Algorithmen und Programmieren II

Sortieralgorithmen

imperativ

Teil II

Prof. Dr. Margarita Esponda

Freie Universität Berlin

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.

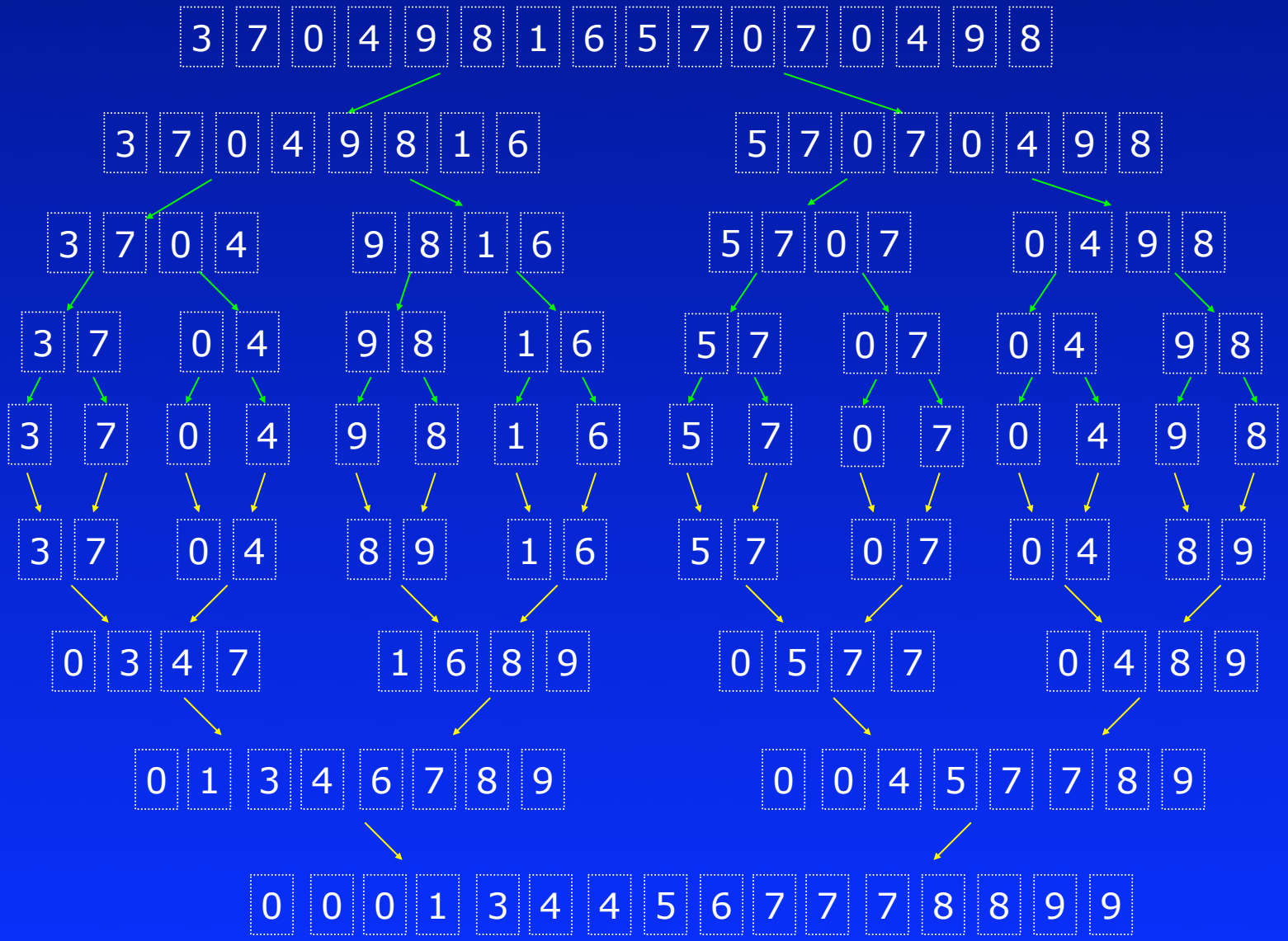
Conquer:

Löse die Teilprobleme auf dieselbe Art (rekursiv).

Merge:

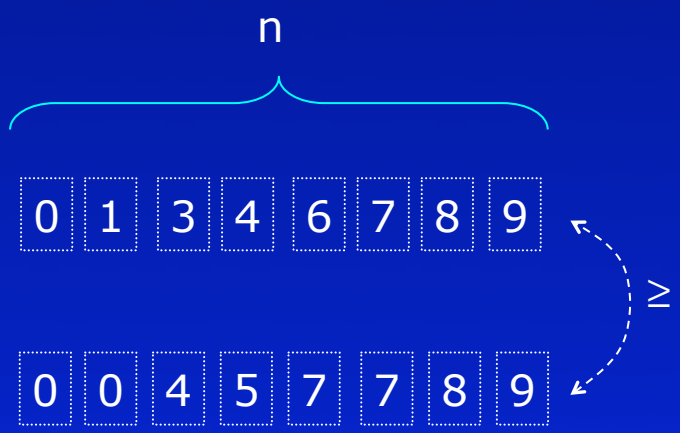
Füge die Teillösung zur Gesamtlösung zusammen.

Mergesort-Algorithmus



M. Esponda

Merge-Algorithmus



Merge-Algorithmus

Merge-Algorithmus

Merge-Algorithmus

Merge-Algorithmus

Merge-Algorithmus

Wir hatten ursprünglich zwei sortierte Mengen mit Länge **n.**

Nach jedem Vergleich wird eine Zahl sortiert,

d.h. im schlimmsten Fall haben wir **2n** Vergleiche.

$T(n) = 2n = O(n)$

Merge-Sort-Algorithmus

M. Esponda

Mergesort-Algorithmus

Eine Teilung kostet c_1

Ein Vergleich kostet c_2

$$T(n) = \underbrace{c_1(\mathbf{n-1})}_{\text{Teiloperationen}} + \underbrace{c_2\mathbf{n}\times\log(n)}_{\text{Vergleiche}}$$

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

Merge-Sort-Algorithmus

```
merge [Int] -> [Int] -> [Int]

merge [] ys = ys

merge xs [] = xs

merge (x:xs) (y:ys)  =  if x <= y
                        then  x: (merge xs (y:ys))
                        else   y: (merge (x:xs) ys)
```

Merge-Sort-Algorithmus

```
mergesortStart  [ ]  = mergesort  0  [ ]

mergesortStart  xs =  mergesort  (length xs)  xs


mergesort _ [] = []

mergesort _ [x] = [x]

mergesort _ [x,y] = if x <= y then [x,y] else [y,x]

mergesort len xs = merge(mergesort  h  (take h xs)) (mergesort (len-h) (drop h xs))
                        where
                            h = len `div` 2
```

Merge-Algorithmus

```
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A) // 2
        return merge( mergesort(A[:m]), mergesort(A[m:]) )
```

```
def merge(low, high):
        res = []
        i, j = 0, 0
        while i<len(low) and j<len(high):
            if low[i] <= high[j]:
                res.append(low[i])
                i = i+1
            else:
                res.append(high[j])
                j = j+1
        res = res + low[i:]
        res = res + high[j:]
        return res
```

Merge-Algorithmus

Eigenschaften:

- **stabiler** Algorithmus
- **1945** von **John von Neumann** entwickelt
- Komplexität **O(n·log(n))**
- das Verfahren arbeitet bei Arrays nicht in-place. Speicherverbrauch O(n)
- sehr einfache Implementierung mit verketteten Listen.

Quick-Sort-Algorithmus

Der Quicksort-Algorithmus (**1962** von Hoare entwickelt) ist einer der beliebtesten Sortieralgorithmen, weil er sehr effizient und einfach zu implementieren ist.

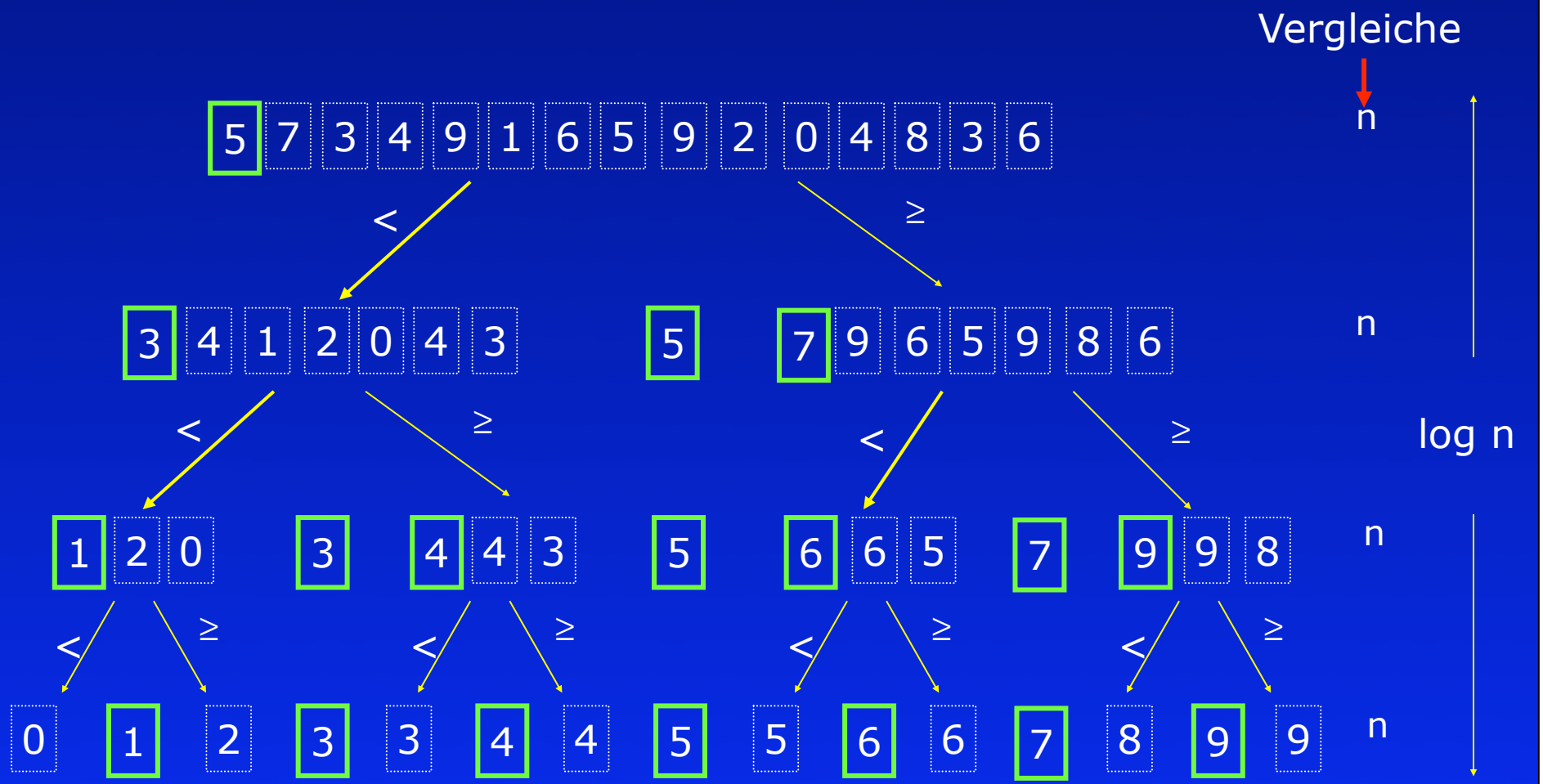
Grundidee:

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 in der Mitte beider Bereiche 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



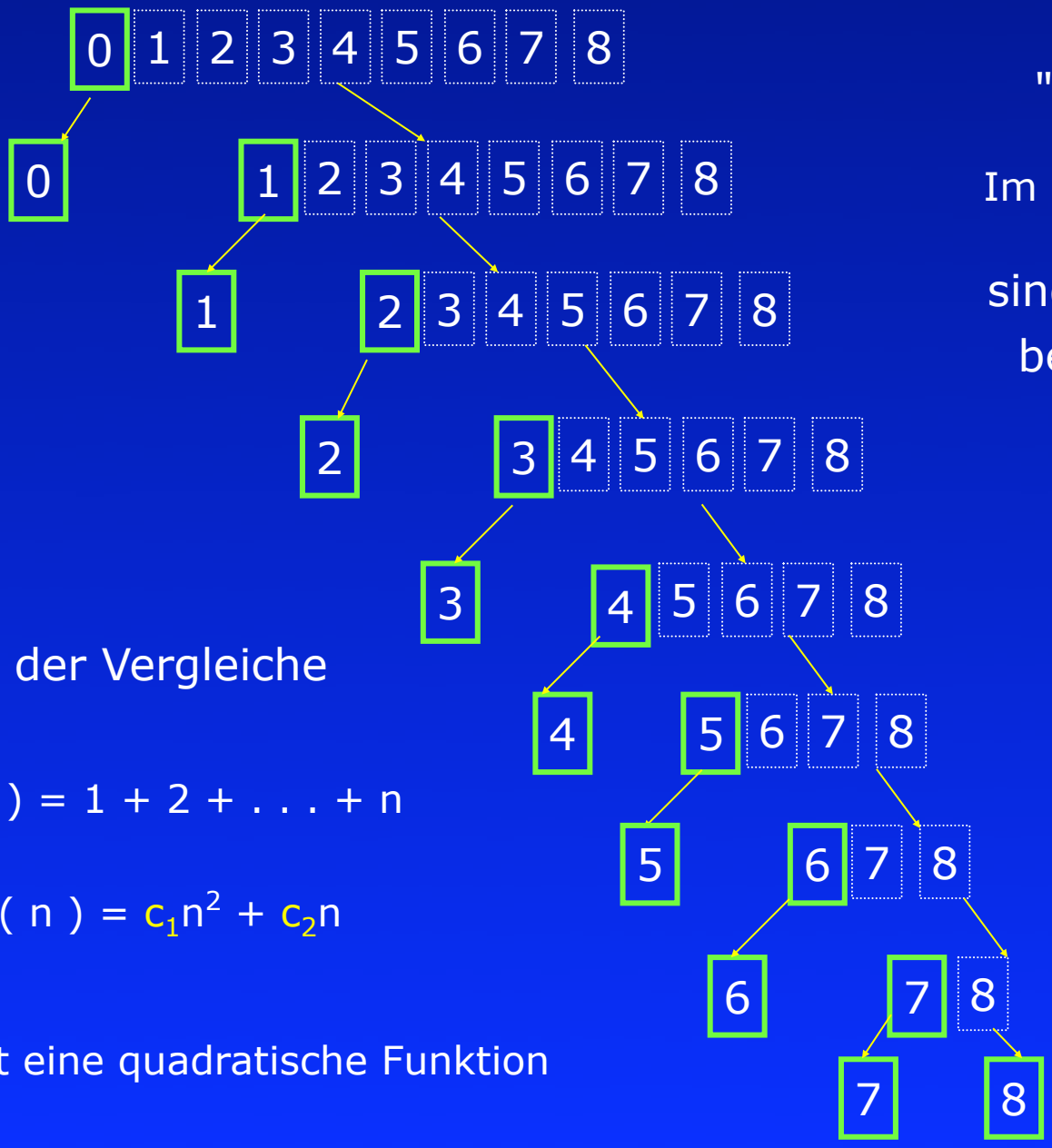
Der Quicksort-Algorithmus funktioniert am besten, wenn die Teilbereiche fast gleich groß sind.

im besten Fall!

$T(n) = n \times \log n$

M. Esponda

Quicksort-Algorithmus



"worst case"

Im schlimmsten Fall!

sind alle Elemente bereits sortiert.

O(n^2)

Anzahl der Vergleiche

⇩

$T(n) = 1 + 2 + \ldots + n$

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

T(n) ist eine quadratische Funktion

M. Esponda

Quicksort-Algorithmus
Haskell

```
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 (++).

Quicksort-Algorithmus

imperativ

Rekursive Implementierung

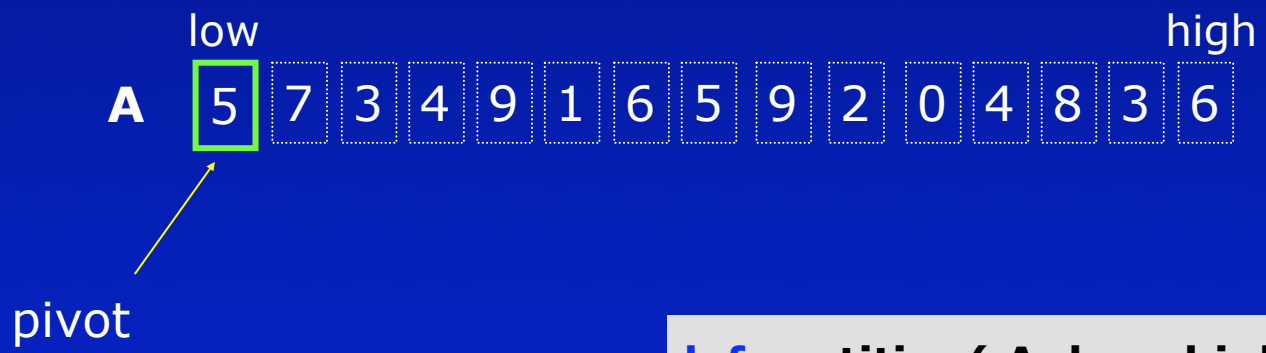
```
def quicksort (A, low, high ):
    if  low<high:
        m = partition(A, low, high )
        quicksort ( A,  low,  m-1 )
        quicksort ( A,  m+1,  high )
```

Quicksort -Algorithmus

Sortieren am Ort

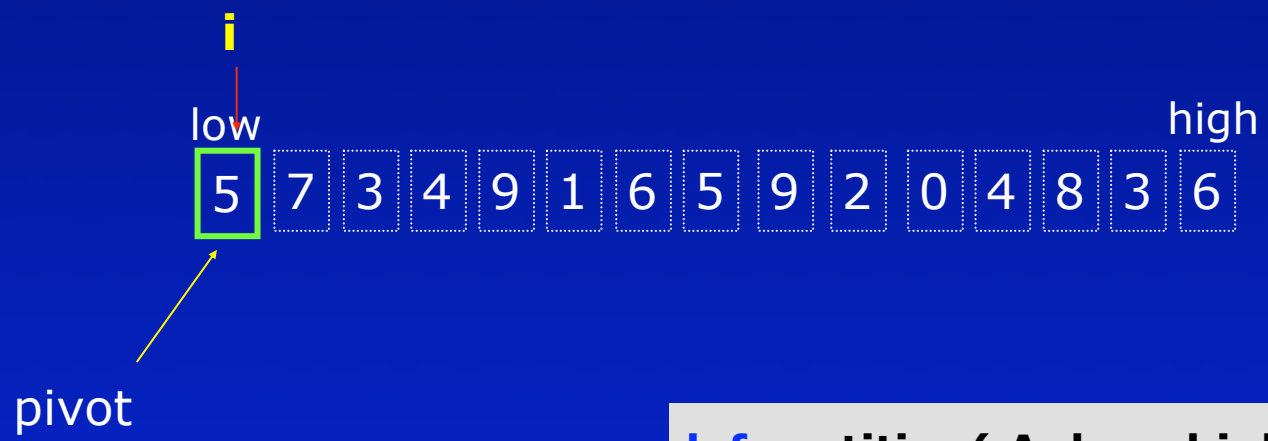

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



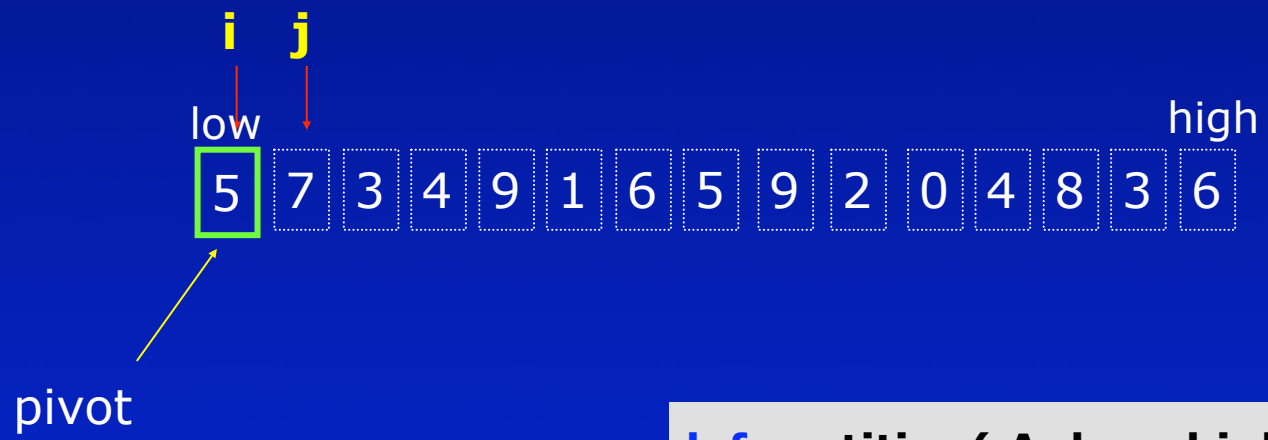

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



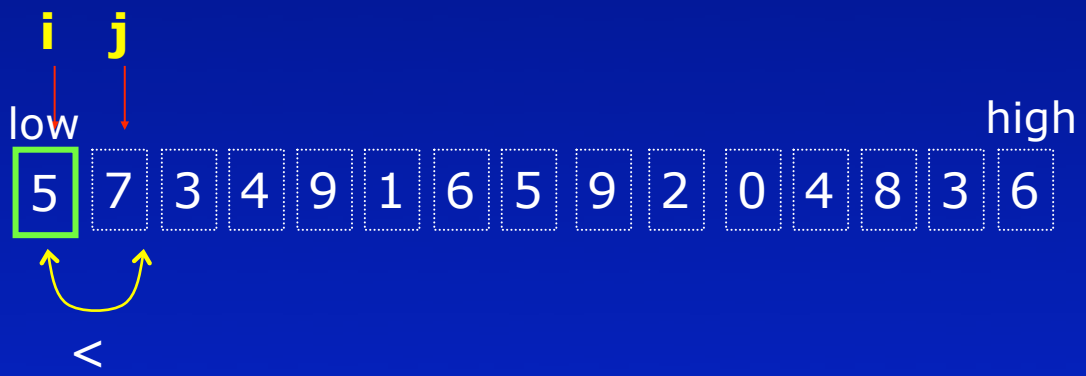

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

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

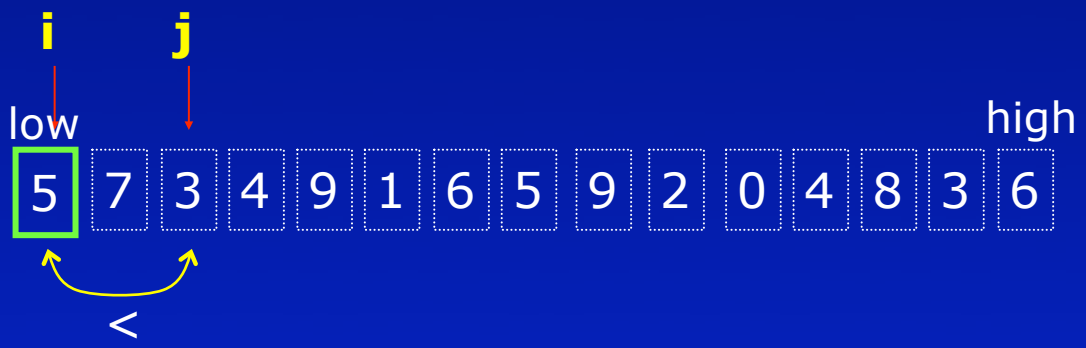



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

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

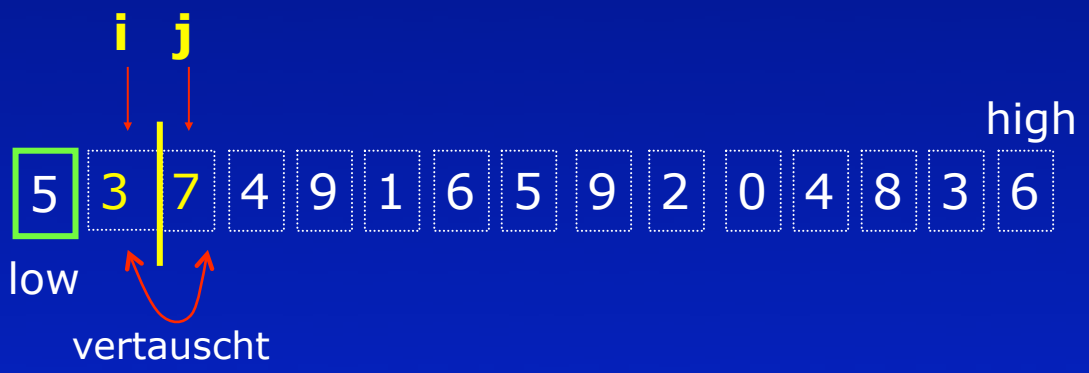



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus

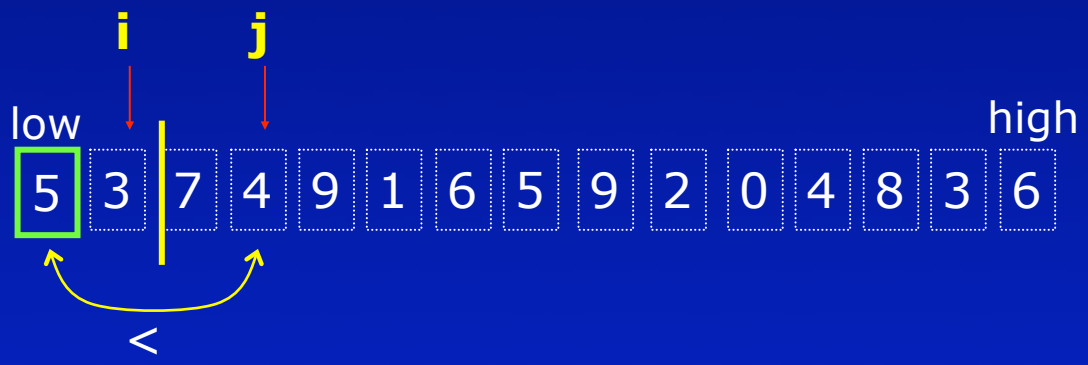

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

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
```

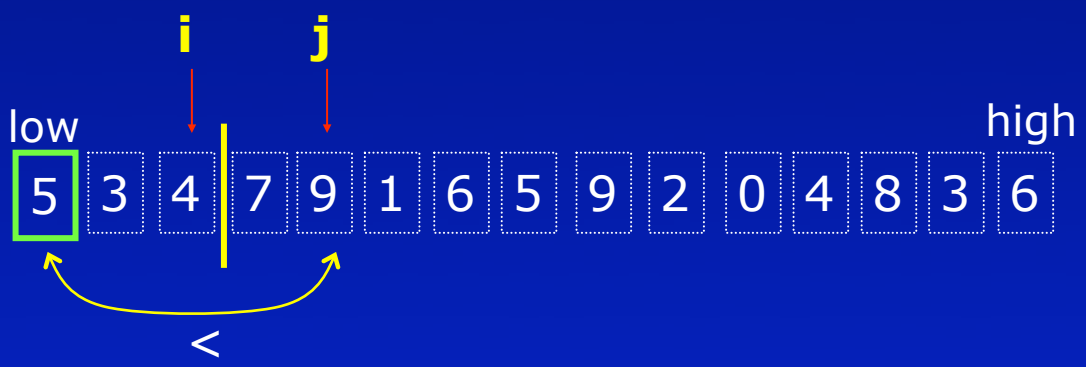
Quicksort -Algorithmus


```
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

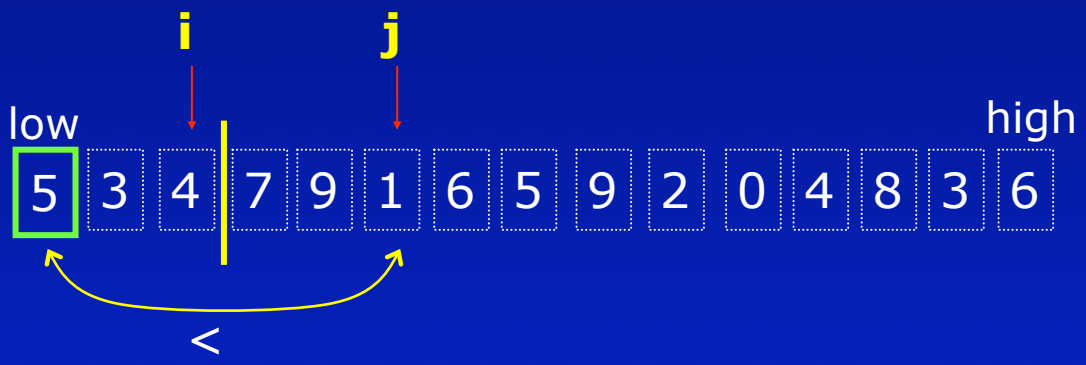

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

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
```

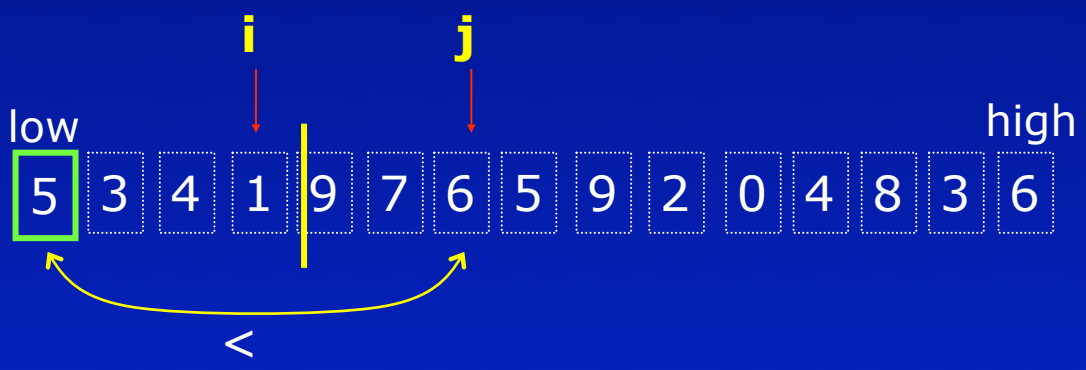
Quicksort -Algorithmus


```
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

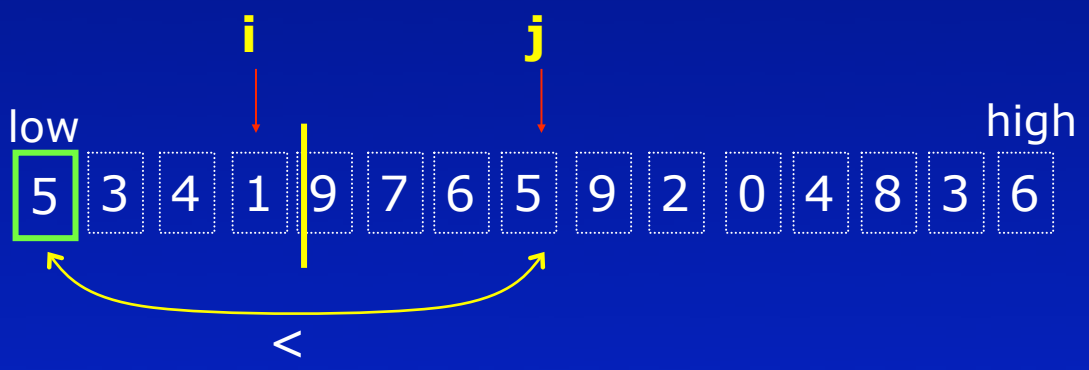

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



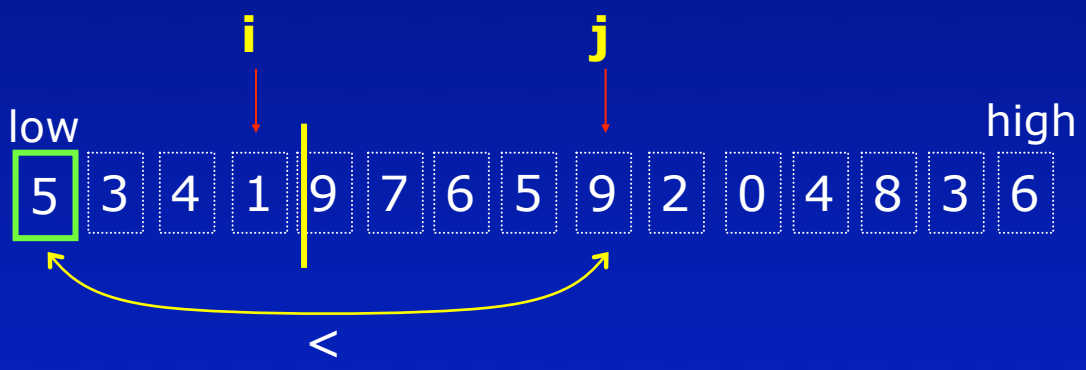

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



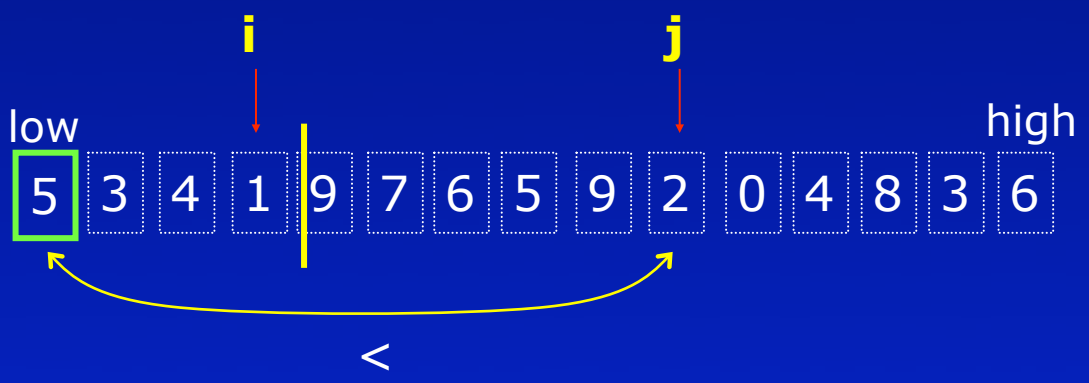

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



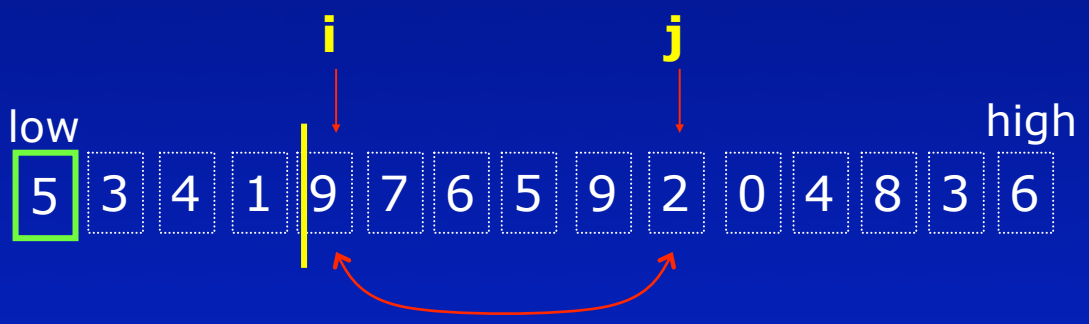

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



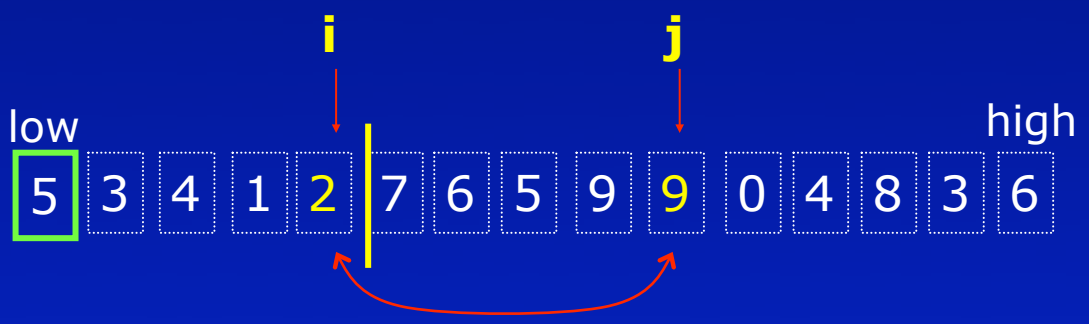

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



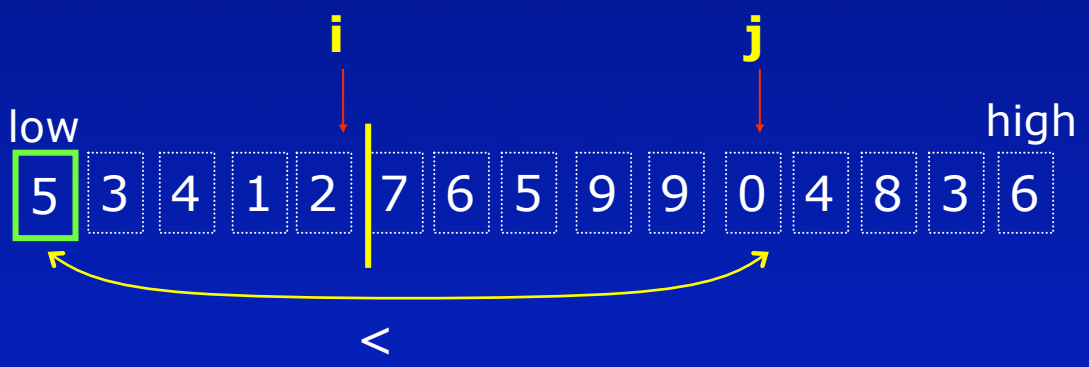

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



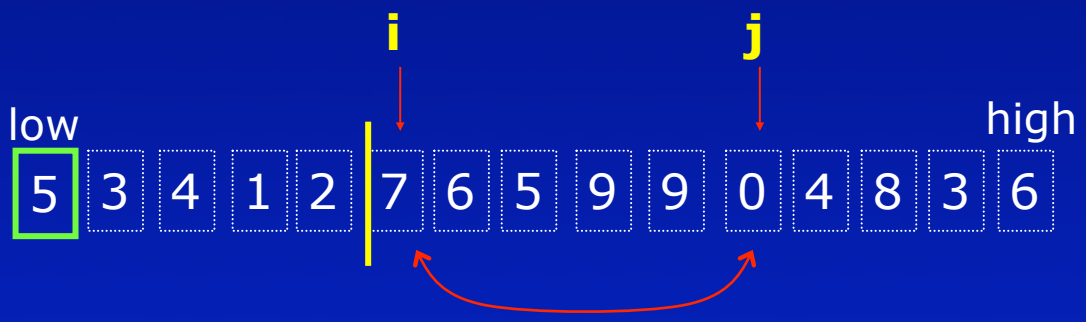

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



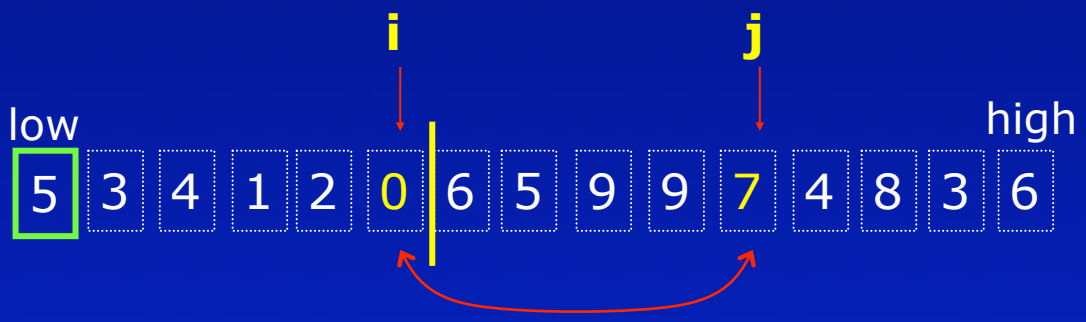

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



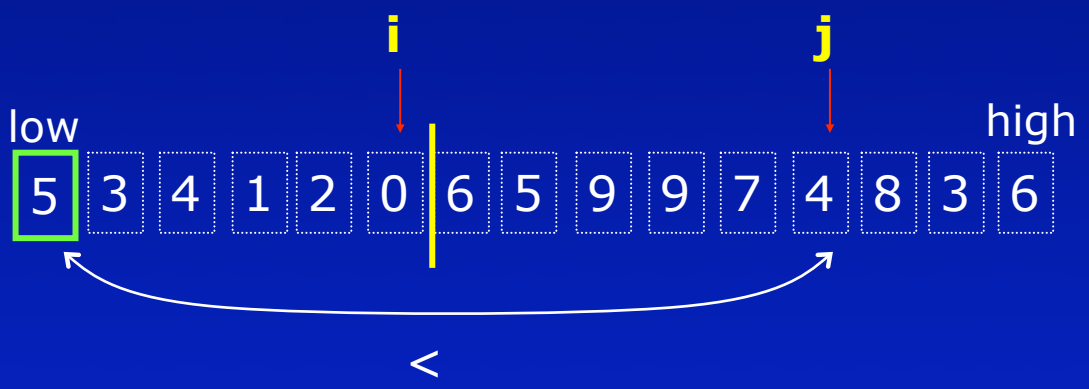

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



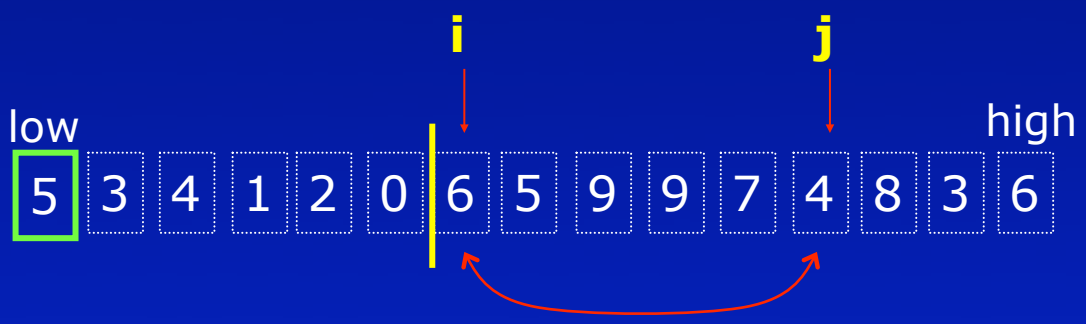

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



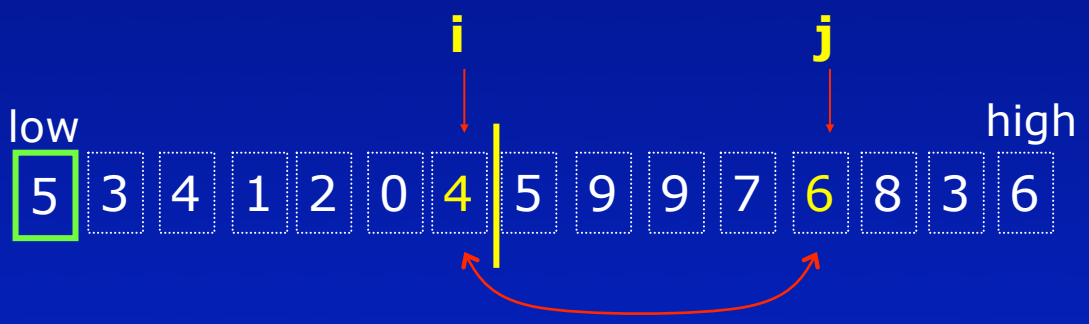

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



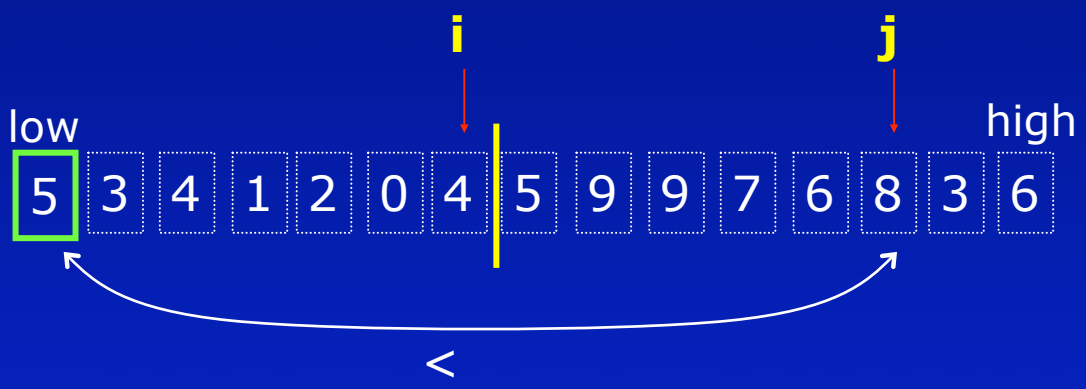

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



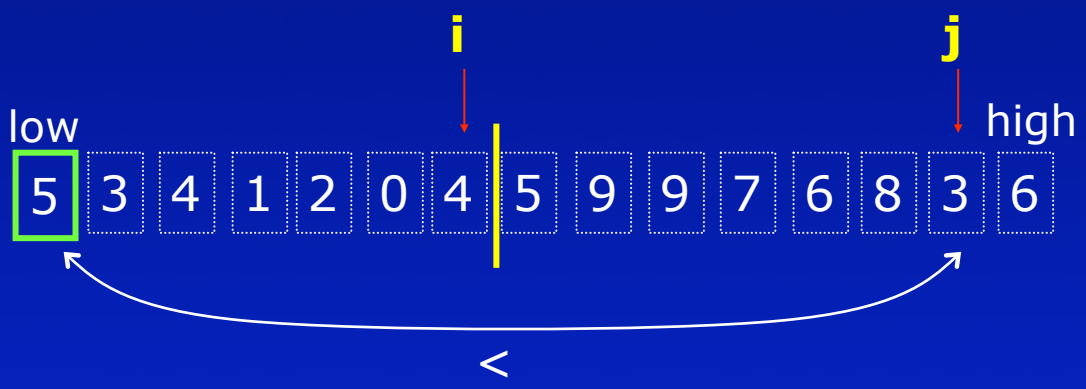

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



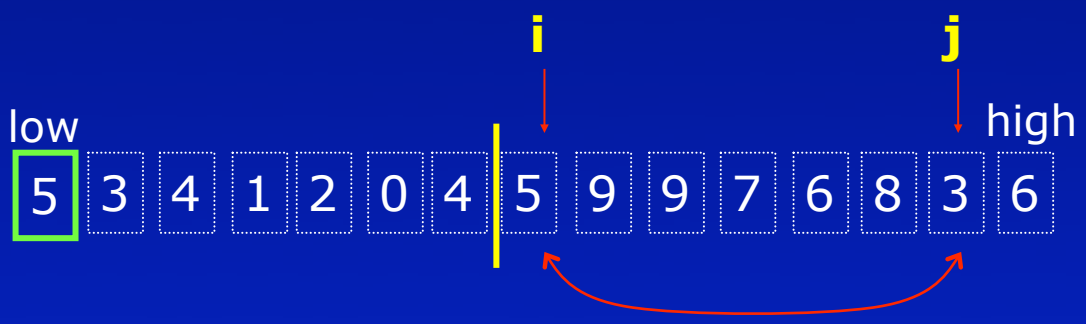

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



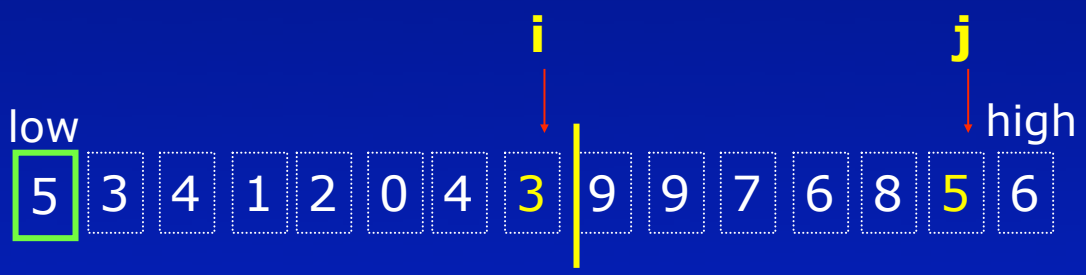

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



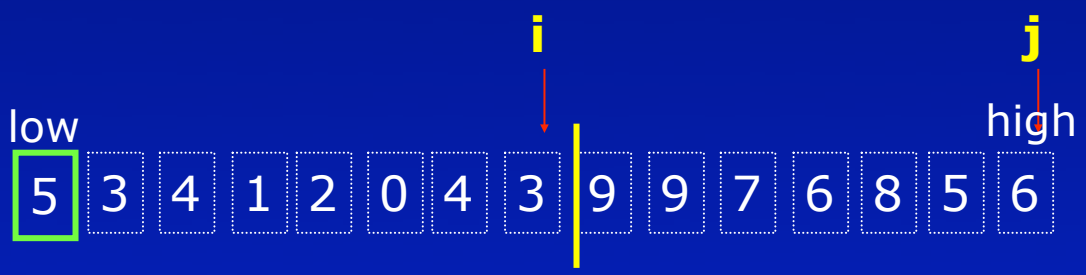

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



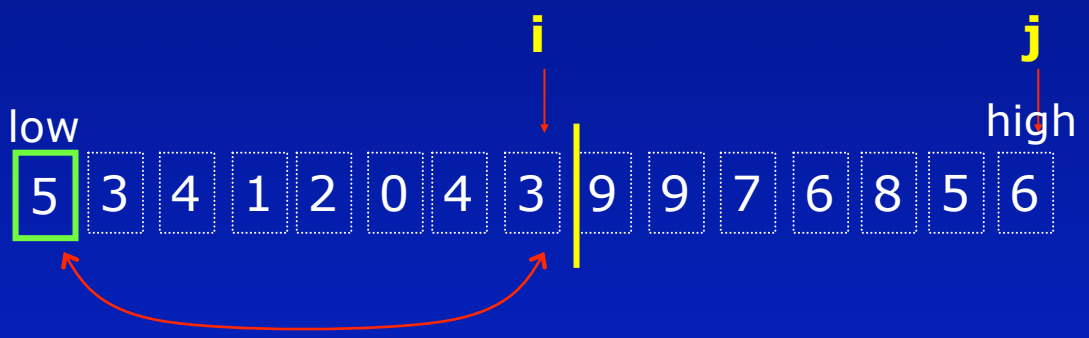

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



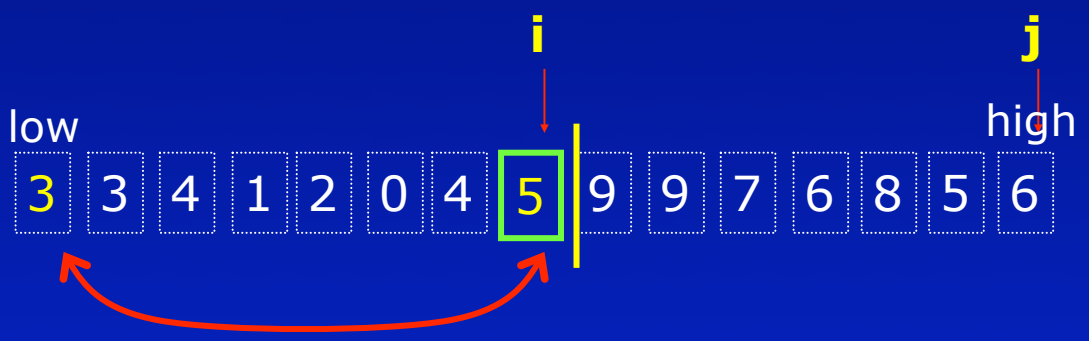

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



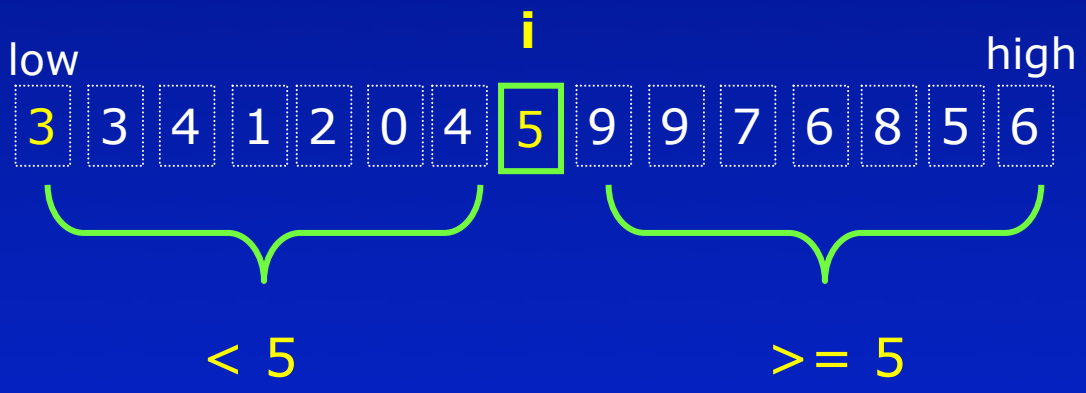

```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus

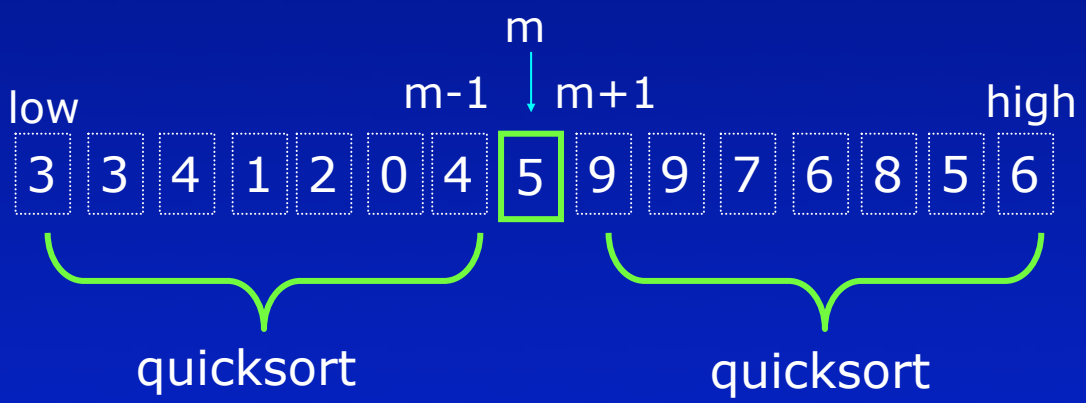



```
def partition( A, low, high ):
    pivot = A[low]
    i = low
    for j in range(low+1,high+1):
        if ( A[j] < pivot ):
            i=i+1
            A[i], A[j] = A[j], A[i]
    A[i], A[low] = A[low], A[i]
    return i
```

Quicksort -Algorithmus



Quicksort -Algorithmus



Dann wird *quicksort* zweimal rekursiv aufgerufen.

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

Sortieralgorithmen