Übung zu Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Komplexität von Programmen

- Laufzeit kann näherungsweise bestimmt werden
- Vereinfachendes RAM-Modell
- Laufzeitfunktion trotzdem umständlich aufzustellen

- Betrachte *Selection Sort*!

Komplexität von **Swap**

Swap(A, i, j)**:**
$k = A[j]$;
$A[j] = A[i]$;
$A[i] = k$;

- Im RAM-Modell 3 Anweisungen
- Damit offensichtlich $O(1)$ (mit $c = 3$, $n_0 = 1$)

Komplexität von **IndexOfMin**

```
1   i = j;
2   k = j + 1;
3   while (k < n) {
4       if (A[k] < A[i]) {
5           i = k;
        }
6       k = k + 1;
    }
```

Zeile	Kosten	Häufigkeit
1	c_1	1
2	c_2	1
3	c_3	$n - j$
4	c_4	$n - j - 1$
5	c_5	t_k
6	c_6	$n - j - 1$

- Laufzeitfunktion

$$
\begin{aligned}
T(n) &= 1 + 1 + (n - j) + (n - j - 1) + t_k + (n - j - 1) \\
&= 3 \cdot n - 3 \cdot j + t_k
\end{aligned}
$$

- t_k ist die Anzahl der Verzweigungen bei 4 bzw. Durchführungen von 5

Best und Worst Case der **IndexOfMin**

- Im *besten Fall* wird Zeile 5 nie ausgeführt. Das passiert, wenn A bereits *aufsteigend* sortiert ist.
- Im *schlechtesten Fall* wird Zeile 5 immer ausgeführt. Das passiert, wenn A bereits *absteigend* sortiert ist.

- Im schlechtesten Fall gilt $t_k = n - j - 1$, und damit

$$T(n) = 4 \cdot n - 4 \cdot j - 1$$

- Somit ergibt sich sogar dann die Komplexität $O(n)$!
- Inoffiziell ergibt sich das direkt aus der **for**-Schleife...

Komplexität von **SelectionSort**

```
1   for j = 0 to n − 2 {
2       i = IndexOfMin(A, j);
3       if (j ≠ i) {
4           Swap(A, i, j);
        }
    }
```

Zeile	Kosten	Häufigkeit
1	c_1	$n-1$
2	c_2	$(n-1)\cdot O(n)$
3	c_3	$n-1$
4	c_4	$t_j \cdot O(1)$

- Laufzeitfunktion ungefähr

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

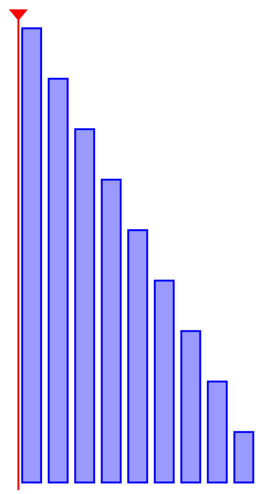
- t_j ist die Anzahl der Vertauschungen bei 4

Best und Worst Case der **SelectionSort**

- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.
- Maximale Anzahl von Vertauschungen, wenn A das maximale Element am Anfang und dann eine aufsteigende Ordnung hat.

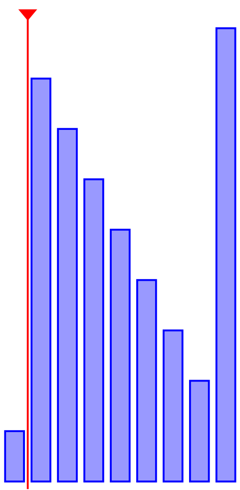
- Sind Vertauschungen oder Updates des minimalen Index teurer?
- Die Laufzeit von Selection Sort ist erfahrungsgemäß eher unabhängig von Vorsortierungen!

Selection Sort: Absteigende Sortierung



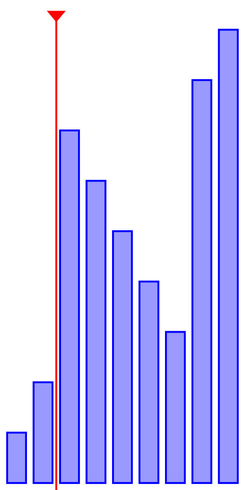
$j = 0$: 0 Updates, 0 Vertauschungen

Selection Sort: Absteigende Sortierung



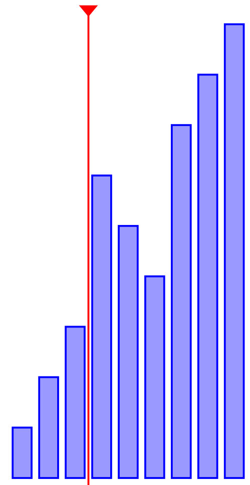
$j = 1$: 8 Updates, 1 Vertauschung

Selection Sort: Absteigende Sortierung



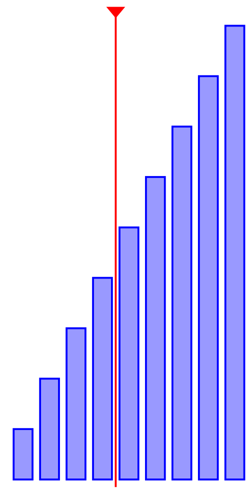
$j = 2$: 14 Updates, 2 Vertauschungen

Selection Sort: Absteigende Sortierung



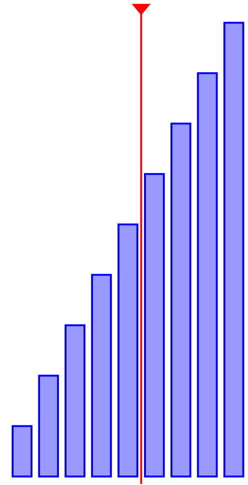
$j = 3$: 18 Updates, 3 Vertauschungen

Selection Sort: Absteigende Sortierung



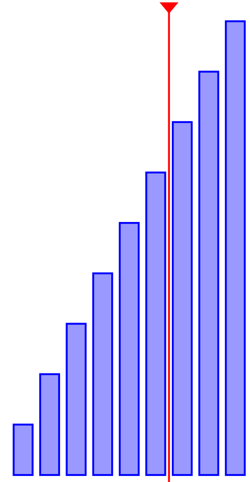
$j = 4$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



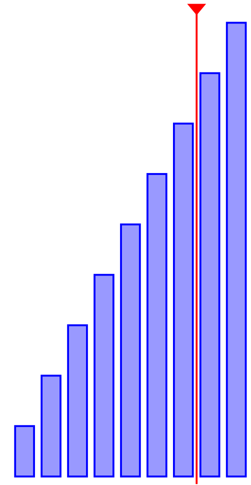
$j = 5$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



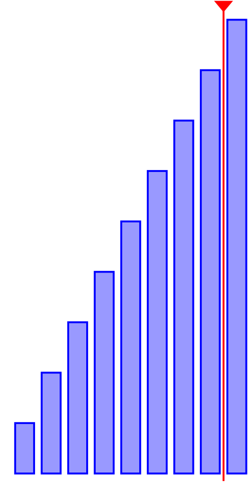
$j = 6$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



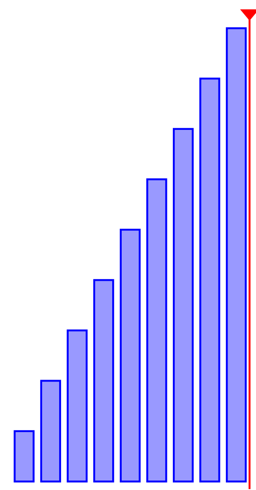
$j = 7$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



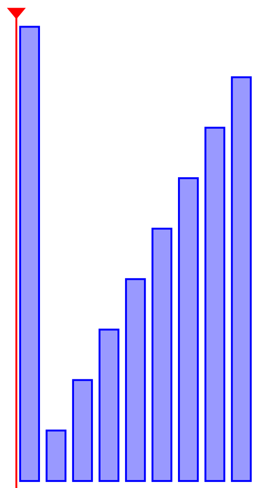
$j = 8$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



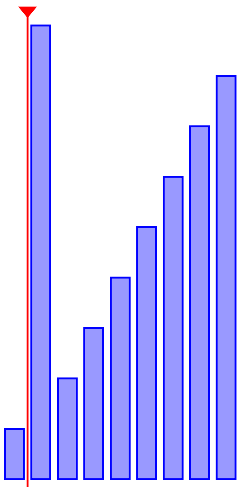
$j = 9$: 20 Updates, 4 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



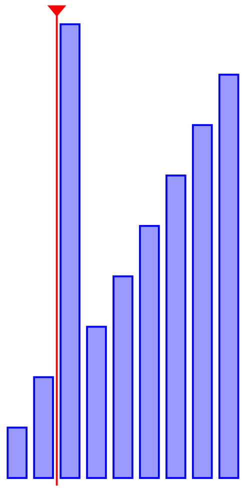
$j = 0$: 0 Updates, 0 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



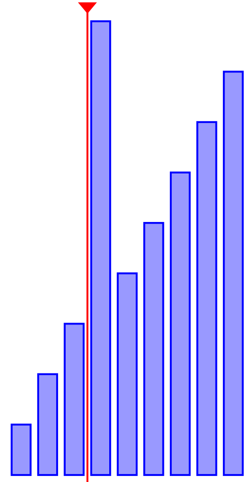
$j = 1$: 1 Update, 1 Vertauschung

Selection Sort: Größtes Element vorne, dann aufsteigend



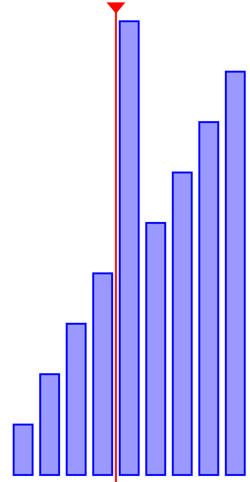
$j = 2$: 2 Updates, 2 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



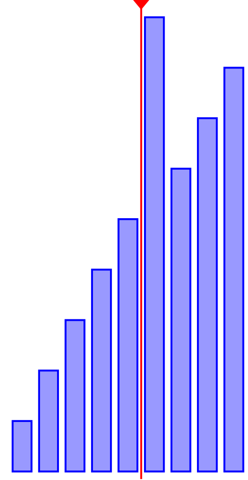
$j = 3$: 3 Updates, 3 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



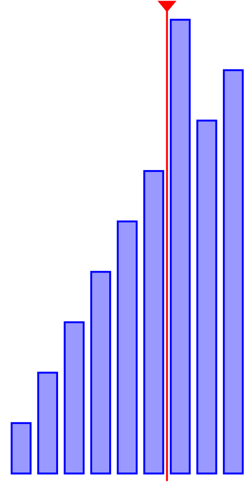
$j = 4$: 4 Updates, 4 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



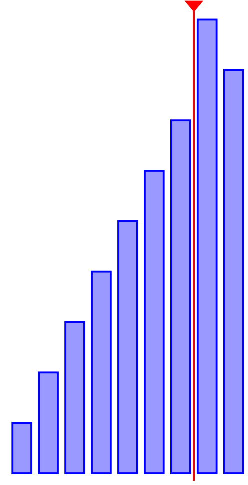
$j = 5$: 5 Updates, 5 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



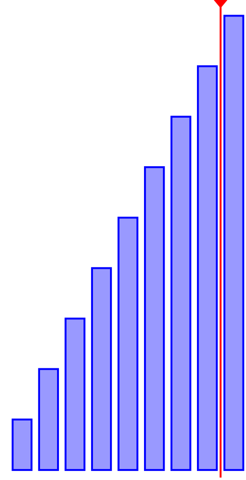
$j = 6$: 6 Updates, 6 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



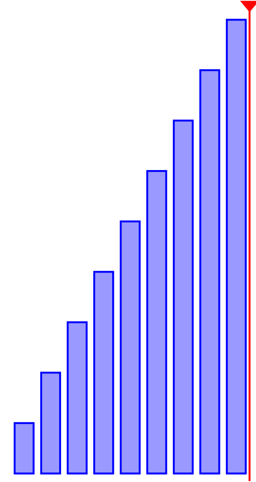
$j = 7$: 7 Updates, 7 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



$j = 8$: 8 Updates, 8 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



$j = 9$: 8 Updates, 8 Vertauschungen

Zusammenfassung

- Komplexitätsabschätzung oft intuitiv
- Beispiel: Eingabelänge n, aber nur lokale Änderung (etwa Anfügung an verkettete Liste) $\Rightarrow O(1)$
- Beispiel: Eingabelänge n, und **for**-Schleife (etwa Minimum in Array) $\Rightarrow O(n)$
- Wichtig ist vor allem Verständnis für die Bedeutung!
  (s. Tabelle zu Rechenzeiten in Vorlesung)

- Einfache Algorithmen/Probleme können extrem komplex zu lösen sein!
  - Traveling Salesman Problem
  - Primfaktorzerlegung

Landau-Symbole

Abschätzung des Wachstumsverhaltens einer (komplexeren) Funktion durch eine andere (intuitivere) Funktion.

$$
\begin{aligned}
\Theta(g) \quad := \quad \{f : \mathbb{R} \to \mathbb{R} \ : \ & \text{es existieren } c_1, c_2 > 0 \text{ und } n_0 \in \mathbb{N}, \\
& \text{so dass für alle } n \geq n_0 \text{ gilt:} \\
& 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}
\end{aligned}
$$

$$
\begin{aligned}
O(g) \quad := \quad \{f : \mathbb{R} \to \mathbb{R} \ : \ & \text{es existieren } c > 0 \text{ und } n_0 \in \mathbb{N}, \\
& \text{so dass für alle } n \geq n_0 \text{ gilt:} \\
& 0 \leq f(n) \leq c \cdot g(n)\}
\end{aligned}
$$

$\Theta(g)$ ist die *asymptotisch scharfe* und $O(g)$ die *asymptotisch obere Schranke*.

Bedeutung der konstanten Faktoren

Die Worst-Case-Komplexität eines Algorithmus sei $\Theta(n)$*. Nachdem auch* $2n = \Theta(n)$*, entspricht die Dauer eines Worst-Case-Beispiels der Länge* $2n$ *ungefähr der Dauer eines Worst-Case-Beispiels der Länge* n*.*

- Konstante Faktoren entfallen, nur grundsätzliches Wachstum ist relevant.
- Laufzeitunterschied also unabhänging von n.
- Doppelt großes Beispiel rechnet auch (etwa) doppelt so lange!

Die Behauptung ist **falsch**! □

Iterationen über Code bekannter Komplexität

Sei die Komplexität einer Funktion f bestimmt als $O(n)$. Dann ist die n-malige Ausführung $O(n^2)$.

- $T_f(n) = O(n)$, also gibt es c und n_0, so dass für alle $n \geq n_0$:

$$0 \leq T_f(n) \leq c \cdot n$$

- Bei n-maliger Ausführung muss einfach multipliziert werden:

$$0 \leq n \cdot T_f(n) \leq n \cdot c \cdot n = c \cdot n^2$$

Das ist direkt die Forderung aus der Definition von $O(n^2)$, die Aussage ist **korrekt**! □

Falls $f(n) = \Theta(g(n))$, dann folgt $2^{f(n)} = \Theta(2^{g(n)})$.

- Für $f(n) = n$ und $g(n) = 2 \cdot n$ gilt $f(n) = \Theta(g(n))$.
- Falls Aussage korrekt, erhalten wir aus $2^{f(n)} = \Theta(2^{g(n)})$:

$$0 \leq c_1 \cdot 2^{2 \cdot n} \leq 2^n \leq c_2 \cdot 2^{2 \cdot n}$$

- Teilung durch $2^n > 0$ $(n > 0)$ ergibt:

$$0 \leq c_1 \cdot 2^n \leq 1 \leq c_2 \cdot 2^n$$

- **Widerspruch**, da Folge 2^n unbeschränkt ist, also kein konstantes c_1 gefunden werden kann.

Damit ist die komplette Aussage **falsch**, da mit der getroffenen Auswahl für f und g ein Gegenbeispiel gefunden wurde. □

$n^n = O(2^n)$

- Falls die Aussage wahr ist, dann gilt:

$$0 \leq n^n \leq c \cdot 2^n$$

- Teilung durch $2^n > 0$ $(n > 0)$ ergibt:

$$\left(\frac{n}{2}\right)^n \leq c$$

- **Widerspruch**, da die Folge unbeschränkt ist. Es existiert keine obere Schranke c.

Damit ist die Aussage als **falsch** widerlegt! □

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)

Merge Sort: Hauptfunktion

MergeSort(A)**:**
    if (A ein-elementig) {
        return A;
    } **else** {
        // Divide: Erzeugung von Teilproblemen
        $(A_1; A_2) = \text{Split}(A)$;

        // Rekursion: Lösung der Teilprobleme
        $A_1 = \text{MergeSort}(A_1)$;
        $A_2 = \text{MergeSort}(A_2)$;

        // Conquer: Kombination der Teillösungen
        return Merge(A_1, A_2);
    }

Merge Sort: Kombination

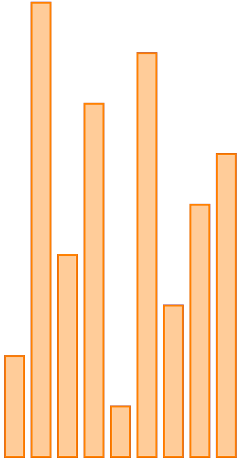
```
Merge(A1, A2):
    B = ∅;

    while (A1 und A2 nicht leer) {
        if (A1[0] ≤ A2[0]) {
            B = push(B, pop(A1));
        } else {
            B = push(B, pop(A2));
        }
    }

    if (A1 nicht leer) {
        B = pushAll(B, A1);
    } else {
        B = pushAll(B, A2);
    }

    return B;
```

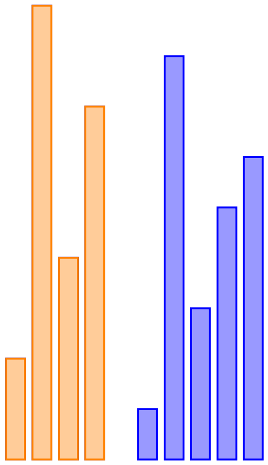
Merge Sort



Call-Stack

Tiefe 0: Anfangszustand

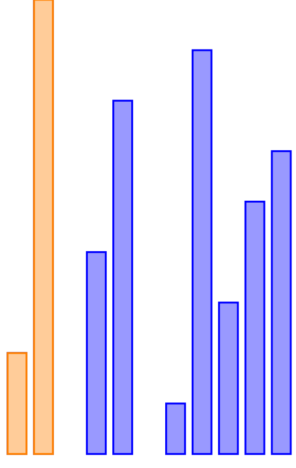
Merge Sort



Call-Stack

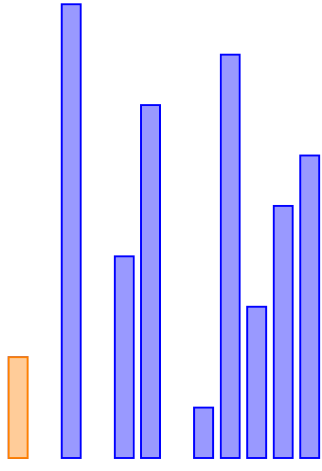
Tiefe 1: Divide

Merge Sort



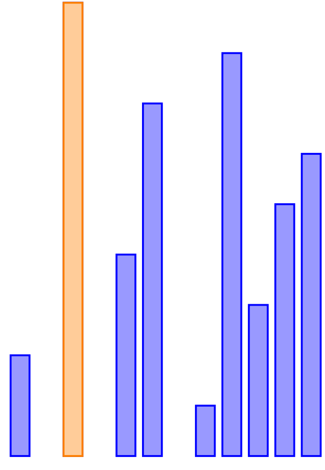
Tiefe 2: Divide

Merge Sort



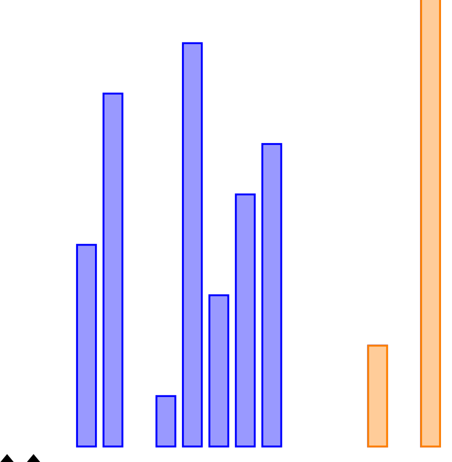
Tiefe 3: Trivialfall

Merge Sort



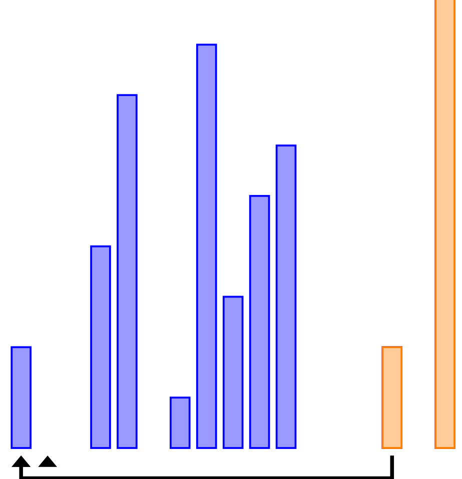
Tiefe 3: Trivialfall

Merge Sort

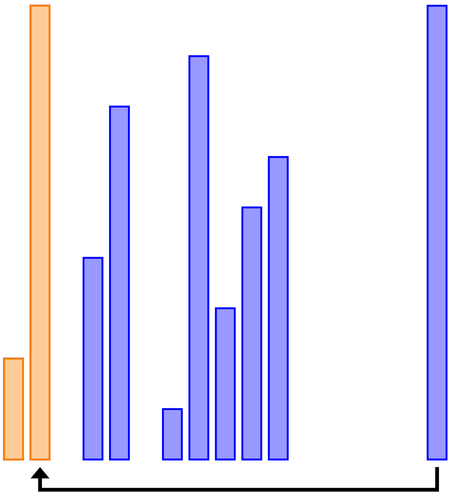


Tiefe 2: Conquer/Merge

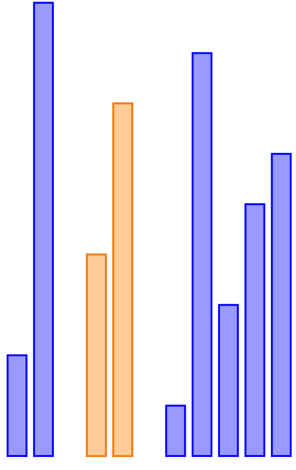
Merge Sort



Merge Sort

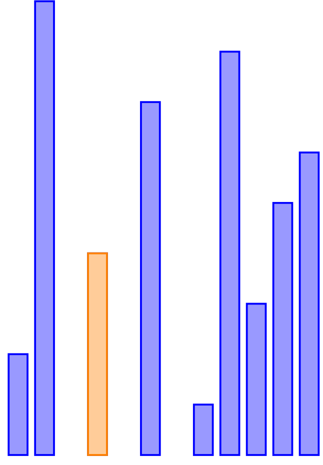


Merge Sort



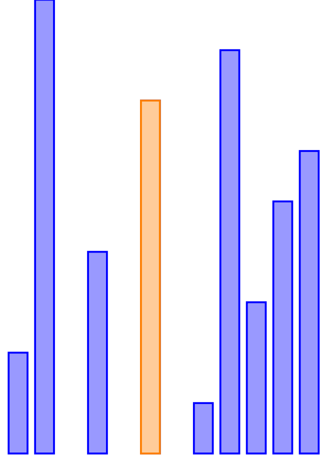
Tiefe 2: Divide

Merge Sort



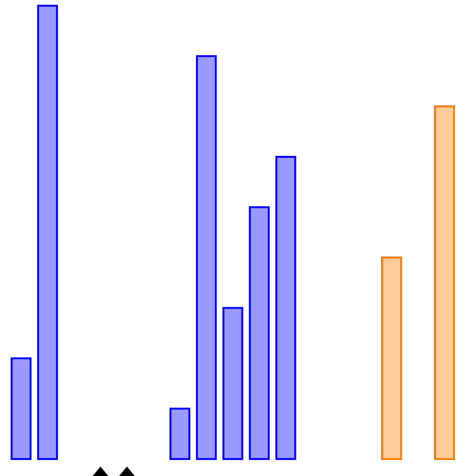
Tiefe 3: Trivialfall

Merge Sort



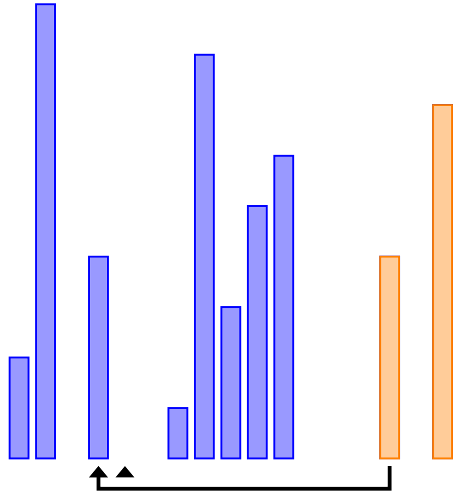
Tiefe 3: Trivialfall

Merge Sort

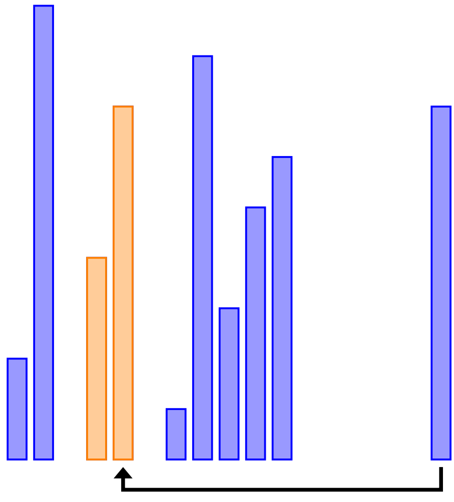


Tiefe 2: Conquer/Merge

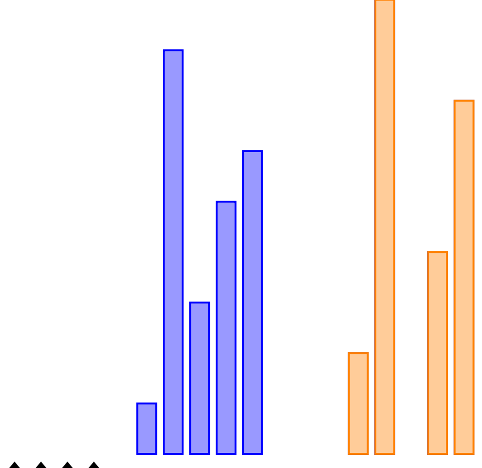
Merge Sort



Merge Sort



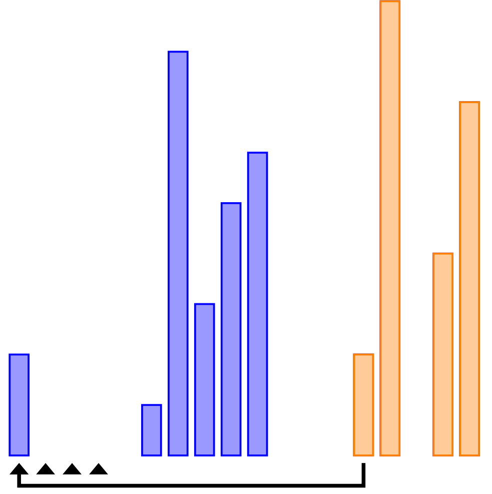
Merge Sort



Call-Stack

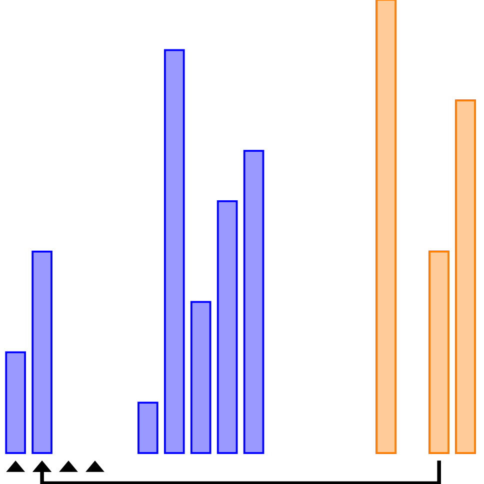
Tiefe 1: Conquer/Merge

Merge Sort

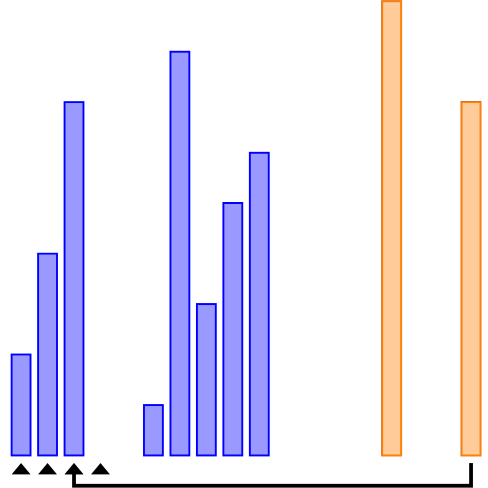


Call-Stack

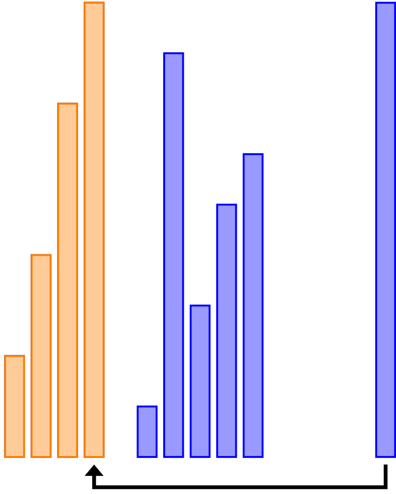
Merge Sort



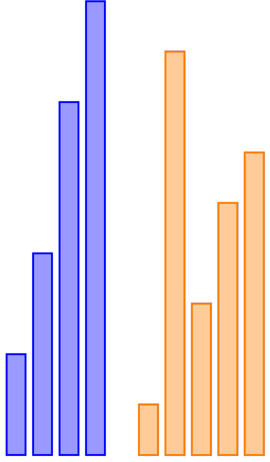
Merge Sort



Merge Sort



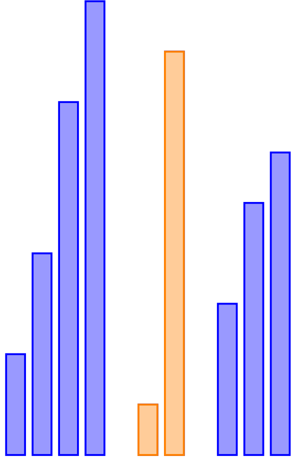
Merge Sort



Tiefe 1: Divide

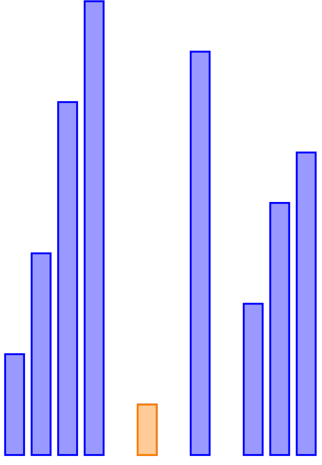
Call-Stack

Merge Sort



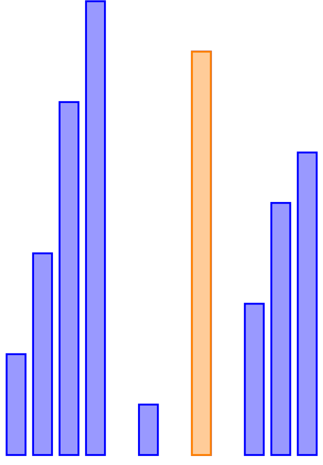
Tiefe 2: Divide

Merge Sort



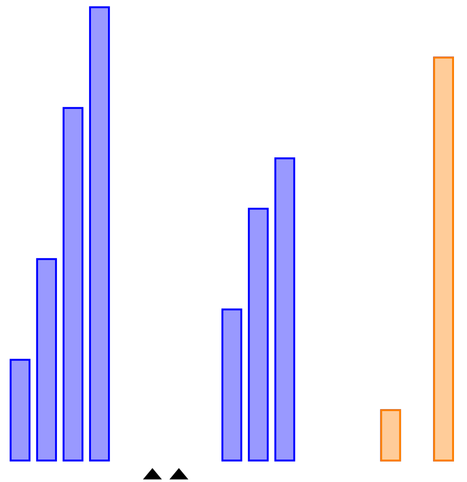
Tiefe 3: Trivialfall

Merge Sort



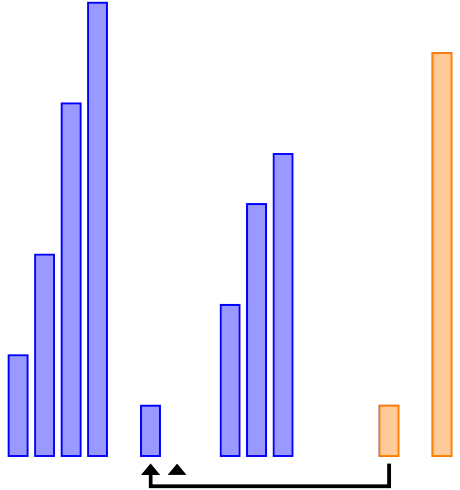
Tiefe 3: Trivialfall

Merge Sort

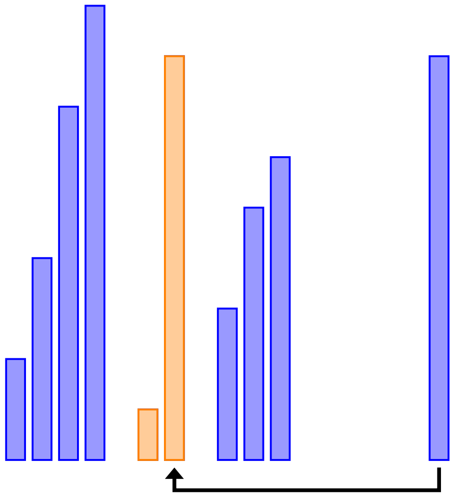


Tiefe 2: Conquer/Merge

Merge Sort

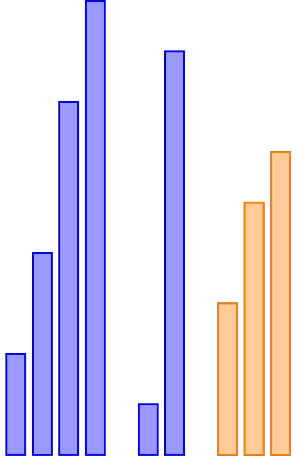


Merge Sort



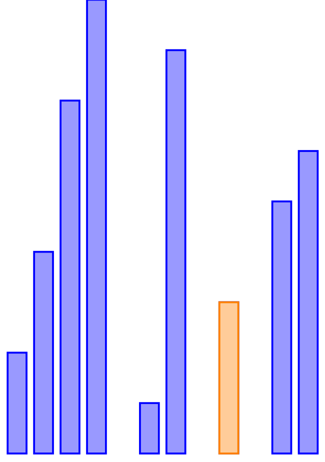
Tiefe 2: Conquer/Merge

Merge Sort



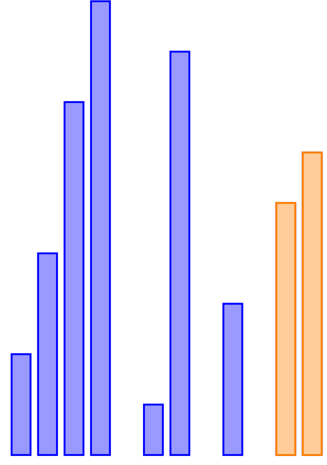
Tiefe 2: Divide

Merge Sort



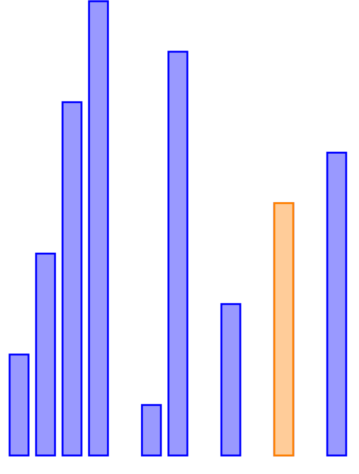
Tiefe 3: Trivialfall

Merge Sort



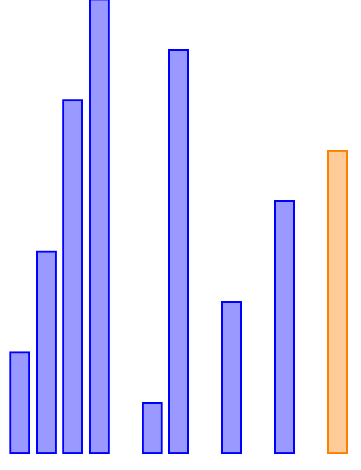
Tiefe 3: Divide

Merge Sort



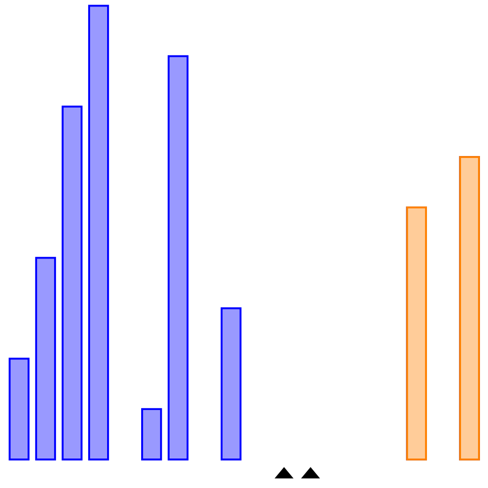
Tiefe 4: Trivialfall

Merge Sort



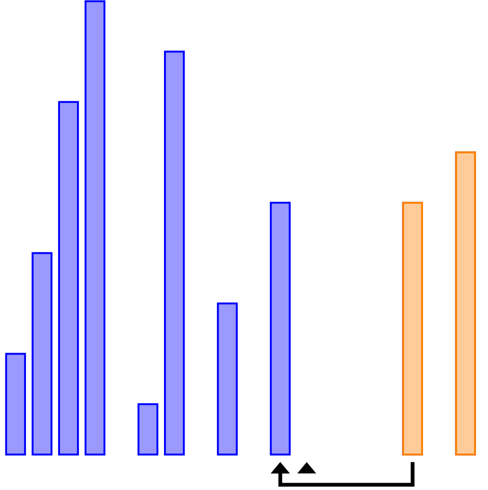
Tiefe 4: Trivialfall

Merge Sort



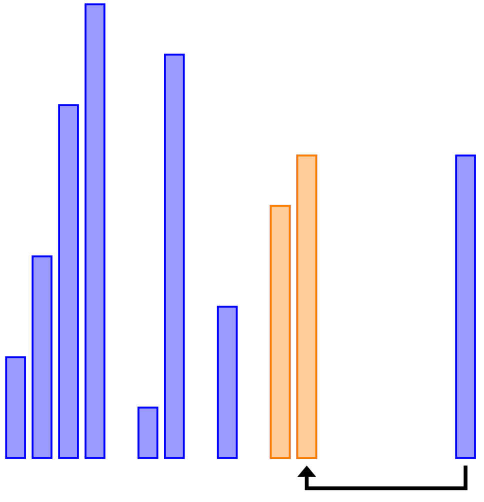
Tiefe 3: Conquer/Merge

Merge Sort



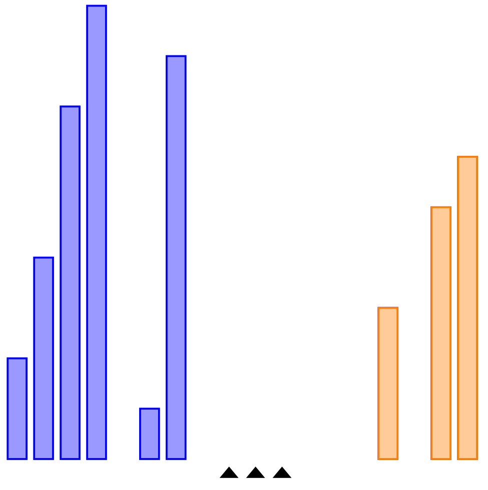
Tiefe 3: Conquer/Merge

Merge Sort



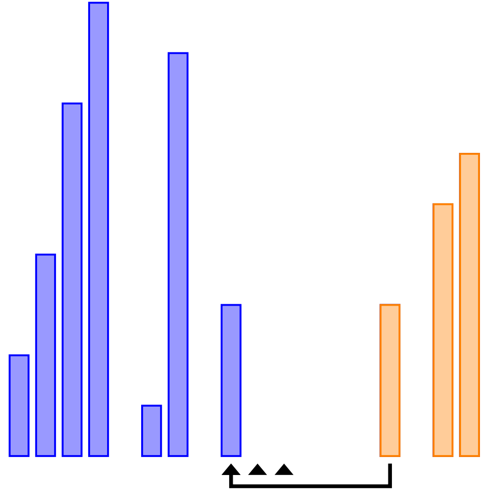
Tiefe 3: Conquer/Merge

Merge Sort

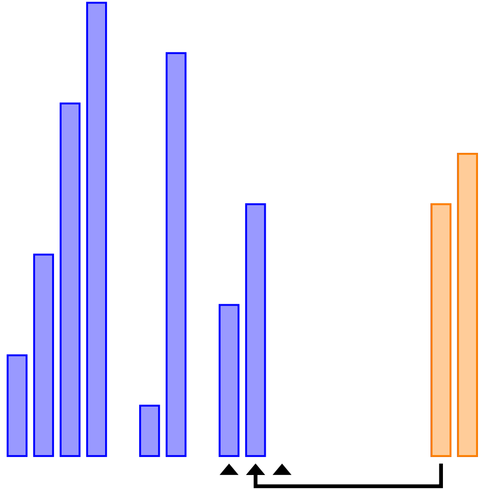


Tiefe 2: Conquer/Merge

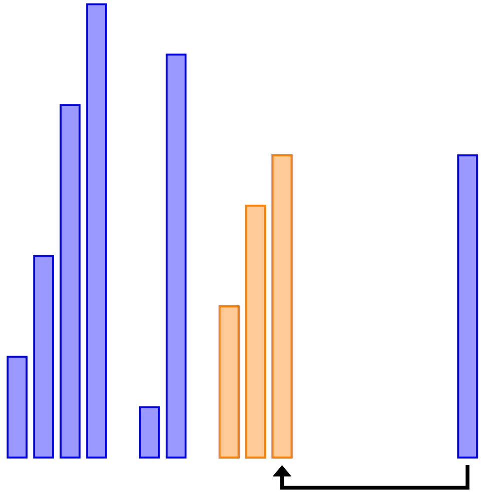
Merge Sort



Merge Sort

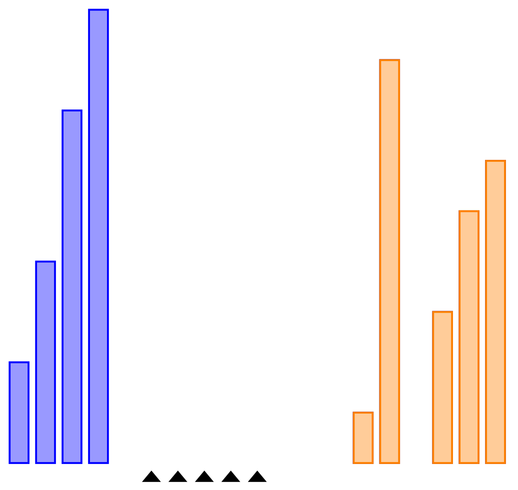


Merge Sort



Tiefe 2: Conquer/Merge

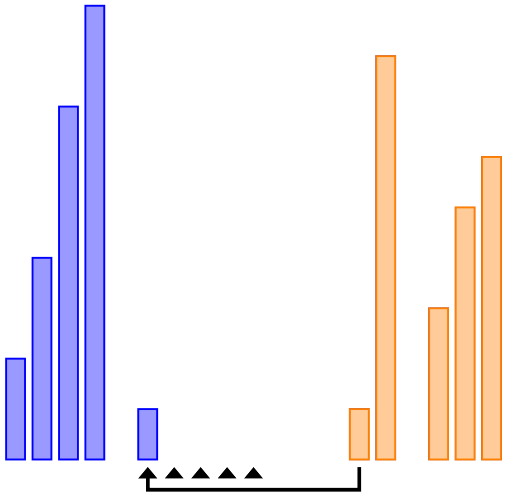
Merge Sort



Tiefe 1: Conquer/Merge

Call-Stack

Merge Sort



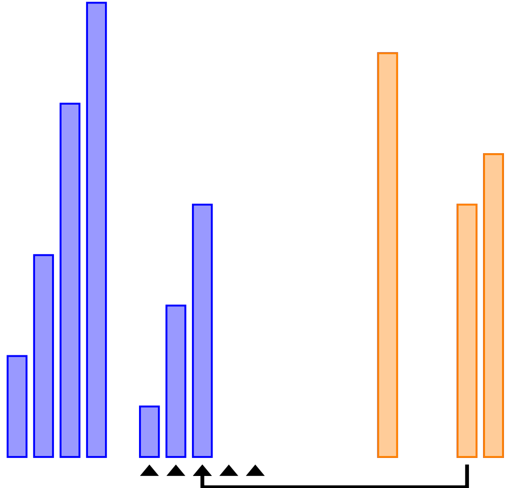
Tiefe 1: Conquer/Merge

Call-Stack

Merge Sort

Tiefe 1: Conquer/Merge

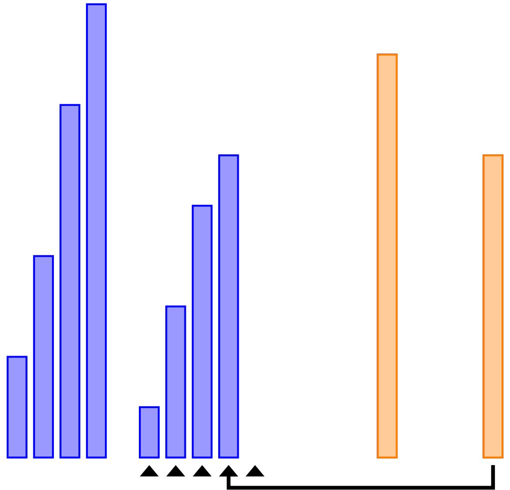
Merge Sort



Tiefe 1: Conquer/Merge

Call-Stack

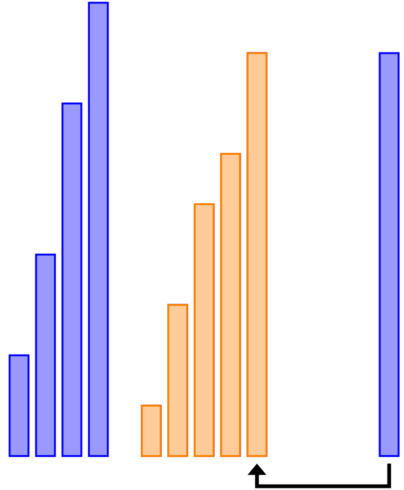
Merge Sort



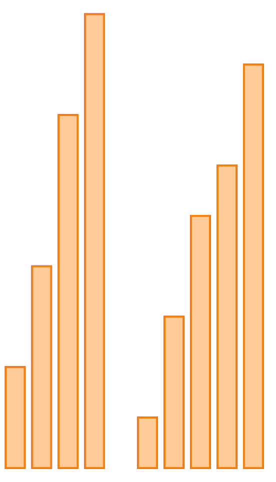
Tiefe 1: Conquer/Merge

Call-Stack

Merge Sort



Merge Sort

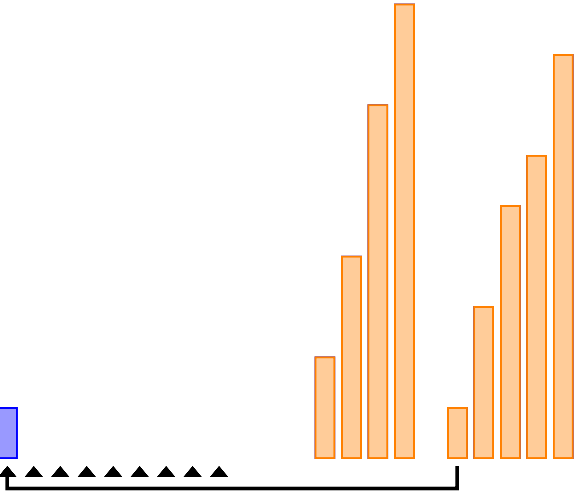


Call-Stack

▲▲▲▲▲▲▲▲▲

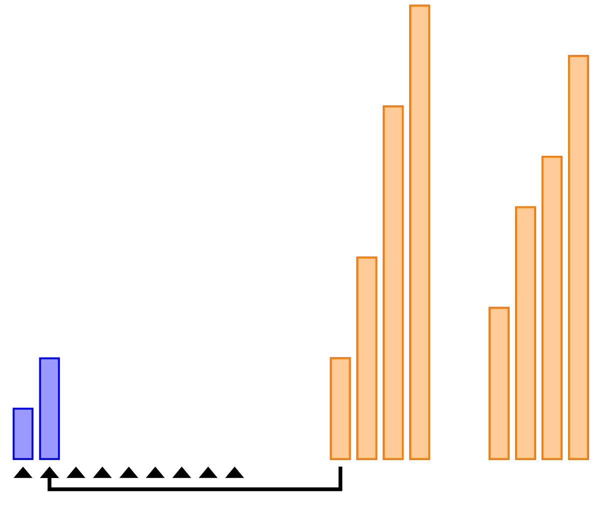
Tiefe 0: Conquer/Merge

Merge Sort



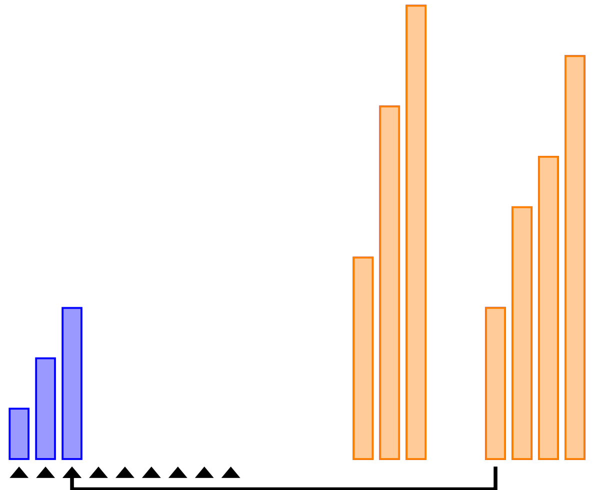
Call-Stack

Merge Sort



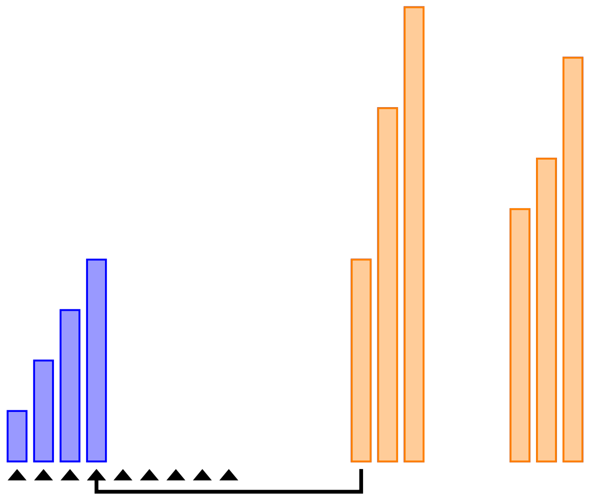
Call-Stack

Merge Sort



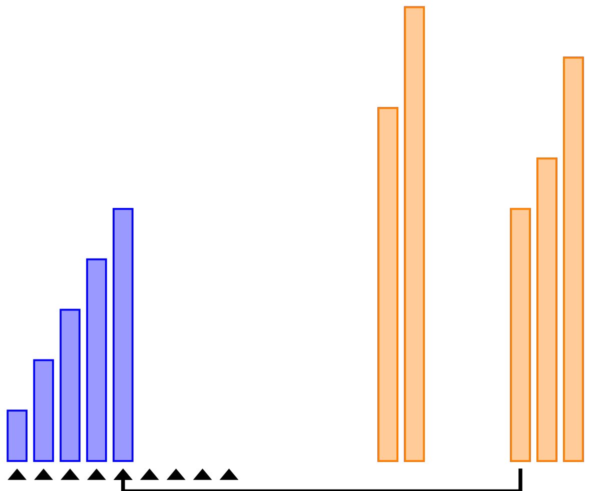
Call-Stack

Merge Sort



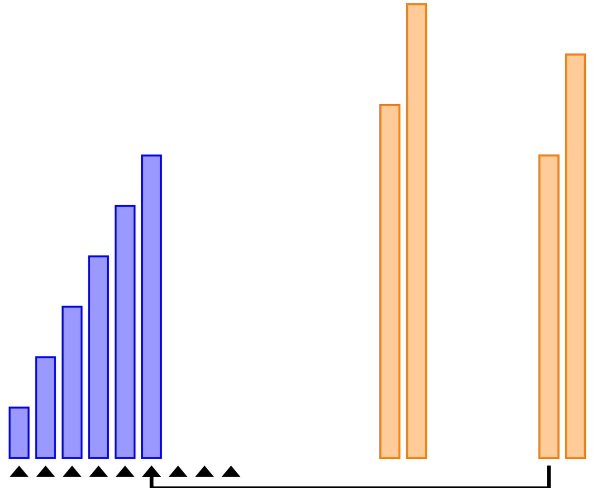
Call-Stack

Merge Sort



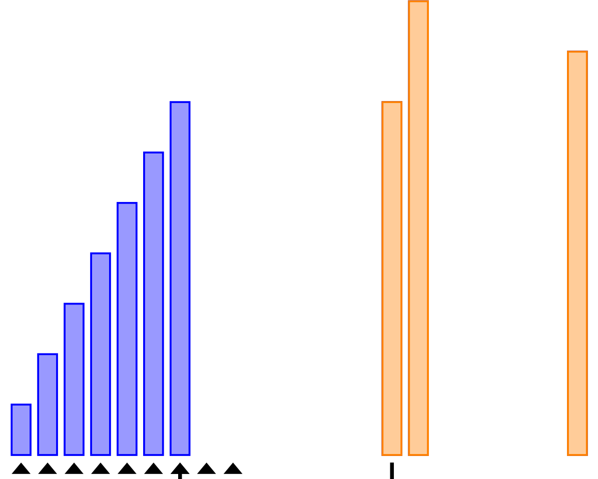
Call-Stack

Merge Sort



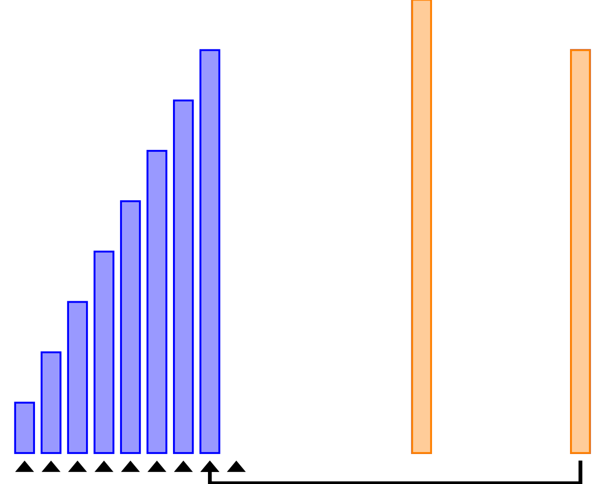
Tiefe 0: Conquer/Merge

Merge Sort



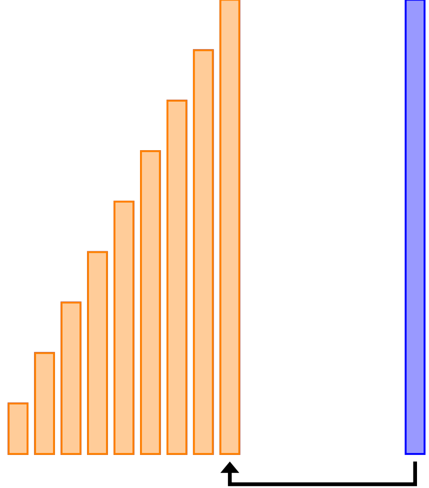
Call-Stack

Merge Sort



Call-Stack

Merge Sort



Call-Stack

☞ Code: Merge Sort

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] ≤ pv) {
            Swap(A, pn, i);
            pn = pn + 1;
        }
    }

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

Quick Sort: Partitionierung

$pv = 5$ $\leq pv$ $> pv$

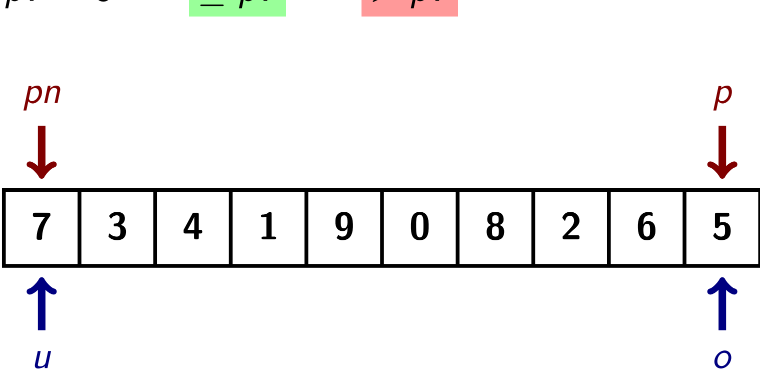
p

7	3	4	1	9	5	8	2	6	0

u o

Anfangszustand

Quick Sort: Partitionierung



Pivot ans Ende

Quick Sort: Partitionierung

$pv = 5$ $\leq pv$ $> pv$

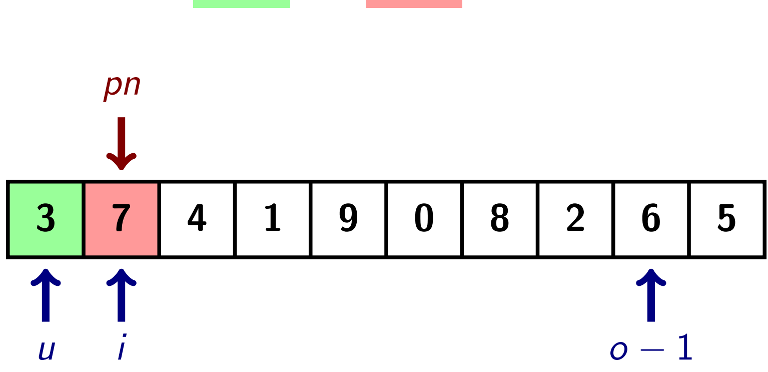
pn

7	3	4	1	9	0	8	2	6	5

i $o - 1$

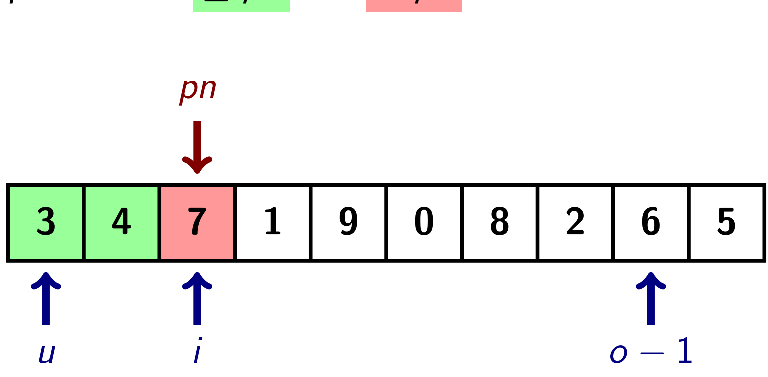
Schleifendurchlauf

Quick Sort: Partitionierung



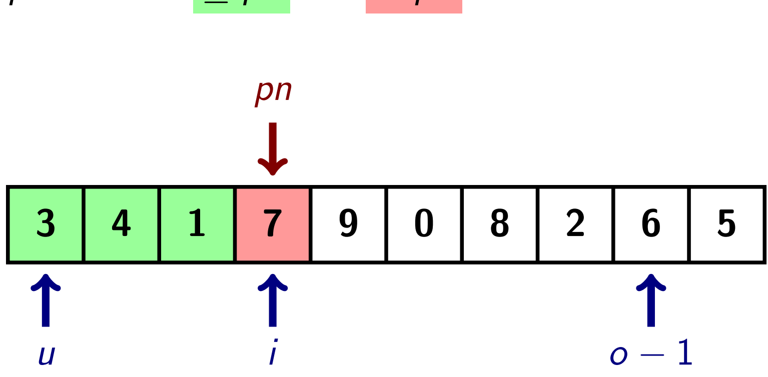
Schleifendurchlauf

Quick Sort: Partitionierung



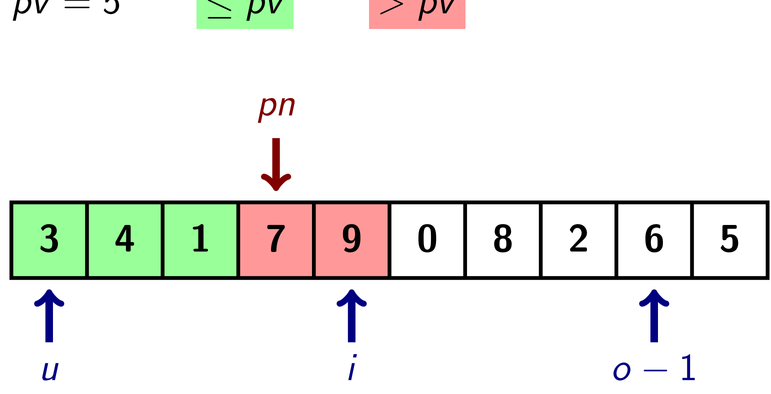
Schleifendurchlauf

Quick Sort: Partitionierung



Schleifendurchlauf

Quick Sort: Partitionierung



Schleifendurchlauf

Quick Sort: Partitionierung

$pv = 5$ $\leq pv$ $> pv$

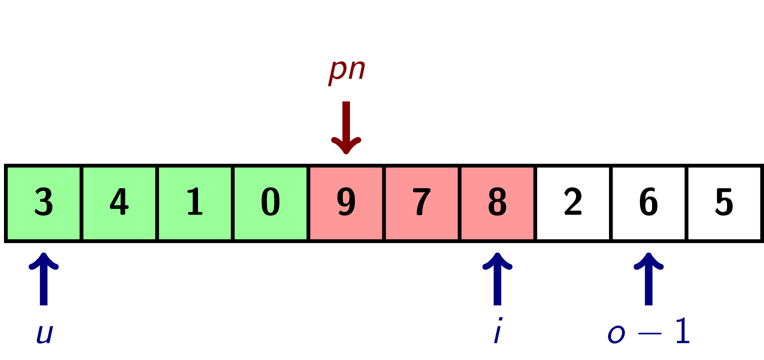
pn

3	4	1	0	9	7	8	2	6	5

u i $o-1$

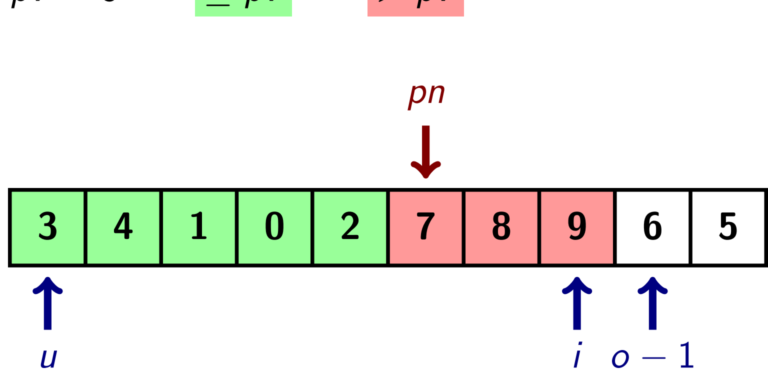
Schleifendurchlauf

Quick Sort: Partitionierung



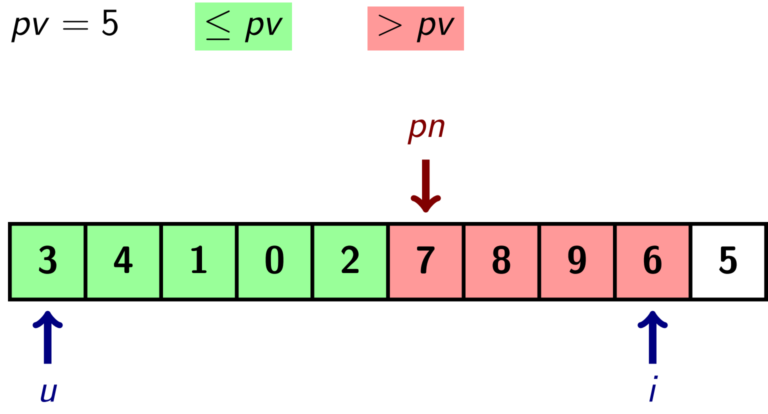
Schleifendurchlauf

Quick Sort: Partitionierung



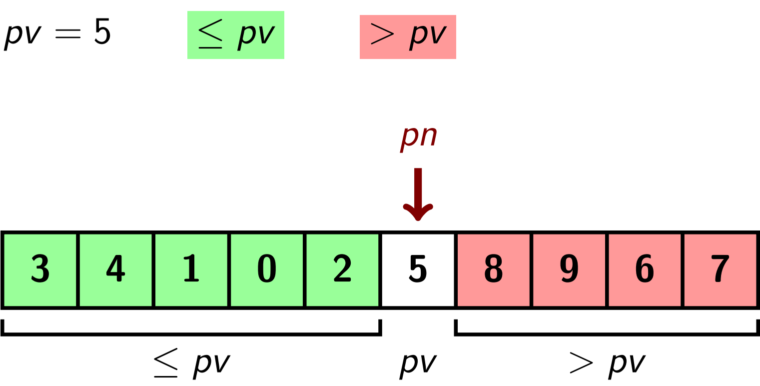
Schleifendurchlauf

Quick Sort: Partitionierung



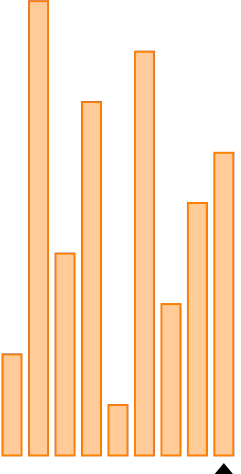
Schleifendurchlauf

Quick Sort: Partitionierung



Pivot zurück, sortiere grünen und roten Teil rekursiv

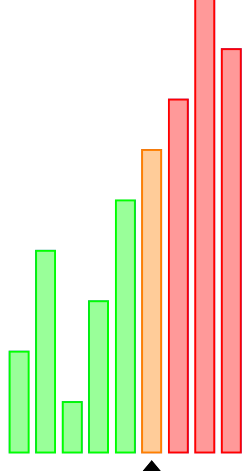
Quick Sort



Tiefe 0: Anfangszustand

Call-Stack

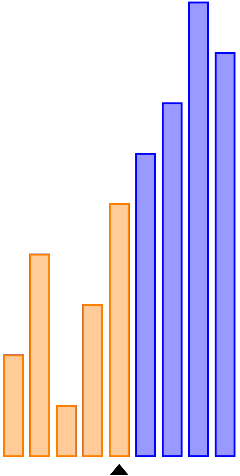
Quick Sort



Tiefe 0: Partitionierung

Call-Stack

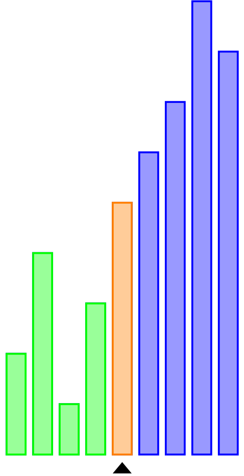
Quick Sort



Tiefe 1: Anfangszustand

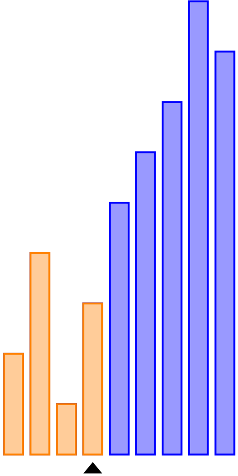
Call-Stack

Quick Sort



Tiefe 1: Partitionierung

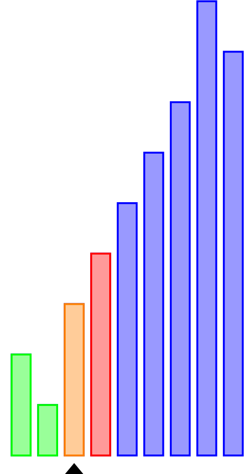
Quick Sort



Tiefe 2: Anfangszustand

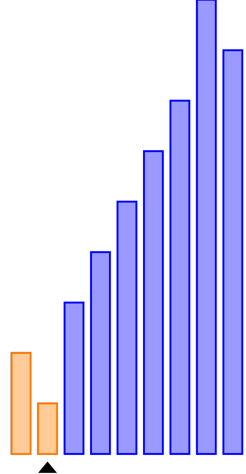
Call-Stack

Quick Sort



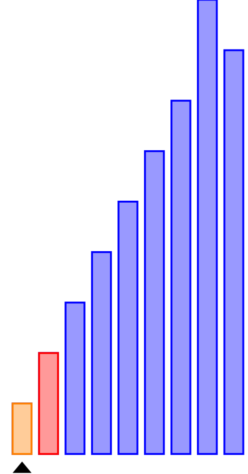
Tiefe 2: Partitionierung

Quick Sort



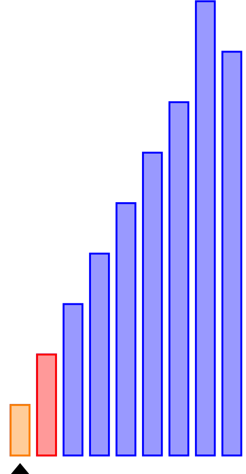
Tiefe 3: Anfangszustand

Quick Sort



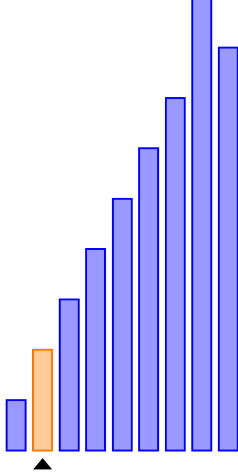
Tiefe 3: Partitionierung

Quick Sort



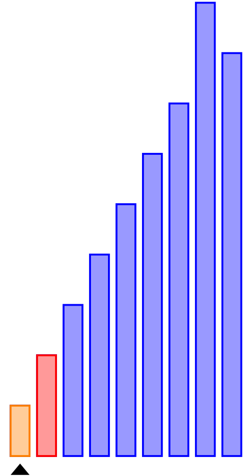
Tiefe 3: Sortierung des unteren Teils beendet

Quick Sort



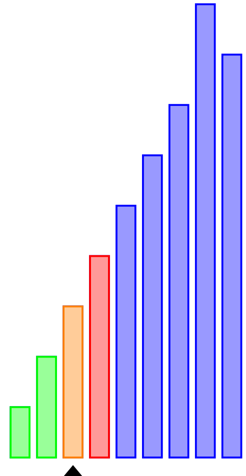
Tiefe 4: Trivialfall

Quick Sort



Tiefe 3: Sortierung des oberen Teils beendet

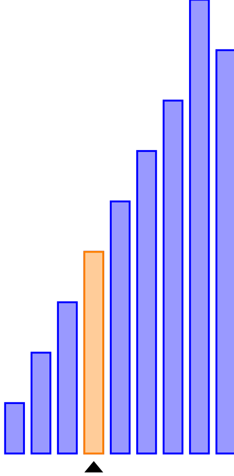
Quick Sort



Call-Stack

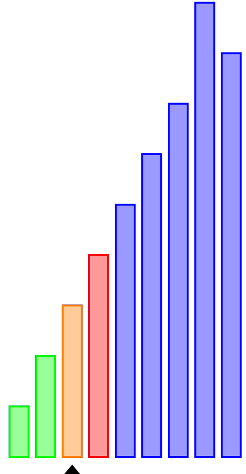
Tiefe 2: Sortierung des unteren Teils beendet

Quick Sort



Tiefe 3: Trivialfall

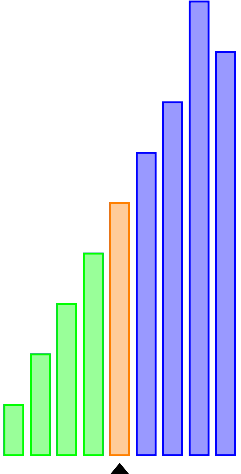
Quick Sort



Call-Stack

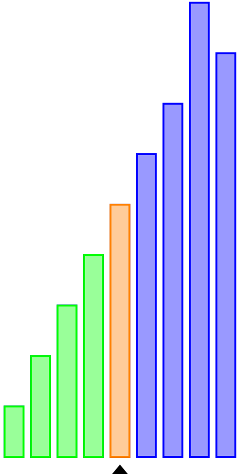
Tiefe 2: Sortierung des oberen Teils beendet

Quick Sort



Tiefe 1: Sortierung des unteren Teils beendet

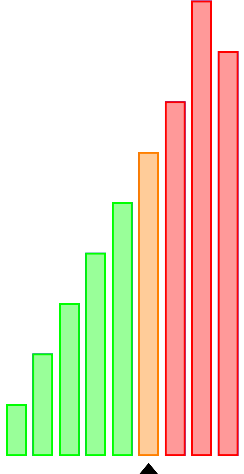
Quick Sort



Call-Stack

Tiefe 1: Sortierung des oberen Teils beendet

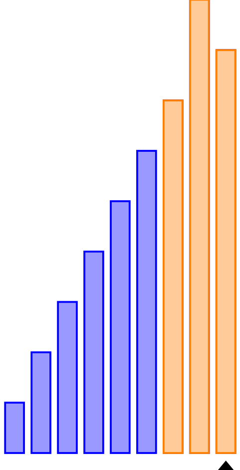
Quick Sort



Call-Stack

Tiefe 0: Sortierung des unteren Teils beendet

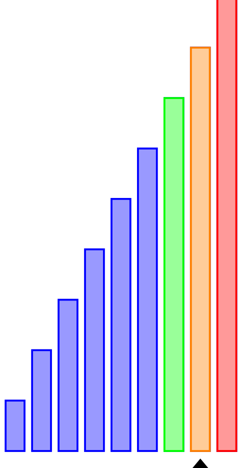
Quick Sort



Tiefe 1: Anfangszustand

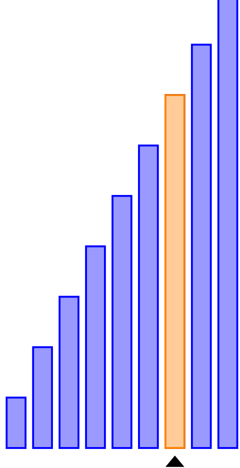
Call-Stack

Quick Sort



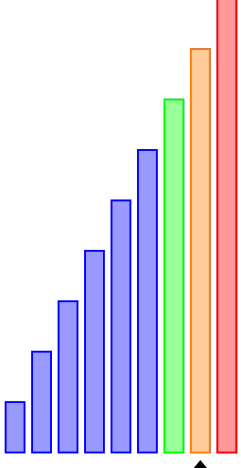
Tiefe 1: Partitionierung

Quick Sort



Tiefe 2: Trivialfall

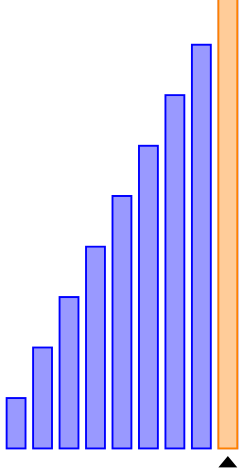
Quick Sort



Call-Stack

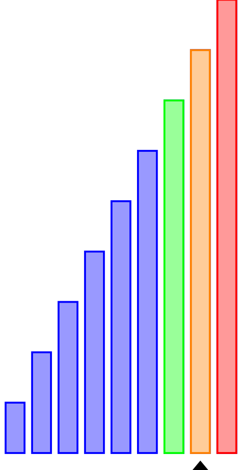
Tiefe 1: Sortierung des unteren Teils beendet

Quick Sort



Tiefe 2: Trivialfall

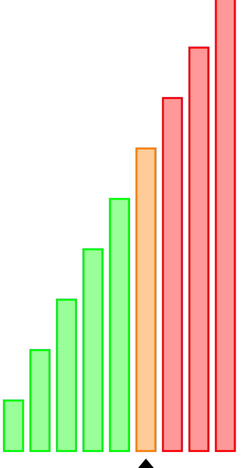
Quick Sort



Call-Stack

Tiefe 1: Sortierung des oberen Teils beendet

Quick Sort



Call-Stack

Tiefe 0: Sortierung des oberen Teils beendet

☞ Code: Quick Sort

Lösungsblatt