

5. Sortieren

Einfache Sortierverfahren, Quicksort, Mergesort

Problemstellung

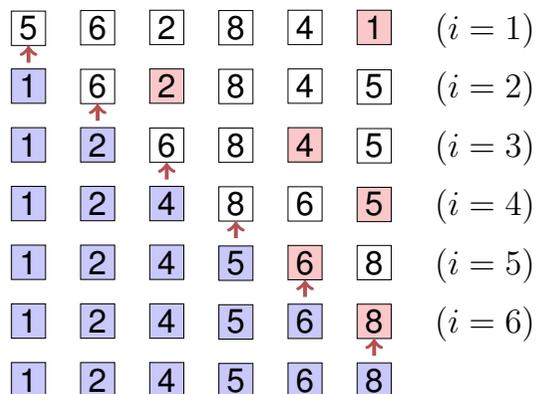
Eingabe: Ein Array $A = (A[1], \dots, A[n])$ der Länge n .

Ausgabe: Eine Permutation A' von A , die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

98

99

Sortieren durch Auswahl



- Iteratives Vorgehen wie bei Bubblesort.
- Auswahl des kleinsten (oder grössten) Elementes durch direkte Suche.

Algorithmus: Sortieren durch Auswahl

Input : Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output : Sortiertes Array A

```
for  $i \leftarrow 1$  to  $n - 1$  do
   $p \leftarrow i$ 
  for  $j \leftarrow i + 1$  to  $n$  do
    if  $A[j] < A[p]$  then
       $p \leftarrow j$ 
  swap( $A[i], A[p]$ )
```

100

101

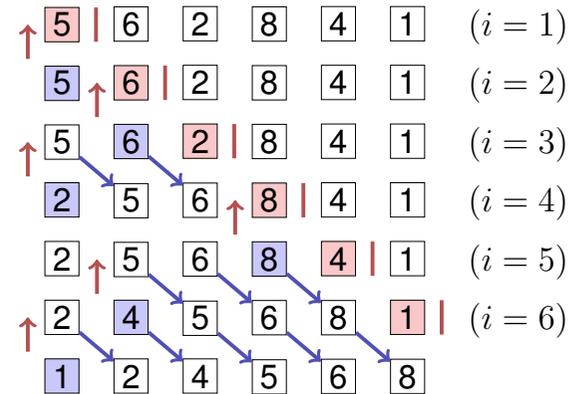
Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

Anzahl Vergleiche im besten Fall: $\Theta(n^2)$.

Sortieren durch Einfügen



- Iteratives Vorgehen:
 $i = 1 \dots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

102

103

Sortieren durch Einfügen

❓ Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

⚠ Im schlechtesten Fall viele Elementverschiebungen.

❓ Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

⚠ Der Suchbereich (Einfügebereich) ist bereits sortiert.
Konsequenz: binäre Suche möglich.

Algorithmus: Sortieren durch Einfügen

Input : Array $A = (A[1], \dots, A[n])$, $n \geq 0$.

Output : Sortiertes Array A

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

$x \leftarrow A[i]$

$p \leftarrow \text{BinarySearch}(A[1..i-1], x)$; // Kleinstes $p \in [1, i]$ mit $A[p] \geq x$

for $j \leftarrow i - 1$ **downto** p **do**

$A[j + 1] \leftarrow A[j]$

$A[p] \leftarrow x$

104

105

Analyse

Anzahl Vergleiche im schlechtesten Fall:

$$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n).$$

Anzahl Vergleiche im besten Fall: $\Theta(n \log n)$.³

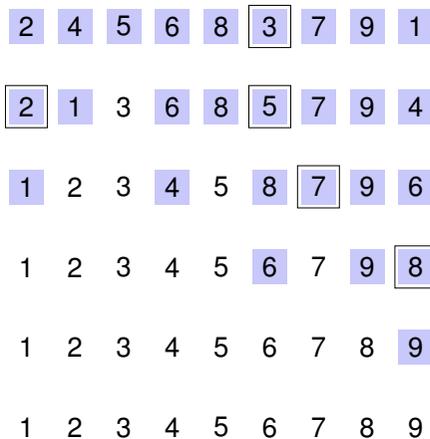
Anzahl Vertauschungen im schlechtesten Fall: $\sum_{k=2}^n (k-1) \in \Theta(n^2)$

³Mit leichter Anpassung der Funktion BinarySearch für das Minimum / Maximum: $\Theta(n)$

5.1 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Quicksort (willkürlicher Pivot)



Algorithmus Quicksort($A[l, \dots, r]$)

Input : Array A der Länge n . $1 \leq l \leq r \leq n$.

Output : Array A , sortiert zwischen l und r .

if $l < r$ **then**

 Wähle Pivot $p \in A[l, \dots, r]$
 $k \leftarrow \text{Partition}(A[l, \dots, r], p)$
 Quicksort($A[l, \dots, k-1]$)
 Quicksort($A[k+1, \dots, r]$)

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

$$T(n) = 2T(n/2) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \mathcal{O}(n \log n)$$

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n-1) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \Theta(n^2)$$

110

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n - 1^4$. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

Kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert $\mathcal{O}(\log n)$ Rekursionstiefe und Speicherplatzbedarf.

⁴Stack-Overflow möglich!

112

Analyse (Randomisiertes Quicksort)

Theorem

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.

111

Praktische Anmerkungen

Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.

113

Mergesort (Sortieren durch Verschmelzen)

5.2 Mergesort

[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

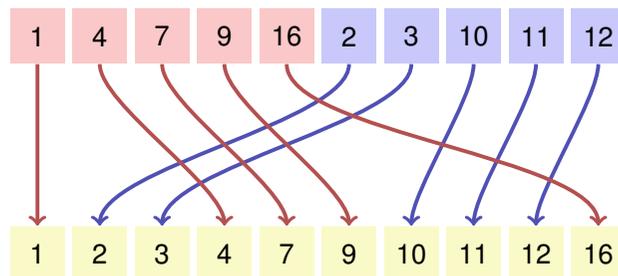
Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Sortierung des so vorsortierten A in $\mathcal{O}(n)$.

114

115

Merge



116

Algorithmus Merge(A, l, m, r)

Input : Array A der Länge n , Indizes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m]$,
 $A[m + 1, \dots, r]$ sortiert

Output : $A[l, \dots, r]$ sortiert

```
1  $B \leftarrow \text{new Array}(r - l + 1)$ 
2  $i \leftarrow l; j \leftarrow m + 1; k \leftarrow 1$ 
3 while  $i \leq m$  and  $j \leq r$  do
4   if  $A[i] \leq A[j]$  then  $B[k] \leftarrow A[i]; i \leftarrow i + 1$ 
5   else  $B[k] \leftarrow A[j]; j \leftarrow j + 1$ 
6    $k \leftarrow k + 1;$ 
7 while  $i \leq m$  do  $B[k] \leftarrow A[i]; i \leftarrow i + 1; k \leftarrow k + 1$ 
8 while  $j \leq r$  do  $B[k] \leftarrow A[j]; j \leftarrow j + 1; k \leftarrow k + 1$ 
9 for  $k \leftarrow l$  to  $r$  do  $A[k] \leftarrow B[k - l + 1]$ 
```

117

Korrektheit

Hypothese: Nach k Durchläufen der Schleife von Zeile 3 ist $B[1, \dots, k]$ sortiert und $B[k] \leq A[i]$, falls $i \leq m$ und $B[k] \leq A[j]$ falls $j \leq r$.

Beweis per Induktion:

Induktionsanfang: Das leere Array $B[1, \dots, 0]$ ist trivialerweise sortiert.

Induktionsschluss ($k \rightarrow k + 1$):

- oBdA $A[i] \leq A[j]$, $i \leq m$, $j \leq r$.
- $B[1, \dots, k]$ ist nach Hypothese sortiert und $B[k] \leq A[i]$.
- Nach $B[k + 1] \leftarrow A[i]$ ist $B[1, \dots, k + 1]$ sortiert.
- $B[k + 1] = A[i] \leq A[i + 1]$ (falls $i + 1 \leq m$) und $B[k + 1] \leq A[j]$ falls $j \leq r$.
- $k \leftarrow k + 1$, $i \leftarrow i + 1$: Aussage gilt erneut.

118

Analyse (Merge)

Lemma

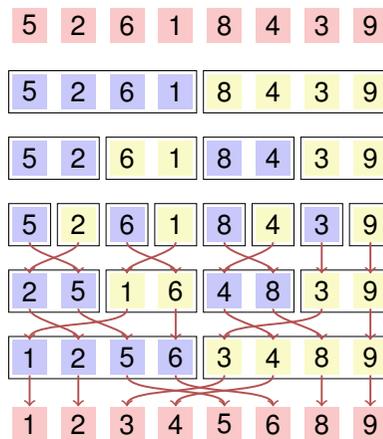
Wenn: Array A der Länge n , Indizes $1 \leq l < r \leq n$. $m = \lfloor (l + r)/2 \rfloor$ und $A[l, \dots, m]$, $A[m + 1, \dots, r]$ sortiert.

Dann: im Aufruf $\text{Merge}(A, l, m, r)$ werden $\Theta(r - l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

Beweis: (Inspektion des Algorithmus und Zählen der Operationen).

119

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithmus Rekursives 2-Wege Mergesort(A, l, r)

Input : Array A der Länge n . $1 \leq l \leq r \leq n$

Output : Array $A[l, \dots, r]$ sortiert.

if $l < r$ **then**

```

     $m \leftarrow \lfloor (l + r)/2 \rfloor$            // Mittlere Position
    Mergesort( $A, l, m$ )                // Sortiere vordere Hälfte
    Mergesort( $A, m + 1, r$ )           // Sortiere hintere Hälfte
    Merge( $A, l, m, r$ )                // Verschmelzen der Teilfolgen
  
```

120

121

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$C(n) = C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

Algorithmus StraightMergesort(A)

Rekursion vermeiden: Verschmelze Folgen der Länge 1, 2, 4... direkt

Input : Array A der Länge n

Output : Array A sortiert

$length \leftarrow 1$

while $length < n$ **do** // Iteriere über die Längen n

$r \leftarrow 0$

while $r + length < n$ **do** // Iteriere über die Teilfolgen

$l \leftarrow r + 1$

$m \leftarrow l + length - 1$

$r \leftarrow \min(m + length, n)$

 Merge(A, l, m, r)

$length \leftarrow length \cdot 2$

122

123

Analyse

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus.

Natürliches 2-Wege Mergesort

Beobachtung: Obige Varianten nutzen nicht aus, wenn vorsortiert ist und führen immer $\Theta(n \log n)$ viele Bewegungen aus.

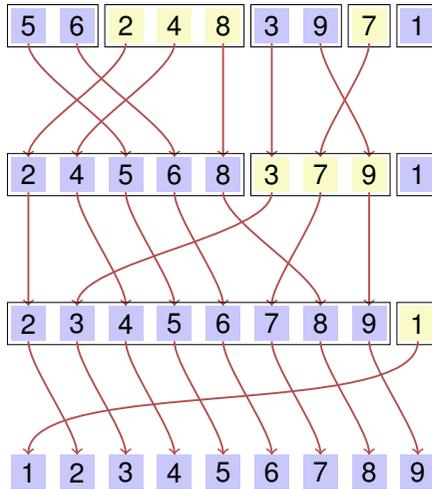
❓ Wie kann man teilweise vorsortierte Folgen besser sortieren?

⚠️ Rekursives Verschmelzen von bereits vorsortierten Teilen (*Runs*) von A .

124

125

Natürliches 2-Wege Mergesort



126

Algorithmus NaturalMergesort(A)

Input : Array A der Länge $n > 0$

Output : Array A sortiert

repeat

$r \leftarrow 0$

while $r < n$ **do**

$l \leftarrow r + 1$

$m \leftarrow l$; **while** $m < n$ **and** $A[m + 1] \geq A[m]$ **do** $m \leftarrow m + 1$

if $m < n$ **then**

$r \leftarrow m + 1$; **while** $r < n$ **and** $A[r + 1] \geq A[r]$ **do** $r \leftarrow r + 1$

 Merge(A, l, m, r);

else

$r \leftarrow n$

until $l = 1$

127

Analyse

Im besten Fall führt natürliches Mergesort nur $n - 1$ Vergleiche durch!

Im schlechtesten Fall und im Durchschnitt führt natürliches Mergesort $\Theta(n \log n)$ viele Vergleiche und Bewegungen aus.

128