Rekursion

Selbstbezug, rekursive Funktionen, rekursive Prozeduren, Terminierung, Effizienz, Korrektheit, Rekursion und Induktion

Ein kleines Problem

- Schreiben Sie eine Methode **`writeBin`**, die eine Dezimalzahl als Binärzahl ausdruckt.
  - Kopf: **`void writeBin(int n)`**
  - Ergebnis: Ausdruck der Binärdarstellung von n.
  - Die Schwierigkeit:
    - Die Ziffern von n entstehen in der falschen Reihenfolge
  - Der folgende Code ist daher keine Lösung:

```
void writeBin(int n){
   while(n > 0) {
    System.out.println(n mod 2);
    n = n div 2
   }
```

Selbstbezug

- Rekursion ist die Kunst, ein Problem auf ein einfacheres, aber gleichartiges zurückzuführen
  - Um einen *Stapel* von Papieren zu *sortieren*:
    - Teile ihn in zwei kleinere Stapel
    - *Sortiere* die kleineren *Stapel*
    - Füge die sortierten Stapel zu einem Stapel zusammen
  - Um eine *positive Zahl* Z *in eine Binärzahl* $b_n b_{n-1} ... b_1 b_0$ zu *verwandeln*
    - *Verwandle* die *positive Zahl* (Z div 2) *in eine Binärzahl* $b_n b_{n-1} ... b_1$
    - Setze b_0 = Z mod 2

- Wichtig ist:
  - Das Problem ist von einem diskreten Parameter n abhängig
    - Sortieren: Die Größe des Stapels
    - Binärzahl: Z
  - Die Parameter der Teilprobleme sind kleiner als n
  - Die Zerlegung in Teilprobleme bricht irgendwann ab. Die Lösung ist dann offensichtlich.
    - Sortieren: Stapel der Höhe 1 ist schon sortiert
    - Binärzahl: Z=0, Z=1 klar.

Summe, Fakultät

- Um die Zahlen von 0 bis n zu summieren
  - falls n=0: Ergebnis ist 0.

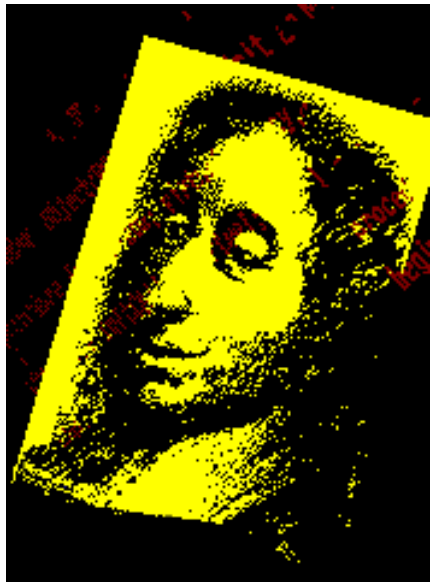
  Ansonsten:
  - summiere die Zahlen von 1 bis (n-1)
  - addiere n
  - ```
    int summe(n){
       if (n==0) return 0;
       else return summe(n-1)+n;
    }
    ```

- Die *Fakultät* von n ist definiert als
  n! := 1*2*3* ... *n.

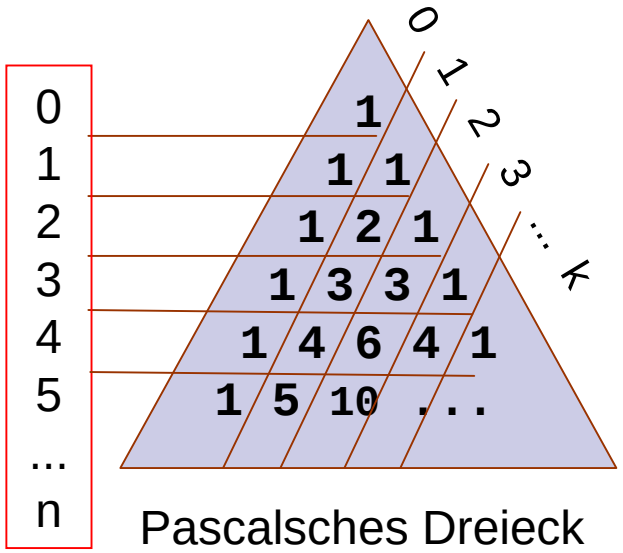
  Um n! zu berechnen:
  - falls n=1: Egebnis ist1.

  Ansonsten:
  - berechne (n-1)!
  - multipliziere mit n.
  - ```
    int fakt(int n){
      if(n==1) return 1;
      else return fakt(n-1)*n;
    }
    ```

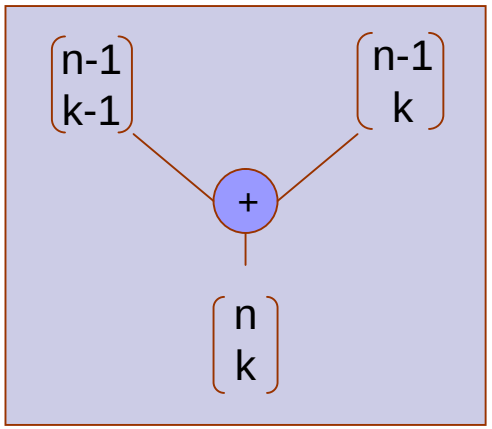
Binomische Formel



- Aus der Schule bekannt:
  - $(a+b)^2 = a^2 + 2*a*b+b^2$
  - $(a+b)^3 = a^3 + 3*a^2*b + 3*a*b^2 + b^3$
  - $(a+b)^n = a^n + \ldots + \binom{n}{k} * a^{(n-k)} * b^k + \ldots + b^n$
- $\binom{n}{k}$ ist Binominalkoeffizient, definiert durch das Pascalsche Dreieck



Pascalsches Dreieck

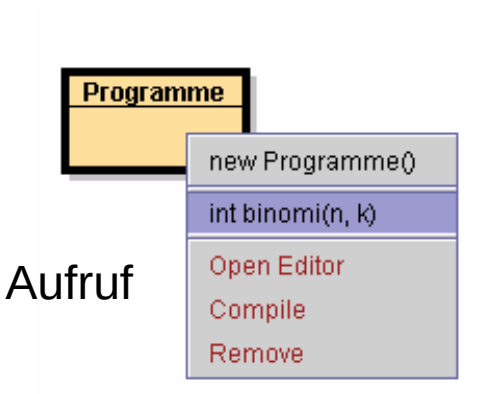


Bauprinzip

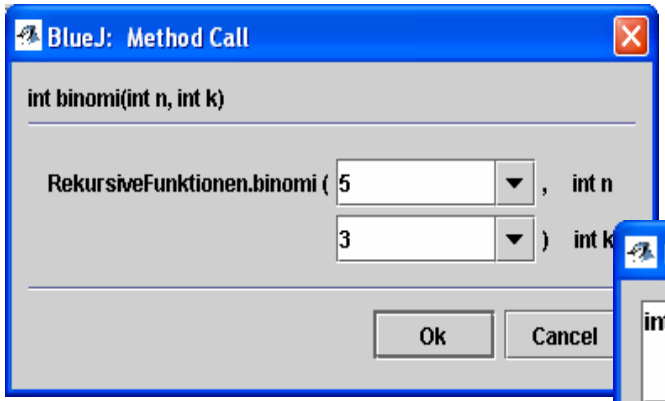
Binomische Formel in Java-BlueJ

```
class RekursiveFunktionen{
    static int binomi(int n, int k){
        if(k==0||n==k) return 1;
        else return binomi(n-1,k-1)+binomi(n-1,k);
    }
}
```

2 rekursive Aufrufe

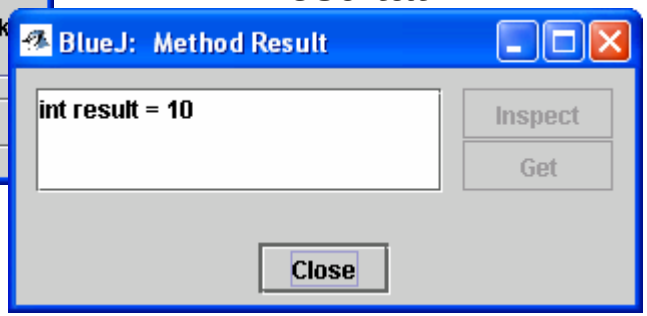


Aufruf



Parameter

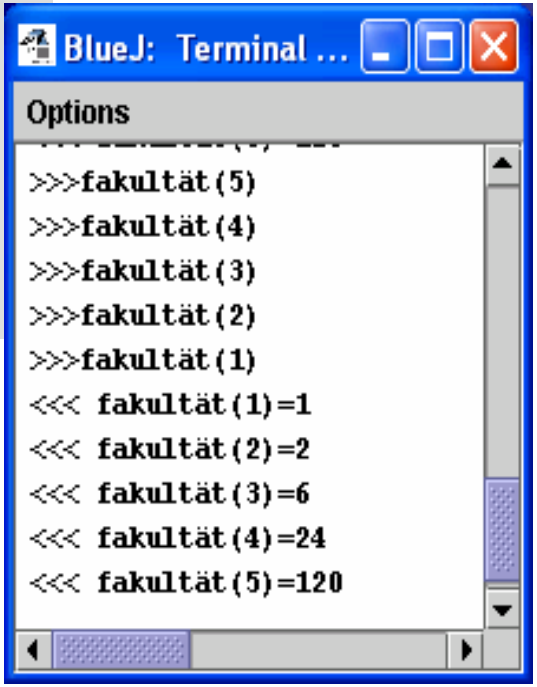
Resultat



Wie funktioniert Rekursion ?

- Wir verfolgen die Berechnung von n!, indem wir diagnostische Ausgabe einbauen:

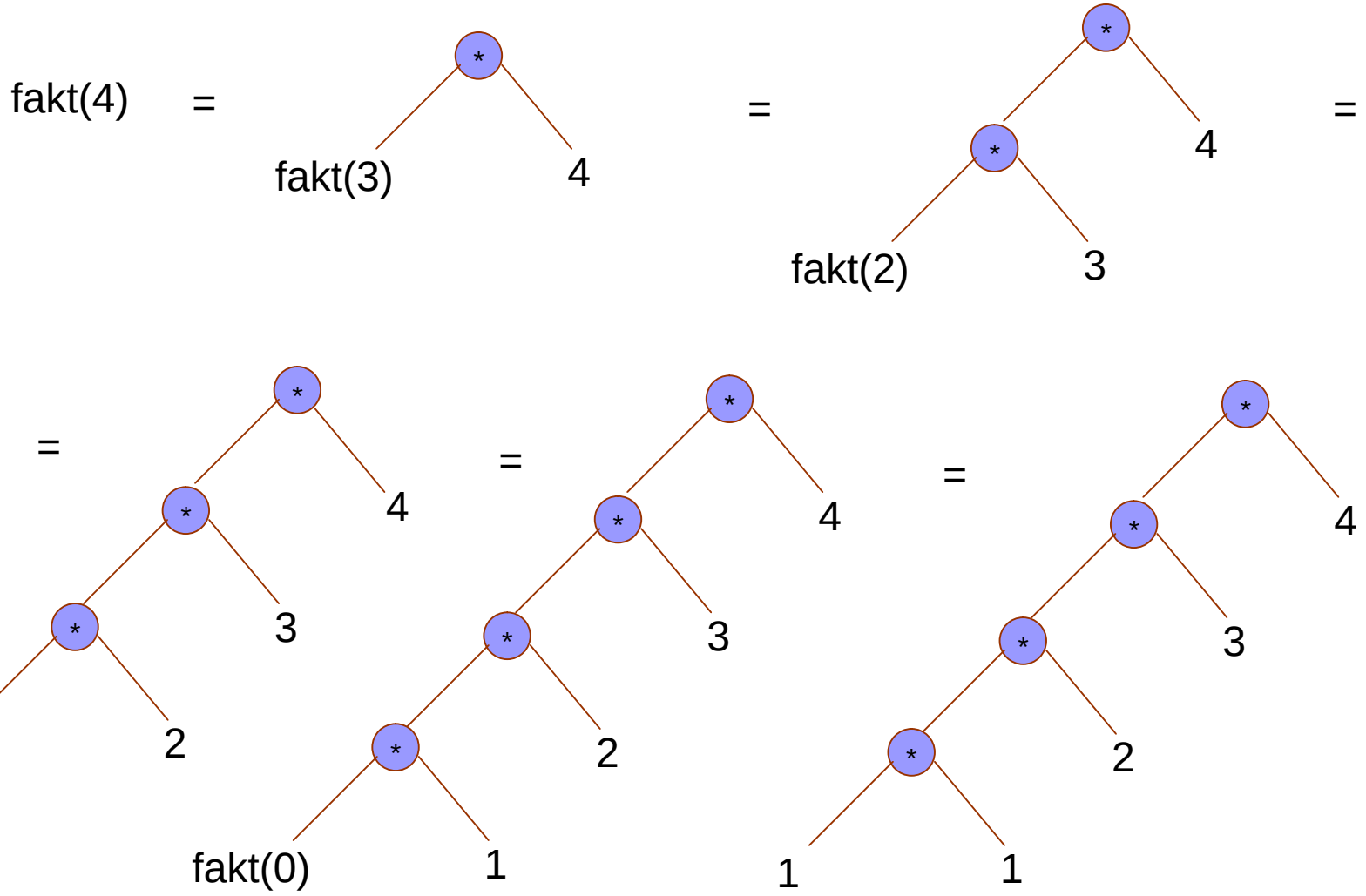
```
static int fakt(int n){
  if (n==0) return 1;
  else { System.out.println(
              ">>>fakt("+ n +")");
         int result = fakt(n-1)*n;
         System.out.println(
             "<<< fakt("+ n +")="+result);
         return result;
       }
  }
```



Wer zuletzt kommt, mahlt zuerst.

Entfaltung der Berechnung

- Die Berechnung kann man als Baum darstellen:



Rekursion kann jede Schleife ersetzen

- Jede Schleife
  - `while(bedingung) { ... tuWas(); ... }`
- lässt sich ersetzen durch einen Aufruf
  - `schleife();`
- wobei schleife definiert ist durch

```
void schleife(){
   if(bedingung) {
      ... tuWas(); ... schleife();
   }
}
```

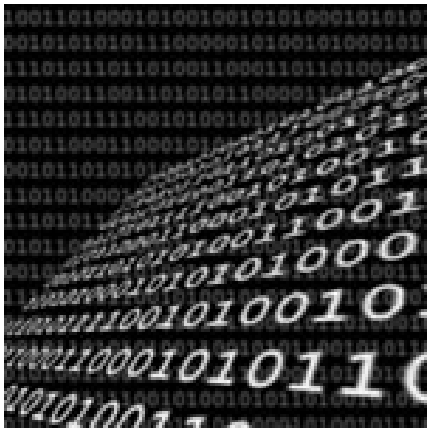
Ulam-Schleife

```
int n = 17;
while (n>1){
   if (n mod 2==0) n=n/2;
   else n=3*n+1
}
```

Ulam - Rekursion

```
void ulam(){
   if(n>1){
     if (n%2==0) n=n/2;
     else n=3*n+1;
     ulam();
  }
}
int n = 17;
ulam();
```

Rekursion ist natürlicher



- **writeBin** rekursiv:

```
void writeBin(int n){
  if (n < 2) System.out.println(n);
  else {
       writeBin(n / 2);
       System.out.println(n % 2);
  }
}
```

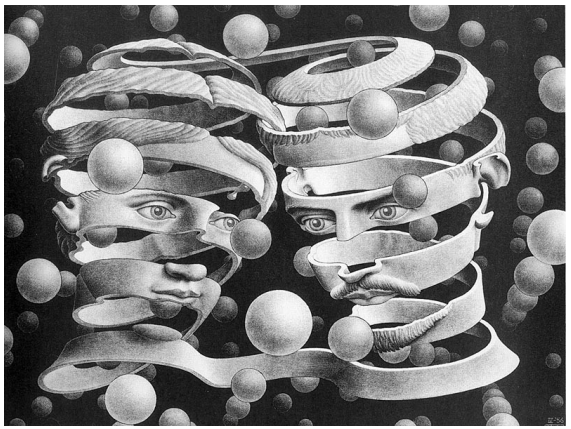
- Zuerst werden die ersten Bits geschrieben
- Danach das letzte

Wechselseitige Rekursion

- Wenn zwei oder mehr Funktionen wechselseitig aufeinander Bezug nehmen, spricht man von wechselseitiger Rekursion

- ```
  boolean istGerade(int n){
      if (n==0) return true;
      else return istUngerade(n-1);
  }
  ```

- ```
  boolean istUngerade(int n){
      if (n==0) return false;
      else return istGerade(n-1);
  }
  ```



Das 3-7-11-Spiel

- Zwei Spieler dürfen von einem Haufen von n Smarties
  - abwechselnd
  - 3, 7, oder 11

  Smarties entfernen
- Wer nicht mehr ziehen kann, hat verloren
- Wir beginnen mit n Smarties (z.B. n=137)
  - Sie machen den ersten Zug
  - Haben Sie eine Gewinnstrategie ?
  - Wie lautet diese ggf. ?
- Schreiben Sie eine Java-Methode, um optimal zu spielen.

Einer wird gewinnen

- Wenn ich am Zug bin
  - Wenn n < 3 habe ich verloren
- Wenn Du am Zug bist
  - Wenn n < 3 hast Du verloren
- Wenn Ich am Zug bin
  - Wenn Du bei (n-3) verloren hast, kann ich gewinnen
  - Wenn Du bei (n-7) verloren hast, kann ich gewinnen
  - Wenn Du bei (n-11) verloren hast, kann ich gewinnen
- Wenn Du am Zug bist
  - Wenn Ich bei (n-3) verloren habe, kannst Du gewinnen
  - Wenn Ich bei (n-7) verloren habe, kannst Du gewinnen
  - Wenn Ich bei (n-11) verloren habe, kannst Du gewinnen

- Klar !
- Ach ja ? Wie ?

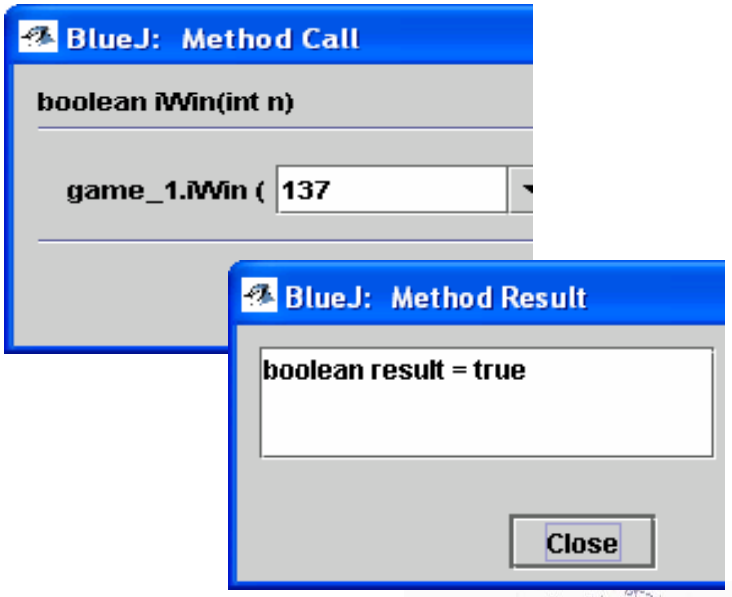
Das Spiel

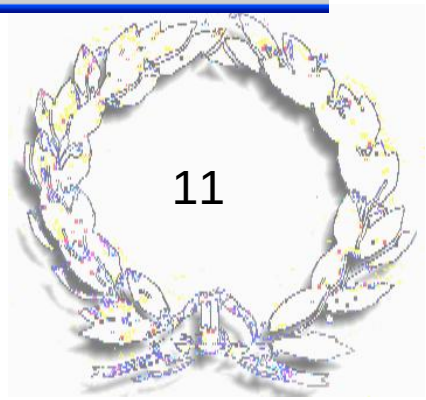
```
class Game {
    boolean iWin(int n){
        return
            n >= 3 && ! youWin(n-3)
         || n >= 7 && ! youWin(n-7)
         || n >=11 && ! youWin(n-11);
    }

    boolean youWin(int n){
        return
            n >= 3 && ! iWin(n-3)
         || n >= 7 && ! iWin(n-7)
         || n >=11 && ! iWin(n-11);
    }
}
```

Man käme auch mit einer Funktion *firstPlayerWins* aus, denn *iWin* und *youWin* stellen die gleiche Funktion dar!
Mit zweien ist es - vielleicht - verständlicher .

Testen der Zugoptionen

- **`iWin(137) = true`**
  - Ich kann also gewinnen
  - Aber wie ?
- Wir testen
  - **`youWin(137-3) = true`**
  - **`youWin(137-7) = true`**
  - **`youWin(137-11)= false`**
- Wir schließen
  - 11 Smarties wegnehmen ist der Siegeszug





Terminierung rekursiver Funktionen

- Damit eine rekursive Funktion


```
t rekFunk(t1 n1, ..., tk nk){
  ... rekFunk(e1,...,ek) ...
  }
```

terminiert, müssen die Parameterwerte $\mathbf{e_1, \ldots, e_k}$ des rekursiven Aufrufes „einfacher“ sein als die Parameterwerte $\mathbf{n_1}$, ..., $\mathbf{n_k}$ des originalen Aufrufs

- Was heißt „einfacher“ ?

  Es muss eine mathematische Funktion

  $$F : t_1 \times \ldots \times t_k \rightarrow N$$

  geben mit

  $$F(n_1,\ldots,n_k) > F(e_1,\ldots,e_k) \geq 0.$$

Beispiele

```
int ggT(int m,n){
     if (m==n) return m;
     else if (m > n) return ggT(m-n,n);
     else return ggT(m,n-m);
}
```

- F(m,n) = m+n

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

- F(n) = n

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

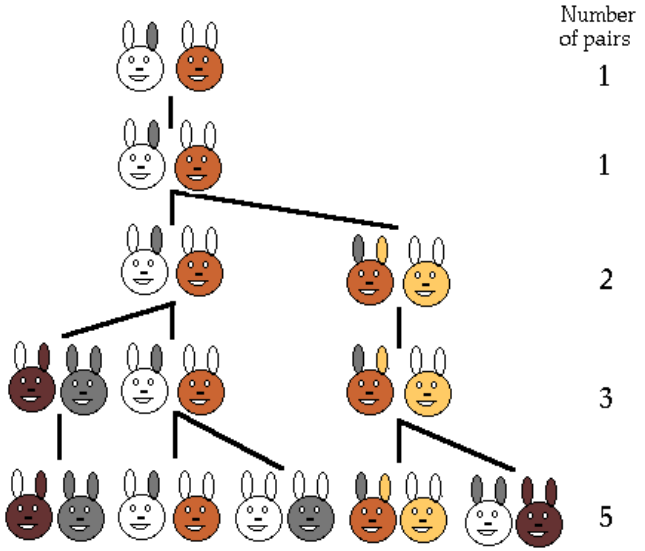
- F(n) = n

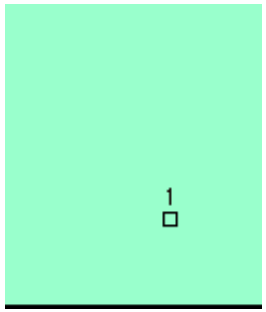
```
int mcCarthy(int n){
     if (n > 100) return n-10;
     else return
       macCarthy(mcCarthy(n+11));
```

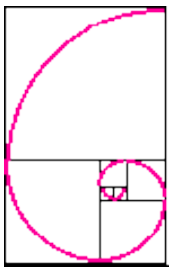
- Experimentieren Sie mit dieser Funktion
- Zeigen Sie, dass sie immer terminiert

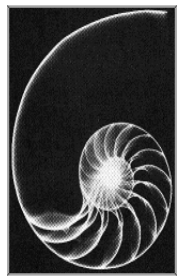
Die Fibonacci-Folge

- $(F_n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$
- Bildungsgesetz:
  - $F_0 = 1, F_1 = 1,$
  - $F_{n+2}=F_n + F_{n-1}$
- Anwendungen
  - Natürliche Wachstumsprozesse
    - Kaninchenvermehrung
    - Blütenstände, Tannenzapfen
    - Schneckenhäuser
  - Die Fibonacci-Folge ist eng mit dem goldenen Schnitt verknüpft
- Populäres Thema im Internet,
  - z.B. hier:
    http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html









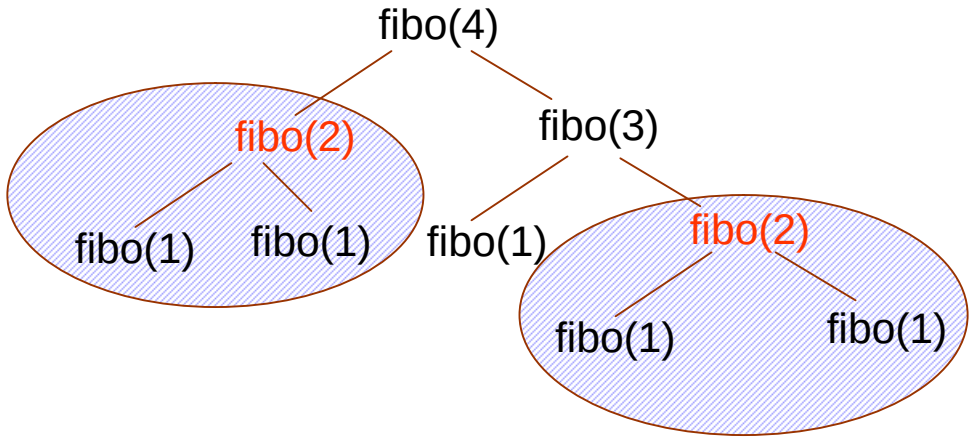
Effizienz

- Mit jeder Programmiermethodologie kann man ineffiziente Programme schreiben
  - Anfängerfehler bei rekursiven Funktionen:
    - Wiederberechnung bereits bekannter Werte
    -
```
int fibo(int n){
      if (n < 2) return 1;
      else return fibo(n-2)+fibo(n-1);
}
```

Dieser Aufruf muss **`fibo(n-2)`** erneut berechnen

Leonardo von Pisa
Genannt: *Fibonaccio*

Aufrufbaum:



Effizienzsteigerung:

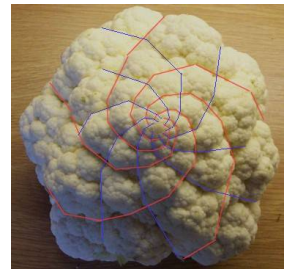
Iterative Version von fibo:

```
static int fibonacci(int n){
  int letzte=1, vorletzte = 1;

  for(int k=0; k<=n; k++){
     {int temp = letzte;
      letzte = letzte+vorletzte;
      vorletzte = temp;
     }
  }
  return vorletzte;
}
```

- Zwei Hilfsvariablen: **`letzte, vorletzte`**
- **`letzte`** ≅ fibo(n+1)
  `vorletzte` ≅ fibo(n)
- In jedem Schritt:
  - **`vorletzte`** ← **`letzte`**
  - **`letzte`** ← **`letzte + vorletzte`**
- Da das nicht gleichzeitig geht, braucht man Hilfsvariable **`temp`**.

fibo(n) :

n	0	1	2	3	4	5	6	7
letzte	1	2	3	5	8	13	21	34
vorletzte	1	1	2	3	5	8	13	21



Rekursiv: effizient und klarer

- Rekursiv geht auch das einfacher
  - Vertauschen der Variablen bei der Parameterübergabe
  - Keine Hilfsvariable benötigt
- Trick: Hilfsvariablen werden zu Parametern

```
static int fibIter(int n, int k, int letzte, int vorletzte){
  if(k==n)
    return vorletzte;
  else
    return fibIter(n, k+1,letzte+vorletzte, letzte);
}

static int fibo(int n){ return fibIter(n,0,1,1); }
```

fibo(n) :

n	0	1	2	3	4	5	6	7
letzte	1	2	3	5	8	13	21	34
vorletzte	1	1	2	3	5	8	13	21

Weitere Vereinfachung

- Wir können **fibIter** noch vereinfachen:
  - In **fibIter(n,k,..., ...)** wird **k** hochgezählt bis n.
  - Stattdessen können wir **n** auf **0** hinuterzählen:

```
static int fibIter(int n, int letzte, int vorletzte){
  if(n==0) return vorletzte;
  else     return fibIter(n-1, letzte+vorletzte, letzte);
}

static int fibo(int n){
   return fibIter(n,1,1);
}
```

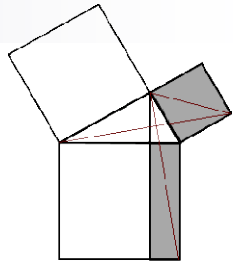
Wie können wir sicher sein, dass alles stimmt ?

- Testen ?
- Beweisen ?

Rekursion und Induktion

- Rekursion und Induktion haben vieles gemeinsam
  - Komplizierte Fälle werden auf einfachere zurückgeführt
  - Basisfall wird separat behandelt – oft trivial.
- Zwei Seiten einer Medaille
  - Funktionen auf induktiv definierten Mengen sind rekursiv
  - Korrektheit rekursiver Funktionen führt auf induktive Beweise
- Beispiel: Fibonacci
  - Sei F die richtige, die mathematische Fibonacci-Funktion
  - Wir behaupten fibo(n) = F_n
  - Wir wollen es induktiv beweisen

Der Beweis



- Verallgemeinerung der Behauptung:
- Für alle n gilt:

  Für alle k gilt: fibIter(n, F_{k+1}, F_k) = F_{n+k} ↔ Hyp(n)

- Induktion über n
- InduktionsAnfang: n=0:
  - Für alle k gilt: fibIter(0,F_{k+1},F_k)=F_k=F_{0+k} ↔ Hyp(0)
- Induktionsschritt:
  - Für alle k gilt:

    fibIter(n+1,F_{k+1},F_k) =

    = fibIter(n,F_{k+1}+F_k,F_{k+1}) =

    = fibIter(n,F_{k+2},F_{k+1}) =

    = $F_{n+(k+1)}$

    = $F_{(n+1)+k}$

Hyp(n)

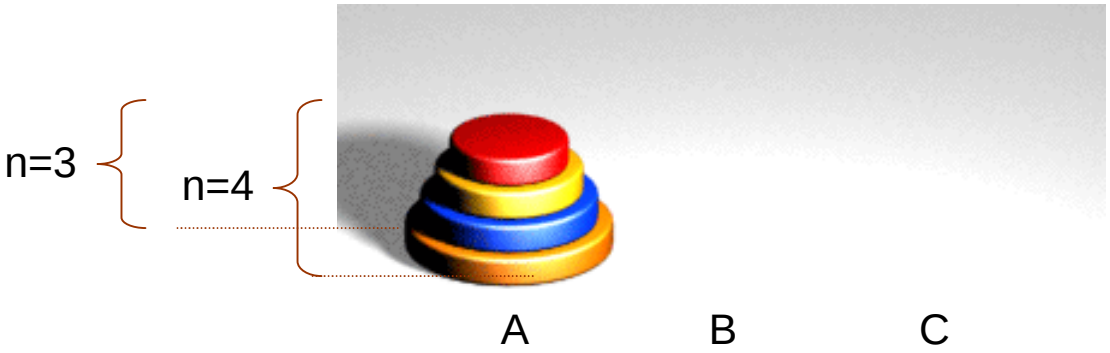
Hyp(n+1)

Unvermeidliche Hanoi

- Aufgabe:
  - Bewege Turm von A nach C
- Regeln
  - 3 mögliche Positionen (A, B, C)
  - 1 Scheibe pro Zug
  - nie eine große auf einer kleineren Scheibe
- Gesucht:
  - Lösung für n Scheiben

Idee (für n=4):

- Bewege 3-er Turm der obersten 3 Scheiben von A nach B
  - wie ? Na rekursiv …
- Bewege unterste Scheibe von A nach C
- Bewege 3-er Turm von B nach C
  - wie ? come on …



Hà Nội

- **`hanoi (int n)`**
  - versteckt die ansonsten globalen Variablen ***von***, ***über***, ***zu***
- **`hanoi (int n, char von, char über, char nach)`**
  - hinschreiben der Idee
- **`bewegeScheibe(char von, char nach)`**
  - müsste man graphisch noch verschönern

```
static void hanoi(int n){
    hanoi(n,'A','B','C');
}
static void hanoi(int n, char von, char über, char nach){
  if(n>0){
    hanoi(n-1,von,nach,über);
    bewegeScheibe(von,nach);
    hanoi(n-1,über,von,nach);
  }
}
static void bewegeScheibe(char von, char nach){
   System.out.println(
     "Scheibe von "+von+" nach "+nach);
   }
```