



# Analyse von Algorithmen

## Sortieralgorithmen (Teil 1)

SoSe 2020

**Prof. Dr. Margarita Esponda**

Freie Universität Berlin

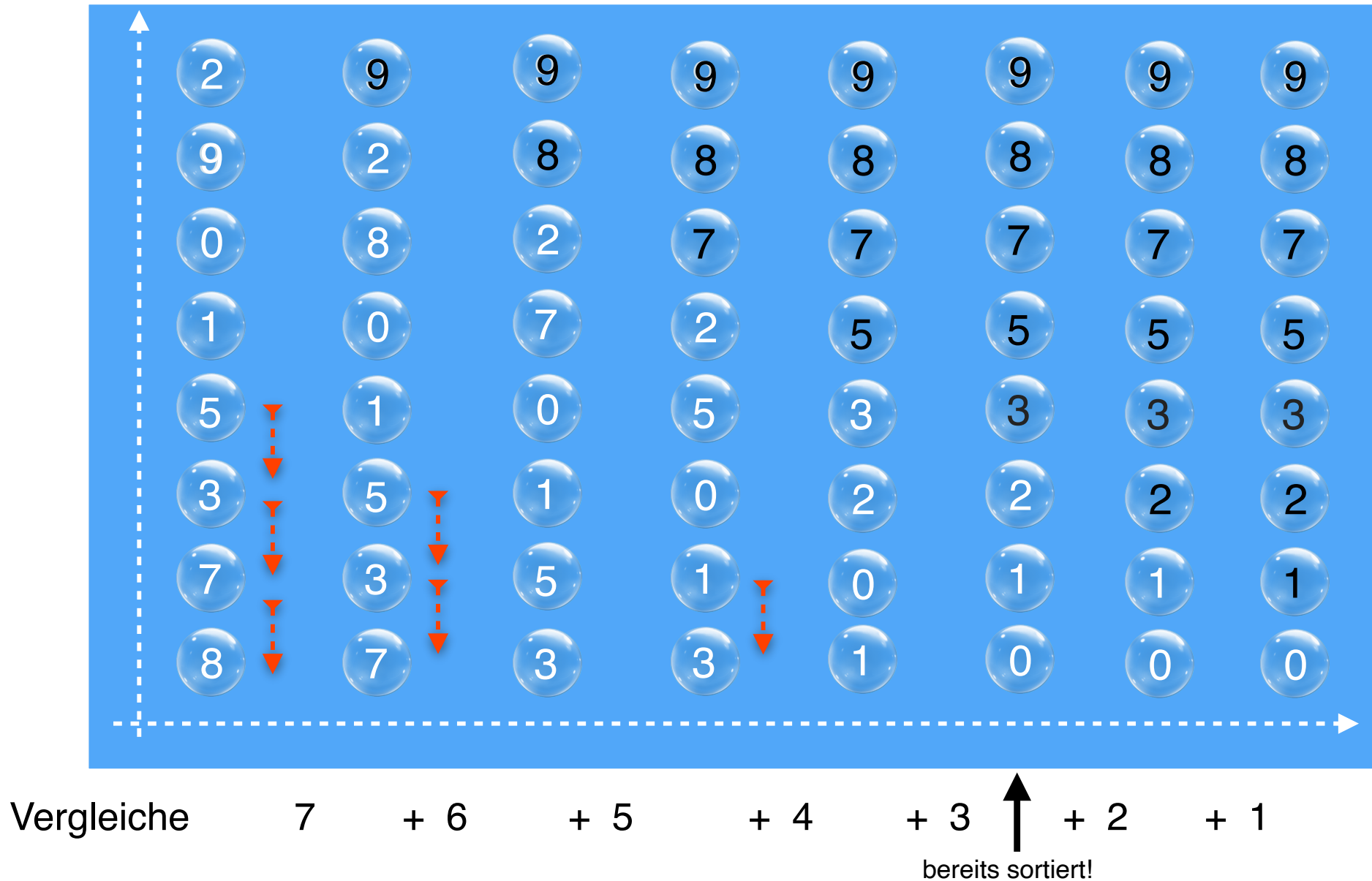
# Sortieralgorithmen

## Vergleichsalgorithmen

Bubble-Sort  
Insert-Sort  
Selection-Sort  
Shell-Sort  
Quicksort  
Mergesort  
Heap-Sort

## Ohne Vergleiche

Counting-Sort  
Radix-Sort  
Bucket-Sort



## Bubble-Sort

```
def bubblesort (A):  
    swap = True  
    stop = len(A)-1  
    while swap:  
        swap = False  
        for i in range(stop):  
            if A[i]>A[i+1]:  
                A[i], A[i+1] = A[i+1], A[i]  
                swap = True  
        stop = stop-1
```

Hilfsvariable für einen linearen Aufwand, wenn die Daten sortiert sind.

Hier wird die Stabilitätseigenschaft des Algorithmus garantiert

# Bubble-Sort

Einfachster und ältester Sortieralgorithmus

- \* **In-Place**

minimaler zusätzlicher konstanter Speicherplatz  **$O(1)$**

- \* **Stabil**

die Reihenfolge von gleichen Daten bleibt unverändert

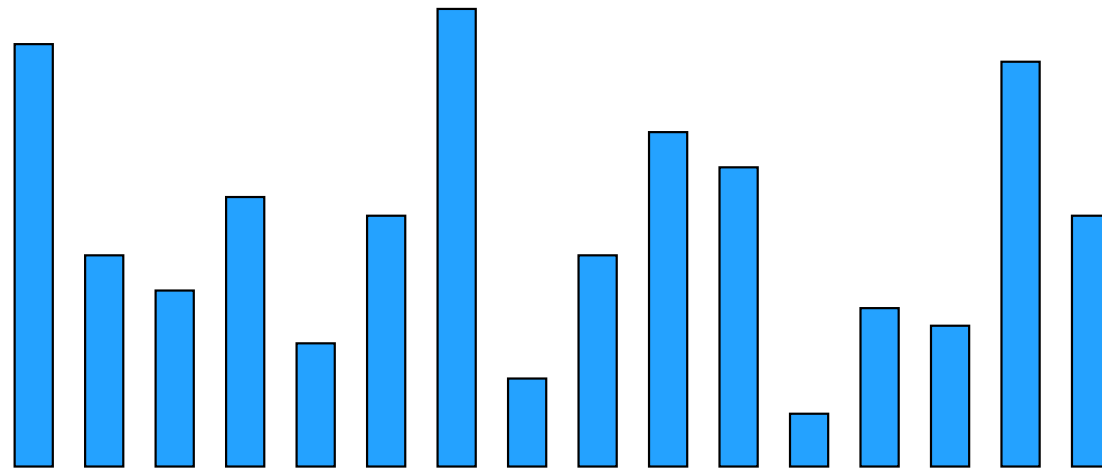
- \* **zu naiv** und **ineffizient** für das Sortieren von im Speicher zusammenhängenden Informationen

- \* jedoch eignet er sich für das Sortieren innerhalb **verketteter Listen**.

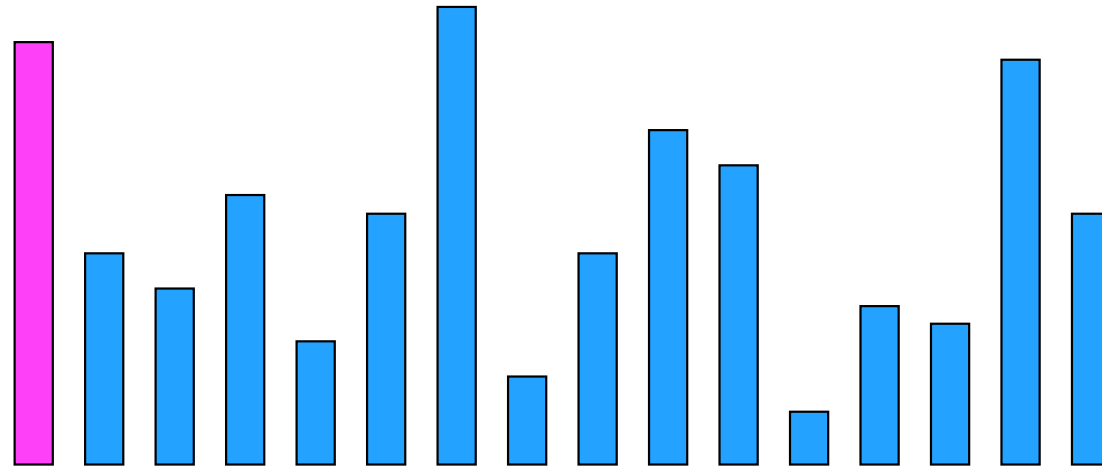
- \* quadratischer Aufwand  **$O(n^2)$**



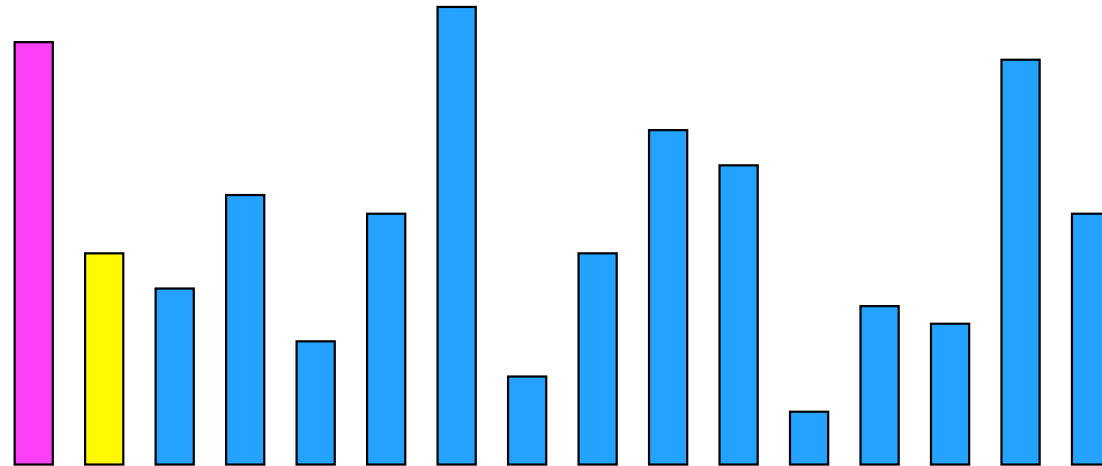
# Insertion-Sort



# Insertion-Sort

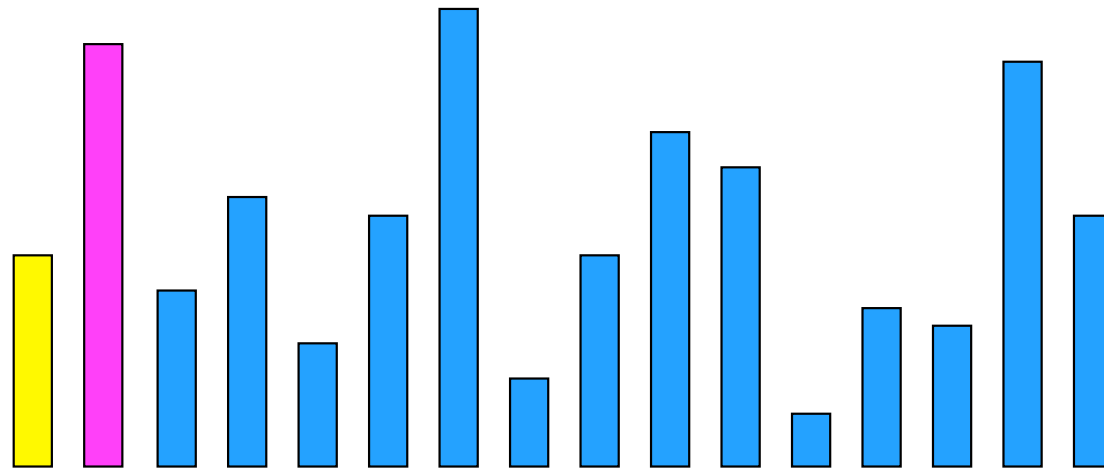


# Insertion-Sort

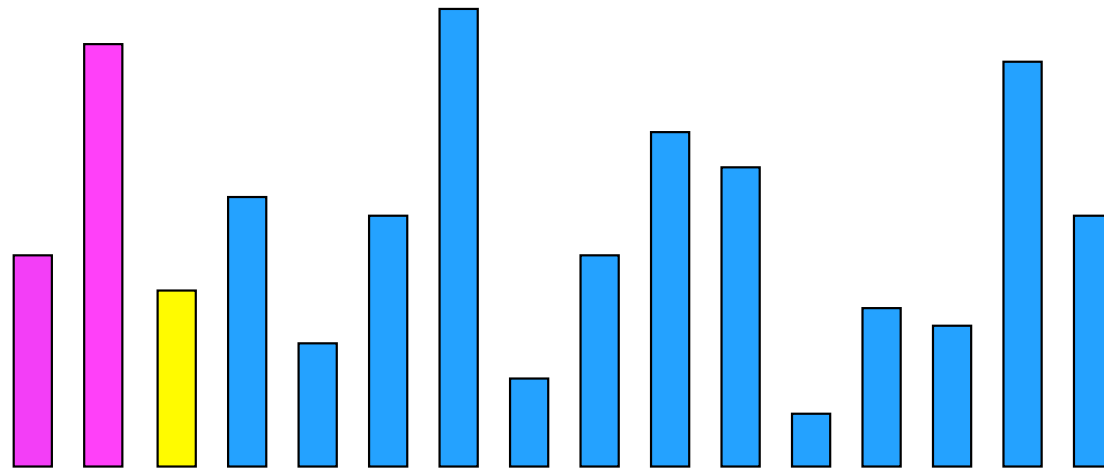




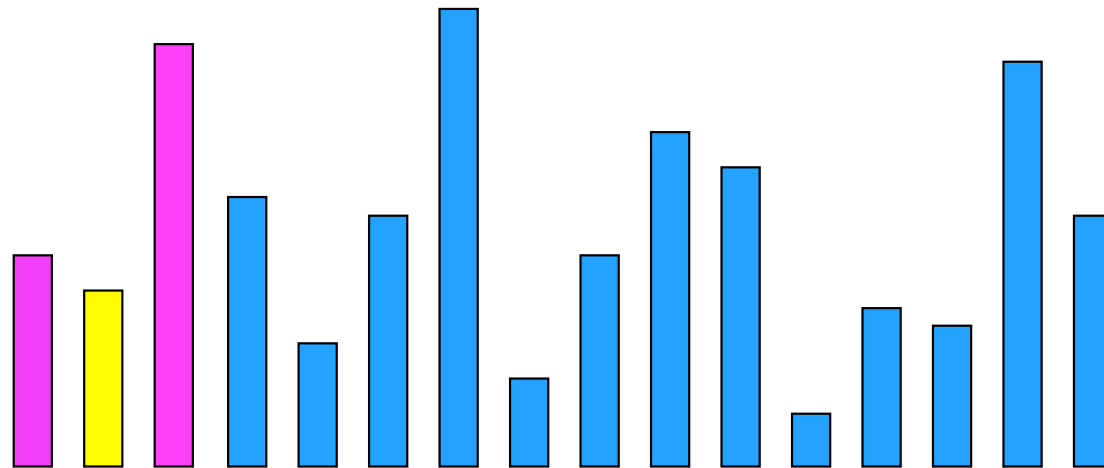
# Insertion-Sort



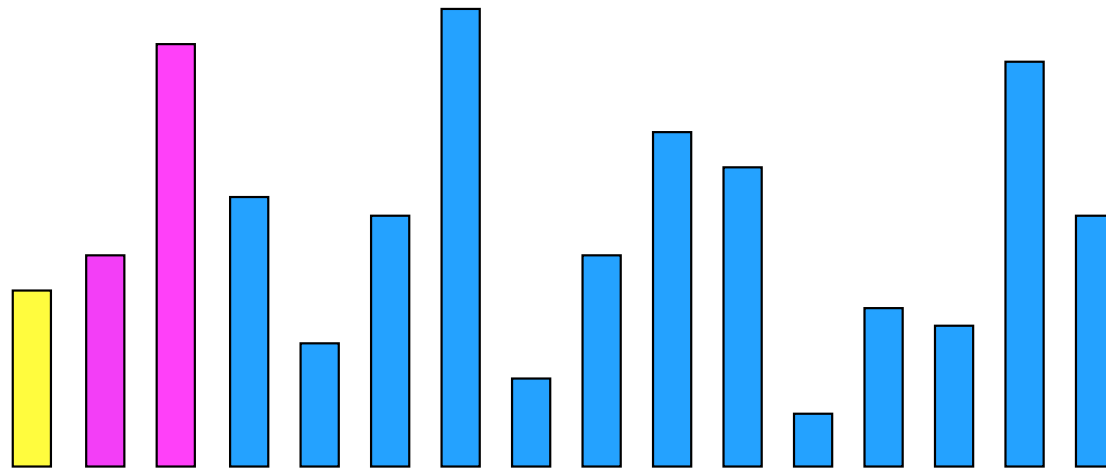
# Insertion-Sort



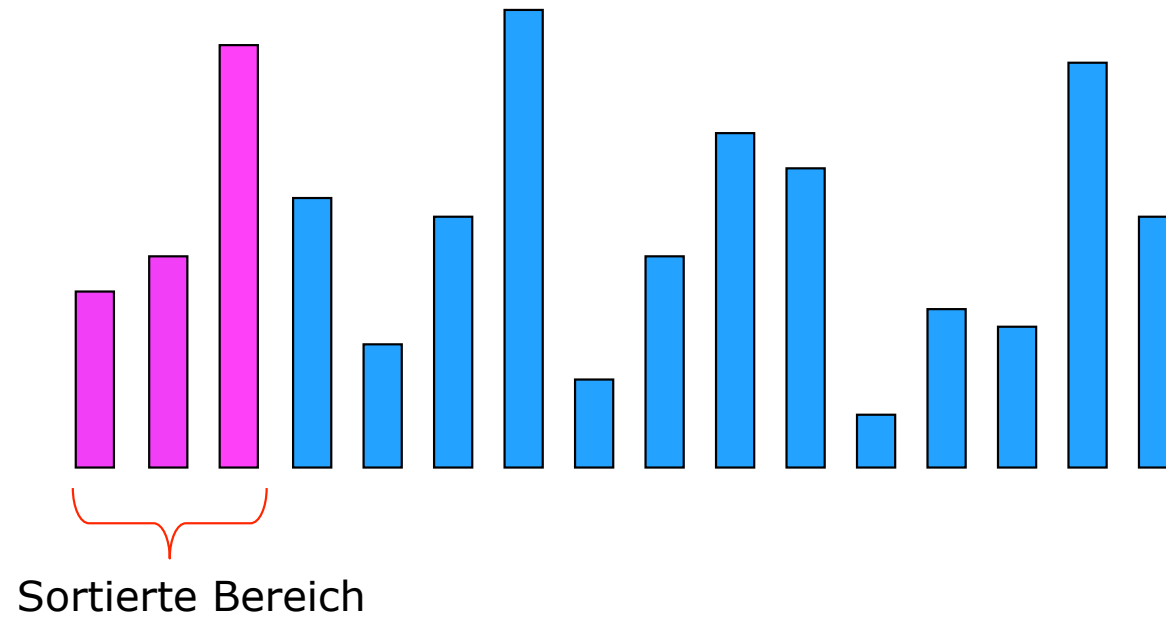
# Insertion-Sort



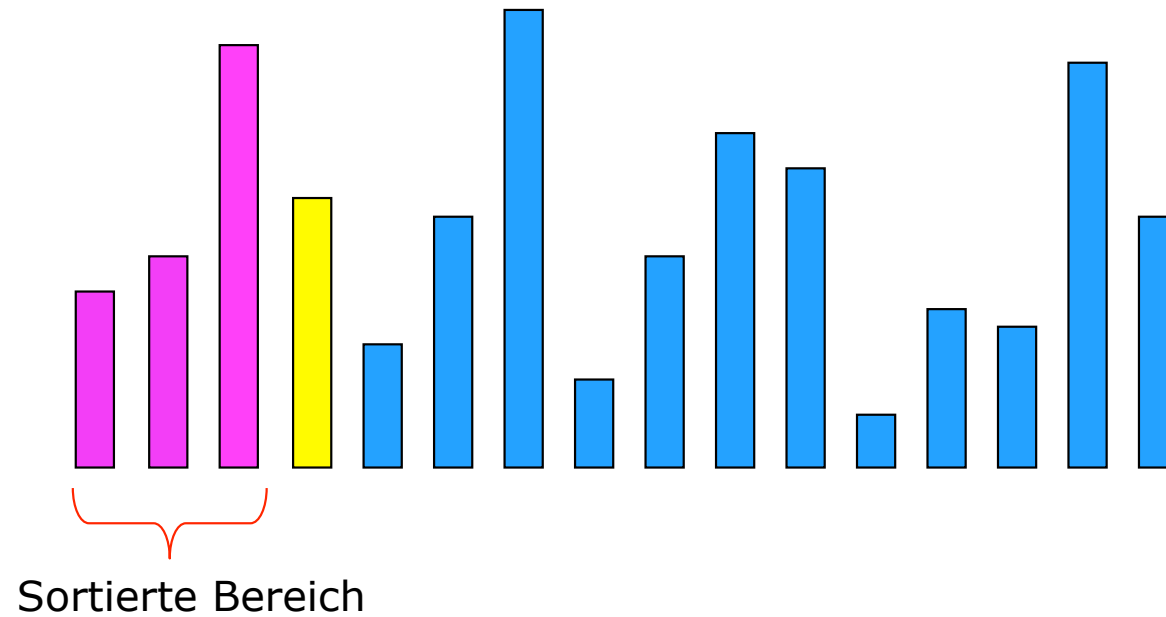
# Insertion-Sort



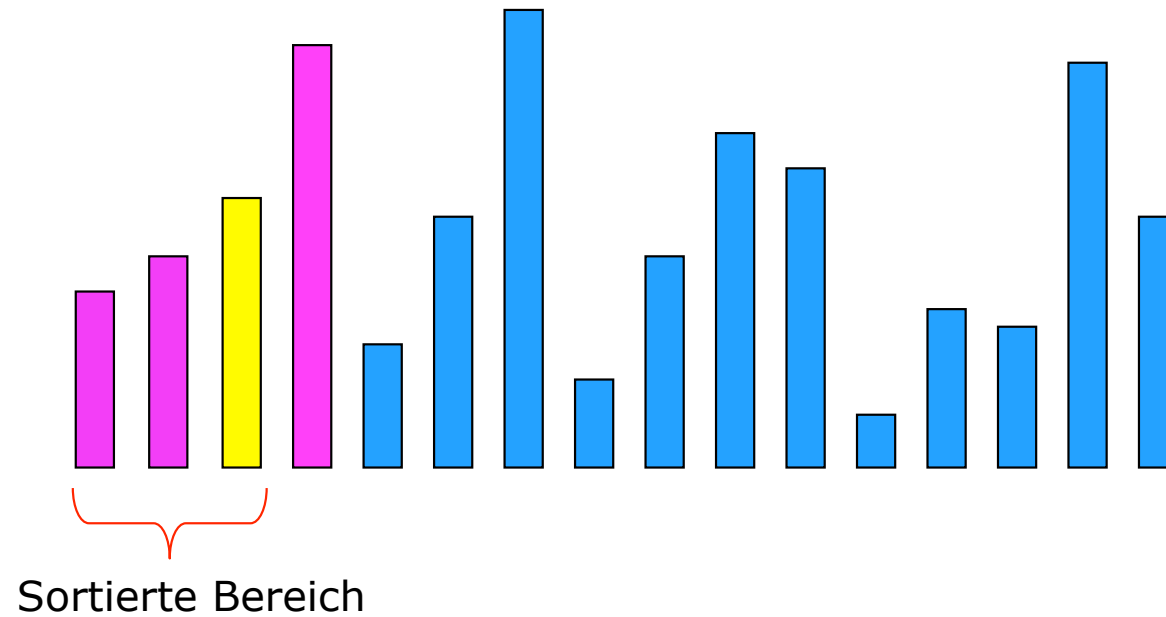
# Insertion-Sort



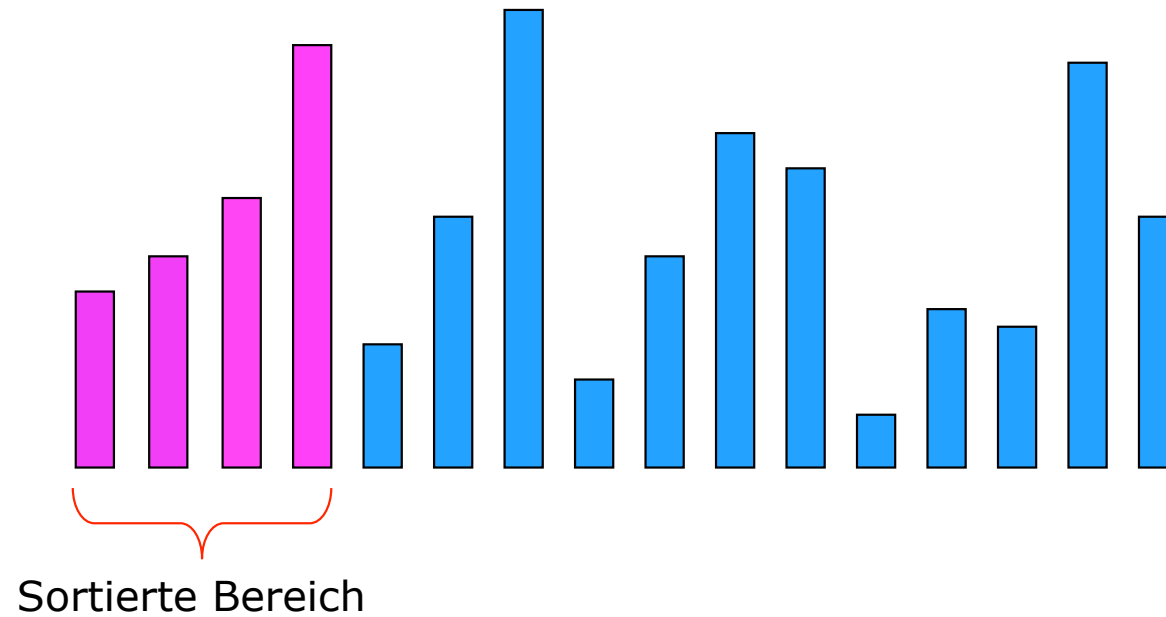
# Insertion-Sort



# Insertion-Sort

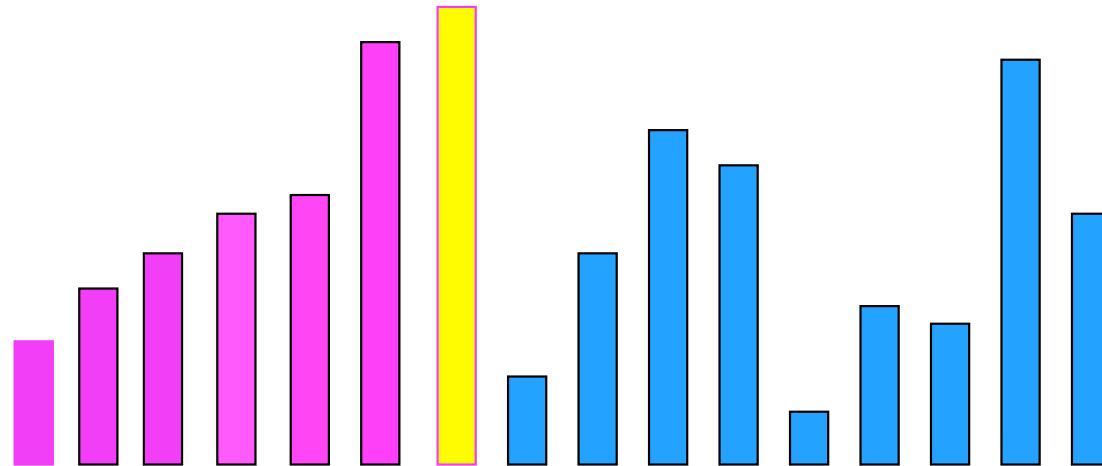


# Insertion-Sort

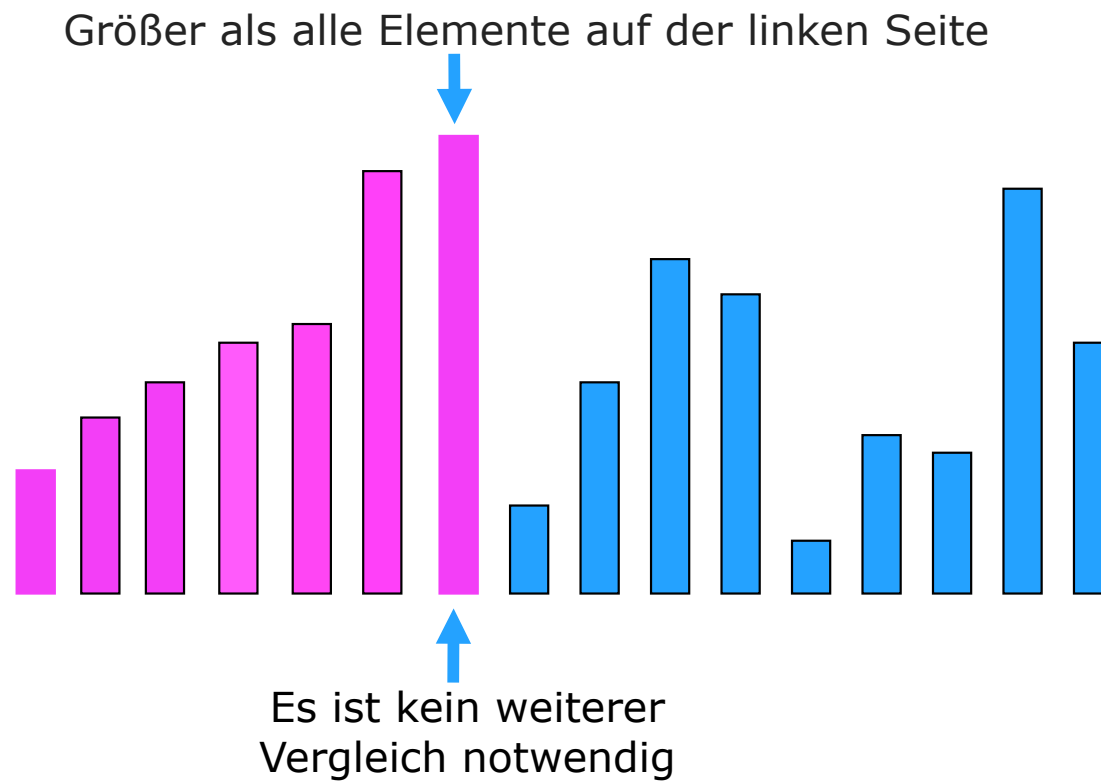




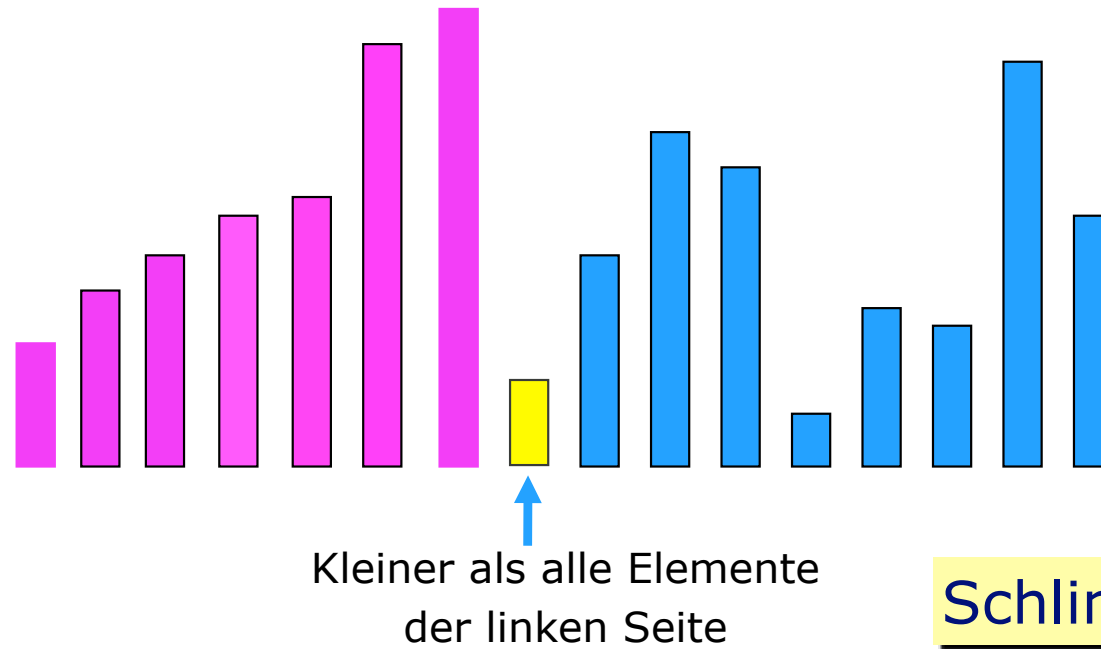
# Insertion-Sort



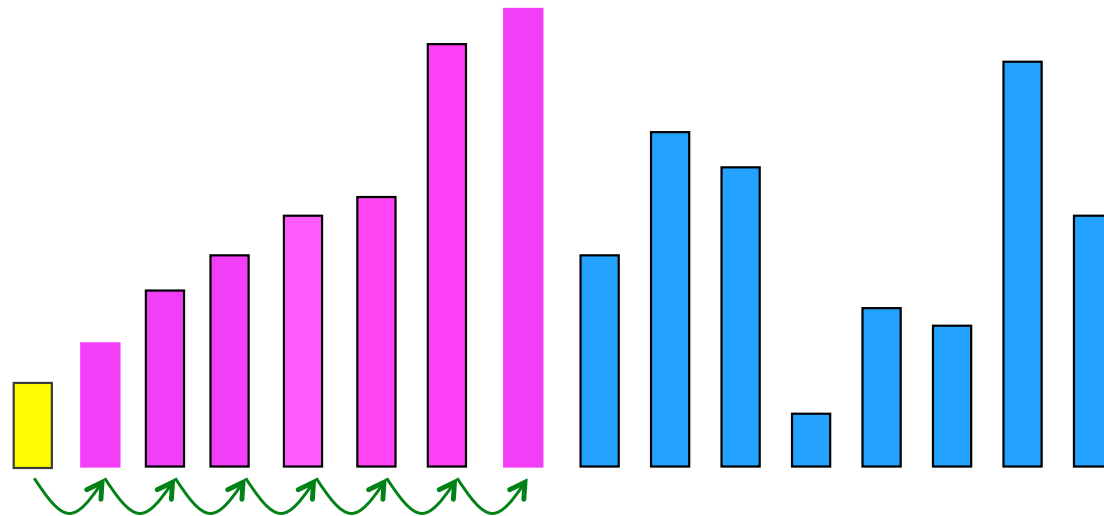
# Insertion-Sort



# Insertion-Sort

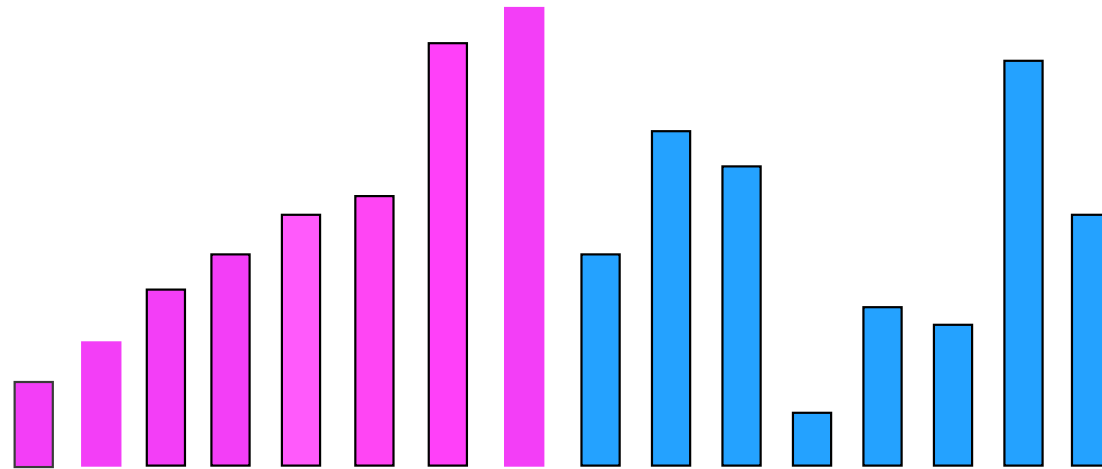


# Insertion-Sort



Alle Elemente müssen verschoben werden

# Insertion-Sort



## Einfacher Sortieralgorithmus

- **In-Place** (zusätzlicher Speicherbedarf  **$O(1)$** )
- **Stabil**
- **gut für kleine Mengen** oder **leicht unsortierte Daten**

# Insertion-Sort

```
def insertsort(seq):  
    for j in range(1, len(seq)):  
        key = seq[j]  
        k = j-1  
        while k >= 0 and seq[k] > key:  
            seq[k+1] = seq[k]  
            k = k-1  
        seq[k+1] = key
```

Eine geeignete Position wird gesucht und die Elemente des sortierten Bereichs verschoben

Die einzusortierende Zahl wird in den gefundenen Platz kopiert

# Laufzeit

```
def insertsort (seq):
```

```
    for j in range(1, len(seq)): .....▶  $C_2$ 
```

```
        key = seq[j] .....▶  $C_3$ 
```

```
        k = j-1; .....▶  $C_4$ 
```

```
        while k >= 0 and seq[k] > key: .....▶  $C_5$ 
```

```
            seq[k+1] = seq[k] .....▶  $C_6$ 
```

```
            k = k-1 .....▶  $C_7$ 
```

```
        seq[k+1] = key .....▶  $C_8$ 
```

Anzahl der Operationen

Max

Min

$n$

$n$

$n-1$

$n-1$

$n-1$

$n-1$

$1+2+\dots+n$

$n-1$

$1+2+\dots+n-1$

0

$1+2+\dots+n-1$

0

$n-1$

$n-1$

## Maximale Laufzeit ("worst case")

$$T(n) = c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5(1+2+\dots+n) + (c_6 + c_7)(1+2+\dots+(n-1))$$

$$1+2+3+\dots+n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

$$\frac{n(n+1)}{2}$$

$$\frac{(n-1)n}{2}$$

$$T(n) = \mathbf{c_2 n} + (\mathbf{c_3 + c_4 + c_8})(\mathbf{n-1}) + \mathbf{c_5 n} + (c_5 + c_6 + c_7)(\mathbf{1+2+\dots+(n-1)})$$

$$T(n) = -c_3 - c_4 - c_8 + (c_2 + c_3 + c_4 + c_8 + c_5)n + (c_5 + c_6 + c_7)\left(\frac{\mathbf{(n-1)n}}{\mathbf{2}}\right)$$

$$T(n) = \underbrace{-(c_3 + c_4 + c_8)}_c + \underbrace{\left(c_2 + c_3 + c_4 + c_8 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2}\right)}_b n + \underbrace{\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)}_a n^2$$

$$T(n) = \underline{a}n^2 + bn + c$$

$$T(n) = \mathbf{O(n^2)}$$



## Minimale Laufzeit ("best case")

$$T(n) = c_1 + c_2n + (c_3 + c_4 + c_5 + c_8)(n-1)$$

$$T(n) = \underbrace{(c_1 - c_3 - c_4 - c_5 - c_8)}_a + \underbrace{(c_2 + c_3 + c_4 + c_5 + c_8)}_b(n)$$

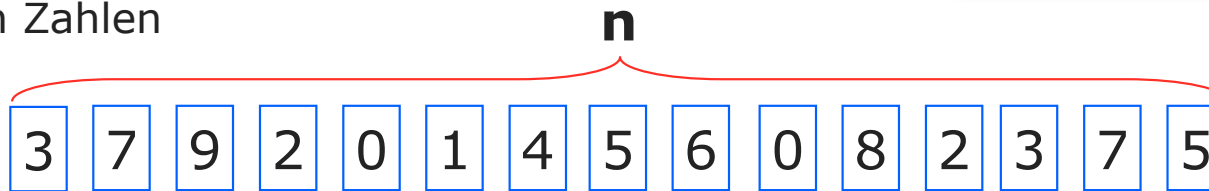
$$T(n) = a + b n$$

$$T(n) = \mathbf{O(n)}$$

# Insertion-Sort

```
def insertsort(seq):  
    for j in range(1, len(seq)):  
        key = seq[j]  
        k = j-1;  
        while k >= 0 and seq[k] > key:  
            seq[k+1] = seq[k]  
            k = k-1  
        seq[k+1] = key
```

**Eingabe:** n Zahlen



**Berechnungsschritte:** Vergleichsoperationen

**Komplexitätsanalyse:**  $T(n) = 1+2+3+\dots+(n-1) + \mathbf{n}$   
$$= \frac{(n+1)n}{2}$$

Im schlimmsten Fall

$$T(n) = an^2 + bn = O(n^2)$$

Im besten Fall

$$T(n) = an + b = O(n)$$

# Teile und Herrsche

## "Divide und Conquer"

Viele Probleme lassen sich **nicht mit trivialen Schleifen** lösen und haben gleichzeitig den Vorteil, dass eine rekursive Lösung keine überflüssigen Berechnungen verursacht.

Solche Probleme lassen sich in Teilprobleme zerlegen, deren Lösung keine überlappenden Berechnungen beinhalten.

### Lösungsschema:

#### Divide:

Teile ein Problem in zwei oder mehrere kleinere ähnliche Teilprobleme, die (rekursiv) isoliert behandelt werden können.

#### Conquer:

Löse die Teilprobleme auf dieselbe Art (rekursiv).

#### Merge:

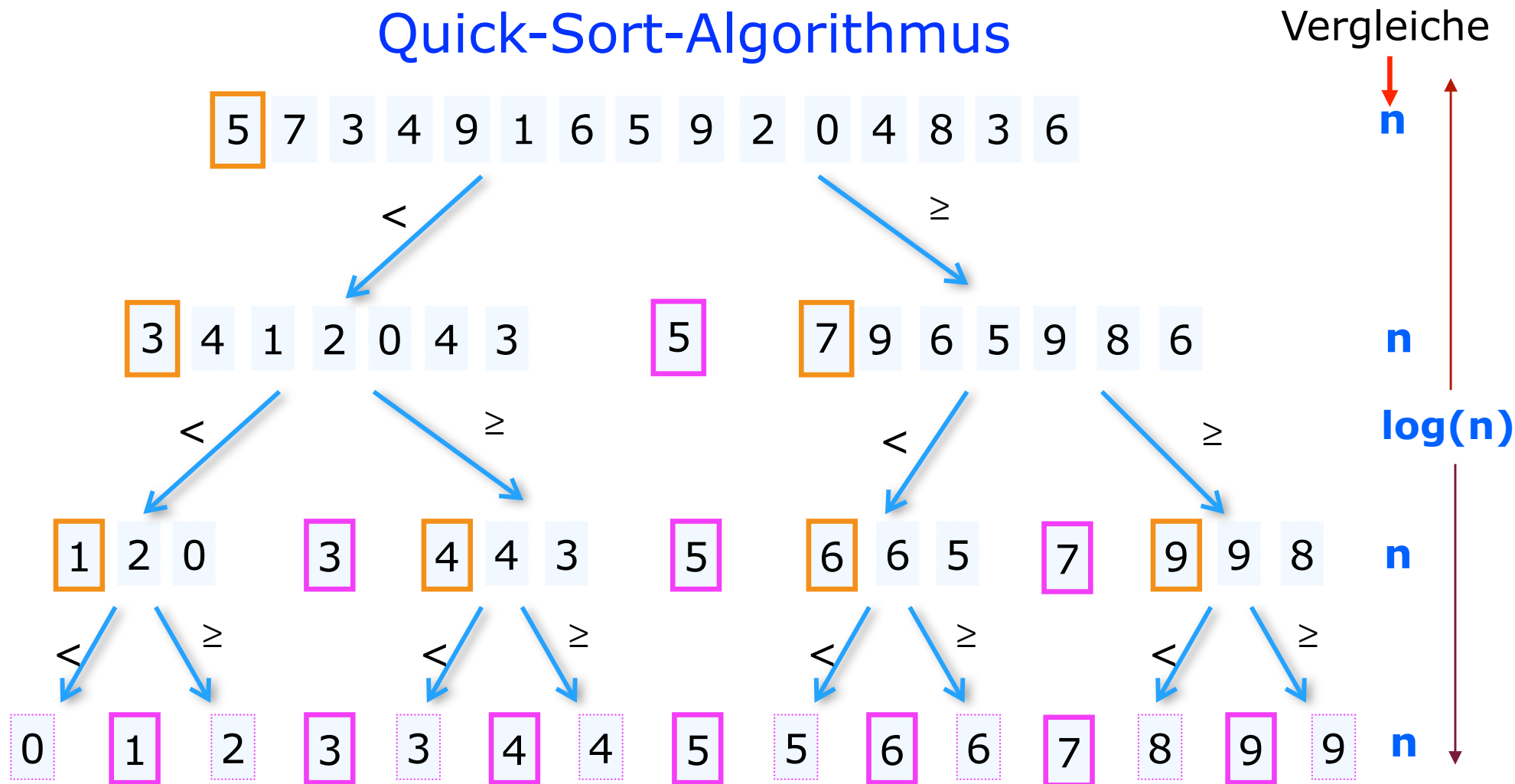
Füge die Teillösung zur Gesamtlösung zusammen.

# Quick-Sort-Algorithmus

## Grundidee:

- 1) Ein Element (**Pivot**) aus dem Array wird gewählt
- 2) Alle Zahlen des Arrays werden mit dem Pivot-Element verglichen und während des Vergleichsdurchlaufs in zwei Bereiche umorganisiert (**Partitionierung**). Der erste Bereich beinhaltet die Zahlen, die kleiner als das Pivot-Element sind und der zweite alle, die größer oder gleich sind. Am Ende des Durchlaufs wird das Pivot-Element zwischen beider Bereichen positioniert.
- 3) Nach jeder Partitionierung wird der Quicksort-Algorithmus rekursiv mit beiden Teilbereichen ausgeführt (solange die Teilbereiche mehr als ein Element beinhalten).

## Quick-Sort-Algorithmus



Der Quicksort-Algorithmus funktioniert am besten, wenn die Teillisten fast gleich gross sind.

**im besten Fall!**  
 $T(n) = n (\log(n))$

## Quick-Sort-Algorithmus

**Im schlimmsten Fall!**

sind alle Elemente  
bereits sortiert.

Anzahl der Vergleiche



$$T(n) = (n-1) + \dots 2 + 1$$

$$T(n) = (n-1)*(n-2)/2$$

$$T(n) = 1/2*n^2 - 3/2*n + 2$$

$$T(n) = c_1 n^2 + c_2 n + c_3$$

$$T(n) = \mathbf{O}(n^2)$$



## Quick-Sort-Algorithmus

```
quicksort :: [Integer] -> [Integer]
quicksort [] = []
quicksort (x:xs) =  quicksort [ y | y <- xs, y <= x ]
                    ++ [x]
                    ++ quicksort [ y | y <- xs, y > x ]
```

Das **Problem** in Haskell ist der **Speicherverbrauch** und die Komplexität der **Verkettungsfunktion** (`++`).

## Quick-Sort-Algorithmus á la Haskell

```
def quick_sort(seq):  
    if len(seq)>1:  
        q1 = [s for s in seq[1:] if s<seq[0]]  
        q2 = [s for s in seq[1:] if s>=seq[0]]  
        return quick_sort(q1) + [seq[0]] + quick_sort(q2)  
    else:  
        return seq
```



Speicherverbrauch?

Komplexität der Verkettungsfunktion (+)?



# Quicksort-Algorithmus

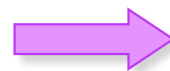
imperativ!

Rekursive Implementierung

```
def quicksort (A, low, high ):  
    if low<high:  
        m = partition(A, low, high )  
        quicksort ( A, low, m-1 )  
        quicksort ( A, m+1, high )
```

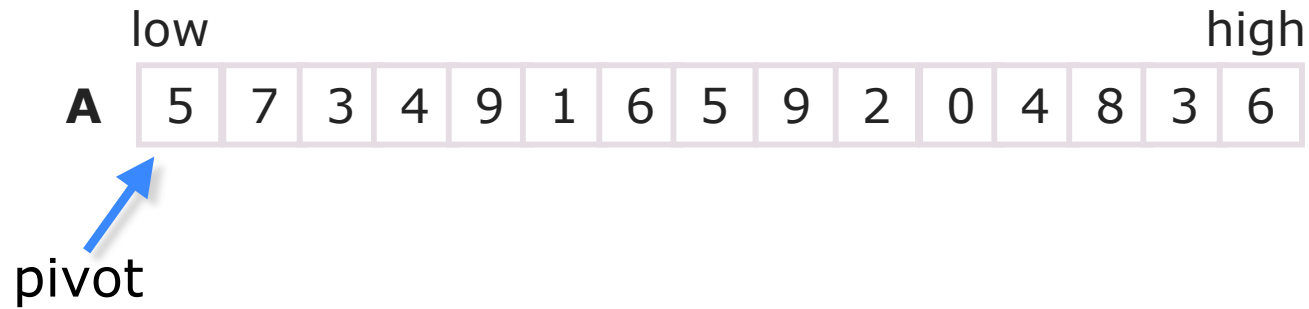
	low													high	
<b>A</b>	5	7	3	4	9	1	6	5	9	2	0	4	8	3	6

Sortieren am Ort

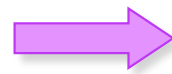


```
def partition( A, low, high ):  
    pivot = A[low]  
    i = low  
    for j in range(low+1,high+1):  
        if ( A[j] < pivot ):  
            i=i+1  
            A[i], A[j] = A[j], A[i]  
    A[i], A[low] = A[low], A[i]  
    return i
```

## Quicksort-Algorithmus



Sortieren am Ort

**def** partition( A, low, high ):

pivot = A[low]

i = low

**for** j **in** range(low+1,high+1):

if ( A[j] &lt; pivot ):

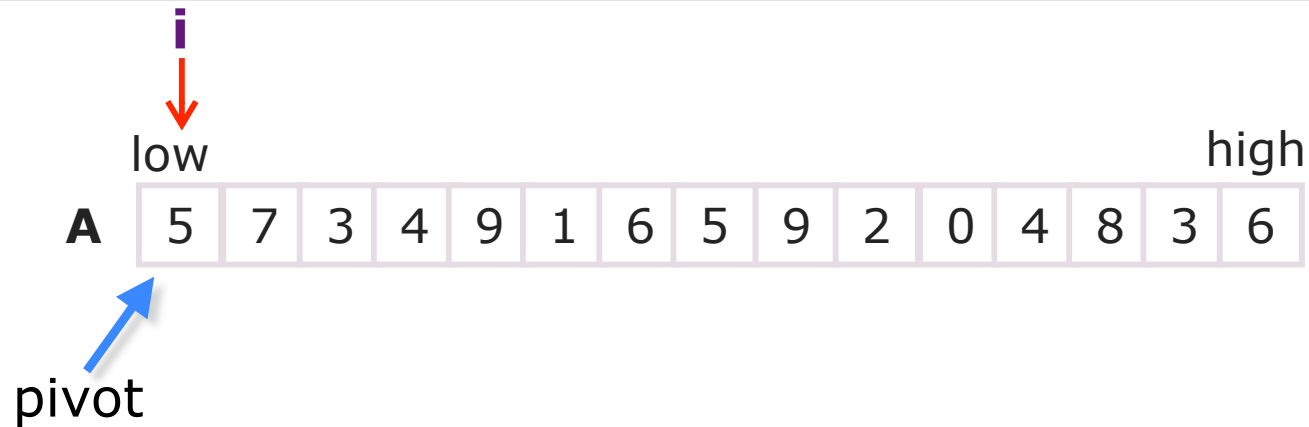
i=i+1

A[i], A[j] = A[j], A[i]

A[i], A[low] = A[low], A[i]

**return** i

## Quicksort-Algorithmus

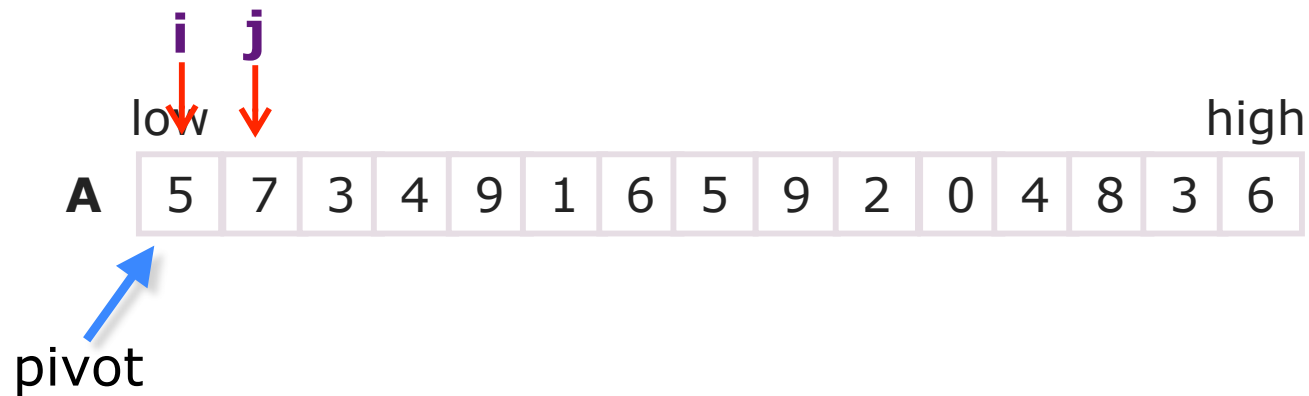


Sortieren am Ort



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus

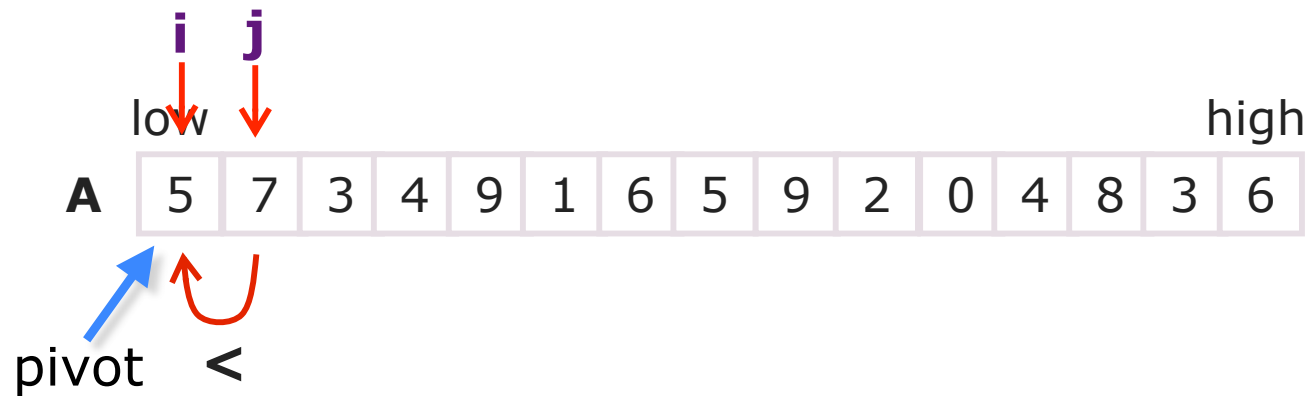


Sortieren am Ort

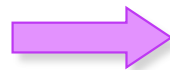


```
def partition( A, low, high ):  
    pivot = A[low]  
    i = low  
    for j in range(low+1,high+1):  
        if ( A[j] < pivot ):  
            i=i+1  
            A[i], A[j] = A[j], A[i]  
    A[i], A[low] = A[low], A[i]  
    return i
```

## Quicksort-Algorithmus

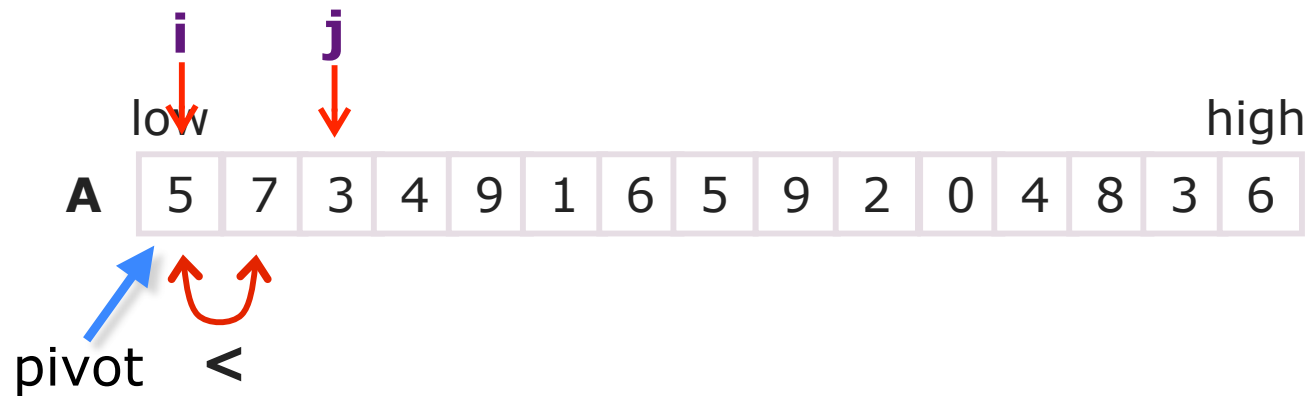


Sortieren am Ort

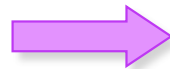


```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus

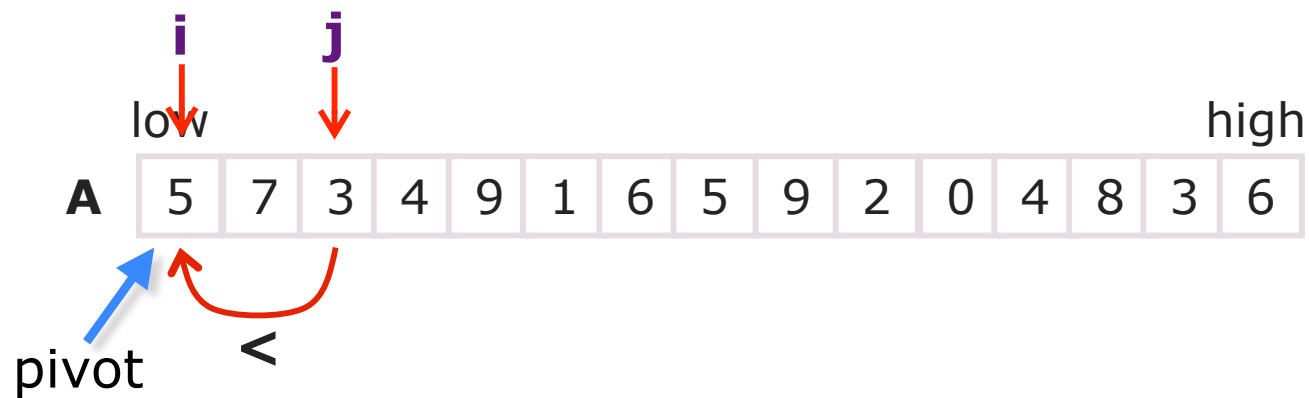


Sortieren am Ort

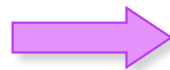


```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



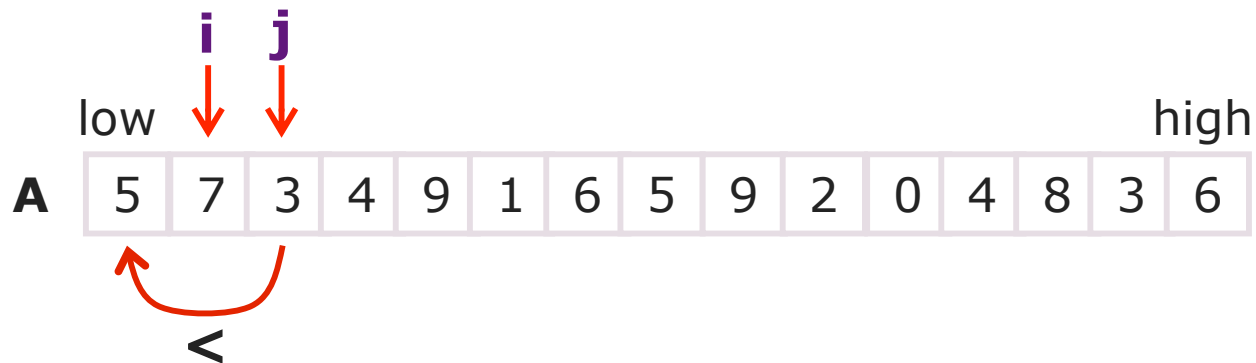
Sortieren am Ort



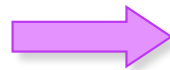
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



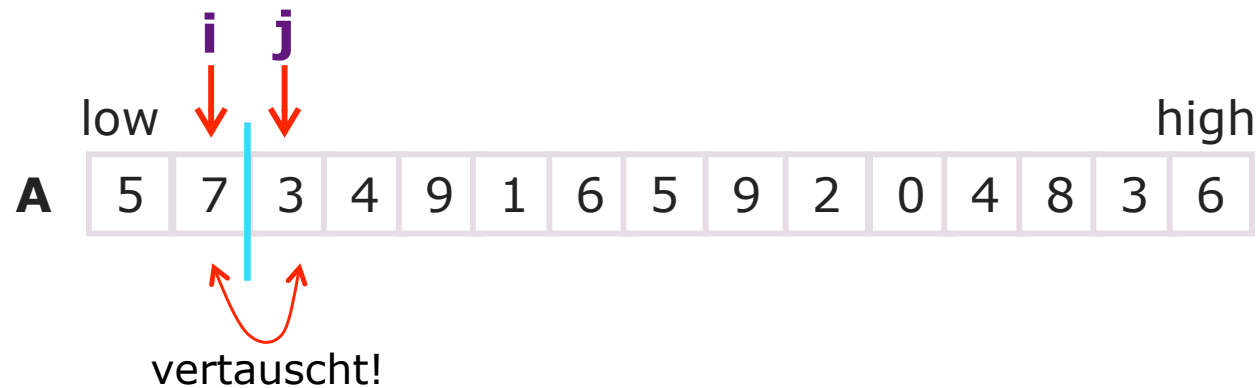


Sortieren am Ort



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

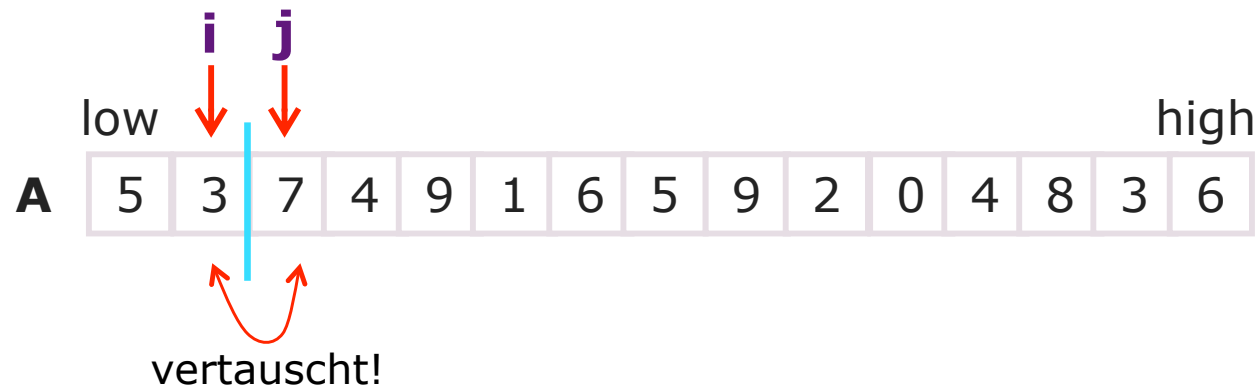
## Quicksort-Algorithmus



Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

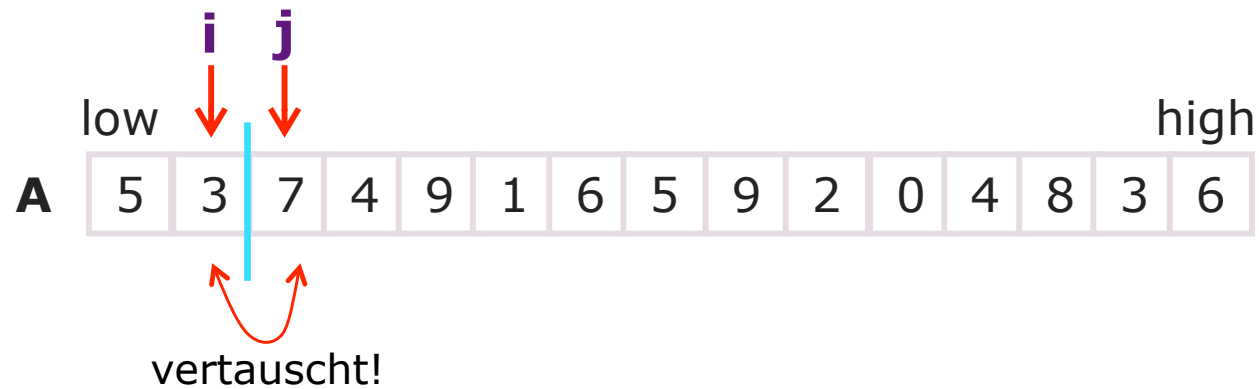
Quicksort-Algorithmus



Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

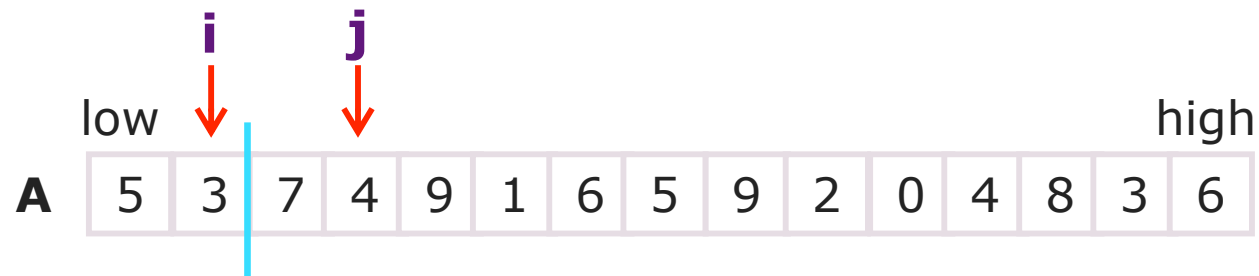
## Quicksort-Algorithmus



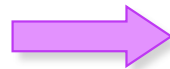
Sortieren am Ort

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus

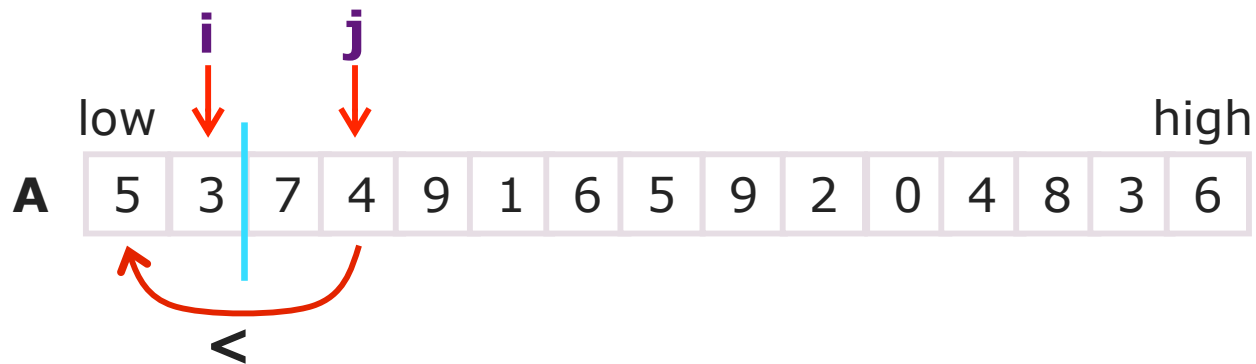


Sortieren am Ort



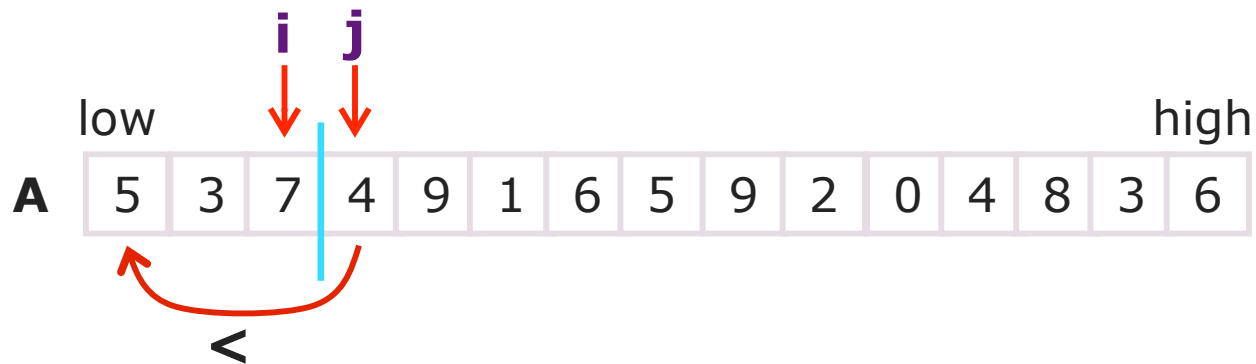
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



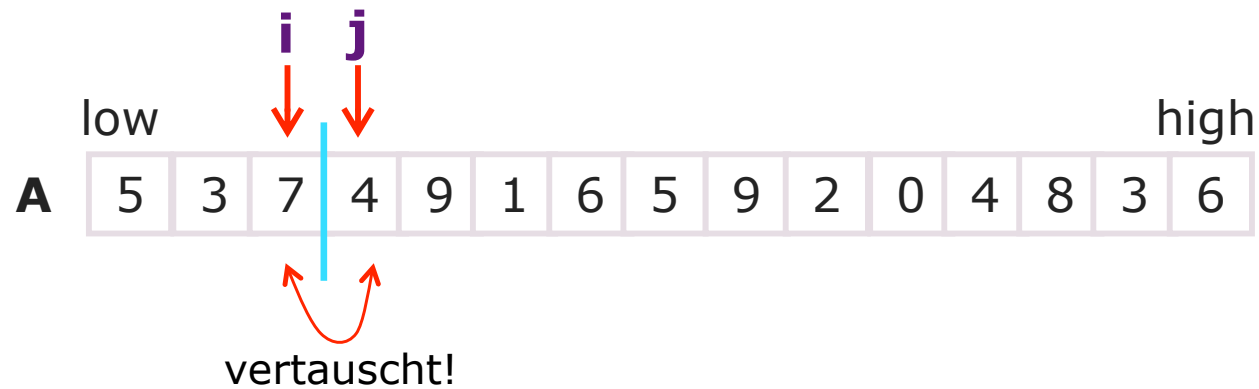
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
        A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

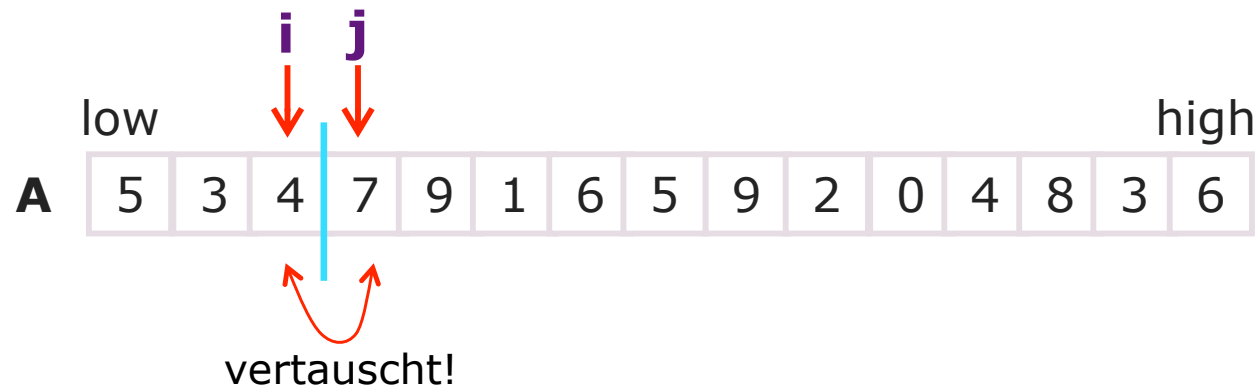
## Quicksort-Algorithmus



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

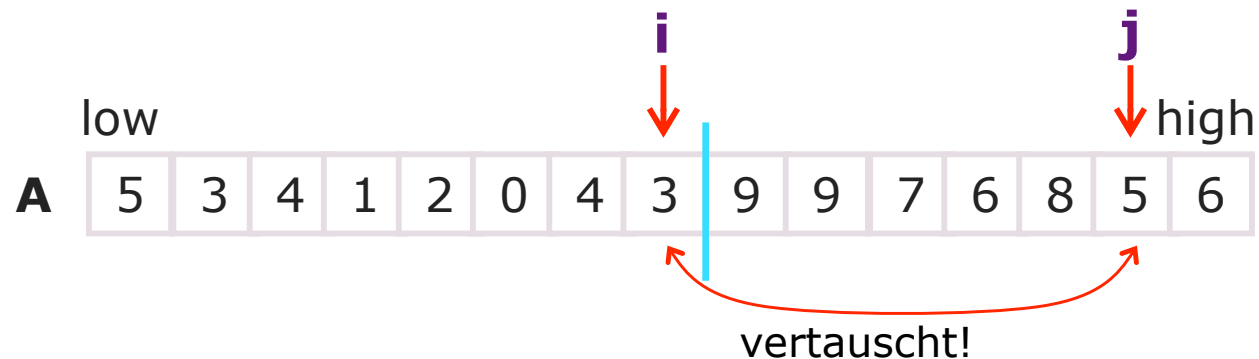
## Quicksort-Algorithmus





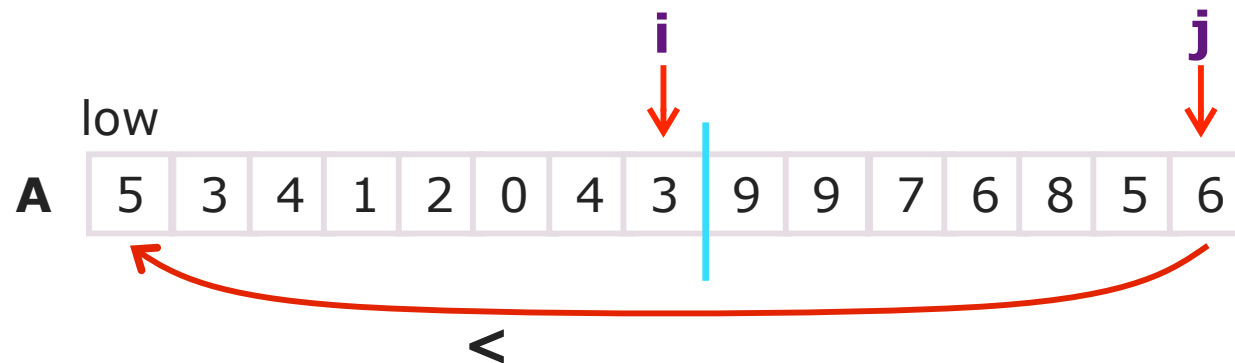
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



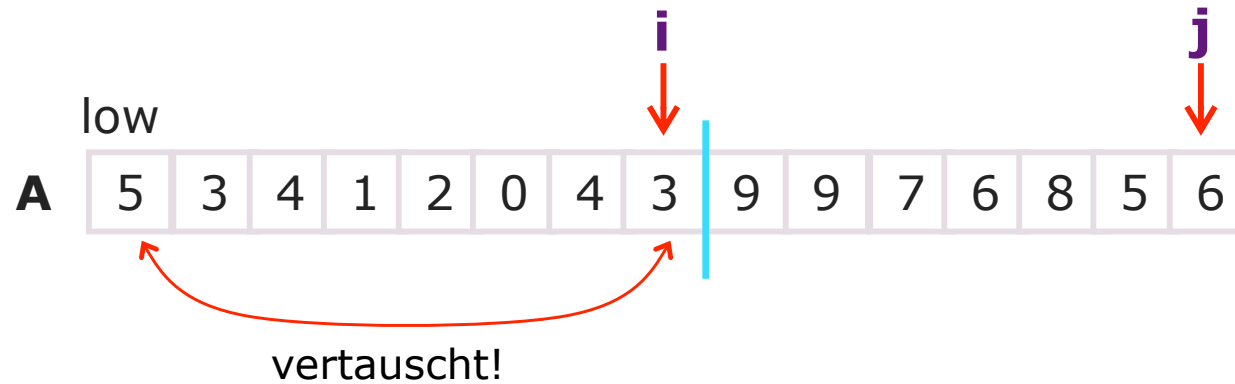
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
            A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



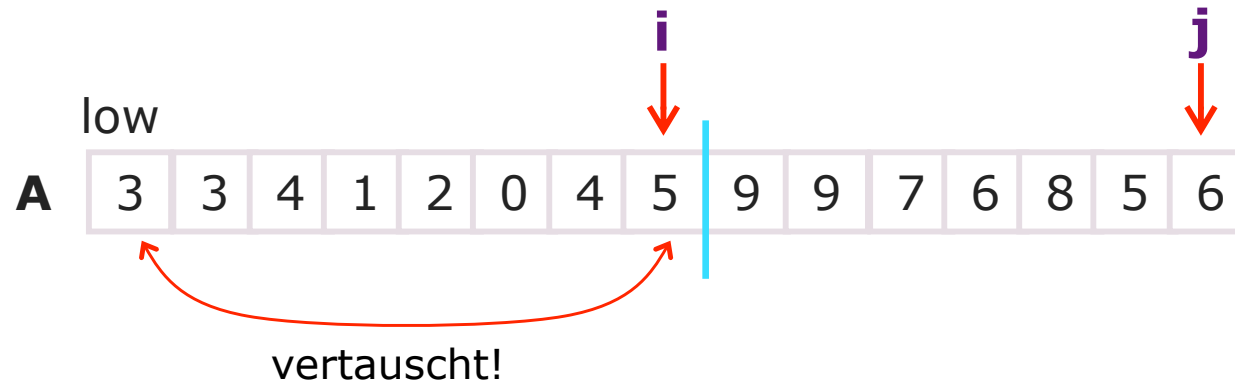
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
        A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



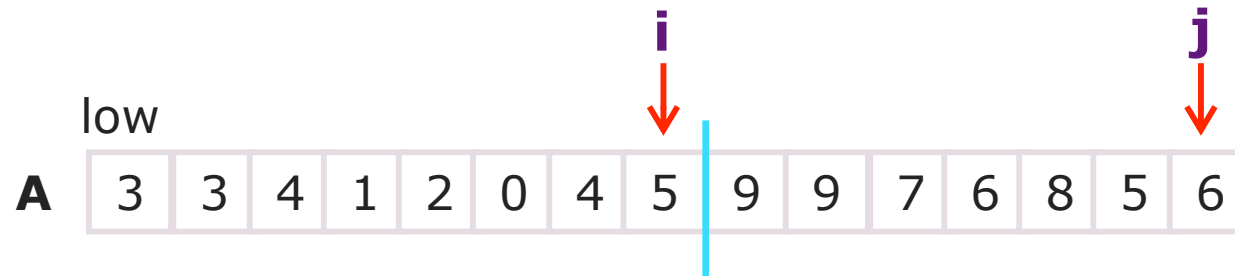
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

## Quicksort-Algorithmus



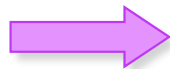
```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1, high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

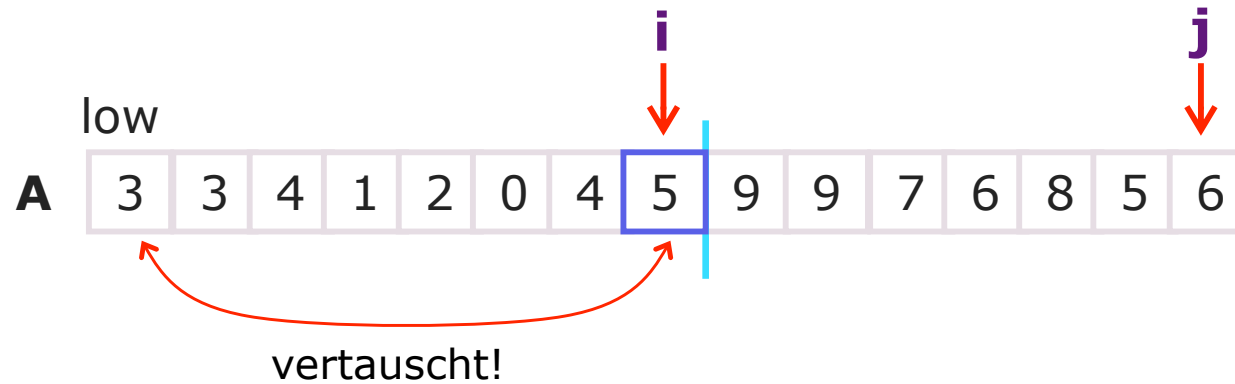
## Quicksort-Algorithmus



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
        A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus



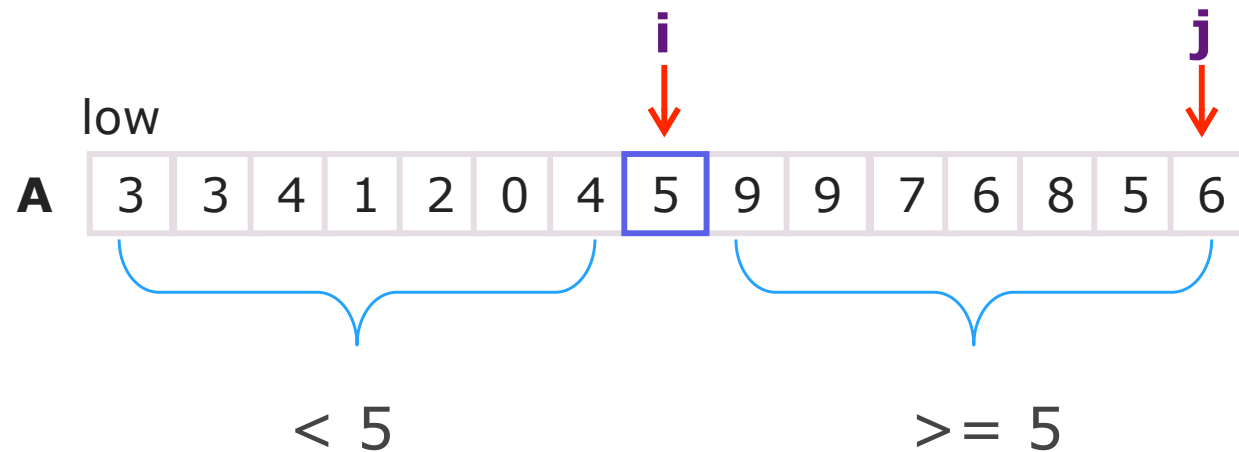


```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort-Algorithmus

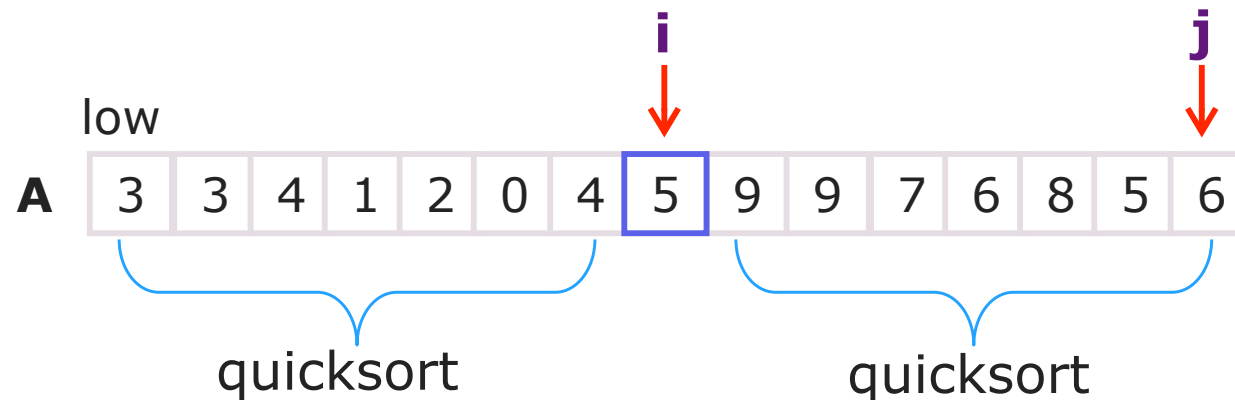


# Quicksort -Algorithmus





## Quicksort -Algorithmus



```
def quicksort (A, low, high ):  
    if low < high:  
        m = partition(A, low, high )  
        quicksort ( A, low, m-1 )  
        quicksort ( A, m+1, high )
```

## Quicksort-Algorithmus

- \* **1962** von **Hoare** entwickelt
- \* ist einer der beliebtesten Sortieralgorithmen
- \* **einfache** Implementierung
- \* **in-place**
- \* Komplexität
  - \*  **$O(n \cdot \log(n))$**  im Durchschnitt
  - \*  **$O(n^2)$**  im schlimmsten Fall
- \* **nicht stabil!**