Föreläsning 7
Introduktion till sortering
TDDC91,TDDE22,725G97: DALG

Utskriftsversion av föreläsning i *Datastrukturer och algoritmer* 24 september 2018

Magnus Nielsen, IDA, Linköpings universitet

1 Sortering

1.1 Introduktion

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 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*, ...

1.2 Nybörjarsortering

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.

7.6

1.3 Insättningssortering

(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 $A[0,\ldots,i-1]$

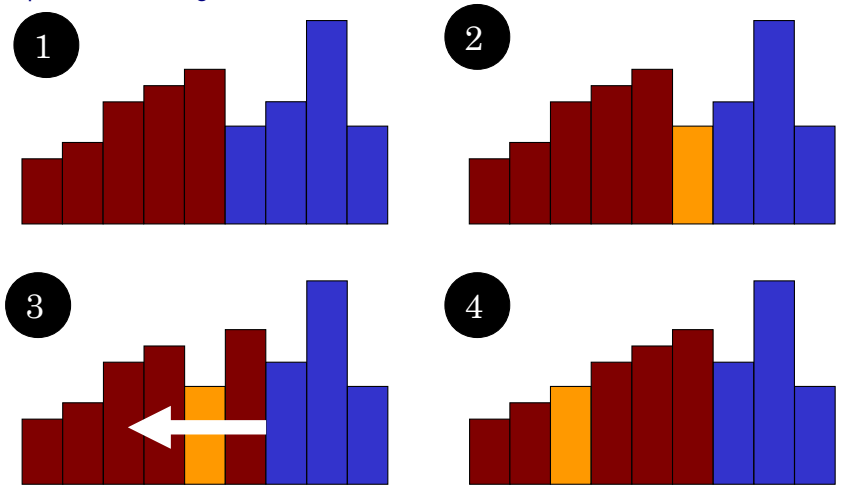
7.7

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

7.8

Exempel: Visualisering av Insertion-sort



7.9

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
```

- 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*

7.10

1.4 Urvalssortering

(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 $A[j]$

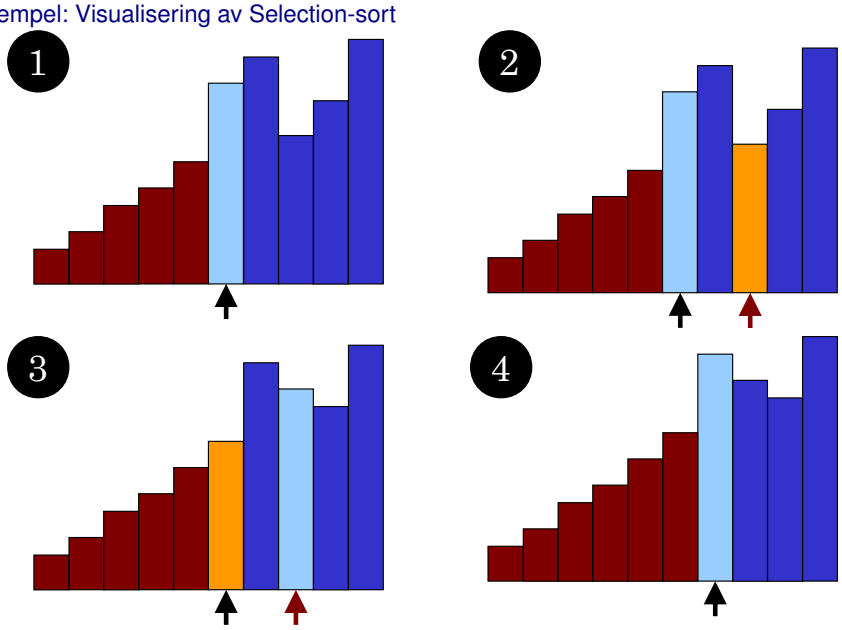
7.11

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

7.12

Exempel: Visualisering av Selection-sort



7.13

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])
```

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

7.14

1.5 Divide-and-conquer

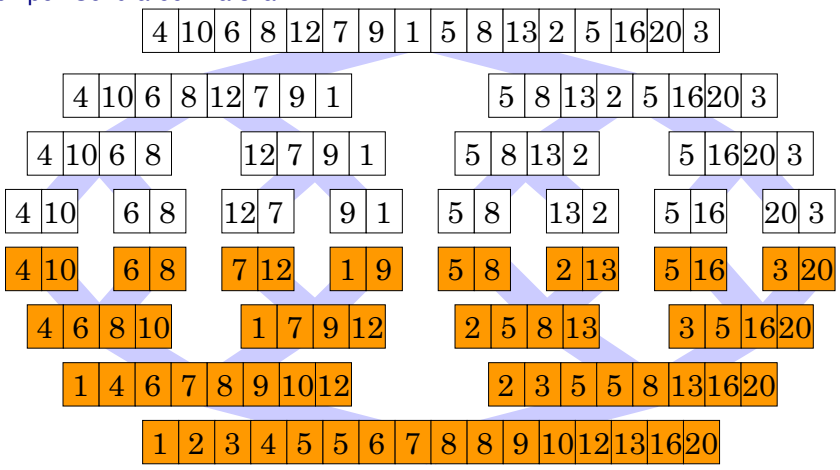
Principen söndra-och-härska för algoritmkonstruktion

- 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*

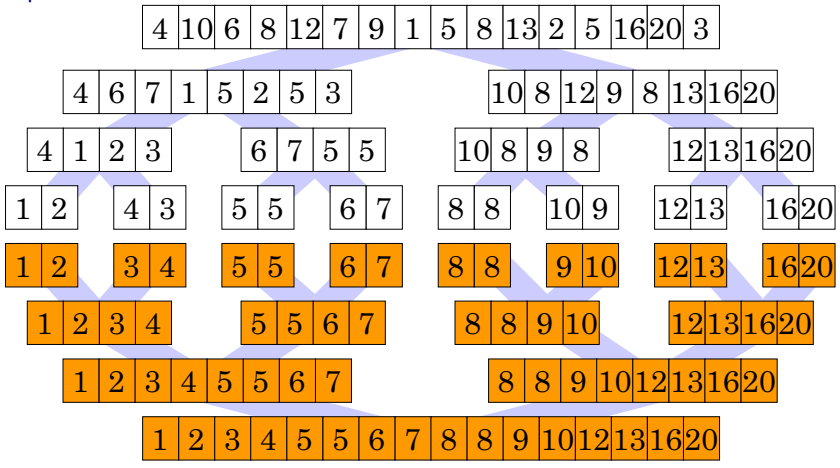
7.15

Exempel: Söndra-och-härska



7.16

Exempel: Söndra-och-härska



7.17

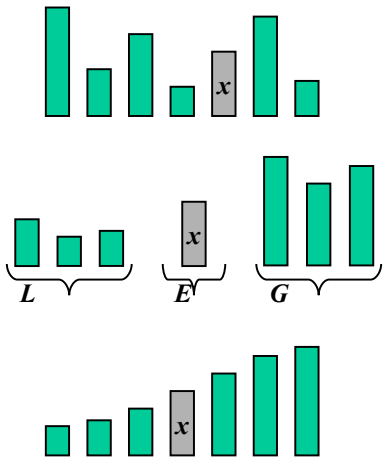
1.6 Quick-sort

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*



7.18

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

7.19

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

7.20

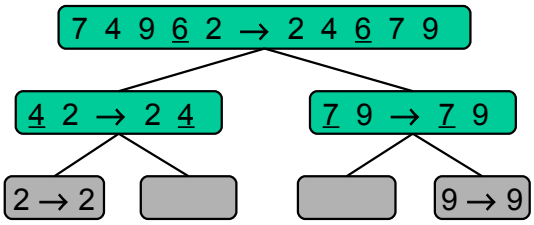
Partitionering

```
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
```

7.21

Quick-sortträdet

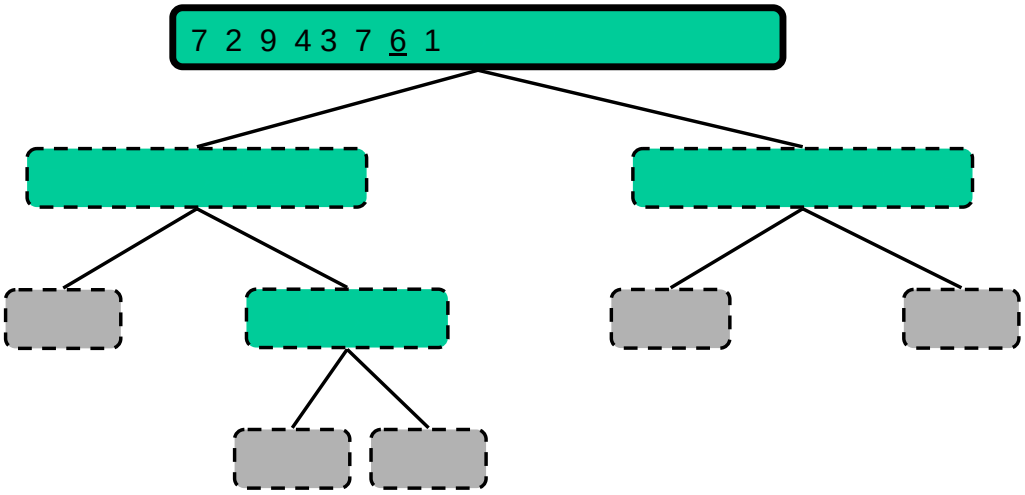
- Exekveringen av quick-sort kan visualiseras som ett binärt träd
  - Varje nod representerar ett rekursivt anrop till quick-sort och lagrar
    - Osorterad sekvens före exekveringen och dess pivot
    - Sorterad sekvens efter exekveringen
  - Roten är ursprungsanropet
  - Löven är anrop på delsekvenser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



7.22

Exempel: Exekvering av quick-sort

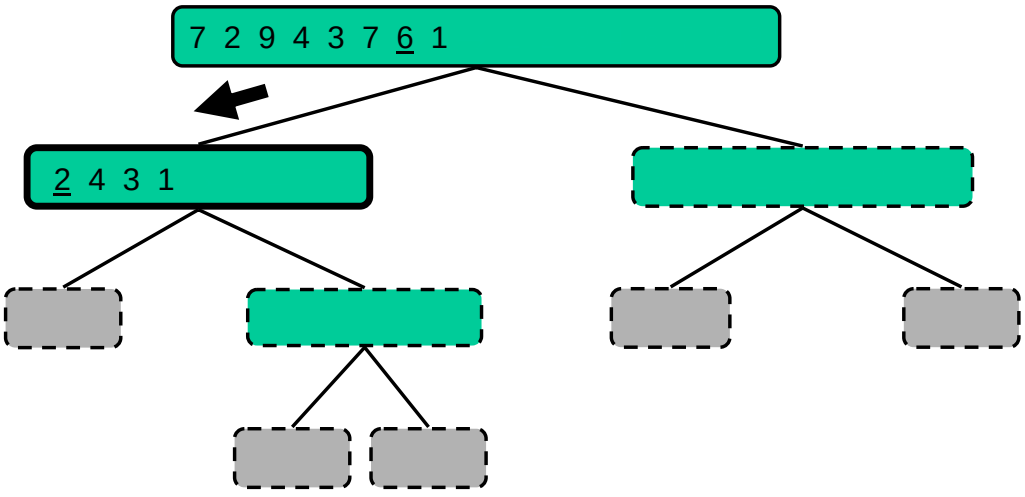
- Val av pivot



7.23

Exempel: Exekvering av quick-sort

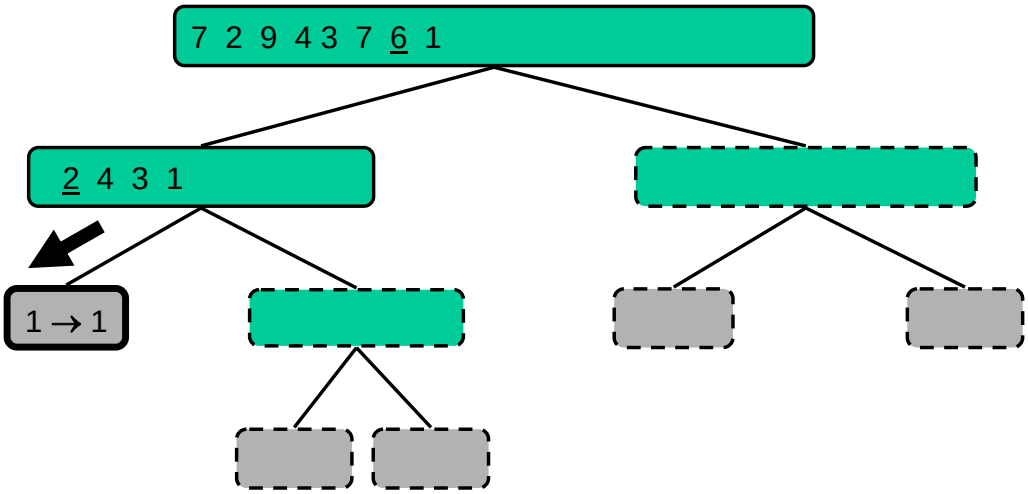
- Partitionering, rekursivt anrop, val av pivot



7.24

Exempel: Exekvering av quick-sort

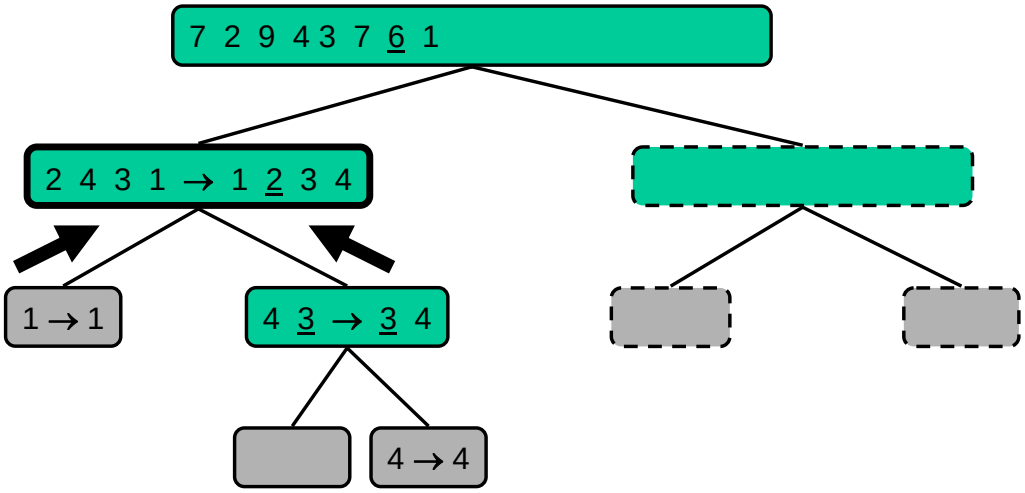
- Partitionering, rekursivt anrop, basfall



7.25

Exempel: Exekvering av quick-sort

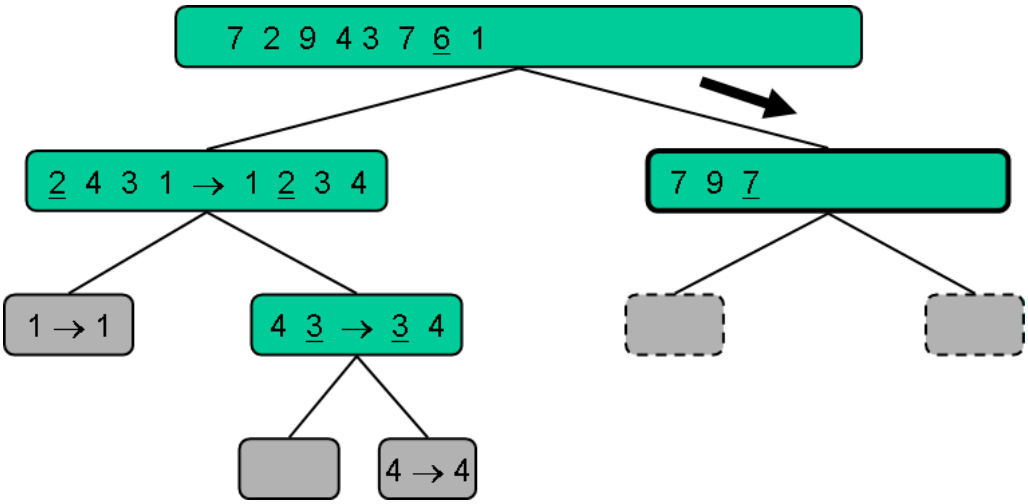
- Rekursivt anrop, . . . , basfall, kombinera



7.26

Exempel: Exekvering av quick-sort

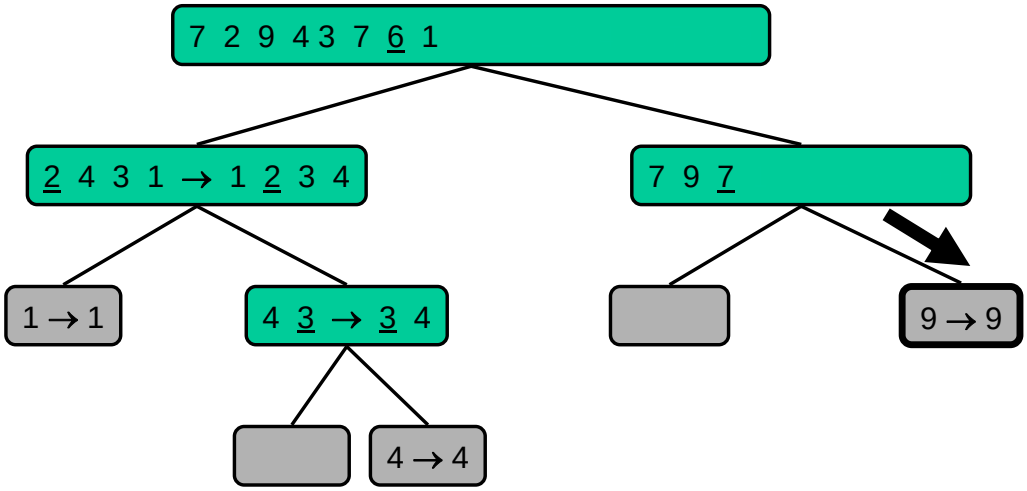
- Rekursivt anrop, val av pivot



7.27

Exempel: Exekvering av quick-sort

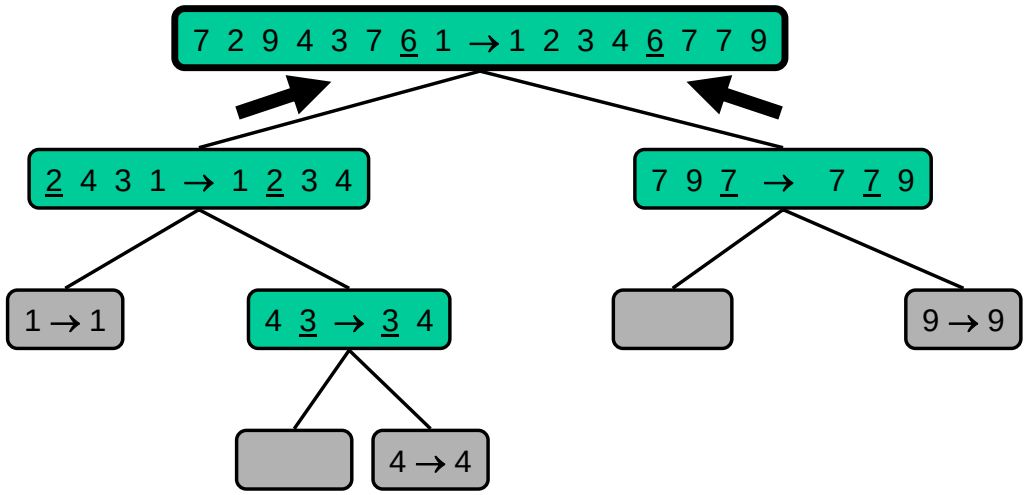
- Partitionering, . . . , rekursivt anrop, basfall



7.28

Exempel: Exekvering av quick-sort

- Kombinera, kombinera



7.29

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

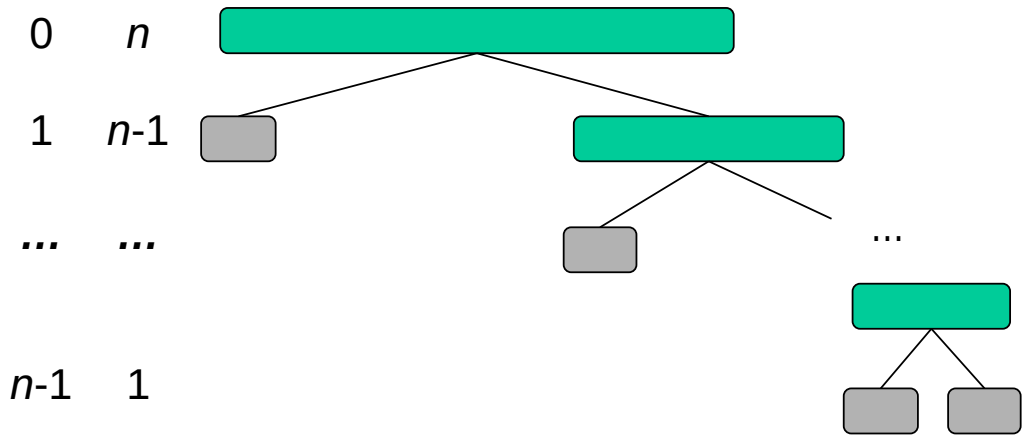
$$n+(n-1)+\ldots+2+1$$

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

7.30

Exekveringstid i värsta fallet

djup tid



7.31