ETH zürich

Info2 - ML J 34.1 14-16

WhatsApp group



Woche 2 – Informatik II

Michal Oskedra
Teaching Assistent D-ITET
28. September 2026

Webseite:
moskedra-web.ethz.ch

01 – Wiederholung der Theorie

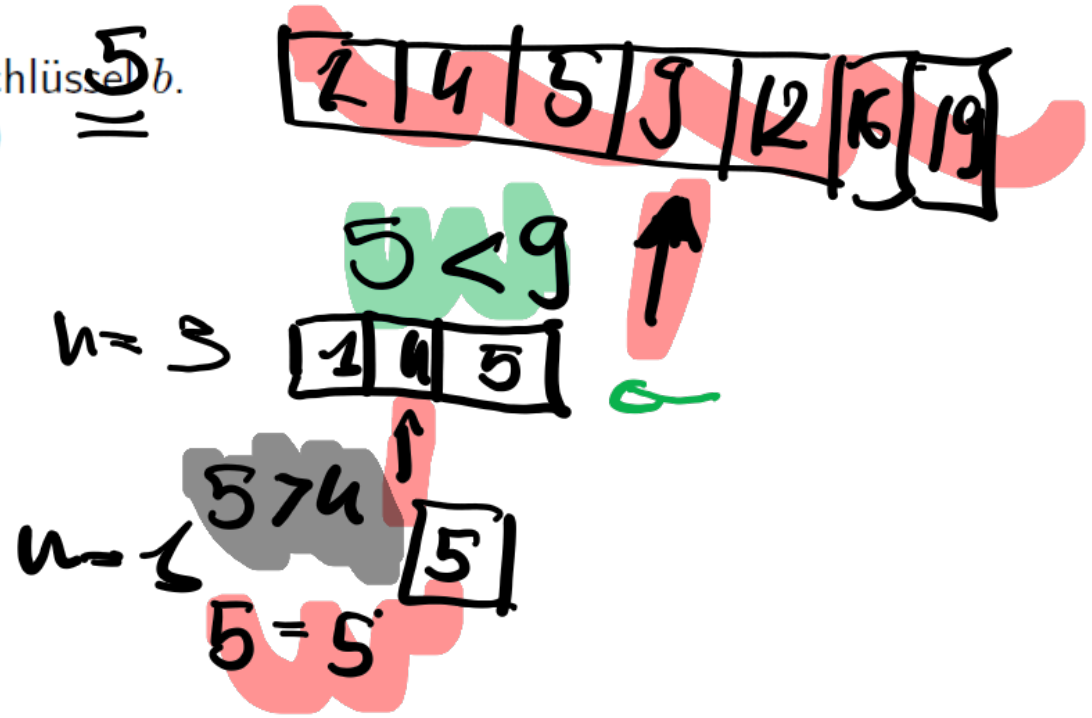
01.1 Rekursive Laufzeiten
01.2 Iterative Suchalgorithmen
01.3 Divide And Counquer
01.4 Stabile und In-situ Algorithmen

BinarySearch Algorithmus

Pseudocode

Input: Aufsteigend sortiertes Array A von n Schlüsseln. Schlüssel b.
Output: Wenn $b \in A$, dann $i \in [1, n]$ mit $A[i] = b$, sonst 0

$m \leftarrow \lfloor (1 + n)/2 \rfloor$
if $n = 0$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then** // gefunden
    return m
else if $b < A[m]$ **then** // Element liegt links
    return BSearch$(A[1, \ldots, m-1], b)$
else // $b > A[m]$: Element liegt rechts
    $k \leftarrow$ BSearch$(A[m+1, \ldots, n], b)$
    if $k = 0$ **then return** 0
    return m+k



$$T(n) = \begin{cases} d & n = 1 \\ T(n/2) + c & \end{cases}$$

$$T(n) = T(n/2) + c = T(n/4) + 2c = T(n/8) + 3c \ \ldots = T\left(\frac{n}{2^k}\right) + k \cdot c$$

$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n = s$

$= d + \log_2 n \cdot c \in \mathcal{O}(\log n)$

BinarySearch Algorithmus

Input: Aufsteigend sortiertes Array A von n Schlüsseln. Schlüssel b.
Output: Wenn $b \in A$, dann $i \in [1, n]$ mit $A[i] = b$, sonst 0

$m \leftarrow \lfloor (1+n)/2 \rfloor$
if $n = 0$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then** // gefunden
    return m
else if $b < A[m]$ **then** // Element liegt links
    return BSearch$(A[1, \ldots, m-1], b)$
else // $b > A[m]$: Element liegt rechts
    $k \leftarrow$ BSearch$(A[m+1, \ldots, n], b)$
    if $k = 0$ **then return** 0
    return m+k

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

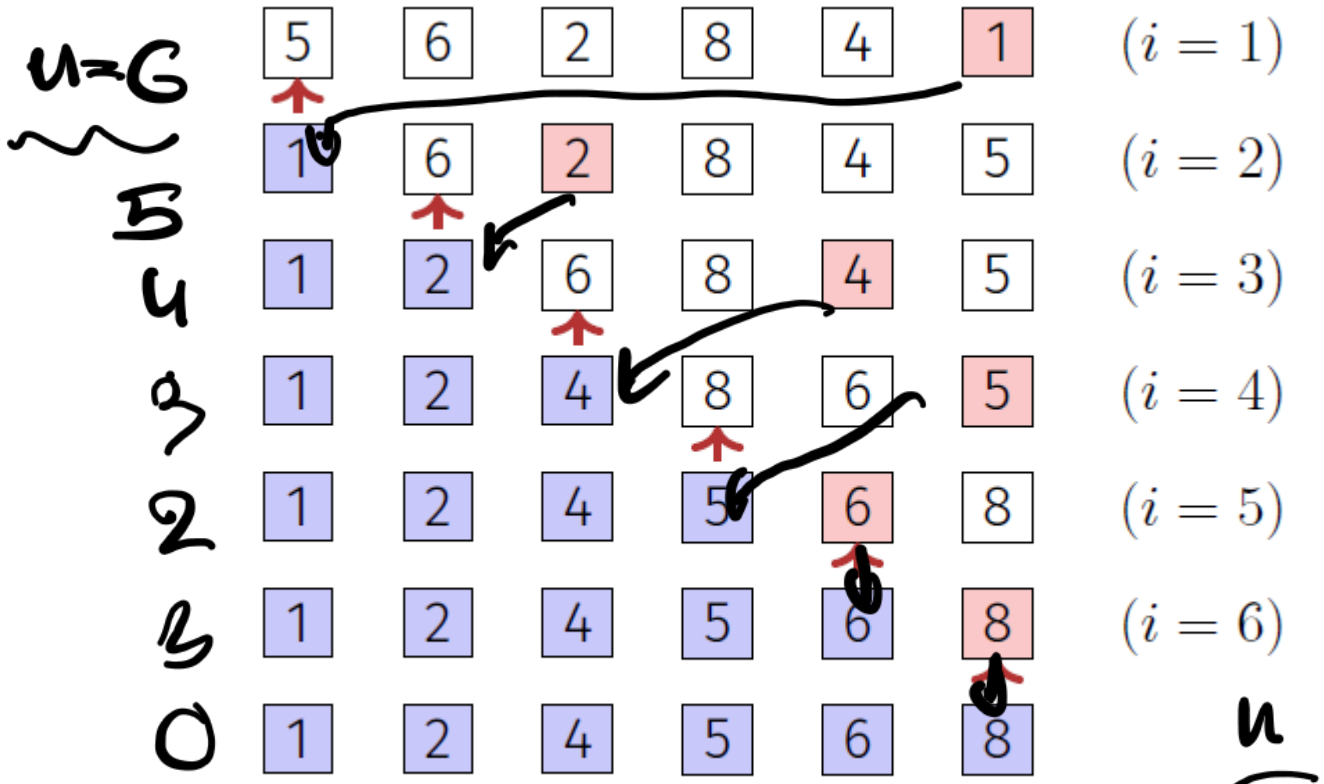
Teleskopieren: [3]

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = \ldots \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c = d + c \cdot \log_2 n \in \Theta(\log n) \end{aligned}$$

01 – Wiederholung der Theorie

01.1 Rekursive Laufzeiten
01.2 Iterative Suchalgorithmen
01.3 Divide And Counquer
01.4 Stabile und In-situ Algorithmen

SelectionSort



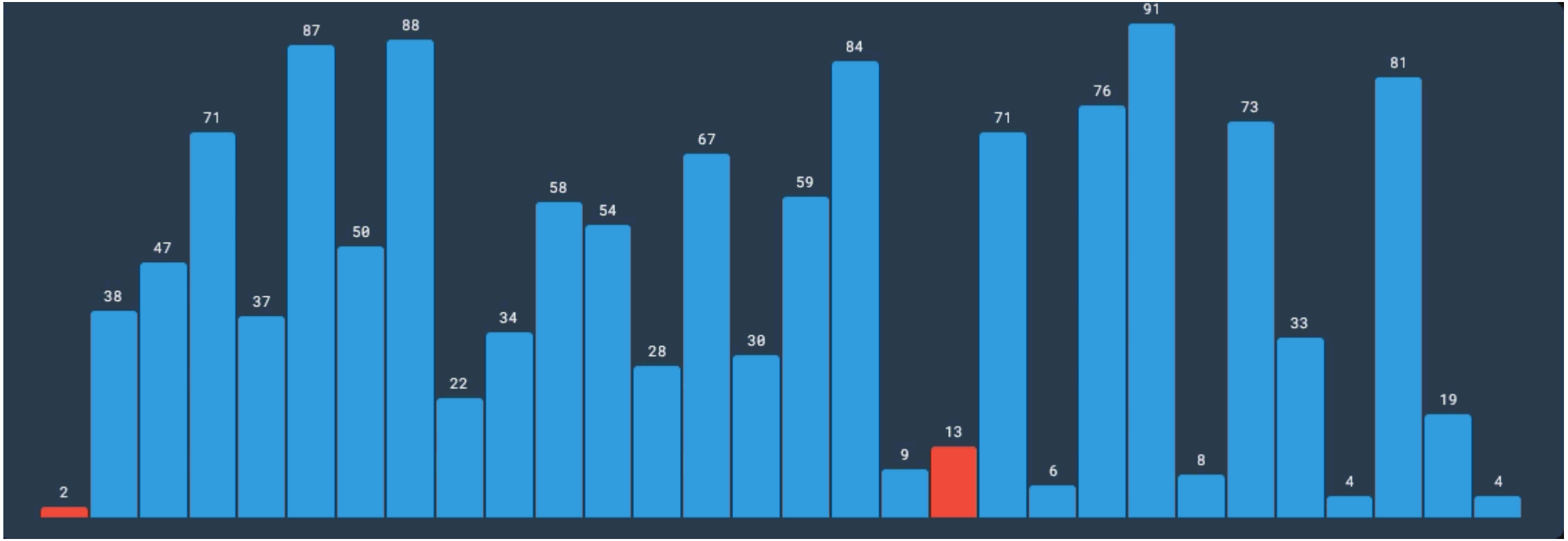
Input: Array $A = (A[1], \ldots, A[n])$.
Output: Sortiertes Array A

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

$$\sum_{k=0}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

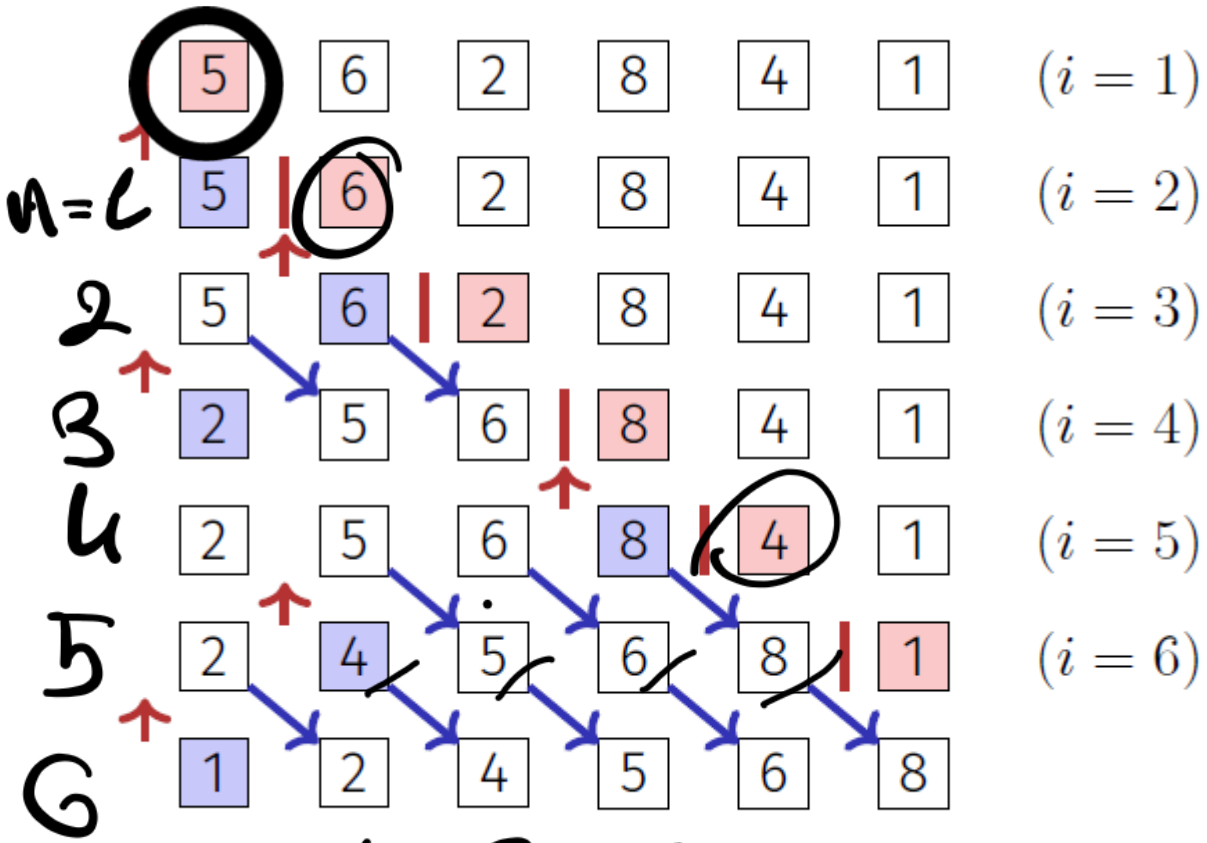
1, 2, 3, 4, 5, ... n

SelectionSort



https://www.publicsofttools.com/tools/sorting-algorithm-visualizer

InsertionSort



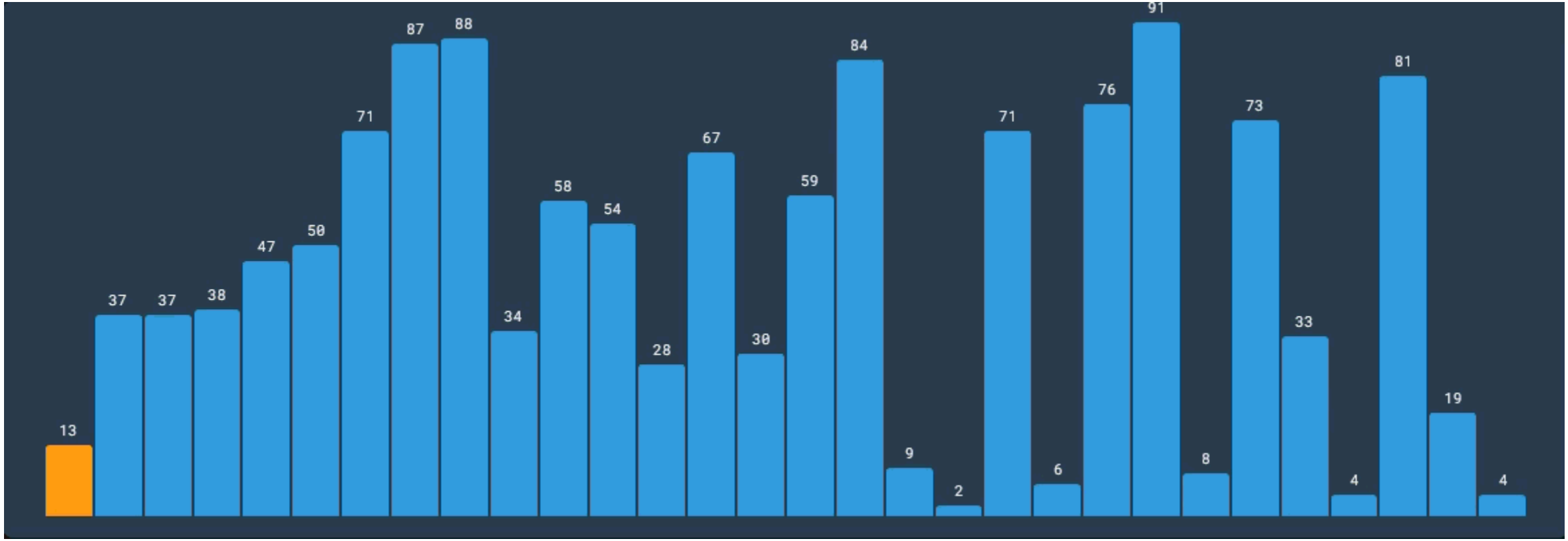
laufzeit $\log n$, aber trotzdem bis zu n shifts notwendig

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

```
for i ← 2 to n do
    x ← A[i]
    p ← BinaryInsSearch(A[1..i − 1], x); // E
    for j ← i downto p do
        A[j] ← A[j − 1]
    A[p] ← x
```

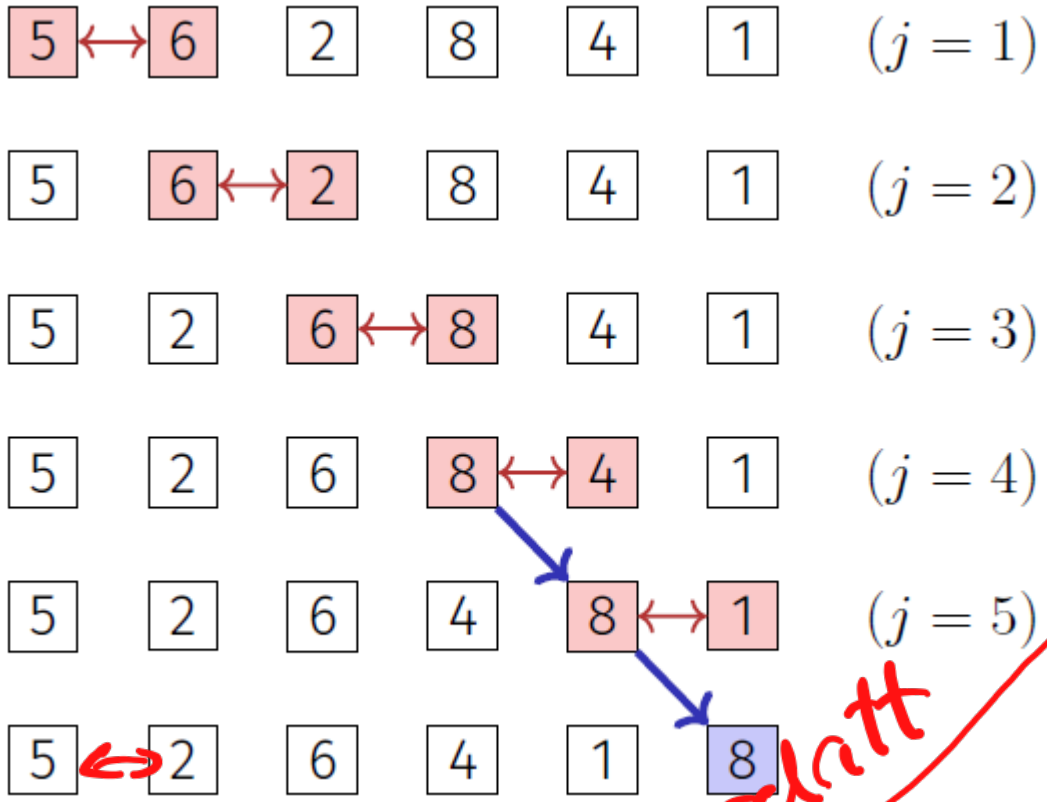
$1 + 2 + 3 + \ldots + n = O(n^2)$

InsertionSort

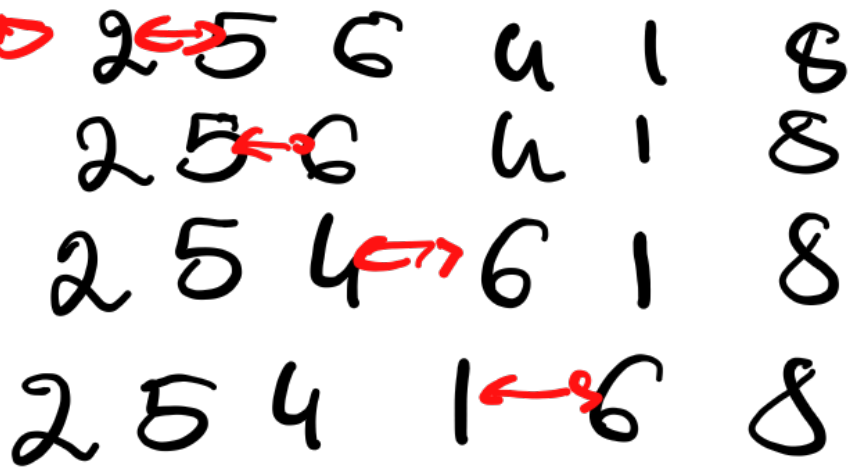


https://www.publicsofttools.com/tools/sorting-algorithm-visualizer

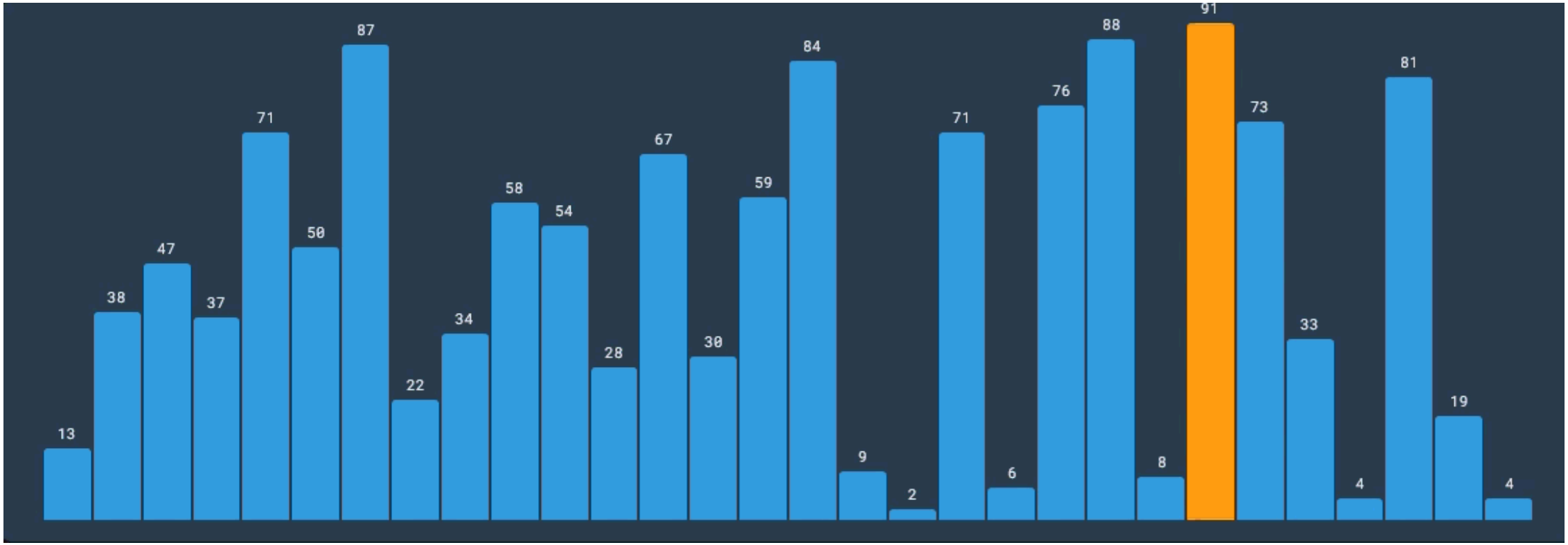
BubbleSort




```
fori ← 1 to length [A] do
for j ← length [A] down-to i +1 do
   if A[A] < A[j-1] then
      Exchange A[j] ⟷ A[j-1]
```



BubbleSort



https://www.publicsofttools.com/tools/sorting-algorithm-visualizer

Vergleich der Algorithmen

Algorithmus	Swaps/Shifs (best / worst case)
Bubble Sort	$0/\Theta(n^2)$
Selection Sort	$0/\Theta(n)$
Insert Sort	$0/\Theta(n^2)$

Algorithmus	Comparisions (best / worst case)
Bubble Sort	$\Theta(n^2)/\Theta(n^2)$
Selection Sort	$\Theta(n^2)/\Theta(n^2)$
Insert Sort	$\Theta(n)/\Theta(n^2)$

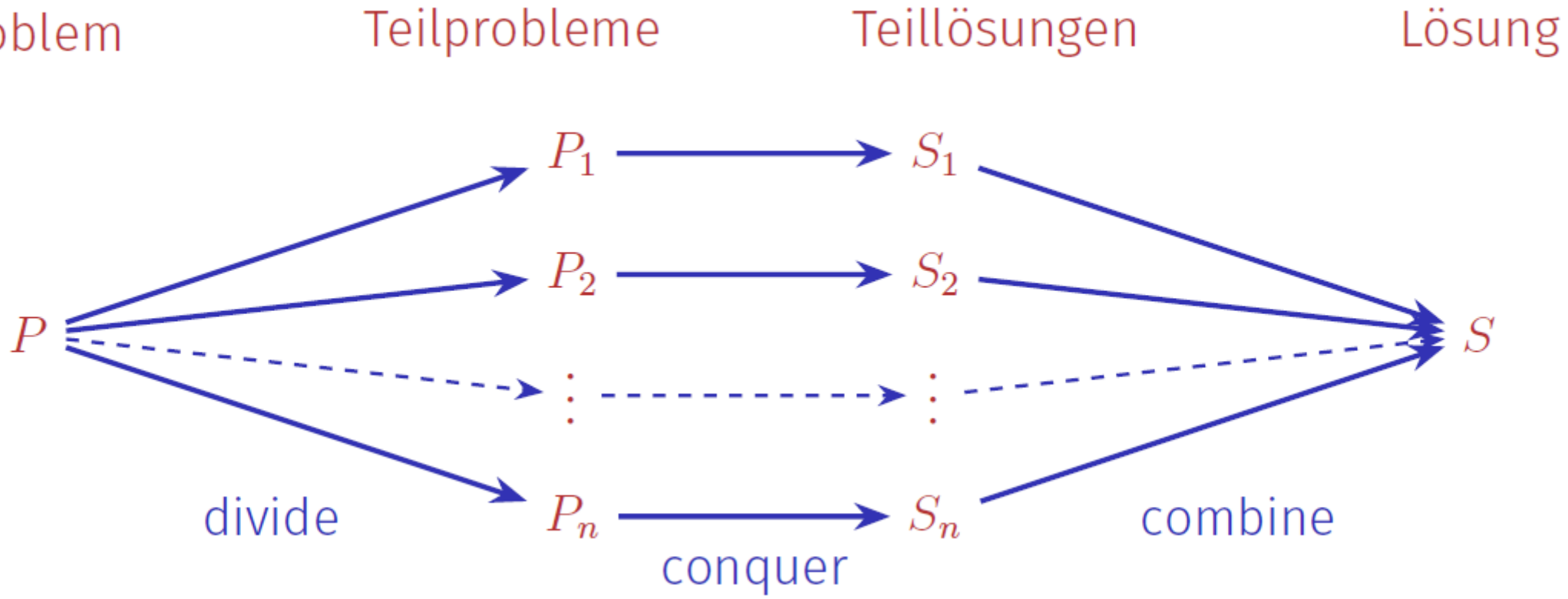
https://www.toptal.com/developers/sorting-algorithms



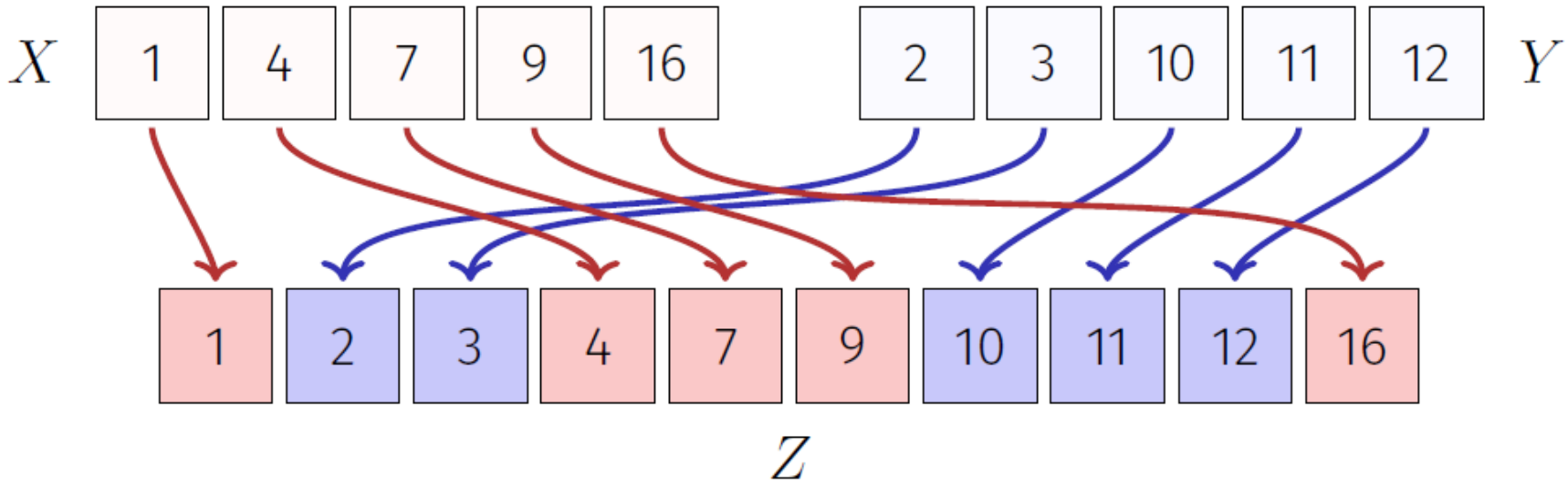
01 – Wiederholung der Theorie

01.1 Rekursive Laufzeiten
01.2 Iterative Suchalgorithmen
01.3 Divide And Counquer
01.4 Stabile und In-situ Algorithmen

Kernidee “Divide and Conquer”

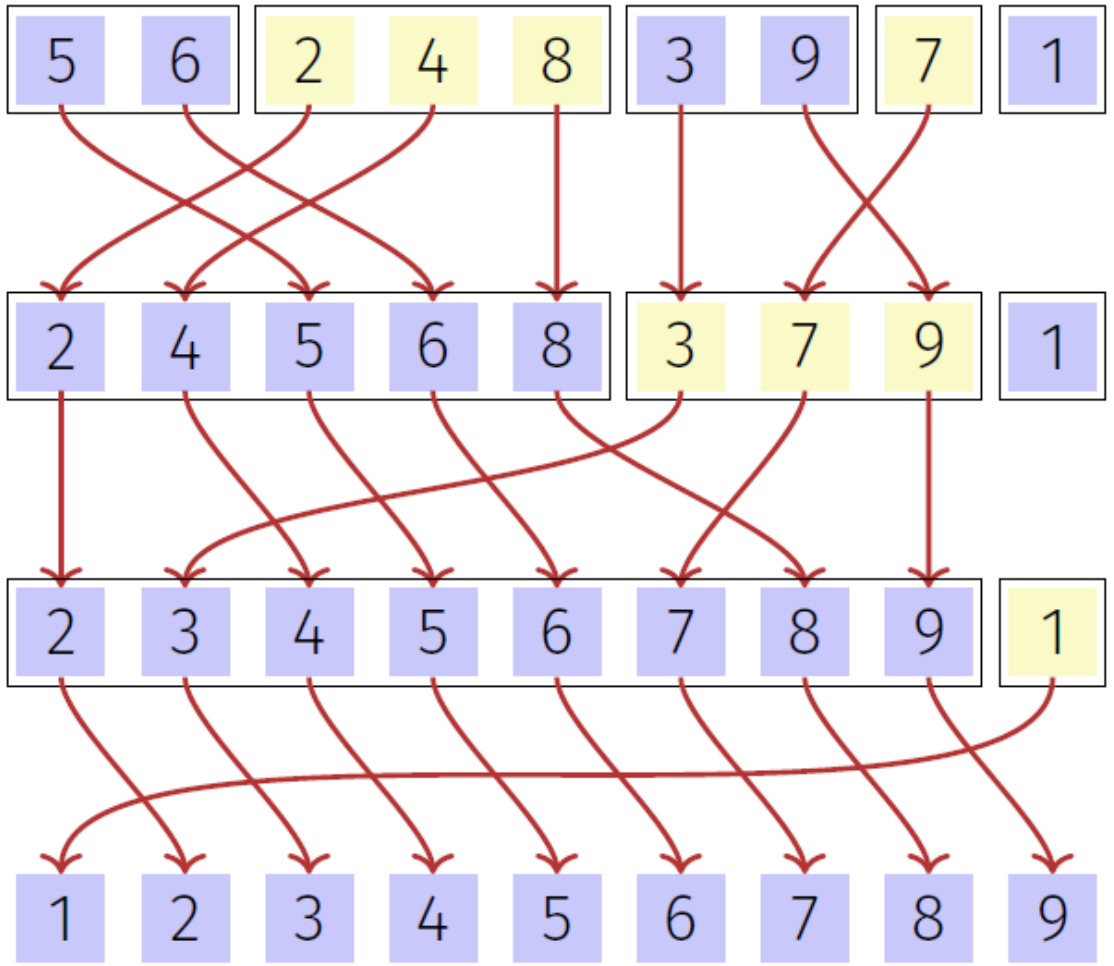


MergeSort

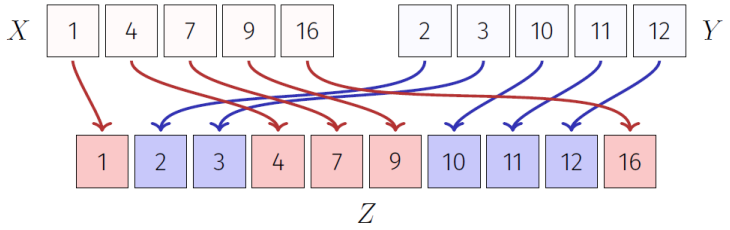


Zwei sortierte Arrays X und Y können mit $\Theta(n)$ Operationen in eine Array Z zusammengefügt werden

MergeSort Beispiel



MergeSort



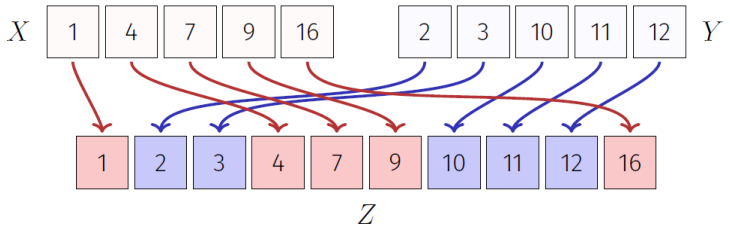
MergeSort(A)

Input: Array A der Länge n.
Output: A sortiert.
if $n > 1$ **then**
    $m \leftarrow \lfloor n/2 \rfloor$
    MergeSort($A[1..m]$)
    MergeSort($A[m+1..n]$)
    $A[1..n] \leftarrow$ Merge($A[1..m]$, $A[m+1..n]$)

Algorithmus Merge(X, Y)

Input: Arrays X der Länge n und Y der Länge l sortiert
Output: Array Z der Länge $n+l$ sortiert mit allen Elementen von $X \cup Y$
$Z \leftarrow$ new Array$(n+l)$
$x \leftarrow 1$; $y \leftarrow 1$; $z \leftarrow 1$
while $x \leq n$ and $y \leq l$ **do**
    if $X[x] \leq Y[y]$ **then** $Z[z] \leftarrow X[x]$; $x \leftarrow x+1$
    else $Z[z] \leftarrow Y[y]$; $y \leftarrow y+1$
    $z \leftarrow z+1$;
while $x \leq n$ **do** $Z[z] \leftarrow X[x]$; $x \leftarrow x+1$; $z \leftarrow z+1$
while $y \leq l$ **do** $Z[z] \leftarrow Y[y]$; $y \leftarrow y+1$; $z \leftarrow z+1$
return Z

MergeSort Laufzeit

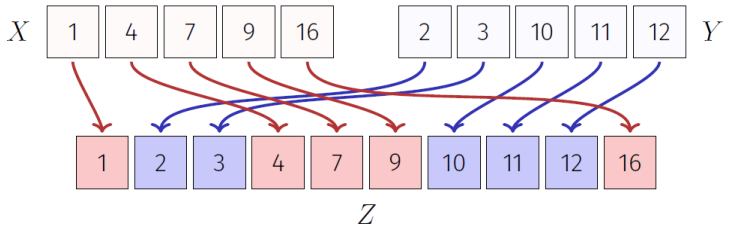



```
MergeSort(A)

Input:  Array A der Länge n.
Output: A sortiert.
if n > 1 then
    m ← ⌊n/2⌋
    MergeSort(A[1..m])
    MergeSort(A[m + 1..n])
    A[1..n] ← Merge(A[1..m], A[m + 1..n])
```

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ 2T(n/2) + cn & \text{falls } n > 1 \end{cases}$$

MergeSort Laufzeit



MergeSort(A)

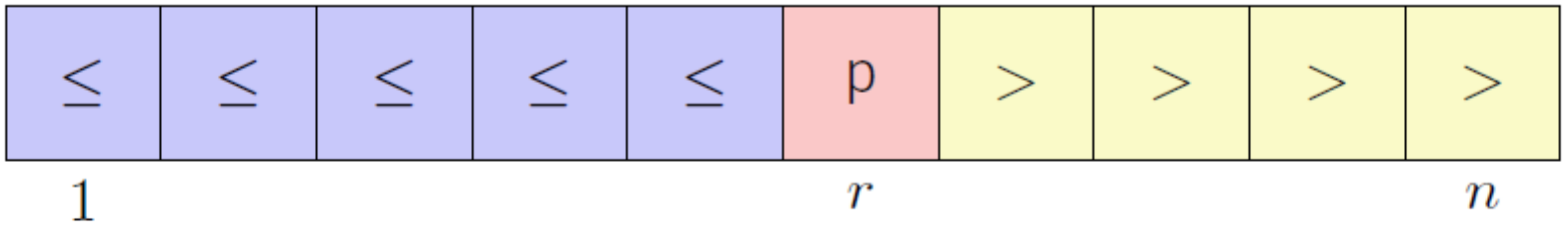
Input: Array A der Länge n.
Output: A sortiert.
if $n > 1$ **then**
    $m \leftarrow \lfloor n/2 \rfloor$
    MergeSort($A[1..m]$)
    MergeSort($A[m+1..n]$)
    $A[1..n] \leftarrow$ Merge($A[1..m]$, $A[m+1..n]$)

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ 2T(n/2) + cn & \text{falls } n > 1 \end{cases}$$

Teleskopieren

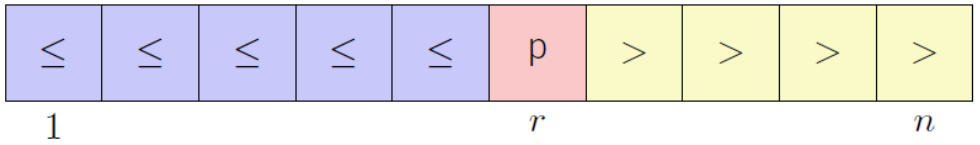
$$\begin{aligned} T(n) &= 2T(n/2) + cn = 2(2T(n/4) + cn/2) + cn \\ &= 2(2(2T(n/8) + cn/4) + cn/2) + cn = \ldots \\ &= 2(2(\ldots(2(2T(n/2^k) + cn/2^{k-1})\ldots) + cn/2^2) + cn/2^1) + cn \\ &= 2^k T(1) + \underbrace{2^{k-1}cn/2^{k-1} + 2^{k-2}cn/2^{k-2} + \ldots + 2^{k-k}cn/2^{k-k}}_{k \text{ Terme}} \\ &= nd + cnk = nd + cn\log_2 n \in \Theta(n \log n). \end{aligned}$$

QuickSort



1. Wähle ein (beliebiges) Element p als Pivotelement
2. Teile A in zwei Teile auf: einen Teil L der Elemente mit $A[i] \leq p$ und einen Teil R der Elemente mit $A[i] > p$.
3. Quicksort: Rekursion auf Teilen L und R

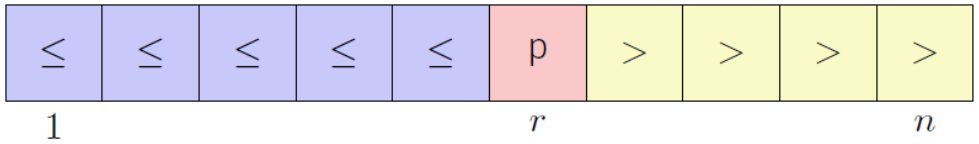
QuickSort Beispiel



2	4	5	6	8	3	7	9	1
2	1	3	6	8	5	7	9	4
1	2	3	6	8	5	7	9	4
1	2	3	6	8	5	7	9	4
1	2	3	4	5	8	7	9	6
1	2	3	4	5	8	7	9	6
1	2	3	4	5	6	7	9	8
1	2	3	4	5	6	7	9	8
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Hier: Das Pivot zufällig

QuickSort

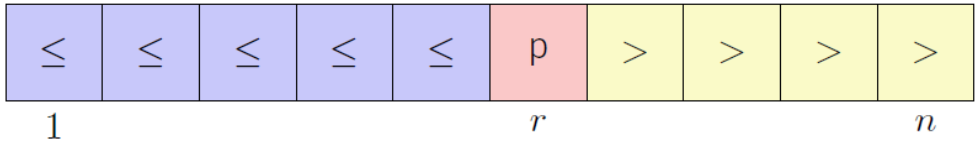


Input: Array A der Länge n.
Output: Array A, sortiert.
if $n > 1$ **then**
    Wähle Pivot $p \in A$
    $k \leftarrow \texttt{Partition}(A, p)$
    $\texttt{QuickSort}(A[1..k-1])$
    $\texttt{QuickSort}(A[k+1..n])$

Algorithmus $\texttt{Partition}(A, p)$

Input: Array A, das den Pivot p mindestend einmal enthält.
Output: Array A partitioniert um Index $k \in [1, n]$, so dass $A_i \leq p \; \forall \; i \in [1, k)$ und $A_i \geq p \; \forall \; i \in (k, n]$ und $A[k] = p$, Rückgabe von k.
$l \leftarrow 1$; $r \leftarrow n$; $\hat{l} \leftarrow \emptyset$; $\hat{r} \leftarrow \emptyset$
loop
    while $A[l] < p$ **do** $l \leftarrow l + 1$
    while $A[r] > p$ **do** $r \leftarrow r - 1$
    if $l \geq r$ **then**
        if $\hat{l} \neq \emptyset$ **then** swap$(A[\hat{l}], A[r])$; **return** r;
        if $\hat{r} \neq \emptyset$ **then** swap$(A[\hat{r}], A[l])$; **return** l;
        return r;
    swap$(A[l], A[r])$
    if $A[l] = p$ **then** $\hat{l} \leftarrow l$
    if $A[r] = p$ **then** $\hat{r} \leftarrow r$
    $l \leftarrow l + 1$; $r \leftarrow r - 1$

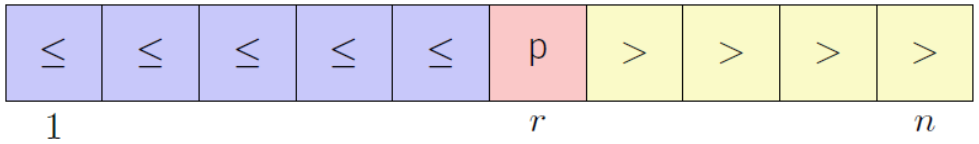
QuickSort



Input: Array A der Länge n.
Output: Array A, sortiert.
if $n > 1$ **then**
    Wähle Pivot $p \in A$
    $k \leftarrow \texttt{Partition}(A, p)$
    $\texttt{QuickSort}(A[1..k-1])$
    $\texttt{QuickSort}(A[k+1..n])$

$$T(n) = \begin{cases} T(n-1) + c \cdot n, & n \geq 1 \\ d, & n = 1 \end{cases} \quad \in \Theta(n^2)$$

QuickSort



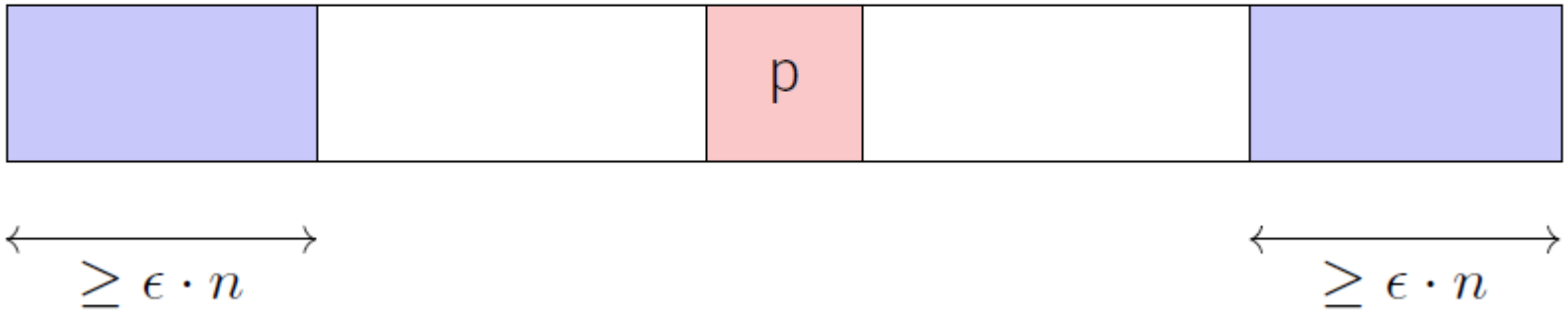
Input: Array A der Länge n.
Output: Array A, sortiert.
if $n > 1$ **then**
    Wähle Pivot $p \in A$
    $k \leftarrow \texttt{Partition}(A, p)$
    $\texttt{QuickSort}(A[1..k-1])$
    $\texttt{QuickSort}(A[k+1..n])$

$$T(n) = \begin{cases} T(n-1) + c \cdot n, & n \geq 1 \\ d, & n = 1 \end{cases} \quad \in \Theta(n^2)$$

$$\begin{aligned} T(n) &= T(n-1) + c \cdot n \\ &= T(n-2) + c \cdot (n-1) + c \cdot n \\ &= T(n-3) + c \cdot (n-2) + c \cdot (n-1) + c \cdot n \\ &= T(1) + c \cdot 2 + c \cdot 3 + \ldots + c \cdot (n-1) + c \cdot n \\ &= d + c \sum_{i=2}^{n} i \in \Theta(n^2) \end{aligned}$$

QuickSort: Wahl des Pivots

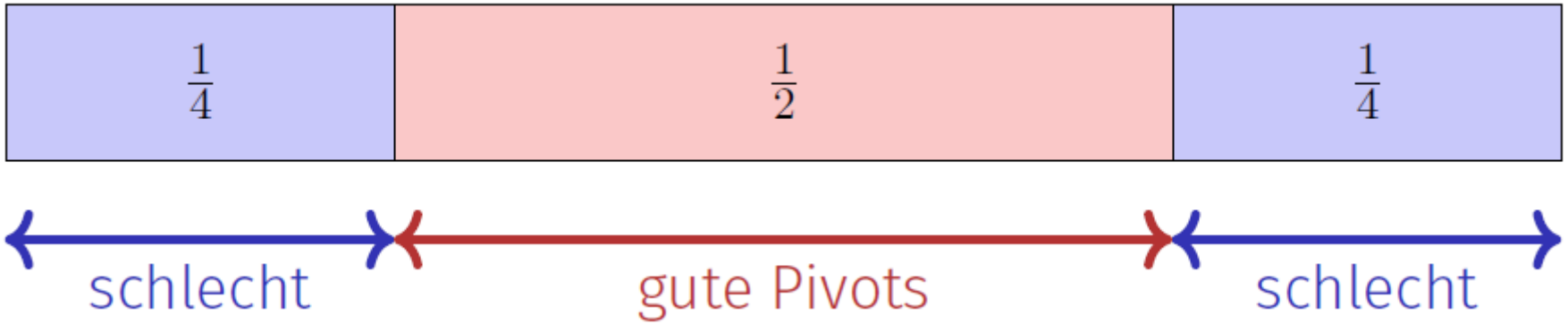
Ein guter Pivot hat linear viele Elemente auf beiden Seiten.



Gute Pivots: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(nlogn)$

Schlechte Pivots: $T(n) = \Theta(n^2)$

QuickSort: Wahl des Pivots



Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.
Wahrscheinlichkeit für guten Pivot nach k Versuchen: $(1-\rho)^{k-1} \cdot \rho$.
Erwartete Anzahl Versuche: $\frac{1}{\rho} = 2$ (Erwartungswert der geometrischen Verteilung:)

QuickSort: Wahl des Pivots

Theorem 3

Im schlechtesten Fall hat Quick Sort eine asymptotische Laufzeit von $\Theta(n^2)$.

Im Mittel hat randomisiertes Quick Sort eine asymptotische Laufzeit von $\Theta(n \cdot \log n)$.

Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: Median3$(A[1], A[n], A[\lfloor n/2 \rfloor])$.

Vergleich der Algorithmen

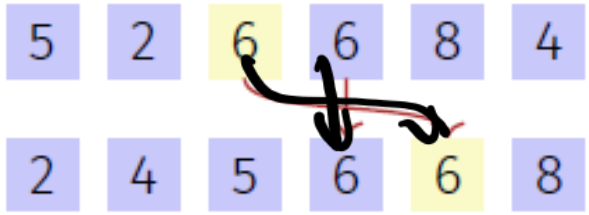
Algorithmus	Swaps/Shifs (best / worst case)	Comparisions (best / worst case)
Bubble Sort	$0/\Theta(n^2)$	$\Theta(n^2)/\Theta(n^2)$
Selection Sort	$0/\Theta(n)$	$\Theta(n^2)/\Theta(n^2)$
Insert Sort	$0/\Theta(n^2)$	$\Theta(n)/\Theta(n^2)$
Quick Sort	$\Theta(n)/\Theta(n\log(n))$	$\Theta(n\log(n))$ / $\Theta(n^2)$
Merge Sort	$\Theta(n\log(n))$	$\Theta(n\log(n))$

01 – Wiederholung der Theorie

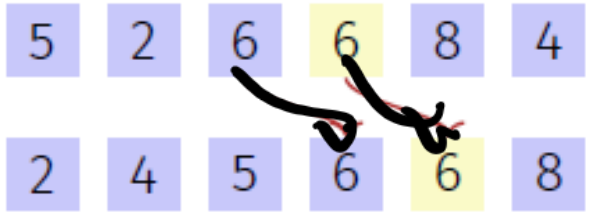
01.1 Rekursive Laufzeiten
01.2 Iterative Suchalgorithmen
01.3 Divide And Counquer
01.4 Stabile und In-situ Algorithmen

Stabile und in-situ Algorithmen

- Stabile Sortieralgorithmen ändern die relative Position von zwei gleichen Elementen nicht.



nicht stabil



stabil

- In-situ-Algorithmen brauchen nur konstant viel zusätzlichen Speicher.

Vergleich der Algorithmen

Algorithmus	Swaps/Shifs (best / worst case)	Comparisions (best / worst case)	Stable	In-situ
Bubble Sort	$0/\Theta(n^2)$	$\Theta(n^2)/\Theta(n^2)$	Yes	Yes
Selection Sort	$0/\Theta(n)$	$\Theta(n^2)/\Theta(n^2)$	No	Yes
Insert Sort	$0/\Theta(n^2)$	$\Theta(n)/\Theta(n^2)$	Yes	Yes
Quick Sort	$\Theta(n)/\Theta(n\log(n))$	$\Theta(n\log(n))$ / $\Theta(n^2)$	No	No
Merge Sort	$\Theta(n\log(n))$	$\Theta(n\log(n))$	Yes	No

Stabile Selection Sort und Quick Sort Algorithmen existieren,
sind aber **kein Teil** dieser Vorlesung

02 – Quiz

Wie oft wird f() aufgerufen?

```
// pre: n is a power of 2 (n = 2^k)
void g(int n){
  if (n>0){
    f();
    g(n/2);
    f();
    g(n/2);
  }
}
```

Wie oft wird f() aufgerufen?

```
// pre: n is a power of 2 (n = 2^k)
void g(int n){
  if (n>0){
    f();
    g(n/2);
    f();
    g(n/2);
  }
}
```

$$M(2^k) = \begin{cases} 2 + 2M(2^{k-1}) & k \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$M(n) = 2M\left(\frac{n}{2}\right) + 2 = 4M\left(\frac{n}{4}\right) + 4 + 2 = 8M\left(\frac{n}{8}\right) + 8 + 4 + 2$$
$$= 2(n + n/2 + n/4 + \ldots + 1) \in \Theta(n)$$

$2(n + n/2 + n/4 + \ldots + 1)$

$2n(1 + 1/2 + 1/4 + \ldots + 1)$

$2n \sum_{L} 1/2^L - 2n \frac{1}{1-1/2} = 4n = O(n)$

geom reihe

Wie oft wird f() aufgerufen?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    g(n/2);
  }
  for (int i = 0; i < n; ++i){
    f();
  }
}
```

Wie oft wird f() aufgerufen?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    g(n/2);
  }
  for (int i = 0; i < n; ++i){
    f();
  }
}
```

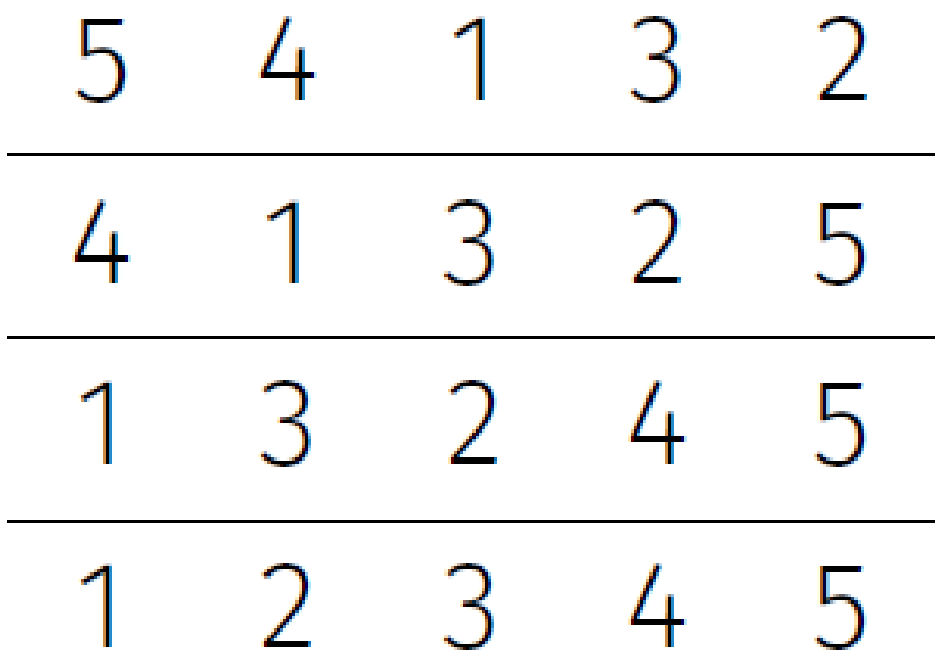
$$M(n) = 2M(n/2) + n = 4M(n/4) + n + 2n/2 = \ldots$$

$$= (k+1)n \in \Theta(n \log n)$$

Welches Algorithmus wird hier aufgerufen?

5	4	1	3	2
4	1	3	2	5
1	3	2	4	5
1	2	3	4	5

Welches Algorithmus wird hier aufgerufen?



Bubble Sort

Welches Algorithmus wird hier aufgerufen?

5	4	1	3	2
1	4	5	3	2
1	2	5	3	4
1	2	3	5	4
1	2	3	4	5

Welches Algorithmus wird hier aufgerufen?

5	4	1	3	2
1	4	5	3	2
1	2	5	3	4
1	2	3	5	4
1	2	3	4	5

Selection Sort

Welches Algorithmus wird hier aufgerufen?

5	4	1	3	2
4	5	1	3	2
1	4	5	3	2
1	3	4	5	2
1	2	3	4	5

Welches Algorithmus wird hier aufgerufen?

5	4	1	3	2
4	5	1	3	2
1	4	5	3	2
1	3	4	5	2
1	2	3	4	5

Insertion Sort

Im Folgenden ist der Zustand eines Arrays vor und nach dem letzten Schritt eines binären rekursiven Mergesort-Algorithmus angegeben. Setzen Sie die fehlenden Zahlen entsprechend ein.

3	10	5	8	6	1	9	2

1	2	3	5	6	8	9	10

Im Folgenden ist der Zustand eines Arrays vor und nach dem letzten Schritt eines binären rekursiven Mergesort-Algorithmus angegeben. Setzen Sie die fehlenden Zahlen entsprechend ein.

3	5	6	8	1	2	9	10

1	2	3	5	6	8	9	10

Gegeben sei das folgende Array, das nach dem ersten Quicksort-Aufteilungsschritt entstanden ist. Welcher Schlüssel wurde als Pivot-Element verwendet? Markieren Sie **alle** möglichen Kandidaten.

1	3	2	5	4	6	9	8	7	10
1	2	3	4	5	6	7	8	9	10

Gegeben sei das folgende Array, das nach dem ersten Quicksort-Aufteilungsschritt entstanden ist. Welcher Schlüssel wurde als Pivot-Element verwendet? Markieren Sie **alle** möglichen Kandidaten.

1	3	2	5	4	6	9	8	7	10
1	2	3	4	5	6	7	8	9	10

03 – Programmierübung

Programmieraufgabe

1. Geh auf CodeExpert → Code Examples → Binary Search
2. Implementiere den Binary Search Algorithmus

```cpp
#include <vector>

using Iterator = std::vector<int>::iterator;

// pre: begin, end provide an interator pair for a sorted vector of integers
// post: return the position of value in the array
//       if it does not exist, return the end iterator
Iterator binarySearch(Iterator begin, Iterator end, int value){
  // try to implement me
  return end;
}
```

Input: Aufsteigend sortiertes Array A von n Schlüsseln. Schlüssel b.
Output: Wenn $b \in A$, dann $i \in [1, n]$ mit $A[i] = b$, sonst 0

$m \leftarrow \lfloor (1 + n)/2 \rfloor$
if $n = 0$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then** // gefunden
    return m
else if $b < A[m]$ **then** // Element liegt links
    return BSearch$(A[1, \ldots, m-1], b)$
else // $b > A[m]$: Element liegt rechts
    $k \leftarrow$ BSearch$(A[m+1, \ldots, n], b)$
    if $k = 0$ **then return** 0
    return m+k

Programmieraufgabe

```cpp
#include <vector>

using Iterator = std::vector<int>::iterator;

// pre: begin, end provide an interator pair for a sorted vector of integers
// post: return the position of value in the array
//       if it does not exist, return the end iterator
Iterator binarySearch(Iterator begin, Iterator end, int value){
  Iterator left = begin;
  Iterator right = end;
  while (left != right){
    unsigned n = right-left;
    unsigned m = n / 2;
    Iterator middle = left + m;
    if (value == *middle)
      return middle;
    else if (value < *middle)
      right = middle;
    else if (value > *middle)
      left = middle + 1;
  }
  return end;
}
```

Input: Aufsteigend sortiertes Array A von n Schlüsseln. Schlüssel b.
Output: Wenn $b \in A$, dann $i \in [1, n]$ mit $A[i] = b$, sonst 0

$m \leftarrow \lfloor (1 + n)/2 \rfloor$
if $n = 0$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then** // gefunden
    return m
else if $b < A[m]$ **then** // Element liegt links
    return BSearch$(A[1, \ldots, m-1], b)$
else // $b > A[m]$: Element liegt rechts
    $k \leftarrow$ BSearch$(A[m+1, \ldots, n], b)$
    if $k = 0$ **then return** 0
    return m+k

ETH zürich

Michal Oskedra
Teaching Assistent D-ITET
moskedra@ethz.ch