9. Rekursion

Was heißt "rekursiv"

Eine Methode *m*() heißt rekursiv, wenn sie sich selbst aufruft

m() Þ m()	direkt rekursiv
m() Þ n() Þ m()	indirekt rekursiv

Beispiel: Berechnung der Fakultät (n!)

n! = 1 * 2 * 3 * ... * (n-1) * n

(n-1)!

rekursive Definition

n! = (n-1)! * n
1! = 1

4! = 3! * 4
3! = 2! * 3
2! = 1! * 2
1! = 1

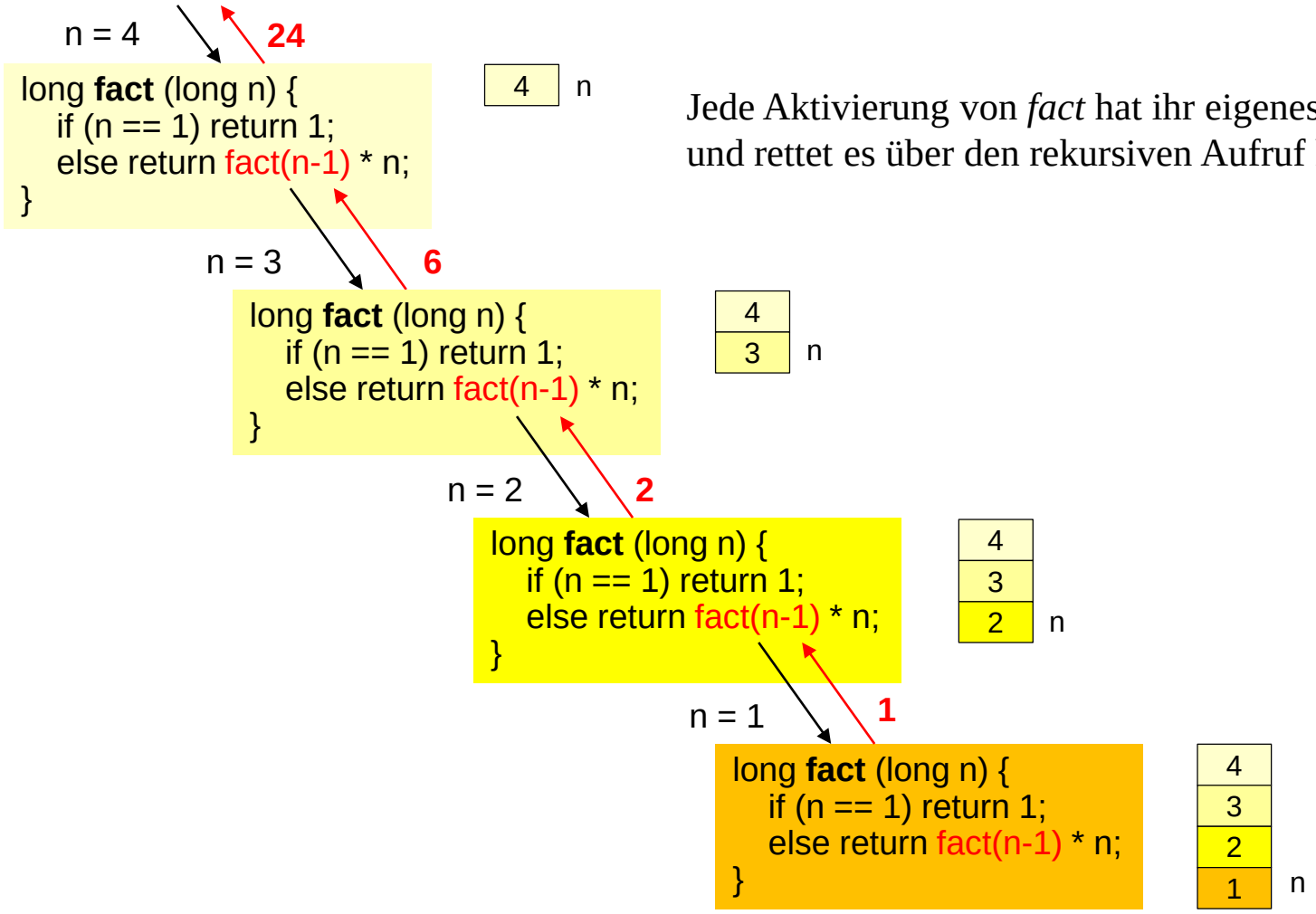
Rekursive Methode zur Berechnung der Fakultät

```
long fact (long n) {
   if (n == 1)
      return 1;
   else
      return fact(n-1) * n;
}
```

Allgemeines Muster

if (*Problem klein genug*)
   nichtrekursiver Zweig;
else
   rekursiver Zweig mit kleinerem Problem

Ablauf einer rekursiven Methode



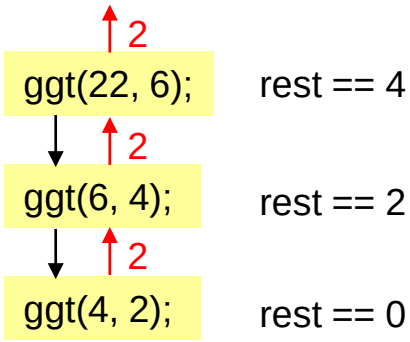
Jede Aktivierung von *fact* hat ihr eigenes *n* und rettet es über den rekursiven Aufruf hinweg

9. Rekursion

Beispiel: größter gemeinsamer Teiler

rekursiv

```
static int ggt (int x, int y) {
    int rest = x % y;
    if (rest == 0) return y;
    else return ggt(y, rest);
}
```



iterativ

```
static int ggt (int x, int y) {
    int rest = x % y;
    while (rest != 0){
        x = y; y = rest;
        rest = x % y;
    }
    return y;
}
```

Jeder rekursive Algorithmus kann auch iterativ programmiert werden

- rekursiv: meist kürzerer Quellcode
- iterativ: meist kürzere Laufzeit

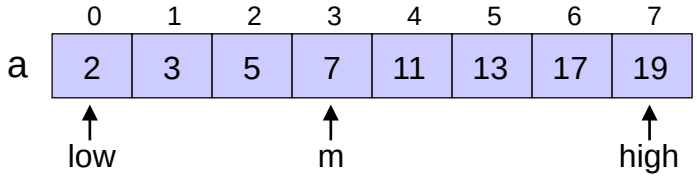
Rekursion v.a. bei rekursiven Datenstrukturen nützlich (Bäume, Graphen, ...)

9. Rekursion

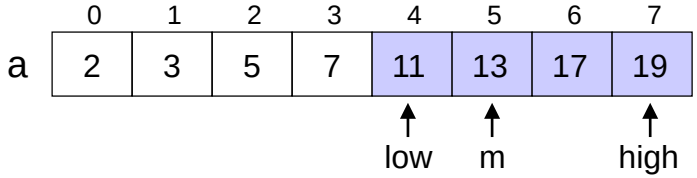
Beispiel: binäres Suchen iterativ

- schneller als sequentielles Suchen
- Array muss allerdings sortiert sein

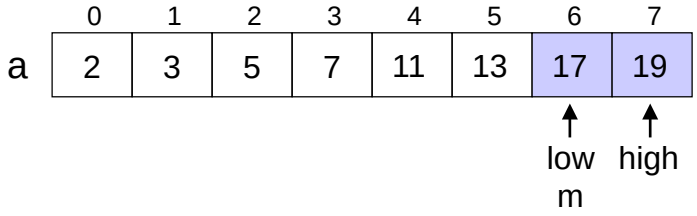
z.B. Suche von 17



- Index des mittleren Element bestimmen (m = (low + high) / 2)
- 17 > a[m] Þ zwischen a[m+1] und a[high] weitersuchen

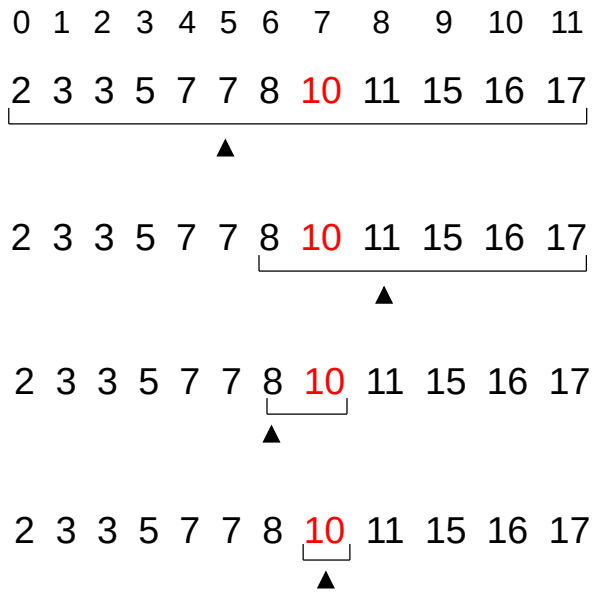


- 17 > a[m] Þ zwischen a[m+1] und a[high] weitersuchen



Binäres Suchen iterativ

```
static int binarySearch (int[] a, int x) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int m = (low + high) / 2;
        if (a[m] == x) return m;
        else if (x > a[m]) low = m + 1;
        else /* x < a[m] */ high = m - 1;
    }
    /* low > high*/
    return -1;
}
```



- Suchraum wird in jedem Schritt halbiert
- bei *n* Arrayelementen sind höchstens $log_2(n)$ Schritte nötig, um jedes Element zu finden

n	seq.Suchen	bin.Suchen
10	10	4
100	100	7
1000	1000	10
10000	10000	14

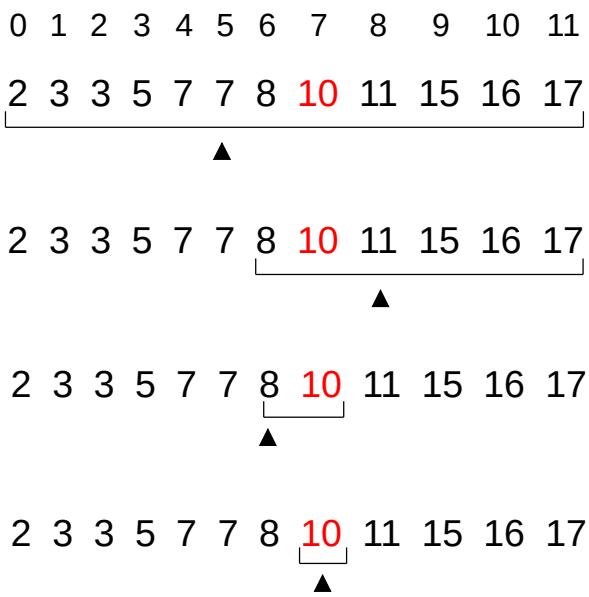
Laufzeitkomplexität = $O(log_2(n))$

Beispiel: binäres Suchen rekursiv

```
static int search (int[] a, int x, int low, int high) {
    if (low > high) return -1; // empty
    int m = (low + high) / 2;
    if (x == a[m]) return m;
    if (x < a[m]) return search(a, x, low, m-1);
    /* x > a[m] */ return search(a, x, m+1, high);
}
```

nichtrekursive Zweige

rekursive Zweige



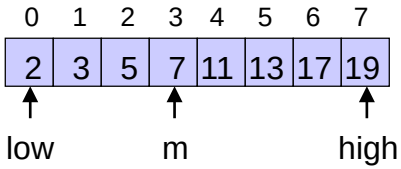
Ablauf des rekursiven binären Suchens

x = 17, low = 0, high = 7

6

```
static int search (int[] a, int x, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (x == a[m]) return m;
    if (x < a[m]) return search(a, x, low, m-1);
    return search(a, x, m+1, high);
}
```

m == 3

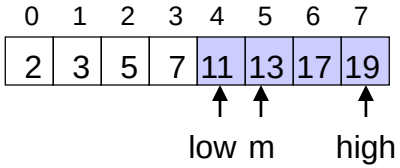


low = 4, high = 7

6

```
static int search (int[] a, int x, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (x == a[m]) return m;
    if (x < a[m]) return search(a, x, low, m-1);
    return search(a, x, m+1, high);
}
```

m = 5

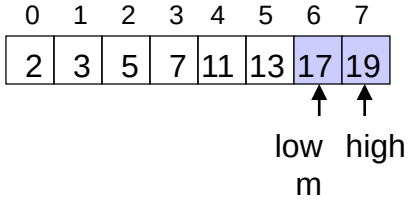


low = 6, high = 7

6

```
static int search (int[] a, int x, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (x == a[m]) return m;
    if (x < a[m]) return search(a, x, low, m-1);
    return search(a, x, m+1, high);
}
```

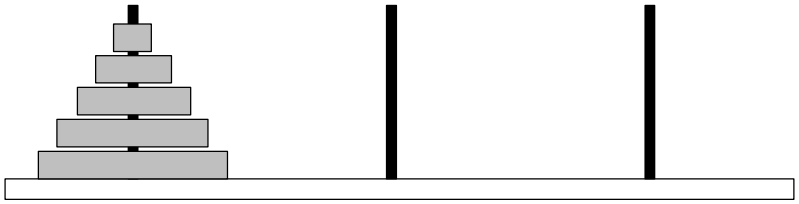
m = 6



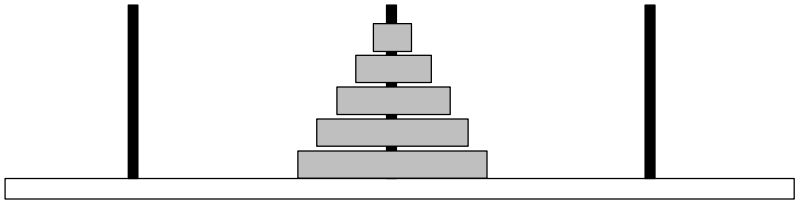
9. Rekursion

Beispiel: Türme von Hanoi

gegeben



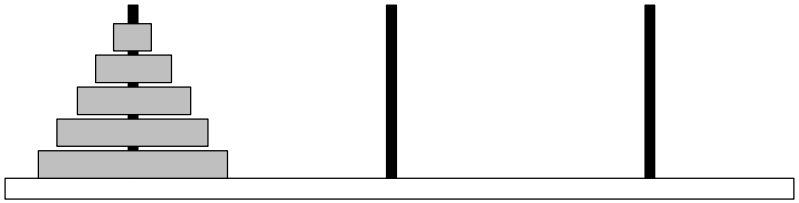
gesucht



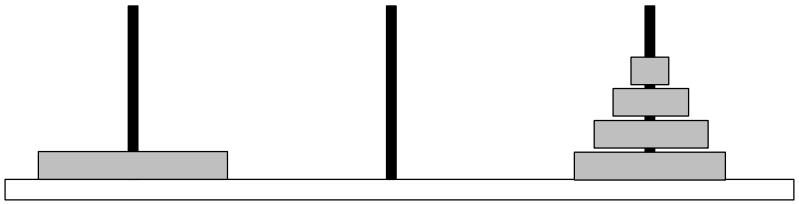
Bewege *n* Scheiben von Stab 1 nach Stab 2

- Nur 1 Scheibe aufs Mal
- Es darf nie größere Scheibe auf kleinerer liegen

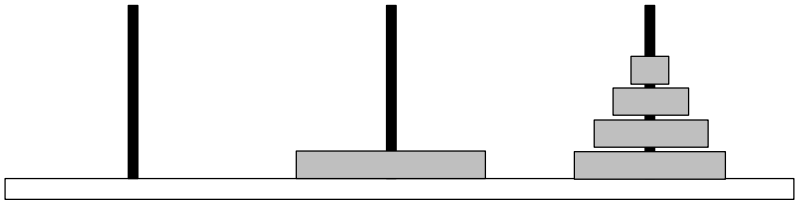
Rekursive Lösung



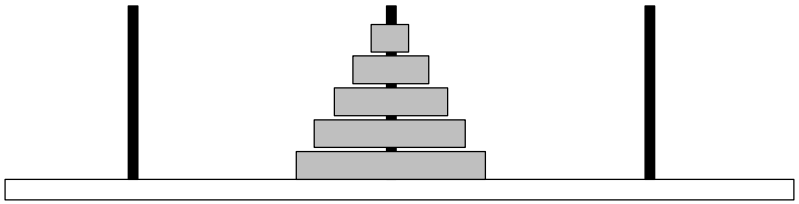
Bewege *n*-1 Scheiben von 1 nach 3 (rekursiv)



Bewege letzte Scheibe von von 1 nach 2



Bewege *n*-1 Scheiben von 3 nach 2 (rekursiv)



Implementierung

```
static void move (int n, int from, int to) {
    if (n == 1) {
        Out.println(from + " => " + to);
    } else {
        int other = 6 - (from + to);
        move(n - 1, from, other);
        Out.println(from + " => " + to);
        move(n - 1, other, to);
    }
}
```

1 + 2 + 3 = 6

6 - (1 + 2) = 3
6 - (1 + 3) = 2
6 - (2 + 3) = 1