

Algorithmen 1

Sortieren: Mergesort, Quicksort, untere Schranke



Anmerkung: untere Laufzeitschranken

Der Algorithmus benötigt $\Omega(n^2)$ Schritte – Was bedeutet das?

- Interpretation 1: auf allen Eingaben ist die Laufzeit mindestens quadratisch
- Interpretation 2: es gibt Eingaben, sodass die Laufzeit mindestens quadratisch ist

Anmerkung: untere Laufzeitschranken

Der Algorithmus benötigt $\Omega(n^2)$ Schritte – Was bedeutet das?

- Interpretation 1: auf allen Eingaben ist die Laufzeit mindestens quadratisch
- Interpretation 2: es gibt Eingaben, sodass die Laufzeit mindestens quadratisch ist

Was gilt für die Vorlesung?

- außer explizit angegeben sprechen wir immer über den Worst Case
- also: Interpretation 2

Anmerkung: untere Laufzeitschranken

Der Algorithmus benötigt $\Omega(n^2)$ Schritte – Was bedeutet das?

- Interpretation 1: auf allen Eingaben ist die Laufzeit mindestens quadratisch
- Interpretation 2: es gibt Eingaben, sodass die Laufzeit mindestens quadratisch ist

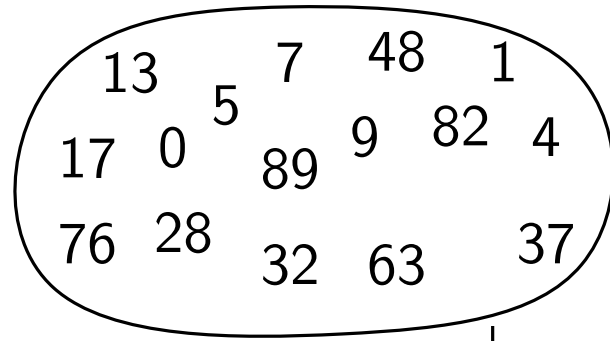
Was gilt für die Vorlesung?

- außer explizit angegeben sprechen wir immer über den Worst Case
- also: Interpretation 2

Warum machen wir das so?

- Interpretation 1 betrachtet den Best Case → der ist selten relevant
- Interpretation 2 nützlich in verschiedenen Situationen:
 - „jeder Algo hat Laufzeit $\Omega(n^2)$ “ → es gibt keinen sub-quadratischen Algo
 - „ein konkreter Algo hat Laufzeit $\Omega(n^2)$ und $O(n^3)$ “ → die Laufzeit ist im Worst Case zwischen quadratisch und kubisch

Sortieren



Gegeben

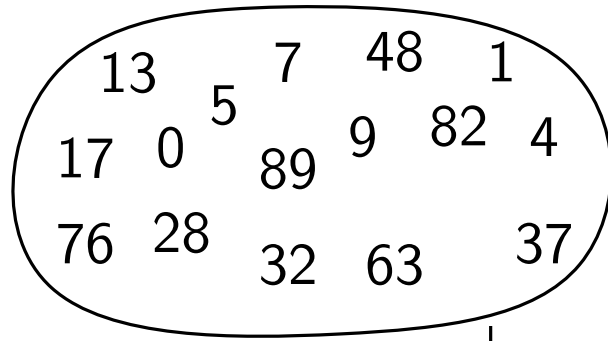
- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

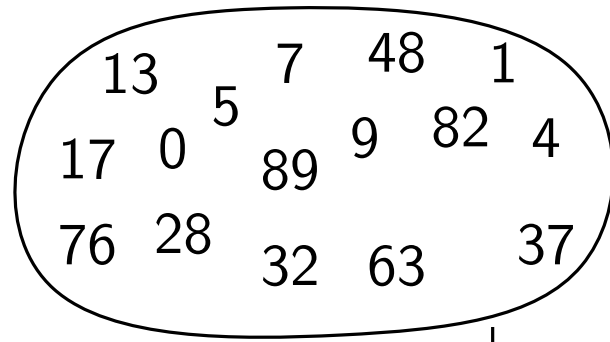
- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟶ $\langle 0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89 \rangle$

Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

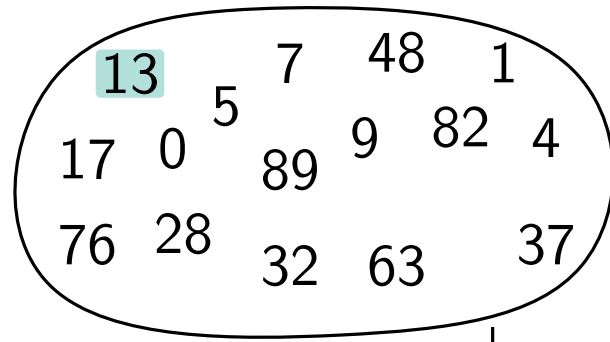
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

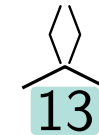
- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

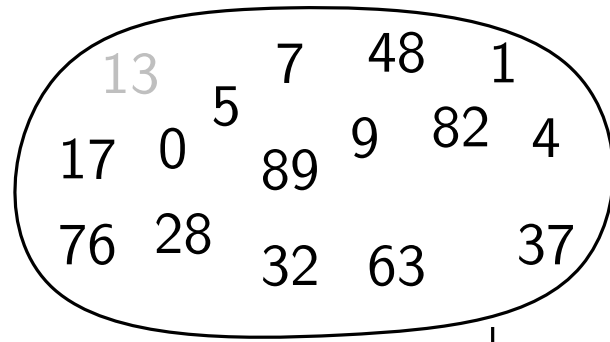
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:



Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

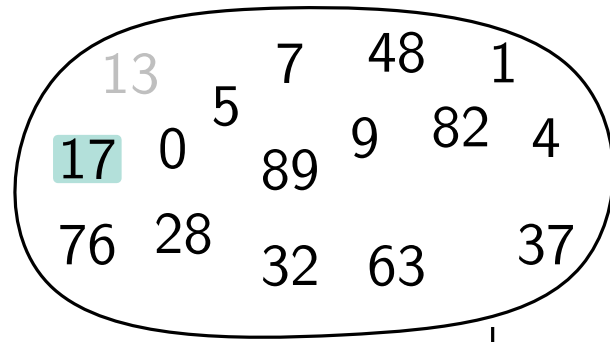
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨13⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

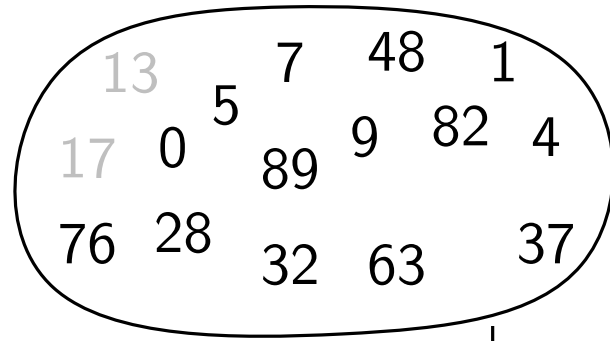
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨13⟩
 17

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

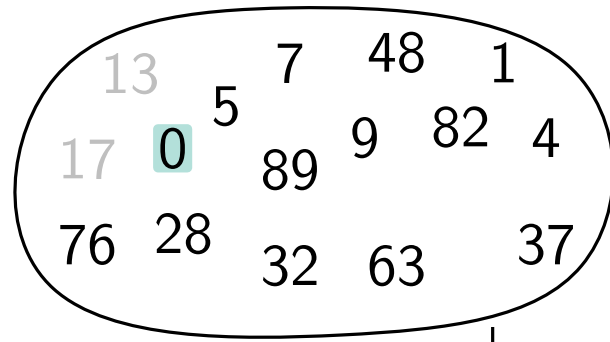
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨13, 17⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

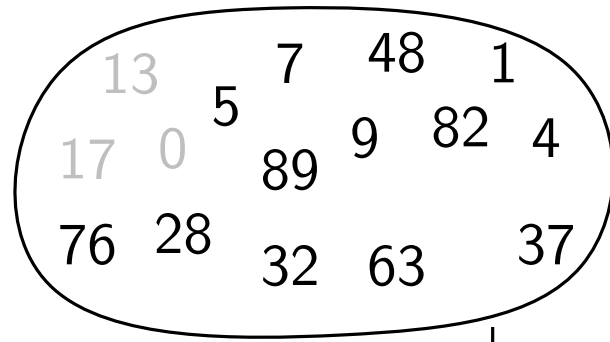
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨13, 17⟩
 0

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

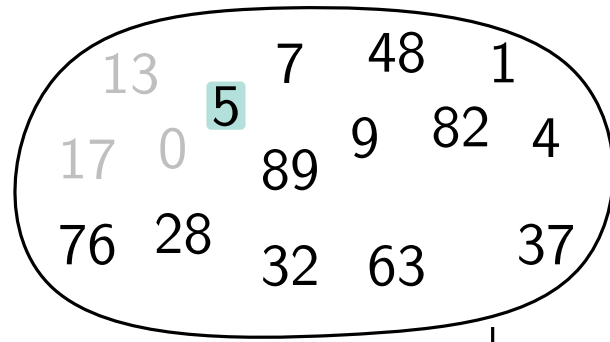
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 13, 17⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

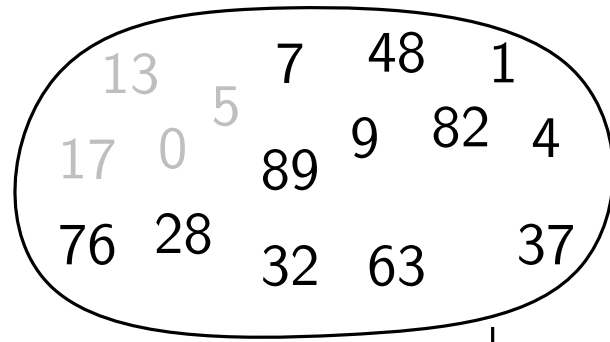
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 13, 17⟩
 5

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

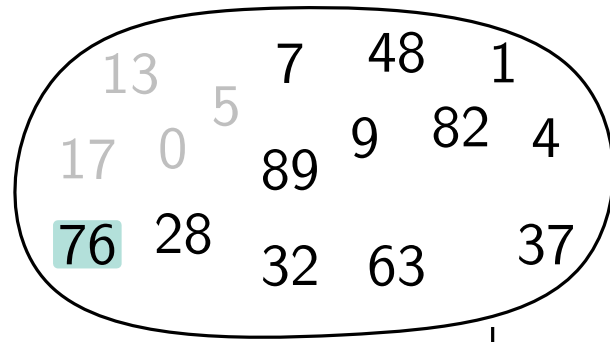
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 5, 13, 17⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

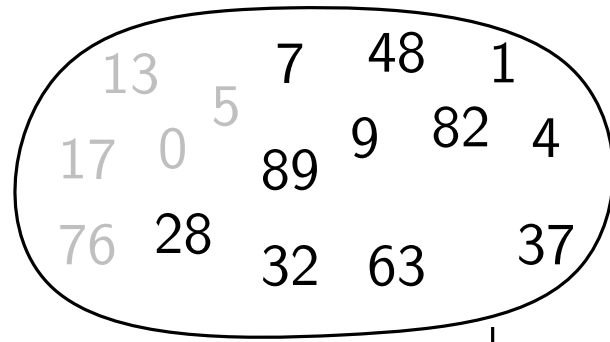
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 5, 13, 17⟩
76

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

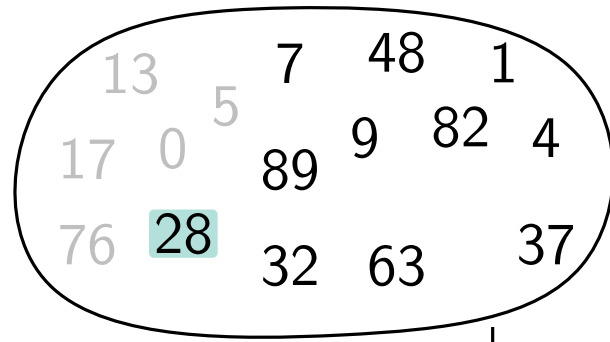
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 76⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

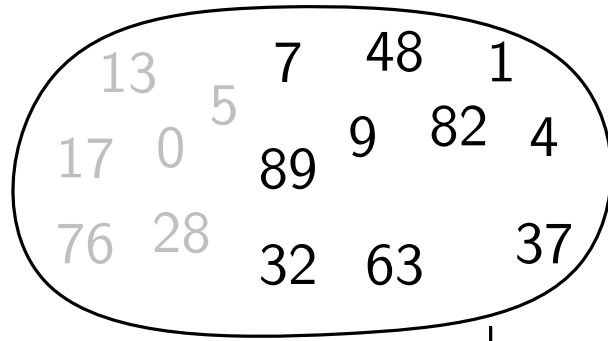
- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 76⟩

28

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

$\langle 0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89 \rangle$

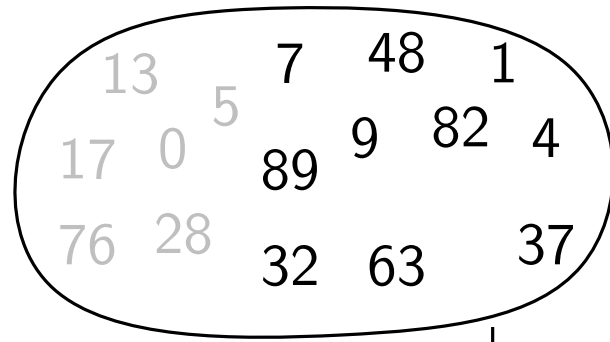
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert

bisher sortierte Teilfolge:

$\langle 0, 5, 13, 17, 28, 76 \rangle$

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

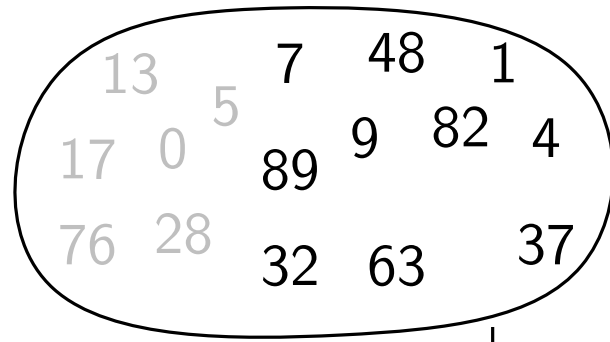
Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:
 - suchen in Folge der Länge $i - 1$
 - einfügen in Folge der Länge $i - 1$

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

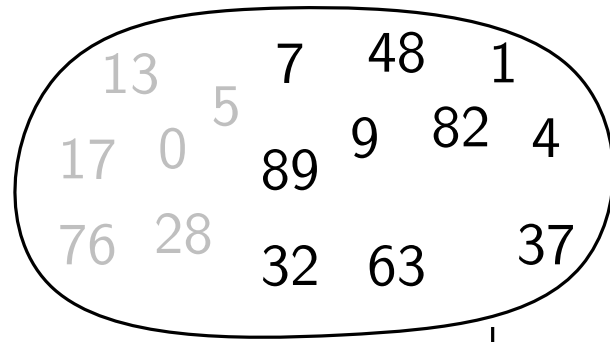
- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

- | | |
|---------------------------------------|---------------------------|
| ■ suchen in Folge der Länge $i - 1$ | binäre Suche: $O(\log n)$ |
| ■ einfügen in Folge der Länge $i - 1$ | einfügen: $O(1)$ |

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

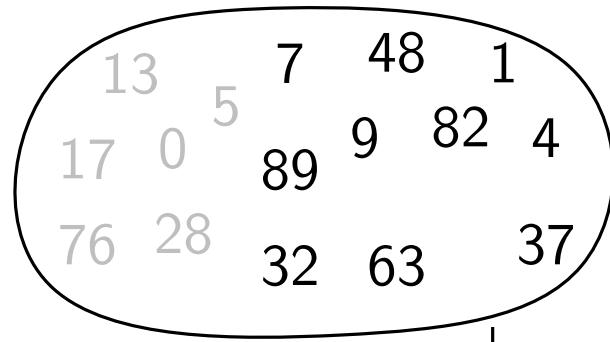
- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

■ suchen in Folge der Länge $i - 1$	binäre Suche: $O(\log n)$	}	$O(\log n)$ pro Element $\Rightarrow O(n \log n)$ gesamt
■ einfügen in Folge der Länge $i - 1$	einfügen: $O(1)$		

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:
 - suchen in Folge der Länge $i - 1$
 - einfügen in Folge der Länge $i - 1$

bisher sortierte Teilfolge:

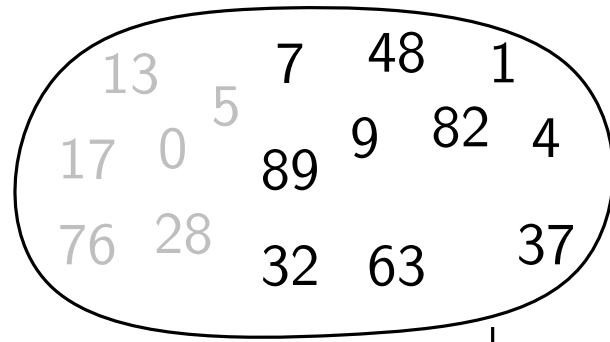
⟨0, 5, 13, 17, 28, 76⟩

für Arrays (random access)

binäre Suche: $O(\log n)$ } $O(\log n)$ pro Element
 einfügen: $O(1)$ } $\Rightarrow O(n \log n)$ gesamt

für Listen

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort


- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:
 - suchen in Folge der Länge $i - 1$
 - einfügen in Folge der Länge $i - 1$

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

für Arrays (random access)

binäre Suche: $O(\log n)$	} $O(\log n)$ pro Element
einfügen: $O(1)$	

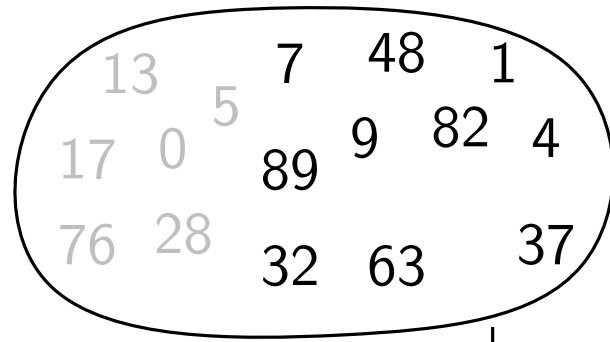


für Listen

Problem

- wir kennen noch keine Datenstruktur, mit der wir schnell suchen **und** einfügen können

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

- suchen in Folge der Länge $i - 1$
- einfügen in Folge der Länge $i - 1$

Liste

Array

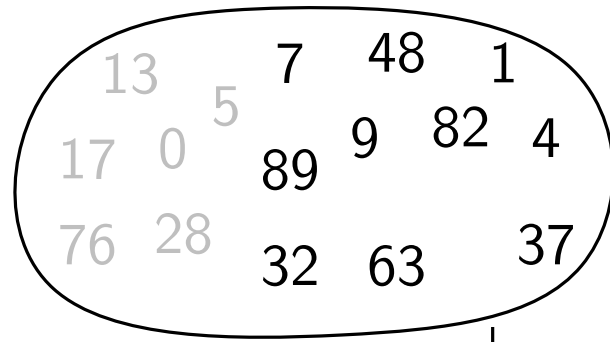
$\Theta(i)$

$\Theta(\log i)$

$\Theta(1)$

$\Theta(i)$

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

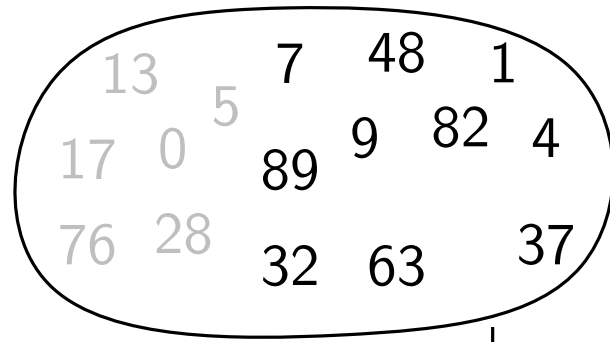
- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

	Liste	Array
■ suchen in Folge der Länge $i - 1$	$\Theta(i)$	$\Theta(\log i)$
■ einfügen in Folge der Länge $i - 1$	$\Theta(1)$	$\Theta(i)$
Kosten in Schritt i :	$\Theta(i)$	$\Theta(i)$

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

- suchen in Folge der Länge $i - 1$
- einfügen in Folge der Länge $i - 1$

Liste

Array

$\Theta(i)$

$\Theta(\log i)$

$\Theta(1)$

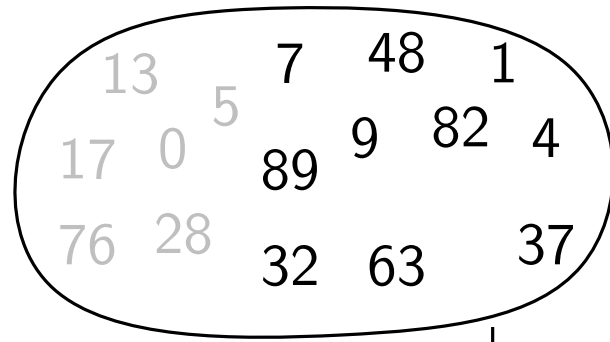
$\Theta(i)$

Kosten in Schritt i : $\Theta(i)$

$\Theta(i)$

- Laufzeit: $\sum_{i=1}^n \Theta(i) = \Theta(n^2)$

Sortieren



Gegeben

- n Elemente aus einer geordneten Menge (z.B. Zahlen)

Gesucht

- sortierte Folge dieser Elemente (z.B. als Array oder Liste)

⟨0, 1, 4, 5, 7, 9, 13, 17, 28, 32, 37, 48, 63, 76, 82, 89⟩

Insertion Sort

- füge Elemente nach und nach ein
- halte Folge dabei sortiert
- Bearbeitung des i ten Elements:

bisher sortierte Teilfolge:

⟨0, 5, 13, 17, 28, 76⟩

- suchen in Folge der Länge $i - 1$
- einfügen in Folge der Länge $i - 1$

Liste

Array

$\Theta(i)$

$\Theta(\log i)$

$\Theta(1)$

$\Theta(i)$

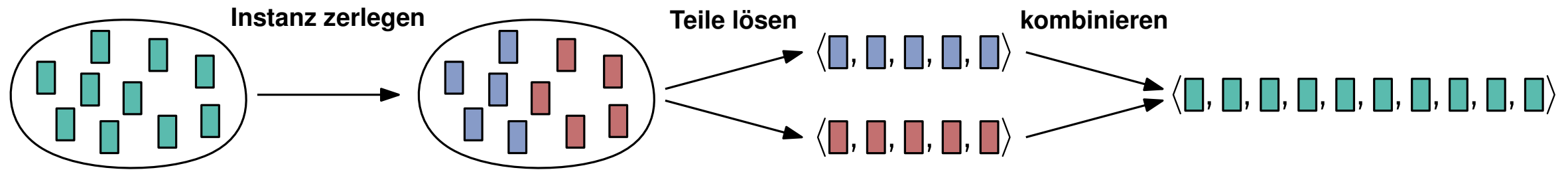
Kosten in Schritt i : $\Theta(i)$

$\Theta(i)$

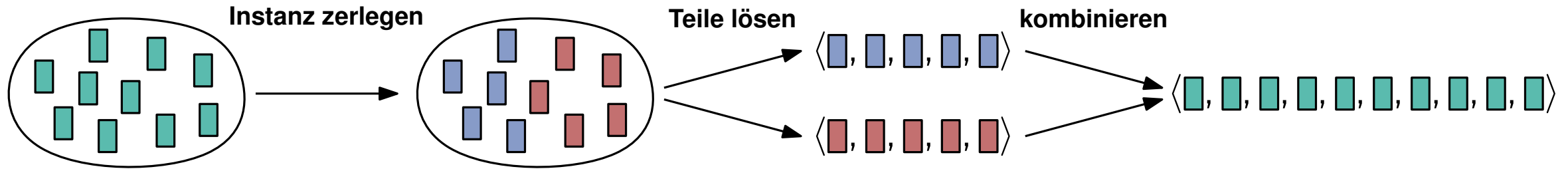
- Laufzeit: $\sum_{i=1}^n \Theta(i) = \Theta(n^2)$

heute: zwei andere Algorithmen mit $\Theta(n \log n)$ Laufzeit

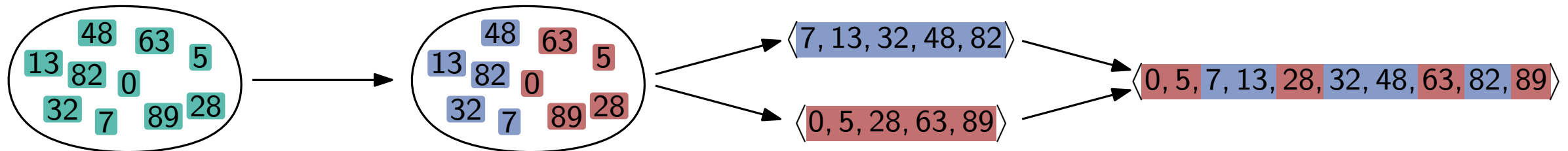
Sortieren mittels Teile und Herrsche



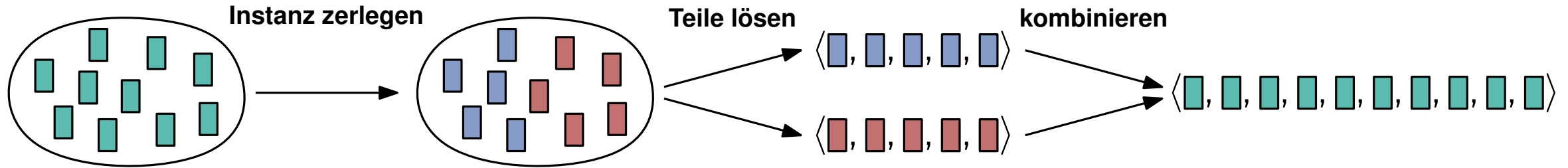
Sortieren mittels Teile und Herrsche



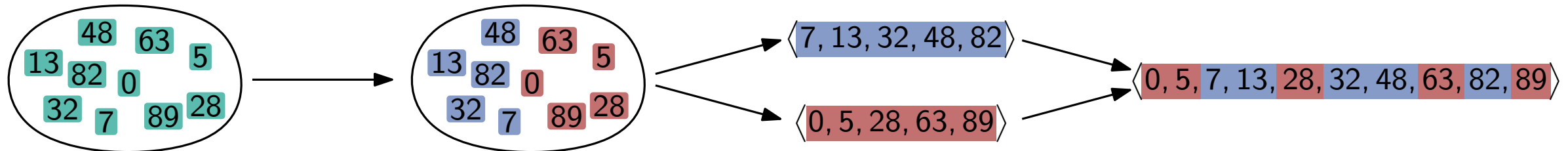
Mergesort: arbeite beim Zusammenfügen



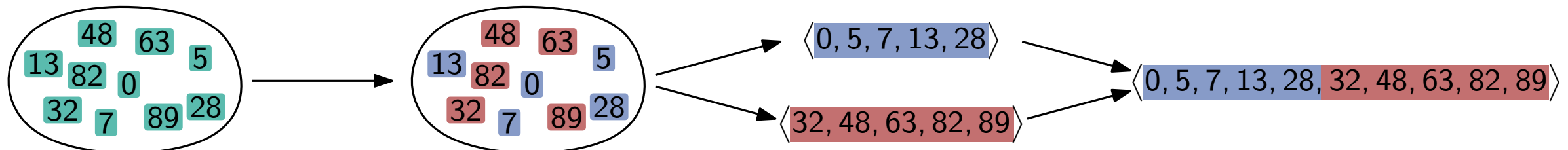
Sortieren mittels Teile und Herrsche



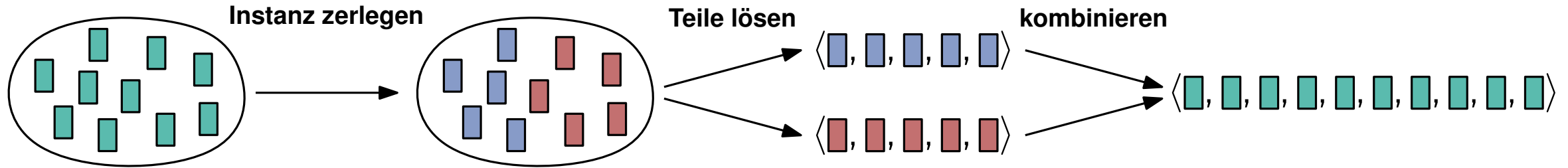
Mergesort: arbeite beim Zusammenfügen



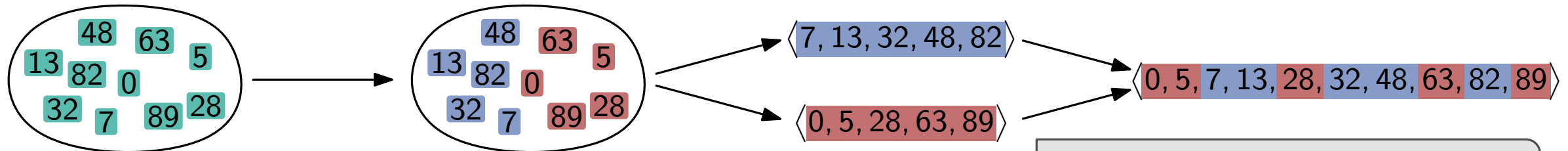
Quicksort: arbeite beim Zerlegen



Sortieren mittels Teile und Herrsche



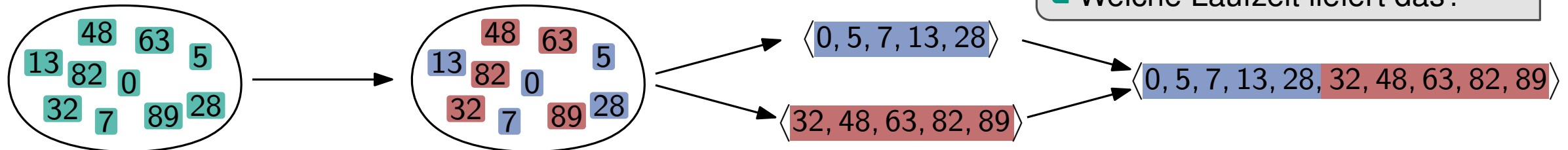
Mergesort: arbeite beim Zusammenfügen



Zwei offene Fragen

- Wie setzt man das im Detail um?
- Welche Laufzeit liefert das?

Quicksort: arbeite beim Zerlegen



Mergesort im Detail

mergesort(*Array A*)

13	48	82	7	32	63	89	0	28	5
----	----	----	---	----	----	----	---	----	---

// base case: small array

if *A.size()* \leq 1 **then return** *A*

Mergesort im Detail

mergesort(Array A)

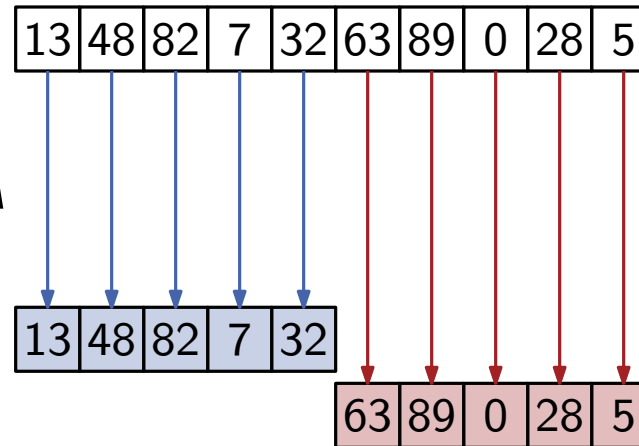
// base case: small array

if A.size() ≤ 1 **then return** A

// partition instance

B := first half of A

C := second half of A



Mergesort im Detail

mergesort(Array A)

// base case: small array

if A.size() ≤ 1 **then return** A

// partition instance

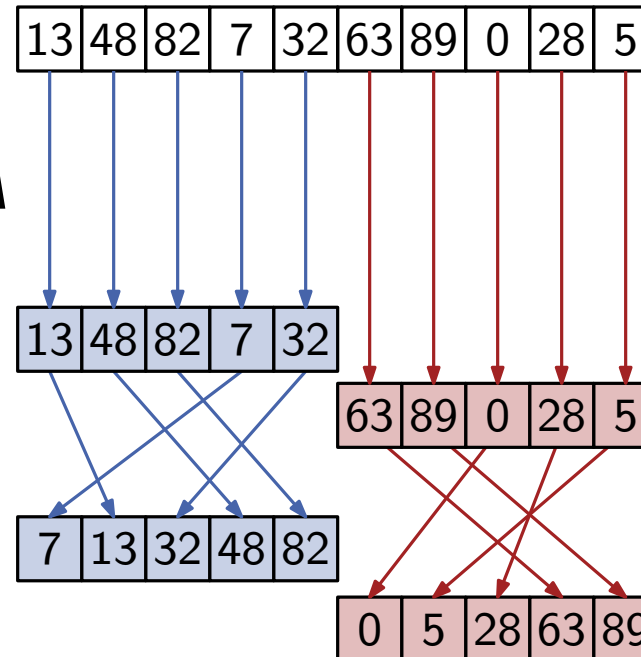
B := first half of A

C := second half of A

// solve parts

B := **mergesort**(B)

C := **mergesort**(C)

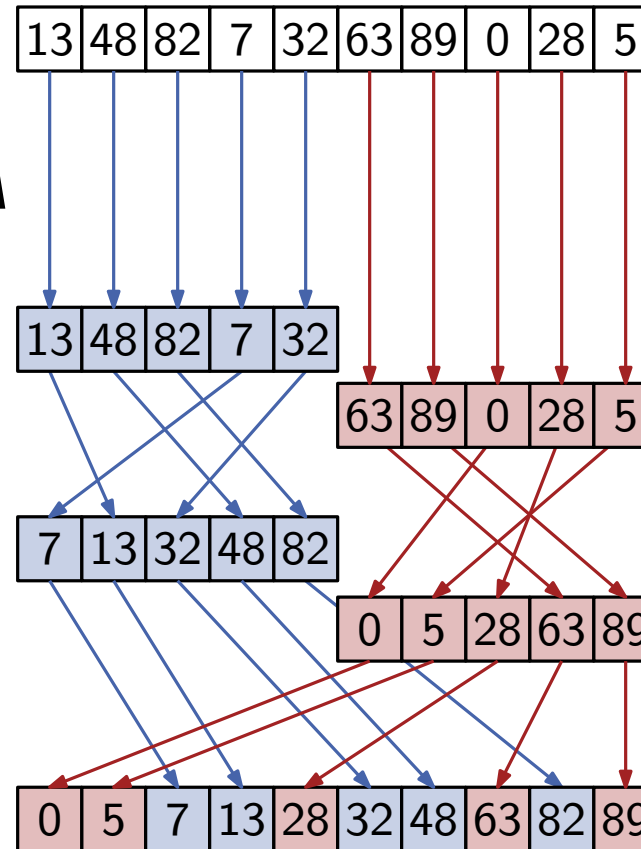


Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)
  
```



merge(Array B, Array C)

```

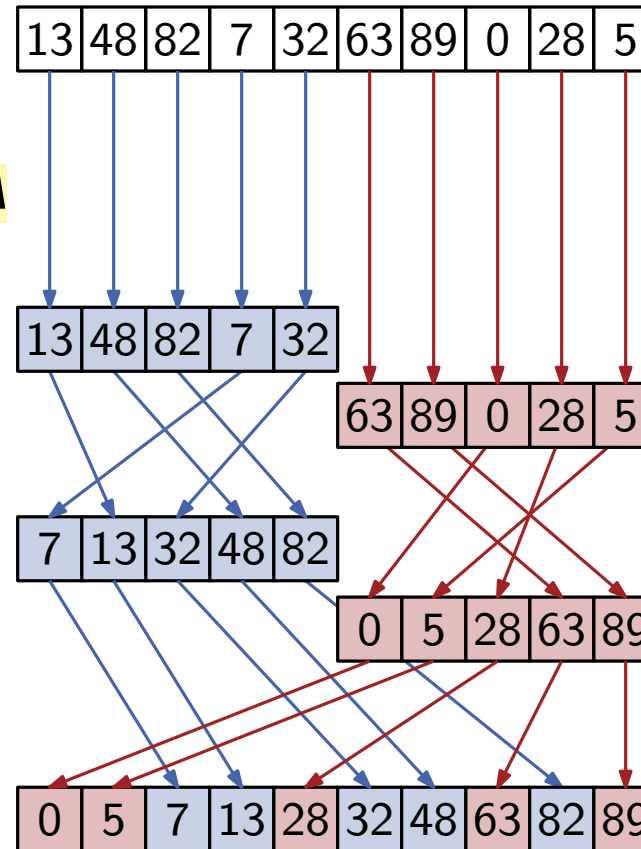
// input: sorted arrays B and C; output: B ∪ C in sorted order
  
```

Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)
  
```



Korrektheit

- Induktion über A.size()
- Anfang: passt für A.size() ≤ 1

merge(Array B, Array C)

```

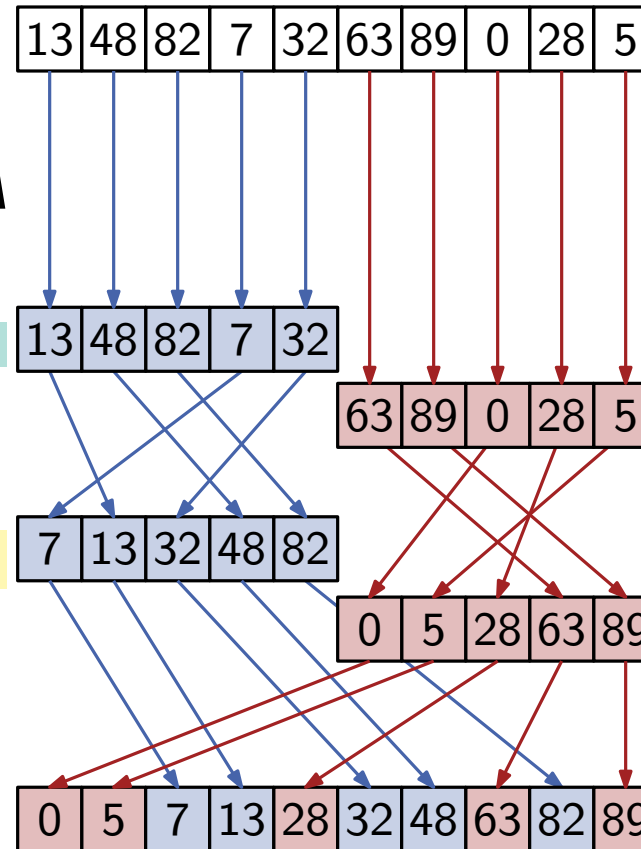
// input: sorted arrays B and C; output: B ∪ C in sorted order
  
```

Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)
  
```



Korrektheit

- Induktion über $A.size()$
- Anfang: passt für $A.size() \leq 1$
- Induktionsschritt:
 - $B.size() < A.size()$
 - Induktionsvoraussetzung \Rightarrow mergesort(B) sortiert B

merge(Array B, Array C)

```

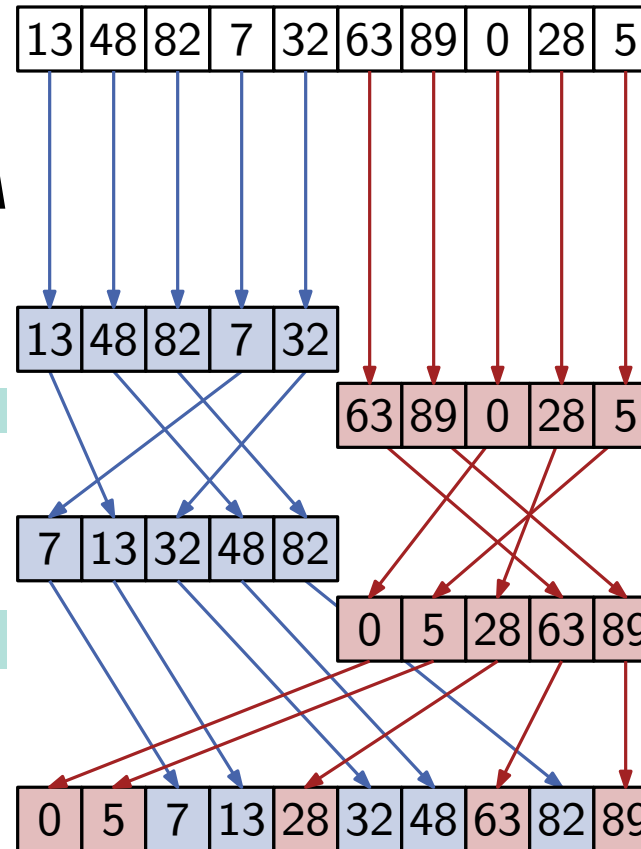
// input: sorted arrays B and C; output: B ∪ C in sorted order
  
```

Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)
  
```



Korrektheit

- Induktion über $A.size()$
- Anfang: passt für $A.size() \leq 1$
- Induktionsschritt:
 - $B.size() < A.size()$
 - Induktionsvoraussetzung \Rightarrow **mergesort(B)** sortiert B
- analog: **mergesort(C)** sortiert C

merge(Array B, Array C)

```

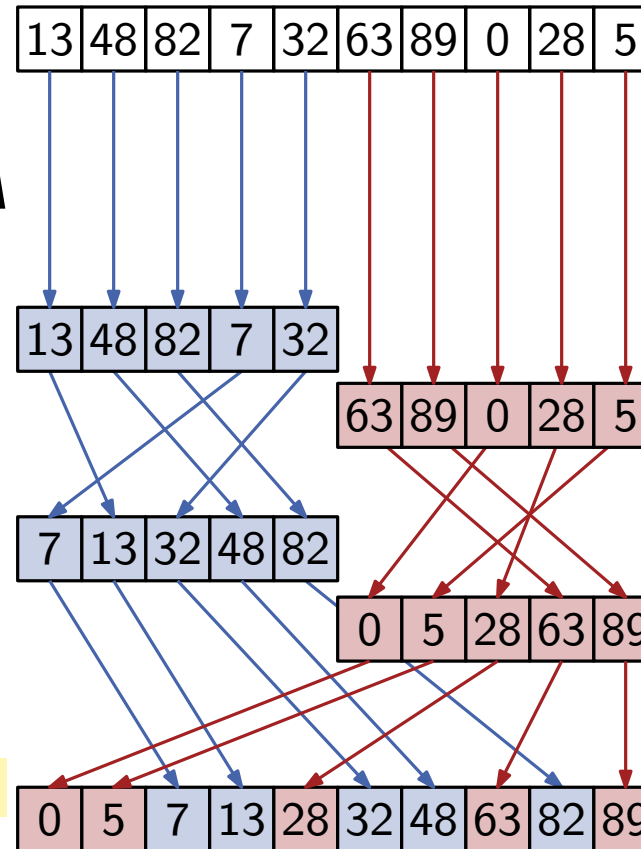
// input: sorted arrays B and C; output: B ∪ C in sorted order
  
```

Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)
  
```



Korrektheit

- Induktion über $A.size()$
- Anfang: passt für $A.size() \leq 1$
- Induktionsschritt:
 - $B.size() < A.size()$
 - Induktionsvoraussetzung \Rightarrow **mergesort**(B) sortiert B
 - analog: **mergesort**(C) sortiert C
- daher: **merge**(B, C) liefert A in sortierter Form
(vorausgesetzt **merge** ist korrekt)

merge(Array B, Array C)

```

// input: sorted arrays B and C; output: B ∪ C in sorted order
  
```

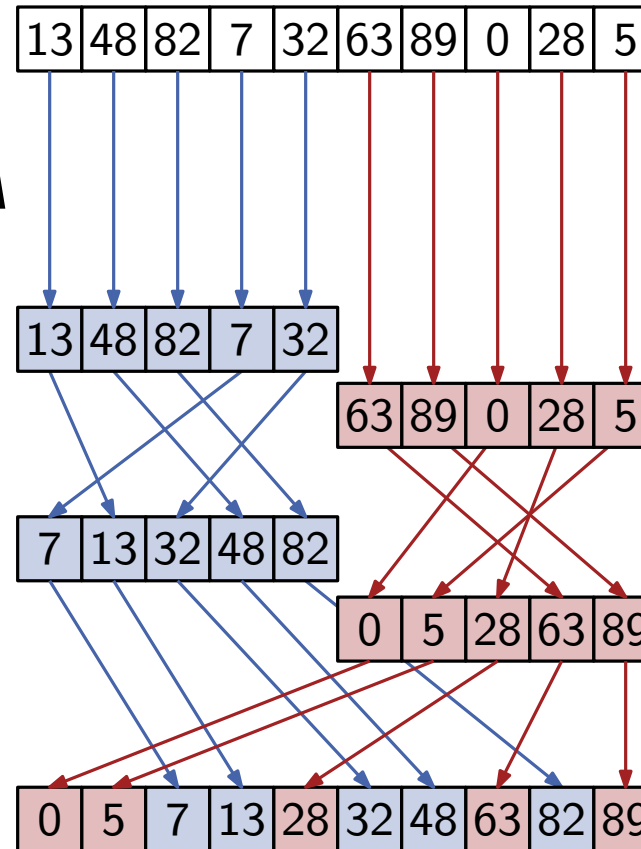

Mergesort im Detail

mergesort(Array A)

```

// base case: small array
if A.size() ≤ 1 then return A
// partition instance
B := first half of A
C := second half of A
// solve parts
B := mergesort(B)
C := mergesort(C)
// combine solutions
return merge(B, C)

```



Korrektheit

- Induktion über $A.size()$
- Anfang: passt für $A.size() \leq 1$
- Induktionsschritt:
 - $B.size() < A.size()$
 - Induktionsvoraussetzung \Rightarrow **mergesort**(B) sortiert B
 - analog: **mergesort**(C) sortiert C
 - daher: **merge**(B, C) liefert A in sortierter Form
(vorausgesetzt **merge** ist korrekt)

merge(Array B, Array C)

```

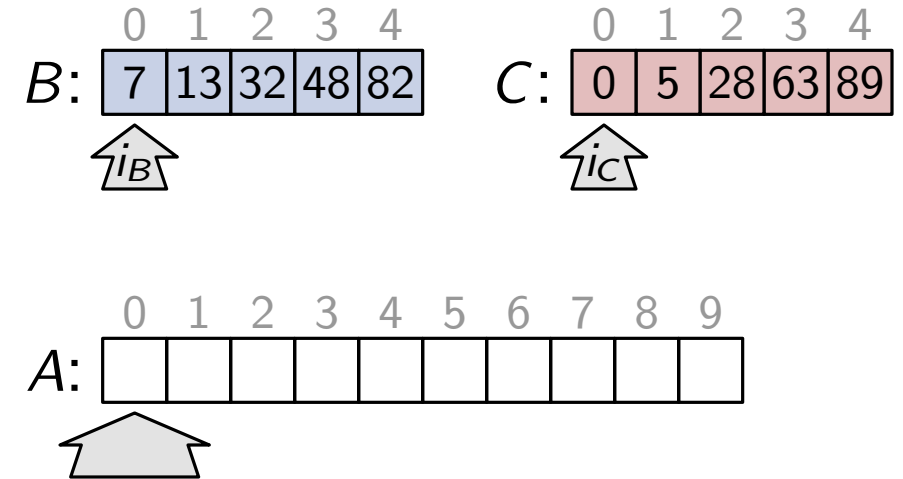
// input: sorted arrays B and C; output: B ∪ C in sorted order

```

Wie funktioniert **merge**?

Plan

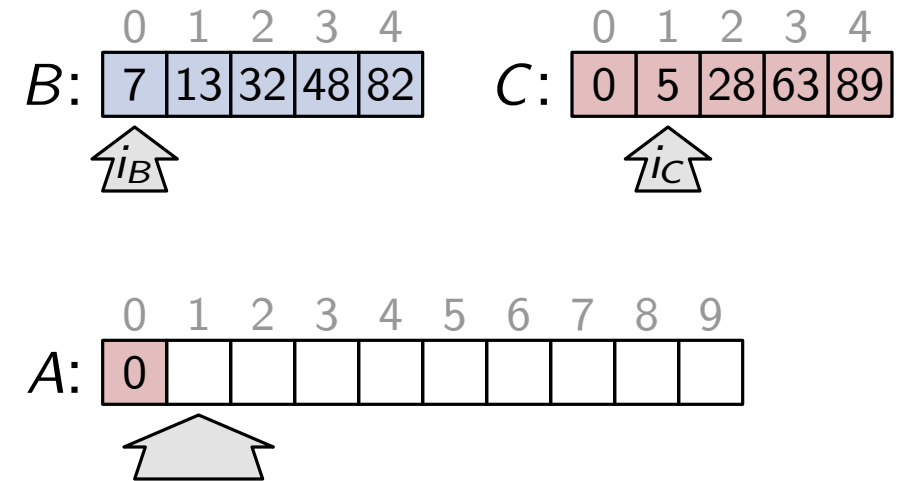
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

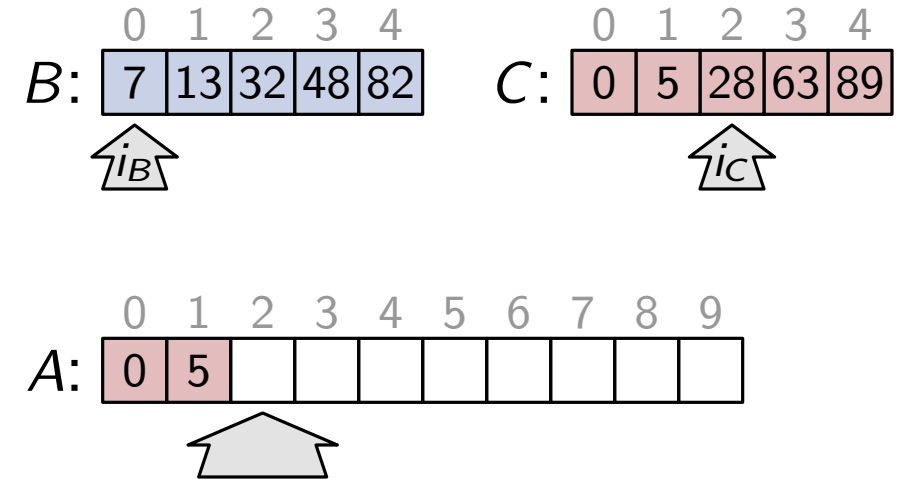
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

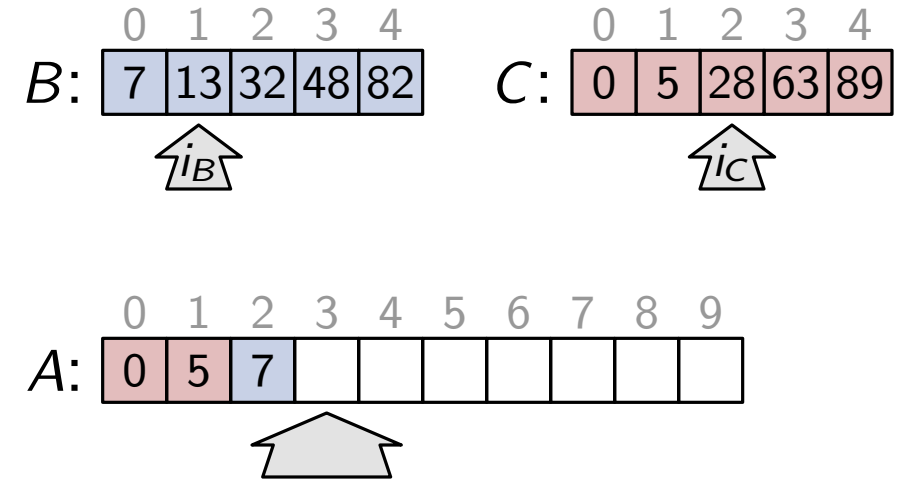
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

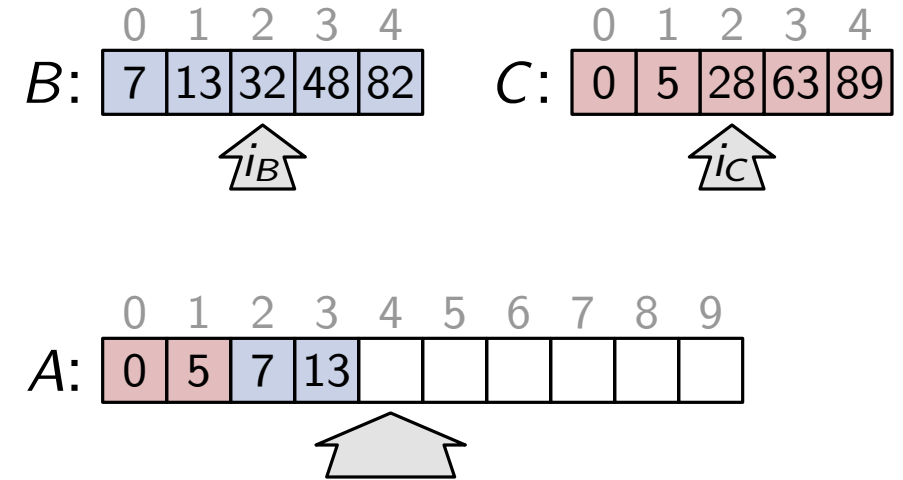
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

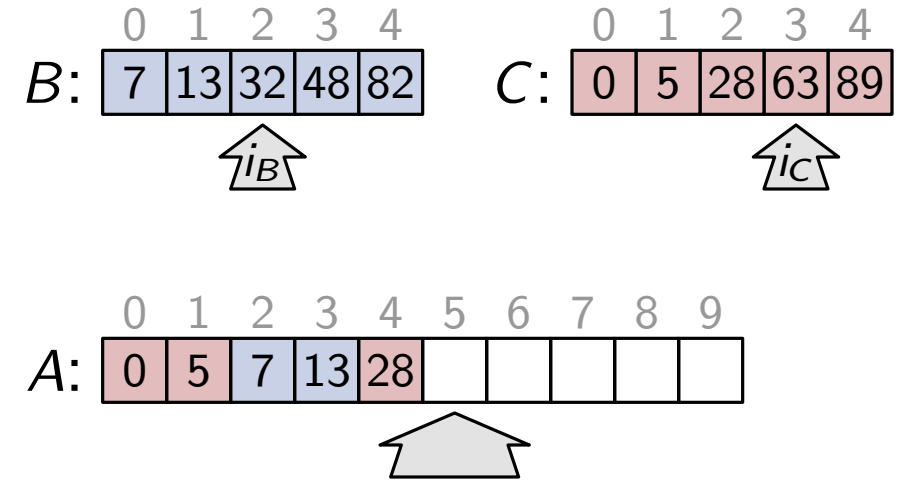
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

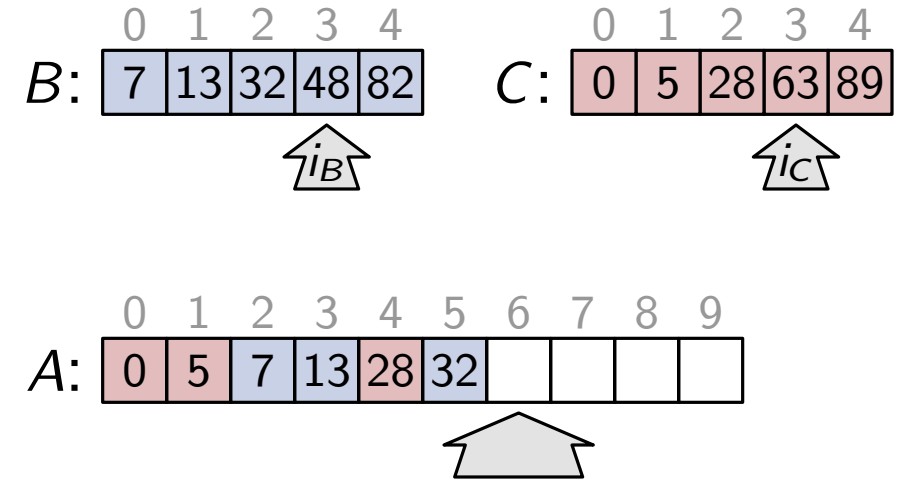
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

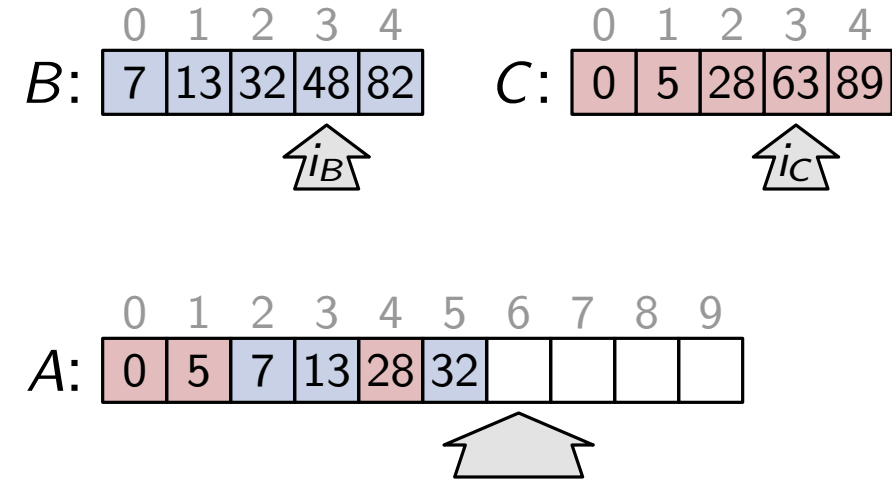
- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Wie funktioniert **merge**?

Plan

- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



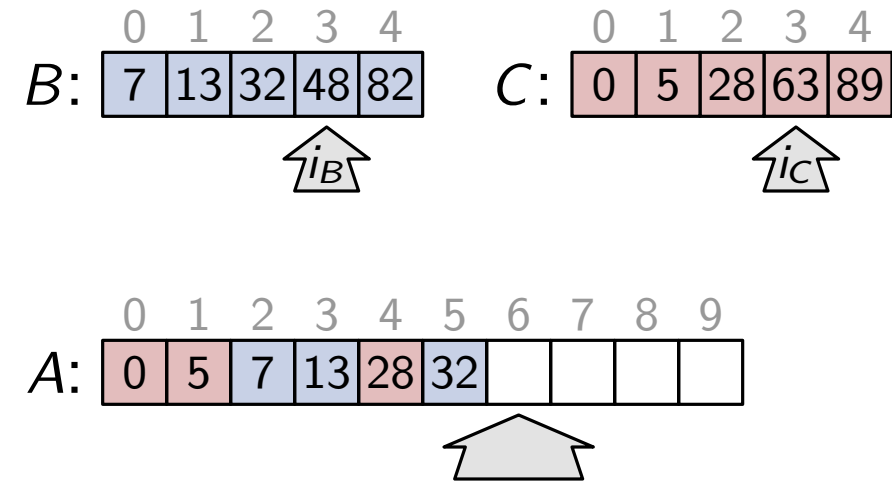
Invariante (nach jedem Einfügen)

- genau die Elemente vor i_B aus B und vor i_C aus C wurden schon in A eingefügt
- alle eingefügten Elemente sind kleiner als die nicht eingefügten Elemente
- bisher in A eingefügte Elemente sind sortiert

Wie funktioniert **merge**?

Plan

- erstelle neues Array A für das Ergebnis
- füge iterativ das kleinste noch nicht eingefügte Element in A ein
- für B, C : kenne Positionen i_B, i_C ab der die noch nicht eingefügten Elemente kommen



Invariante (nach jedem Einfügen)

- genau die Elemente vor i_B aus B und vor i_C aus C wurden schon in A eingefügt
- alle eingefügten Elemente sind kleiner als die nicht eingefügten Elemente
- bisher in A eingefügte Elemente sind sortiert

Bleibt die Invariante in jedem Schritt erhalten?

(beachte: sie gilt am Anfang)

- B und C sortiert \Rightarrow kleinstes nicht eingefügtes Element ist $B[i_B]$ oder $C[i_C]$
- damit: $\min\{B[i_B], C[i_C]\}$ in A einfügen (und i_B bzw. i_C erhöhen) erhält Invariante

Merge im Detail

merge(Array B , Array C)

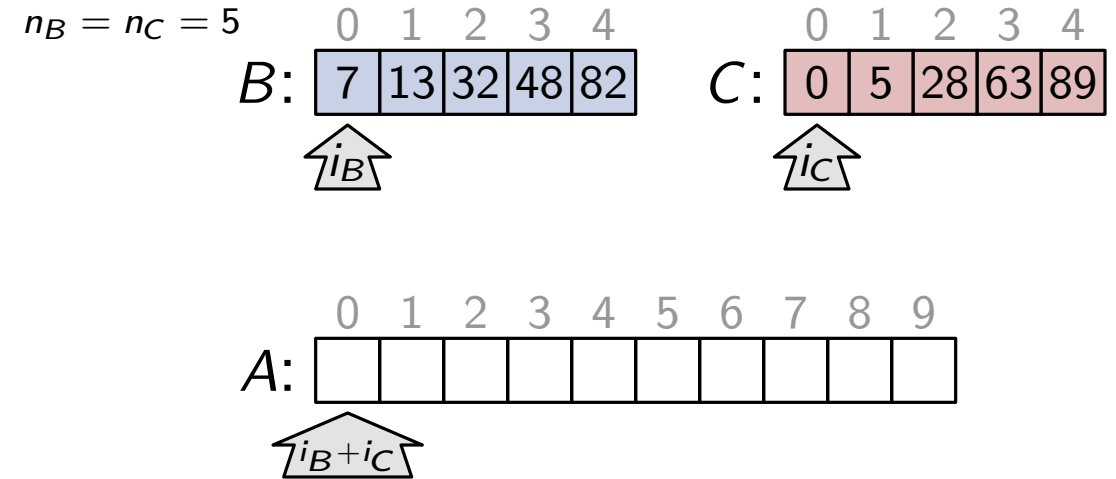
$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

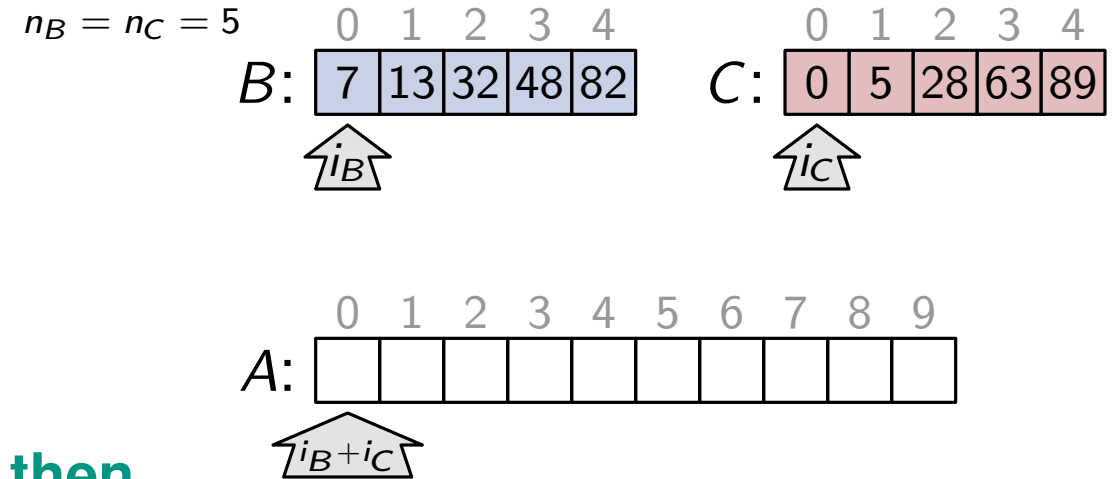
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

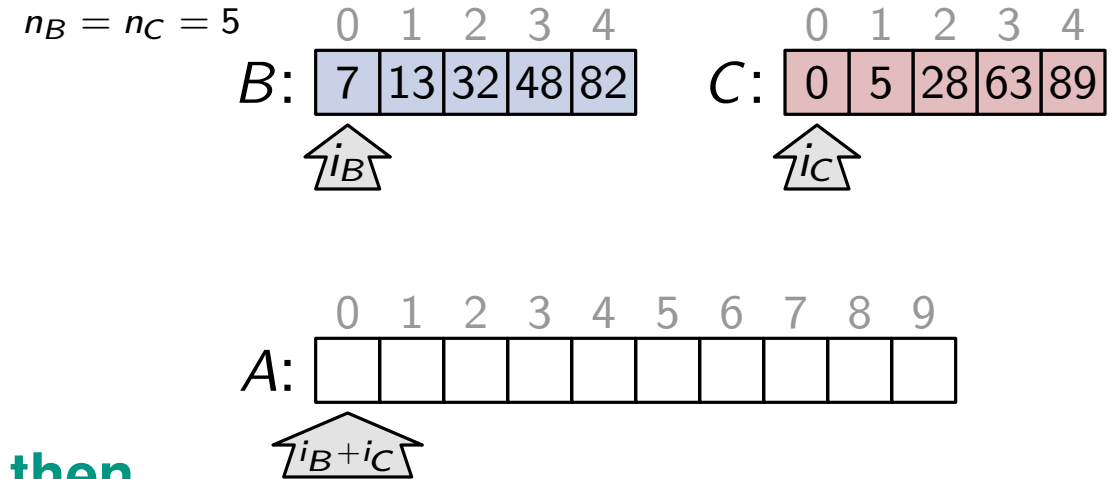
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

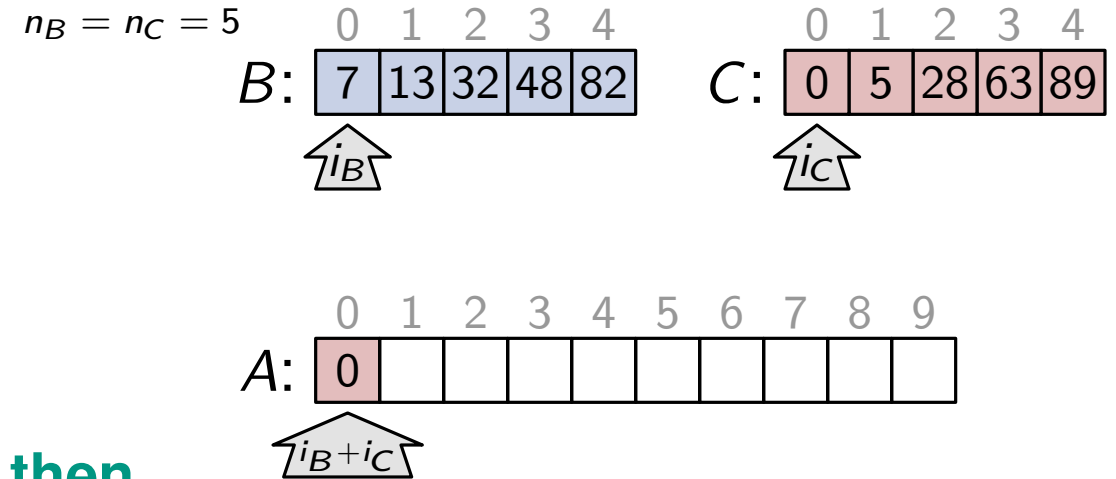
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

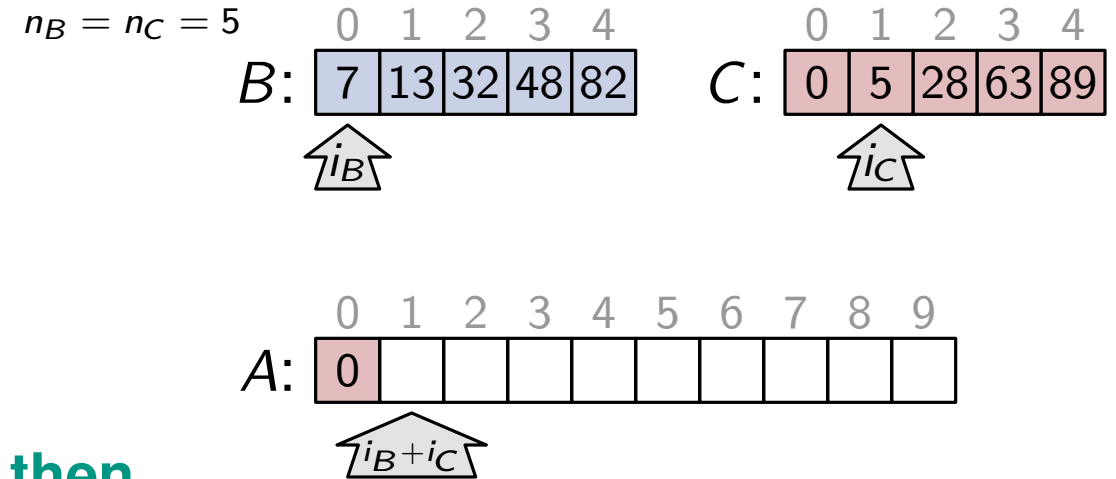
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

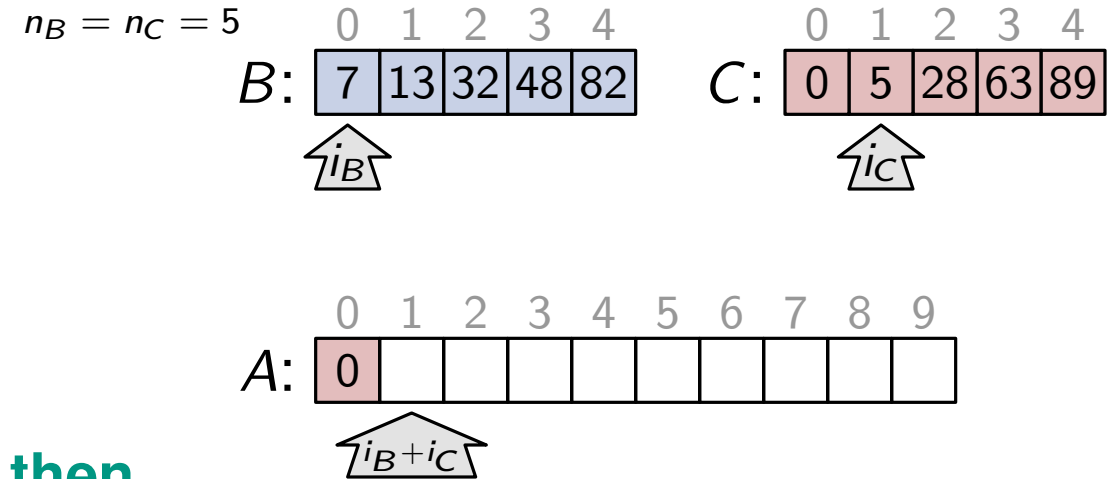
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

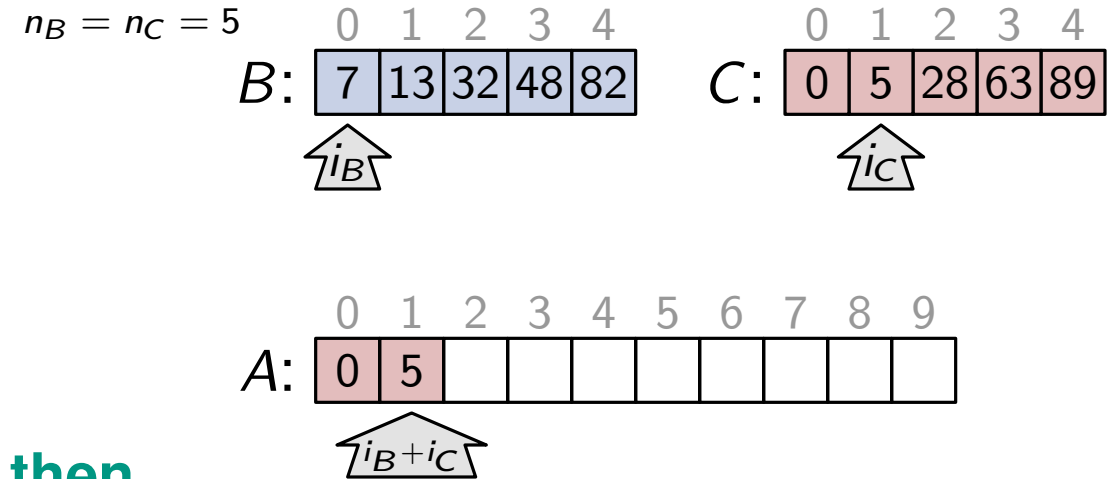
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

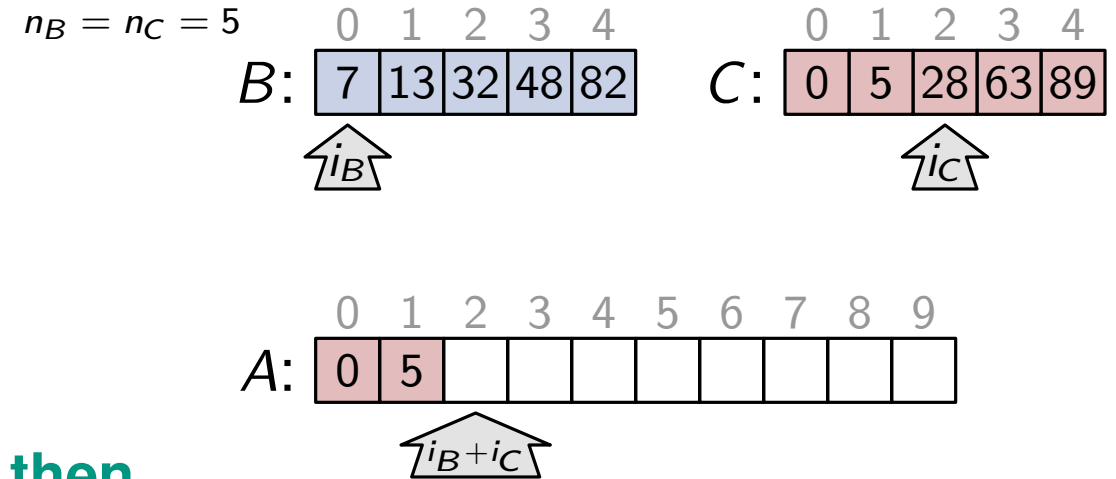
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

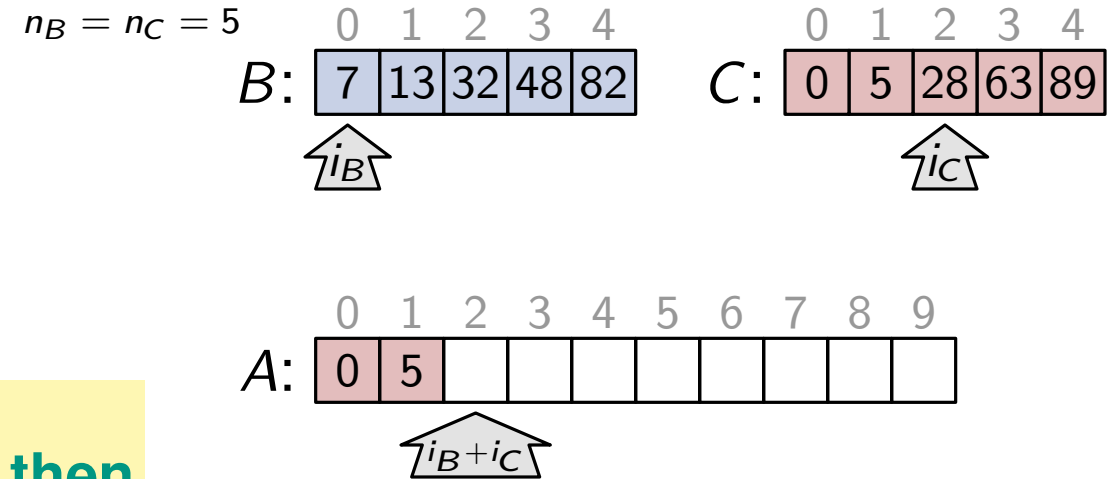
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

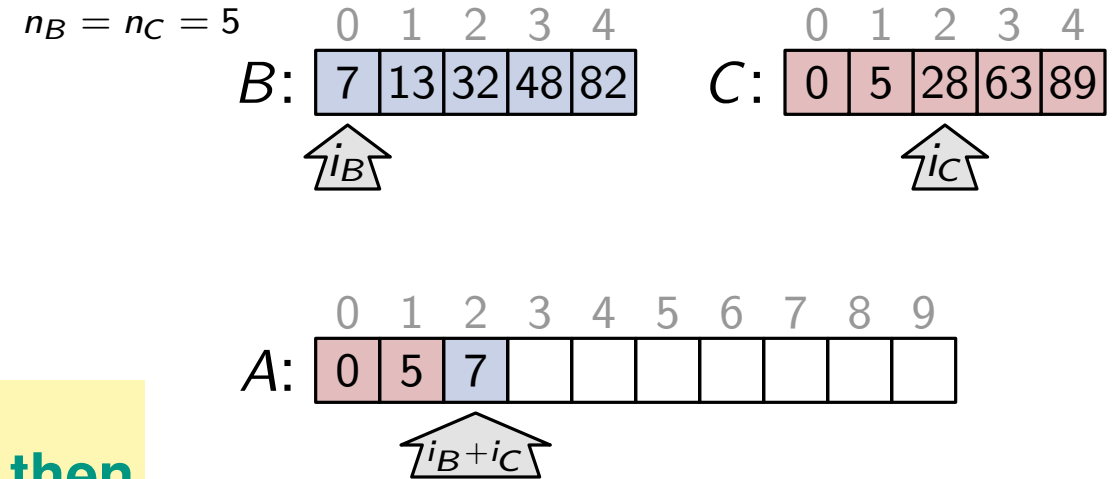
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

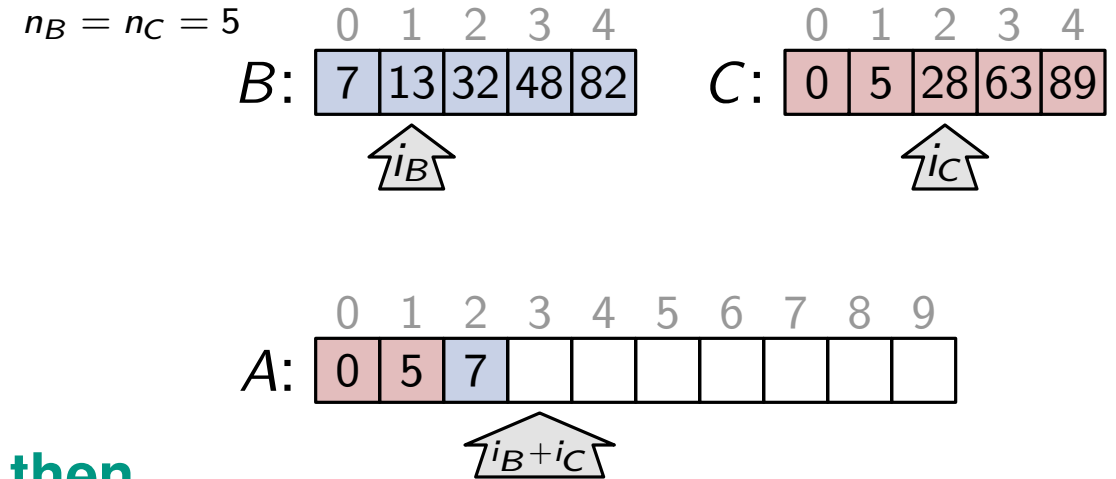
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

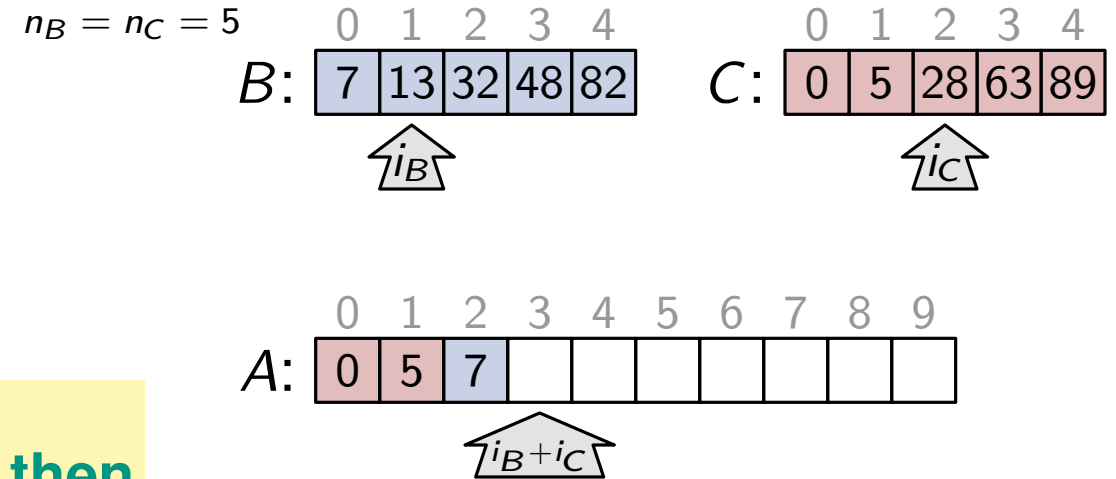
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

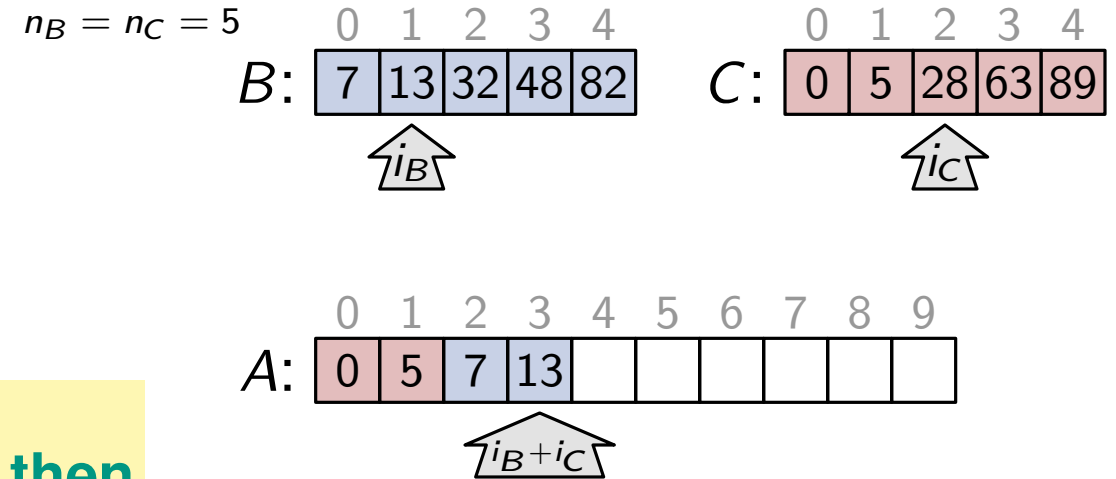
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

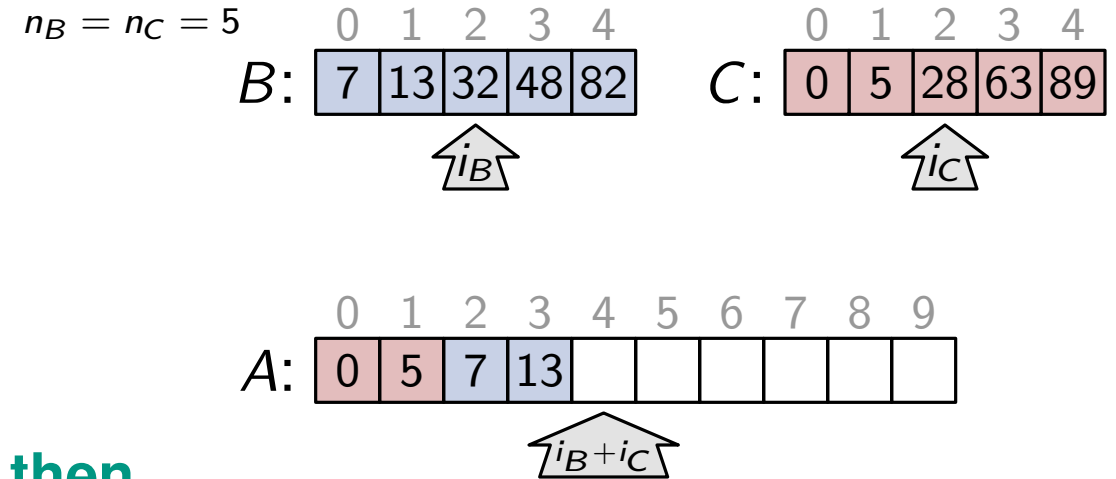
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

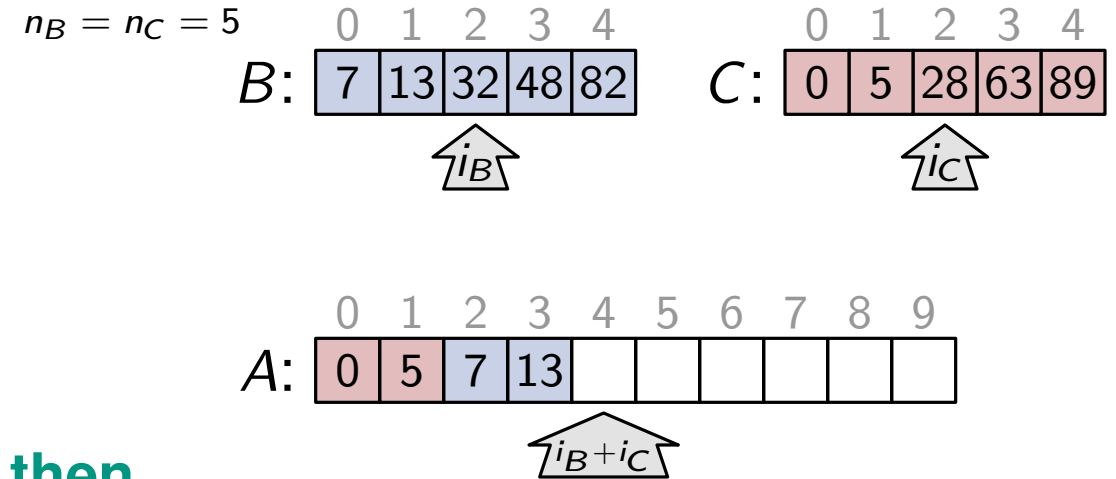
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

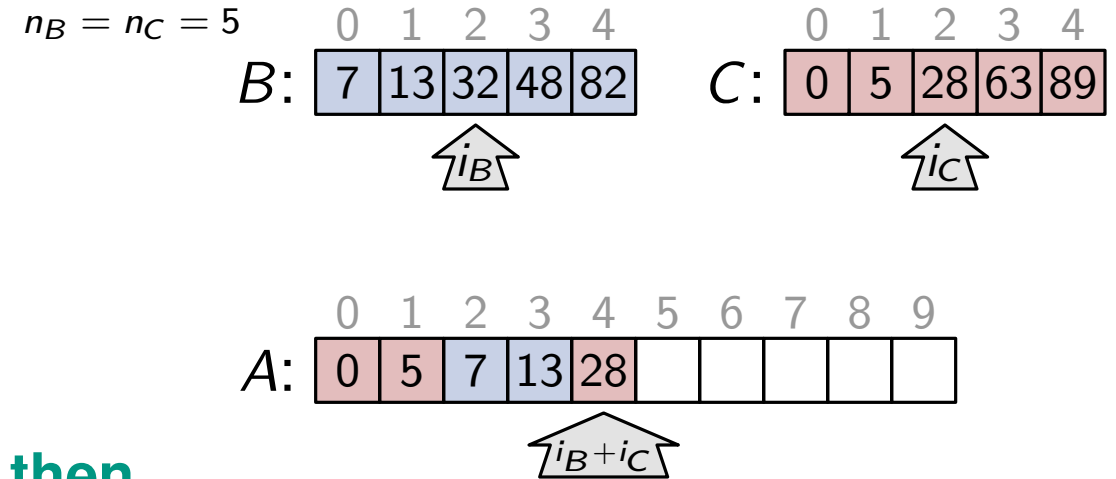
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

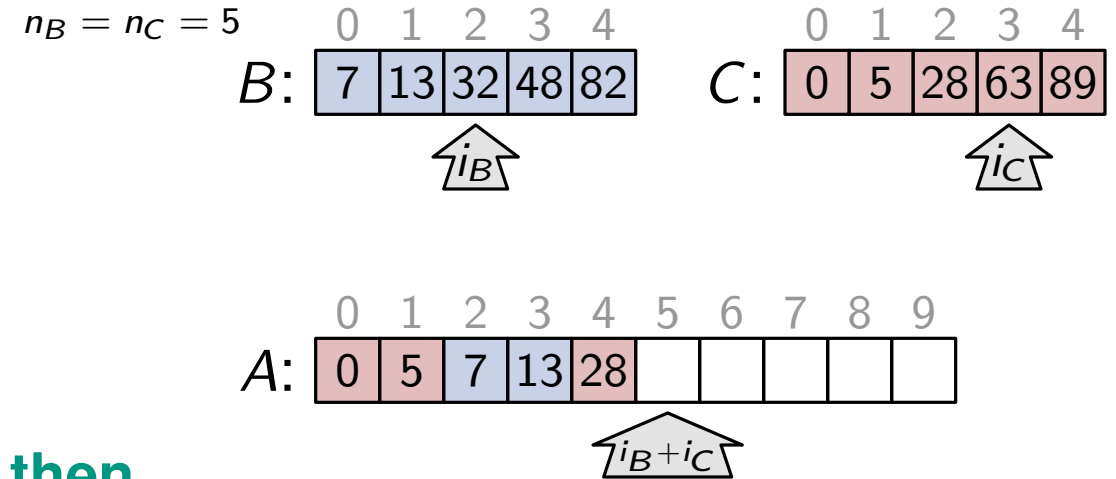
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

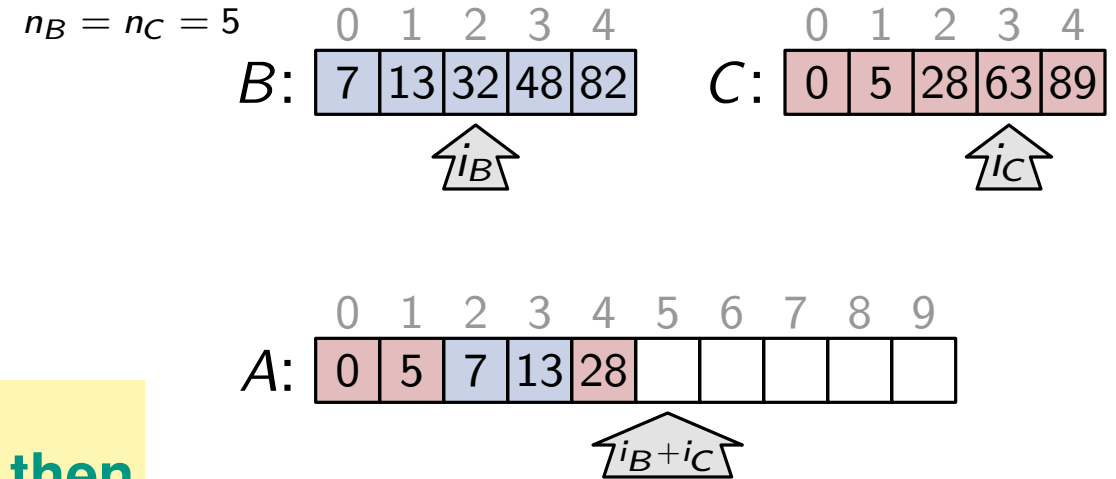
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

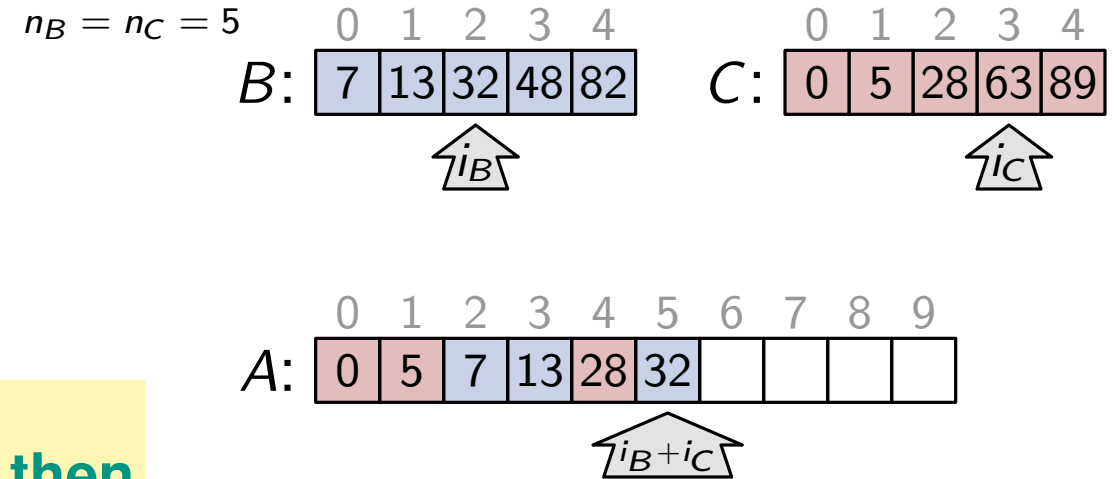
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

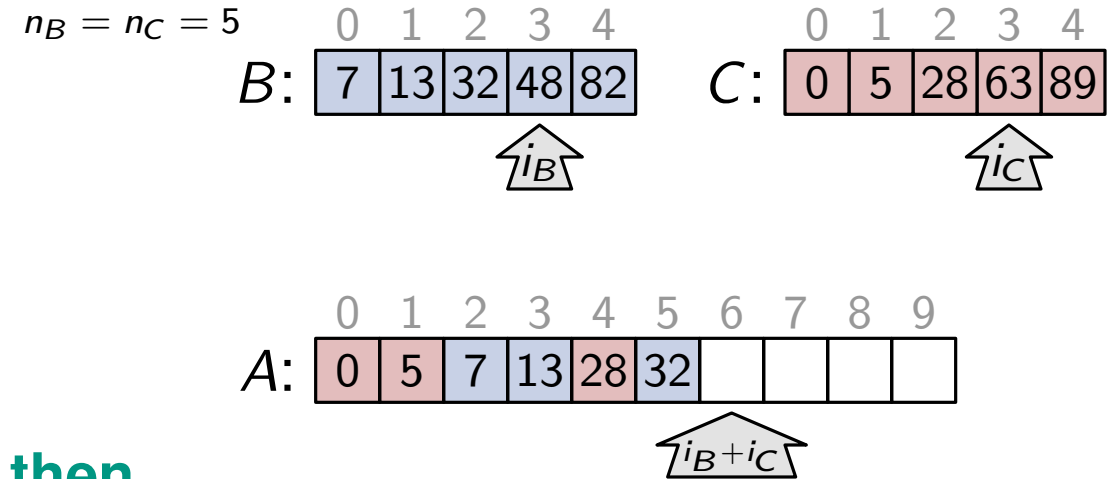
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

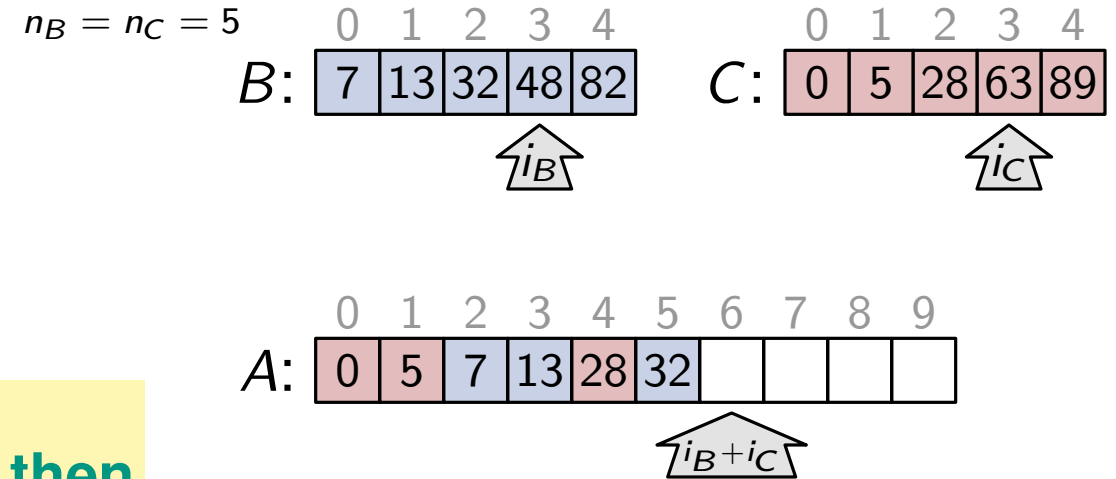
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

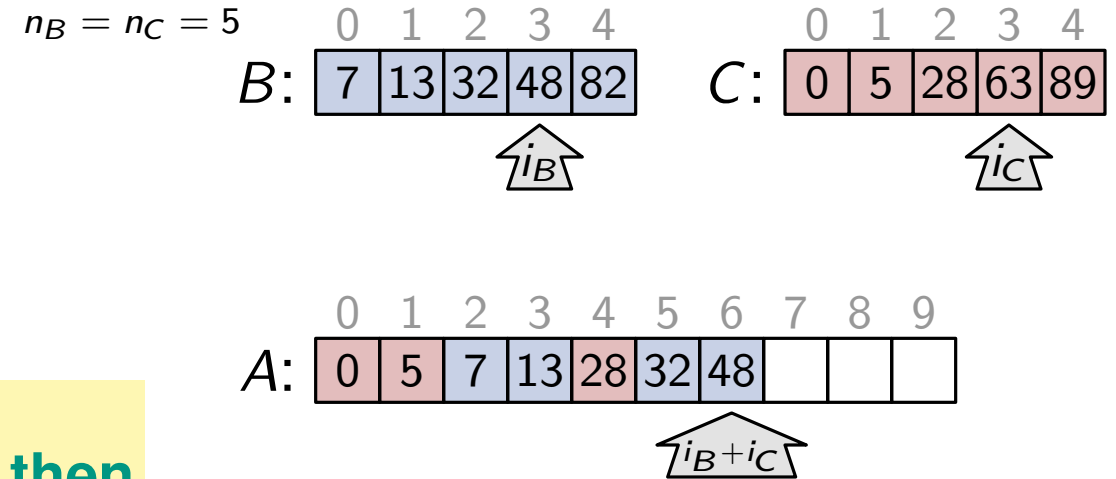
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

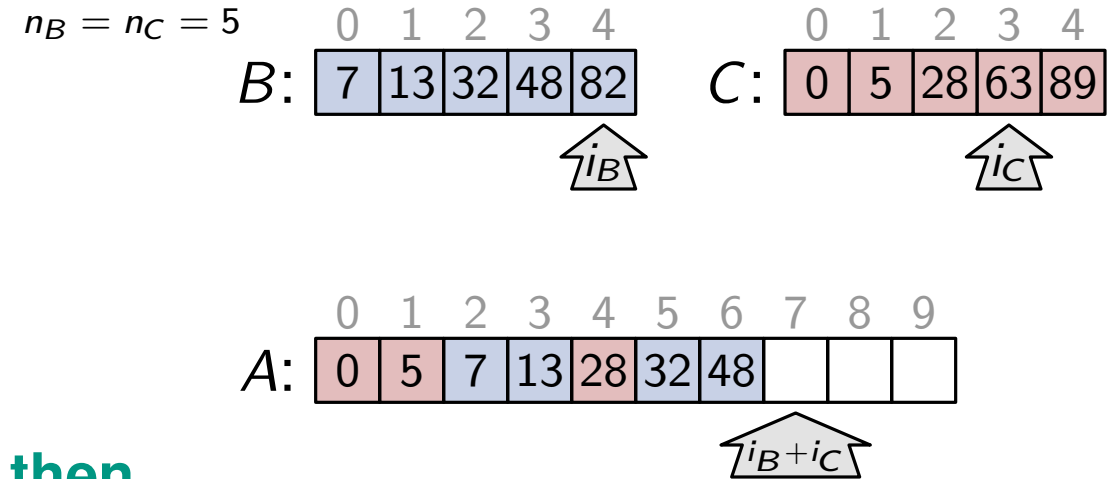
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

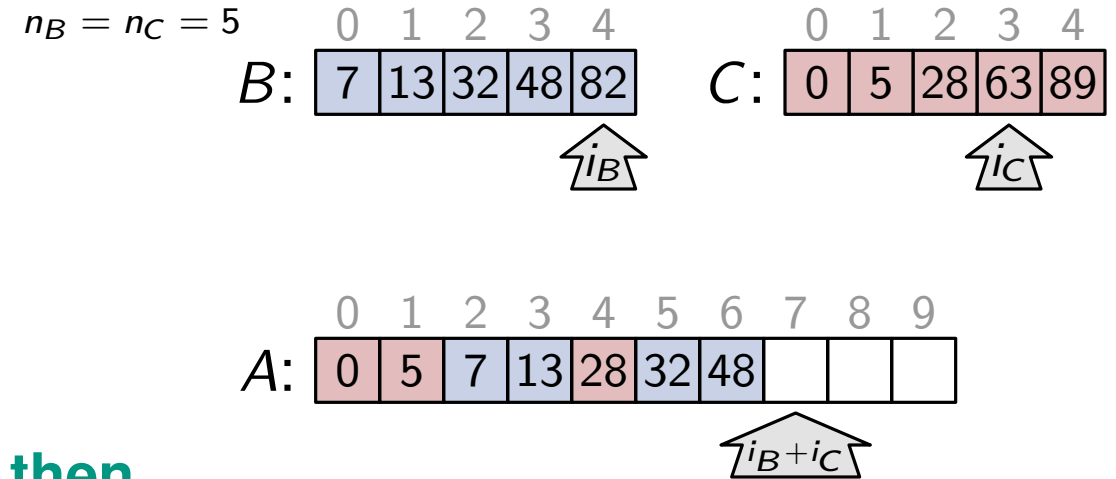
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

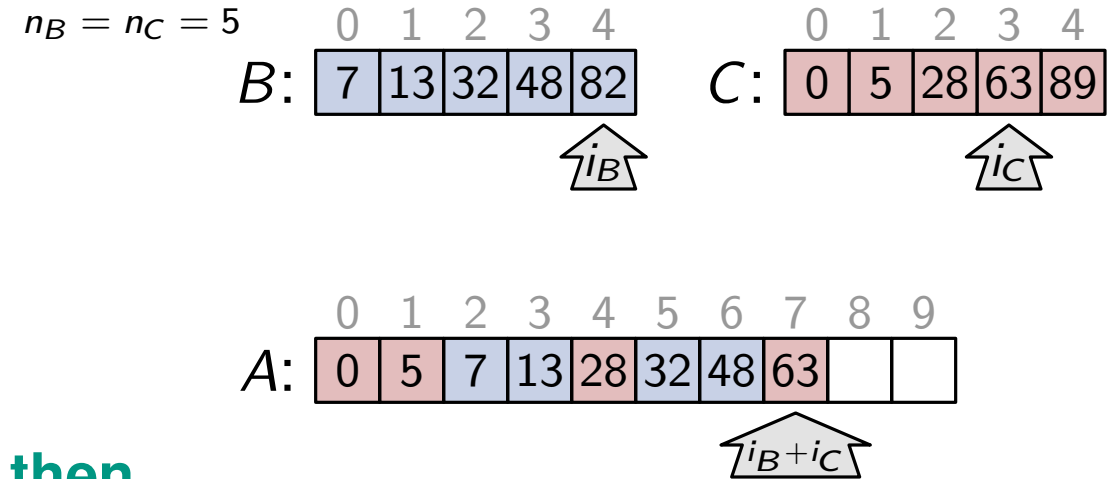
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

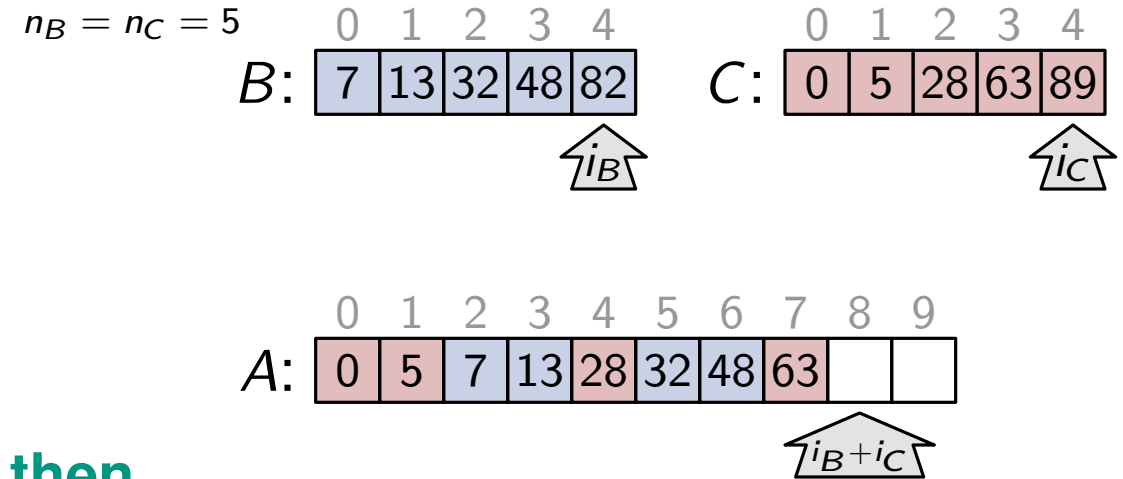
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

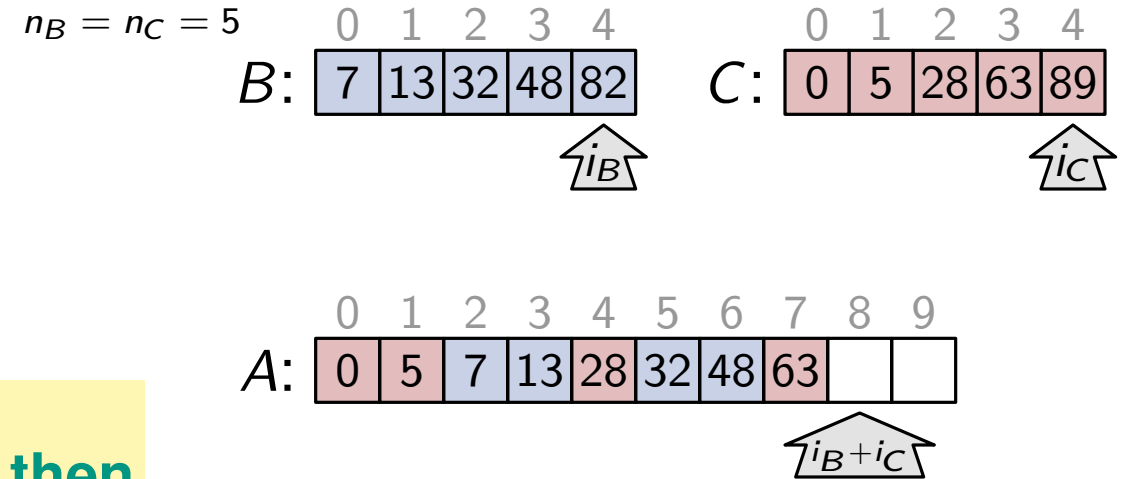
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array B , Array C)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

$A :=$ Array of size $n_B + n_C$ // result

// iterate over B and C simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from B

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

$A[i_B + i_C] := B[i_B]$

$i_B := i_B + 1$

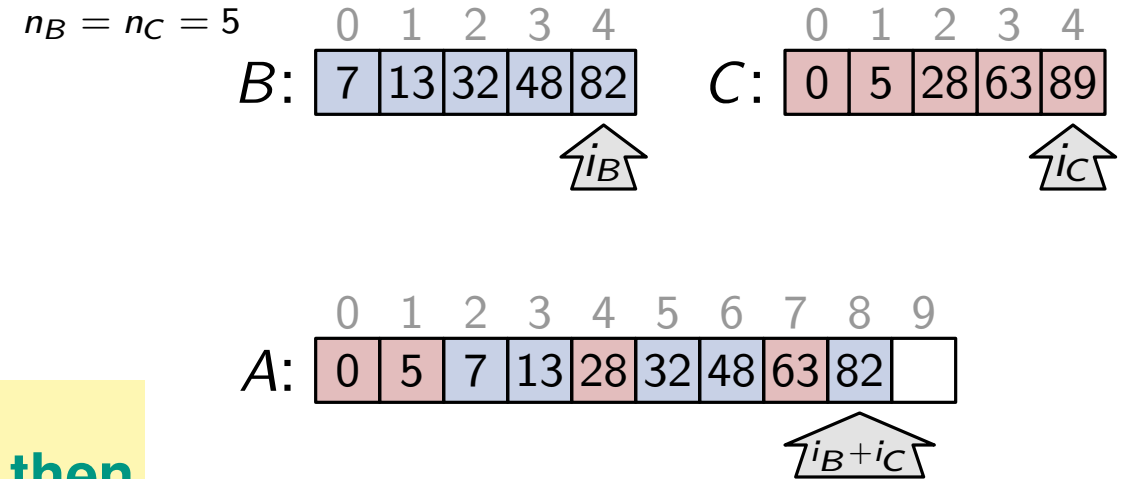
// next element comes from C

else

$A[i_B + i_C] := C[i_C]$

$i_C := i_C + 1$

return A



Merge im Detail

merge(Array *B*, Array *C*)

$i_B, i_C, n_B, n_C := 0, 0, B.size(), C.size()$

A := Array of size $n_B + n_C$ // result

// iterate over *B* and *C* simultaneously

while $i_B < n_B$ **or** $i_C < n_C$ **do**

// next element comes from *B*

if $i_C = n_C$ **or** ($i_B \neq n_B$ **and** $B[i_B] \leq C[i_C]$) **then**

| $A[i_B + i_C] := B[i_B]$

| $i_B := i_B + 1$

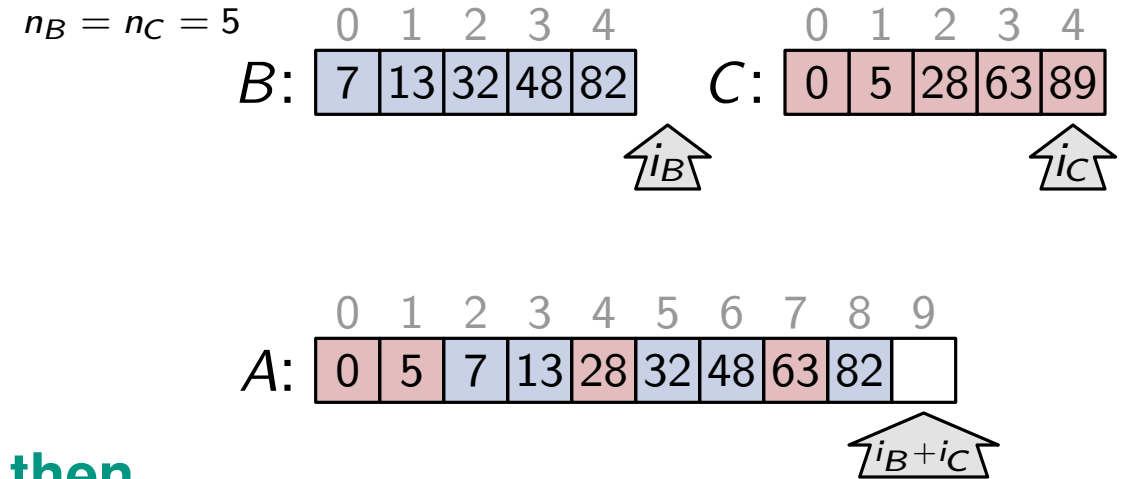
// next element comes from *C*

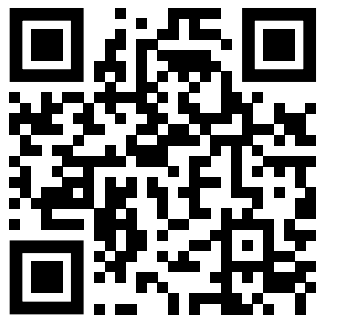
else

| $A[i_B + i_C] := C[i_C]$

| $i_C := i_C + 1$

return *A*





Laufzeit Mergesort

mergesort(*Array A*)

// base case: small array

if $A.size() \leq 1$ **then return** *A*

// partition instance

B := first half of *A*

C := second half of *A*

// solve parts

B := **mergesort**(*B*)

C := **mergesort**(*C*)

// combine solutions

return merge(*B*, *C*)

Welche Rekurrenz beschreibt die Laufzeit?

Laufzeit Mergesort

```

mergesort(Array A)            $T(n)$ 
  // base case: small array
  if A.size() ≤ 1 then return A   $\Theta(1)$ 
  // partition instance
  B := first half of A            $\Theta(n)$ 
  C := second half of A
  // solve parts
  B := mergesort(B)              $T(\frac{n}{2})$ 
  C := mergesort(C)              $T(\frac{n}{2})$ 
  // combine solutions
  return merge(B, C)            $\Theta(n)$ 
  
```

Laufzeit ($n = A.size()$)

- Rekurrenz: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$
- Mastertheorem $\Rightarrow T(n) \in \Theta(n \log n)$

Laufzeit Mergesort

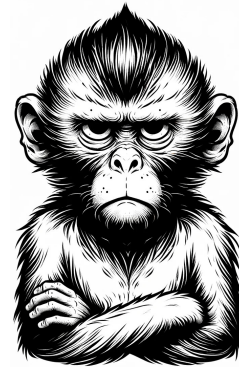
```

mergesort(Array A)            $T(n)$ 
  // base case: small array
  if A.size() ≤ 1 then return A   $\Theta(1)$ 
  // partition instance
  B := first half of A            $\Theta(n)$ 
  C := second half of A
  // solve parts
  B := mergesort(B)              $T(\frac{n}{2})$ 
  C := mergesort(C)              $T(\frac{n}{2})$ 
  // combine solutions
  return merge(B, C)            $\Theta(n)$ 
  
```

Laufzeit

($n = A.size()$)

- Rekurrenz: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$
- Mastertheorem $\Rightarrow T(n) \in \Theta(n \log n)$



Mastertheorem?

Voll anstrengend!

Muss ich das auswendig können?

Laufzeit Mergesort

```

mergesort(Array A)            $T(n)$ 
  // base case: small array
  if A.size() ≤ 1 then return A   $\Theta(1)$ 
  // partition instance
  B := first half of A           $\Theta(n)$ 
  C := second half of A
  // solve parts
  B := mergesort(B)              $T(\frac{n}{2})$ 
  C := mergesort(C)              $T(\frac{n}{2})$ 
  // combine solutions
  return merge(B, C)            $\Theta(n)$ 
  
```

Laufzeit

($n = A.size()$)

- Rekurrenz: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$
- Mastertheorem $\Rightarrow T(n) \in \Theta(n \log n)$



Mastertheorem?

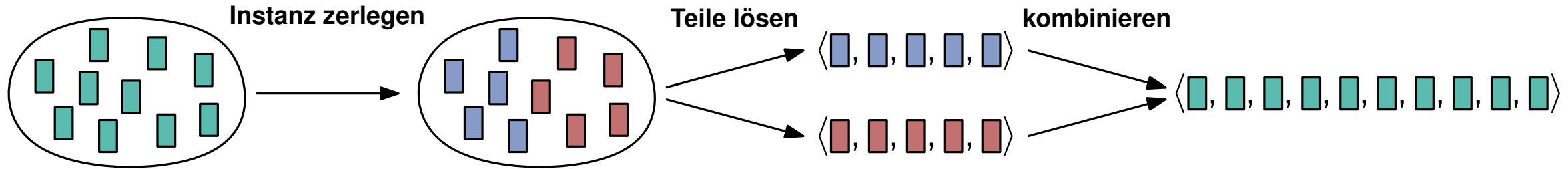
Voll anstrengend!

Muss ich das auswendig können?

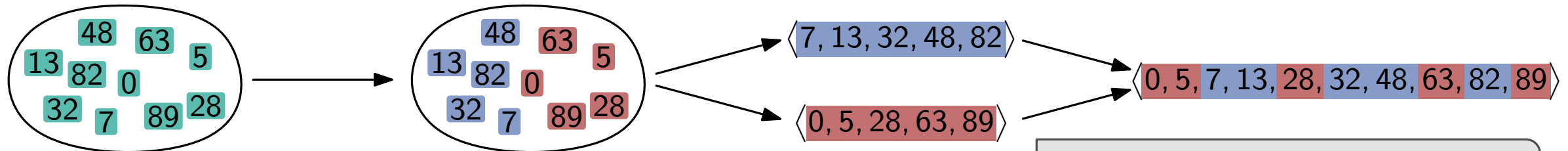
Herleitung für $\Theta(n \log n)$ in diesem Fall

- Rekursionsbaum: Binärbaum der Tiefe $\log_2 n$
- Aufwand auf Ebene i :
 - 2^i Knoten mit $n \cdot 2^{-i}$ Elementen
 - $\Theta(n)$ Kosten für n Elem. $\Rightarrow \Theta(n)$ pro Ebene
- gesamt: $\Theta(n \log n)$

Sortieren mittels Teile und Herrsche



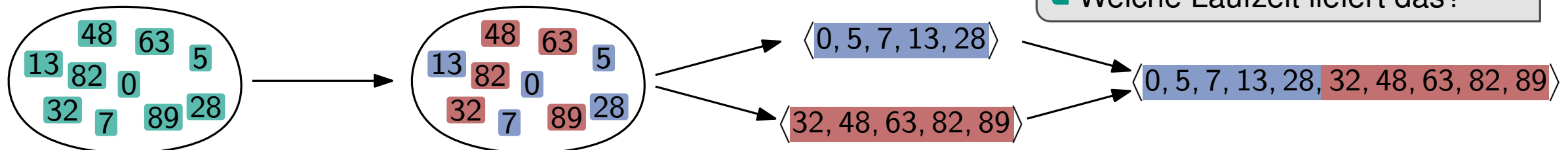
Mergesort: arbeite beim Zusammenfügen



Zwei offene Fragen

- Wie setzt man das im Detail um?
- Welche Laufzeit liefert das?

Quicksort: arbeite beim Zerlegen



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

13	48	82	7	32	63	89	0	28	5
----	----	----	---	----	----	----	---	----	---

Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

13	48	82	7	32	63	89	0	28	5
----	----	----	---	----	----	----	---	----	---

Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

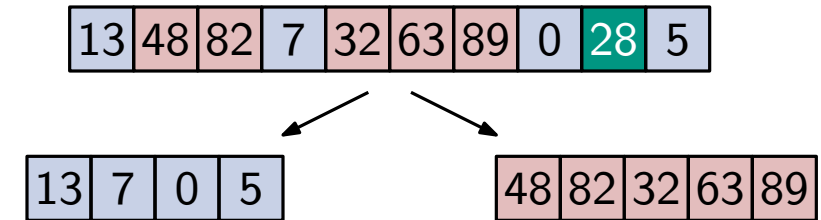
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

13	48	82	7	32	63	89	0	28	5
----	----	----	---	----	----	----	---	----	---

Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

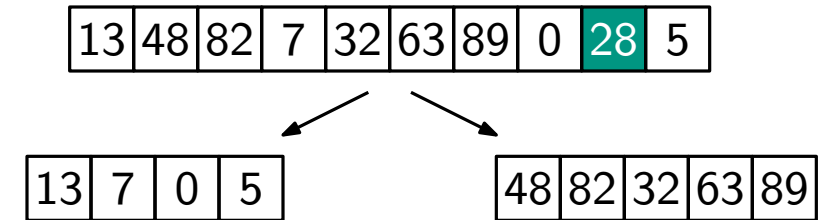
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

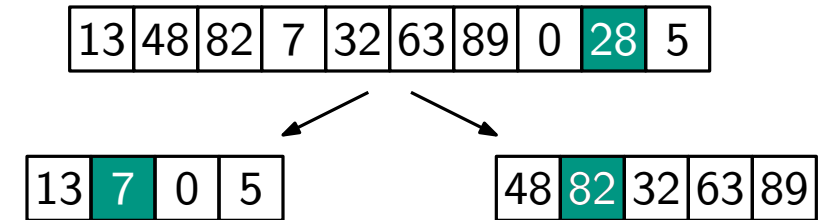
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

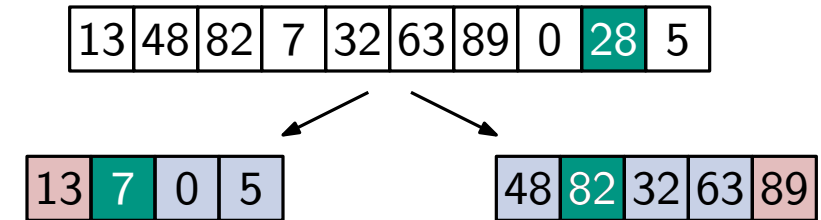
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

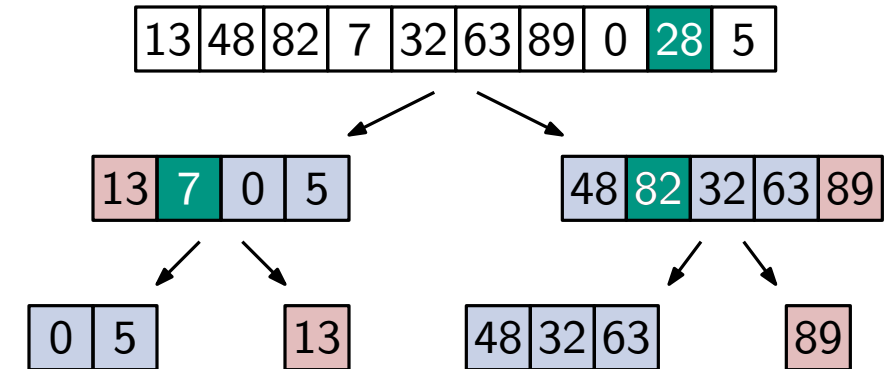
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

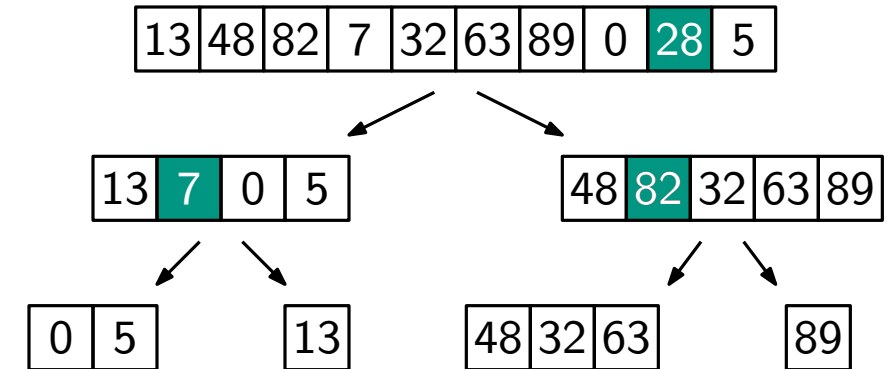
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

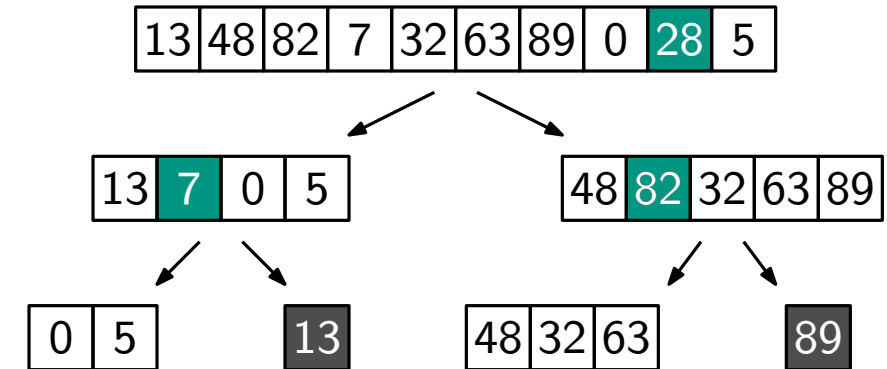
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

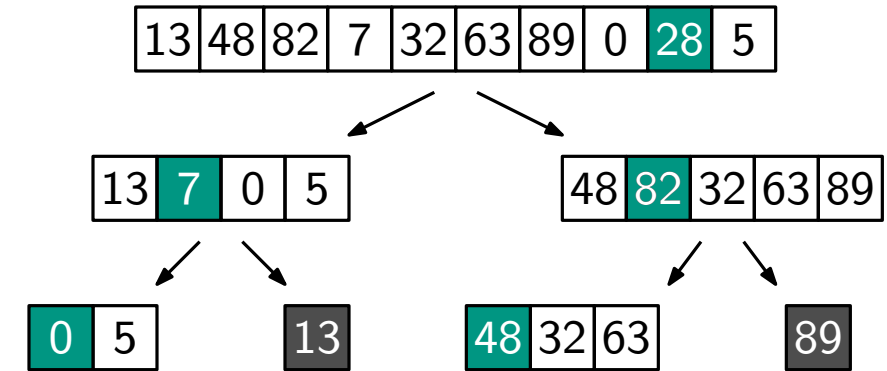
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

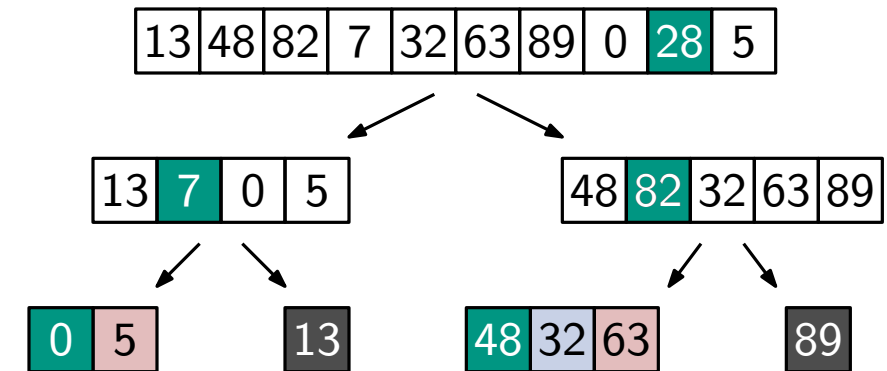
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

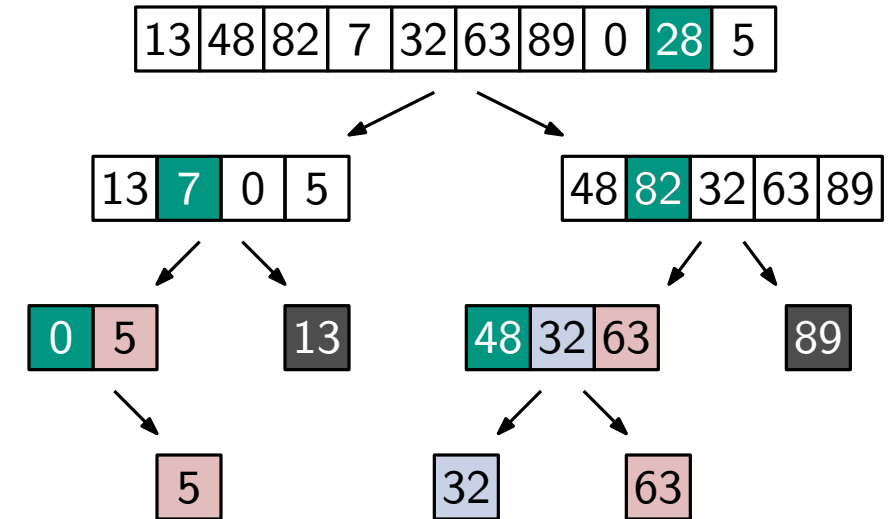
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

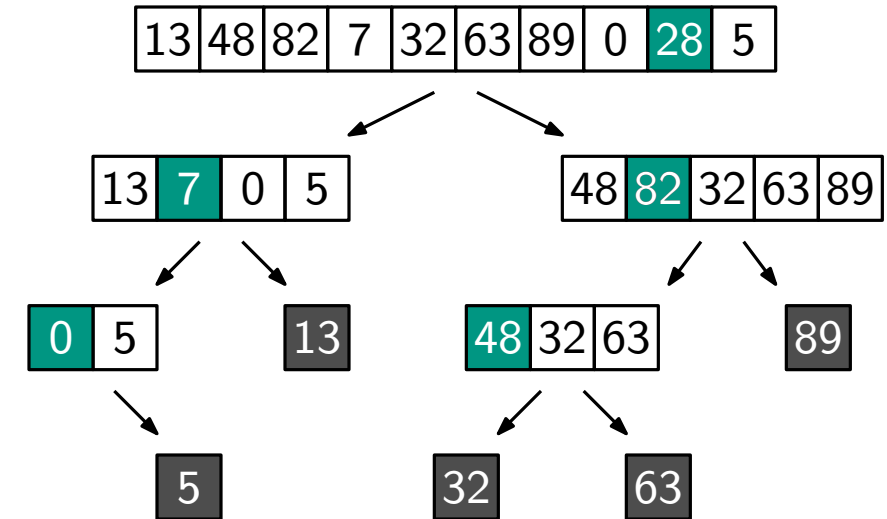
- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot



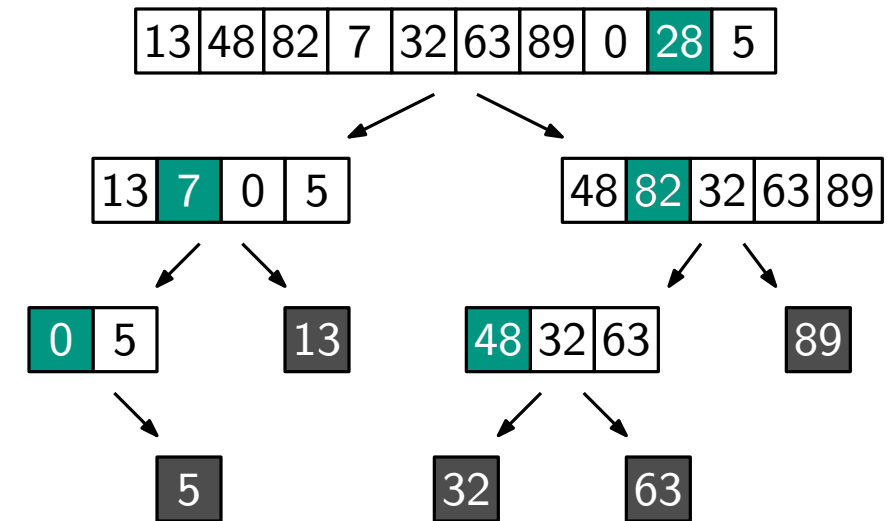
Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

Best Case

- die zwei Teilarrays sind etwa gleich groß
- $\Theta(\log n)$ Ebenen, $\Theta(n)$ Vergleiche pro Ebene
- Gesamtkosten: $\Theta(n \log n)$



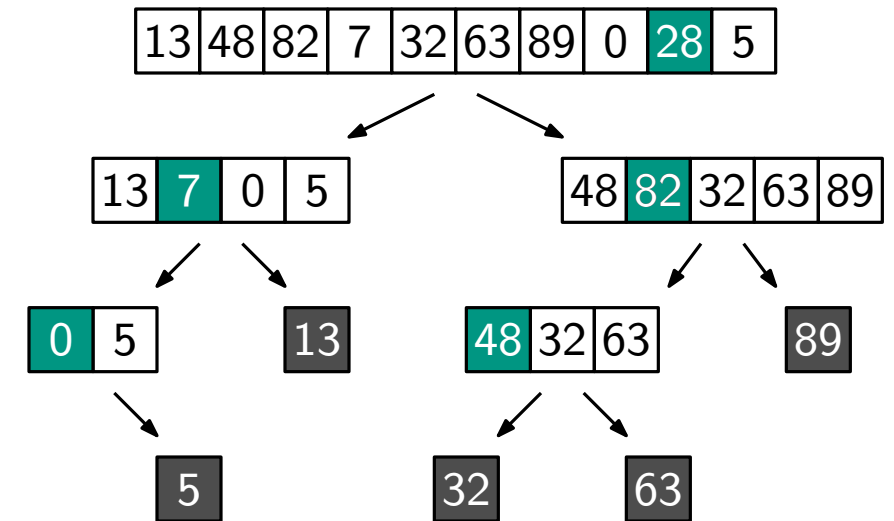
Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

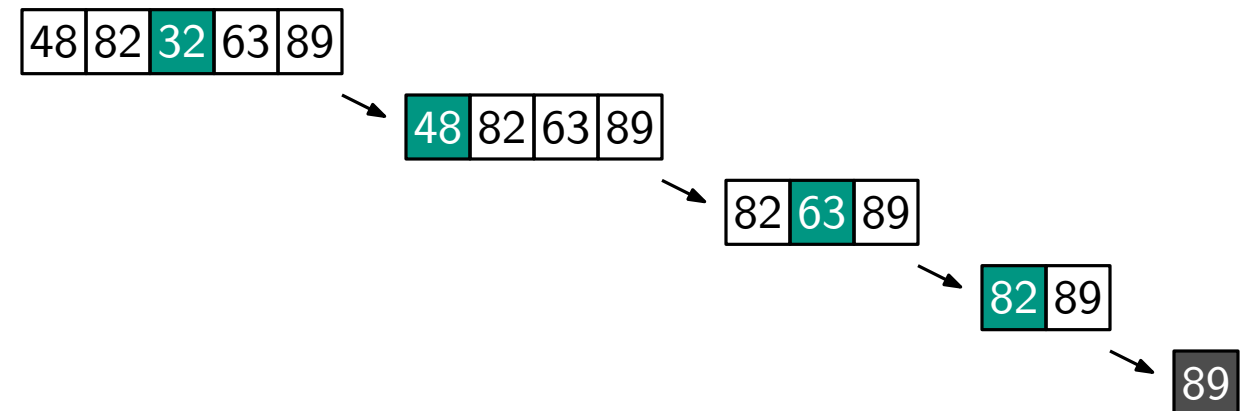
Best Case

- die zwei Teilarrays sind etwa gleich groß
- $\Theta(\log n)$ Ebenen, $\Theta(n)$ Vergleiche pro Ebene
- Gesamtkosten: $\Theta(n \log n)$



Worst Case

- Pivot ist kleinstes oder größtes Element
- Vergleich zwischen jedem Elementpaar
- Gesamtkosten: $\Theta(n^2)$



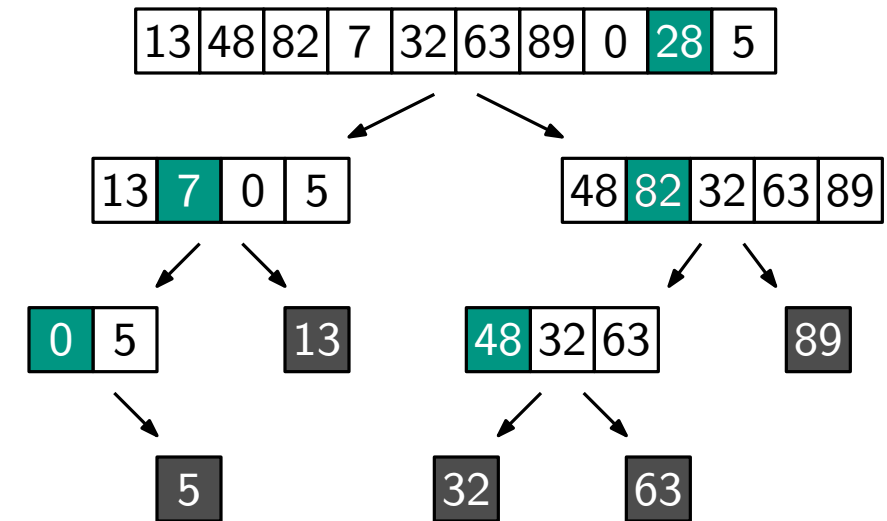
Quicksort: arbeiten beim Zerlegen

Zerlegung in große und kleine Elemente

- wähle ein Element als **Pivot Element**
- zwei Teilarrays: Elemente die kleiner/größer sind als das Pivot

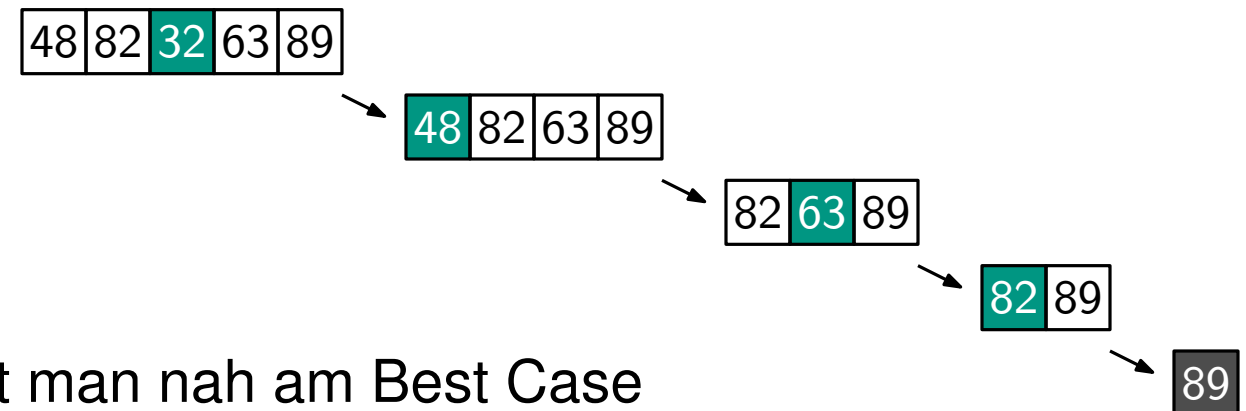
Best Case

- die zwei Teilarrays sind etwa gleich groß
- $\Theta(\log n)$ Ebenen, $\Theta(n)$ Vergleiche pro Ebene
- Gesamtkosten: $\Theta(n \log n)$



Worst Case

- Pivot ist kleinstes oder größtes Element
- Vergleich zwischen jedem Elementpaar
- Gesamtkosten: $\Theta(n^2)$



Hoffnung: mit zufällig gewähltem Pivot ist man nah am Best Case

Erwartete Laufzeit bei zufälligen Pivots

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot

Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i]$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot

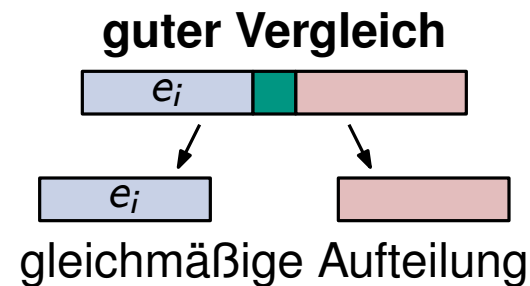
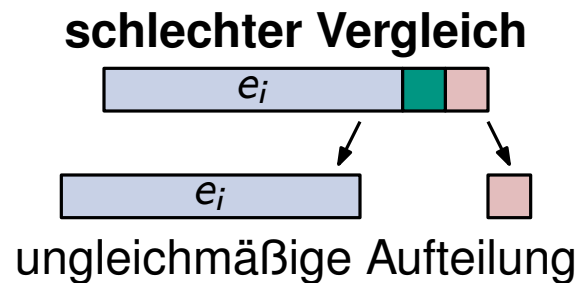
Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i]$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Vergleiche des i -ten Elements e_i



Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

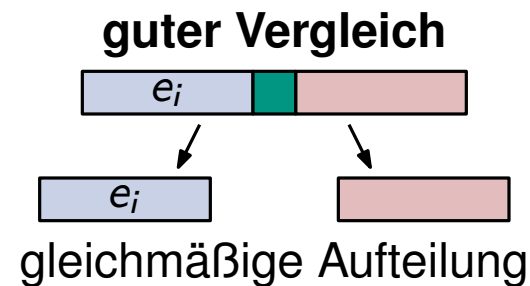
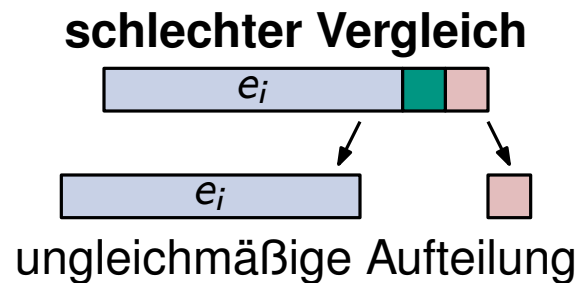
Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i]$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Vergleiche des i -ten Elements e_i



Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

$$\begin{aligned} \mathbb{E}[X_i] &= \mathbb{E}[X_i^- + X_i^+] \\ &= \mathbb{E}[X_i^-] + \mathbb{E}[X_i^+] \end{aligned}$$

Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i]$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Vergleiche des i -ten Elements e_i



- Ziel: wähle genaue Definition sodass
 - weniger schlecht als gut: $\mathbb{E}[X_i^-] \leq \mathbb{E}[X_i^+]$
 - wenige gute Vergleiche: $X_i^+ \in O(\log n)$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

$$\begin{aligned} \mathbb{E}[X_i] &= \mathbb{E}[X_i^- + X_i^+] \\ &= \mathbb{E}[X_i^-] + \mathbb{E}[X_i^+] \end{aligned}$$

Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i]$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Vergleiche des i -ten Elements e_i



- Ziel: wähle genaue Definition sodass

- weniger schlecht als gut: $\mathbb{E}[X_i^-] \leq \mathbb{E}[X_i^+]$
- wenige gute Vergleiche: $X_i^+ \in O(\log n)$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

$$\begin{aligned} \mathbb{E}[X_i] &= \mathbb{E}[X_i^- + X_i^+] \\ &= \mathbb{E}[X_i^-] + \mathbb{E}[X_i^+] \\ &\leq 2\mathbb{E}[X_i^+] \in O(\log n) \end{aligned}$$

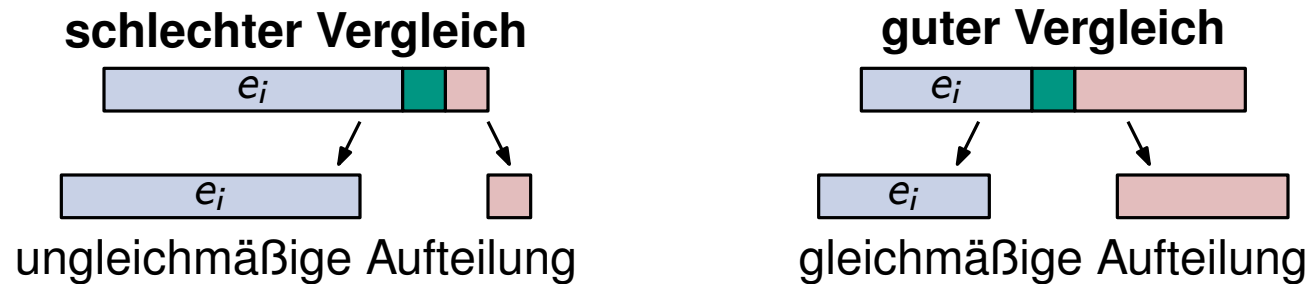
Erwartete Laufzeit bei zufälligen Pivots

Nutze Linearität des Erwartungswerts

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i] \in O(n \log n)$$

- Ziel: schätze $\mathbb{E}[X_i]$ ab für beliebiges aber festes i

Vergleiche des i -ten Elements e_i



- Ziel: wähle genaue Definition sodass

- weniger schlecht als gut: $\mathbb{E}[X_i^-] \leq \mathbb{E}[X_i^+]$
- wenige gute Vergleiche: $X_i^+ \in O(\log n)$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_i : Anzahl Vergleiche des i -ten Elements e_i mit einem Pivot
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

$$\begin{aligned} \mathbb{E}[X_i] &= \mathbb{E}[X_i^- + X_i^+] \\ &= \mathbb{E}[X_i^-] + \mathbb{E}[X_i^+] \\ &\leq 2\mathbb{E}[X_i^+] \in O(\log n) \end{aligned}$$

Gute und schlechte Vergleiche im Detail

Wann nennen wir den Vergleich mit einem Pivot gut?

Zufallsvariablen

- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

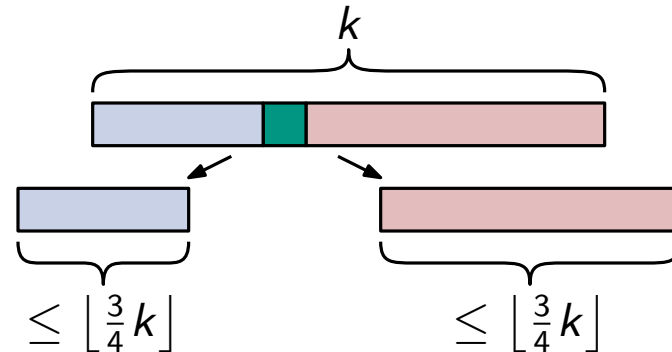
Wenige gute Vergleiche: $X_i^+ \in O(\log n)$

Mehr gute als schlechte Vergleiche: $\mathbb{E} [X_i^+] \geq \mathbb{E} [X_i^-]$

Gute und schlechte Vergleiche im Detail

Wann nennen wir den Vergleich mit einem Pivot gut?

- k : Größe des aktuellen Arrays
- guter Vergleich: beide Teilarrays $\leq \lfloor \frac{3}{4}k \rfloor$



Zufallsvariablen

- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

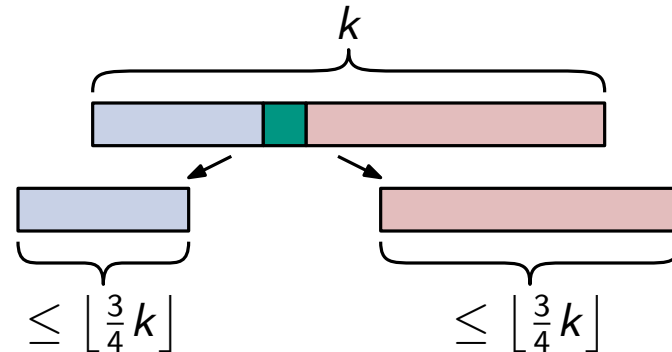
Wenige gute Vergleiche: $X_i^+ \in O(\log n)$

Mehr gute als schlechte Vergleiche: $\mathbb{E} [X_i^+] \geq \mathbb{E} [X_i^-]$

Gute und schlechte Vergleiche im Detail

Wann nennen wir den Vergleich mit einem Pivot gut?

- k : Größe des aktuellen Arrays
- guter Vergleich: beide Teilarrays $\leq \lfloor \frac{3}{4}k \rfloor$



Zufallsvariablen

- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

Wenige gute Vergleiche: $X_i^+ \in O(\log n)$

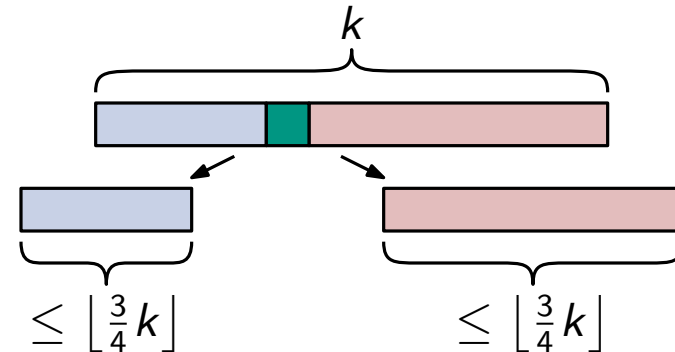
- Teilarray von e_i schrumpft bei jedem guten Vergleich um den Faktor $\frac{3}{4}$
- das kann nur $O(\log n)$ oft passieren

Mehr gute als schlechte Vergleiche: $\mathbb{E}[X_i^+] \geq \mathbb{E}[X_i^-]$

Gute und schlechte Vergleiche im Detail

Wann nennen wir den Vergleich mit einem Pivot gut?

- k : Größe des aktuellen Arrays
- guter Vergleich: beide Teilarrays $\leq \lfloor \frac{3}{4}k \rfloor$



Zufallsvariablen

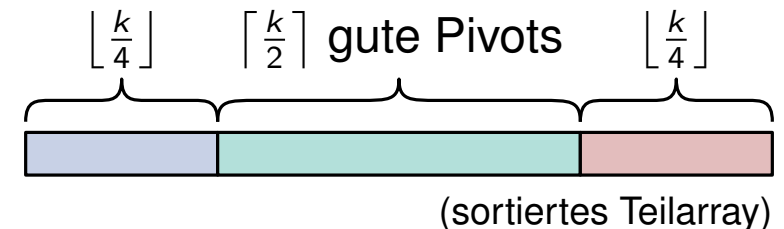
- X_i^- : Anzahl schlechte Vergleiche von e_i mit einem Pivot
- X_i^+ : Anzahl gute Vergleiche von e_i mit einem Pivot

Wenige gute Vergleiche: $X_i^+ \in O(\log n)$

- Teilarray von e_i schrumpft bei jedem guten Vergleich um den Faktor $\frac{3}{4}$
- das kann nur $O(\log n)$ oft passieren

Mehr gute als schlechte Vergleiche: $\mathbb{E}[X_i^+] \geq \mathbb{E}[X_i^-]$

- Hälfte der möglichen Pivots führen zu gutem Vergleich
- daher: $\Pr[\text{guter Vergleich}] \geq \frac{1}{2} \geq \Pr[\text{schlechter Vergleich}]$



Quicksort: Ergebnis

Theorem

Auf jeder Eingabe der Länge n benötigt Quicksort mit zufälligen Pivots im Erwartungswert $O(n \log n)$ Vergleiche.

Quicksort: Ergebnis

Theorem

Auf jeder Eingabe der Länge n benötigt Quicksort mit zufälligen Pivots im Erwartungswert $O(n \log n)$ Vergleiche.

Anmerkungen

- damit ist auch die Erwartete Laufzeit in $O(n \log n)$
- wir sprechen hier auch vom **Average Case**
- der Erwartungswert bezieht sich nur auf die zufälligen Entscheidungen des Algorithmus
- bei der Eingabe wird weiterhin der Worst Case betrachtet

Quicksort: Ergebnis

Theorem

Auf jeder Eingabe der Länge n benötigt Quicksort mit zufälligen Pivots im Erwartungswert $O(n \log n)$ Vergleiche.

Anmerkungen

- damit ist auch die Erwartete Laufzeit in $O(n \log n)$
- wir sprechen hier auch vom **Average Case**
- der Erwartungswert bezieht sich nur auf die zufälligen Entscheidungen des Algorithmus
- bei der Eingabe wird weiterhin der Worst Case betrachtet

Bonus am Ende der Folien: alternative Analyse

- liefert alternative Perspektive
- gibt genaueres Ergebnis

Untere Schranke: Geht es schneller?

Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

Anmerkung: vergleichsbasiert

- Ausführung des Algo: Sequenz von Lese- und Schreibzugriffen auf den Speicher
- vergleichsbasiert: diese Sequenz hängt nur von den Vergleichen ab
- selbe Vergleichsergebnisse \Rightarrow selbe Ausführung

Untere Schranke: Geht es schneller?

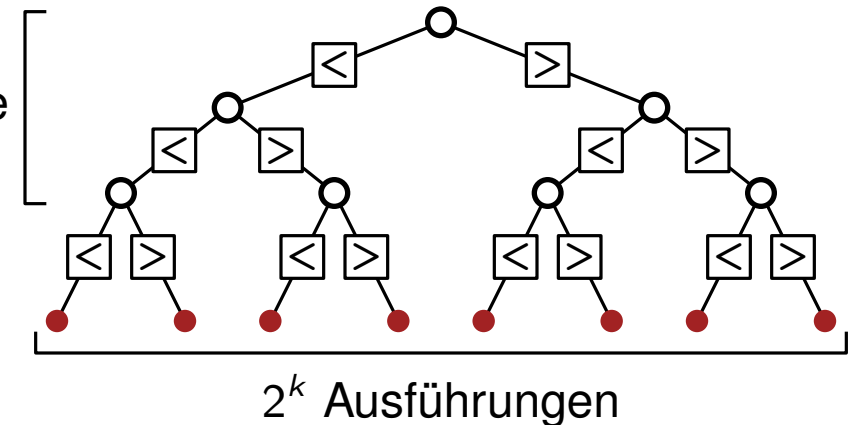
Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

- ähnlich wie bei der binären Suche:
 - binäre Entscheidung pro Vergleich
 - k Vergleiche $\rightarrow 2^k$ verschiedene Ausführungen

k Vergleiche



Anmerkung: vergleichsbasiert

- Ausführung des Algo: Sequenz von Lese- und Schreibzugriffen auf den Speicher
- vergleichsbasiert: diese Sequenz hängt nur von den Vergleichen ab
- selbe Vergleichsergebnisse \Rightarrow selbe Ausführung



Untere Schranke: Geht es schneller?

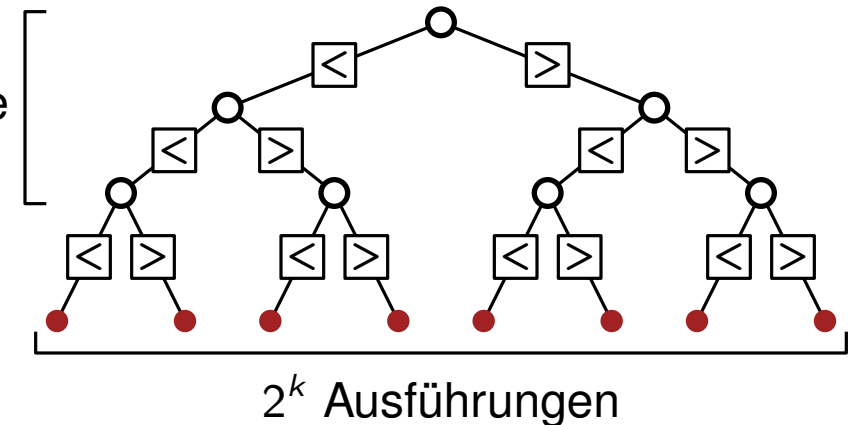
Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

- ähnlich wie bei der binären Suche:
 - binäre Entscheidung pro Vergleich
 - k Vergleiche $\rightarrow 2^k$ verschiedene Ausführungen

k Vergleiche



Wie viele verschiedene Ausführungen muss jeder korrekte Algorithmus haben können?

Anmerkung: vergleichsbasiert

- Ausführung des Algo: Sequenz von Lese- und Schreibzugriffen auf den Speicher
- vergleichsbasiert: diese Sequenz hängt nur von den Vergleichen ab
- selbe Vergleichsergebnisse \Rightarrow selbe Ausführung

Untere Schranke: Geht es schneller?

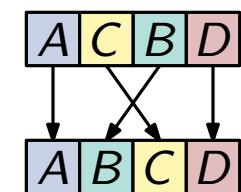
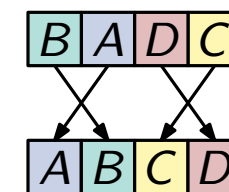
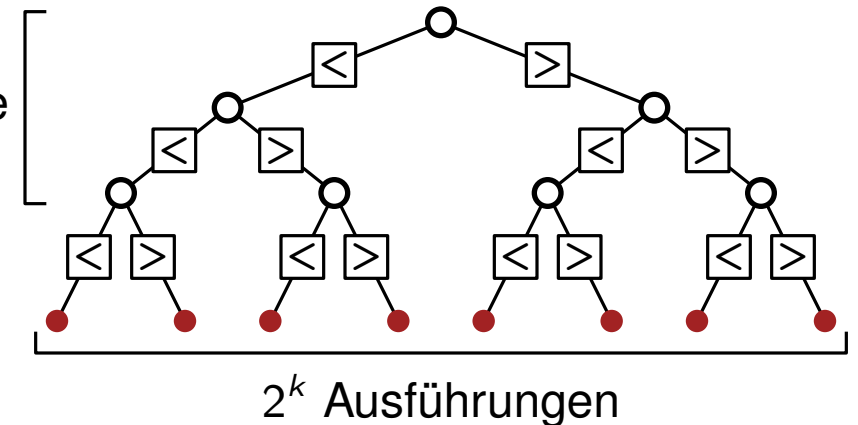
Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

- ähnlich wie bei der binären Suche:
 - binäre Entscheidung pro Vergleich
 - k Vergleiche $\rightarrow 2^k$ verschiedene Ausführungen
- verschiedene Ausführungen:
 - betrachte zwei Permutationen der Eingabe
 - Elemente müssen unterschiedlich umsortiert werden
 - jede Permutation führt zu unterschiedlicher Ausführung
 - jeder korrekte Algo braucht $n!$ verschiedene Ausführungen ($n!$ Blätter im Entscheidungsbaum)

k Vergleiche



Untere Schranke: Geht es schneller?

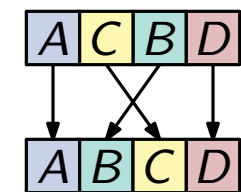
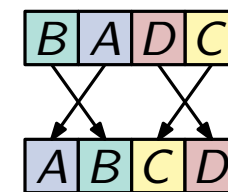
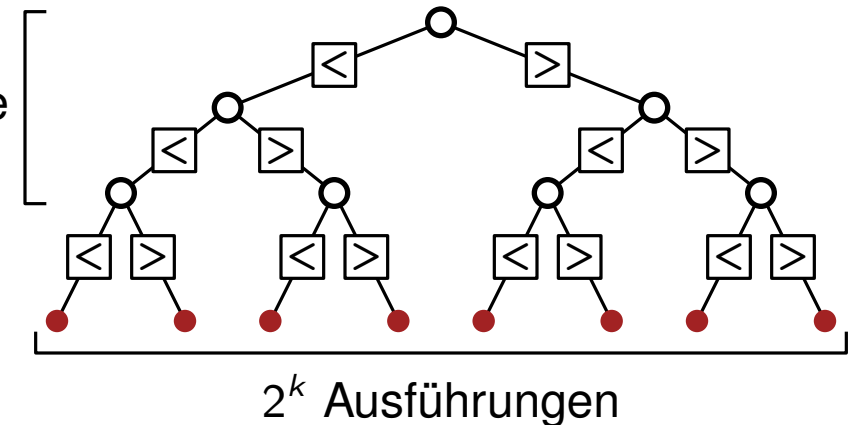
Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

- ähnlich wie bei der binären Suche:
 - binäre Entscheidung pro Vergleich
 - k Vergleiche $\rightarrow 2^k$ verschiedene Ausführungen
- verschiedene Ausführungen:
 - betrachte zwei Permutationen der Eingabe
 - Elemente müssen unterschiedlich umsortiert werden
 - jede Permutation führt zu unterschiedlicher Ausführung
 - jeder korrekte Algo braucht $n!$ verschiedene Ausführungen ($n!$ Blätter im Entscheidungsbaum)
- min. $n!$ Ausführungen \Rightarrow min. $\log(n!)$ Vergleiche

k Vergleiche



Untere Schranke: Geht es schneller?

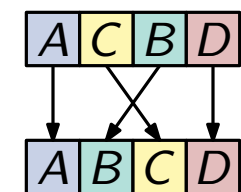
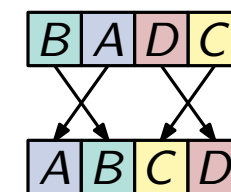
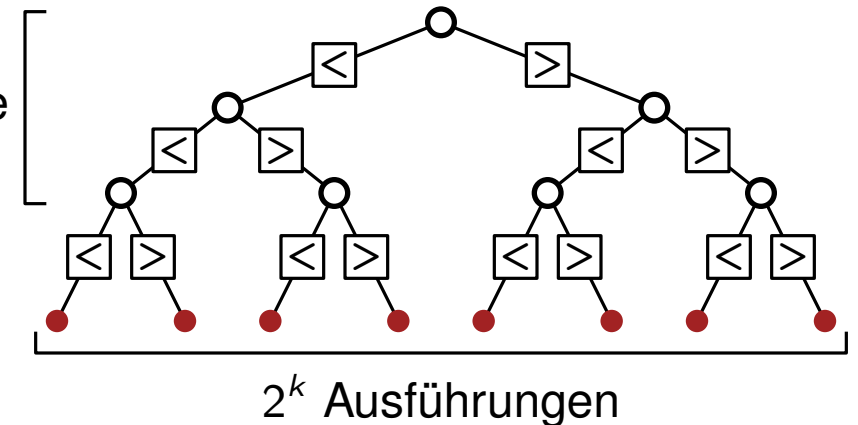
Theorem

Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche um eine Folge von n Elementen zu sortieren.

Beweis

- ähnlich wie bei der binären Suche:
 - binäre Entscheidung pro Vergleich
 - k Vergleiche $\rightarrow 2^k$ verschiedene Ausführungen
- verschiedene Ausführungen:
 - betrachte zwei Permutationen der Eingabe
 - Elemente müssen unterschiedlich umsortiert werden
 - jede Permutation führt zu unterschiedlicher Ausführung
 - jeder korrekte Algo braucht $n!$ verschiedene Ausführungen ($n!$ Blätter im Entscheidungsbaum)
- min. $n!$ Ausführungen \Rightarrow min. $\log(n!)$ Vergleiche

k Vergleiche



zu zeigen: $\log(n!) \in \Theta(n \log n)$

Nebenrechnung: $\log(n!) \in \Theta(n \log n)$

Möglichkeit 1: Stirlingformel

- die Stirlingformel sagt: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)$

Logarithmengesetze

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$

Nebenrechnung: $\log(n!) \in \Theta(n \log n)$

Möglichkeit 1: Stirlingformel

- die Stirlingformel sagt: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)$
- daher: $\log(n!) \in \Theta\left(\log\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)\right)$
 $= \Theta\left(\log\left(\sqrt{n}\right) + n \cdot (\log n - \log e)\right) = \Theta(n \log n)$

Logarithmengesetze

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$

Nebenrechnung: $\log(n!) \in \Theta(n \log n)$

Möglichkeit 1: Stirlingformel

- die Stirlingformel sagt: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)$
- daher: $\log(n!) \in \Theta\left(\log\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)\right)$
 $= \Theta\left(\log(\sqrt{n}) + n \cdot (\log n - \log e)\right) = \Theta(n \log n)$

Logarithmengesetze

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$

Möglichkeit 2: von Hand

- $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$
 - $\log(n!) = \log(n) + \log(n-1) + \log(n-2) + \dots + \log(1) \in O(n \log n)$
- n Summanden, jeder $\leq \log n$
 \downarrow

Nebenrechnung: $\log(n!) \in \Theta(n \log n)$

Möglichkeit 1: Stirlingformel

- die Stirlingformel sagt: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)$
- daher: $\log(n!) \in \Theta\left(\log\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)\right)$
 $= \Theta\left(\log(\sqrt{n}) + n \cdot (\log n - \log e)\right) = \Theta(n \log n)$

Logarithmengesetze

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$

Möglichkeit 2: von Hand

- $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$ n Summanden, jeder $\leq \log n$
- $\log(n!) = \underbrace{\log(n) + \log(n-1) + \log(n-2) + \dots + \log(1)}_{\text{ersten } \frac{n}{2} \text{ Summanden } \geq \log\left(\frac{n}{2}\right)} \in O(n \log n)$
- also: $\log(n!) \geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2} \log n - \frac{n}{2} \log 2 \in \Omega(n \log n)$

Nebenrechnung: $\log(n!) \in \Theta(n \log n)$

Möglichkeit 1: Stirlingformel

- die Stirlingformel sagt: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \in \Theta\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)$
- daher: $\log(n!) \in \Theta\left(\log\left(\sqrt{n} \left(\frac{n}{e}\right)^n\right)\right)$
 $= \Theta\left(\log(\sqrt{n}) + n \cdot (\log n - \log e)\right) = \Theta(n \log n)$

Logarithmengesetze

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$

Möglichkeit 2: von Hand

- $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$ n Summanden, jeder $\leq \log n$
- $\log(n!) = \underbrace{\log(n) + \log(n-1) + \log(n-2) + \dots + \log(1)}_{\text{ersten } \frac{n}{2} \text{ Summanden } \geq \log\left(\frac{n}{2}\right)} \in O(n \log n)$
- also: $\log(n!) \geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \frac{n}{2} \log n - \frac{n}{2} \log 2 \in \Omega(n \log n)$

$$\Rightarrow \log(n!) \in \Theta(n \log n)$$

Vergleichsbasiertes Sortieren

Heute gesehen

- Teile und Herrsche zum Sortieren von Folgen
- Mergesort: arbeite beim Zusammenfügen der Teillösungen
- Quicksort: arbeite beim Zerlegen in Teilprobleme

Vergleichsbasiertes Sortieren

Heute gesehen

- Teile und Herrsche zum Sortieren von Folgen
- Mergesort: arbeite beim Zusammenfügen der Teillösungen
- Quicksort: arbeite beim Zerlegen in Teilprobleme

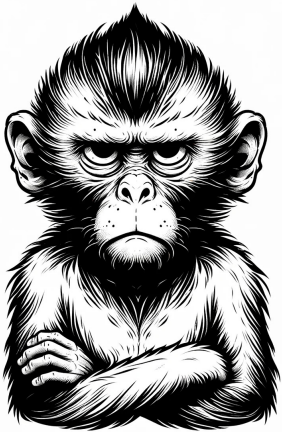


Was soll das? Hätte nicht ein $\Theta(n \log n)$ Algorithmus gereicht?

Vergleichsbasiertes Sortieren

Heute gesehen

- Teile und Herrsche zum Sortieren von Folgen
- Mergesort: arbeite beim Zusammenfügen der Teillösungen
- Quicksort: arbeite beim Zerlegen in Teilprobleme



Was soll das? Hätte nicht ein $\Theta(n \log n)$ Algorithmus gereicht?

- Mergesort: konzeptuell einfacher, deterministisch
- Quicksort: geht in-place (ohne zusätzlichen Speicher), in der Praxis schneller

Vergleichsbasiertes Sortieren

Heute gesehen

- Teile und Herrsche zum Sortieren von Folgen
- Mergesort: arbeite beim Zusammenfügen der Teillösungen
- Quicksort: arbeite beim Zerlegen in Teilprobleme



Was soll das? Hätte nicht ein $\Theta(n \log n)$ Algorithmus gereicht?

- Mergesort: konzeptuell einfacher, deterministisch
- Quicksort: geht in-place (ohne zusätzlichen Speicher), in der Praxis schneller
- Einstellungsgespräch: ggf. wird erwartet, dass ihr beide kennt

Vergleichsbasiertes Sortieren

Heute gesehen

- Teile und Herrsche zum Sortieren von Folgen
- Mergesort: arbeite beim Zusammenfügen der Teillösungen
- Quicksort: arbeite beim Zerlegen in Teilprobleme



Was soll das? Hätte nicht ein $\Theta(n \log n)$ Algorithmus gereicht?

- Mergesort: konzeptuell einfacher, deterministisch
- Quicksort: geht in-place (ohne zusätzlichen Speicher), in der Praxis schneller
- Einstellungsgespräch: ggf. wird erwartet, dass ihr beide kennt

Analysetechniken

- Analyse erwarteter Laufzeit eines randomisierten Algorithmus
- Entscheidungsbaum für untere Schranke

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\mathbb{E}[X] = \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}]$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

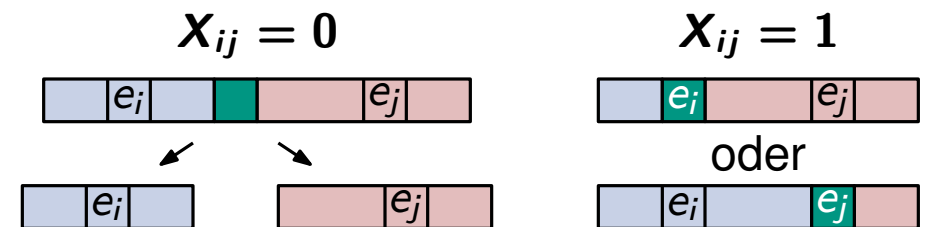
$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}]$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j





Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

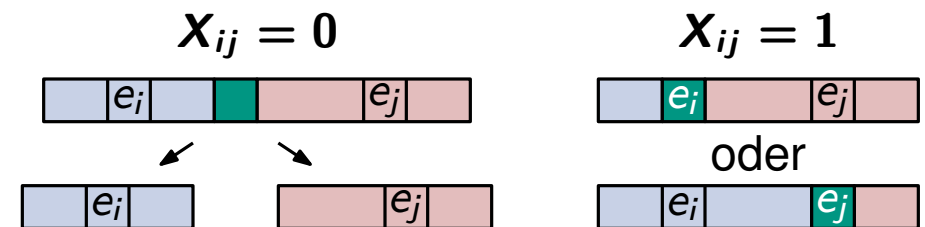
$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}]$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j



Mit welcher Wahrscheinlichkeit wird e_i mit e_j verglichen?

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

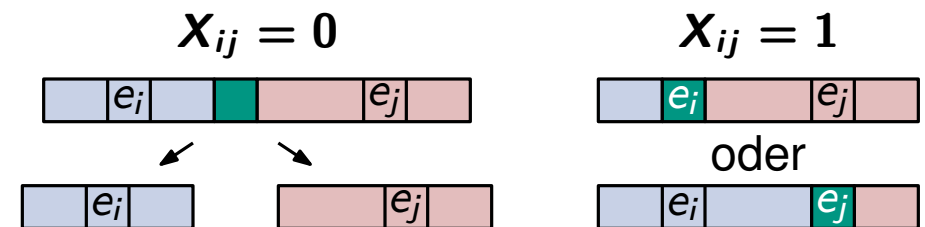
$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}]$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \end{aligned}$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j

$X_{ij} = 0$



$X_{ij} = 1$



oder



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}]$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1}$$

j	$k = j - i + 1$
$i + 1$	2
$i + 2$	3
\vdots	\vdots
n	$n - i + 1$

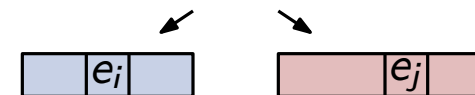
Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j

$X_{ij} = 0$



$X_{ij} = 1$



oder



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \end{aligned}$$

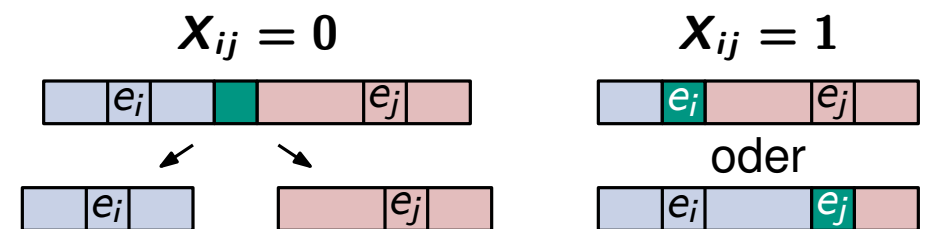
j	$k = j - i + 1$
$i + 1$	2
$i + 2$	3
\vdots	\vdots
n	$n - i + 1$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \end{aligned}$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j

$X_{ij} = 0$



$X_{ij} = 1$



oder



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned}
 \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
 &= 2n \cdot \sum_{k=2}^n \frac{1}{k}
 \end{aligned}$$

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j

$X_{ij} = 0$



$X_{ij} = 1$



oder



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned}
 \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
 &= 2n \cdot \sum_{k=2}^n \frac{1}{k} \leq 2n \ln n
 \end{aligned}$$

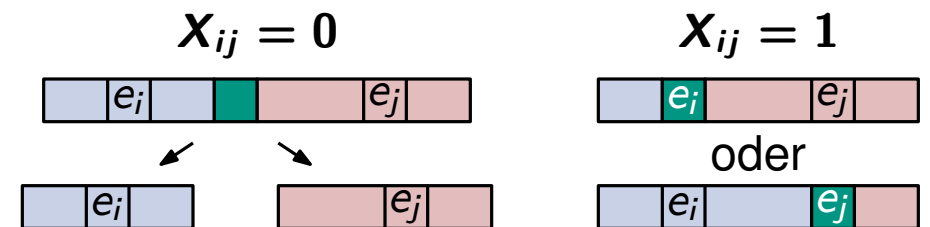
↑
harmonische Summe
 (gleich mehr dazu)

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Alternative Analyse

Summe der einzelnen Vergleiche

- seien $e_1 \leq e_2 \leq \dots \leq e_n$ die sortierten Elemente
- $X_{ij} = 1$ wenn e_i und e_j verglichen werden, $X_{ij} = 0$ sonst

$$\begin{aligned}
 \mathbb{E}[X] &= \mathbb{E} \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^n \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^n \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
 &= 2n \cdot \sum_{k=2}^n \frac{1}{k} \leq 2n \ln n
 \end{aligned}$$

Theorem

Auf jeder Eingabe der Länge n benötigt Quicksort mit zufälligen Pivots erwartet $\leq 2n \ln n$ Vergleiche.

Zufallsvariablen

- X : Anzahl Vergleiche insgesamt
- X_{ij} : Indikatorvariable für Vergleich zwischen e_i und e_j

Wahrscheinlichkeit für $X_{ij} = 1$

- $X_{ij} = 1 \Leftrightarrow$ das erste Pivot aus $\{e_i, e_{i+1}, \dots, e_{j-1}, e_j\}$ ist e_i oder e_j

$X_{ij} = 0$



$X_{ij} = 1$



oder



- $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}$

Bonus: Nebenbemerkung zur harmonischen Summe

Harmonische Zahl H_n

- $H_n = \sum_{k=1}^n \frac{1}{k}$
- nützlich zu wissen: $H_n \in \Theta(\log n)$

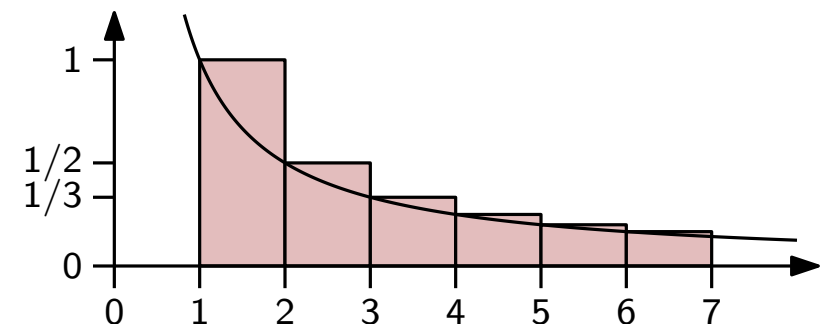
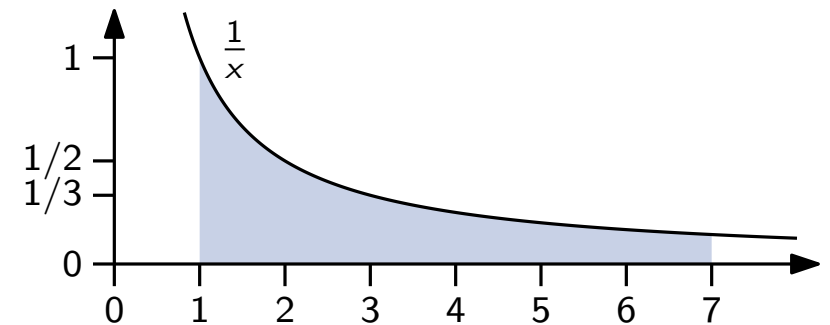
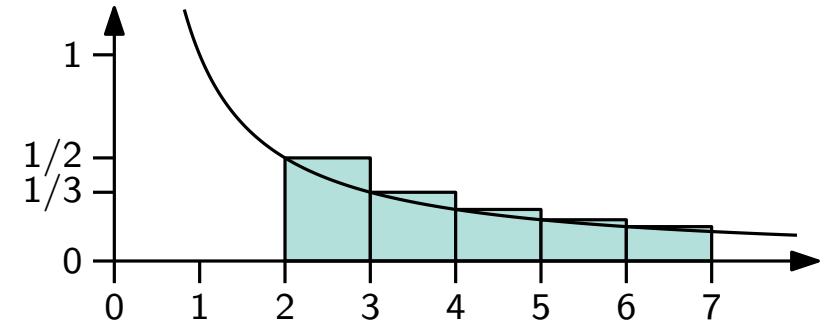
Bonus: Nebenbemerkung zur harmonischen Summe

Harmonische Zahl H_n

- $H_n = \sum_{k=1}^n \frac{1}{k}$
- nützlich zu wissen: $H_n \in \Theta(\log n)$

Beweis und genauere Abschätzung

$$H_n - 1 = \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx \leq \sum_{k=1}^n \frac{1}{k} = H_n$$



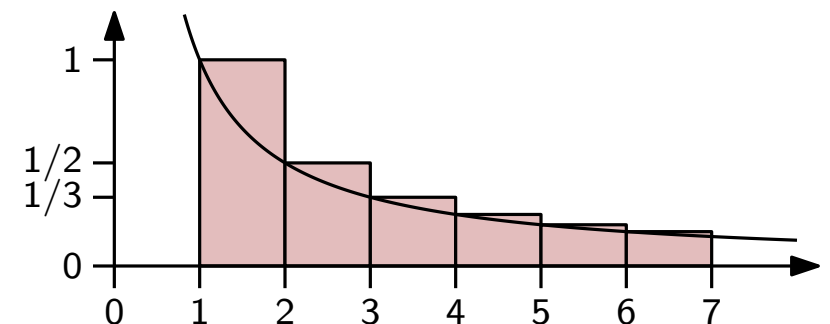
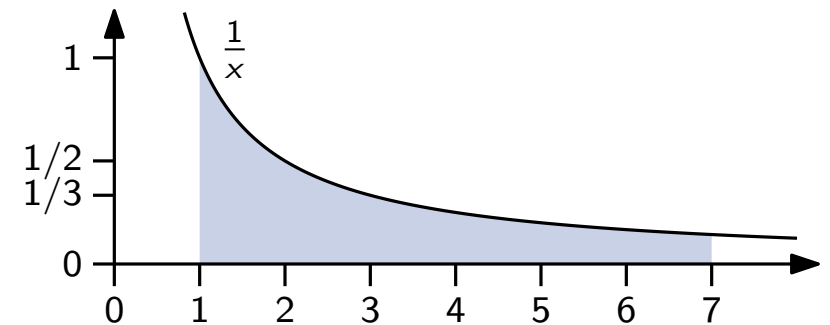
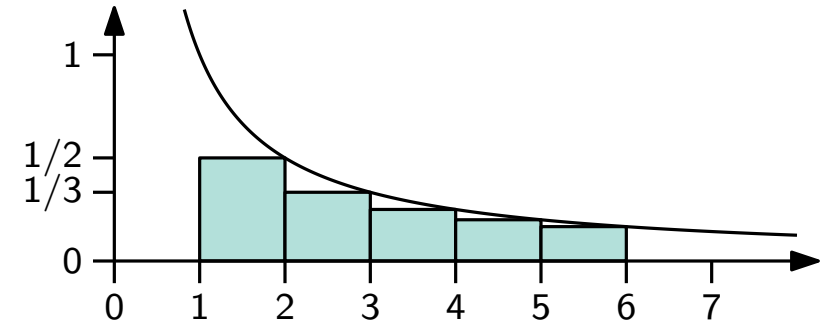
Bonus: Nebenbemerkung zur harmonischen Summe

Harmonische Zahl H_n

- $H_n = \sum_{k=1}^n \frac{1}{k}$
- nützlich zu wissen: $H_n \in \Theta(\log n)$

Beweis und genauere Abschätzung

$$H_n - 1 = \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx \leq \sum_{k=1}^n \frac{1}{k} = H_n$$



Bonus: Nebenbemerkung zur harmonischen Summe

Harmonische Zahl H_n

- $H_n = \sum_{k=1}^n \frac{1}{k}$
- nützlich zu wissen: $H_n \in \Theta(\log n)$

Beweis und genauere Abschätzung

$$H_n - 1 = \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx \leq \sum_{k=1}^n \frac{1}{k} = H_n$$

$$\int_1^n \frac{1}{x} dx = [\ln x]_1^n = \ln n$$

