5. Sortieren

Einfache Sortierverfahren, Quicksort, Mergesort

Problemstellung

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

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

Sortieren durch Auswahl

$\boxed{5}$ $\boxed{6}$ $\boxed{2}$ $\boxed{8}$ $\boxed{4}$ $\boxed{1}$ $(i = 1)$

- Iteratives Vorgehen wie bei Bubblesort.

Sortieren durch Auswahl

$\boxed{5}$ $\boxed{6}$ $\boxed{2}$ $\boxed{8}$ $\boxed{4}$ $\boxed{1}$ $(i = 1)$

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

Sortieren durch Auswahl

$\boxed{5}\ \boxed{6}\ \boxed{2}\ \boxed{8}\ \boxed{4}\ \boxed{1} \quad (i = 1)$

$\boxed{1}\ \boxed{6}\ \boxed{2}\ \boxed{8}\ \boxed{4}\ \boxed{5} \quad (i = 2)$

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

Sortieren durch Auswahl

$\boxed{5}\quad\boxed{6}\quad\boxed{2}\quad\boxed{8}\quad\boxed{4}\quad\boxed{1}\qquad(i=1)$

$\boxed{1}\quad\boxed{6}\quad\boxed{2}\quad\boxed{8}\quad\boxed{4}\quad\boxed{5}\qquad(i=2)$

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

Sortieren durch Auswahl

$\boxed{5}\ \boxed{6}\ \boxed{2}\ \boxed{8}\ \boxed{4}\ \boxed{1}\quad (i = 1)$

$\boxed{1}\ \boxed{6}\ \boxed{2}\ \boxed{8}\ \boxed{4}\ \boxed{5}\quad (i = 2)$

$\boxed{1}\ \boxed{2}\ \boxed{6}\ \boxed{8}\ \boxed{4}\ \boxed{5}\quad (i = 3)$

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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$

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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$

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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$

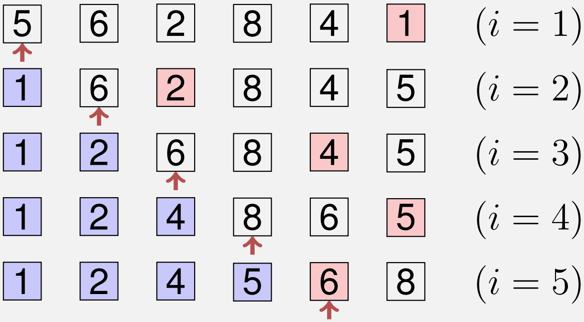
- Iteratives Vorgehen wie bei Bubblesort.
- Auswahl des kleinsten (oder grössten) Elementes durch direkte Suche.

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$
1	2	4	5	6	8	$(i = 5)$

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

Sortieren durch Auswahl



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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$
1	2	4	5	6	8	$(i = 5)$
1	2	4	5	6	8	$(i = 6)$

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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$
1	2	4	5	6	8	$(i = 5)$
1	2	4	5	6	8	$(i = 6)$

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

Sortieren durch Auswahl

5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$
1	2	4	8	6	5	$(i = 4)$
1	2	4	5	6	8	$(i = 5)$
1	2	4	5	6	8	$(i = 6)$
1	2	4	5	6	8	

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

Algorithmus: Sortieren durch Auswahl

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

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

Analyse

Anzahl Vergleiche im schlechtesten Fall:

Analyse

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

Anzahl Vertauschungen im schlechtesten Fall:

Analyse

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

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

Anzahl Vergleiche im besten Fall:

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

$\boxed{5}$ | $\boxed{6}$ $\boxed{2}$ $\boxed{8}$ $\boxed{4}$ $\boxed{1}$ $(i = 1)$

Sortieren durch Einfügen

$\uparrow \boxed{5} \mid \boxed{6} \quad \boxed{2} \quad \boxed{8} \quad \boxed{4} \quad \boxed{1} \qquad (i = 1)$

- Iteratives Vorgehen: $i = 1...n$

Sortieren durch Einfügen

$$\boxed{5} \mid \boxed{6} \quad \boxed{2} \quad \boxed{8} \quad \boxed{4} \quad \boxed{1} \qquad (i = 1)$$

$$\boxed{5} \quad \boxed{6} \mid \boxed{2} \quad \boxed{8} \quad \boxed{4} \quad \boxed{1} \qquad (i = 2)$$

- Iterative Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.

Sortieren durch Einfügen

$\boxed{5}$ | $\boxed{6}$ $\boxed{2}$ $\boxed{8}$ $\boxed{4}$ $\boxed{1}$ $(i = 1)$

$\boxed{5}$ $\boxed{6}$ | $\boxed{2}$ $\boxed{8}$ $\boxed{4}$ $\boxed{1}$ $(i = 2)$

- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen,

Sortieren durch Einfügen

5 | 6 2 8 4 1 $(i = 1)$

5 6 | 2 8 4 1 $(i = 2)$

5 6 2 | 8 4 1 $(i = 3)$

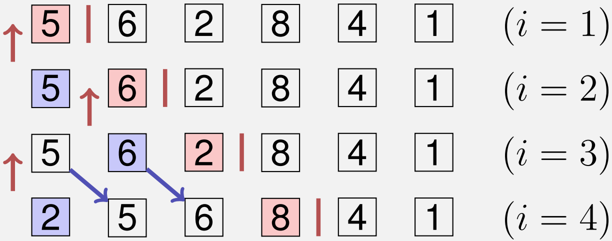
- Iterative Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen,

Sortieren durch Einfügen

5	6	2	8	4	1	$(i = 1)$
5	6	2	8	4	1	$(i = 2)$
5	6	2	8	4	1	$(i = 3)$

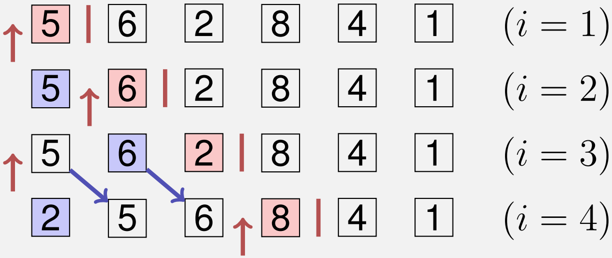
- Iterative Vorgehen:
  $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen,

Sortieren durch Einfügen



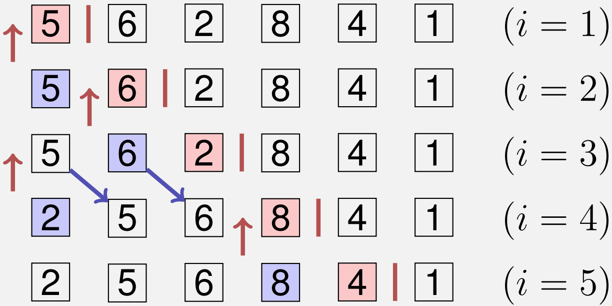
- Iteratives Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen



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

Sortieren durch Einfügen



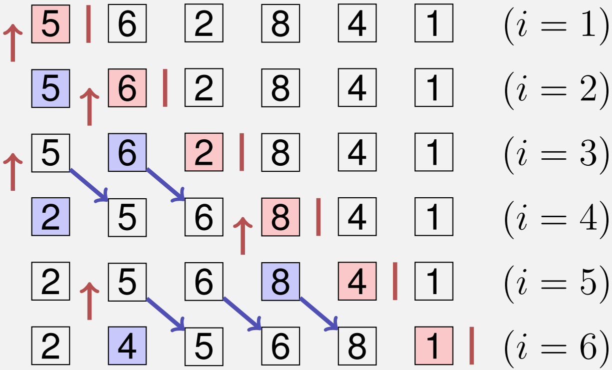
- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

5	6	2	8	4	1	$(i = 1)$
5	6	2	8	4	1	$(i = 2)$
5	6	2	8	4	1	$(i = 3)$
2	5	6	8	4	1	$(i = 4)$
2	5	6	8	4	1	$(i = 5)$

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

Sortieren durch Einfügen

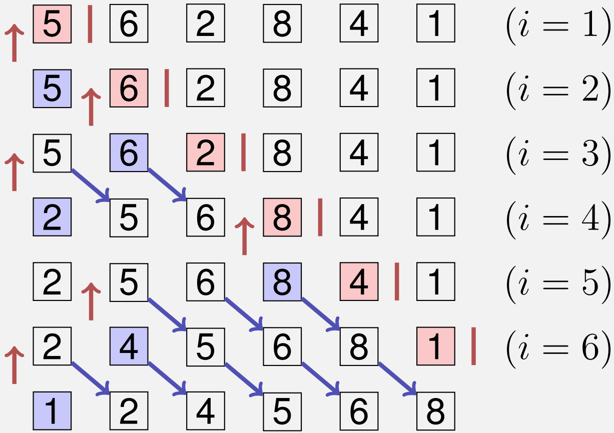


- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

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

Sortieren durch Einfügen



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

Sortieren durch Einfügen

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

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?

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**
  $x \leftarrow A[i]$
  $p \leftarrow$ BinarySearch($A[1...i-1], x$); // Kleinstes $p \in [1, i]$ mit $A[p] \geq x$
  for $j \leftarrow i-1$ **downto** p **do**
    $A[j+1] \leftarrow A[j]$
  $A[p] \leftarrow x$

Analyse

Anzahl Vergleiche im schlechtesten Fall:

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

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:

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

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:

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

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: $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

[3] Mit 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)

2	4	5	6	8	3	7	9	1

Quicksort (willkürlicher Pivot)

2	4	5	6	8	3	7	9	1

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
2	1	3	6	8	5	7	9	4

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	7	9	6

Quicksort (willkürlicher Pivot)

$2 \quad 4 \quad 5 \quad 6 \quad 8 \quad \boxed{3} \quad 7 \quad 9 \quad 1$

$\boxed{2} \quad 1 \quad 3 \quad 6 \quad 8 \quad \boxed{5} \quad 7 \quad 9 \quad 4$

$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 8 \quad \boxed{7} \quad 9 \quad 6$

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	8

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	$\boxed{8}$

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	$\boxed{8}$
1	2	3	4	5	6	7	8	9

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	$\boxed{8}$
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

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

Analyse: Anzahl Vergleiche

Bester Fall.

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

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

Schlechtester Fall.

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

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

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

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

Analyse (Randomisiertes Quicksort)

Theorem

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

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.

[4] Stack-Overflow möglich!

Praktische Anmerkungen

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

5.2 Mergesort

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

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

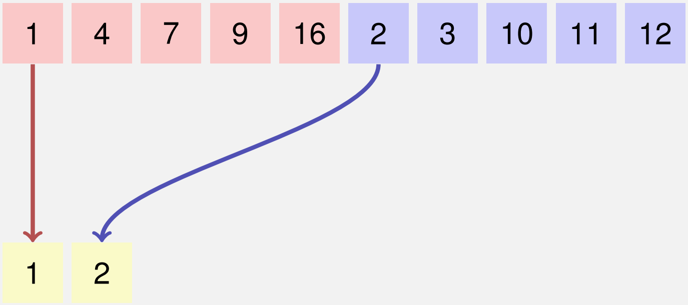
- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Sortierung des so vorsortierten A in $\mathcal{O}(n)$.

Merge

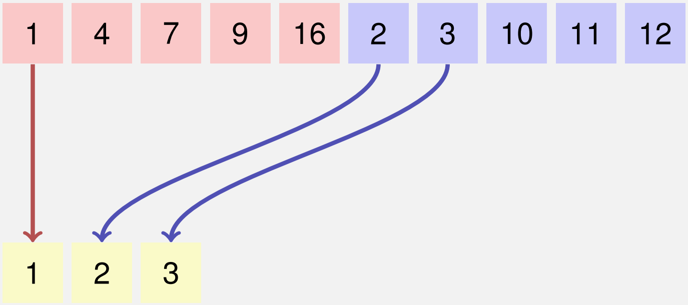
1	4	7	9	16	2	3	10	11	12

Merge

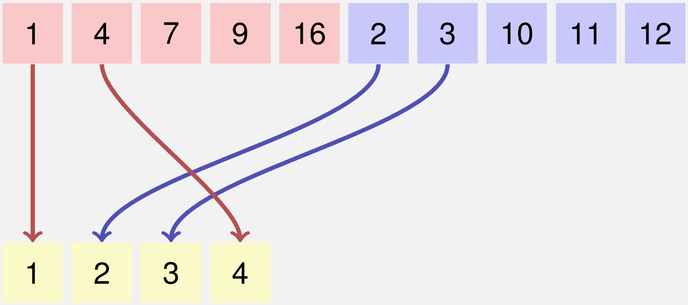
Merge



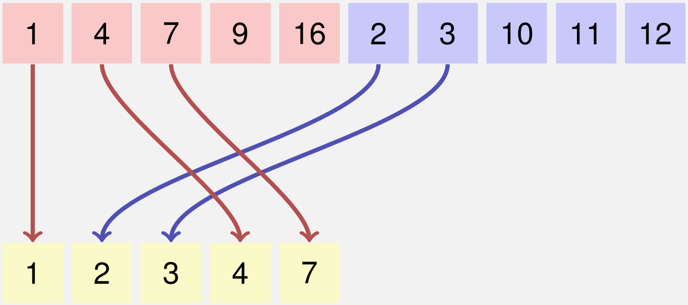
Merge



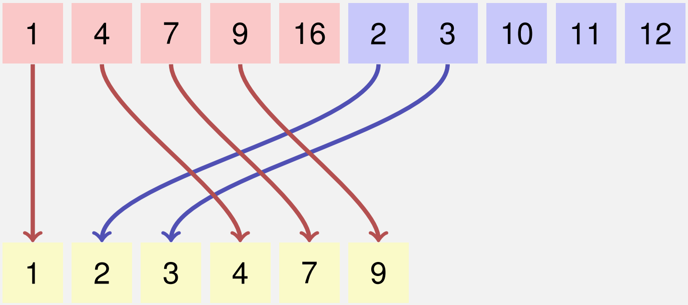
Merge



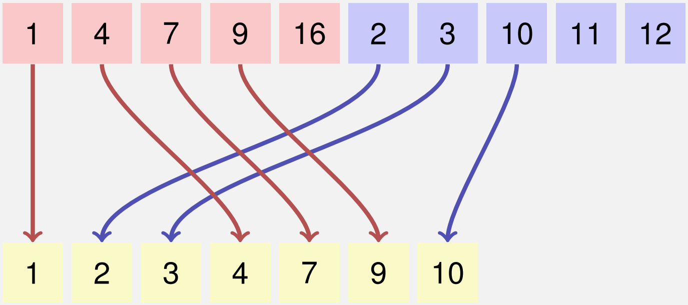
Merge



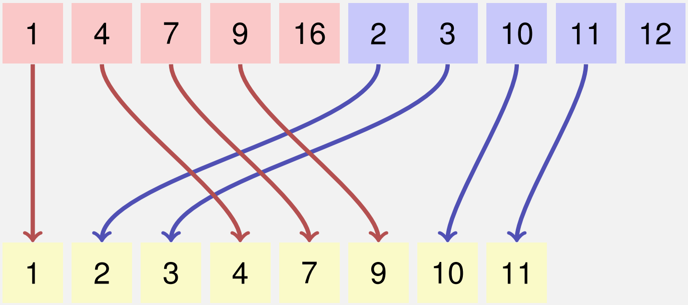
Merge



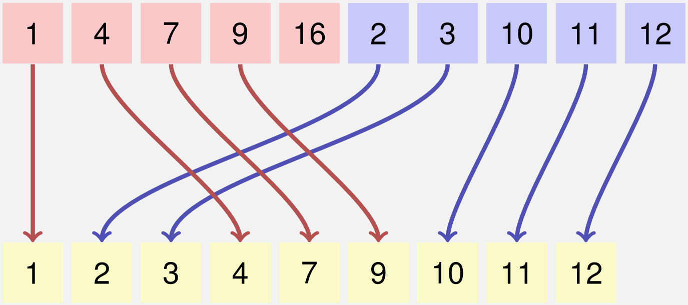
Merge



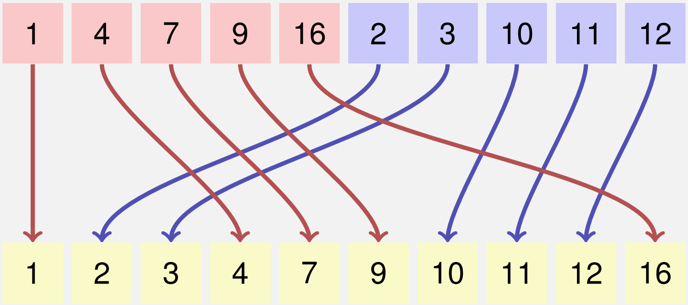
Merge



Merge



Merge



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

$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$
while $i \leq m$ and $j \leq r$ **do**
    if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$
    else $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$;
while $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

Korrektheit

Hypothese: Nach k Durchläufen der Schleife von Zeile 3 ist $B[1, \ldots, k]$ 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 $B[1, \ldots, 0]$ ist trivialerweise sortiert.
Induktionsschluss ($k \to k+1$):

- oBdA $A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1, \ldots, k]$ ist nach Hypothese sortiert und $B[k] \leq A[i]$.
- Nach $B[k+1] \leftarrow A[i]$ ist $B[1, \ldots, k+1]$ 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.

Analyse (Merge)

Lemma

Wenn: Array A der Länge n, Indizes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ und $A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sortiert.
Dann: im Aufruf Merge(A, l, m, r) werden $\Theta(r-l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

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

Mergesort

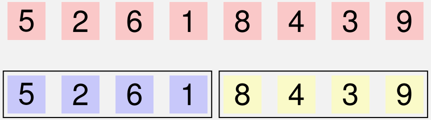
5	2	6	1	8	4	3	9

Mergesort

5	2	6	1	8	4	3	9

Split

Mergesort



Split

Mergesort

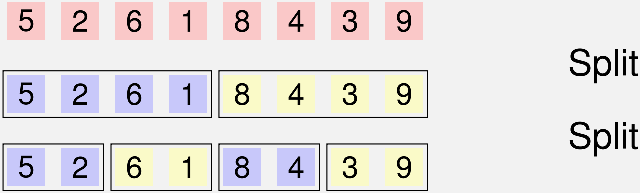
5 2 6 1 8 4 3 9

Split

5 2 6 1 | 8 4 3 9

Split

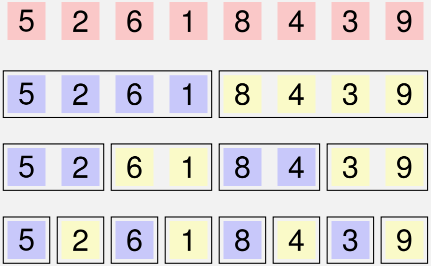
Mergesort



Mergesort

5	2	6	1	8	4	3	9	
5	2	6	1	8	4	3	9	Split
5	2	6	1	8	4	3	9	Split
								Split

Mergesort

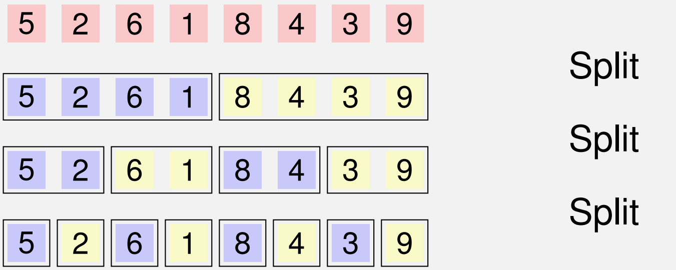


Split

Split

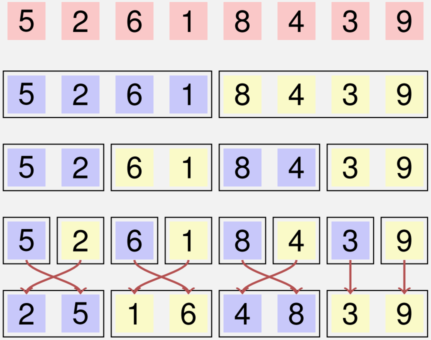
Split

Mergesort



5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
Split
Split
Split
Merge

Mergesort



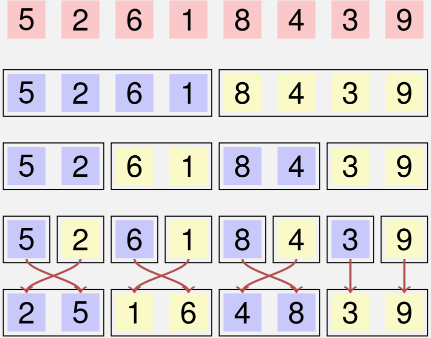
Split

Split

Split

Merge

Mergesort



Split

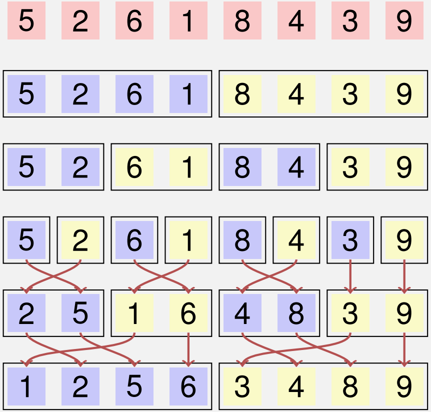
Split

Split

Merge

Merge

Mergesort



Split

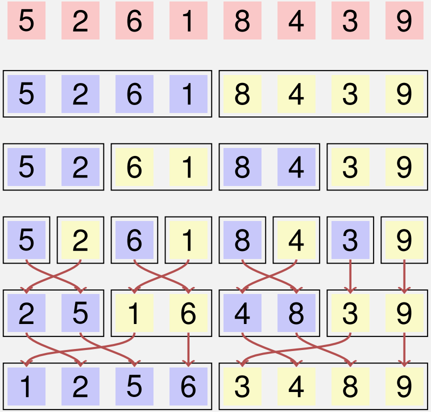
Split

Split

Merge

Merge

Mergesort



Split

Split

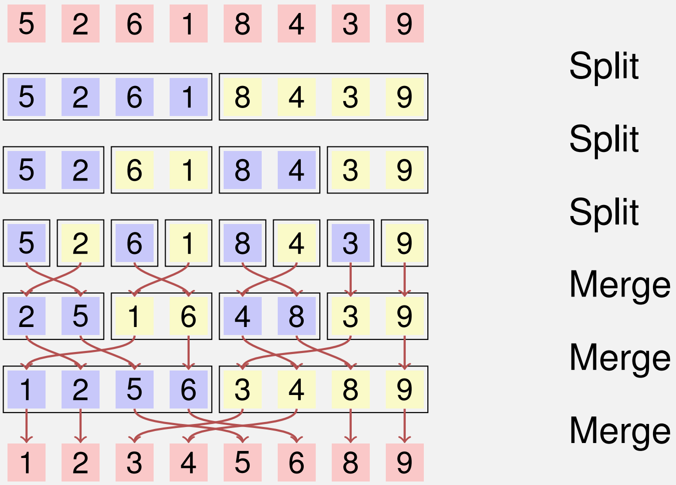
Split

Merge

Merge

Merge

Mergesort



5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
1 2 5 6 3 4 8 9
1 2 3 4 5 6 8 9
Split
Split
Split
Merge
Merge
Merge

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

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

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

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

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

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

Algorithmus StraightMergesort(A)

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

Input : Array A der Länge n
Output : Array A sortiert

```
length ← 1
while length < n do                        // Iteriere über die Längen n
    r ← 0
    while r + length < n do                // Iteriere über die Teilfolgen
        l ← r + 1
        m ← l + length − 1
        r ← min(m + length, n)
        Merge(A, l, m, r)
    length ← length · 2
```

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?

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.

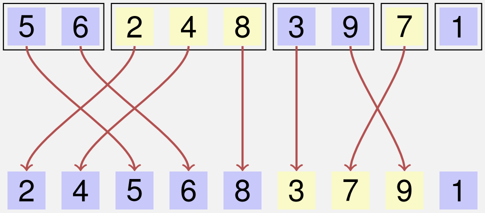
Natürliches 2-Wege Mergesort

5	6	2	4	8	3	9	7	1

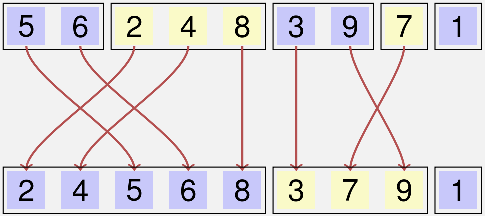
Natürliches 2-Wege Mergesort

5	6	2	4	8	3	9	7	1

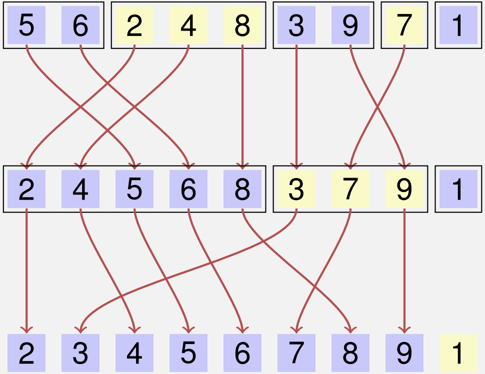
Natürliches 2-Wege Mergesort



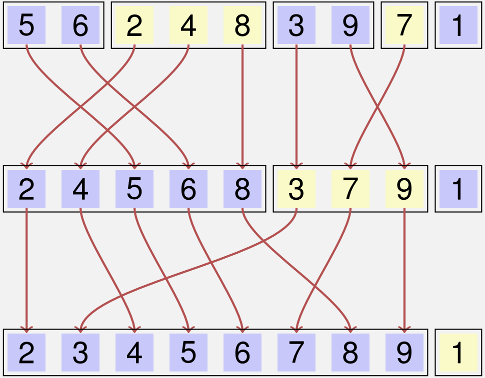
Natürliches 2-Wege Mergesort



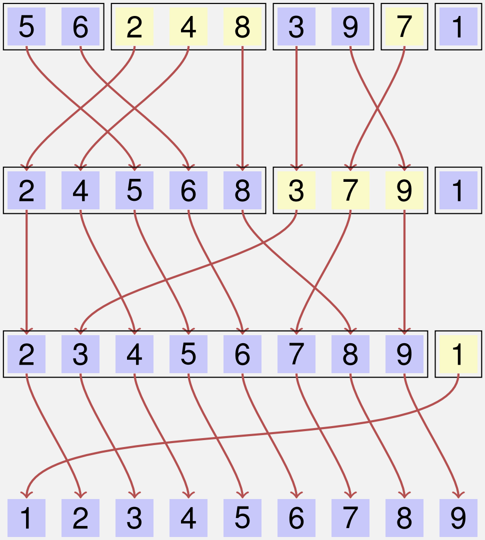
Natürliches 2-Wege Mergesort



Natürliches 2-Wege Mergesort



Natürliches 2-Wege Mergesort



Algorithmus NaturalMergesort(A)

Input : Array A der Länge $n > 0$
Output : Array A sortiert

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m + 1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r + 1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```

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.