

# Sortering

# Sortering

**Input:**  $n$  tal

**Output:** De  $n$  tal i sorteret orden

Eksempel:

6, 2, 9, 4, 5, 1, 4, 3  $\rightarrow$  1, 2, 3, 4, 4, 5, 6, 9

# Sortering

**Input:**  $n$  tal

**Output:** De  $n$  tal i sorteret orden

Eksempel:

6, 2, 9, 4, 5, 1, 4, 3  $\rightarrow$  1, 2, 3, 4, 4, 5, 6, 9

Mange opgaver er hurtigere i sorteret information (tænk på ordbøger, telefonbøger, adresselister i telefoner, . . . ). Dette gælder både for mennesker og for computere. Sortering er ofte en byggesten i algoritmer for andre problemer.

# Sortering

**Input:**  $n$  tal

**Output:** De  $n$  tal i sorteret orden

Eksempel:

6, 2, 9, 4, 5, 1, 4, 3  $\rightarrow$  1, 2, 3, 4, 4, 5, 6, 9

Mange opgaver er hurtigere i sorteret information (tænk på ordbøger, telefonbøger, adresselister i telefoner, . . . ). Dette gælder både for mennesker og for computere. Sortering er ofte en byggesten i algoritmer for andre problemer.

Sortering af information er en fundamental og central opgave.

# Sortering

**Input:**  $n$  tal

**Output:** De  $n$  tal i sorteret orden

Eksempel:

6, 2, 9, 4, 5, 1, 4, 3  $\rightarrow$  1, 2, 3, 4, 4, 5, 6, 9

Mange opgaver er hurtigere i sorteret information (tænk på ordbøger, telefonbøger, adresselister i telefoner, . . . ). Dette gælder både for mennesker og for computere. Sortering er ofte en byggesten i algoritmer for andre problemer.

Sortering af information er en fundamental og central opgave.

Mange algoritmer er udviklet: Insertionsort, Selectionsort, Bubblesort, Mergesort, Quicksort, Heapsort, Radixsort, Countingsort, . . .

Vi skal møde alle ovenstående i dette kursus.

# Sortering

Kommentarer:

- ▶ Sorteret orden kan være stigende eller faldende. Vi vil i dette kursus altid bruge stigende (mere præcist: ikke-faldende). Skal man sortere faldende, skal alle sammenligninger bare vendes.

# Sortering

Kommentarer:

- ▶ Sorteret orden kan være stigende eller faldende. Vi vil i dette kursus altid bruge stigende (mere præcist: ikke-faldende). Skal man sortere faldende, skal alle sammenligninger bare vendes.
- ▶ Vi vil antage, at input ligger i et array (Python: en liste).

# Sortering

Kommentarer:

- ▶ Sorteret orden kan være stigende eller faldende. Vi vil i dette kursus altid bruge stigende (mere præcist: ikke-faldende). Skal man sortere faldende, skal alle sammenligninger bare vendes.
- ▶ Vi vil antage, at input ligger i et array (Python: en liste).
- ▶ Man sorterer ofte elementer sammensat af en sorteringsnøgle samt yderligere information. Sorteringsnøglen kan være et tal, eller andet der kan sammenlignes (f.eks. strenge/ord). Vi viser i dette kursus blot elementer som rene tal.



# Insertionsort

Bruges af mange når man sorterer en hånd i kort:

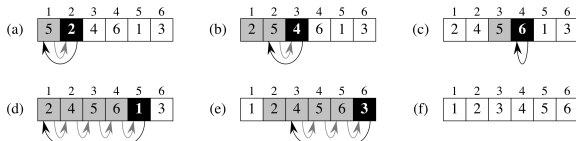


# Insertionsort

Bruges af mange når man sorterer en hånd i kort:



Samme idé udført på tal i et array:

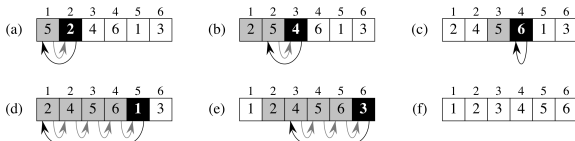


# Insertionsort

Bruges af mange når man sorterer en hånd i kort:



Samme idé udført på tal i et array:



Argument for korrekthed: Del af array til venstre for sorte felt er altid sorteret. Denne del udvides med én hele tiden ( $\Rightarrow$  algoritmen stopper, og når den stopper er alle elementer sorteret).

# Insertionsort

Som pseudo-kode:

INSERTION-SORT( $A, n$ )

**for**  $j = 2$  **to**  $n$

$key = A[j]$

    // Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .

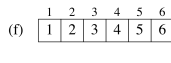
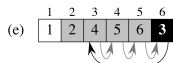
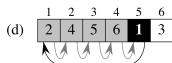
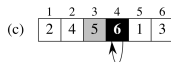
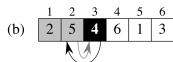
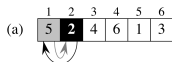
$i = j - 1$

**while**  $i > 0$  and  $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$



# Køretid for Insertionsort

Analyse:

	<i>cost</i>	<i>times</i>
INSERTION-SORT( $A, n$ )		
<b>for</b> $j = 2$ <b>to</b> $n$	$c_1$	$n$
$key = A[j]$	$c_2$	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$ .	0	$n - 1$
$i = j - 1$	$c_4$	$n - 1$
<b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
$A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
$i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
$A[i + 1] = key$	$c_8$	$n - 1$

Her er  $t_j$  hvor mange gange testen i den indre **while**-løkke udføres. Dvs.  $t_j - 1$  er hvor mange gange løkken kører (hvilket er hvor mange elementer det  $j$ 'te element skal forbi under indsættelsen). Sæt  $c = c_1 + c_2 + \dots + c_8$ .

# Køretid for Insertionsort

Analyse:

INSERTION-SORT( $A, n$ )	<i>cost</i>	<i>times</i>
<b>for</b> $j = 2$ <b>to</b> $n$	$c_1$	$n$
$key = A[j]$	$c_2$	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$ .	0	$n - 1$
$i = j - 1$	$c_4$	$n - 1$
<b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
$A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
$i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
$A[i + 1] = key$	$c_8$	$n - 1$

Her er  $t_j$  hvor mange gange testen i den indre **while**-løkke udføres. Dvs.  $t_j - 1$  er hvor mange gange løkken kører (hvilket er hvor mange elementer det  $j$ 'te element skal forbi under indsættelsen). Sæt  $c = c_1 + c_2 + \dots + c_8$ .

Best case:  $t_j = 1$  for alle  $j$ . Samlet tid  $\leq c \cdot n$ .

# Køretid for Insertionsort

Analyse:

INSERTION-SORT( $A, n$ )	<i>cost</i>	<i>times</i>
<b>for</b> $j = 2$ <b>to</b> $n$	$c_1$	$n$
$key = A[j]$	$c_2$	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$ .	0	$n - 1$
$i = j - 1$	$c_4$	$n - 1$
<b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
$A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
$i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
$A[i + 1] = key$	$c_8$	$n - 1$

Her er  $t_j$  hvor mange gange testen i den indre **while**-løkke udføres. Dvs.  $t_j - 1$  er hvor mange gange løkken kører (hvilket er hvor mange elementer det  $j$ 'te element skal forbi under indsættelsen). Sæt  $c = c_1 + c_2 + \dots + c_8$ .

Best case:  $t_j = 1$  for alle  $j$ . Samlet tid  $\leq c \cdot n$ .

Worst case:  $t_j = j$  for alle  $j$ . Samlet tid  $\leq c \cdot n^2$ , da

$$\sum_{j=1}^n j = (1 + 2 + 3 + \dots + n) = \frac{(n+1)n}{2} = \frac{n^2 + n}{2} \leq \frac{2n^2}{2} = n^2.$$

# Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

*IndListe* = input

*UdListe* = tom liste

While *IndListe* ikke tom:

    find mindste element  $x$  i *IndListe*

    flyt  $x$  fra *IndListe* til enden af *UdListe*



# Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

*IndListe* = input

*UdListe* = tom liste

While *IndListe* ikke tom:

    find mindste element  $x$  i *IndListe*

    flyt  $x$  fra *IndListe* til enden af *UdListe*

Klart korrekt.

# Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

*IndListe* = input

*UdListe* = tom liste

While *IndListe* ikke tom:

    find mindste element  $x$  i *IndListe*

    flyt  $x$  fra *IndListe* til enden af *UdListe*

Klart korrekt.

Køretid?

# Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

*IndListe* = input

*UdListe* = tom liste

While *IndListe* ikke tom:

    find mindste element  $x$  i *IndListe*

    flyt  $x$  fra *IndListe* til enden af *UdListe*

Klart korrekt.

Køretid?

I alt  $n$  gange findes mindste element i *IndListe*. Simpel metode er lineær søgning  $\Rightarrow$  tid  $\leq c \cdot (n + (n - 1) + (n - 2) + \dots + 1) \leq c \cdot n^2$ .

# Merge

Input:

To *sorterede* rækker

2,4,5,7

1,2,3,6

# Merge

**Input:** To *sorterede* rækker 2,4,5,7 1,2,3,6

**Output:** De samme elementer i én sorteret række 1,2,2,3,4,5,6,7

## Merge

**Input:** To *sorterede* rækker 2,4,5,7 1,2,3,6  
**Output:** De samme elementer i én sorteret række 1,2,2,3,4,5,6,7

Vi kan naturligvis sortere.

# Merge

**Input:** To *sorterede* rækker 2,4,5,7 1,2,3,6  
**Output:** De samme elementer i én sorteret række 1,2,2,3,4,5,6,7

Vi kan naturligvis sortere. Men det er hurtigere at **flette** (merge):

Repeat:

Flyt det mindste af de to forreste elementer

# Merge

Input: To *sorterede* rækker

2,4,5,7 1,2,3,6

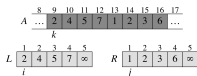
Output: De samme elementer i én sorteret række

1,2,2,3,4,5,6,7

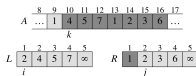
Vi kan naturligvis sortere. Men det er hurtigere at **flette** (merge):

Repeat:

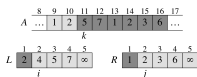
Flyt det mindste af de to forreste elementer



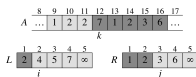
(a)



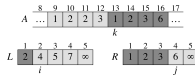
(b)



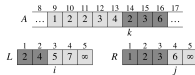
(c)



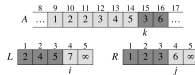
(d)



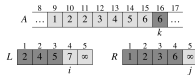
(e)



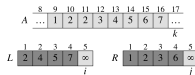
(f)



(g)



(h)



(i)



# Merge

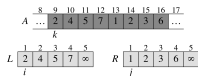
Input: To *sorterede* rækker 2,4,5,7 1,2,3,6

Output: De samme elementer i én sorteret række 1,2,2,3,4,5,6,7

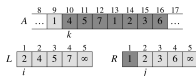
Vi kan naturligvis sortere. Men det er hurtigere at **flette** (merge):

Repeat:

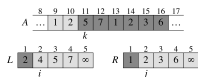
Flyt det mindste af de to forreste elementer



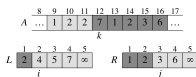
(a)



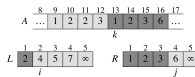
(b)



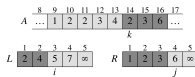
(c)



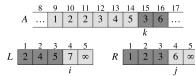
(d)



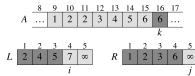
(e)



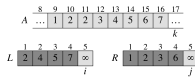
(f)



(g)



(h)



(i)

Tid:  $\leq c \cdot n$ , hvor  $n$  = antal elementer i alt.

# Merge

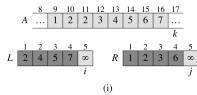
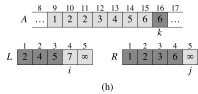
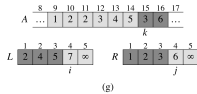
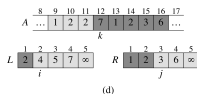
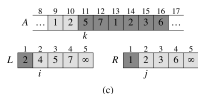
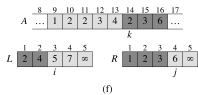
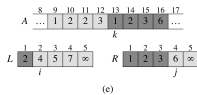
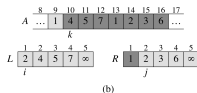
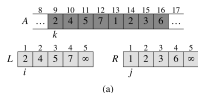
Input: To *sorterede* rækker 2,4,5,7 1,2,3,6

Output: De samme elementer i én sorteret række 1,2,2,3,4,5,6,7

Vi kan naturligvis sortere. Men det er hurtigere at **flette** (merge):

Repeat:

Flyt det mindste af de to forreste elementer



Tid:  $\leq c \cdot n$ , hvor  $n$  = antal elementer i alt.

Korrekthed: Merge kan ses som en udgave af Selectionsort.

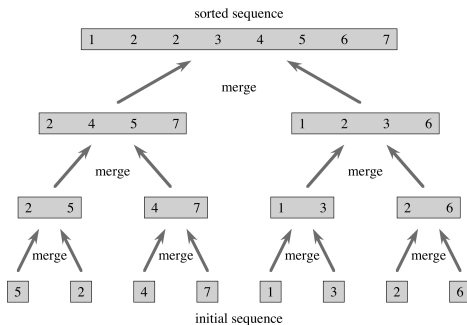
# Merge

Som pseudo-kode, med to rækker  $A[p \dots q]$  and  $A[q + 1 \dots r]$ :

```
MERGE( $A, p, q, r$ )  
   $n_1 = q - p + 1$   
   $n_2 = r - q$   
  let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays  
  for  $i = 1$  to  $n_1$   
     $L[i] = A[p + i - 1]$   
  for  $j = 1$  to  $n_2$   
     $R[j] = A[q + j]$   
   $L[n_1 + 1] = \infty$   
   $R[n_2 + 1] = \infty$   
   $i = 1$   
   $j = 1$   
  for  $k = p$  to  $r$   
    if  $L[i] \leq R[j]$   
       $A[k] = L[i]$   
       $i = i + 1$   
    else  $A[k] = R[j]$   
       $j = j + 1$ 
```

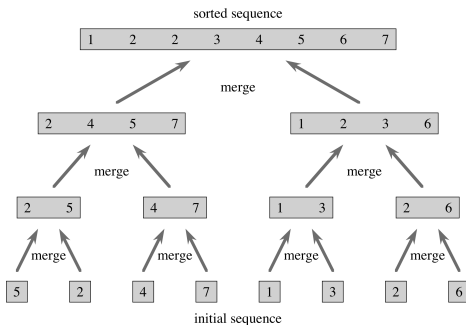
# Mergesort

Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:



# Mergesort

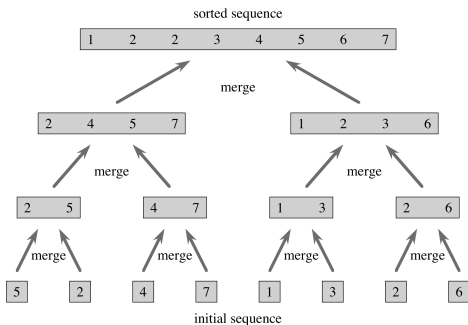
Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:



Tid:

# Mergesort

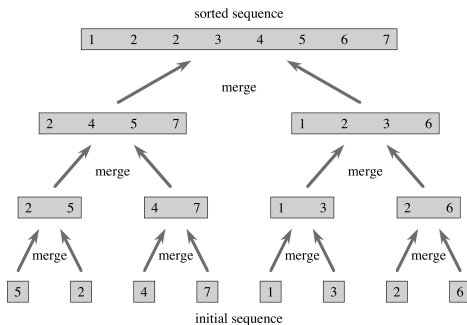
Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:



Tid: Hver merge bruger højst  $c \cdot n_1$  tid, hvor  $n_1$  er antal elementer der merges.

# Mergesort

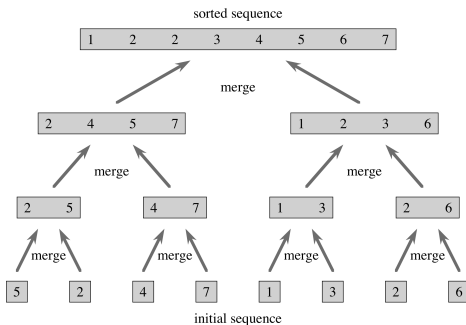
Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:



Tid: Hver merge bruger højst  $c \cdot n_1$  tid, hvor  $n_1$  er antal elementer der merges. Så alle merge-operationer i ét lag bruger tilsammen højst  $c \cdot (n_1 + n_2 + \dots) = c \cdot n$ . Dette gælder alle lagene.

# Mergesort

Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:



Tid: Hver merge bruger højst  $c \cdot n_1$  tid, hvor  $n_1$  er antal elementer der merges. Så alle merge-operationer i ét lag bruger tilsammen højst  $c \cdot (n_1 + n_2 + \dots) = c \cdot n$ . Dette gælder alle lagene. Der er i alt  $\log_2 n$  lag, så den samlede tid er højst  $c \cdot n \cdot \log_2 n$ .



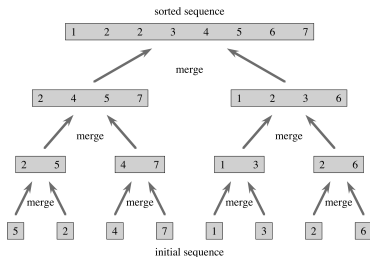
# Mergesort

Hvorfor er der  $\log_2 n$  merge-lag?

# Mergesort

Hvorfor er der  $\log_2 n$  merge-lag?

Antal sorterede lister efter  $k$  merge-lag (antag  $n$  er en potens af 2):

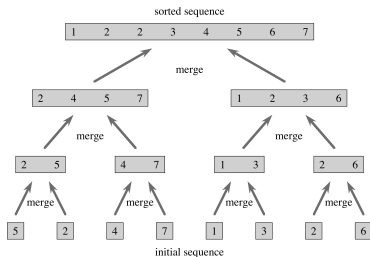


$k$	Antal lister
$\vdots$	$\vdots$
$k$	$n/2^k$
$\vdots$	$\vdots$
3	$n/2^3$
2	$n/2^2$
1	$n/2$
0	$n$

# Mergesort

Hvorfor er der  $\log_2 n$  merge-lag?

Antal sorterede lister efter  $k$  merge-lag (antag  $n$  er en potens af 2):



$k$	Antal lister
$\vdots$	$\vdots$
$k$	$n/2^k$
$\vdots$	$\vdots$
3	$n/2^3$
2	$n/2^2$
1	$n/2$
0	$n$

Algoritmen stopper når der er én sorteret liste:

$$n/2^k = 1 \Leftrightarrow n = 2^k \Leftrightarrow \log_2 n = k$$

# Mergesort

Hvis  $n$  ikke er en potens af 2?

## Mergesort

Hvis  $n$  ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister bliver til 6 lister mens 13 ( $= 12 + 1$ ) lister til 7 ( $= 6 + 1$ ) lister.

# Mergesort

Hvis  $n$  ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister bliver til 6 lister mens 13 ( $= 12 + 1$ ) lister til 7 ( $= 6 + 1$ ) lister.

Generelt: Hvis der er  $x$  lister før et merge-lag, er der  $\lceil x/2 \rceil$  lister efter.

# Mergesort

Hvis  $n$  ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister bliver til 6 lister mens 13 ( $= 12 + 1$ ) lister til 7 ( $= 6 + 1$ ) lister.

Generelt: Hvis der er  $x$  lister før et merge-lag, er der  $\lceil x/2 \rceil$  lister efter.

Se på to input størrelse  $n_1$  og  $n_2$ , med  $n_1 \leq n_2$ . Da  $\lceil x/2 \rceil$  er en voksende funktion af  $x$ , kan antallet af lister i hvert lag ikke være mindre for  $n_2$  end for  $n_1$ . Derfor kan antallet af merge-lag (før algoritmen når ned på én liste) ikke være mindre for  $n_2$  end for  $n_1$ .

# Mergesort

Hvis  $n$  ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister bliver til 6 lister mens 13 ( $= 12 + 1$ ) lister til 7 ( $= 6 + 1$ ) lister.

Generelt: Hvis der er  $x$  lister før et merge-lag, er der  $\lceil x/2 \rceil$  lister efter.

Se på to input størrelse  $n_1$  og  $n_2$ , med  $n_1 \leq n_2$ . Da  $\lceil x/2 \rceil$  er en voksende funktion af  $x$ , kan antallet af lister i hvert lag ikke være mindre for  $n_2$  end for  $n_1$ . Derfor kan antallet af merge-lag (før algoritmen når ned på én liste) ikke være mindre for  $n_2$  end for  $n_1$ .

Sæt  $n_2$  til mindste potens af to, som er  $\geq n_1$ . Som set tidligere er der præcis  $\log_2 n_2$  merge-lag for  $n_2$ , og dermed højst så mange merge-lag for  $n_1$ .

Så der er  $\lceil \log_2 n \rceil$  merge lag for generelt  $n$ .

$n$	7	$8 = 2^3$	9	10	11	12	13	14	15	$16 = 2^4$	17
$\log_2(n)$	2.807	3	3.169	3.321	3.459	3.584	3.700	3.807	3.906	4	4.087
Antal lag	3	3	4	4	4	4	4	4	4	4	5



# Mergesort

Mergesort som pseudo-kode, i en variant formuleret med rekursion:

MERGE-SORT( $A, p, r$ )

**if**  $p < r$

$q = \lfloor (p + r) / 2 \rfloor$

MERGE-SORT( $A, p, q$ )

MERGE-SORT( $A, q + 1, r$ )

MERGE( $A, p, q, r$ )

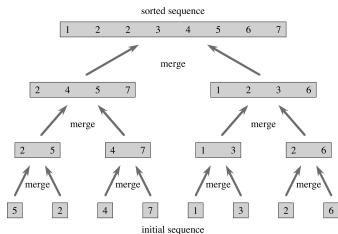
// check for base case

// divide

// conquer

// conquer

// combine



Et kald  $\text{MERGE-SORT}(A, p, r)$  har til opgave at stille elementerne i  $A[p \dots r]$  i sorteret orden. Første kald er  $\text{MERGE-SORT}(A, 1, n)$ , som har til opgave at sortere hele  $A$ .

# Quicksort

Mergesort:

- ▶ Del input op i to dele  $X$  og  $Y$  (trivielt)
- ▶ Sorter hver del for sig (rekursion)
- ▶ Merge de to sorterede dele til én sorteret del (reelt arbejde)

Basistilfælde:  $n \leq 1$  (allerede sorteret, gør intet)

# Quicksort

Mergesort:

- ▶ Del input op i to dele  $X$  og  $Y$  (trivielt)
- ▶ Sorter hver del for sig (rekursion)
- ▶ Merge de to sorterede dele til én sorteret del (reelt arbejde)

Basistilfælde:  $n \leq 1$  (allerede sorteret, gør intet)

Quicksort:

- ▶ Del input op i to dele  $X$  og  $Y$  så  $X \leq Y$  (reelt arbejde)
- ▶ Sorter hver del for sig (rekursion)
- ▶ Returner  $X$  efterfulgt af  $Y$  (trivielt)

Basistilfælde:  $n \leq 1$  (allerede sorteret, gør intet)

[Hoare, 1960]

# Quicksort

Som pseudo-kode:

```
QUICKSORT( $A, p, r$ )  
  if  $p < r$   
     $q = \text{PARTITION}(A, p, r)$   
    QUICKSORT( $A, p, q - 1$ )  
    QUICKSORT( $A, q + 1, r$ )
```

# Quicksort

Som pseudo-kode:

```
QUICKSORT( $A, p, r$ )  
  if  $p < r$   
     $q = \text{PARTITION}(A, p, r)$   
    QUICKSORT( $A, p, q - 1$ )  
    QUICKSORT( $A, q + 1, r$ )
```

Et kald  $\text{QUICKSORT}(A, p, r)$  har til opgave at stille elementerne i  $A[p \dots r]$  i sorteret orden. Første kald er  $\text{QUICKSORT}(A, 1, n)$ , som har til opgave at sortere hele  $A$ .

Et kald  $\text{PARTITION}(A, p, r)$  vælger et element  $x \in A$  og opdeler  $A[p \dots r]$  således at:

- ▶  $A[q] = x$
- ▶  $A[p \dots q - 1] \leq x$
- ▶  $A[q + 1 \dots r] > x$

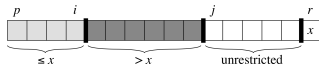
# Partition

Hvordan lave PARTITION?

# Partition

Hvordan lave PARTITION?

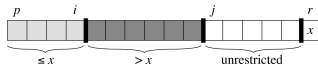
Idé: Vælg et element  $x$  fra input at opdele efter (her sidste element i array-del). Opbyg de to dele under et gennemløb af array ud fra flg. princip:



# Partition

Hvordan lave PARTITION?

Idé: Vælg et element  $x$  fra input at opdele efter (her sidste element i array-del). Opbyg de to dele under et gennemløb af array ud fra flg. princip:



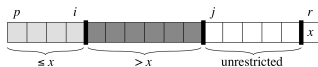
Hvordan tage et skridt under gennemløb?



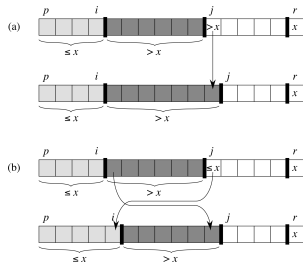
# Partition

Hvordan lave PARTITION?

Idé: Vælg et element  $x$  fra input at opdele efter (her sidste element i array-del). Opbyg de to dele under et gennemløb af array ud fra flg. princip:

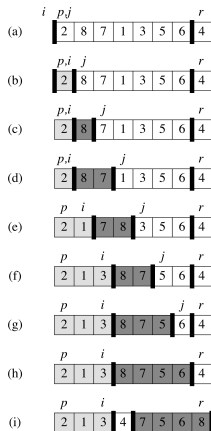


Hvordan tage et skridt under gennemløb?



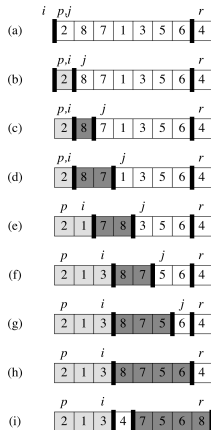
# Partition

Et eksempel på gennemløb:



# Partition

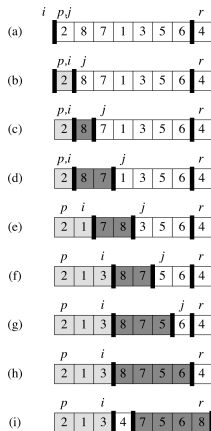
Et eksempel på gennemløb:



Tid:

# Partition

Et eksempel på gennemløb:



Tid:  $O(n)$ , hvor  $n$  er antal elementer i  $A[p \dots r]$ .

# Partition

Som pseudo-kode:

**PARTITION**( $A, p, r$ )

$x = A[r]$

$i = p - 1$

**for**  $j = p$  **to**  $r - 1$

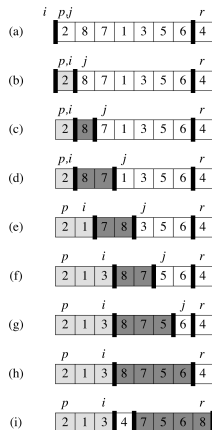
**if**  $A[j] \leq x$

$i = i + 1$

        exchange  $A[i]$  with  $A[j]$

exchange  $A[i + 1]$  with  $A[r]$

**return**  $i + 1$



# Quicksort køretid

## Quicksort køretid

Hænger på, hvordan partitions gennem rekursionen deler input.

## Quicksort køretid

Hænger på, hvordan partitions gennem rekursionen deler input.

To ekstremer af størrelser på rekursive kald:

- ▶ Helt ubalanceret:  $0$  og  $n - 1$
- ▶ Helt balanceret:  $\lceil (n - 1)/2 \rceil$  og  $\lfloor (n - 1)/2 \rfloor$



## Quicksort køretid

Hænger på, hvordan partitions gennem rekursionen deler input.

To ekstremer af størrelser på rekursive kald:

- ▶ Helt ubalanceret: 0 og  $n - 1$
- ▶ Helt balanceret:  $\lceil (n - 1)/2 \rceil$  og  $\lfloor (n - 1)/2 \rfloor$
  
- ▶ Hvis alle partitions er helt balancerede:  $O(n \log n)$  (ca. samme analyse som for Mergesort).
- ▶ Hvis alle partitions er helt ubalancerede:  
 $O(n + (n - 1) + (n - 2) + \dots + 2 + 1) = O(n^2)$ .

Man kan vise at dette er henholdsvis best case og worst case for Quicksort.

## Quicksort køretid

- ▶ I praksis:  $O(n \log n)$  for næsten alle input.
- ▶ Dog: sorteret input giver  $\Theta(n^2)$  for ovenstående valg af opdelingselement  $x$  i partition (brug *ikke* det valg i praksis).
- ▶ Mere robuste valg af opdelingselement  $x$ : enten som midterelementet, som medianen af flere elementer, som et tilfældigt element, eller som medianen af flere tilfældigt valgte elementer.
- ▶ Quicksort er *inplace*: bruger ikke mere plads end input-array'et.
- ▶ Kode er meget effektiv i praksis. En godt implementeret Quicksort er ofte bedste all-round sorteringsalgoritme (og valgt i mange biblioteker, f.eks. Java og C++/STL).

# Heapsort

En **Heap** er:

# Heapsort

En **Heap** er:

1. et binært træ
2. med heap-orden
3. og heap-facon
4. udlagt i et array

# Heapsort

En **Heap** er:

1. et binært træ
2. med heap-orden
3. og heap-facon
4. udlagt i et array

(Note: “heap” bruges også om et hukommelsesområde brugt til allokering af objekter under et programs udførelse. De to anvendelser er urelaterede.)

[Williams, 1964]

# 1) Binært træ

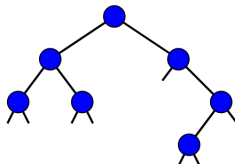
Et binært træ er enten

- ▶ det **tomme træ**

eller

- ▶ en **knude**  $v$  (evt. med indhold af data) samt **to undertræer** (et højre og et venstre).

Visualisering:



# 1) Binært træ

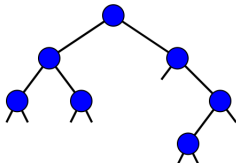
Et binært træ er enten

- ▶ det **tomme træ**

eller

- ▶ en **knude**  $v$  (evt. med indhold af data) samt **to undertræer** (et højre og et venstre).

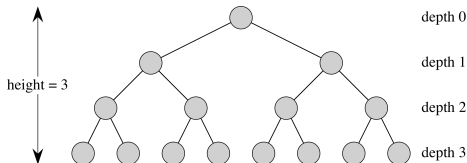
Visualisering:



Knuden  $v$  kaldes også **rod** for træet. Roden af et (ikke-tomt) undertræ af  $v$  kaldes for et **barn** af  $v$ , og  $v$  kaldes dennes **forælder**. Hvis begge  $v$ 's undertræer er tomme, kaldes  $v$  et **blad**. Stregerne mellem børn og forældre kaldes for **kanter**.

# 1) Binært træ

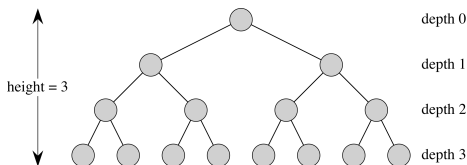
- ▶ **Dybde** af knude = antal kanter til rod
- ▶ **Højde af knude** = max antal kanter til blad
- ▶ **Højde af træ** = højde af dets rod
- ▶ **Fuldt** (complete) binært træ = træ med alle blade i samme dybde.





# 1) Binært træ

- ▶ **Dybde** af knude = antal kanter til rod
- ▶ **Højde af knude** = max antal kanter til blad
- ▶ **Højde af træ** = højde af dets rod
- ▶ **Fuldt** (complete) binært træ = træ med alle blade i samme dybde.



Et fuldt binært træ af højde  $h$  har

$$1 + 2 + 4 + 8 + \dots + 2^h = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

knuder (formel A.5 side 1147), heraf  $2^h$  blade.

## 2) Heaporden

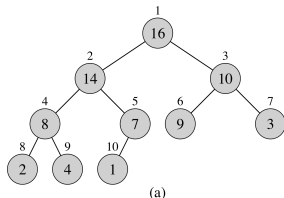
Et binært træ med nøgler i alle knuder er max-**heapordnet** hvis det for alle par af knuder  $v$  og  $u$ , hvor  $v$  er forældre til  $u$ , gælder

$$\text{nøgle i } v \geq \text{nøgle i } u$$

## 2) Heaporden

Et binært træ med nøgler i alle knuder er max-**heapordnet** hvis det for alle par af knuder  $v$  og  $u$ , hvor  $v$  er forældre til  $u$ , gælder

$$\text{nøgle i } v \geq \text{nøgle i } u$$

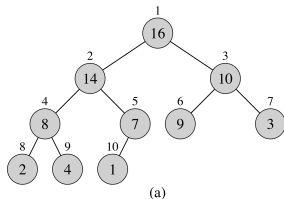


NB: dubletter er tilladt (ikke vist).

## 2) Heaporden

Et binært træ med nøgler i alle knuder er max-**heapordnet** hvis det for alle par af knuder  $v$  og  $u$ , hvor  $v$  er forældre til  $u$ , gælder

$$\text{nøgle i } v \geq \text{nøgle i } u$$



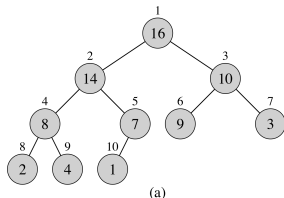
NB: dubletter er tilladt (ikke vist).

Specielt gælder at roden indeholder den største nøgle i hele heapen.

## 2) Heaporden

Et binært træ med nøgler i alle knuder er max-**heapordnet** hvis det for alle par af knuder  $v$  og  $u$ , hvor  $v$  er forældre til  $u$ , gælder

$$\text{nøgle i } v \geq \text{nøgle i } u$$



NB: dubletter er tilladt (ikke vist).

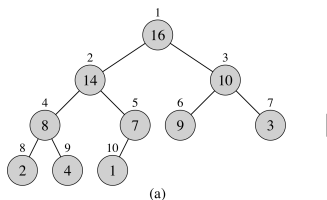
Specielt gælder at roden indeholder den største nøgle i hele heapen.

Det er min-**heapordnet** hvis der gælder

$$\text{nøgle i } v \leq \text{nøgle i } u$$

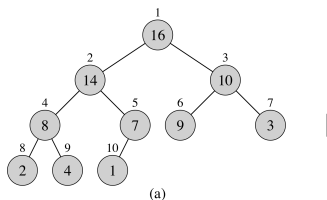
### 3) Heapfacon

Et binært træ har heapfacon hvis alle lag i træet er helt fyldte, undtagen det sidste lag, hvor alle knuder findes længst til venstre. (Specielt har et fuldt træ heapfacon).



### 3) Heapfacon

Et binært træ har heapfacon hvis alle lag i træet er helt fyldte, undtagen det sidste lag, hvor alle knuder findes længst til venstre. (Specielt har et fuldt træ heapfacon).

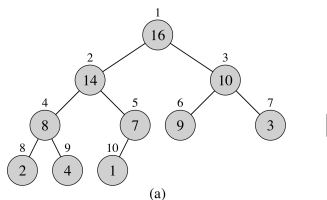


For et træ af heapfacon af højde  $h$  med  $n$  knuder:

$$n > \text{antal knuder i fuldt træ af højde } h - 1 = 2^h - 1$$

### 3) Heapfacon

Et binært træ har heapfacon hvis alle lag i træet er helt fyldte, undtagen det sidste lag, hvor alle knuder findes længst til venstre. (Specielt har et fuldt træ heapfacon).



For et træ af heapfacon af højde  $h$  med  $n$  knuder:

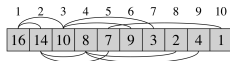
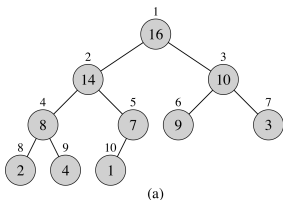
$$n > \text{antal knuder i fuldt træ af højde } h - 1 = 2^h - 1$$

$$n > 2^h - 1 \Leftrightarrow n + 1 > 2^h \Leftrightarrow \log_2(n + 1) > h$$



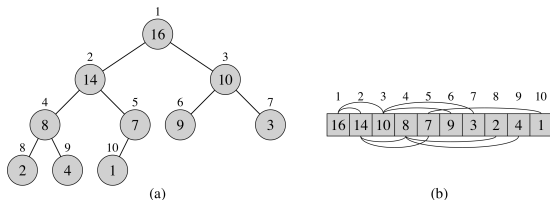
## 4) Heap udlagt i et array

Et binært træ i heapfacon kan naturligt udlægges i et array ved at tildele array-indeksler til knuder ved et top-down, venstre-til-højre gennemløb af træets lag:



## 4) Heap udlagt i et array

Et binært træ i heapfacon kan naturligt udlægges i et array ved at tildele array-indeksler til knuder ved et top-down, venstre-til-højre gennemløb af træets lag:



Navigation mellem børn og forældre i array-versionen kan udføres ved simple beregninger: Knuden på plads  $i$  har

- ▶ Forælder på plads  $\lfloor i/2 \rfloor$
- ▶ Børn på plads  $2i$  og  $2i + 1$

(Se figur ovenfor. Formelt bevis til eksaminatorier.)

# Operationer på en heap

Vi ønsker at lave følgende operationer på en heap:

- ▶ **MAX-HEAPIFY**: Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens undertræ til at overholde heap-orden.
- ▶ **BUILD-MAX-HEAP**: Lav  $n$  input elementer (uordnede) til en heap.

[Navnene ovenfor er for en max-heap. For en min-heap findes de samme operationer med “min-” i stedet for “max-” i navnet.]

## Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- ▶ Problem: knudens nøgle kan være mindre end en af sine børns nøgler.

## Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- ▶ Problem: knudens nøgle kan være mindre end en af sine børns nøgler.
- ▶ Løsning:

## Max-Heapify

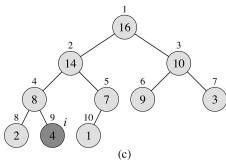
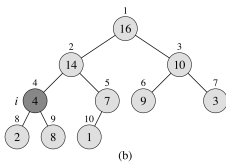
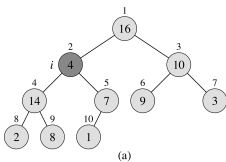
Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- ▶ Problem: knudens nøgle kan være mindre end en af sine børns nøgler.
- ▶ Løsning: byt nøgle med barnet med den største nøgle, kør derefter MAX-HEAPIFY på dette barn.

# Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

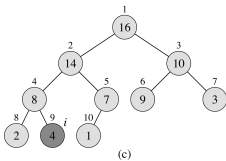
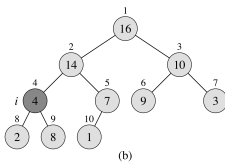
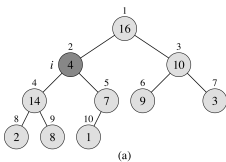
- ▶ Problem: knudens nøgle kan være mindre end en af sine børns nøgler.
- ▶ Løsning: byt nøgle med barnet med den største nøgle, kød derefter MAX-HEAPIFY på dette barn.



# Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- ▶ Problem: knudens nøgle kan være mindre end en af sine børns nøgler.
- ▶ Løsning: byt nøgle med barnet med den største nøgle, kød derefter MAX-HEAPIFY på dette barn.



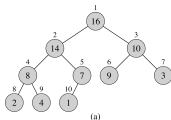
Tid:  $O(\text{højde af knude})$ .



# Max-Heapify

Som pseudo-kode (med indarbejdet check for at man ikke kigger “for langt” i arrayet, dvs. længere end plads  $n$ ):

```
MAX-HEAPIFY( $A, i, n$ )  
   $l = \text{LEFT}(i)$   
   $r = \text{RIGHT}(i)$   
  if  $l \leq n$  and  $A[l] > A[i]$   
     $largest = l$   
  else  $largest = i$   
  if  $r \leq n$  and  $A[r] > A[largest]$   
     $largest = r$   
  if  $largest \neq i$   
    exchange  $A[i]$  with  $A[largest]$   
    MAX-HEAPIFY( $A, largest, n$ )
```



# Build-Heap

Lav  $n$  input elementer (uordnede) til en heap.

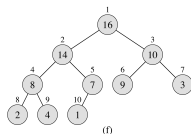
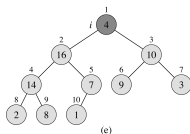
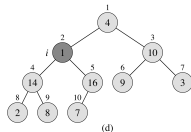
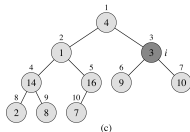
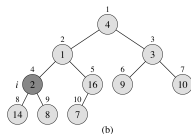
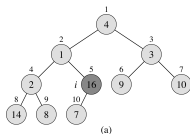
- ▶ Ide: arranger elementerne i heap-facon, bring derefter træet i heap-orden nedefra og op.
- ▶ Observation: et træ af størrelse  $n$  overholder altid heaporden.

# Build-Heap

Lav  $n$  input elementer (uordnede) til en heap.

- ▶ Ide: arranger elementerne i heap-facon, bring derefter træet i heap-orden nedefra og op.
- ▶ Observation: et træ af størrelse én overholder altid heaporden.

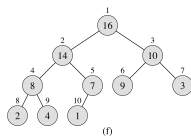
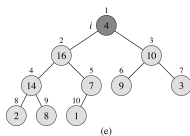
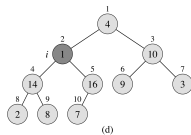
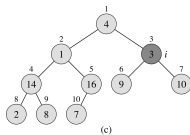
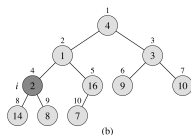
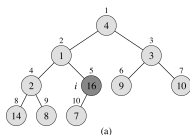
A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



# Build-Heap

A 

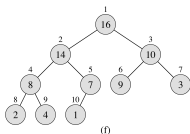
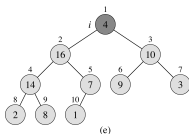
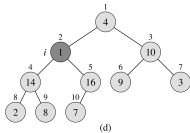
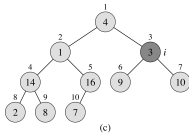
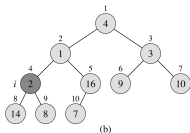
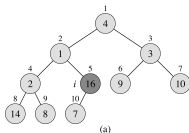
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Tid:

# Build-Heap

A 4 1 3 2 16 9 10 14 8 7



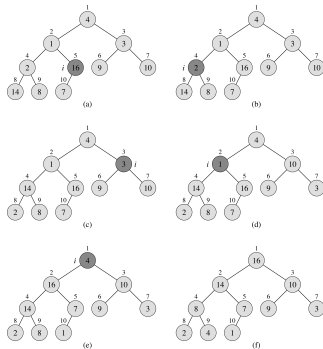
Tid:  $O(n \log_2 n)$  klart. Bedre analyse giver  $O(n)$ .

# Build-Heap

Som pseudo-kode:

**BUILD-MAX-HEAP**( $A, n$ )  
**for**  $i = \lfloor n/2 \rfloor$  **downto** 1  
  **MAX-HEAPIFY**( $A, i, n$ )

A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



# Heapsort

# Heapsort

En form for selectionsort hvor der bruges en heap til hele tiden at udtage det største tilbageværende element:

byg en heap

gentag til heap er tom:

udtag rod (største element i heapen)

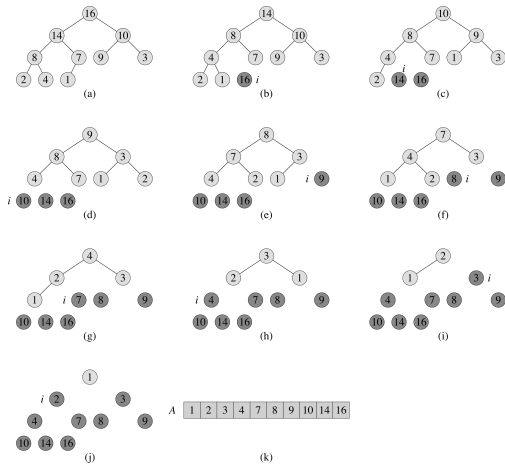
sæt sidste element op som ny rod

genskab heap-struktur ved MAX-HEAPIFY på ny rod.



# Heapsort

Eksempel:



# Heapsort

Som pseudo-kode:

```
HEAPSORT( $A, n$ )  
  BUILD-MAX-HEAP( $A, n$ )  
  for  $i = n$  downto 2  
    exchange  $A[1]$  with  $A[i]$   
    MAX-HEAPIFY( $A, 1, i - 1$ )
```

# Heapsort

Som pseudo-kode:

```
HEAPSORT( $A, n$ )  
  BUILD-MAX-HEAP( $A, n$ )  
  for  $i = n$  downto 2  
    exchange  $A[1]$  with  $A[i]$   
    MAX-HEAPIFY( $A, 1, i - 1$ )
```

Tid:  $O(n) + O(n \log n) = O(n \log n)$

## Tre $n \log n$ sorteringsalgoritmer

	Worstcase	Inplace
QuickSort		✓
MergeSort	✓	
HeapSort	✓	✓

Heapsort kører dog langsommere end Mergesort og Quicksort pga. ineffektiv brug af hukommelse (random access).

**Introsort** [Musser, 1997]: brug Quicksort, men skift under rekursionen til heapsort hvis rekursionen bliver for dyb. Dette giver en inplace, worst case  $O(n \log n)$  algoritme, med god køretid i praksis (dette er sorteringsalgoritmen i standardbiblioteket STL for C++).