Kapitel 10

Rekursion

Ziele

- Das Prinzip der rekursiven Berechnungsvorschrift verstehen.
- Rekursive Methoden in Java implementieren können.
- Verschiedene Formen der Rekursion kennen lernen.
- Quicksort als rekursive Methode zur Sortierung eines Arrays formulieren können und verstehen.

Rekursive Algorithmen und Methoden

- Ein Algorithmus ist **rekursiv**, wenn in seiner (endlichen) Beschreibung derselbe Algorithmus wieder aufgerufen wird. Der Algorithmus ist dann selbstbezüglich definiert.
- Rekursive Algorithmen können in Java durch **rekursive Methoden** implementiert werden.
- Eine Methode ist rekursiv, wenn in ihrem Rumpf (Anweisungsteil) die Methode selbst wieder aufgerufen wird. (mit anderen Argumenten)

Die Fakultätsfunktion

- Rekursive Definition der Fakultät:

  `0! = 1`

  `n! = n * (n-1)!` für alle natürlichen Zahlen `n ≥ 1`

- Rekursive Methode:

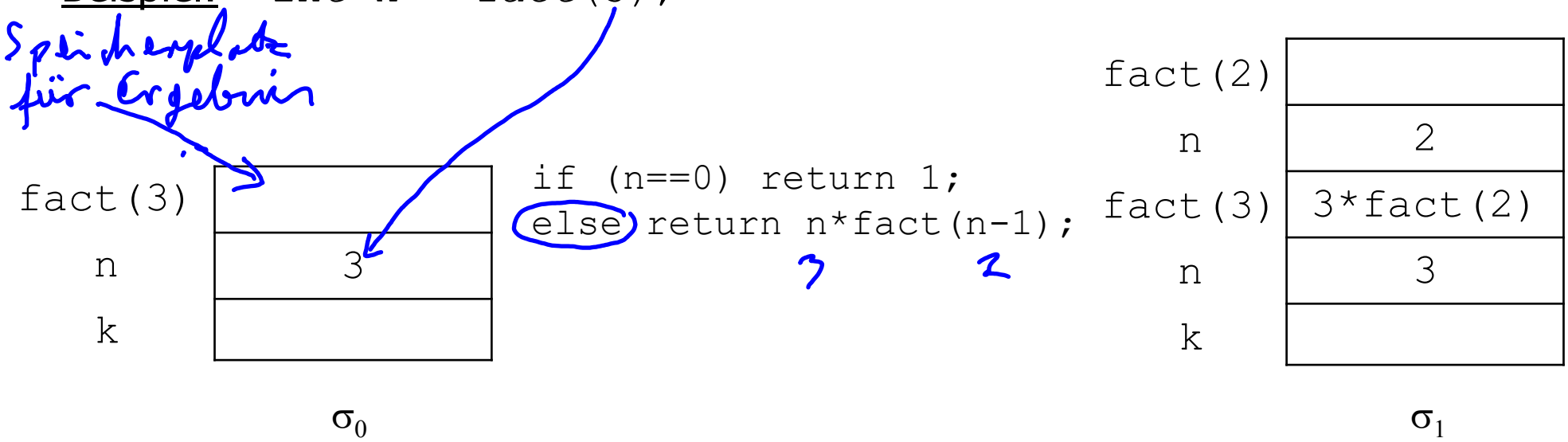
```
public static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

rekursiver Aufruf!

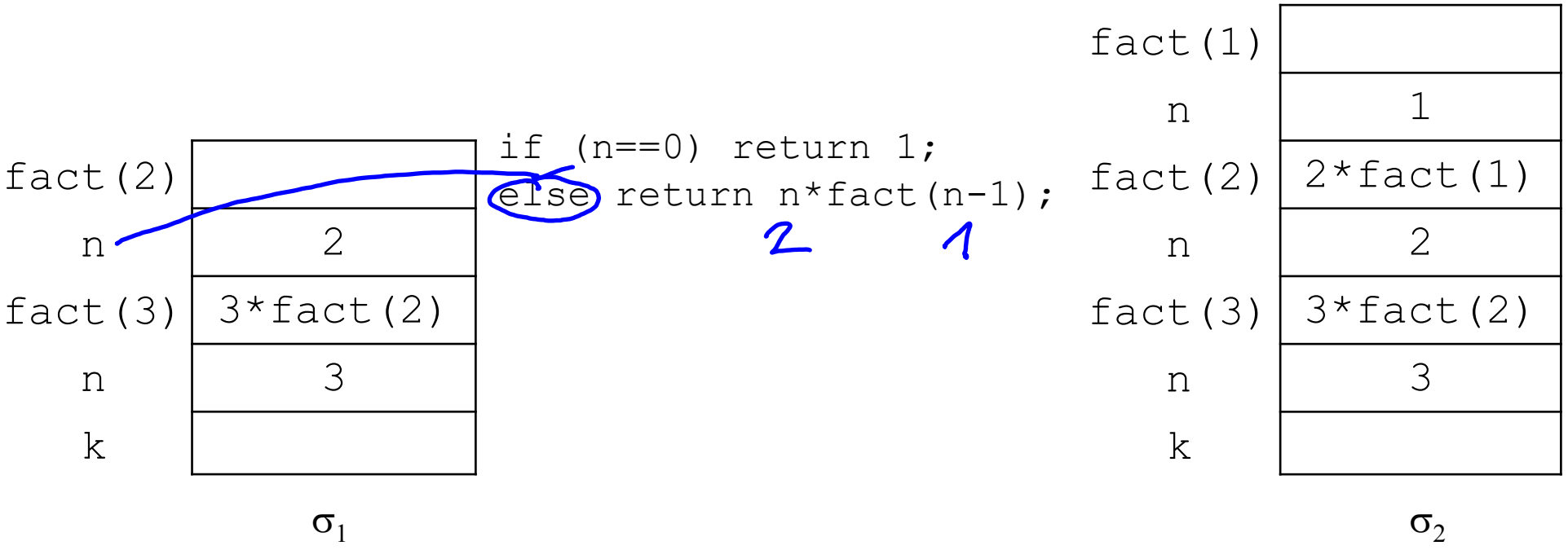
Auswertung rekursiver Methodenaufrufe

Bei der Auswertung wird ein Stack für die Zwischenergebnisse der geschachtelten Methodenaufrufe aufgebaut, der am Ende gemäß des Rekursionsschemas rückwärts abgearbeitet wird.

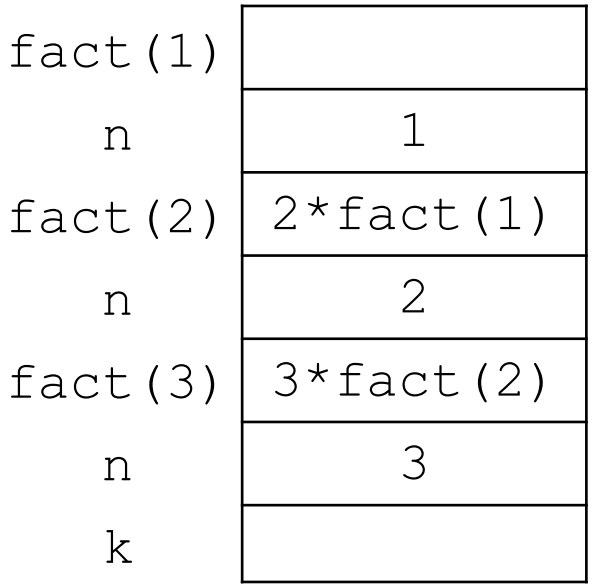
Beispiel: `int k = fact(3);`



Aufbau des Stacks zur Berechnung von `fact(2)`



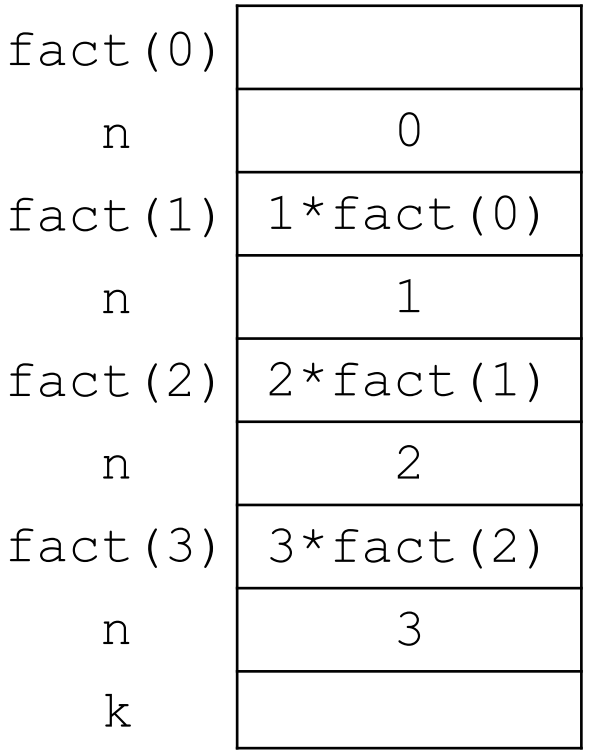
Aufbau des Stacks zur Berechnung von `fact(1)`



σ_2

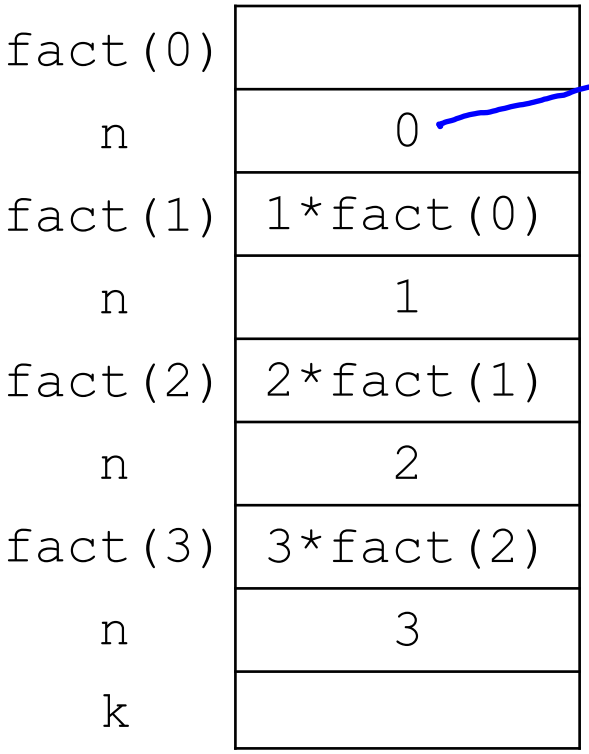
```
if (n==0) return 1;
else return n*fact(n-1);
```

1 0



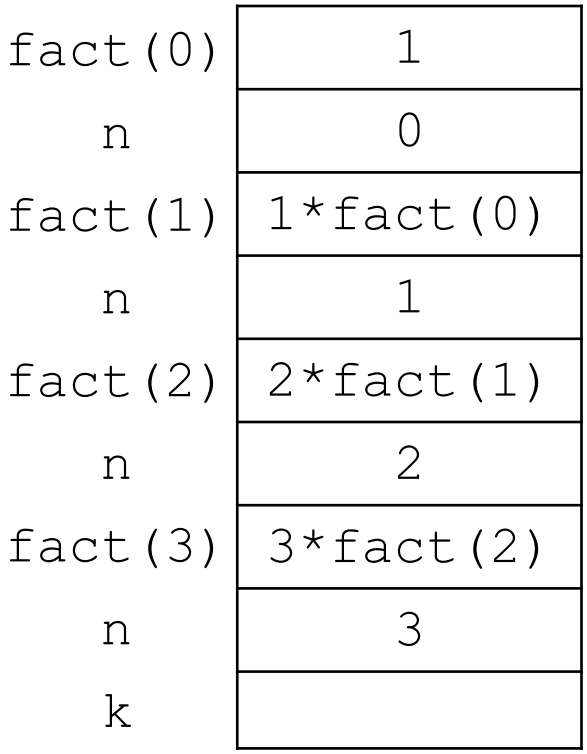
σ_3

Berechnung von `fact(0)`



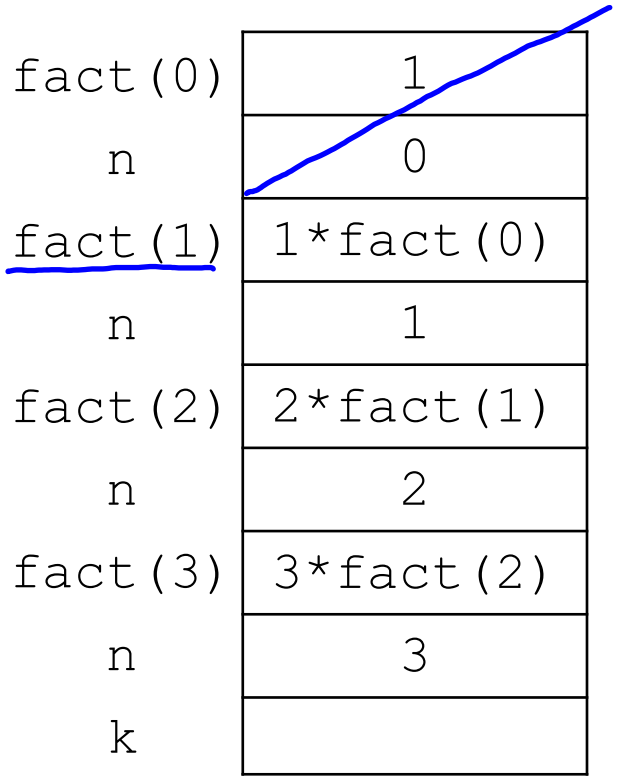
σ_3

```
if (n==0) return 1;
else return n*fact(n-1);
```



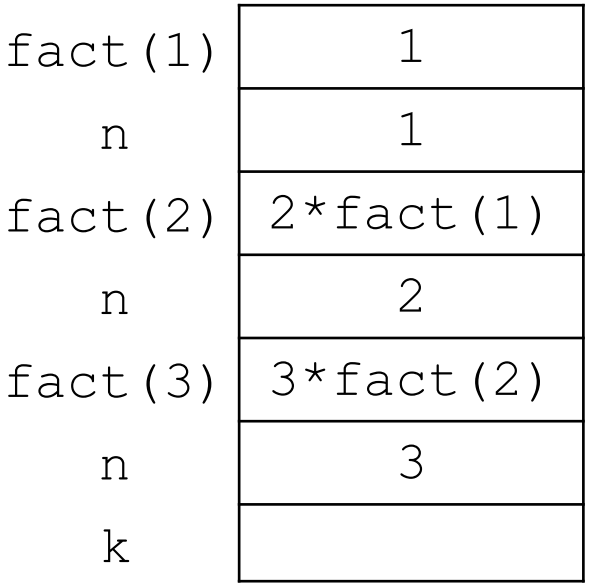
σ_4

Berechnung von `fact(1)` und Abbau des Stacks



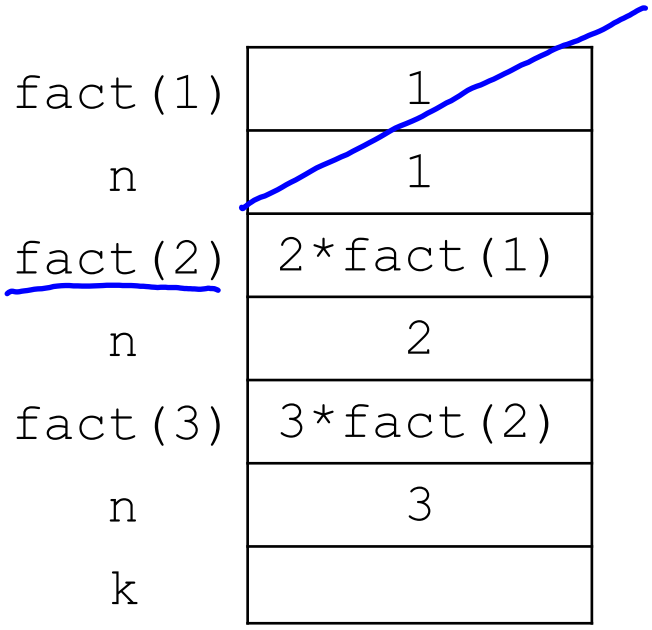
σ_4

`fact(1)=1*fact(0);`



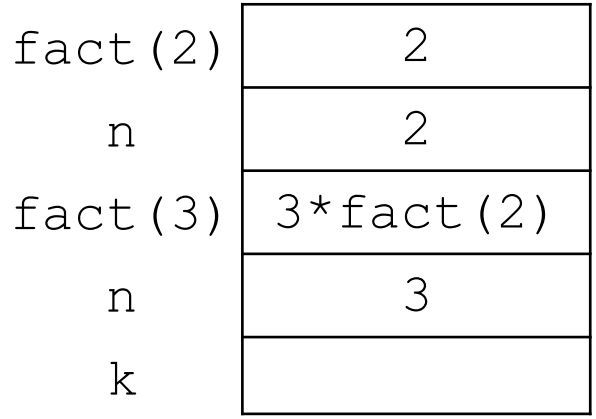
σ_5

Berechnung von `fact(2)` und Abbau des Stacks



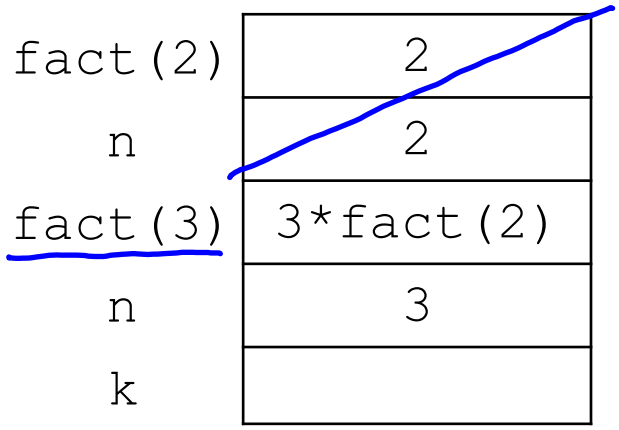
σ_5

`fact(2)=2*fact(1);`



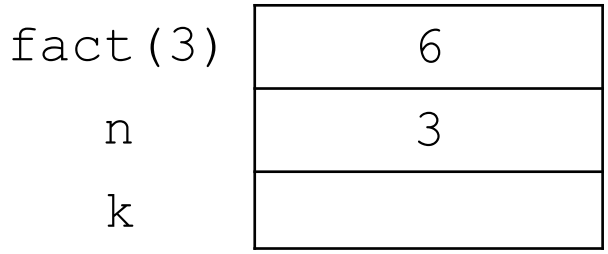
σ_6

Berechnung von `fact(3)`, Abbau des Stacks und Zuweisung des Ergebnisses

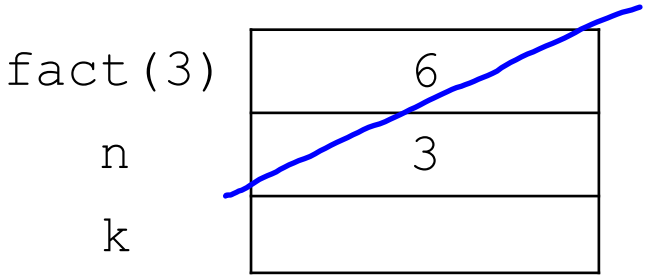


σ_6

`fact(3)=3*fact(2);`



σ_7



σ_7

`k = fact(3);`

σ_8

Terminierung

Der Aufruf einer rekursiven Methode **terminiert**, wenn nach endlich vielen rekursiven Aufrufen ein Abbruchfall erreicht wird.

Beispiel:

- Für alle natürlichen Zahlen `n ≥ 0` terminiert der Methodenaufruf `fact(n)`.
- Für alle negativen ganzen Zahlen `n < 0` terminiert der Methodenaufruf `fact(n)` nicht.

z.B. fact(-1) = (-1) * fact(-2) =

= (-1) * ((-2) * fact(-3))

= . . . . .

Rekursion und Iteration (1)

Zu jedem rekursiven Algorithmus gibt es einen semantisch äquivalenten iterativen Algorithmus, d.h. einen Algorithmus mit Wiederholungs-anweisungen, der dasselbe Problem löst.

Beispiel:

```
static int factIterativ(int n) {
   int result = 1;
   while (n != 0) {
      result = result * n;
      n--;
   }
   return result;
}
```

// Akkumulator

fact Iterativ (3) =

((1 * 3) * 2) * 1)

3

6

6

Rekursion und Iteration (2)

- Rekursive Algorithmen sind häufig eleganter und übersichtlicher als iterative Lösungen.
- Gute Compiler können aus rekursiven Programmen auch effizienten Code erzeugen; trotzdem sind iterative Programme meist schneller als rekursive.
- Für manche Problemstellungen kann es wesentlich einfacher sein einen rekursiven Algorithmus anzugeben als einen iterativen.
  (z.B. „Türme von Hanoi"; vgl. Übungen)

Fibonacci-Zahlen: rekursive Definition und Methode

- Rekursive Definition der Fibonacci-Zahlen:

  `fib(0) = 1,  fib(1) = 1,`

  `fib(n) = fib(n-2) + fib(n-1)` für alle natürlichen Zahlen `n ≥ 2`

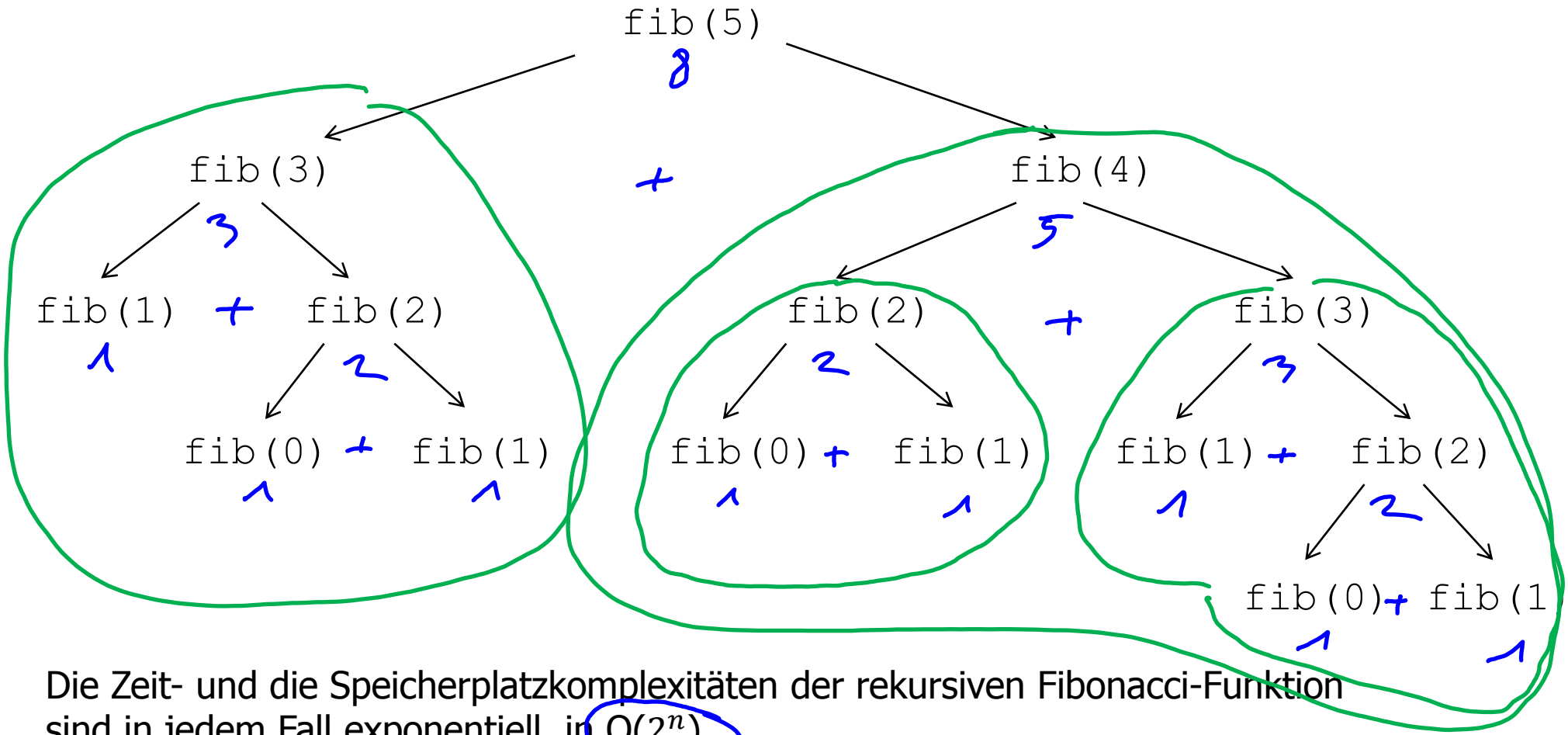
- Rekursive Methode:

```
static int fib(int n) {
   if (n <= 1) return 1;
   else return fib(n-2) + fib(n-1);
}
```

Deutung: fib(n) = Anzahl der neu geborenen Kaninchenpaare im Jahr n.

Annahmen:
- Im Jahr 0 wird ein Paar geboren
- Im Jahr 1 hat dieses Paar ein neues Paar geboren
- In jedem Jahr n ≥ 2 haben die ein- und zweijährigen Paare je ein neues Paar geboren

Kaskade rekursiver Aufrufe



Die Zeit- und die Speicherplatzkomplexitäten der rekursiven Fibonacci-Funktion sind in jedem Fall exponentiell, in $O(2^n)$.

Fibonacci-Zahlen: Iterative Methode

```
static int fibIterativ(int n) {
    int f0 = 1;
    int f1 = 1;
    int f = 1;
    for (int i = 2; i <=  n; i++) {
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}
```

immer 6 Speicherplätze nötig.

Die Zeitkomplexität der iterativen Methode ist linear, d.h. in O(n).

Die Speicherplatzkomplexität der iterativen Methode ist konstant, d.h. in O(1).

Formen der Rekursion

- *Lineare Rekursion:*
  In jedem Zweig (der Fallunterscheidung) kommt höchstens ein rekursiver Aufruf vor, z.B. Fakultätsfunktion `fact`.
- *Kaskadenartige Rekursion:*
  Mehrere rekursive Aufrufe stehen nebeneinander und sind durch Operationen verknüpft, z.B. Fibonacci-Zahlen `fib`.
- *Verschachtelte Rekursion:*
  Rekursive Aufrufe kommen in Parametern von rekursiven Aufrufen vor, z.B. Ackermann-Funktion.

Die Ackermann-Funktion

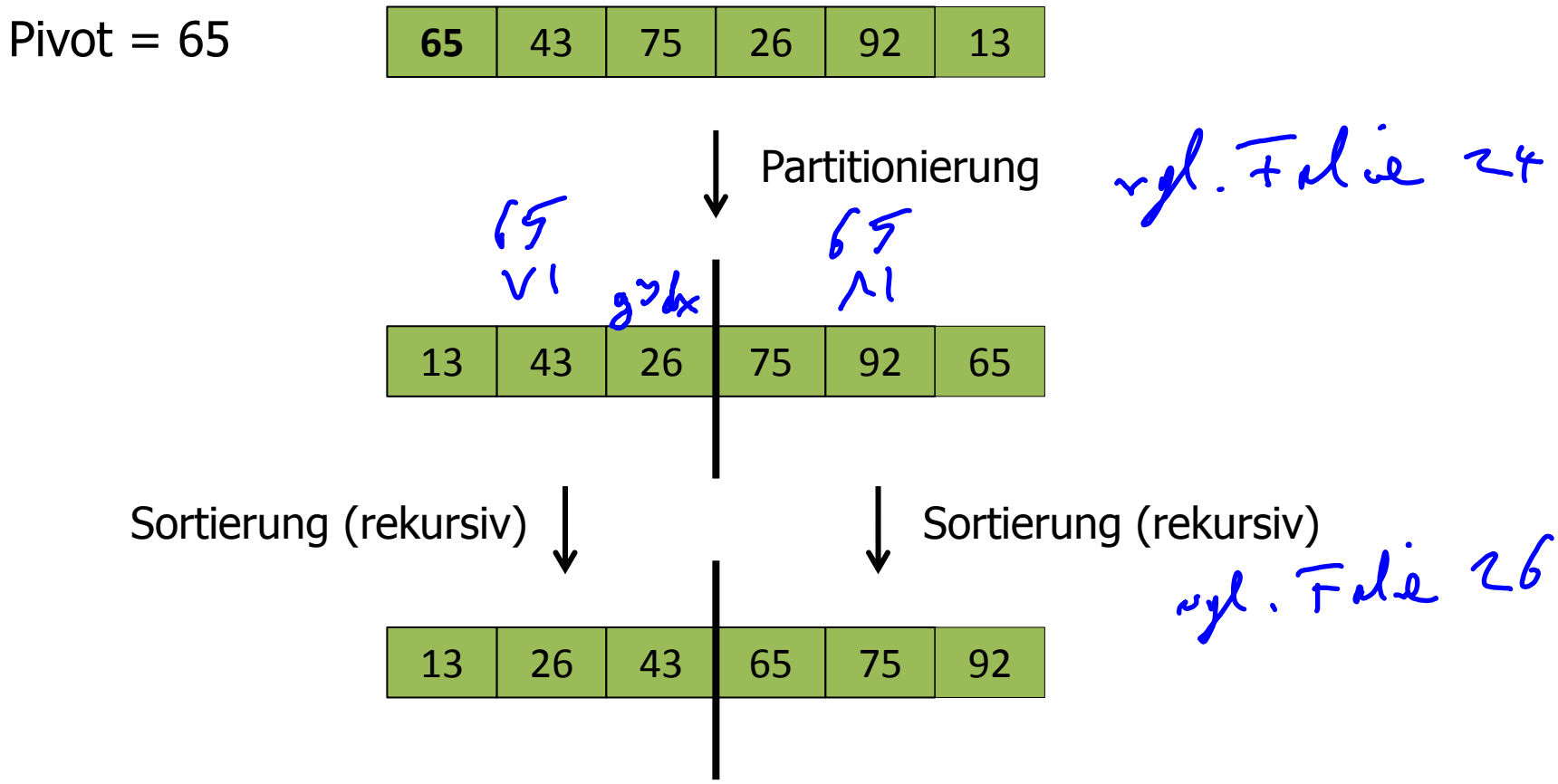
```
static int ack(int n, int m) {
    if (n == 0) return m + 1;
    else if (m == 0) return ack(n - 1, 1);
    else return ack(n - 1, ack(n, m - 1));
}
```

- Die Ackermann-Funktion ist eine Funktion mit exponentieller Zeitkomplexität, die extrem schnell wächst.
- Sie ist das klassische Beispiel für eine berechenbare, terminierende Funktion, die nicht primitiv-rekursiv ist (erfunden 1926 von Ackermann).
- Beispiele:
  `ack(4,0)` = 13
  `ack(4,1)` = 65533
  `ack(4,2)` = $2^{65536}-3$ (eine Zahl mit 19729 Dezimalstellen).
  `ack(4,4)` > Anzahl der Atome im Universum

Quicksort

- Einer der schnellsten Sortieralgorithmen (von C.A.R. Hoare, 1960).
- **Idee**: Falls das zu sortierende Array mindestens zwei Elemente hat:
  1. Wähle irgendein Element aus dem Array als Pivot („Dreh- und Angelpunkt"), z.B. das erste Element.
  2. Partitioniere das Array in einen linken und einen rechten Teil, so dass
     - alle Elemente im linken Teil kleiner-gleich dem Pivot sind und
     - alle Elemente im rechten Teil größer-gleich dem Pivot sind.
  3. Wende das Verfahren rekursiv auf die beiden Teilarrays an.
- Der Quicksort-Algorithmus folgt einem ähnlichen Lösungsansatz wie die binäre Suche. Diesen Lösungsansatz nennt man „Divide-and-Conquer" („Teile und herrsche").

Quicksort: Beispiel



Quicksort in Java

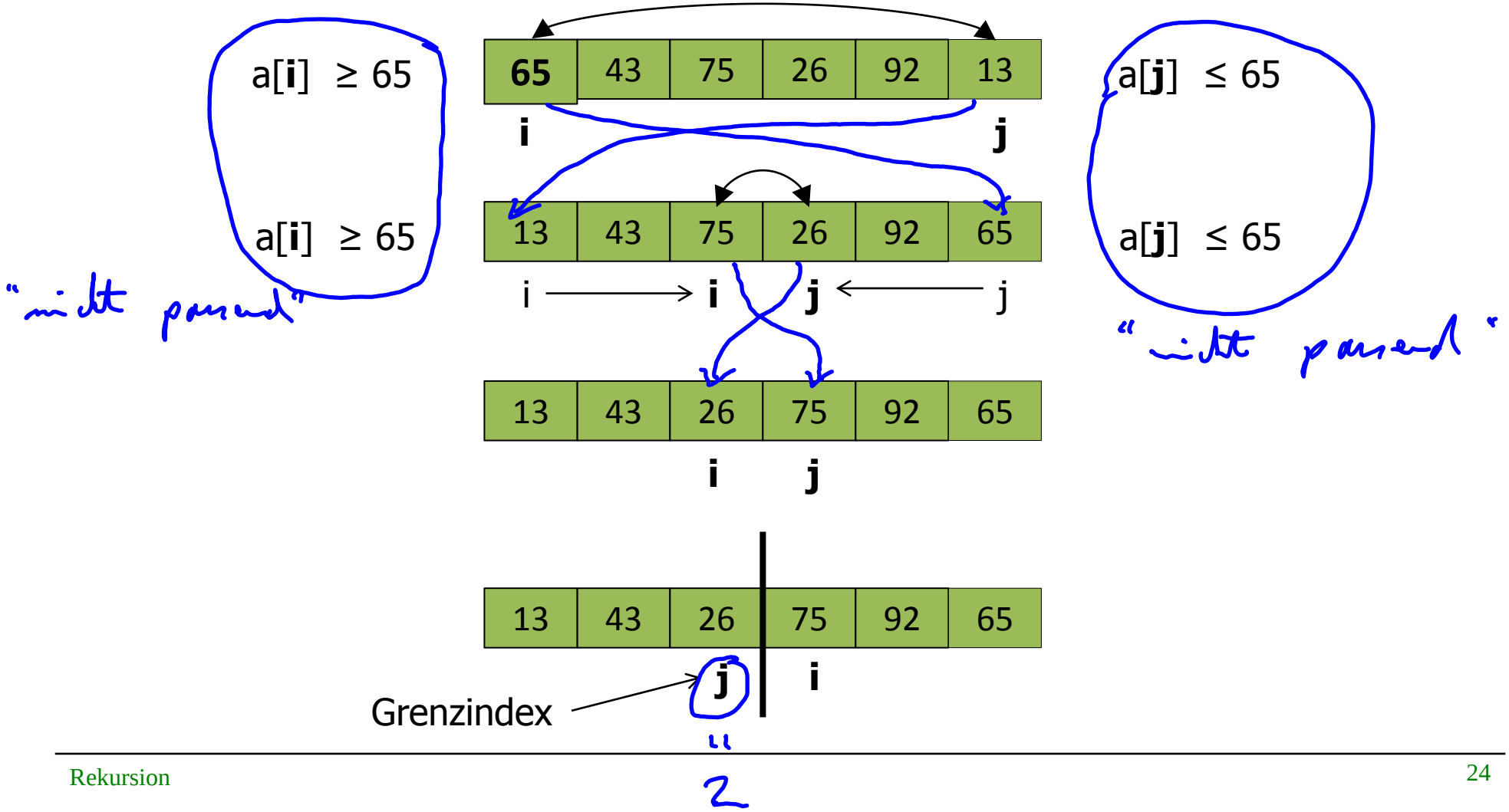
```
static void quicksort(double[] a) {
   qsort(a, 0, a.length - 1);
}
  // Sortiert den Teilbereich a[from]...a[to] von a.
static void qsort(double[] a, int from, int to) {
   if (from < to) {  \\mehr als ein Element zu sortieren
    double pivot = a[from]; //waehle erstes Element als Pivot
        //Partitionierung und Rückgabe des Grenzindex
    int gIdx = partition(a, from, to, pivot);
       //rekursiver Aufruf für den linken Teilarray
    qsort(a, from, gIdx);
        //rekursiver Aufruf für den rechten Teilarray
     qsort(a, gIdx + 1, to);
    }
}
```

Partitionierung: Vorgehensweise

- Laufe von der unteren und der oberen Arraygrenze mit Indizes `i` und `j` nach innen und vertausche „nicht passende“ Elemente `a[i]` und `a[j]` bis sich die Indizes treffen oder überkreuzt haben.
- Der zuletzt erreichte Index `j` wird als Grenzindex der Partitionierung zurückgegeben.
- Von unten kommend sind Elemente "nicht passend," wenn sie größer-gleich dem Pivot sind.
- Von oben kommend sind Elemente "nicht passend," wenn sie kleiner-gleich dem Pivot sind.
- *Bemerkung:*
  Gegebenenfalls werden auch gleiche Elemente vertauscht. Dies ist aus technischen Gründen nötig, damit der Index `j` so stoppt, dass der letzte Wert von `j` immer der richtige Grenzindex ist.

Partitionierung: Beispiel

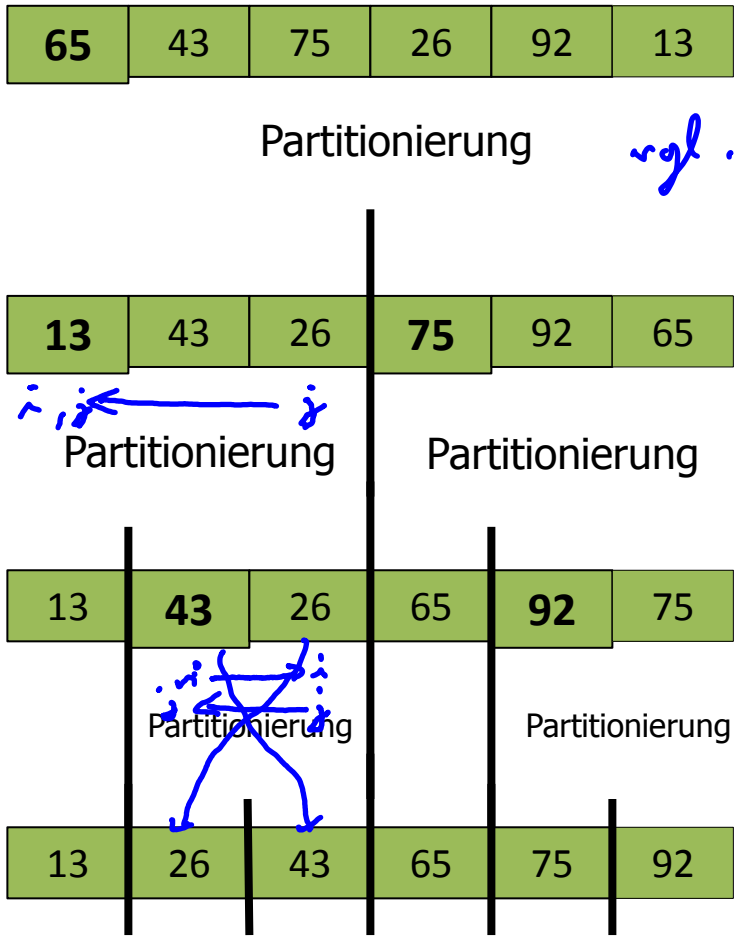
Pivot = 65



Partitionierung in Java

```
static int partition(double[] a, int from, int to, double pivot) {
   int i = from - 1;
   int j = to + 1;
   while (i < j) {
      i++; //naechste Startposition von links
        //von links nach innen laufen solange Elemente kleiner als Pivot
      while  (a[i] < pivot) i++;
      j--; //naechste Startposition von rechts
        //von rechts nach innen laufen solange Elemente größer als Pivot
      while  (pivot < a[j]) j--;
      if (i < j) { //vertausche a[i] und a[j]
         double temp = a[i]; a[i] = a[j]; a[j] = temp;
      }
   } //Ende while
   return j; //Rückgabe des Grenzindex
}
```

Partitionierungshierarchie des Quicksort



Zeitkomplexität von Quicksort (1)

- Beispiel: Das Array von oben hat die Länge 6.
  - Die Hierarchie der Partitionierungen stellt einen Baum dar mit 3 Etagen, wobei $3 = \log_2(6) + 1$.
  - Alle Partitionierungen einer Etage benötigen zusammen maximal c * 6 Schritte (mit einer Konstanten c).
  - Folglich ist die Zeitkomplexität in diesem Fall durch $6 * \log_2(6)$ beschränkt.
- Allgemein:
  - Wenn ein Array der Länge n immer wieder in zwei etwa gleich große Teile aufgeteilt wird, dann ist die Anzahl der Partitionierungs-Etagen durch $\log_2(n)$ beschränkt.
  - Die Anzahl der Schritte pro Etage ist durch n beschränkt und damit die gesamte Zeitkomplexität in diesem Fall durch $n * \log_2(n)$.
  - Man kann zeigen, dass die Zeitkomplexität des Quicksort **im durchschnittlichen Fall** von der Ordnung $n * \log_2(n)$ ist.
  -

$O(n) \subseteq O(n \cdot \log(n)) \subseteq O(n^2)$

Zeitkomplexität des Quicksort (2)

Im **schlechtesten Fall** ist die Zeitkomplexität des Quicksort quadratisch, d.h. von der Ordnung n^2. Dieser Fall tritt z.B. ein, wenn das Array schon sortiert ist.