



Übungslektion 7 – Suchen und Sortieren

Informatik II

1. / 2. April 2025

Willkommen!

Polybox



Passwort: jschul

Personal Website



<https://n.ethz.ch/~jschul>

Heutiges Programm

Recap

Verbesserter Insertion Sort

Asymptotische Laufzeit Rekursiver Funktionen

Gruppenübung

Classes and Programming Concepts

Divide & Conquer Sortialgorithmen

Prüfungsaufgaben

Hausaufgaben

1. Recap

Asymptotische Laufzeit

A: $\Theta(1)$	B: $\Theta(\log n)$	C: $\Theta(\log^2 n)$	D: $\Theta(\log \log n)$
E: $\Theta(n)$	F: $\Theta(n^2)$	G: $\Theta(n^3)$	H: $\Theta(n^4)$
I: $\Theta(n \log n)$	K: $\Theta(n^2 \log n)$	L: $\Theta(n \log^2 n)$	M: $\Theta(n^2 \log^2 n)$
N: $\Theta(n^{\log_2 3})$	O: $\Theta(\sqrt{n})$	P: $\Theta(\sqrt{n^3})$	Q: $\Theta(\sqrt{\log n})$
R: $\Theta(\sqrt{n \log n})$	S: $\Theta(\sqrt[3]{2^n})$	T: $\Theta(2^n)$	U: $\Theta(3^n)$
V: $\Theta(n!)$	W: $\Theta(n^n)$	X: N/A	

$f(n)$	$f \in \dots$	$f(n)$	$f \in \dots$
0. $2n$	E		
1. 20^{23}	A	2. $\log(5^n)$	E
3. $n^2+n \cdot n^{1/2}$	F	4. $\log(n^{2023})$	B
5. $\sum_{i=1}^n n \cdot i$	G	6. $4n^2 + 2^n$	T

Asymptotische Laufzeit

----- (1.c.1) -----

Wenn Algorithmus A asymptotische Laufzeit $O(n)$ hat und Algorithmus B asymptotische Laufzeit $O(n^3)$, dann muss A asymptotisch schneller sein als B . / *If algorithm A has an asymptotic running time of $O(n)$ and algorithm B has asymptotic running time of $O(n^3)$ then A must be asymptotically faster than B .*

☐ Wahr / True

☒ Falsch / False

----- (1.c.2) -----

Wenn ein Problem Komplexität $\Omega(n^2)$ hat, so kann es keinen Algorithmus mit Laufzeit $O(n \log n)$ für dieses Problem geben. / *If a problem has complexity $\Omega(n^2)$, there cannot be an algorithm with running time $O(n \log n)$ for this problem.*

☒ Wahr / True

☐ Falsch / False

2. Verbesserter Insertion Sort

Letztes Mal: Insertion Sort

5 2 8 4 1

- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

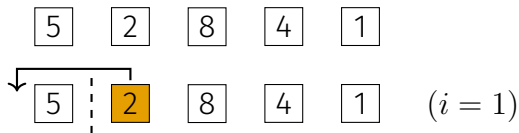
Letztes Mal: Insertion Sort

5 2 8 4 1

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

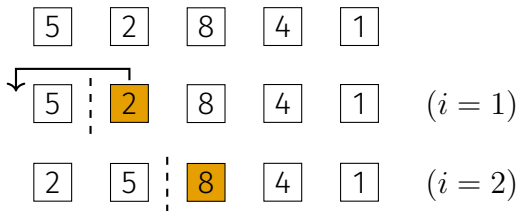
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



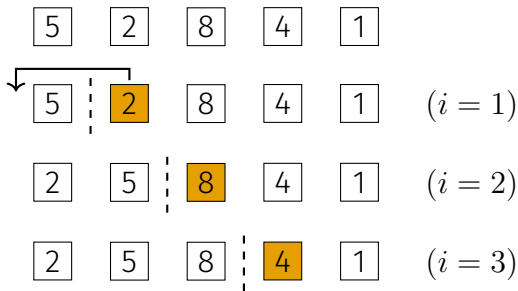
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



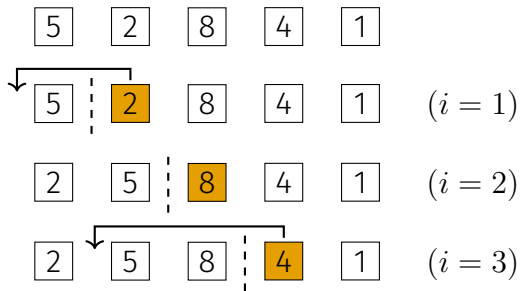
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



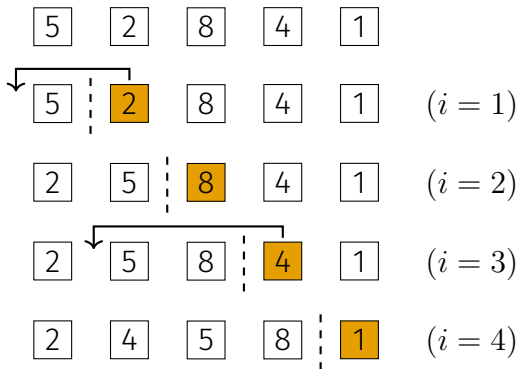
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



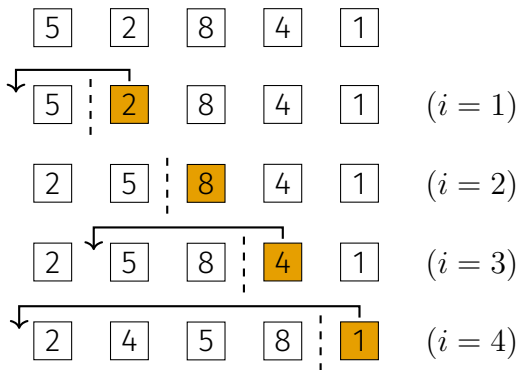
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



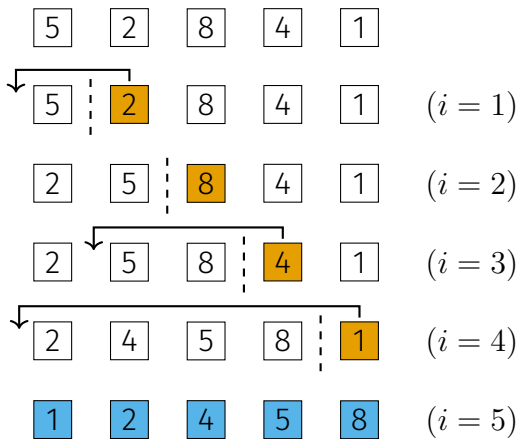
- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in $1i[:i]$ sortiert.
- Bei Iteration i , das i -te Element an der korrekten Position in die sortierte Teilliste $1i[:i]$ einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

■ Anzahl Vergleiche im schlechtesten Fall?

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

■ Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

- Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$
- Anzahl Vertauschungen im schlechtesten Fall?

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

- Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$
- Anzahl Vertauschungen im schlechtesten Fall? $\Theta(n^2)$

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

- Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$
- Anzahl Vertauschungen im schlechtesten Fall? $\Theta(n^2)$
- **Frage:** Wie können wir die Anzahl Vergleiche für den schlechtesten Fall verbessern?

Letztes Mal: Insertion Sort

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

Output: Sortiertes Array A

```
1 for i in range(1, len(A)):  
2     j = i  
3     while j > 0 and A[j-1] > A[j]:  
4         A[j], A[j-1] = A[j-1], A[j]  
5         j = j-1
```

- Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$
- Anzahl Vertauschungen im schlechtesten Fall? $\Theta(n^2)$
- **Frage:** Wie können wir die Anzahl Vergleiche für den schlechtesten Fall verbessern?

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

Binärer Insertion Sort

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

Output: Sortiertes Array A

```
1      for i in range(1,len(A)):  
2          x = A[i]  
3          p = BinarySearchIndex(A,0,i-1,x)  
4          for j in range(i-1,p-1,-1):  
5              A[j+1] = A[j]  
6          A[p] = x
```

Binärer Insertion Sort

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

Output: Sortiertes Array A

```
1      for i in range(1,len(A)):  
2          x = A[i]  
3          p = BinarySearchIndex(A,0,i-1,x)  
4          for j in range(i-1,p-1,-1):  
5              A[j+1] = A[j]  
6          A[p] = x
```

■ Anzahl Vergleiche im schlechtesten Fall?

Binärer Insertion Sort

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

Output: Sortiertes Array A

```
1      for i in range(1, len(A)):  
2          x = A[i]  
3          p = BinarySearchIndex(A, 0, i-1, x)  
4          for j in range(i-1, p-1, -1):  
5              A[j+1] = A[j]  
6          A[p] = x
```

■ Anzahl Vergleiche im schlechtesten Fall?

$$\sum_{i=1}^{n-1} a \cdot \log i = a \log((n-1)!) \in \Theta(n \log n)$$

Binärer Insertion Sort

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

Output: Sortiertes Array A

```
1      for i in range(1, len(A)):  
2          x = A[i]  
3          p = BinarySearchIndex(A, 0, i-1, x)  
4          for j in range(i-1, p-1, -1):  
5              A[j+1] = A[j]  
6          A[p] = x
```

- Anzahl Vergleiche im schlechtesten Fall?

$$\sum_{i=1}^{n-1} a \cdot \log i = a \log((n-1)!) \in \Theta(n \log n)$$

- Anzahl Kopiervorgänge im schlechtesten Fall?

Binärer Insertion Sort

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

Output: Sortiertes Array A

```
1      for i in range(1, len(A)):  
2          x = A[i]  
3          p = BinarySearchIndex(A, 0, i-1, x)  
4          for j in range(i-1, p-1, -1):  
5              A[j+1] = A[j]  
6          A[p] = x
```

- Anzahl Vergleiche im schlechtesten Fall?

$$\sum_{i=1}^{n-1} a \cdot \log i = a \log((n-1)!) \in \Theta(n \log n)$$

- Anzahl Kopiervorgänge im schlechtesten Fall? $\sum_{i=1}^{n-1} i \in \Theta(n^2)$

3. Asymptotische Laufzeit Rekursiver Funktionen

Übung 1

Geben Sie die Anzahl Aufrufe der Funktion f abhängig von n in der Θ -Notation für den untenstehenden Beispiel an. Kürzen Sie ihr Ergebnis soweit wie möglich. Geben Sie eine kurze Begründung.

```
1  #pre: n is a positive integer
2  def g(n):
3      count = 0
4      while n // (2 ** count) >= 1:
5          f()
6          count += 1
```

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

- Das Programm ruft die Funktion f , solange $n/2^{count} \geq 1$ ist.

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

- Das Programm ruft die Funktion f , solange $n/2^{count} \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq count$.

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

- Das Programm ruft die Funktion f , solange $n/2^{count} \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq count$.
- Die Variable *count* misst die Anzahl der Iterationen der while Schleife, und dabei auch die Anzahl Aufrufe der Funktion f .

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

- Das Programm ruft die Funktion f , solange $n/2^{\text{count}} \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq \text{count}$.
- Die Variable *count* misst die Anzahl der Iterationen der while Schleife, und dabei auch die Anzahl Aufrufe der Funktion f .
- Darum ist diese Anzahl $\log n$.

Antwort 1

```
1 #pre: n is a positive integer
2 def g(n):
3     count = 0
4     while n // (2 ** count) >= 1:
5         f()
6         count += 1
```

- Das Programm ruft die Funktion f , solange $n/2^{\text{count}} \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq \text{count}$.
- Die Variable *count* misst die Anzahl der Iterationen der while Schleife, und dabei auch die Anzahl Aufrufe der Funktion f .
- Darum ist diese Anzahl $\log n$.
- Das bedeutet, dass diese Anzahl $\Theta(\log n)$ ist.

Übung 2

Geben Sie die Anzahl Aufrufe der Funktion f abhängig von n in der Θ -Notation für den untenstehenden Beispiel an. Kürzen Sie ihr Ergebnis soweit wie möglich. Geben Sie eine kurze Begründung.

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         f()
5         g(n // 2)
```

Antwort 2

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         f()
5         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f .

Antwort 2

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         f()
5         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
 $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = k + T(n/2^k)$.

Antwort 2

```
1  # pre: n is a positive integer
2  def g(n):
3      if n >= 1:
4          f()
5          g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
 $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = k + T(n/2^k)$.

■ f ist aufgerufen, solange $n/2^k \geq 1$ ist.

Antwort 2

```
1  # pre: n is a positive integer
2  def g(n):
3      if n >= 1:
4          f()
5          g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
 $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = k + T(n/2^k)$.

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq k$.

Antwort 2

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         f()
5         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
 $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = k + T(n/2^k)$.

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq k$.
- Das bedeutet, dass die Anzahl der Aufrufe der Funktion f ist $\log n$ liegt, und dabei diese Anzahl $\Theta(\log n)$ ist.

Übung 3

Geben Sie die Anzahl Aufrufe der Funktion f abhängig von n in der Θ -Notation für den untenstehenden Beispiel an. Kürzen Sie ihr Ergebnis soweit wie möglich. Geben Sie eine kurze Begründung.

```
1  # pre: n is a positive integer
2  def g(n):
3      if n >= 1:
4          for i in range(n):
5              f()
6              g(n // 2)
```

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
$$T(n) = n + T(n/2)$$

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \end{aligned}$$

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$T(n) = n + T(n/2)$$

$$= n + n/2 + T(n/2^2)$$

$$= n + n/2 + \dots + n/2^k + T(n/2^{k+1})$$

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \end{aligned}$$

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

- Die letzte Ungleichung gilt, weil $\forall k : \sum_{j \leq k} 1/2^j \leq 2$.

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

- Die letzte Ungleichung gilt, weil $\forall k : \sum_{j \leq k} 1/2^j \leq 2$.
- Es gibt k , wofür $\lfloor n/2^{k+1} \rfloor = 0$, dann auch $T(\lfloor n/2^{k+1} \rfloor) = T(0) = 0$.

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

- Die letzte Ungleichung gilt, weil $\forall k : \sum_{j \leq k} 1/2^j \leq 2$.
- Es gibt k , wofür $\lfloor n/2^{k+1} \rfloor = 0$, dann auch $T(\lfloor n/2^{k+1} \rfloor) = T(0) = 0$.
- Das bedeutet, dass $T(n) \leq 2n \in O(n)$.

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

- Die letzte Ungleichung gilt, weil $\forall k : \sum_{j \leq k} 1/2^j \leq 2$.
- Es gibt k , wofür $\lfloor n/2^{k+1} \rfloor = 0$, dann auch $T(\lfloor n/2^{k+1} \rfloor) = T(0) = 0$.
- Das bedeutet, dass $T(n) \leq 2n \in O(n)$.
- Analog dazu kann man zeigen, dass $T(n) \in \Omega(n)$, weil $g(n)$ ruft f zumindest n -mal.

Antwort 3

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &= n + n/2 + T(n/2^2) \\ &= n + n/2 + \dots + n/2^k + T(n/2^{k+1}) \\ &= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1}) \\ &\leq 2n + T(n/2^{k+1}). \end{aligned}$$

- Die letzte Ungleichung gilt, weil $\forall k : \sum_{j \leq k} 1/2^j \leq 2$.
- Es gibt k , wofür $\lfloor n/2^{k+1} \rfloor = 0$, dann auch $T(\lfloor n/2^{k+1} \rfloor) = T(0) = 0$.
- Das bedeutet, dass $T(n) \leq 2n \in O(n)$.
- Analog dazu kann man zeigen, dass $T(n) \in \Omega(n)$, weil $g(n)$ ruft f zumindest n -mal.
- Darum, $T(n) \in \Theta(n)$.

Übung 4

Geben Sie die Anzahl Aufrufe der Funktion f abhängig von n in der Θ -Notation für den untenstehenden Beispiel an. Kürzen Sie ihr Ergebnis soweit wie möglich. Geben Sie eine kurze Begründung.

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
7             g(n // 2)
```

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6             g(n // 2)
7             g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:
$$T(n) = n + 2 T(n/2)$$

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$T(n) = n + 2 T(n/2) \\ = n + 2(n/2) + 2^2 T(n/2^2)$$

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \end{aligned}$$

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \\ &= n(k+1) + 2^{k+1} T(n/2^{k+1}). \end{aligned}$$

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \\ &= n(k+1) + 2^{k+1} T(n/2^{k+1}). \end{aligned}$$

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \\ &= n(k+1) + 2^{k+1} T(n/2^{k+1}). \end{aligned}$$

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq k$.

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \\ &= n (k + 1) + 2^{k+1} T(n/2^{k+1}). \end{aligned}$$

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq k$.
- Das bedeutet, dass $k = \log n$. Darum ist
$$T(n) = n (\log n + 1) + 2n T(1/2) = n (\log n + 1) + 2n T(0) = n (\log n + 1).$$

Antwort 4

```
1 # pre: n is a positive integer
2 def g(n):
3     if n >= 1:
4         for i in range(n):
5             f()
6         g(n // 2)
7         g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f . Wenn $n \geq 1$, es gilt:

$$\begin{aligned} T(n) &= n + 2 T(n/2) \\ &= n + 2(n/2) + 2^2 T(n/2^2) \\ &= n + 2(n/2) + \dots + 2^k(n/2^k) \\ &\quad + 2^{k+1}T(n/2^{k+1}) \\ &= n(k+1) + 2^{k+1} T(n/2^{k+1}). \end{aligned}$$

- f ist aufgerufen, solange $n/2^k \geq 1$ ist.
- Diese Ungleichung gilt genau dann, wenn $\log n \geq k$.
- Das bedeutet, dass $k = \log n$. Darum ist
$$T(n) = n(\log n + 1) + 2n T(1/2) = n(\log n + 1) + 2n T(0) = n(\log n + 1).$$
- Darum ist die Anzahl Aufrufe der Funktion f zwischen $\Theta(n \log n)$.

4. Gruppenübung

- Insertion Sort verbessern

Verbessern die Leistung des Insertion Sort mittels Binärsuche.
Mehr dazu in der detaillierten Aufgabenbeschreibung auf Code Expert.
<https://expert.ethz.ch/enrolled/SS25/mavt2/codeExamples>

Bezüglich Exercise 4: Classes and Programming Concepts

Übung 4: Classes and Programming Concepts

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
```

Übung 4: Classes and Programming Concepts

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
```

Hier `a` ist ein *Klassenattribut*. Klassenattribute werden von allen Instanzen einer Klasse geteilt.

Übung 4: Classes and Programming Concepts

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
```

```
1 class MyClass:
2     def __init__(self):
3         self.a = 'hello'
4         return
```

Hier *a* ist ein *Klassenattribut*. Klassenattribute werden von allen Instanzen einer Klasse geteilt.

Übung 4: Classes and Programming Concepts

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
```

Hier *a* ist ein *Klassenattribut*. Klassenattribute werden von allen Instanzen einer Klasse geteilt.

```
1 class MyClass:
2     def __init__(self):
3         self.a = 'hello'
4         return
```

Hier *a* ist ein *Instanzattribut*. Instanzattribute werden **nicht** zwischen verschiedenen Instanzen geteilt.

Übung 4: Classes and Programming Concepts

■ Beispiel 1:

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
5
6 first = MyClass()
7 MyClass.a = 'world'
8
9 second = MyClass()
10 print(second.a)
```

Übung 4: Classes and Programming Concepts

■ Beispiel 1:

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
5
6 first = MyClass()
7 MyClass.a = 'world'
8
9 second = MyClass()
10 print(second.a)
```

In Zeile 7 ändern wir ein Klassenattribut, welches von allen Instanzen geteilt wird.

Prints: world

Übung 4: Classes and Programming Concepts

■ Beispiel 2:

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
5
6 first = MyClass()
7 first.a = 'world'
8
9 second = MyClass()
10 print(first.a + ' ' + second.a)
```

Übung 4: Classes and Programming Concepts

■ Beispiel 2:

```
1 class MyClass:
2     a = 'hello'
3     def __init__(self):
4         return
5
6 first = MyClass()
7 first.a = 'world'
8
9 second = MyClass()
10 print(first.a + ' ' + second.a)
```

In Zeile 7 erstellen wir ein *Instanzattribut*, welches das Klassenattribut verbirgt.

Prints: world hello

Übung 4: Classes and Programming Concepts

■ Beispiel 3:

```
1 class MyClass:
2     def __init__(self):
3         self.a = 'hello'
4
5 first = MyClass()
6 first.a = 'world'
7
8 second = MyClass()
9 print(second.a)
```

Übung 4: Classes and Programming Concepts

■ Beispiel 3:

```
1 class MyClass:
2     def __init__(self):
3         self.a = 'hello'
4
5 first = MyClass()
6 first.a = 'world'
7
8 second = MyClass()
9 print(second.a)
```

Prints: hello

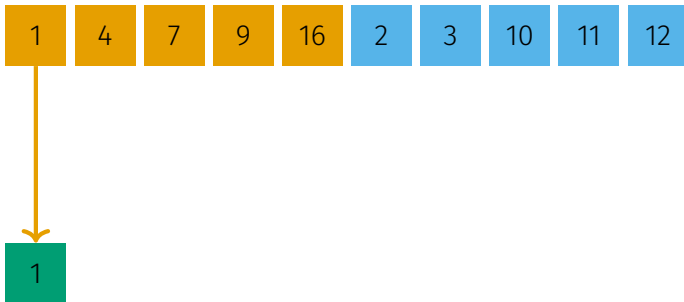
`self.a` bezieht sich auf ein *Instanzattribut* und wird **nicht** zwischen verschiedenen Instanzen geteilt.

6. Divide & Conquer Sortieralgorithmen

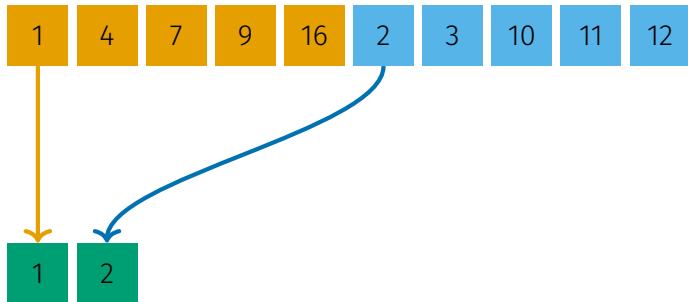
Merge



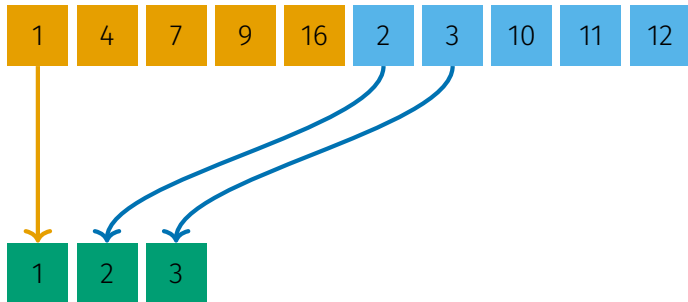
Merge



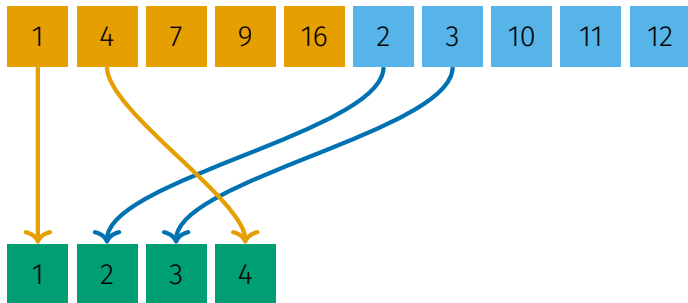
Merge



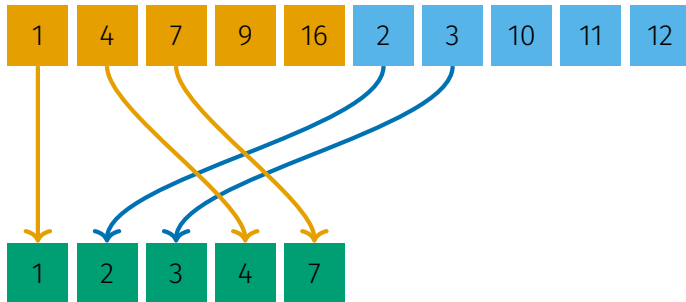
Merge



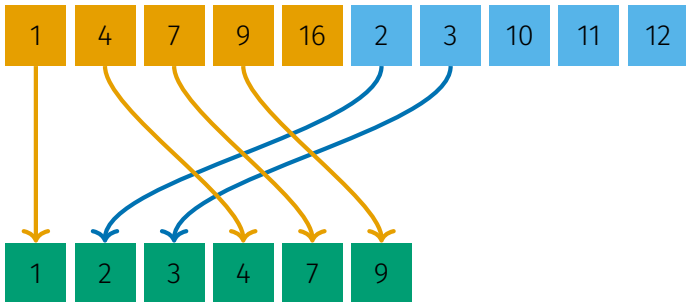
Merge



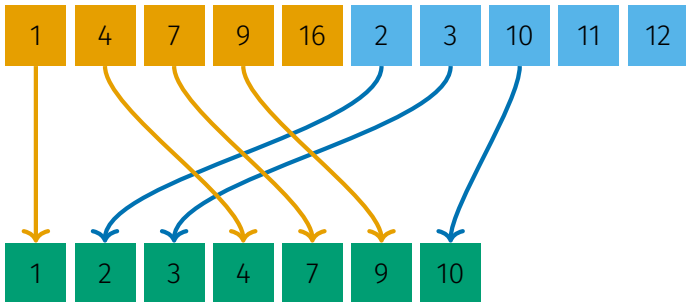
Merge



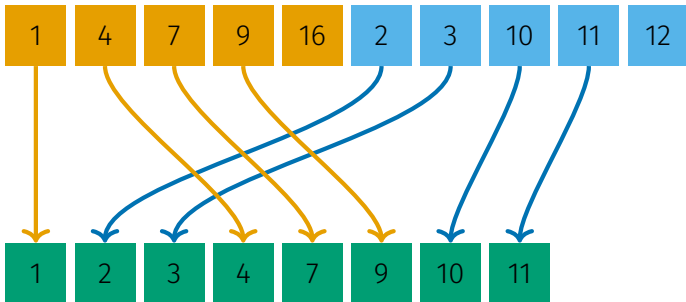
Merge



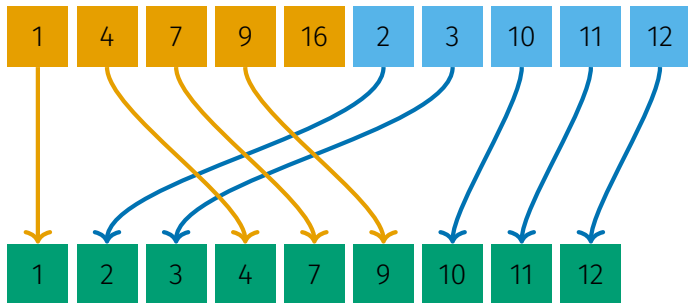
Merge



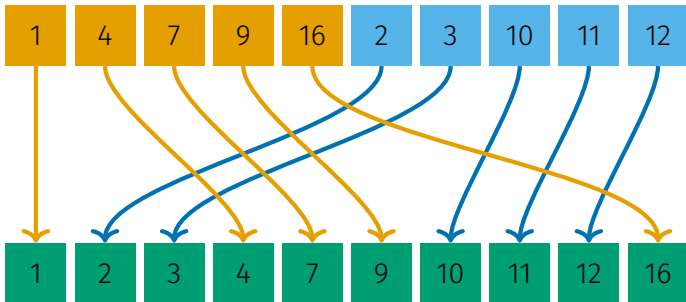
Merge



Merge



Merge



Algorithmus merge

```
1 def merge(a1, a2):
2     b, i, j = [], 0, 0
3     while i < len(a1) and j < len(a2):
4         if a1[i] < a2[j]:
5             b.append(a1[i])
6             i += 1
7         else:
8             b.append(a2[j])
9             j += 1
10    b += a1[i:]
11    b += a2[j:]
12    return b
```

Algorithmus merge_sort

```
1 def merge_sort(a):
2     if len(a) <= 1:
3         return a
4     else:
5         sorted_a1 = merge_sort(a[:len(a) // 2])
6         sorted_a2 = merge_sort(a[len(a) // 2:])
7         return merge(sorted_a1, sorted_a2)
```

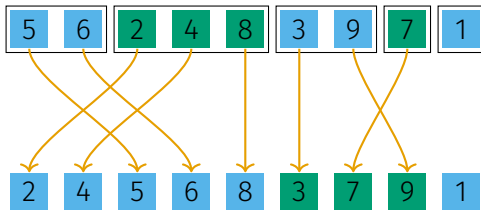
Natürlicher 2-Wege Mergesort

5 6 2 4 8 3 9 7 1

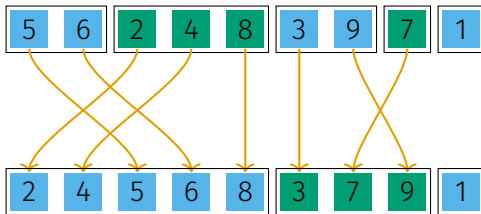
Natürlicher 2-Wege Mergesort



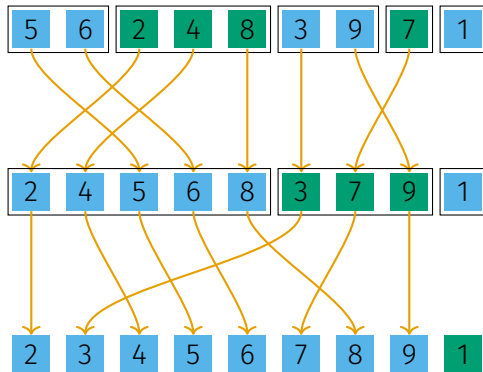
Natürlicher 2-Wege Mergesort



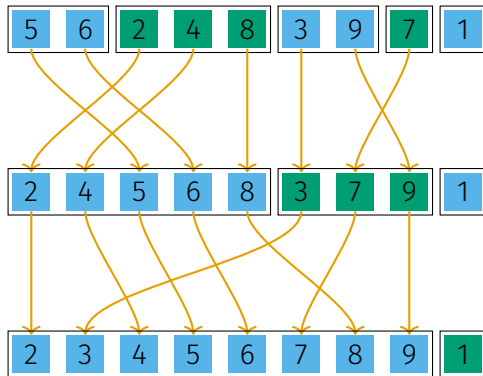
Natürlicher 2-Wege Mergesort



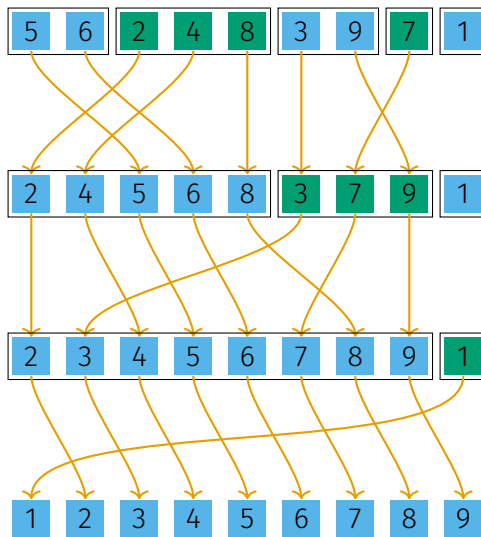
Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Algorithmus NaturalMergesort(A)

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

```
1  def NaturalMergesort(A):
2      l = -1
3      while l = 0: #eigentlich != aber es geht hier nicht
4          r = 0
5          while r < len(A)-1:
6              l = r
7              m = l
8              while m < len(A)-1 and A[m+1] >= A[m]:
9                  m = m+1
10             if m < len(A)-1:
11                 r = m+1
12                 while r < len(A)-1 and A[r+1] >= A[r]:
13                     r = r+1
14                 Merge(A,l,m,r)
15             else:
16                 r = len(A)-1
17             r+=1
```

7. Prüfungsaufgaben

Sorting Algorithm

Geben Sie für die folgenden Algorithmen jeweils an, welche der folgenden Strategien dem Algorithmus zugrunde liegt.

- Divide and Conquer (DC)
- Greedy (G)
- Dynamische Programmierung / Dynamic Programming (DP)
- Brute Force (BF)

For the following algorithms, indicate which of the following strategies underlies the algorithm.

Merge Sort / *Merge Sort*

DC

Lineare Suche in einem Array / *Linear search in an array*

BF

Sorting Algorithm

Nehmen Sie an, dass Sie in jedem Schritt von Quick Sort ein Pivot-Element finden, das Ihre Daten in zwei Teile aufteilt mit Grössenverhältnis 1:2 (der eine Teil ist doppelt so gross wie der andere). Was ist dann die Laufzeit von Quick Sort?

- ☐ $\Theta(n^2)$
- ☐ $\Theta(n)$
- ☒ $\Theta(n \log n)$
- ☐ $\Theta(n^3)$
- ☐ $\Theta(1)$

Sorting Algorithm

Sie lassen Insertion Sort auf einem Array für mindestens 3 Iterationen laufen. Wie könnte das Array aussehen?

You let insertion sort run on an array for at least 3 iterations. How could the array look like?

☒ 1 2 3 4 5 6 ☐ 4 5 2 6 3 1 ☒ 4 5 6 3 2 1 ☒ 1 2 3 6 5 4
☒ 2 4 6 1 3 5 ☒ 1 3 5 2 4 6 ☐ 6 5 4 3 2 1

Sorting Algorithm

Ordnen Sie den folgenden Aussagen die Sortieralgorithmen Merge Sort (M), Selection Sort (S) und Insertion Sort (I) zu. Nach 7 Schritten (d.h., Iterationen oder Merge-Schritten) ...

Assign the sorting algorithms merge sort (M), selection sort (S), and insertion sort (I) to the following invariants. After 4 steps (i.e., iterations or merge steps), ...

- 1... sind die ersten 7 Elemente sortiert. / *the first 7 elements are in sorted order.*
- 2... sind die ersten 8 Elemente sortiert. / *the first 8 elements are in sorted order.*
- 3... sind die 7 kleinsten Elemente sortiert an den ersten 7 Stellen. / *the 7 smallest elements are sorted at the first 7 positions.*

☐ MIS ☐ MSI ☒ IMS ☐ ISM ☐ SMI ☐ SIM

Sorting Algorithm

(3.a) Was ist die asymptotische Komplexität von binärer Suche in einem sortierten Array im schlechtesten Fall? / *What is the worst-case complexity of binary search on a sorted array?*

- ☐ $O(n)$ ☐ $O(n^2)$ ☒ $O(\log n)$ ☐ $O(n \log n)$

(3.b) Welche der folgende ist eine Vorbedingung für binäre Suche? / *Which of the following is a precondition for binary search?*

☒ Eingabeliste ist in aufsteigender Reihenfolge sortiert / *Input list is sorted in increasing order*

☐ Eingabeliste ist nicht sortiert / *Input list is not sorted*

☐ Erste Hälfte der Eingabeliste ist in aufsteigender Reihenfolge sortiert / *First half of the input list is sorted*

☐ Zweite Hälfte der Eingabeliste ist in aufsteigender Reihenfolge sortiert / *Second half of the input list is sorted*

Sorting Algorithm

Unter welcher Bedingung hätte Quicksort eine Laufzeit von $O(n^2)$? / *Under which condition would quicksort have a runtime of $O(n^2)$?*

- ☒ Eingabeliste ist in absteigender Reihenfolge sortiert / *Input list is sorted in decreasing order*
- ☐ Eingabeliste ist nicht sortiert / *Input list is not sorted*
- ☐ Erste Hälfte der Eingabeliste ist sortiert / *First half of the input list is sorted*
- ☐ Zweite Hälfte der Eingabeliste ist sortiert / *Second half of the input list is sorted*

Nehmen wir an, wir sortieren eine Liste mit Quicksort. Nach der ersten Partitionierung sieht die Liste folgendermassen aus: / *Assume that we are sorting a list using quicksort. After the first partitioning, the list looks like this:* 3, 6, 2, 8, 10, 13, 12, 11. Welche der folgenden Aussagen ist wahr? / *Which of the following is true?*

- ☐ Der Pivot könnte 8 aber nicht 10 sein / *The pivot could be 8, but not 10*
- ☐ Der Pivot könnte 10 aber nicht 8 sein / *The pivot could be 10, but not 8*
- ☒ Der Pivot könnte 8 oder 10 sein / *The pivot could be 8 or 10*
- ☐ Der Pivot könnte weder 8 noch 10 sein / *The pivot cannot be 8 nor 10*

Sorting Algorithm

Welche ist die asymptotische Komplexität von Quicksort im schlechtesten Fall? / *What is quicksort's worst-case asymptotic complexity?*

- ☐ $O(1)$ ☐ $O(\log n)$ ☐ $O(n)$ ☐ $O(n \log n)$ ☒ $O(n^2)$

-----**(2.a.2)**-----

Welche ist die asymptotische Komplexität des Sortierens eines Arrays mit Heapsort im schlechtesten Fall? / *What is the worst-case asymptotic complexity of sorting an array with heap sort?*

- ☐ $O(1)$ ☐ $O(\log n)$ ☐ $O(n)$ ☒ $O(n \log n)$ ☐ $O(n^2)$

Sorting Algorithm

Die Komplexität von Mergesort hängt nicht von der anfänglichen Reihenfolge des Arrays ab. / *Mergesort's complexity does not depend on the initial order of the array.*

☒ Wahr / *True*

☐ Falsch / *False*

----- (2.b.2) -----

Quicksort benötigt zusätzlichen Speicherplatz in der Grösse von $O(n)$. / *Quicksort needs $O(n)$ additional space.*

☐ Wahr / *True*

☒ Falsch / *False*

Rekursive Runtime Analysis

```
def g(n):  
    f()  
    f()  
    if n >= 1:  
        g(n - 2)
```

Asymptotische Anzahl Aufrufe von `f` / *Asymptotic number of calls to `f`*:

- ☐ 1 ☒ ~~n~~ ☐ $\log n$ ☐ \sqrt{n} ☐ n^2 ☐ 2^n ☐ $n \log n$ ☐ n^3

8. Hausaufgaben

Übung 6: Search and Sort

Auf <https://expert.ethz.ch/enrolled/SS25/mavt2/exercises>

Übung 6: Search and Sort **Einfach**, **Mittel**, **Schwer**

- **Eierwerfen** (Probiert es, sonst Youtube)
- **Vergleich von Sortieralgorithmen** (ZF und Vorlesung helfen!)
- **Verbesserter Insertion Sort** (Vorlesung, Binäre Suche verstehen!)
- **Invarianten der Suchalgorithmen** (Versucht euch zu erinnern! Sonst slides)

Abgabedatum: Montag 07.04.2025, 20:00 CET

KEINE HARDCODIERUNG

Feedback

Ich bin euch sehr dankbar für jegliche Rückmeldung, damit ich die Übungsstunde für **Euch** besser gestalten kann. Seien es Anmerkungen zu:

- **Aufbau**
- **Inhalt**
- **Redetempo**
- **Beispielen**
- **Website/Polybox**

oder anderem. Ich freue mich über **alles!**



<https://n.ethz.ch/~jschul/>
Feedback

Feedback



<https://n.ethz.ch/~jschul/Feedback>

Fragen oder Anregungen?