ETH zürich

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 Sortieralgorithmen

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	

	$f(n)$	$f \in \ldots$		$f(n)$	$f \in \ldots$
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

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

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

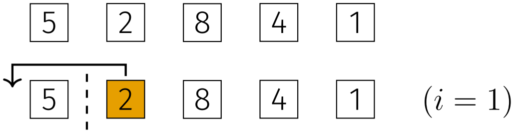
Letztes Mal: Insertion Sort

5 2 8 4 1

5 ¦ 2 8 4 1 $(i = 1)$

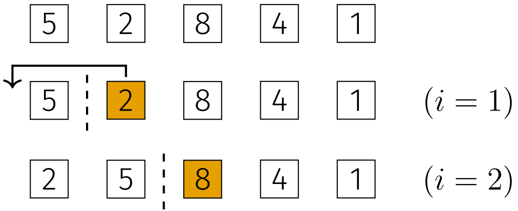
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis das gesamte Array sortiert ist $(i = n)$.

Letztes Mal: Insertion Sort



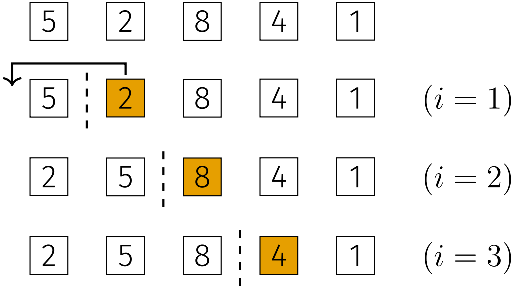
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



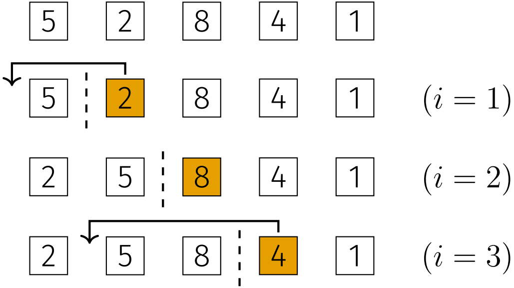
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



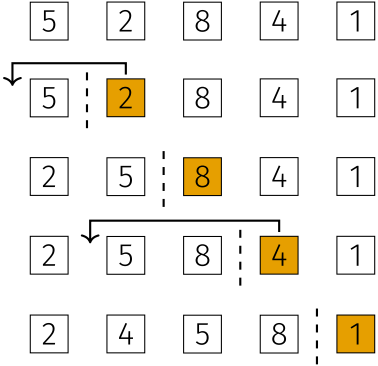
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



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

Letztes Mal: Insertion Sort



$(i = 1)$

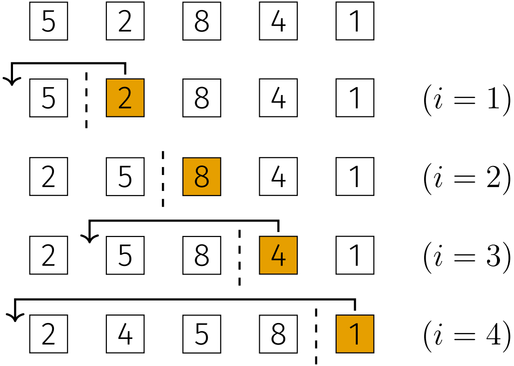
$(i = 2)$

$(i = 3)$

$(i = 4)$

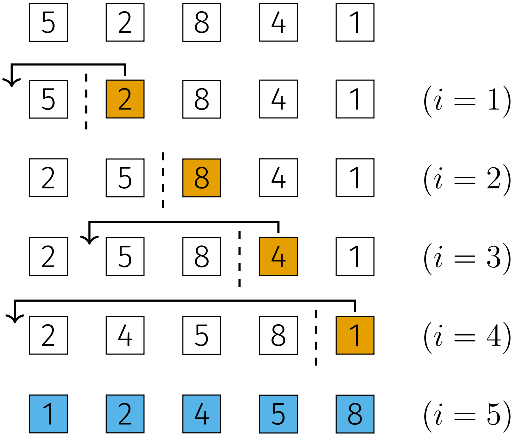
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis das gesamte Array sortiert ist ($i = n$).

Letztes Mal: Insertion Sort



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

Letztes Mal: Insertion Sort



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

Letztes Mal: Insertion Sort

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

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

- Anzahl Vergleiche im schlechtesten Fall?

Letztes Mal: Insertion Sort

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

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

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

Letztes Mal: Insertion Sort

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

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

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

Letztes Mal: Insertion Sort

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

```
for i in range(1,len(A)):
    j = i
    while j>0 and A[j-1] > A[j]:
        A[j], A[j-1] = A[j-1], A[j]
        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 $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

```
for i in range(1,len(A)):
    j = i
    while j>0 and A[j-1] > A[j]:
        A[j], A[j-1] = A[j-1], A[j]
        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 $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

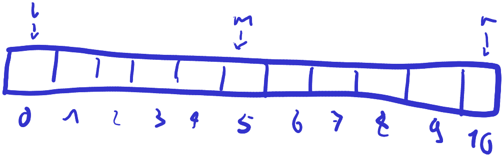
```
for i in range(1,len(A)):
    j = i
    while j>0 and A[j-1] > A[j]:
        A[j], A[j-1] = A[j-1], A[j]
        j = j-1
```

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

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



x = 7
l = 0
r = 10

Binärer Insertion Sort

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

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

- Anzahl Vergleiche im schlechtesten Fall?

Binärer Insertion Sort

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

```
for i in range(1,len(A)):
  x = A[i]
  p = BinarySearchIndex(A,0,i-1,x)
  for j in range(i-1,p-1,-1):
      A[j+1] = A[j]
  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 $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

```
for i in range(1,len(A)):
  x = A[i]
  p = BinarySearchIndex(A,0,i-1,x)
  for j in range(i-1,p-1,-1):
      A[j+1] = A[j]
  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 $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

```
for i in range(1,len(A)):
  x = A[i]
  p = BinarySearchIndex(A,0,i-1,x)
  for j in range(i-1,p-1,-1):
      A[j+1] = A[j]
  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.

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

Antwort 1

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

Antwort 1

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

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

Antwort 1

```python
#pre: n is a positive integer
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        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

```python
#pre: n is a positive integer
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        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

```python
#pre: n is a positive integer
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        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.
- Darum ist diese Anzahl $\log n$.

Antwort 1

```python
#pre: n is a positive integer
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        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.
- 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.

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

$T(n) = 1 + T(n/2)$

Antwort 2

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

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

Antwort 2

```
# pre: n is a positive integer
  def g(n):
    if n >= 1:
        f()
        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

```python
# pre: n is a positive integer
  def g(n):
    if n >= 1:
        f()
        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

```python
# pre: n is a positive integer
  def g(n):
    if n >= 1:
        f()
        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

```
# pre: n is a positive integer
  def g(n):
    if n >= 1:
        f()
        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.

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

Antwort 3

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

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

Antwort 3

```python
# pre: n is a positive integer
  def g(n):
    if n >= 1:
        for i in range(n):
            f()
        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

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

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

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

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

Antwort 3

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

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

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

Antwort 3

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

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

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

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

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

$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$

Antwort 3

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

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

$T(n) = n + T(n/2)$
$= n + n/2 + T(n/2^2)$
$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$
$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$
$\leq 2n + T(n/2^{k+1})$.

Antwort 3

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

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

$T(n) = n + T(n/2)$
$= n + n/2 + T(n/2^2)$
$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$
$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$
$\leq 2n + T(n/2^{k+1})$.

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

Antwort 3

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

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

$T(n) = n + T(n/2)$
$= n + n/2 + T(n/2^2)$
$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$
$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$
$\leq 2n + T(n/2^{k+1})$.

- 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

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

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

$T(n) = n + T(n/2)$
$= n + n/2 + T(n/2^2)$
$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$
$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$
$\leq 2n + T(n/2^{k+1})$.

- 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

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

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

$$T(n) = n + T(n/2)$$
$$= n + n/2 + T(n/2^2)$$
$$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$$
$$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$$
$$\leq 2n + T(n/2^{k+1}).$$

- 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

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

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

$T(n) = n + T(n/2)$
$= n + n/2 + T(n/2^2)$
$= n + n/2 + \ldots + n/2^k + T(n/2^{k+1})$
$= n \sum_{j \leq k} 1/2^j + T(n/2^{k+1})$
$\leq 2n + T(n/2^{k+1})$.

- 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.

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

Antwort 4

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

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

Antwort 4

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

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

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

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

$= n + 2(n/2) + \underline{2^2}\,T(n/\underline{2^2})$

g

n aufruf

2 · g(n/2)

2·

g(n/2)

2·n/2 aufruf

2·g(n/4)

Antwort 4

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

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

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

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

$= n + 2(n/2) + \ldots + 2^k(n/2^k)$

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

Antwort 4

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

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

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

$k=0$ $k=1$

$$= n + 2(n/2) + \ldots + \cancel{2}^k(n/\cancel{2}^k)$$
$$+ 2^{k+1}T(n/2^{k+1})$$
$$= n\,(k + \underline{1}) + \underline{2^{k+1}\,T(n/2^{k+1})}.$$

Antwort 4

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

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

$$T(n) = n + 2\,T(n/2)$$
$$= n + 2(n/2) + 2^2\,T(n/2^2)$$
$$= n + 2(n/2) + \ldots + 2^k(n/2^k)$$
$$+ 2^{k+1}T(n/2^{k+1})$$
$$= n\,(k+1) + 2^{k+1}\,T(n/2^{k+1}).$$

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

Antwort 4

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

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

$$T(n) = n + 2\,T(n/2)$$
$$= n + 2(n/2) + 2^2\,T(n/2^2)$$
$$= n + 2(n/2) + \ldots + 2^k(n/2^k)$$
$$+\, 2^{k+1}T(n/2^{k+1})$$
$$= n\,(k+1) + 2^{k+1}\,T(n/2^{k+1}).$$

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

Antwort 4

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

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

$T(n) = n + 2\,T(n/2)$
$= n + 2(n/2) + 2^2\,T(n/2^2)$
$= n + 2(n/2) + \ldots + 2^k(n/2^k)$
$+ 2^{k+1}T(n/2^{k+1})$
$= n\,(k+1) + 2^{k+1}\,T(n/2^{k+1})$.

$n/2n = \frac{1}{2}$

$2^k \cdot 2$

$2^{\log(n)} = n$

$\log(n) = k$

$\log_2(n) = k$

- 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

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

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

$$T(n) = n + 2\,T(n/2)$$
$$= n + 2(n/2) + 2^2\,T(n/2^2)$$
$$= n + 2(n/2) + \ldots + 2^k(n/2^k)$$
$$+\,2^{k+1}T(n/2^{k+1})$$
$$= n\,(k+1) + 2^{k+1}\,T(n/2^{k+1}).$$

- 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

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

```python
class MyClass:
  a = 'hello'
  def __init__(self):
    return
```

Übung 4: Classes and Programming Concepts

```python
class MyClass:
  a = 'hello'
  def __init__(self):
    return
```

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

Übung 4: Classes and Programming Concepts

```python
class MyClass:
  a = 'hello'
  def __init__(self):
    return
```

```python
class MyClass:
  def __init__(self):
    self.a = 'hello'
    return
```

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

Übung 4: Classes and Programming Concepts

```python
class MyClass:
  a = 'hello'
  def __init__(self):
    return
```

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

```python
class MyClass:
  def __init__(self):
    self.a = 'hello'
    return
```

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

Übung 4: Classes and Programming Concepts

- Beispiel 1:

```
class MyClass:
  a = 'hello'
  def __init__(self):
    return

first = MyClass()
MyClass.a = 'world'

second = MyClass()
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)
```

Prints: `world`

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

Übung 4: Classes and Programming Concepts

- Beispiel 2:

```python
class MyClass:
  a = 'hello'
  def __init__(self):
    return

first = MyClass()
first.a = 'world'

second = MyClass()
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)
```

Prints: `world hello`

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

Übung 4: Classes and Programming Concepts

- Beispiel 3:

```python
class MyClass:
  def __init__(self):
    self.a = 'hello'

first = MyClass()
first.a = 'world'

second = MyClass()
print(second.a)
```

Übung 4: Classes and Programming Concepts

- Beispiel 3:

```
class MyClass:
  def __init__(self):
    self.a = 'hello'

first = MyClass()
first.a = 'world'

second = MyClass()
print(second.a)
```

Prints: `hello`

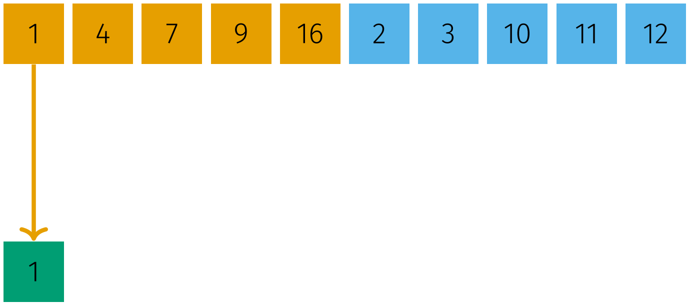
`self.a` bezieht sich auf ein *Instanzattribut* und wird **nicht** zwischen verschiedenen Instanzen geteilt.

6. Divide & Conquer Sortieralgorithmen

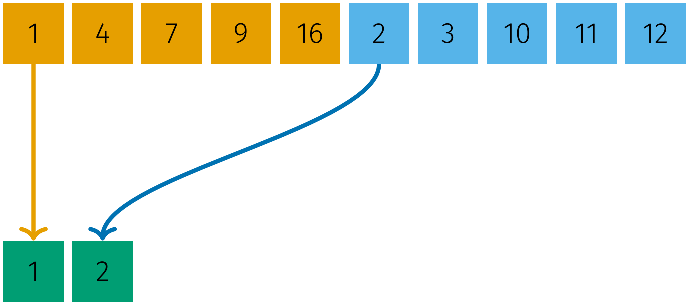
Merge

1	4	7	9	16	2	3	10	11	12

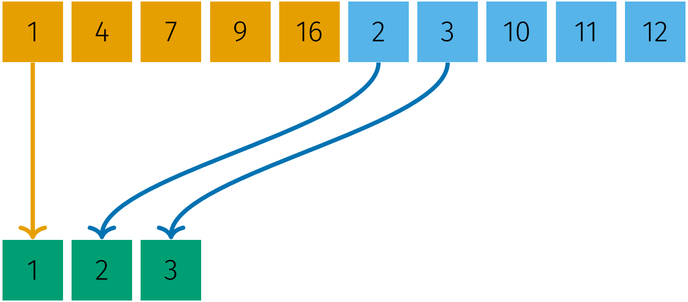
Merge



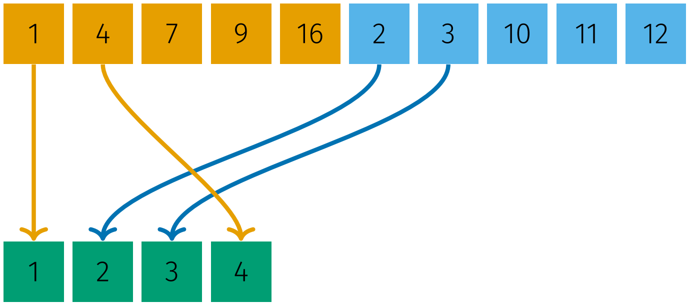
Merge



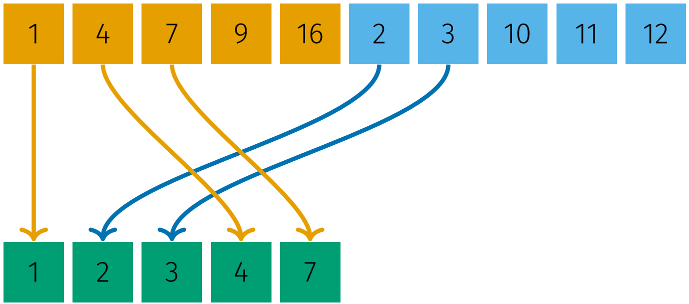
Merge



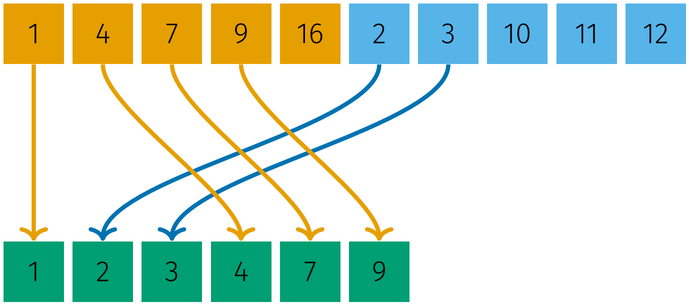
Merge



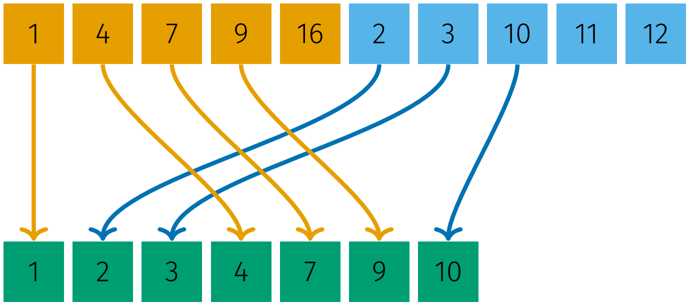
Merge



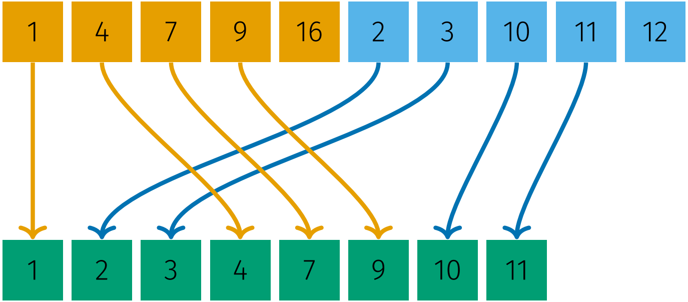
Merge



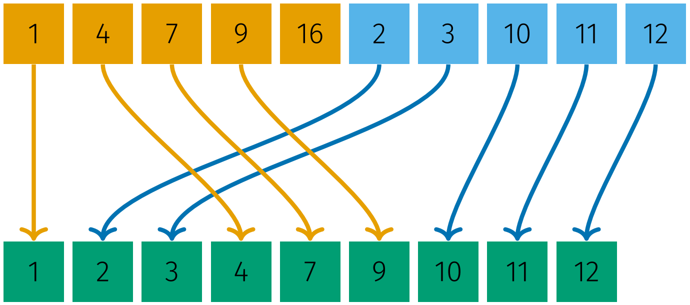
Merge



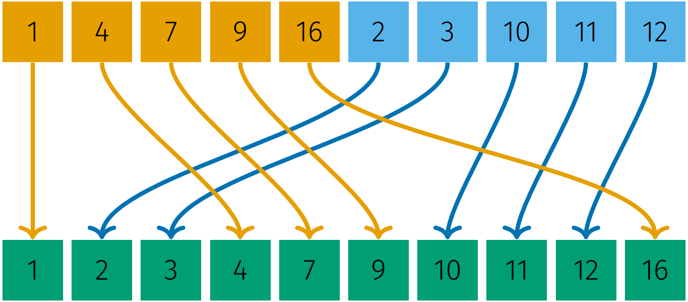
Merge



Merge



Merge

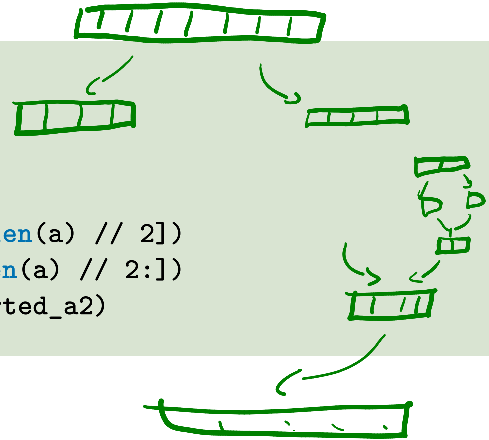


Algorithmus `merge`

```python
def merge(a1, a2):
  b, i, j = [], 0, 0
  while i < len(a1) and j < len(a2):
    if a1[i] < a2[j]:
      b.append(a1[i])
      i += 1
    else:
      b.append(a2[j])
      j += 1
  b += a1[i:]
  b += a2[j:]
  return b
```

Algorithmus `merge_sort`

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



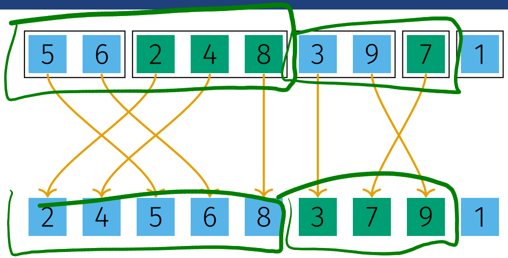
Natürlicher 2-Wege Mergesort

5	6	2	4	8	3	9	7	1

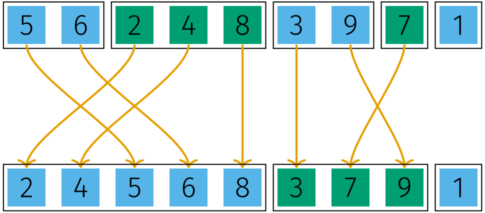
Natürlicher 2-Wege Mergesort

5	6	2	4	8	3	9	7	1

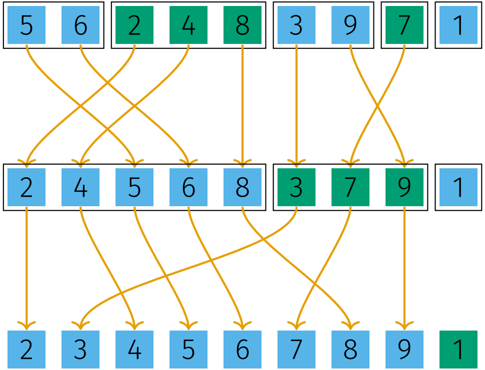
Natürlicher 2-Wege Mergesort



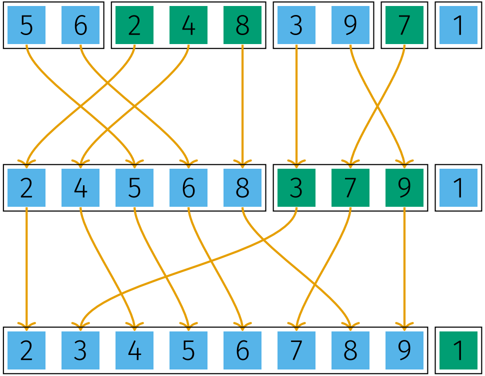
Natürlicher 2-Wege Mergesort



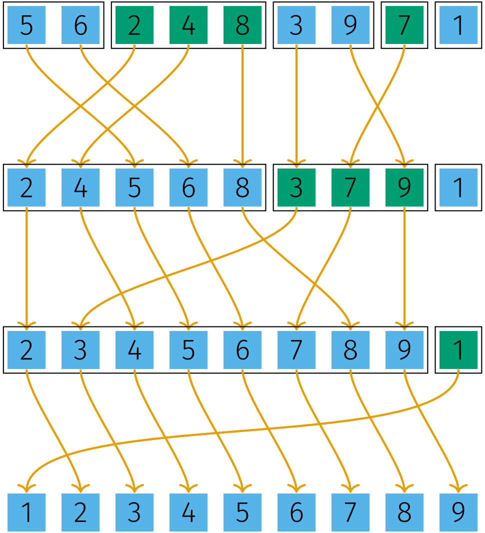
Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Algorithmus `NaturalMergesort`(A)

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

```
def NaturalMergesort(A):
    l = -1
    while l = 0: #eigentlich != aber es geht hier nicht
        r = 0
        while r < len(A)-1:
            l = r
            m = l
            while m < len(A)-1 and A[m+1] >= A[m]:
                m = m+1
            if m < len(A)-1:
                r = m+1
                while r < len(A)-1 and A[r+1] >= A[r]:
                    r = r+1
                Merge(A,l,m,r)
            else:
                r = len(A)-1
            r+=1
```

7. Prüfungsaufgaben

Sorting Algorithm

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

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

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

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

vor der i-ten iteration

ist li[:i]

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.*

○ M I S ○ M S I ☒ I M S ○ I S M ○ S M I ○ S I M

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?*

○ $O(1)$ ○ $O(\log n)$ ○ $O(n)$ ○ $O(n \log n)$ ☒ $O\left(n^2\right)$

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?