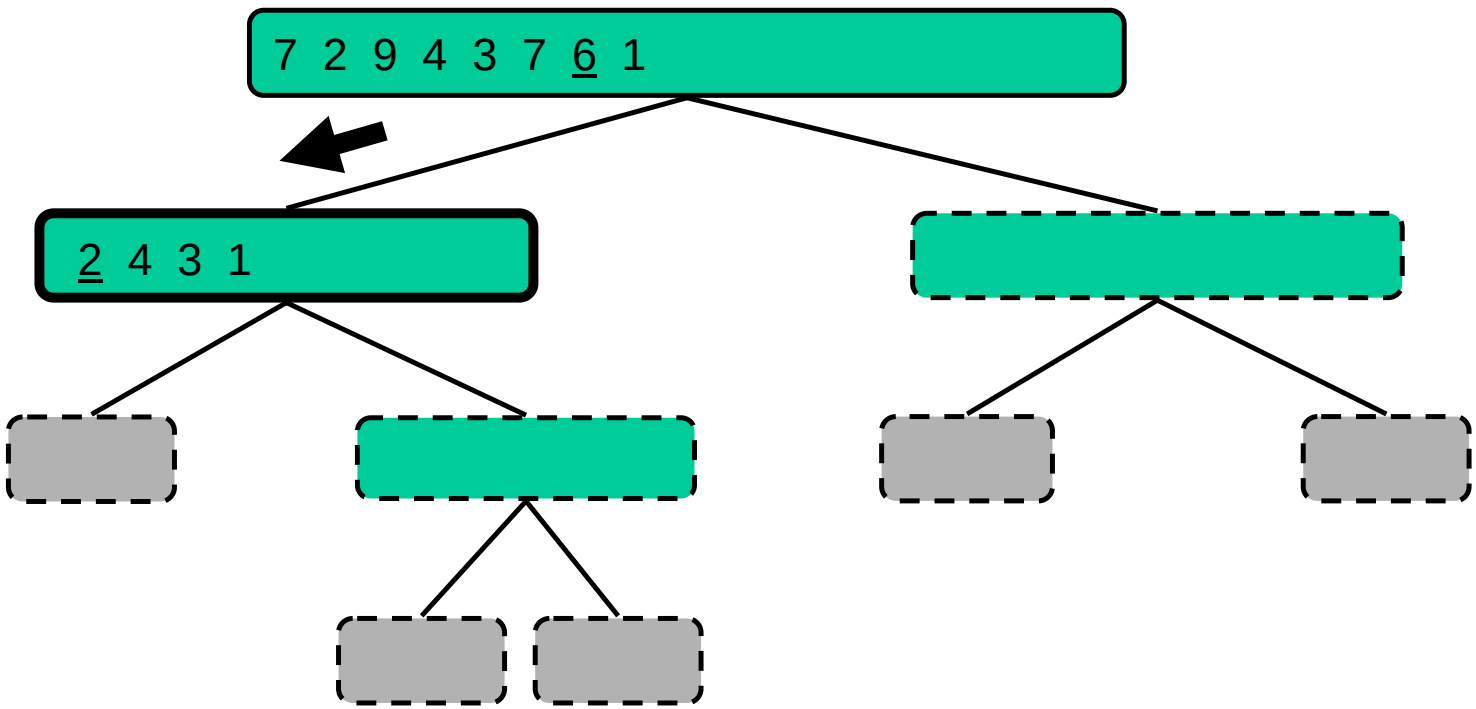
Exempel: Exekvering av quick-sort

- Val av pivot



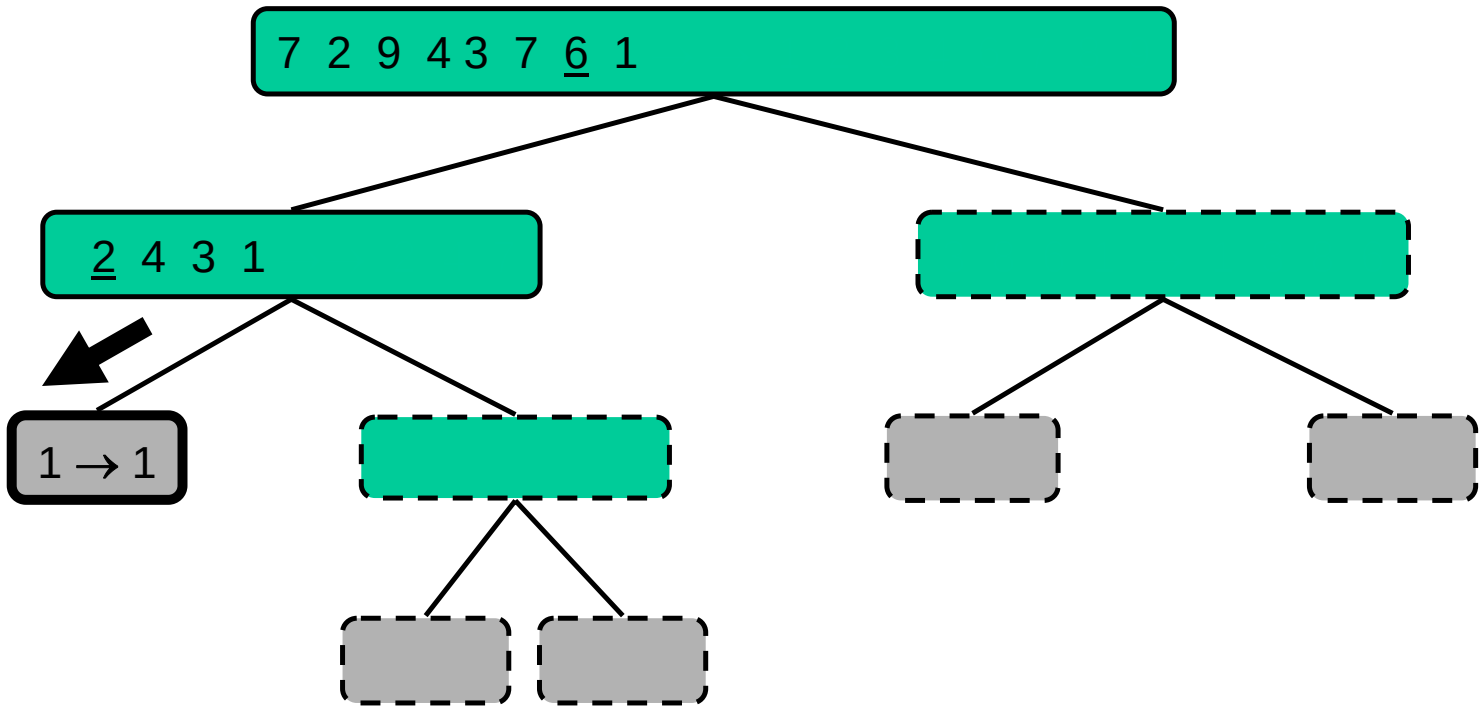
Exempel: Exekvering av quick-sort

- Partitionering, rekursivt anrop, val av pivot



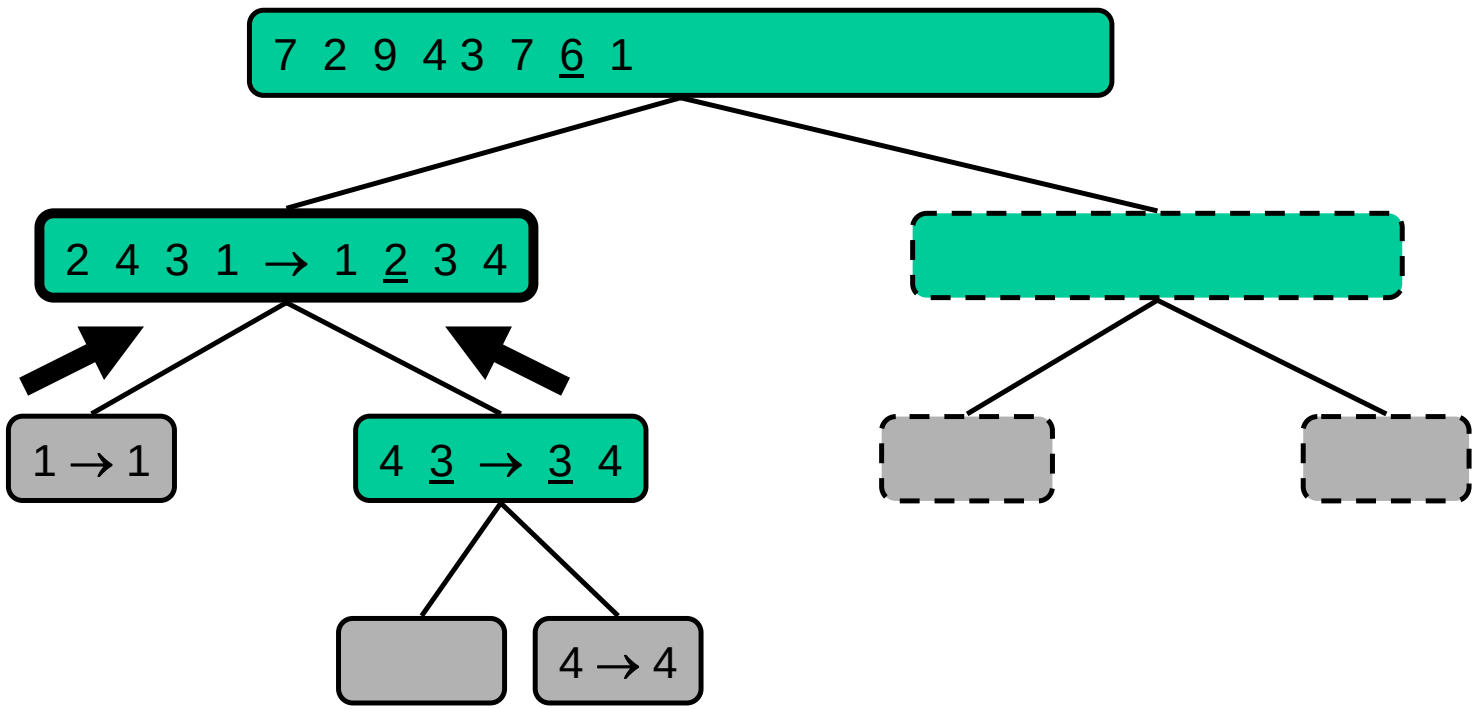
Exempel: Exekvering av quick-sort

- Partitionering, rekursivt anrop, basfall



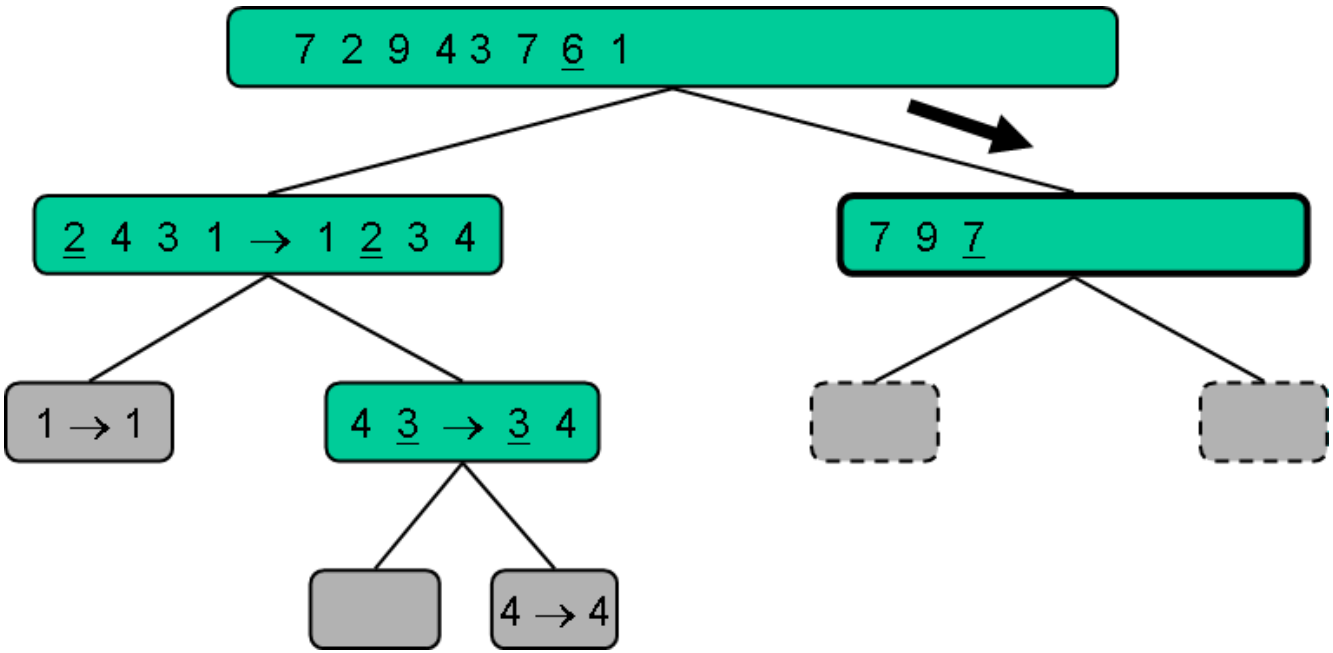
Exempel: Exekvering av quick-sort

- Rekursivt anrop, ..., basfall, kombinera



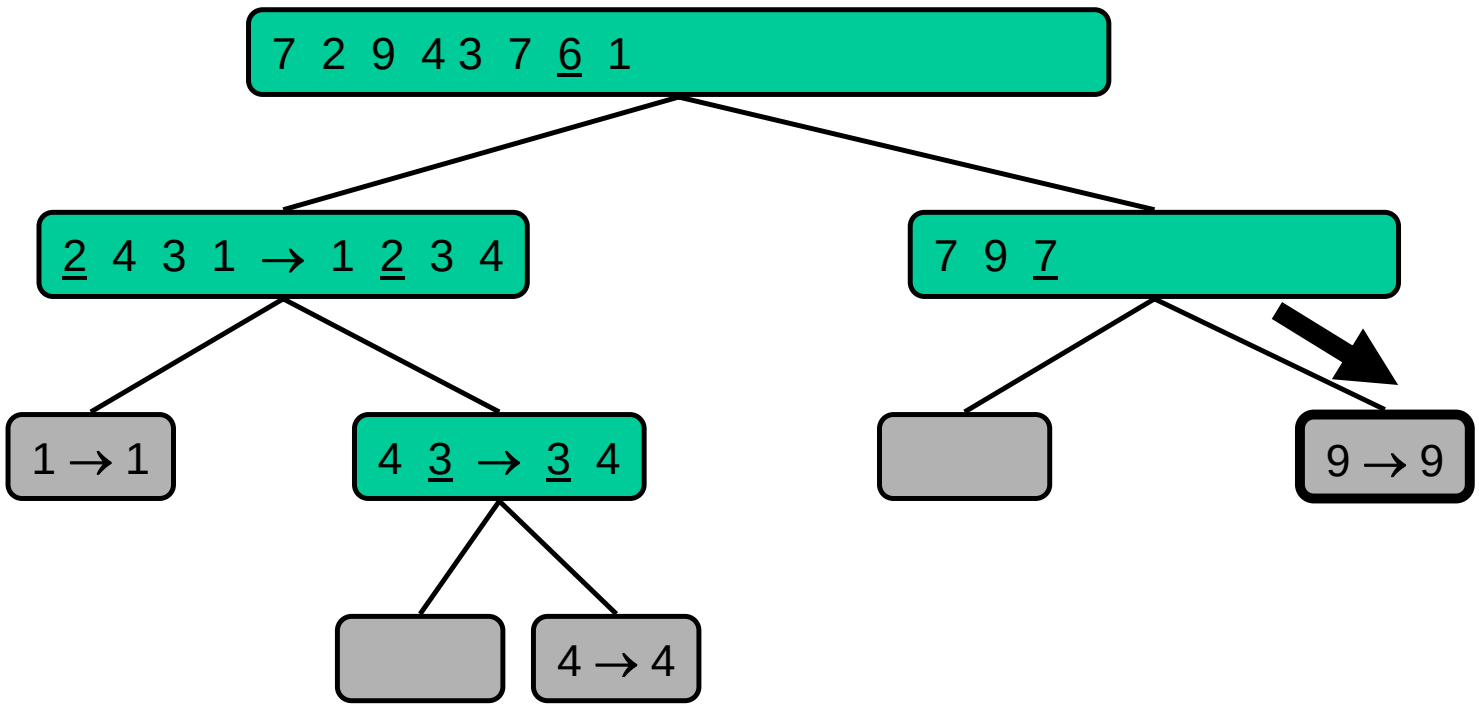
Exempel: Exekvering av quick-sort

- Rekursivt anrop, val av pivot



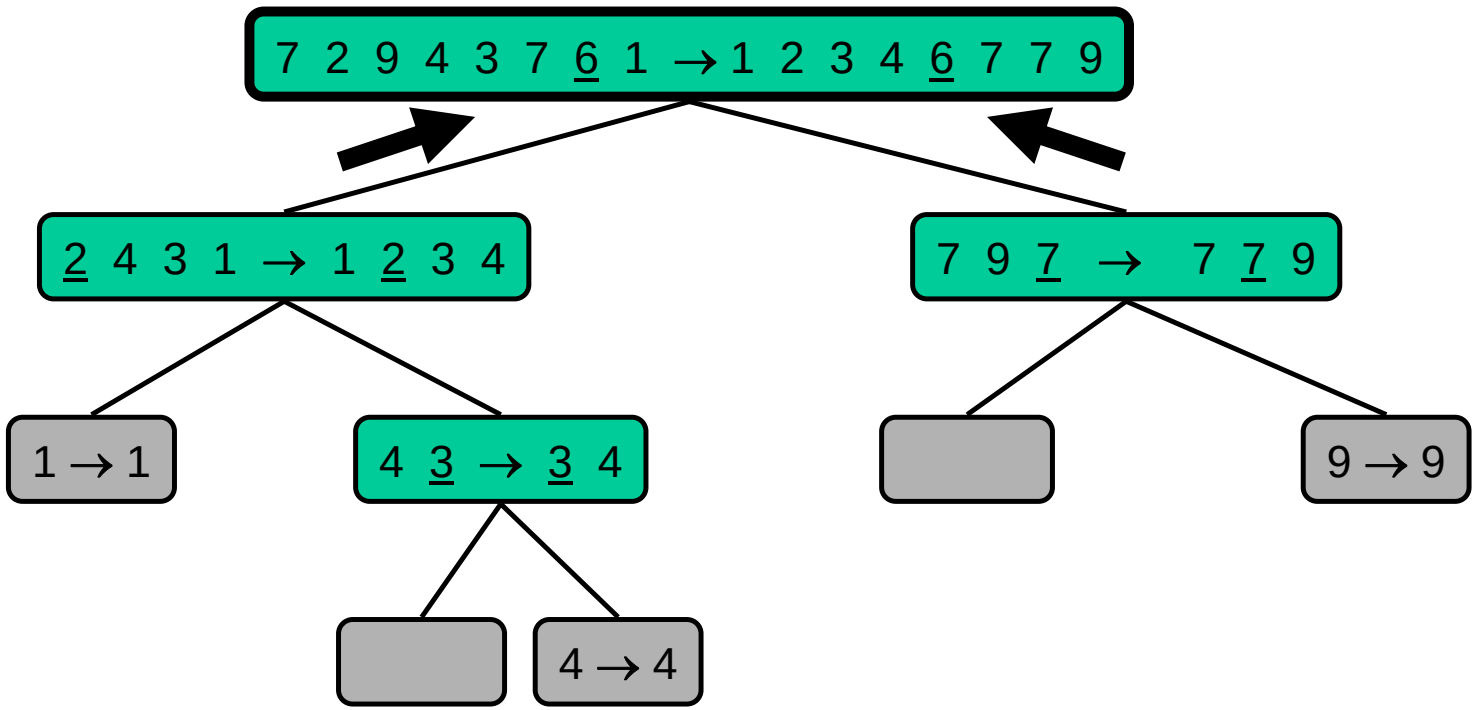
Exempel: Exekvering av quick-sort

- Partitionering, ..., rekursivt anrop, basfall



Exempel: Exekvering av quick-sort

- Kombinera, kombinera



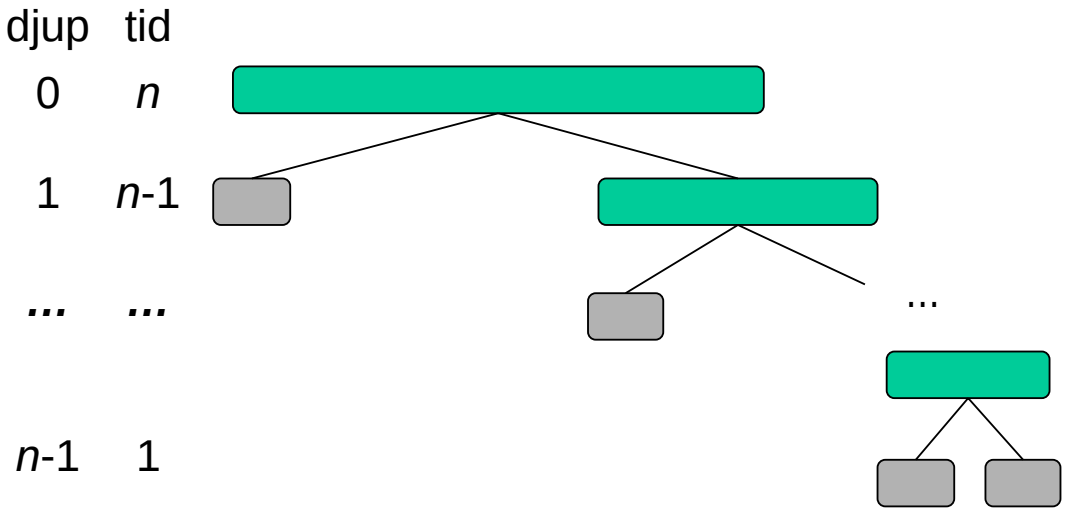
Exekveringstid i värsta fallet

- Värsta fallet för quick-sort uppkommer när pivotelementet är ett unikt minimalt eller maximalt element
- en av L eller G har storlek $n-1$ och den andra har storlek 0
- Exekveringstiden blir proportionell mot summan

$$n + (n-1) + \ldots + 2 + 1$$

- Alltså, värstafallstiden för quick-sort är $O(n^2)$

Exekveringstid i värsta fallet



Nästa gång:
Mer sortering...

www.liu.se

LINKÖPING UNIVERSITY