Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2017

Rüdiger Göbl, Mai Bui

Computer Aided Medical Procedures
Technische Universität München

Fibonacci Zahlen

Fibonacci Folge

Die Fibonacci Folge ist eine Folge natürlicher Zahlen $f_1, f_2, f_3, \ldots$, für die gilt

$$f_n = f_{n-1} + f_{n-2} \qquad \text{für } n \geq 3$$

mit Anfangswerten $f_1 = 1$, $f_2 = 1$.

- eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- berechenbar z.B. via Rekursion

Komplexität Fibonacci Rekursiv

Input: Index n der Fibonacci Folge
Output: Wert f_n
fib(n):

```
    if (n == 1||n == 2) {
        return 1;
    } else {
        return fib(n − 1) + fib(n − 2);
    }
```

- Für $(n == 1 \vee n == 2)$ gilt:
  $T(1) = T(2) = 1$
- Für $n \geq 3$:
  $T(n) = 1 + T(n-1) + T(n-2)$
- Für $n \geq 4$ Abschätzung:
  $1 + T(n-1) + T(n-2) \leq 2 \cdot T(n-1)$
  $\Rightarrow T(n) \leq 2 \cdot T(n-1) \leq ... \leq 2^{n-3} \cdot T(3) = 2^{n-3} \cdot 3$
  $\Rightarrow T = O(2^n)$

Komplexität Fibonacci Rekursiv

Input: Index n der Fibonacci Folge
Output: Wert f_n
fib(n):

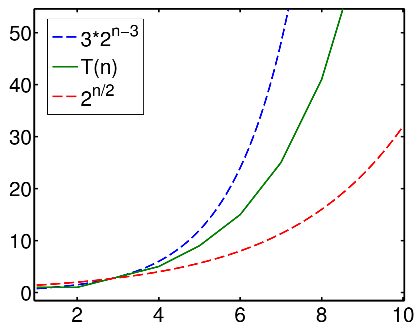
```
if (n == 1||n == 2) {
        return 1;
} else {
        return fib(n − 1) + fib(n − 2);
}
```

- Für $(n == 1 \lor n == 2)$ gilt:
  $T(1) = T(2) = 1$
- Für $n \geq 3$:
  $T(n) = 1 + T(n-1) + T(n-2)$
- Für $n \geq 3$ Abschätzung:
  $1 + T(n-1) + T(n-2) \geq 2 \cdot T(n-2)$
  $\Rightarrow T(n) \geq 2 \cdot T(n-2) \geq 2^k \cdot T(n-2k)$
  $k = \frac{n}{2} \Rightarrow T(n) \geq 2^{\frac{n}{2}}$

Komplexität Fibonacci Rekursiv

Input: Index n der Fibonacci Folge
Output: Wert f_n
fib(n):

```
if (n == 1||n == 2) {
        return 1;
} else {
        return fib(n − 1) + fib(n − 2);
}
```



Komplexität Fibonacci Dynamisch

Input: Index n der Fibonacci Folge
Output: Wert f_n
fibDyn(n)**:**

```
1      fib = leeres Feld Grösse n;
2      fib[1] = 1;
3      fib[2] = 1;
4      for k = 3 to n {
5          fib[k] = fib[k − 1] + fib[k − 2];
       }
```

Zeile	Häufigkeit
1	1
2	1
3	1
4	$2n - 2$
5	$n - 2$

- vom kleinsten Teilproblem "aufwärts"
- Zwischenergebnisse in Tabelle

Komplexität Fibonacci Dynamisch

Input: Index n der Fibonacci Folge
Output: Wert f_n
fibDyn(n):

```
1    fib = leeres Feld Grösse n;
2    fib[1] = 1;
3    fib[2] = 1;
4    for k = 3 to n {
5        fib[k] = fib[k − 1] + fib[k − 2];
     }
```

Zeile	Häufigkeit
1	1
2	1
3	1
4	$2n-2$
5	$n-2$

Laufzeit für Zeile 4: Umwandlung der for- in eine while-Schleife

fibDyn(n):

```
    ...
    k = 3;
    while k ≤ n {
        ...
        k = k + 1;
    }
```

$$\begin{aligned} T &= 1 + (n-1) + (n-2) \\ &= 2n-2 \end{aligned}$$

Komplexität Fibonacci Dynamisch

Input: Index n der Fibonacci Folge
Output: Wert f_n
fibDyn(n)**:**

```
1        fib = leeres Feld Grösse n;
2        fib[1] = 1;
3        fib[2] = 1;
4        for k = 3 to n {
5                fib[k] = fib[k − 1] + fib[k − 2];
         }
```

Zeile	Häufigkeit
1	1
2	1
3	1
4	$2n - 2$
5	$n - 2$

- $$T(n) = 1 + 1 + 1 + 1 + (n - 2 + 1) + (n - 2) + (n - 2)$$
  $$= 1 + 1 + 1 + (2n - 2) + (n - 2) = 3n - 1$$
  $$\Rightarrow T = O(n)$$

Berechnungsdauer der Fibonaccizahlen

- Rechner mit 2.5 GHz Taktfrequenz
- Jede Operation benötigt 1 Takt $\rightarrow \frac{1}{2.5ns} = 0.4ns$ pro Operation

- Abschätzung obere Schranke: $3 \cdot 2^{n-3} \geq T(n) \geq 2^{\frac{n}{2}}$
  $0.4 \cdot 3 \cdot 2^{97} ns \approx 1.9015 \cdot 10^{20} s$ mit $1ns = 10^-9s$
  $\approx 6 \cdot 10^{12} Jahre$

- Abschätzung untere Schranke:
  $0.4 \cdot 2^{50} ns \approx 4.5035 \cdot 10^5 s \approx 5 Tage$

- Dynamische Programmierung:
  $3n - 1 = 0.4 \cdot 299 = 199,6ns$

Komplexität der Polynom-Auswertung

Input: Koeffizienten-Array a der Länge n+1, Auswertungspunkt x
Output: Ergebnis $p = a[n] \cdot x^n + ... + a[1]x + a[0]$
EvaluatePolynomial(a, x)**:**

```
p = 0;
for i = n down to 0 {
    m = 1;
    for j = 1 to i {
        m = m · x;
    }
    m = a[i] · m;
    p = p + m;
}
```

- Abschätzung Multiplikationen:
  $(\sum_{i=1}^{n} i) + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$
- Abschätzung der Additionen: $n + 1$

  $\Rightarrow T(n) = (\frac{1}{2}n^2 + \frac{3}{2}n + 1) + (n+1) = O(n^2)$

Komplexität der Polynom-Auswertung

Input: Koeffizienten-Array a der Länge n+1, Auswertungspunkt x
Output: Ergebnis $p = a[n] \cdot x^n + ... + a[1]x + a[0]$
EvaluatePolynomial(a, x)**:**

```
p = 0;
h = 1;
for i = 0 to n {
    m = a[i] · h;
    h = h · x;
    p = p + m;
}
```

- Abschätzung Multiplikationen:
  $2(n+1) = 2n+2$
- Abschätzung der Additionen: $n+1$

  $\Rightarrow T(n) = 2n + 2 + n + 1 = O(n)$

Komplexität der Polynom-Auswertung Horner Schema

Horner Schema: Ausklammern von x

$a_n x^n + ... + a_1 x + a_0 = (((...)x + a_2)x + a_1)x + a_0$

Input: Koeffizienten-Array a der Länge n+1, Auswertungspunkt x
Output: Ergebnis $p = a[n] \cdot x^n + ... + a[1]x + a[0]$
EvaluatePolynomial(a, x)**:**

$p = a[n];$
for $i = n - 1$ **down to** 0 $\{$
  $p = p \cdot x;$
  $p = p + a[i];$
$\}$

- Abschätzung Multiplikationen n
- Abschätzung der Additionen: n

$\Rightarrow T(n) = n + n = O(n)$

Komplexität der Polynom-Auswertung Vergleich

O-Notation	$O(n^2)$	$O(n)$	$O(n)$
Insgesamt	$\frac{1}{2}n^2 + \frac{5}{2}n + 2$	$3n + 3$	$2n$
Multiplikation	$\frac{1}{2}n^2 + \frac{3}{2}n + 1$	$2n + 2$	n

- konstante Faktoren beeinflussen die Laufzeit
- Multiplikation als Vergleich:
  Horner Schema etwa doppelt so schnell

Divide and Conquer

- „Teile und Herrsche“ bzw. „Divide et Impera“
- Problemlösung durch
  - Aufteilung in Teilprobleme gleicher Art aber kleinerer Größe (Beispiel: Sortiere nur die erste Hälfte)
  - Gesamtlösung durch Zusammenführung der Teillösungen
- Die Aufteilung erfolgt rekursiv bis zum Trivialfall (Beispiel: Sortiere Liste der Länge 1)

Divide and Conquer: MergeSort

Sei $A = \{a_1, \dots, a_n\}$ Feld mit n natürlichen Zahlen $a_i \in \mathbb{N}$.

Aufgabe: sortiere A in aufsteigender Reihenfolge.

- Lösung mit Divide and Conquer-Muster: Merge Sort
- Idee:
  - Divide: teile A auf in zwei gleich große Teilfelder
  - Rekursion: rufe Merge Sort rekursiv für die zwei Teilfelder auf
  - Conquer: setze die Teilfelder zusammen (merge bzw. mischen)
- Wann ist Teilfolge "klein", d.h. wann löst man explizit?
  $\rightarrow$ Teilfolge mit nur einem Element $\rightarrow$ sortiert!

Merge Sort: Hauptfunktion

```
MergeSort(A):
    if (A ein-elementig) {
        return A;
    } else {
        // Divide: Erzeugung von Teilproblemen
        (A1; A2) = Split(A);

        // Rekursion: Lösung der Teilprobleme
        A1 = MergeSort(A1);
        A2 = MergeSort(A2);

        // Conquer: Kombination der Teillösungen
        return Merge(A1, A2);
    }
```

Merge Sort: Kombination

Merge(A_1, A_2)**:**

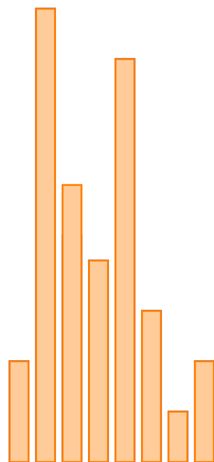
$B = \emptyset$;

while (A_1 und A_2 nicht leer) {
  if ($A_1[0] \leq A_2[0]$) {
    $B = \text{push}(B, \text{pop}(A_1))$;
  } **else** {
    $B = \text{push}(B, \text{pop}(A_2))$;
  }
}

if (A_1 nicht leer) {
  $B = \text{pushAll}(B, A_1)$;
} **else** {
  $B = \text{pushAll}(B, A_2)$;
}

return B;

Merge Sort

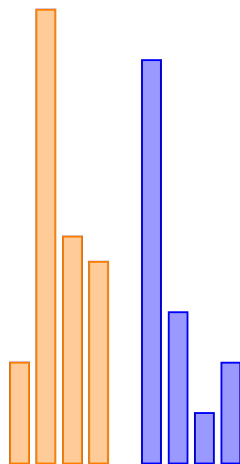


Call-Stack

2 9 5 4 8 3 1 2

Tiefe 0: Anfangszustand

Merge Sort

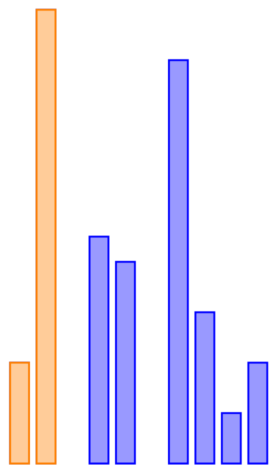


2 9 5 4 8 3 1 2

Tiefe 1: Divide

Call-Stack

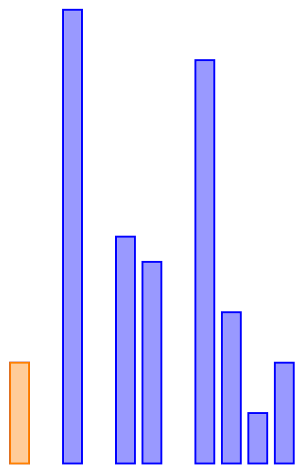
Merge Sort



Tiefe 2: Divide

Call-Stack

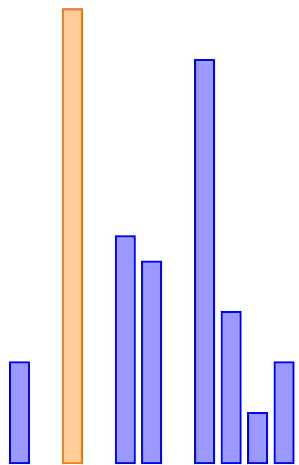
Merge Sort



Tiefe 3: Trivialfall

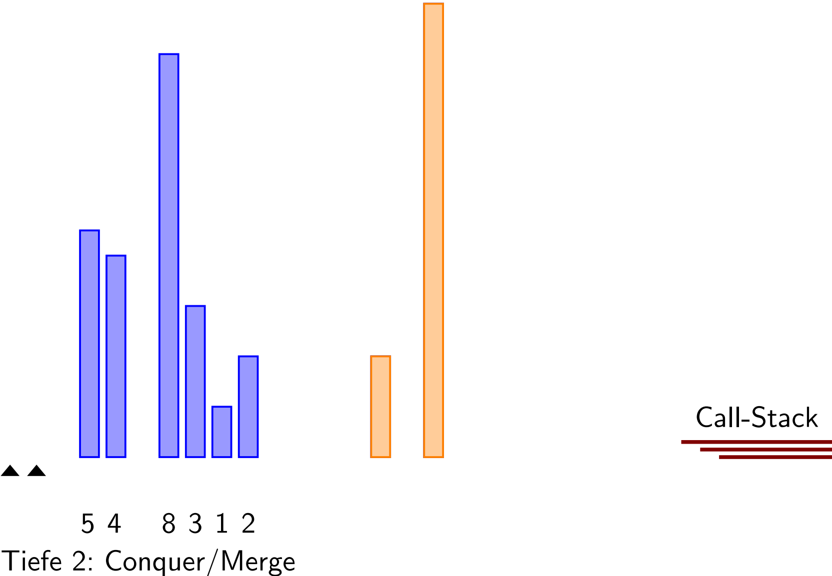
Call-Stack

Merge Sort

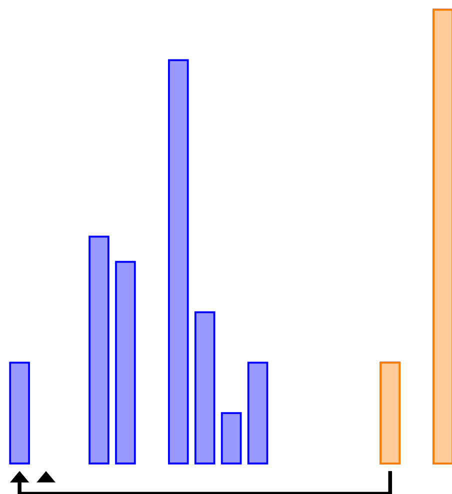


Call-Stack

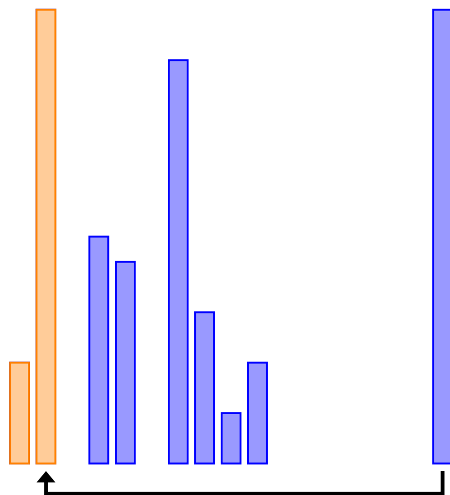
Merge Sort



Merge Sort

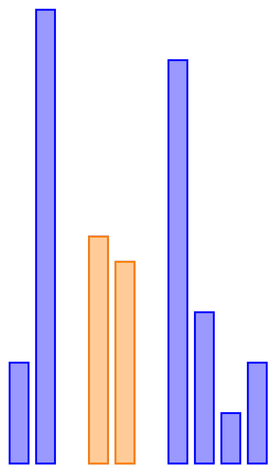


Merge Sort



Call-Stack

Merge Sort

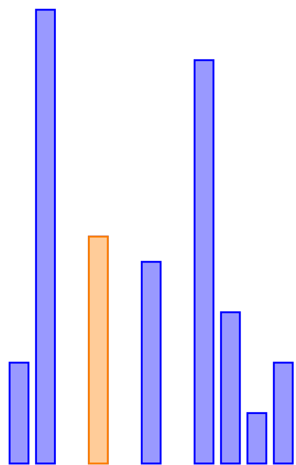


2 9 5 4 8 3 1 2

Tiefe 2: Divide

Call-Stack

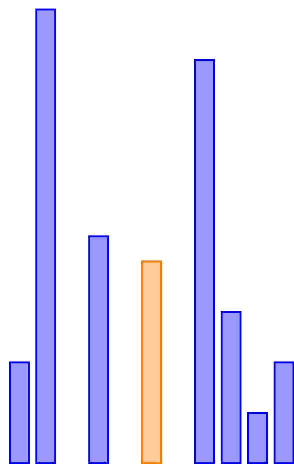
Merge Sort



Tiefe 3: Trivialfall

Call-Stack

Merge Sort

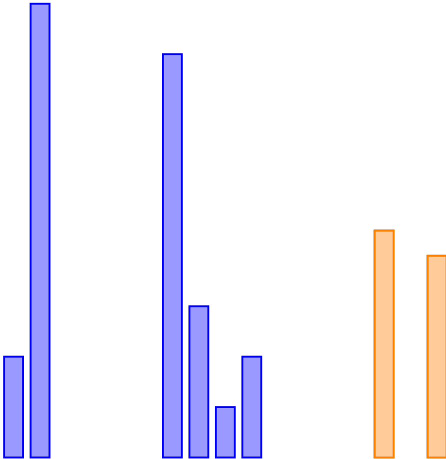


2 9 5 4 8 3 1 2

Tiefe 3: Trivialfall

Call-Stack

Merge Sort

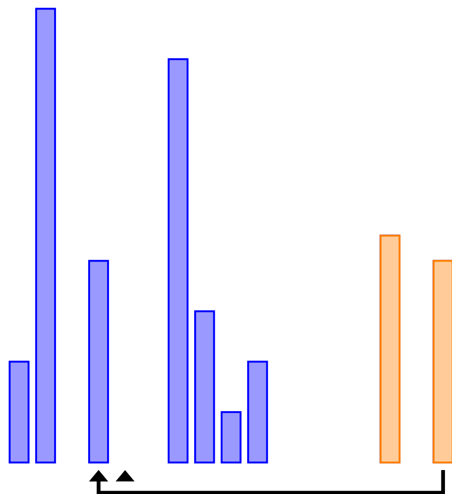


2 9 8 3 1 2

Tiefe 2: Conquer/Merge

Call-Stack

Merge Sort

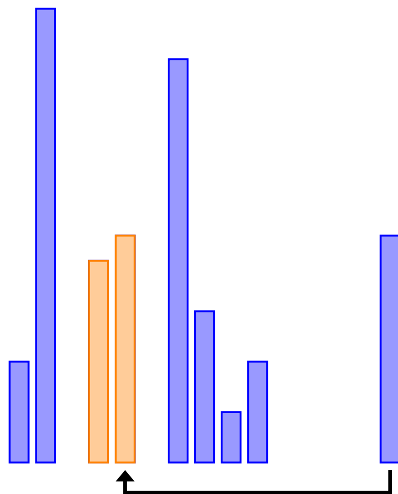


2 9 4 8 3 1 2

Tiefe 2: Conquer/Merge

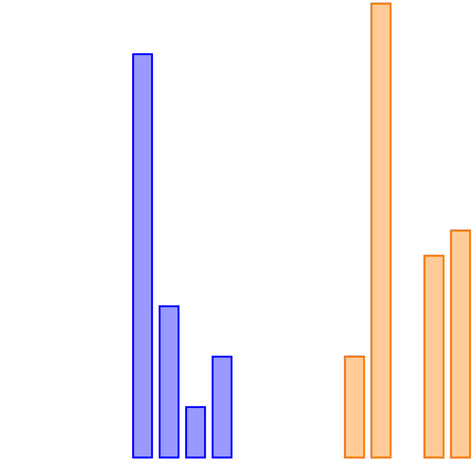
Call-Stack

Merge Sort



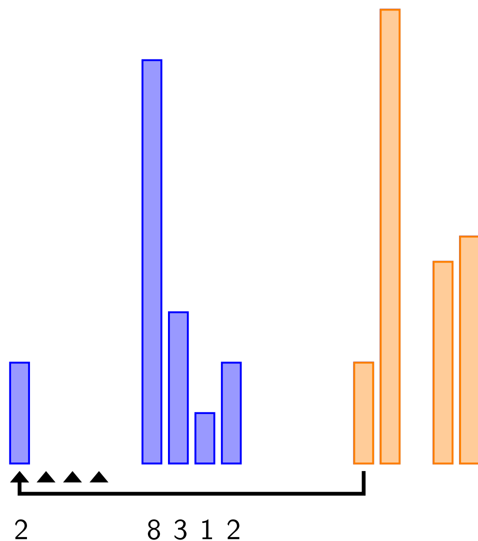
Call-Stack

Merge Sort



Call-Stack

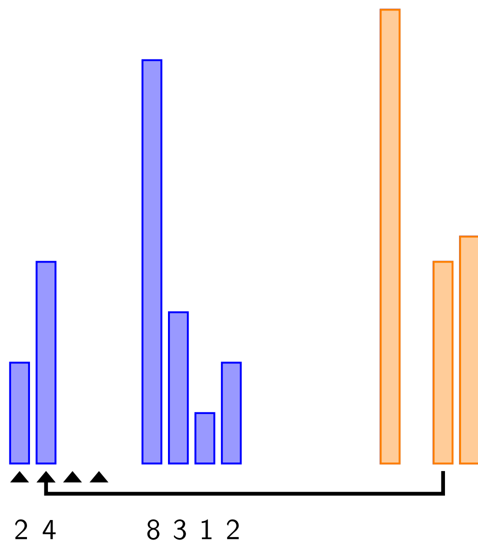
Merge Sort



Tiefe 1: Conquer/Merge

Call-Stack

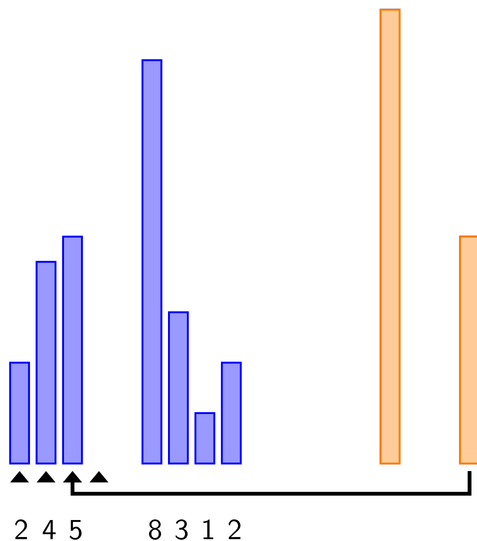
Merge Sort



Tiefe 1: Conquer/Merge

Call-Stack

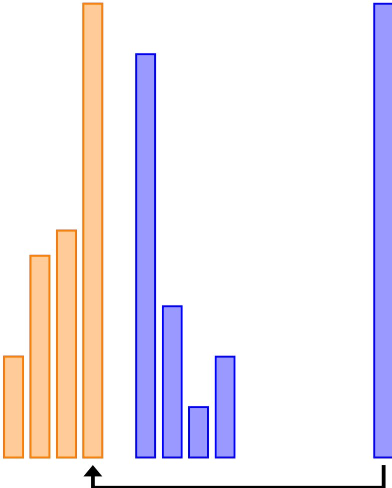
Merge Sort



Tiefe 1: Conquer/Merge

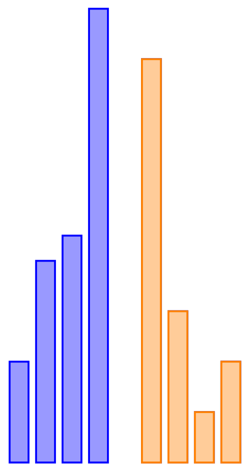
Call-Stack

Merge Sort



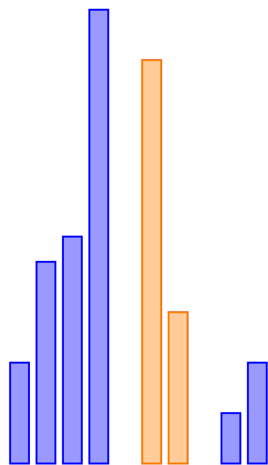
Call-Stack

Merge Sort



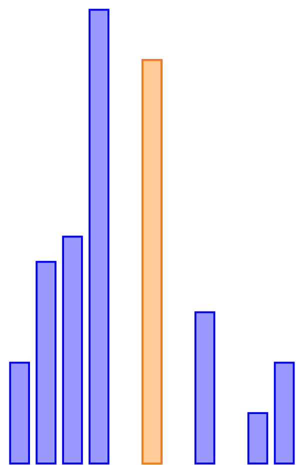
Call-Stack

Merge Sort



Call-Stack

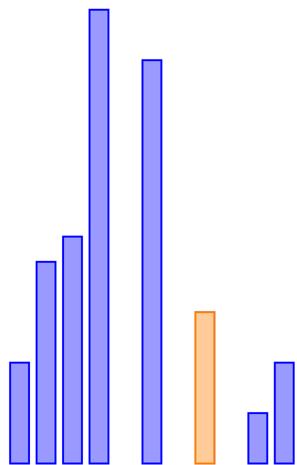
Merge Sort



Tiefe 3: Trivialfall

Call-Stack

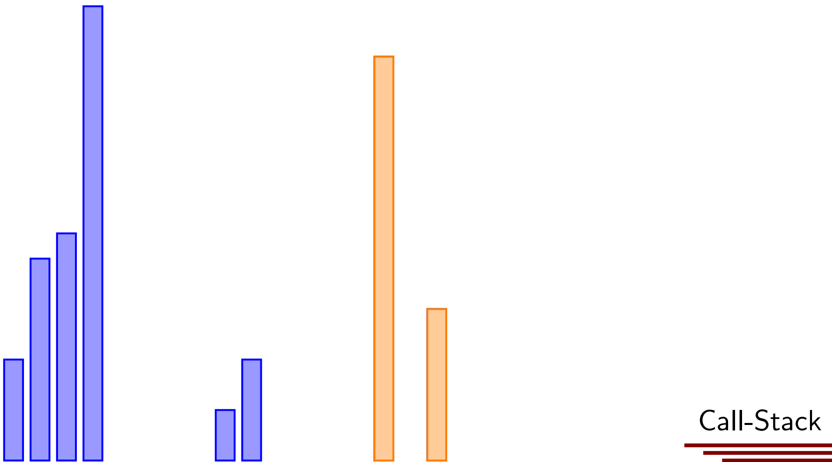
Merge Sort



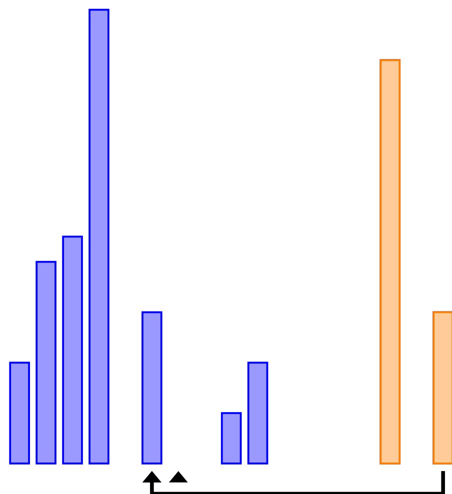
Tiefe 3: Trivialfall

Call-Stack

Merge Sort



Merge Sort

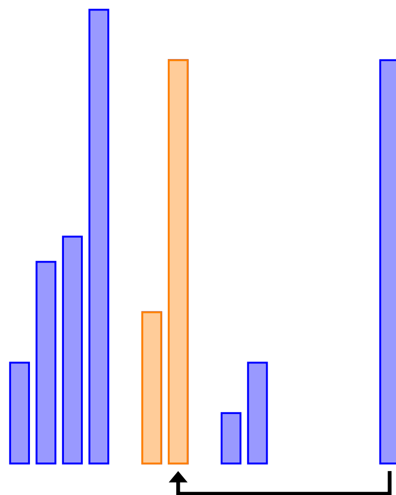


2 4 5 9 3 1 2

Tiefe 2: Conquer/Merge

Call-Stack

Merge Sort

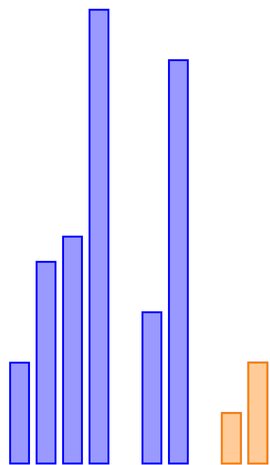


2 4 5 9 3 8 1 2

Tiefe 2: Conquer/Merge

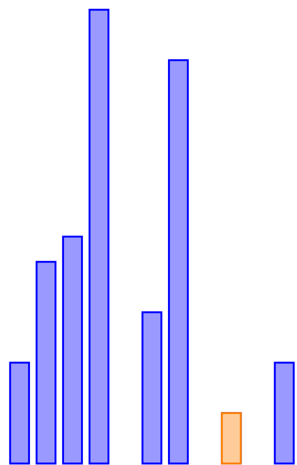
Call-Stack

Merge Sort



Call-Stack

Merge Sort

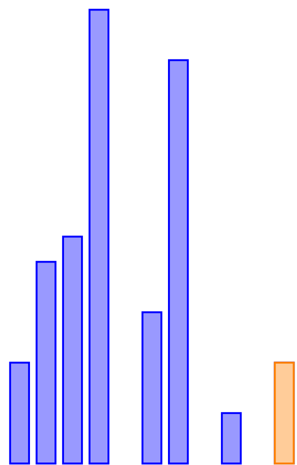


2 4 5 9 3 8 1 2

Tiefe 3: Trivialfall

Call-Stack

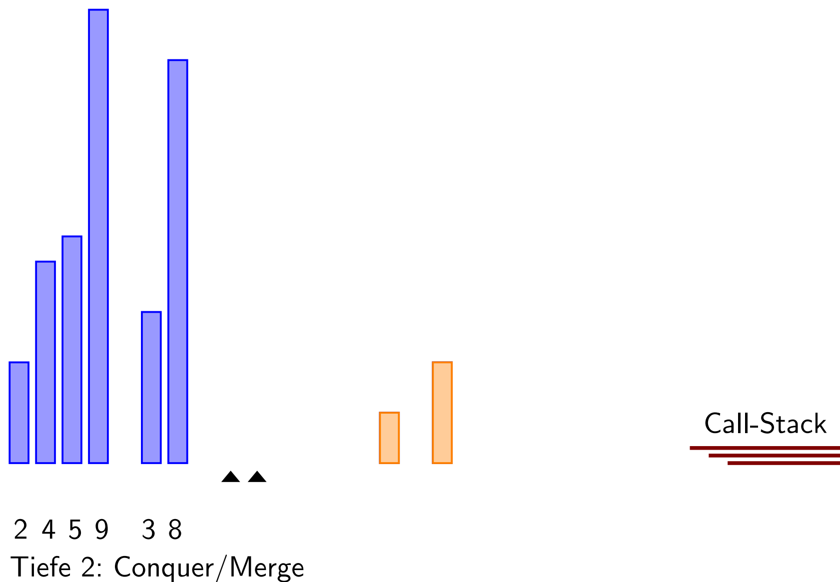
Merge Sort



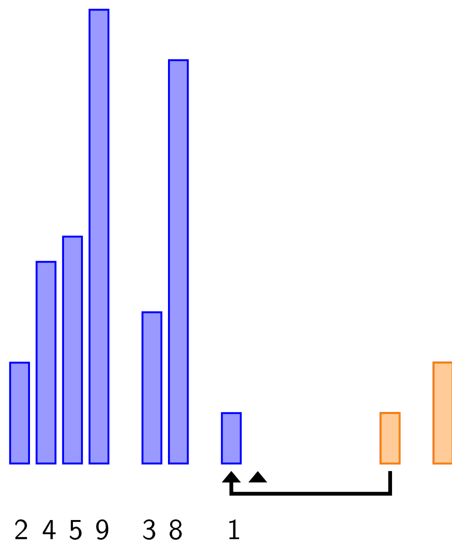
Tiefe 3: Trivialfall

Call-Stack

Merge Sort

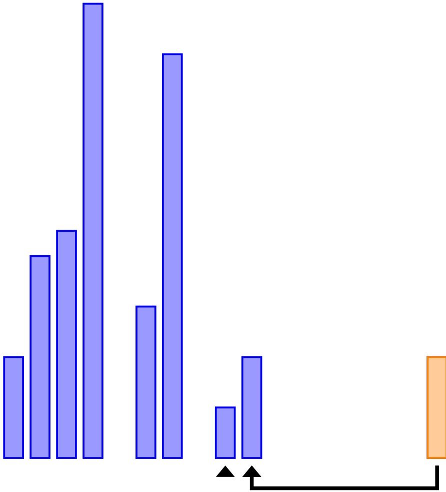


Merge Sort



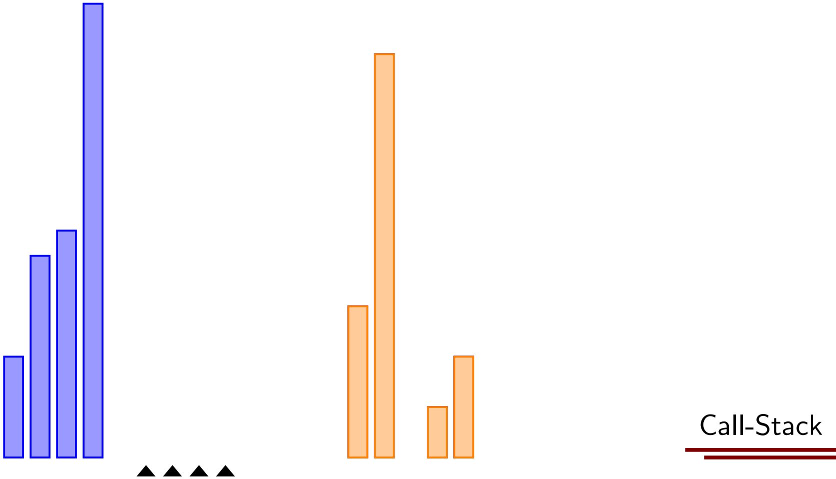
Call-Stack

Merge Sort



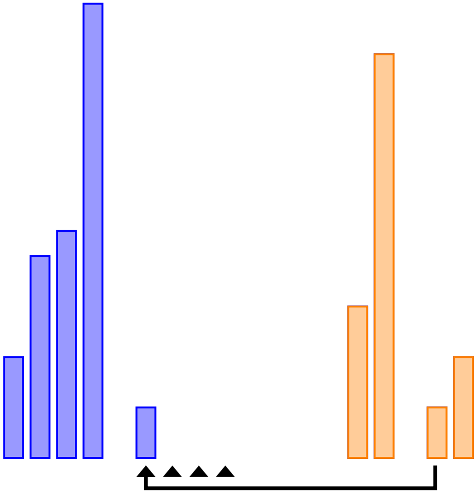
Call-Stack

Merge Sort



Tiefe 1: Conquer/Merge

Merge Sort

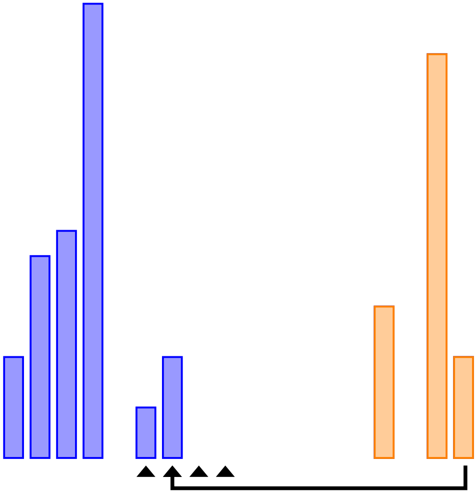


2 4 5 9 1

Tiefe 1: Conquer/Merge

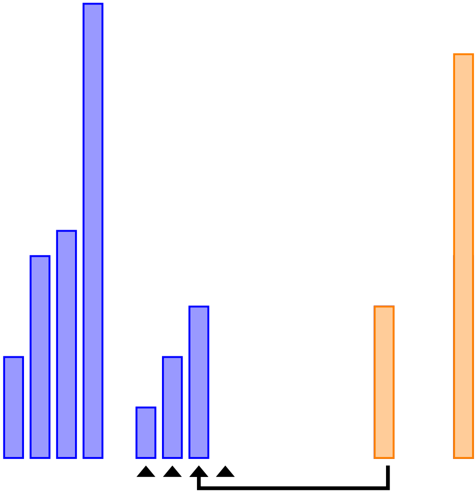
Call-Stack

Merge Sort



Call-Stack

Merge Sort

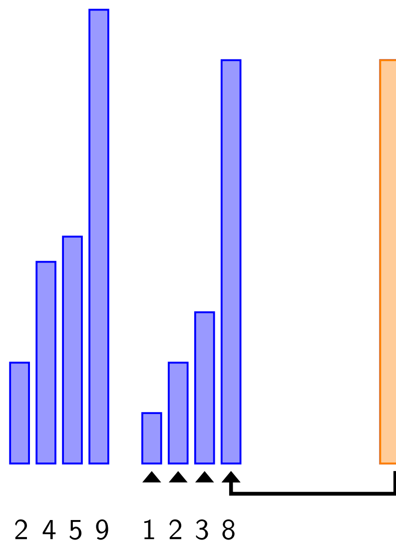


2 4 5 9 1 2 3

Tiefe 1: Conquer/Merge

Call-Stack

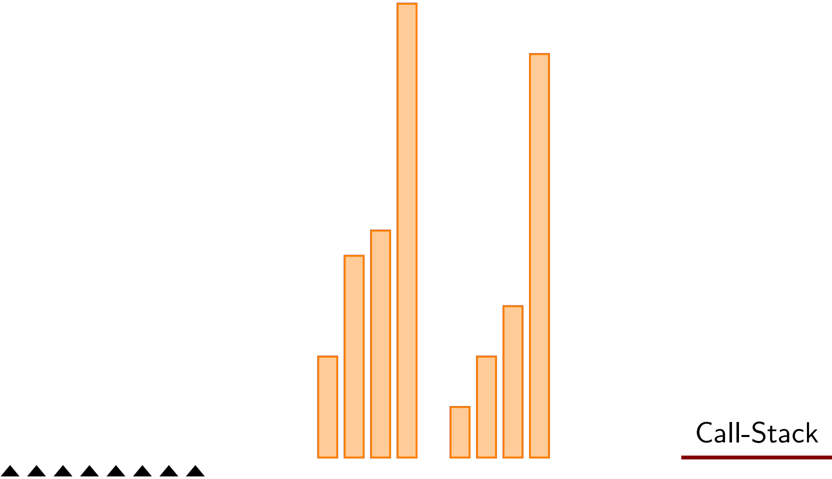
Merge Sort



Tiefe 1: Conquer/Merge

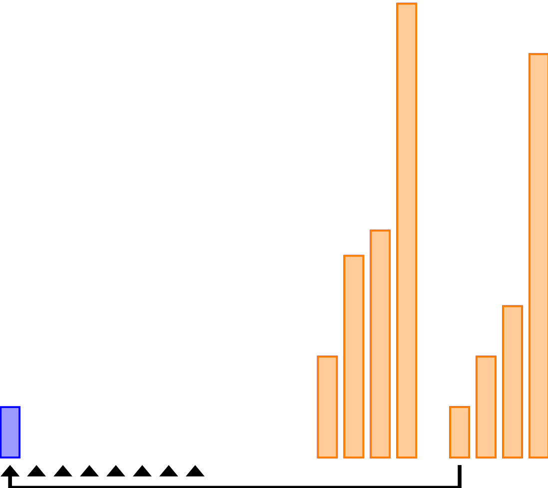
Call-Stack

Merge Sort



Tiefe 0: Conquer/Merge

Merge Sort

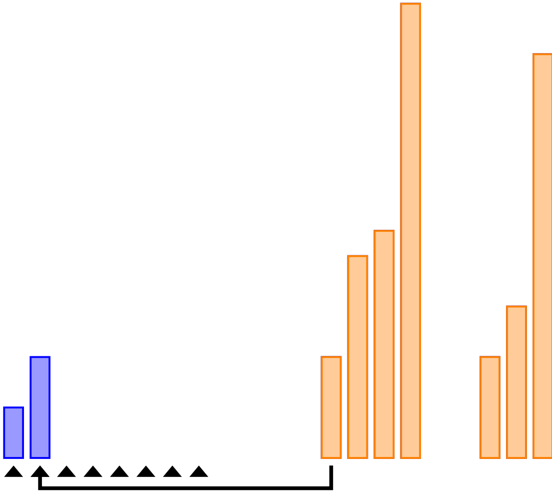


Call-Stack

1

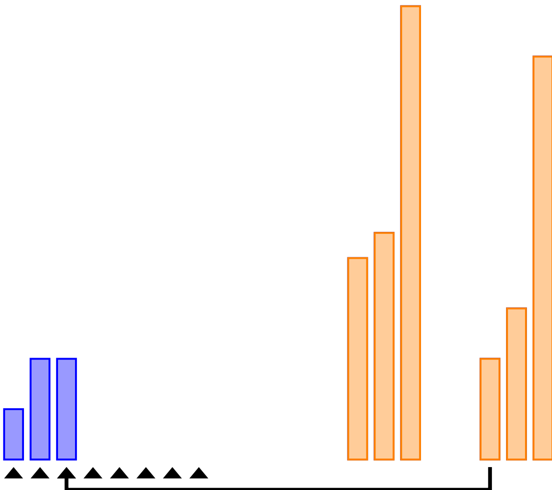
Tiefe 0: Conquer/Merge

Merge Sort



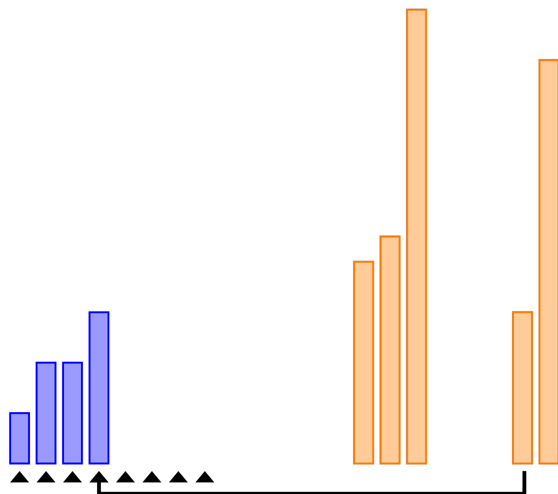
Call-Stack

Merge Sort



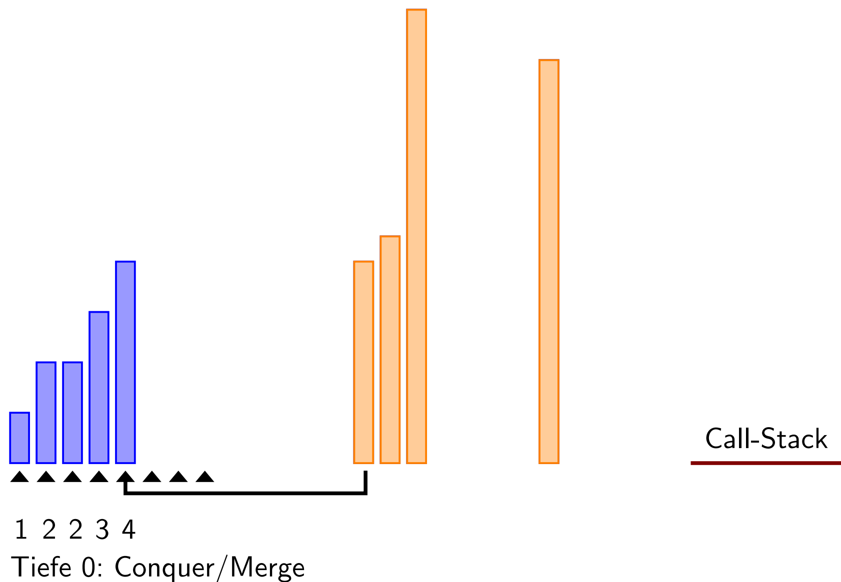
Tiefe 0: Conquer/Merge

Merge Sort

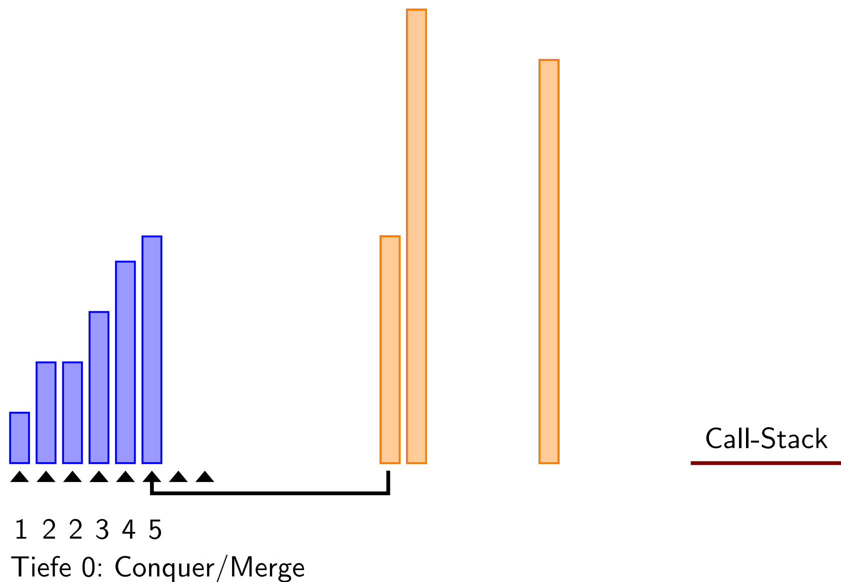


Call-Stack

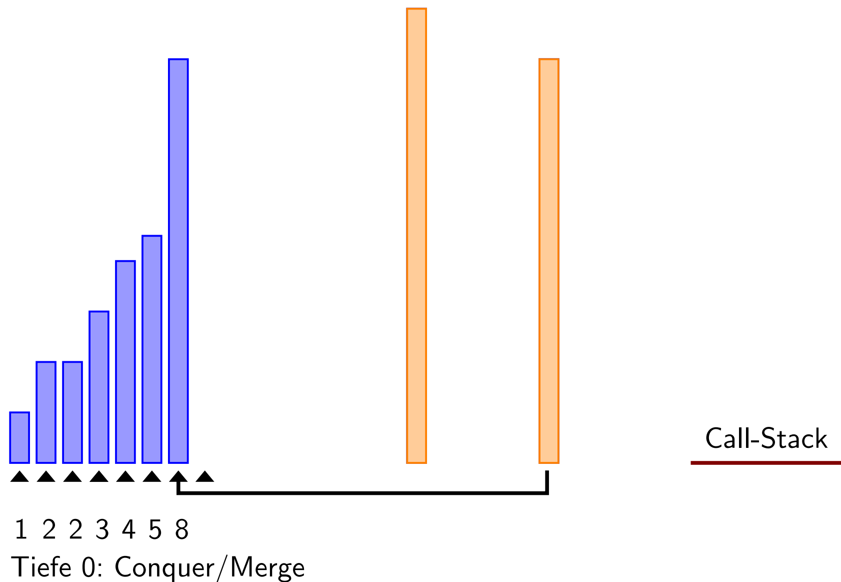
Merge Sort



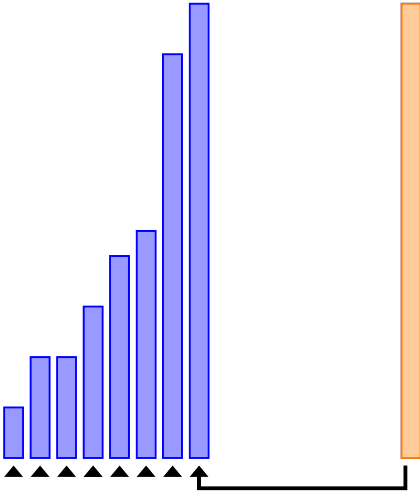
Merge Sort



Merge Sort



Merge Sort



Call-Stack

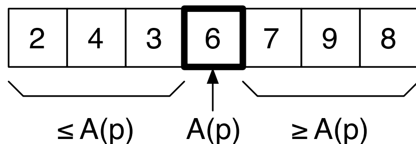
Divide and Conquer: QuickSort

QuickSort:

- Muster: Divide and Conquer
- in-place (ohne extra Speicherbedarf)

Hauptidee:

- teile Feld A mittels Pivot-Element $A(p)$ in zwei Teile
- links vom Pivot-Element $A(p)$ sind alle Elemente kleiner als $A(p)$
- rechts vom Pivot-Element $A(p)$ sind alle Elemente größer als $A(p)$



Quick Sort: Hauptfunktion

QuickSort(A, u, o)**:**

if $(o > u)$ $\{$

// Pivot-Bestimmung nach beliebiger Strategie

$p = \text{PivotIndex}(A)$;

// Divide: Partitionierung relativ zum Pivot

$pn = \text{Partition}(A, u, o, p)$;

// Rekursion: Lösung der Teilprobleme

$\text{QuickSort}(A, u, pn - 1)$;

$\text{QuickSort}(A, pn + 1, o)$;

// Conquer erfolgt implizit dank In-Place-Sortierung

$\}$

Quick Sort: Partitionierung

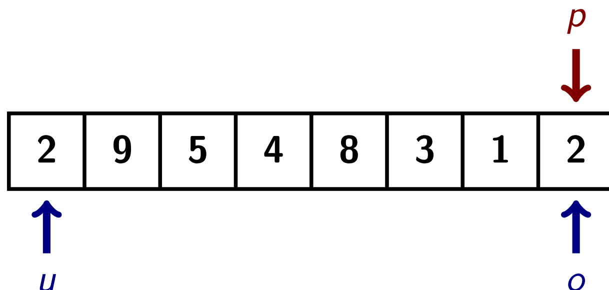
Partition(A, u, o, p)**:**

$pn = u$;
$pv = A[p]$;
Swap(A, p, o);

for $i = u$ **to** $o - 1$ $\{$
  if $(A[i] \leq pv)$ $\{$
    Swap(A, pn, i);
    $pn = pn + 1$;
  $\}$
$\}$

Swap(A, o, pn);
return pn;

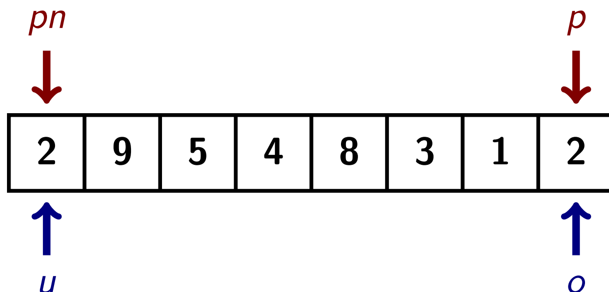
Quick Sort: Partitionierung



Anfangszustand

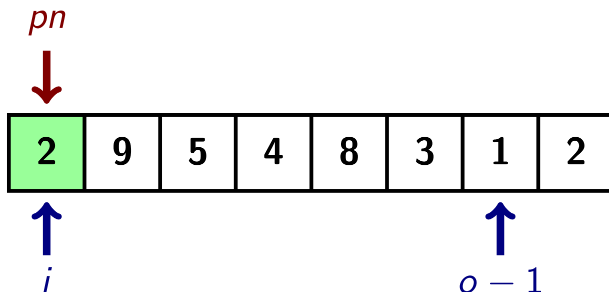
Quick Sort: Partitionierung

$pv = 2$ $\leq pv$ $> pv$



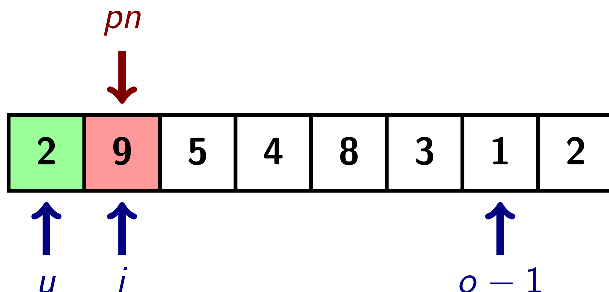
Pivot ans Ende

Quick Sort: Partitionierung



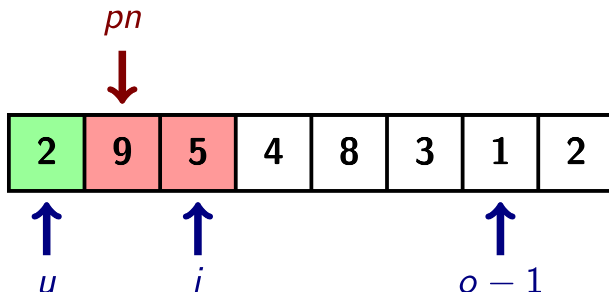
Schleifendurchlauf

Quick Sort: Partitionierung



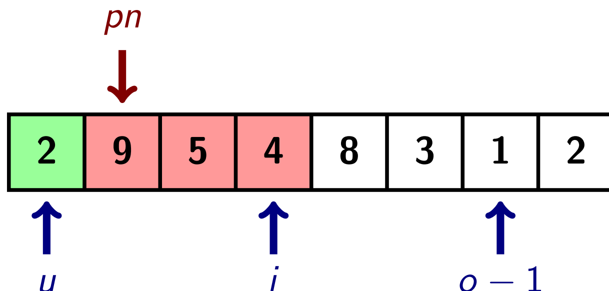
Schleifendurchlauf

Quick Sort: Partitionierung



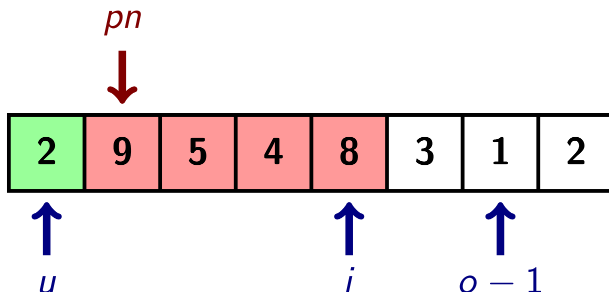
Schleifendurchlauf

Quick Sort: Partitionierung



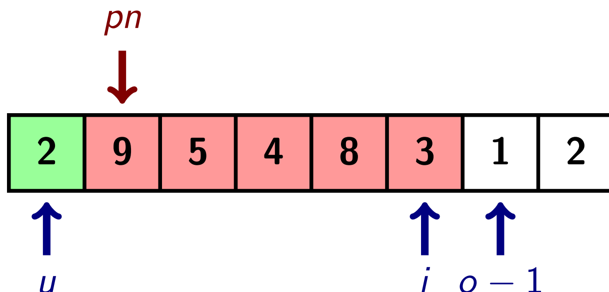
Schleifendurchlauf

Quick Sort: Partitionierung



Schleifendurchlauf

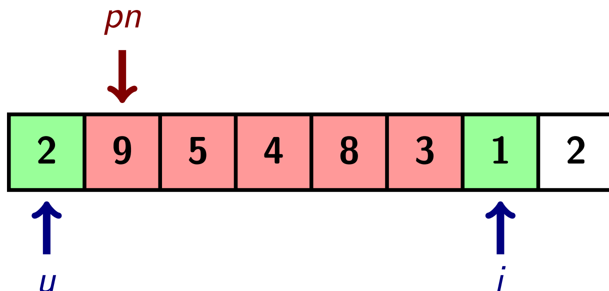
Quick Sort: Partitionierung



Schleifendurchlauf

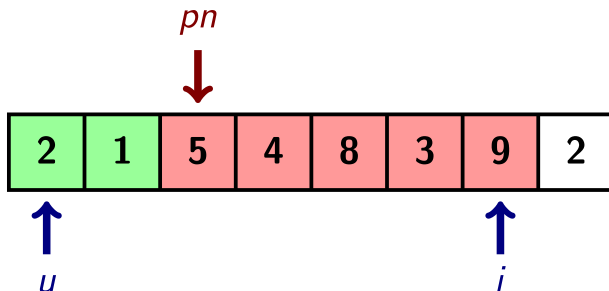
Quick Sort: Partitionierung

$pv = 2$ $\leq pv$ $> pv$



Schleifendurchlauf

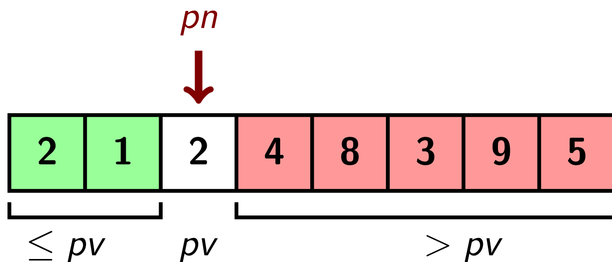
Quick Sort: Partitionierung



Schleifendurchlauf

Quick Sort: Partitionierung

$pv = 2$ $\leq pv$ $> pv$



Pivot zurück, sortiere grünen und roten Teil rekursiv

Quick Sort

Sortierung linker Teil: ≤ 2

2	1

2

4	8	3	9	5

Pivot-Wahl Element 1

Quick Sort

Sortierung linker Teil: ≤ 2

Partitionierung nach Schleifendurchlauf

Quick Sort

Sortierung linker Teil: ≤ 2

Trivial-Schritt: Partitionierung eines ein elementigen Arrays

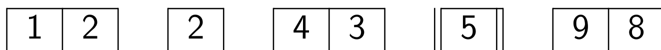
Quick Sort

Sortierung rechter Teil: > 2

Pivot-Wahl Element 5

Quick Sort

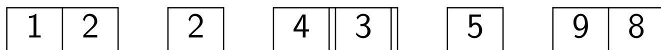
Sortierung rechter Teil: > 2



Partitionierung nach Schleifendurchlauf

Quick Sort

Sortierung rechter Teil: > 2 **und dessen linker Teil** ≤ 5



Pivot-Wahl Element 3

Quick Sort

Sortierung rechter Teil: > 2 **und dessen linker Teil** ≤ 5

Partitionierung nach Schleifendurchlauf

Quick Sort

Sortierung rechter Teil: > 2 **und dessen linker Teil** ≤ 5

Trivial-Schritt: Partitionierung eines ein elementigen Arrays

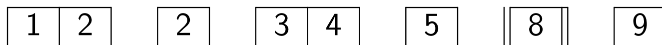
Quick Sort

Sortierung rechter Teil: > 2 **und dessen rechter Teil** > 5

Pivot-Wahl Element 8

Quick Sort

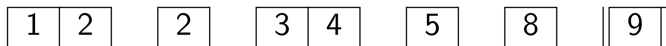
Sortierung rechter Teil: > 2 **und dessen rechter Teil** > 5



Partitionierung nach Schleifendurchlauf

Quick Sort:

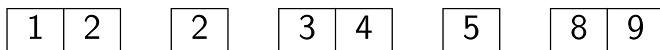
Sortierung rechter Teil: > 2 **und dessen rechter Teil** > 5



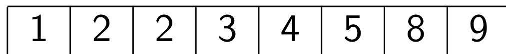
Trivial-Schritt: Partitionierung eines ein elementigen Arrays

Quick Sort

Implizite Sortierung ohne eigentlichen "Merge" Schritt



Ergebnis



Merge-Sort

Eingabe: $A = \{d, b, f, g, e, a, c\}$

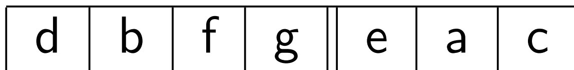
```
MergeSort(A):
    if (A ein-elementig) {
        return A;
    } else {
        // Divide: Erzeugung von Teilproblemen
        (A1; A2) = Split(A);

        // Rekursion: Lösung der Teilprobleme
        A1 = MergeSort(A1);
        A2 = MergeSort(A2);

        // Conquer: Kombination der Teillösungen
        return Merge(A1, A2);
    }
```

- Eingabelänge n ungerade
  $\rightarrow A = a_1, ..., a_n \rightarrow A_{left} = a_1, ..., a_{\frac{n+1}{2}}, A_{right} = a_{\frac{n+1}{2}+1}, ..., a_n$
- lexikographische Ordnung für Sortierung von Buchstaben

Merge-Sort



Divide Schritt für $n = 7$, bei Index $\frac{n+1}{2} = 4$

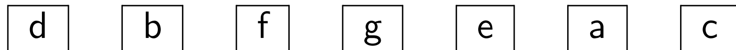
Merge-Sort

Rekursiver Divide Schritt für linkes und rechtes Teilarray

Merge-Sort

Rekursiver Divide Schritt für 4 Teilarrays

Merge-Sort



Conquer/Merge Schritt, nach Aufteilung in minimales Teilproblem

Merge-Sort

Conquer/Merge der jeweiligen Teilarrays mit Sortierung in Arrays der Größe ≤ 2

Merge-Sort

Conquer/Merge der jeweiligen Teilarrays der Größe ≤ 2 mit Sortierung in Arrays der Größe ≤ 4

Merge-Sort

Ergebnis

a	b	c	d	e	f	g

letzter Merge Schritt zu Array der Originalgröße

Quick-Sort

Ausgangssituation:

9	8	7	6	5	4	3	2	1

Pivot-Wahl: erstes Element

Quick-Sort

Sortierung linker Teil: ≤ 9

1	8	7	6	5	4	3	2		9

Partitionierung nach Schleifendurchlauf

Quick-Sort

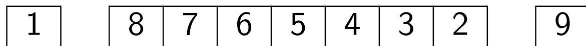
Sortierung linker Teil: ≤ 9

1	8	7	6	5	4	3	2		9

Pivot-Wahl erstes Element 1

Quick-Sort

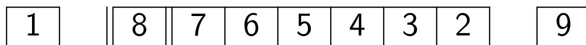
Sortierung linker Teil: ≤ 9



Partitionierung nach Schleifendurchlauf

Quick-Sort

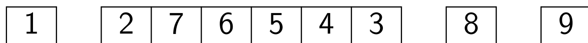
Sortierung linker Teil: ≤ 9 **und dessen rechter Teil** > 1



Pivot-Wahl erstes Element 8

Quick-Sort

Sortierung linker Teil: ≤ 9 **und dessen rechter Teil** > 1



Partitionierung nach Schleifendurchlauf

Arraygrösse wird bei jeder Partitionierung nur um eins reduziert

Worst-Case Laufzeit: $O(n^2)$

Quick-Sort

Ausgangssituation:

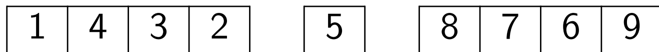
9	8	7	6	5	4	3	2	1

Pivot-Wahl: 5

mittlere Element (bzw. Index $\frac{n}{2} + 1$ bei gerader Eingabelänge)

Quick-Sort

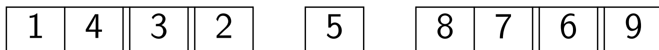
Sortierung linker Teil und rechter Teil rekursiv



Partitionierung nach Schleifendurchlauf

Quick-Sort

Sortierung linker Teil und rechter Teil rekursiv



Pivot-Wahl linker Teil: 3 und rechter Teil: 6

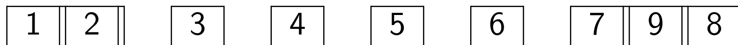
Quick-Sort

Sortierung linker Teil und rechter Teil rekursiv

Partitionierung nach Schleifendurchlauf

Quick-Sort

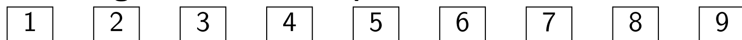
Sortierung linker Teil und rechter Teil rekursiv



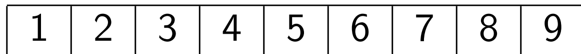
Pivot-Wahl 2 im Teil ≤ 3 und 9 im Teil > 6
Trivialfall im Teil > 3

Quick-Sort

Auflösung in minimale Teilprobleme

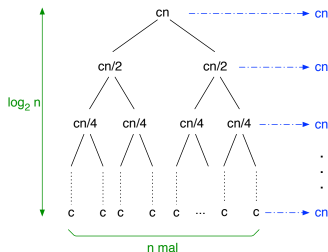


Ergebnis:



ausgewogenere Partitionierung und bessere Laufzeit $O(nlogn)$

Komplexität $O(nlogn)$



- Baum hat Höhe $\log_2 n$, also $\log_2 n + 1$ Ebenen
- pro Ebene cn Kosten

$$\rightarrow \text{Gesamtkosten } cn(\log_2 n + 1) = cn \log_2 n + cn$$

$$\longrightarrow T(n) = \Theta(n \log n)$$

Sortier-Algorithmen Zusammenfassung

- Insertion Sort
  - in-place
  - Komplexität $O(n^2)$, best case $O(n)$
- Selection Sort
  - in-place
  - Komplexität $O(n^2)$
- Merge Sort
  - benötigt zusätzlichen Speicher
  - Komplexität $O(nlogn)$
- Quick Sort
  - in-place
  - Komplexität im Mittel $O(nlogn)$, worst case $O(n^2)$