7. Sortieren I

Einfache Sortierverfahren

7.1 Einfaches Sortieren

Sortieren durch Auswahl, Sortieren durch Einfügen, Bubblesort
[Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

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 $1 \leq i \leq j \leq n$.

Algorithmus: IsSorted(A)

Input : Array $A = (A[1], ..., A[n])$ der Länge n.
Output : Boolesche Entscheidung "sortiert" oder "nicht sortiert"

for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $A[i] > A[i + 1]$ **then**
    return "nicht sortiert";
return "sortiert";

Beobachtung

IsSorted(A):“nicht sortiert”, wenn $A[i] > A[i+1]$ für ein i.

Beobachtung

IsSorted(A):“nicht sortiert”, wenn $A[i] > A[i+1]$ für ein i.

$\Rightarrow$ Idee:

Beobachtung

IsSorted(A):“nicht sortiert”, wenn $A[i] > A[i+1]$ für ein i.

$\Rightarrow$ Idee:

for $j \leftarrow 1$ **to** $n-1$ **do**
  if $A[j] > A[j+1]$ **then**
    swap($A[j], A[j+1]$);

Ausprobieren

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

Ausprobieren

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

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

Ausprobieren

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

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

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

Ausprobieren

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

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

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

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

Ausprobieren

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

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

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

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

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

Ausprobieren

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

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

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

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

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

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

Ausprobieren

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

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

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

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

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

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

- Nicht sortiert! ☹.

Ausprobieren

$5 \leftrightarrow 6 \quad 2 \quad 8 \quad 4 \quad 1 \qquad (j = 1)$

$5 \quad 6 \leftrightarrow 2 \quad 8 \quad 4 \quad 1 \qquad (j = 2)$

$5 \quad 2 \quad 6 \leftrightarrow 8 \quad 4 \quad 1 \qquad (j = 3)$

$5 \quad 2 \quad 6 \quad 8 \leftrightarrow 4 \quad 1 \qquad (j = 4)$

$5 \quad 2 \quad 6 \quad 4 \quad 8 \leftrightarrow 1 \qquad (j = 5)$

$5 \quad 2 \quad 6 \quad 4 \quad 1 \quad 8$

- Nicht sortiert! ☹.

Ausprobieren

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

- Nicht sortiert! ☹.
- Aber das grösste Element wandert ganz nach rechts.
  $\Rightarrow$ Neue Idee! ☺

Ausprobieren

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

- Wende das Verfahren iterativ an.

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \ldots, n]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \ldots, n]$, dann $A[1, \ldots, n-1]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1,\ldots,n]$, dann $A[1,\ldots,n-1]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \ldots, n]$, dann $A[1, \ldots, n-1]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \ldots, n]$, dann $A[1, \ldots, n-1]$, dann $A[1, \ldots, n-2]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \dots, n]$, dann $A[1, \dots, n-1]$, dann $A[1, \dots, n-2]$,

Ausprobieren

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

- Wende das Verfahren iterativ an.
- Für $A[1, \ldots, n]$, dann $A[1, \ldots, n-1]$, dann $A[1, \ldots, n-2]$, etc.

Algorithmus: Bubblesort

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

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

Analyse

Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

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

? Was ist der schlechteste Fall?

Analyse

Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

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

? Was ist der schlechteste Fall?

! Wenn A absteigend sortiert ist.

Analyse

Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

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

? Was ist der schlechteste Fall?

! Wenn A absteigend sortiert ist.

? Algorithmus kann so angepasst werden, dass er dann abbricht, wenn das Array sortiert ist. Schlüsselvergleiche und Vertauschungen des modifizierten Algorithmus im besten Fall?

Analyse

Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

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

? Was ist der schlechteste Fall?

! Wenn A absteigend sortiert ist.

? Algorithmus kann so angepasst werden, dass er dann abbricht, wenn das Array sortiert ist. Schlüsselvergleiche und Vertauschungen des modifizierten Algorithmus im besten Fall?

! Schlüsselvergleiche = $n-1$. Vertauschungen = 0.

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

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

- 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

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

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

- 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

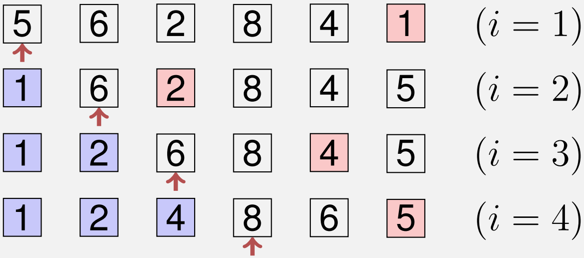
5	6	2	8	4	1	$(i = 1)$
1	6	2	8	4	5	$(i = 2)$
1	2	6	8	4	5	$(i = 3)$

- 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.



- 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

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

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	$(i = 6)$

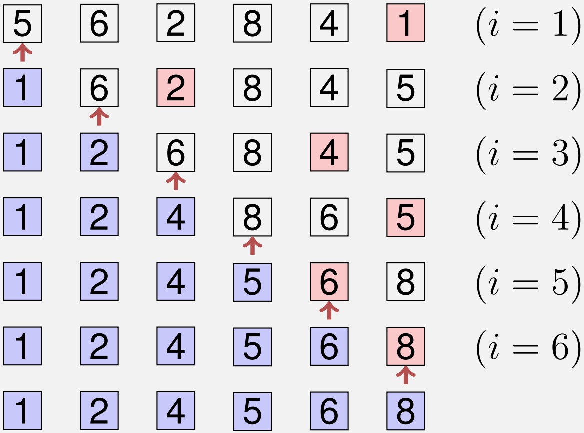
- Iteratives Vorgehen wie bei Bubblesort.
- Auswahl des kleinsten (oder grössten) Elementes durch direkte Suche.

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



- 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 ← 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

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

Sortieren durch Einfügen

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

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

Sortieren durch Einfügen

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

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

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

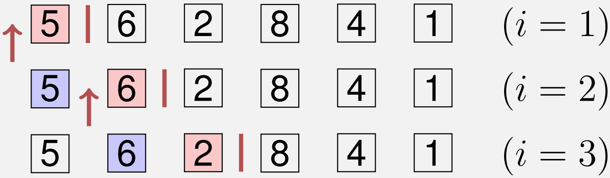
Sortieren durch Einfügen

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

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

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

Sortieren durch Einfügen



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

Sortieren durch Einfügen

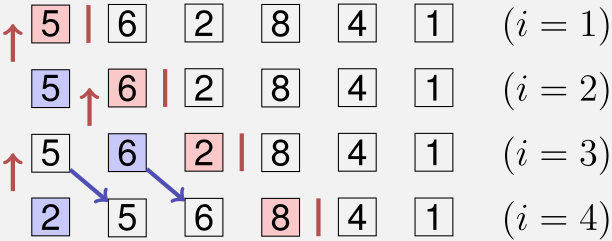
$\uparrow \boxed{5} \mid \boxed{6}\ \boxed{2}\ \boxed{8}\ \boxed{4}\ \boxed{1} \quad (i = 1)$

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

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

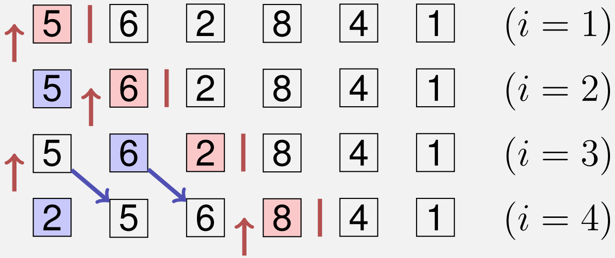
- Iteratives Vorgehen: $i = 1 \dots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen,

Sortieren durch Einfügen



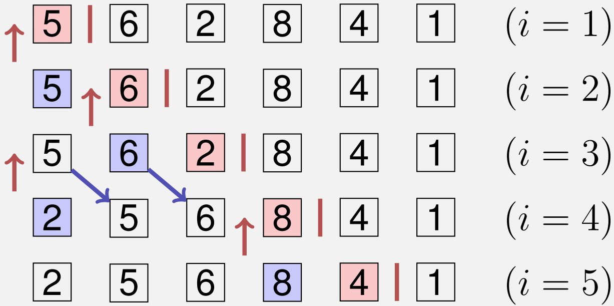
- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen



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

Sortieren durch Einfügen



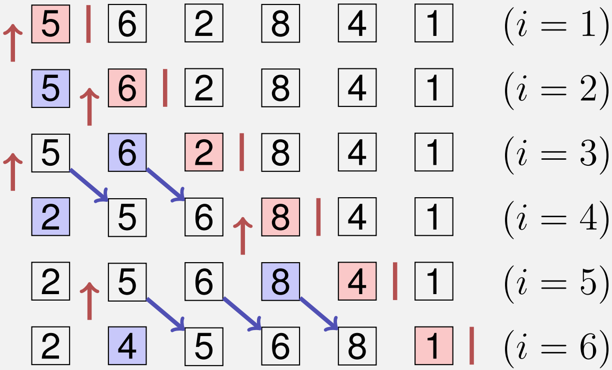
- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

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

Sortieren durch Einfügen

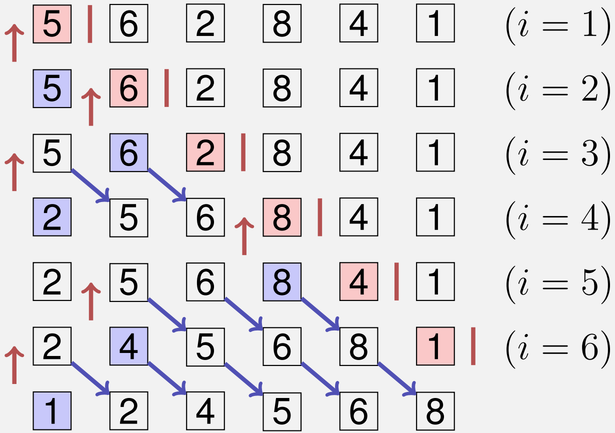


- Iteratives Vorgehen: $i = 1 \ldots n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

- Iteratives Vorgehen: $i = 1 \dots n$
- 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

? 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:

[4]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:

[4] 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: $\Theta(n \log n)$.[4]

Anzahl Vertauschungen im schlechtesten Fall:

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

Analyse

Anzahl Vergleiche im schlechtesten Fall:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

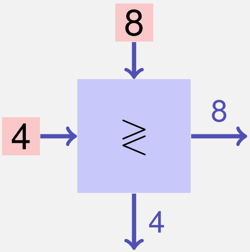
Anzahl Vergleiche im besten Fall: $\Theta(n \log n)$.[4]

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

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

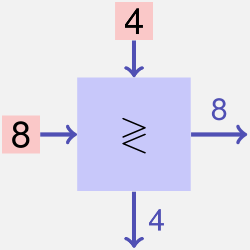
Anderer Blickwinkel

Sortierknoten:



Anderer Blickwinkel

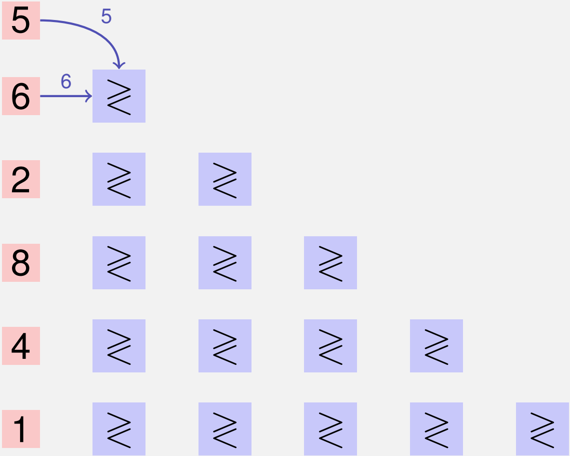
Sortierknoten:



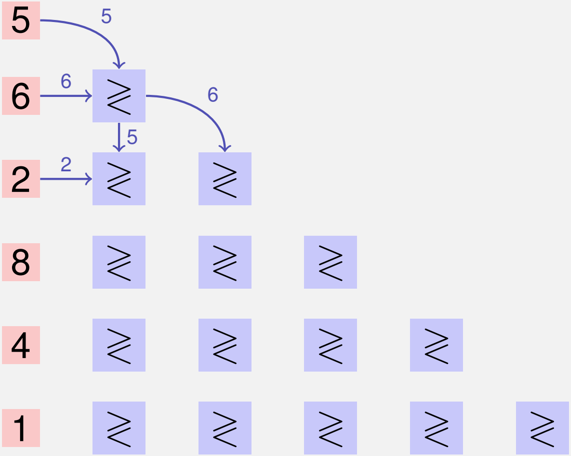
Anderer Blickwinkel

5					
6	$\gtrless$				
2	$\gtrless$	$\gtrless$			
8	$\gtrless$	$\gtrless$	$\gtrless$		
4	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	
1	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$

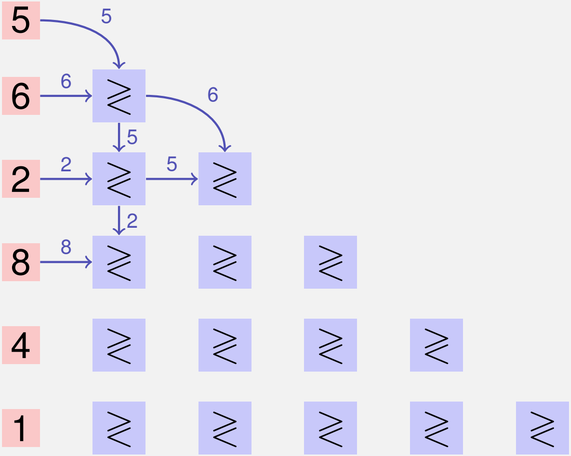
Anderer Blickwinkel



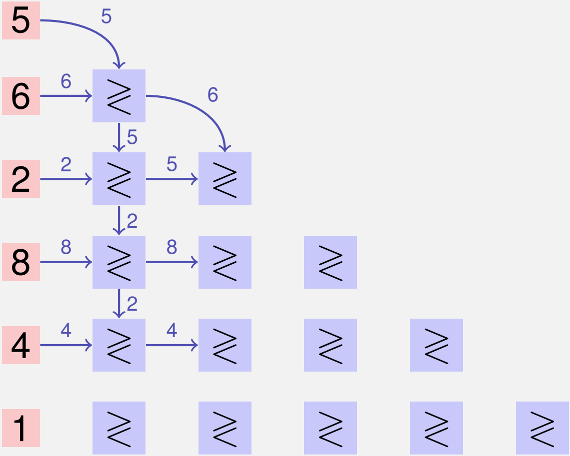
Anderer Blickwinkel



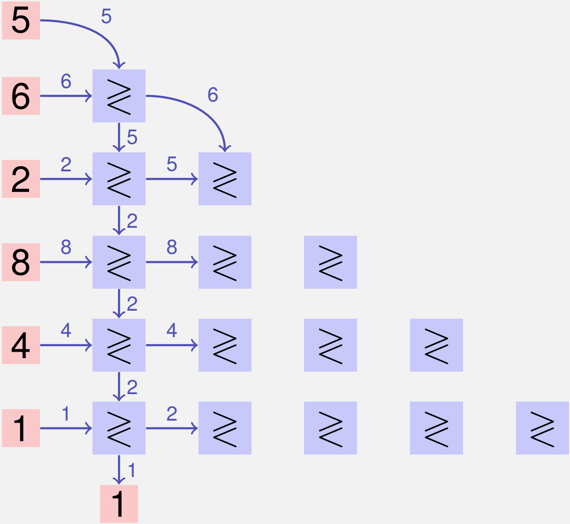
Anderer Blickwinkel



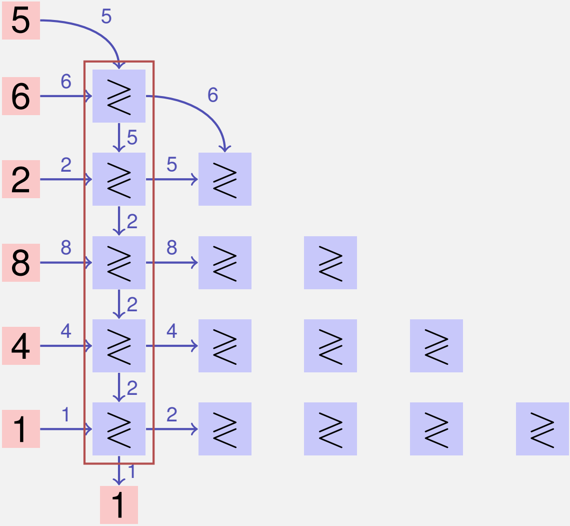
Anderer Blickwinkel



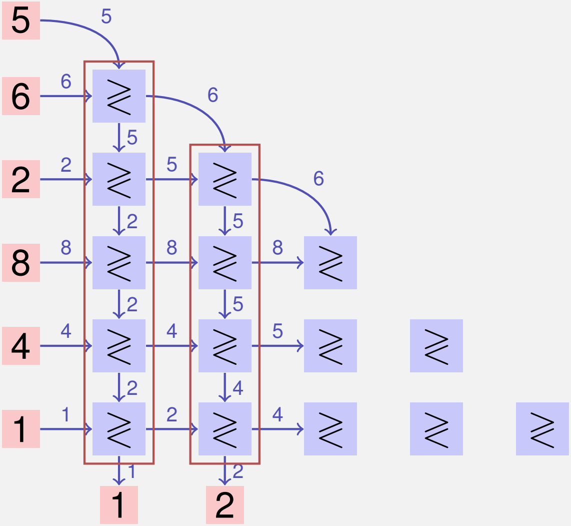
Anderer Blickwinkel



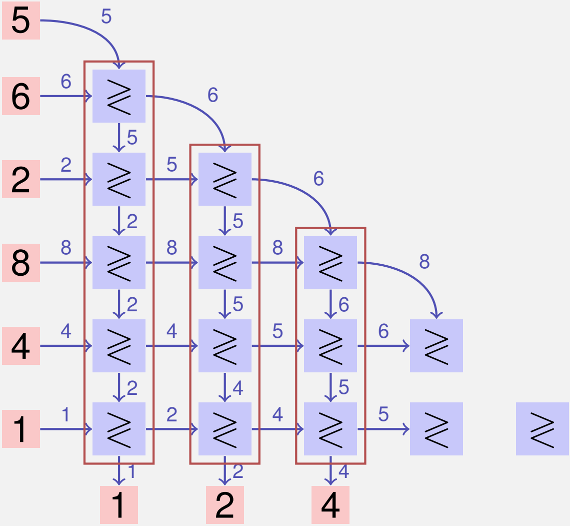
Anderer Blickwinkel



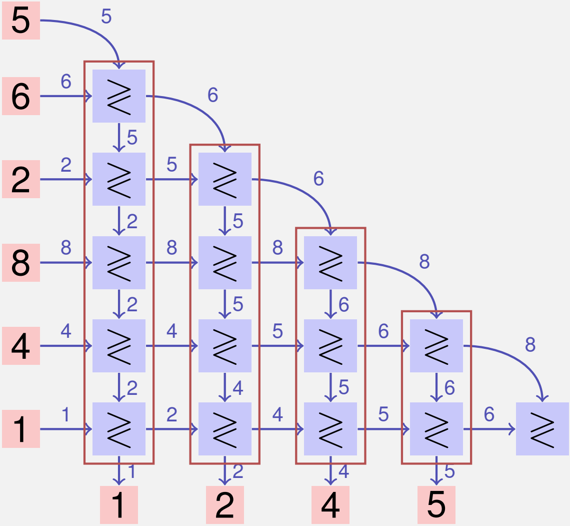
Anderer Blickwinkel



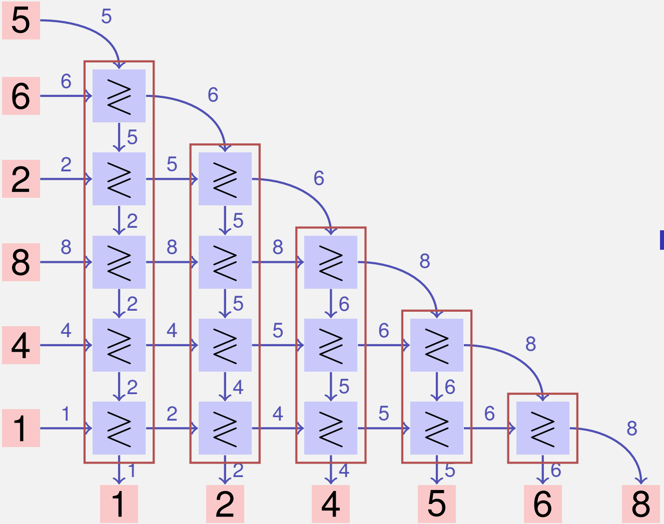
Anderer Blickwinkel



Anderer Blickwinkel

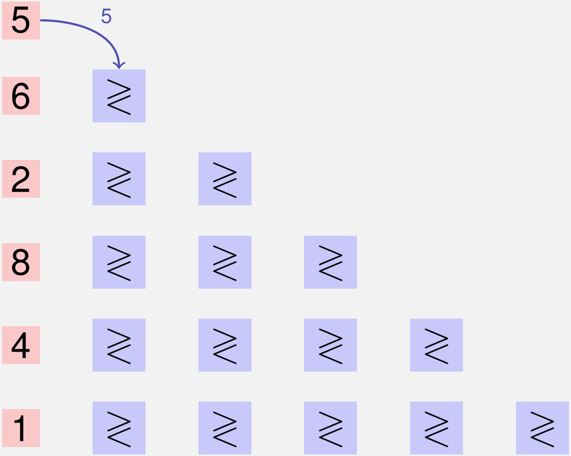


Anderer Blickwinkel

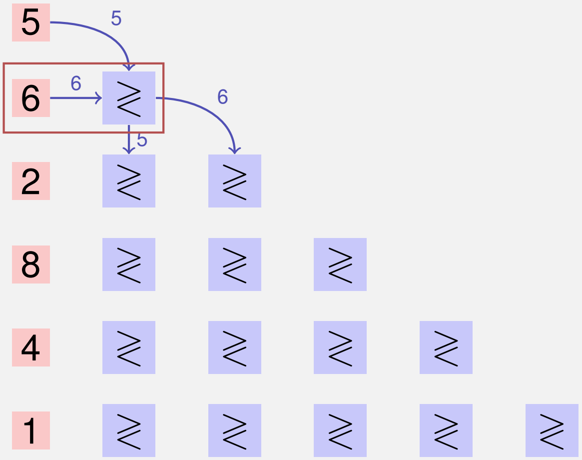


- Wie Selection Sort [und wie Bubble Sort]

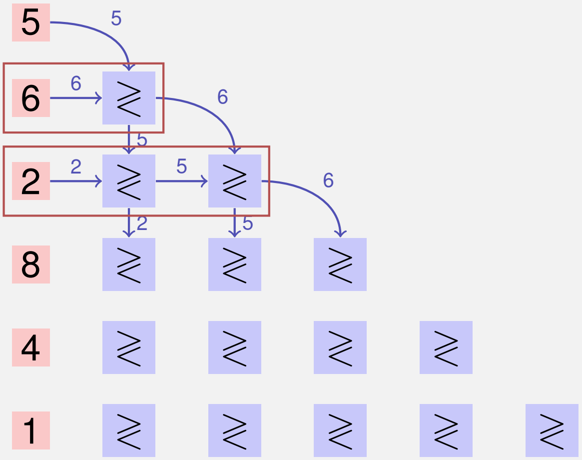
Anderer Blickwinkel



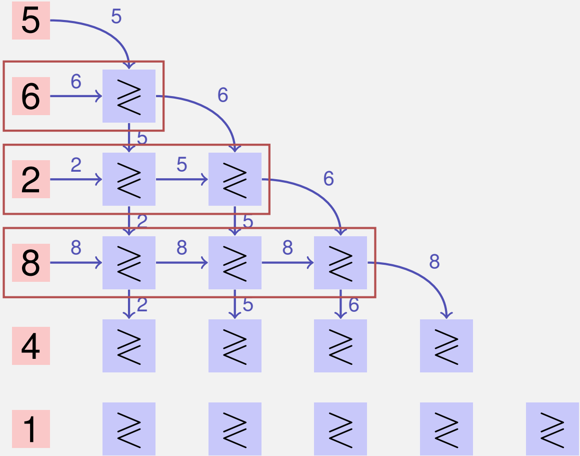
Anderer Blickwinkel



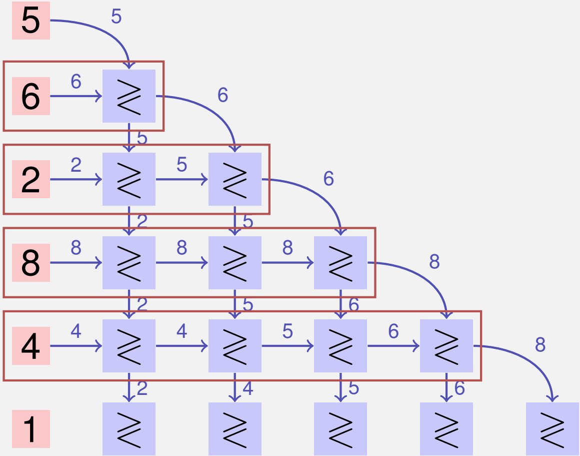
Anderer Blickwinkel



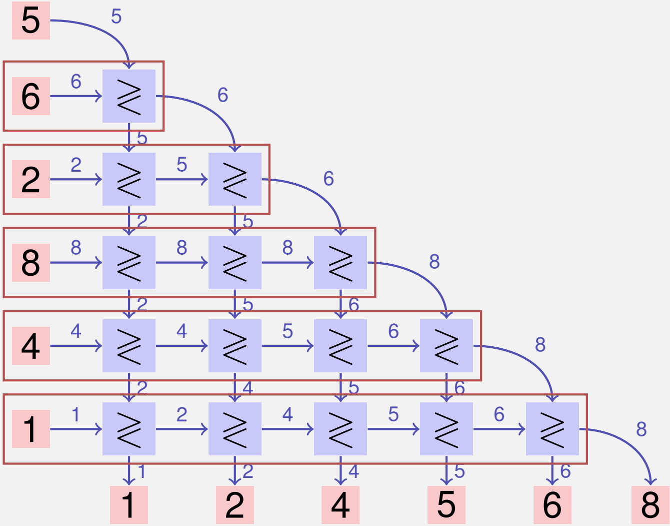
Anderer Blickwinkel



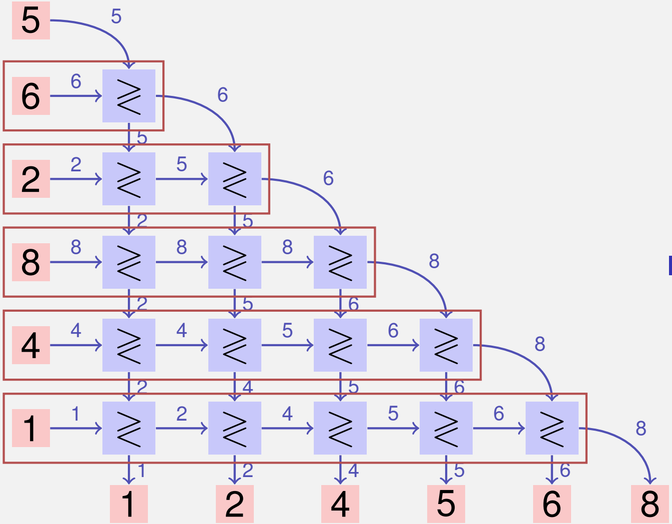
Anderer Blickwinkel



Anderer Blickwinkel



Anderer Blickwinkel



- Wie Insertion Sort

Schlussfolgerung

Selection Sort, Bubble Sort und Insertion Sort sind in gewissem Sinne dieselben Sortieralgorithmen. Wird später präzisiert. [5]

[5] Im Teil über parallele Sortiernetzwerke. Für sequentiellen Code gelten natürlich weiterhin die zuvor gemachten Feststellungen.

Shellsort

Insertion Sort auf Teilfolgen der Form $(A_{k\cdot i})$ ($i \in \mathbb{N}$) mit absteigenden Abständen k. Letzte Länge ist zwingend $k = 1$.

Gute Folgen: z.B. Folgen mit Abständen $k \in \{2^i 3^j | 0 \leq i, j\}$.

Shellsort

9 8 7 6 5 4 3 2 1 0

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$
1	0	3	2	5	4	7	6	9	8	

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$
1	0	3	2	5	4	7	6	9	8	
0	1	2	3	4	5	6	7	8	9	insertion sort, $k = 1$

8. Sortieren II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

Inspiration von Insertionsort: Schnelles Finden der Position

? Können wir das beste der beiden Welten haben?

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

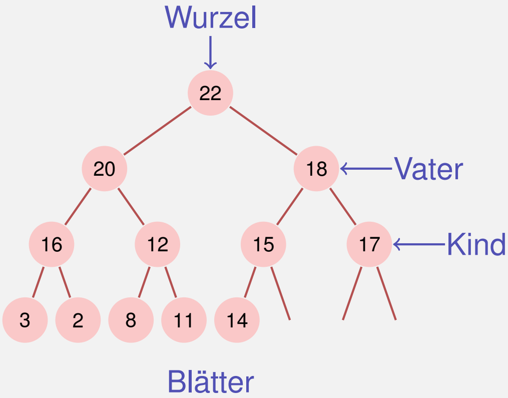
Inspiration von Insertionsort: Schnelles Finden der Position

? Können wir das beste der beiden Welten haben?

! Ja, aber nicht ganz so einfach...

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

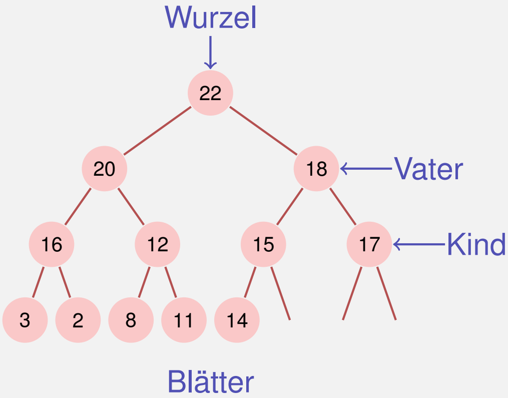


[6]Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene



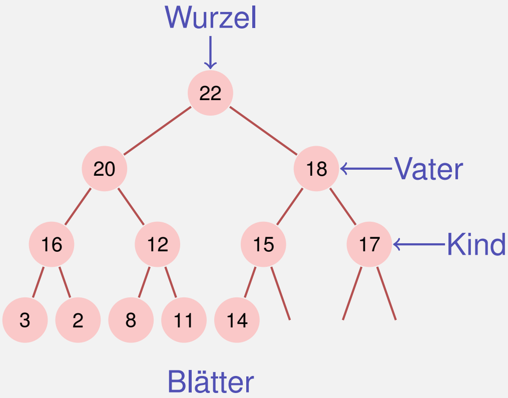
---

[6] Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.

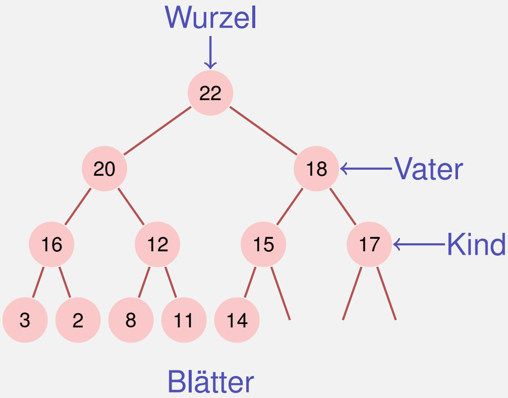


[6] Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. *Heap-Bedingung:* Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Vaters

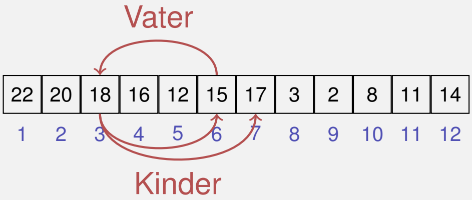


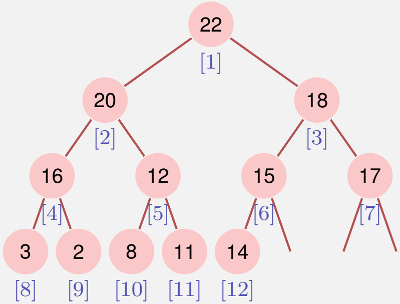
[6]Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

Heap und Array

Baum $\rightarrow$ Array:

- $\text{Kinder}(i) = \{2i, 2i+1\}$
- $\text{Vater}(i) = \lfloor i/2 \rfloor$



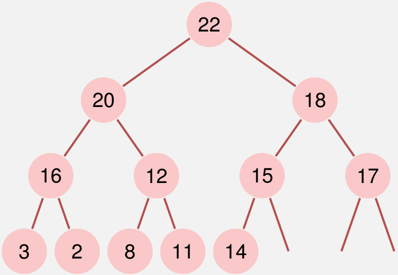


Abhängig von Startindex![7]

[7] Für Arrays, die bei 0 beginnen: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

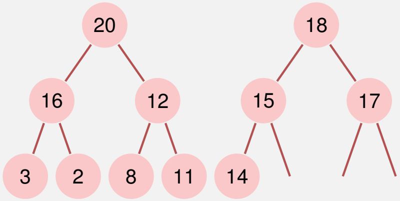
Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

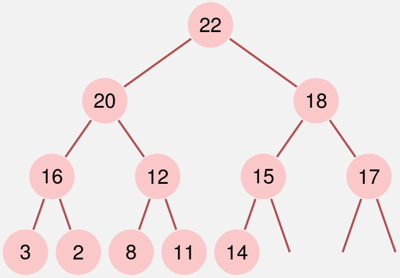


Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

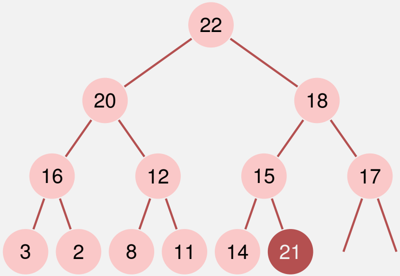


Einfügen



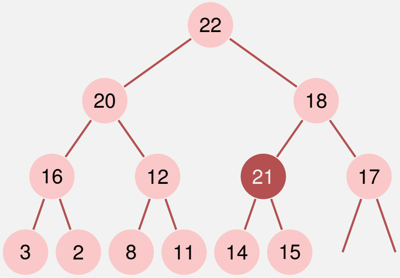
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



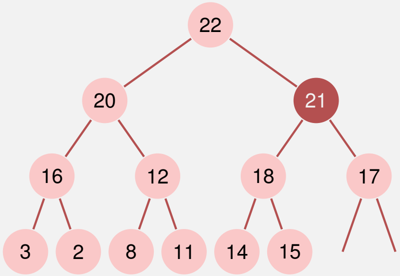
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



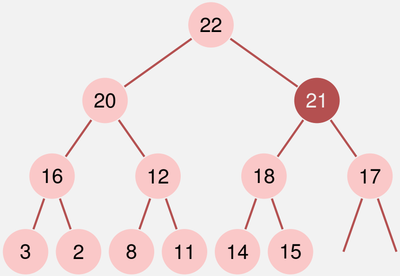
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.

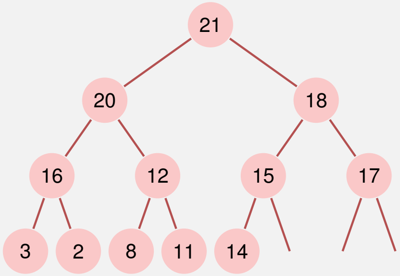


Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$

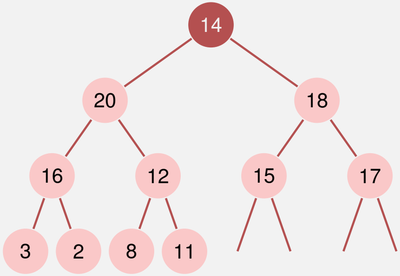


Maximum entfernen



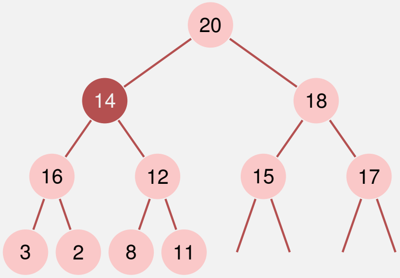
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



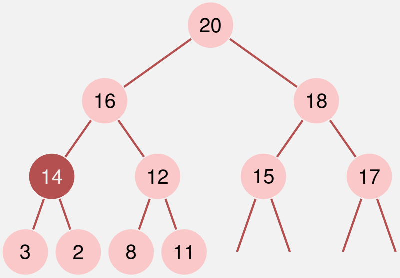
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



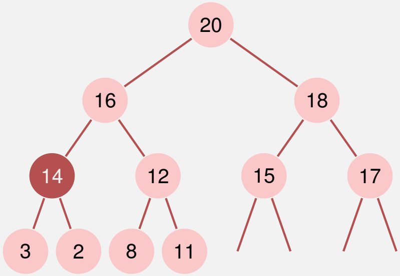
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Algorithmus Versickern(A, i, m)

Input : Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output : Array A mit Heapstruktur für i mit letztem Element m.

while $2i \leq m$ **do**
    $j \leftarrow 2i$; // j linkes Kind
    if $j < m$ and $A[j] < A[j+1]$ **then**
        $j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel
    if $A[i] < A[j]$ **then**
        swap($A[i], A[j]$)
        $i \leftarrow j$; // weiter versickern
    else
        $i \leftarrow m$; // versickern beendet

Heap Sortieren

7	6	4	5	1	2

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n - 1$);
- $n \leftarrow n - 1$

Heap Sortieren

	7	6	4	5	1	2
Tauschen $\Rightarrow$	2	6	4	5	1	7

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7

$A[1,...,n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n-1$

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n - 1$);
- $n \leftarrow n - 1$

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7

Heap Sortieren

$A[1,...,n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7
Versickern	$\Rightarrow$	5	4	2	1	6	7
Tauschen	$\Rightarrow$	1	4	2	5	6	7
Versickern	$\Rightarrow$	4	1	2	5	6	7
Tauschen	$\Rightarrow$	2	1	4	5	6	7
Versickern	$\Rightarrow$	2	1	4	5	6	7
Tauschen	$\Rightarrow$	1	2	4	5	6	7

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung:

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

Algorithmus HeapSort(A, n)

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

// Heap Bauen.
for $i \leftarrow n/2$ **downto** 1 **do**
    Versickere(A, i, n);

// Nun ist A ein Heap.
for $i \leftarrow n$ **downto** 2 **do**
    swap($A[1], A[i]$)
    Versickere($A, 1, i-1$)

// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. $\Rightarrow$ Heap Sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

---

[8] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade aber im Mittel viel kürzer, also sogar:

$$v(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \in \mathcal{O}(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

mit $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ [8] und $s(\frac{1}{2}) = 2$:

$$v(n) \in \mathcal{O}(n).$$

[8] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

8.2 Mergesort

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

Zwischenstand

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

Zwischenstand

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).

Zwischenstand

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

? Nachteile von Heapsort?

- ! Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).
- ! Zwei Vergleiche vor jeder benötigten Bewegung.

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

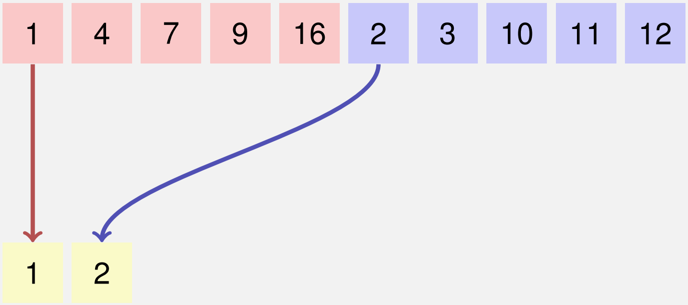
- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Sortierung des so vorsortierten A in $\mathcal{O}(n)$.

Merge

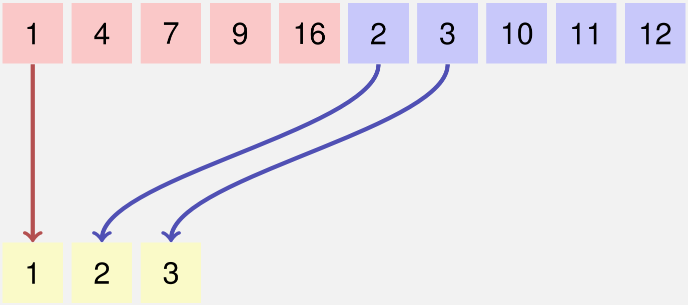
1	4	7	9	16	2	3	10	11	12

Merge

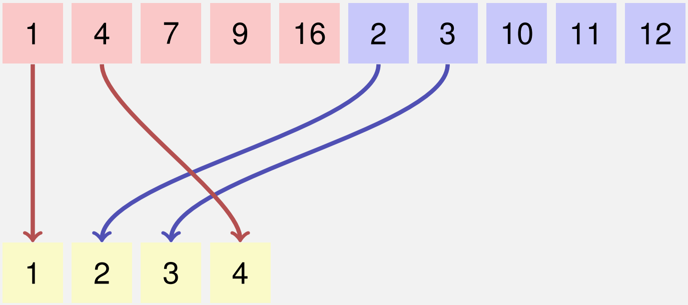
Merge



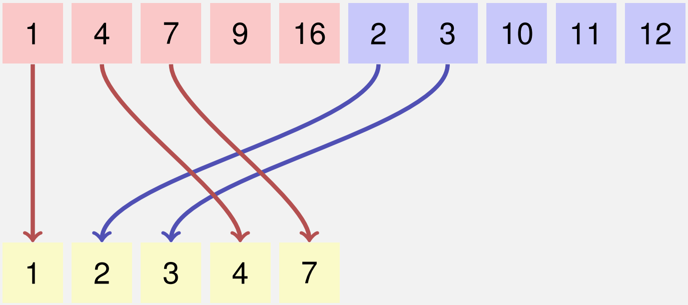
Merge



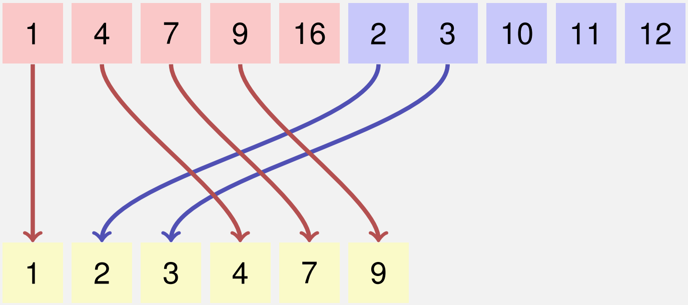
Merge



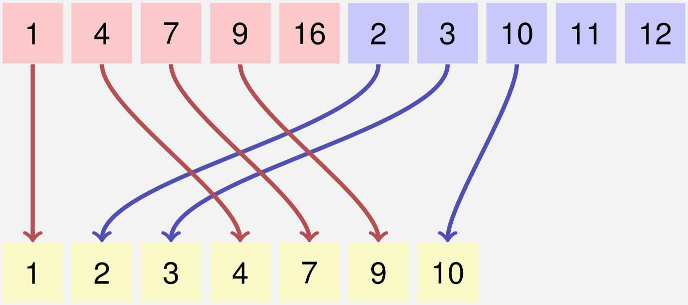
Merge



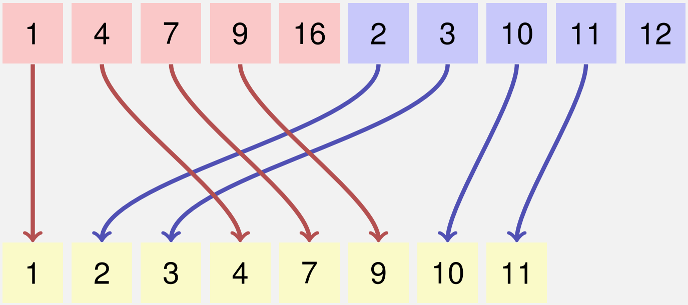
Merge



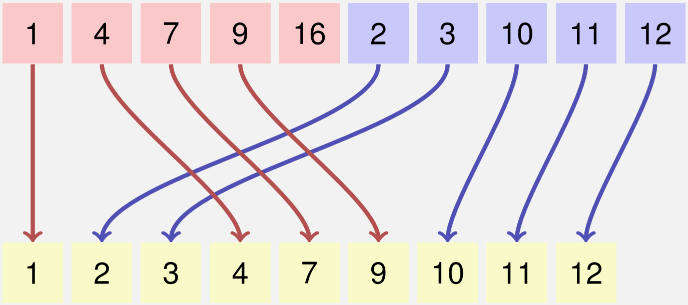
Merge



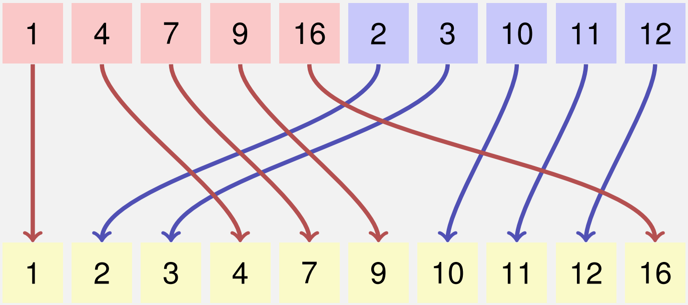
Merge



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 $B[1,\dots,k]$ 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 ($k \to k+1$):

- 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.

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 Merge(A, l, m, r) werden $\Theta(r-l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

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

Mergesort

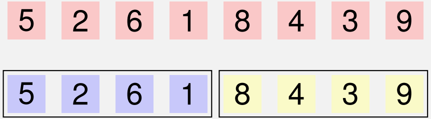
5	2	6	1	8	4	3	9

Mergesort

5	2	6	1	8	4	3	9

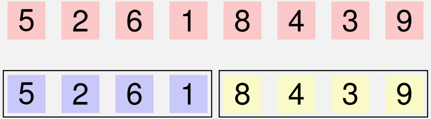
Split

Mergesort



Split

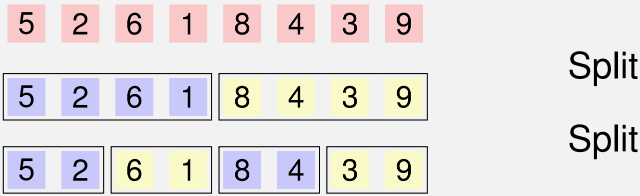
Mergesort



Split

Split

Mergesort



Mergesort

5 2 6 1 8 4 3 9

Split

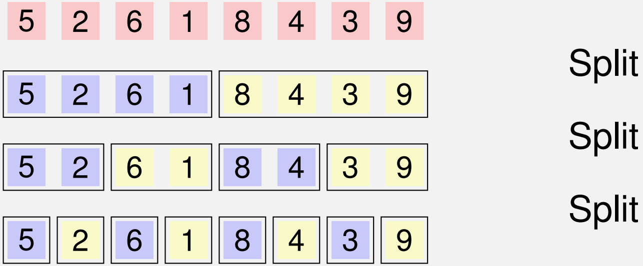
[5 2 6 1] [8 4 3 9]

Split

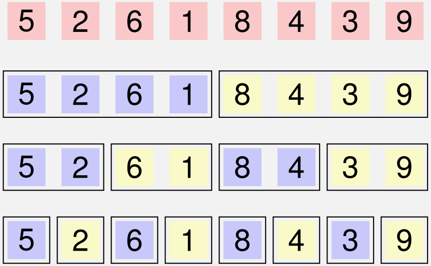
[5 2] [6 1] [8 4] [3 9]

Split

Mergesort



Mergesort



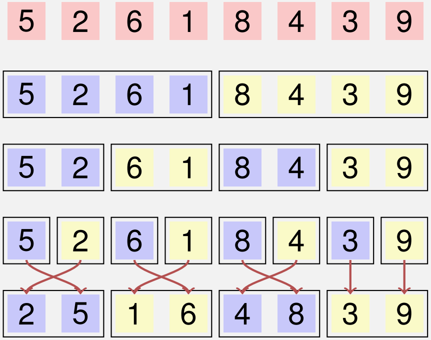
Split

Split

Split

Merge

Mergesort



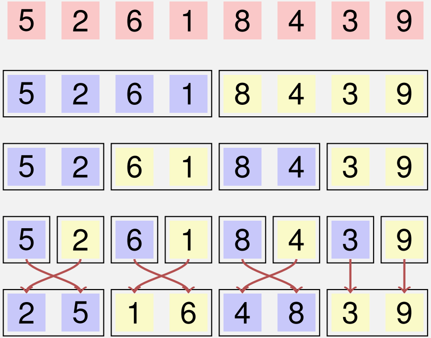
Split

Split

Split

Merge

Mergesort



Split

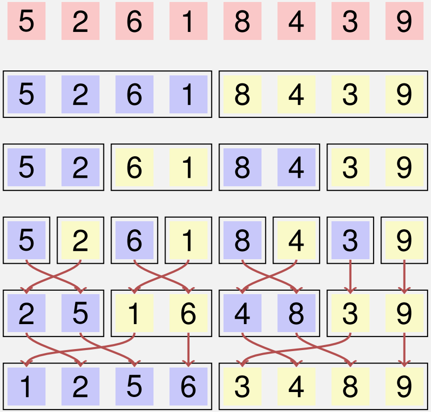
Split

Split

Merge

Merge

Mergesort



Split

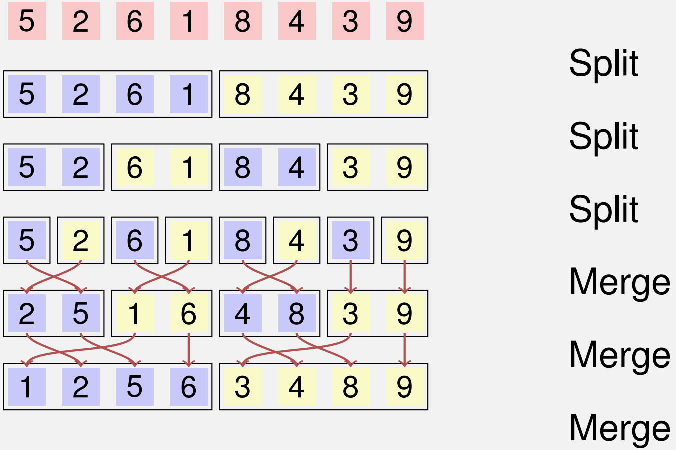
Split

Split

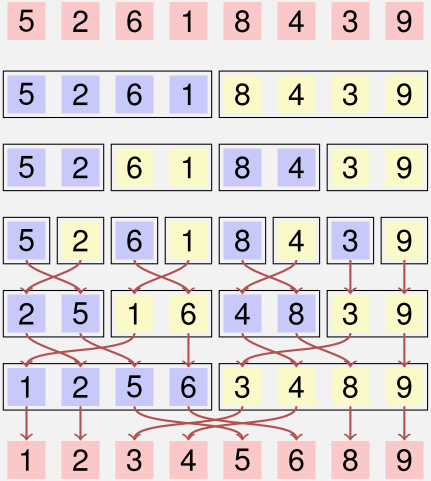
Merge

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
Split
Split
Split
Merge
Merge
Merge

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

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

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

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

$length \leftarrow 1$
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)
  $length \leftarrow length \cdot 2$

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.*

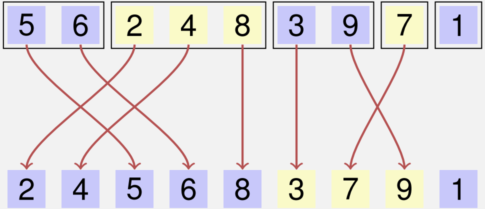
Natürliches 2-Wege Mergesort

5	6	2	4	8	3	9	7	1

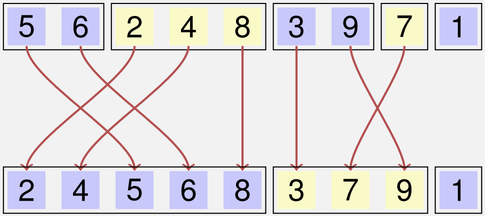
Natürliches 2-Wege Mergesort

5 6	2 4 8	3 9	7	1

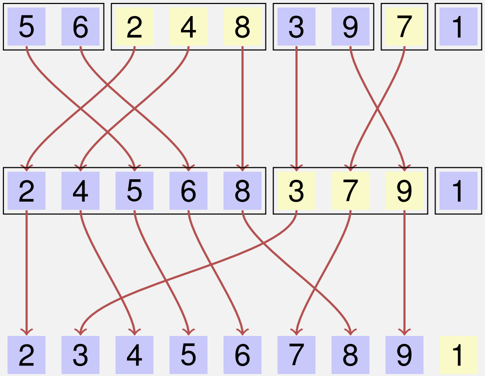
Natürliches 2-Wege Mergesort



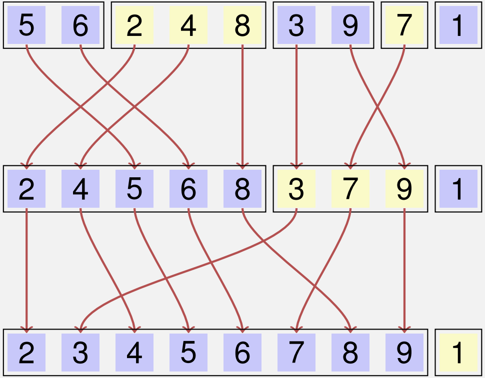
Natürliches 2-Wege Mergesort



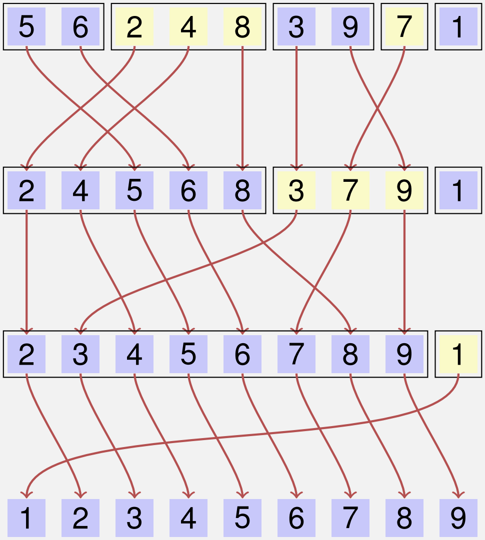
Natürliches 2-Wege Mergesort



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!

? Ist es auch im Mittel asymptotisch besser als StraightMergesort?

Analyse

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

? Ist es auch im Mittel asymptotisch besser als StraightMergesort?

! Nein. Unter Annahme der Gleichverteilung der paarweise unterschiedlichen Schlüssel haben wir im Mittel $n/2$ Stellen i mit $k_i > k_{i+1}$, also $n/2$ Runs und sparen uns lediglich einen Durchlauf, also n Vergleiche.

Natürliches Mergesort führt im schlechtesten und durchschnittlichen Fall $\Theta(n \log n)$ viele Vergleiche und Bewegungen aus.

8.3 Quicksort

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

Quicksort

? Was ist der Nachteil von Mergesort?

Quicksort

Was ist der Nachteil von Mergesort?

Benötigt $\Theta(n)$ Speicherplatz für das Verschmelzen.

Quicksort

? Was ist der Nachteil von Mergesort?

! Benötigt $\Theta(n)$ Speicherplatz für das Verschmelzen.

? Wie könnte man das Verschmelzen einsparen?

Quicksort

? Was ist der Nachteil von Mergesort?

! Benötigt $\Theta(n)$ Speicherplatz für das Verschmelzen.

? Wie könnte man das Verschmelzen einsparen?

! Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

? Wie?

Quicksort

? Was ist der Nachteil von Mergesort?

! Benötigt $\Theta(n)$ Speicherplatz für das Verschmelzen.

? Wie könnte man das Verschmelzen einsparen?

! Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

? Wie?

! Pivotieren und Aufteilen!

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

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 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

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	3	7	9	1
2	1	3	6	8	5	7	9	4
1	2	3	4	5	8	7	9	6
1	2	3	4	5	6	7	9	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$(A[k+1, \dots, r])$

Zur Erinnerung: Algorithmus Partition($A[l, \dots, r], p$)

Input : Array A, welches den Pivot p im Intervall $[l, r]$ mindestens einmal enthält.
Output : Array A partitioniert um p. Rückgabe der Position von p.

while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l + 1$
  while $A[r] > p$ **do**
    $r \leftarrow r - 1$
  swap($A[l]$, $A[r]$)
  if $A[l] = A[r]$ **then** // Nur für nicht paarweise verschiedene Schlüssel
    $l \leftarrow l + 1$

return l-1

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: Anzahl Vertauschungen

Resultat eines Aufrufes an Partition (Pivot 3):

2	1	3	6	8	5	7	9	4

? Wie viele Vertauschungen haben hier maximal stattgefunden?

Analyse: Anzahl Vertauschungen

Resultat eines Aufrufes an Partition (Pivot 3):

2	1	3	6	8	5	7	9	4

? Wie viele Vertauschungen haben hier maximal stattgefunden?

! 2. Die maximale Anzahl an Vertauschungen ist gegeben durch die Anzahl Schlüssel im kleineren Bereich.

Analyse: Anzahl Vertauschungen

Gedankenspiel

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.
- Jeder Schlüssel muss also maximal $\log n$ Münzen zahlen. Es gibt aber nur n Schlüssel.

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.
- Jeder Schlüssel muss also maximal $\log n$ Münzen zahlen. Es gibt aber nur n Schlüssel.

Folgerung: Es ergeben sich $\mathcal{O}(n \log n)$ viele Schlüsselvertauschungen im schlechtesten Fall!

Randomisiertes Quicksort

Quicksort wird trotz $\Theta(n^2)$ Laufzeit im schlechtesten Fall oft eingesetzt.

Grund: Quadratische Laufzeit unwahrscheinlich, sofern die Wahl des Pivots und die Vorsortierung nicht eine ungünstige Konstellation aufweisen.

Vermeidung: Zufälliges Ziehen eines Pivots. Mit gleicher Wahrscheinlichkeit aus $[l, r]$.

Analyse (Randomisiertes Quicksort)

Erwartete Anzahl verglichener Schlüssel bei Eingabe der Länge n:

$$T(n) = (n-1) + \frac{1}{n}\sum_{k=1}^{n}\left(T(k-1) + T(n-k)\right),\ T(0) = T(1) = 0$$

Behauptung $T(n) \leq 4n\log n$.

Beweis per Induktion:

Induktionsanfang: klar für $n = 0$ (mit $0\log 0 := 0$) und für $n = 1$.

Hypothese: $T(n) \leq 4n\log n$ für ein n.

Induktionsschritt: $(n-1 \to n)$

Analyse (Randomisiertes Quicksort)

$$
\begin{aligned}
T(n) = n-1+\frac{2}{n}\sum_{k=0}^{n-1} T(k) &\overset{\text{H}}{\leq} n-1+\frac{2}{n}\sum_{k=0}^{n-1} 4k\log k \\
&= n-1+\sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
&\leq n-1+\frac{8}{n}\left((\log n - 1)\sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k\right) \\
&= n-1+\frac{8}{n}\left((\log n)\cdot\frac{n(n-1)}{2} - \frac{n}{4}\left(\frac{n}{2}+1\right)\right) \\
&= 4n\log n - 4\log n - 3 \leq 4n\log n
\end{aligned}
$$

Analyse (Randomisiertes Quicksort)

Theorem

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

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n - 1$[9]. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

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

[9]Stack-Overflow möglich!

Quicksort mit logarithmischem Speicherplatz

Input : Array A der Länge n. $1 \leq l \leq r \leq n$.
Output : Array A, sortiert zwischen l und r.

while $l < r$ **do**
    Wähle Pivot $p \in A[l, \dots, r]$
    $k \leftarrow$ Partition$(A[l, \dots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \dots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \dots, r])$
        $r \leftarrow k - 1$

Der im ursprünglichen Algorithmus verbleibende Aufruf an Quicksort$(A[l, \dots, r])$ geschieht iterativ (Tail Recursion ausgenutzt!): die If-Anweisung wurde zur While Anweisung.

Praktische Anmerkungen

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

Es existiert eine Variante von Quicksort mit konstanten Speicherplatzbedarf. Idee: Zwischenspeichern des alten Pivots am Ort des neuen Pivots.