

Teil

Imperative Programmierung

Unterrichtseinheit 16

Rekursion

Dr. Dietrich Boles

- Rekursion
 - Definitionen
 - Rekursive Prozeduren
 - Rekursive Funktionen
 - Lokale Variablen
 - Parameter
 - Endlosrekursion
 - Anmerkungen
- Beispiele
- Backtracking
- Zusammenfassung

- "Definition eines Problems, einer Funktion oder eines Verfahrens durch sich selbst"
- bereits bekannt:

- direkt rekursive Syntaxdiagramme/EBNF:

```
<boolescher Ausdruck> ::= "true" | "false" |  
                           "(" <boolescher Ausdruck> ")" ;
```

- indirekt rekursive Syntaxdiagramme/EBNF:

```
<Anweisung> ::= ... | <while-Anweisung> ;  
<while-Anweisung> ::= "while" "(" <bA> ")" <Anweisung> ;
```

- Mathematik:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

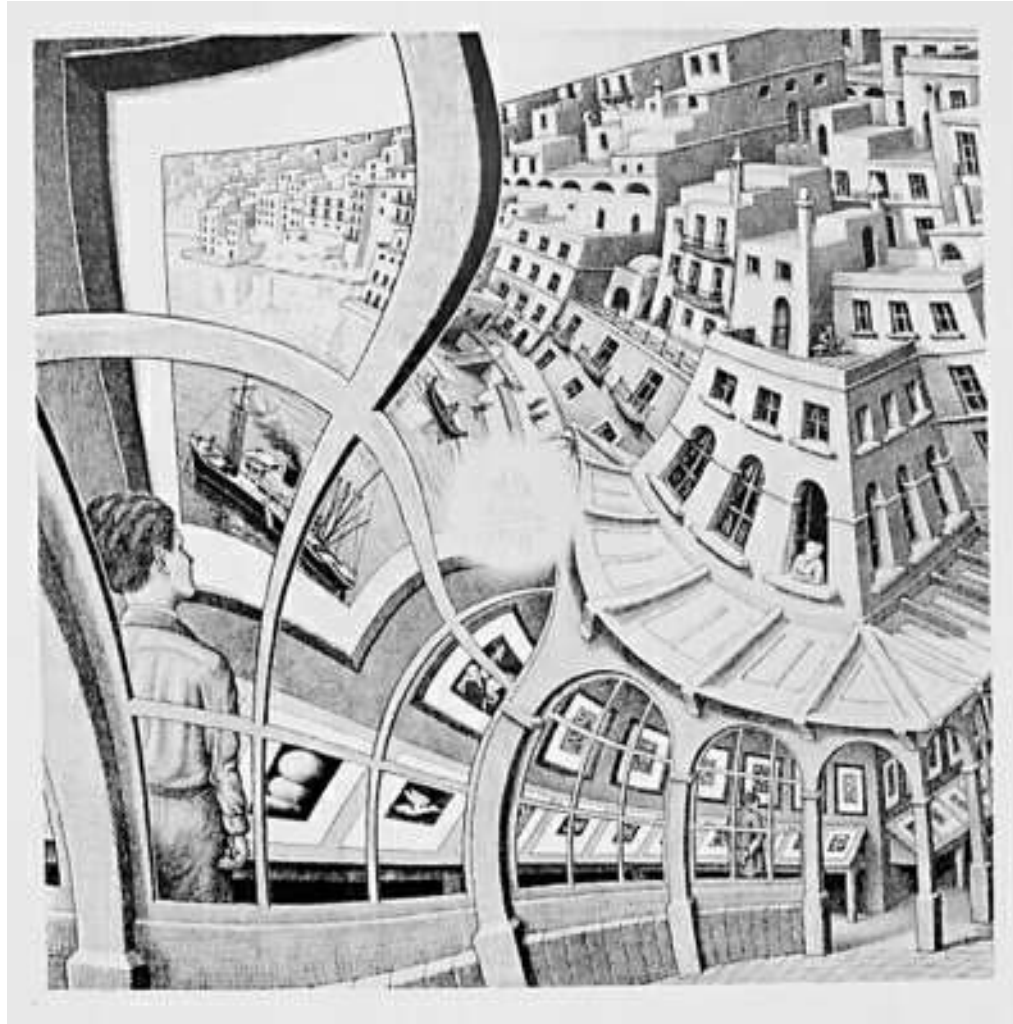
➤ Spielzeug:



Matroschka

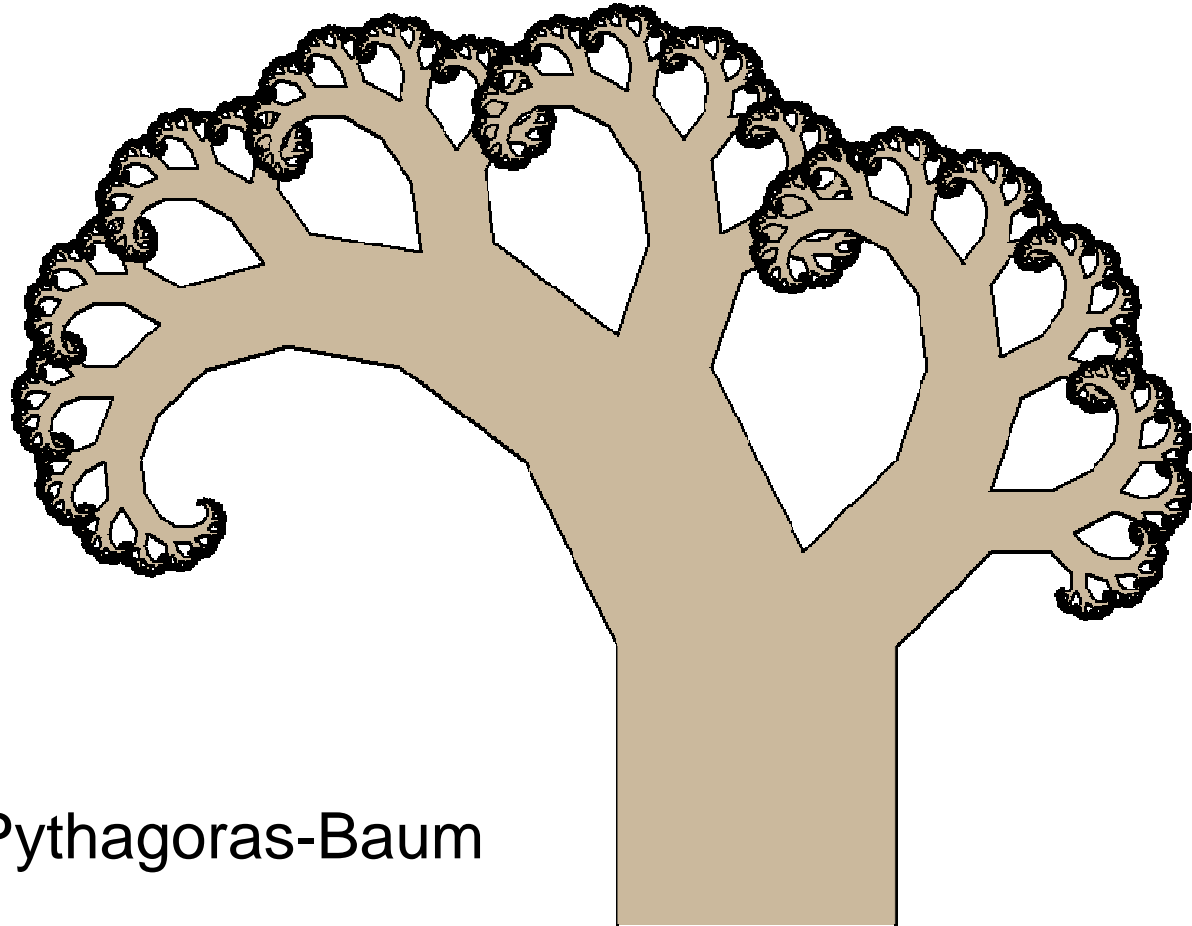
Quelle: German Wikipedia

➤ Kunst:



M.C. Escher; Bildgalerie

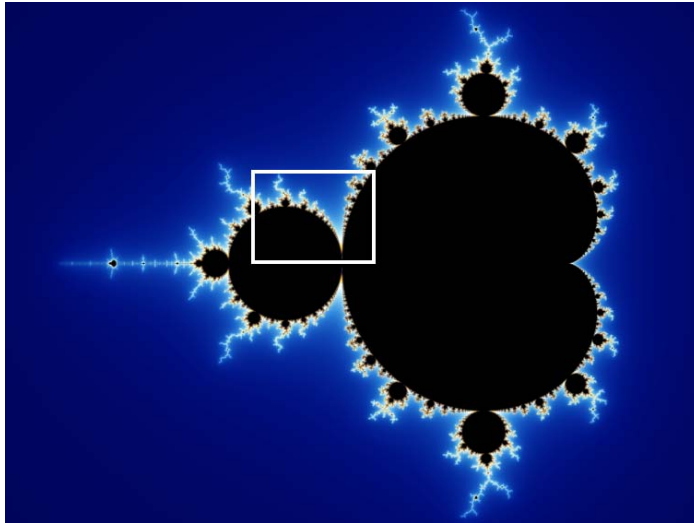
➤ Fraktale:



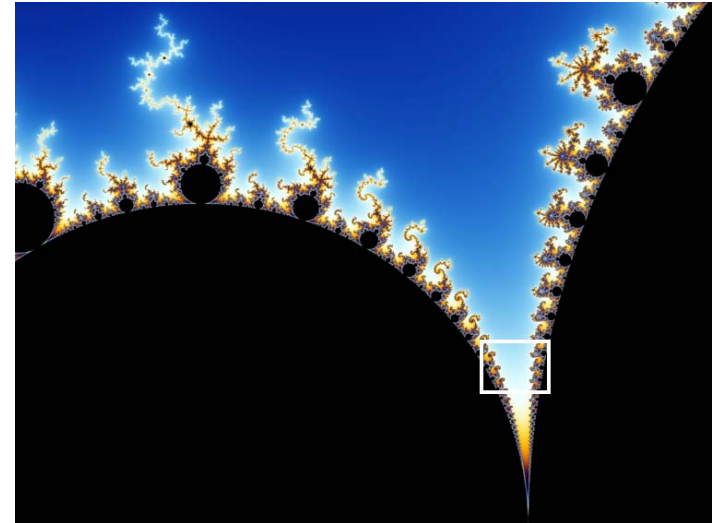
Pythagoras-Baum

Quelle: German Wikipedia

➤ Fraktale:



Mandelbrotmenge



Quelle: German Wikipedia, Wolfgang Beyer

<http://www.mathematik.ch/anwendungenmath/fractal/julia/MandelbrotApplet.php>

Definition (Rekursion):

Eine Funktion heißt **rekursiv**, wenn sie während ihrer Abarbeitung erneut aufgerufen wird.

Definition (direkte Rekursion):

Eine Funktion heißt **direkt rekursiv**, wenn der erneute Aufruf im Funktionsrumpf der Funktion erfolgt.

Definition (indirekte Rekursion):

Eine Funktion heißt **indirekt rekursiv**, wenn der erneute Aufruf nicht im Funktionsrumpf der Funktion selbst sondern in einer anderen Funktion erfolgt.

Definition (Funktionsinkarnation):

konkreter Aufruf einer Funktion

Definition (Rekursionstiefe):

Anzahl der aktuellen Inkarnationen einer Funktion minus 1

Definition (Variableninkarnation):

konkrete Ausprägung (Speicherplatz) einer Variablen

Iterativer Algorithmus:

Algorithmus, der Wiederholungsanweisungen verwendet

Rekursiver Algorithmus:

Algorithmus, der rekursive Funktionen verwendet

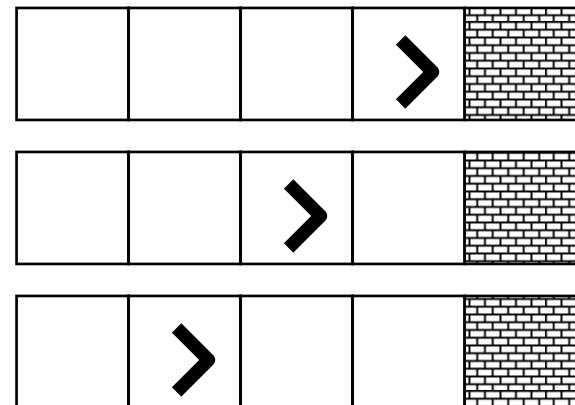
Der Hamster soll bis zur nächsten Wand laufen!

Iterative Lösung:

```
void zurMauer() {  
    while (vornFrei())  
        vor();  
}
```

Direkt rekursive Lösung:

```
void zurMauerR() {  
    if (vornFrei()) {  
        vor();  
        zurMauerR();  
    }  
}
```



Der Hamster soll alle Körner auf dem aktuellen Feld einsammeln!

Iterative Lösung:

```
void sammle() {  
    while (kornDa())  
        nimm();  
}
```

Direkt rekursive Lösung:

```
void sammleR() {  
    if (kornDa()) {  
        nimm();  
        sammleR();  
    }  
}
```

Der Hamster soll bis zur nächsten Wand und dann zurück zur Ausgangsposition laufen!

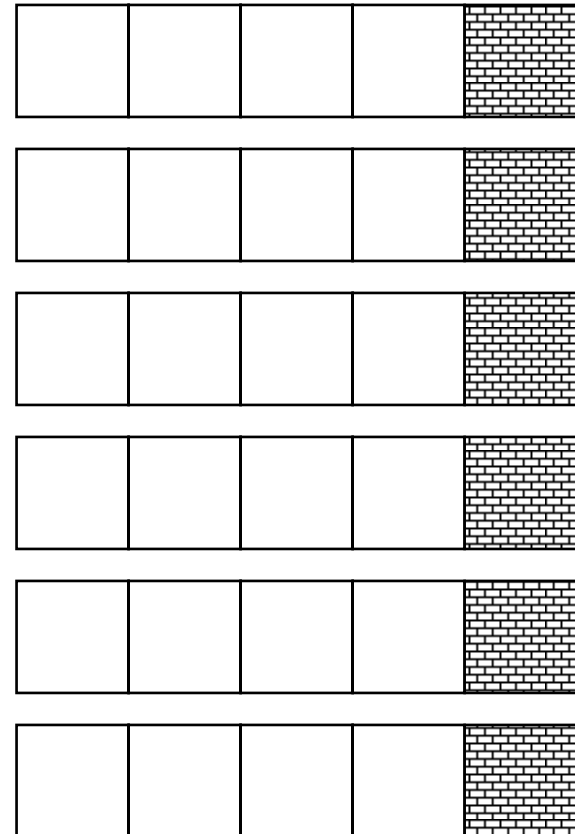
Iterative Lösung:

```
void hinUndZurueck() {  
    int anzahl = 0;  
    while (vornFrei()) {  
        vor();  
        anzahl++;  
    }  
  
    linksUm(); linksUm();  
  
    while (anzahl > 0) {  
        vor();  
        anzahl--;  
    }  
}
```

Der Hamster soll bis zur nächsten Wand und dann zurück zur Ausgangsposition laufen!

Direkt rekursive Lösung:

```
void hinUndZurueckR() {  
    if (vornFrei()) {  
        vor();  
        hinUndZurueckR();  
        vor();  
    } else {  
        kehrt();  
    }  
}  
  
void kehrt() {  
    linksUm();  
    linksUm();  
}
```



Rekursive Prozeduren (5)

Schema:



main: hUZR (1.) hUZR (2.) hUZR (3.)

```
hUZR();    vornFrei -> t
           vor();
           hUZR(); -----> vornFrei -> t
                               vor();
                               hUZR(); -----> vornFrei -> f
                                       kehrt();
                                           <-----
                                           vor();
                                           <-----
                                           vor();
                                           <-----
```

Befehlsfolge: vor(); vor(); kehrt(); vor(); vor();

Der Hamster soll bis zur nächsten Wand und dann zurück zur Ausgangsposition laufen!

Indirekt rekursive Lösung:

```
void hinUndZurueckR() {  
    if (vornFrei()) {  
        laufe();  
    } else {  
        linksUm(); linksUm();  
    }  
}  
  
void laufe() {  
    vor();  
    hinUndZurueckR();  
    vor();  
}
```

Der Hamster soll die Anzahl an Schritten bis zur nächsten Mauer zählen!

Iterative Lösung:

```
int anzahlSchritte() {  
    int anzahl = 0;  
    while (vornFrei()) {  
        vor();  
        anzahl++;  
    }  
    return anzahl;  
}
```

Rekursive Lösung:

```
int anzahlSchritteR() {  
    if (vornFrei()) {  
        vor();  
        return anzahlSchritteR() + 1;  
    } else  
        return 0;  
}
```


Rekursive Funktionen (2)

Schema:

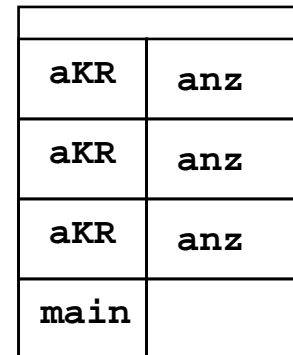
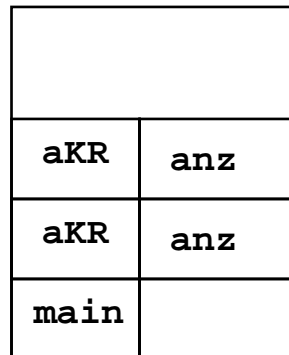
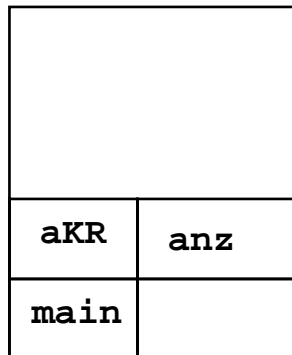
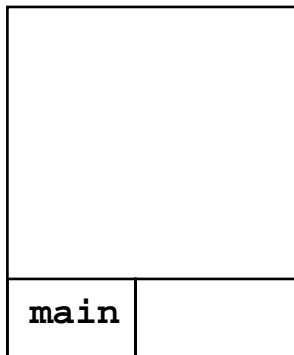


main:	aSR (1.)	aSR (2.)	aSR (3.)
i=aSR();	vornFrei -> t vor(); aSR() ----->	vornFrei -> t vor(); aSR() ----->	vornFrei -> f return 0;
		0 <----- return 0 + 1;	
	1 <----- return 1 + 1;		
2 <----- i=2;			

Der Hamster soll die Anzahl an Körnern im Maul zählen!

```
int anzahlKoernerR() {  
    if (!maulLeer()) {  
        int anz = 0;  
        gib();  
        anz = anzahlKoernerR();  
        nimm(); // Vermeidung von Seiteneffekten!  
        return anz + 1;  
    } else  
        return 0;  
}
```

Speicher

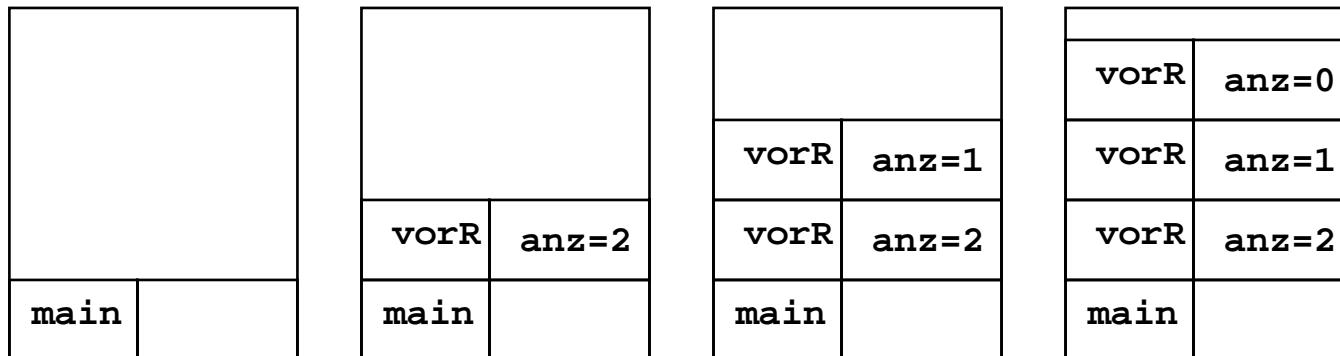


...

Der Hamster soll "anz"-Schritte nach vorne gehen!

```
void vorR(int anz) {  
    if ((anz > 0) && vornFrei()) {  
        vor();  
        vorR(anz-1);  
    }  
}
```

Speicher



Endlosrekursion:

```
void sammleR() {  
    if (kornDa()) {  
        sammleR();  
        nimm();  
    }  
}
```

Rekursionstiefe: im Prinzip "unendlich"!

erzeugt im allgemeinen einen Laufzeitfehler: Stack overflow!

Dem Java-Interpreter kann man die gewünschte Stackgröße mitteilen!

- Satz:
 - zu jedem rekursiv formulierten Algorithmus gibt es einen äquivalenten iterativen Algorithmus

- Vorteile rekursiver Algorithmen:
 - kürzere Formulierung
 - leichter verständliche Lösung
 - Einsparung von Variablen
 - teilweise sehr effiziente Problemlösungen (z.B. Quicksort)

- Nachteile rekursiver Algorithmen:
 - weniger effizientes Laufzeitverhalten (Overhead beim Funktionsaufruf)
 - Verständnisprobleme bei Programmieranfängern
 - Konstruktion rekursiver Algorithmen "gewöhnungsbedürftig"

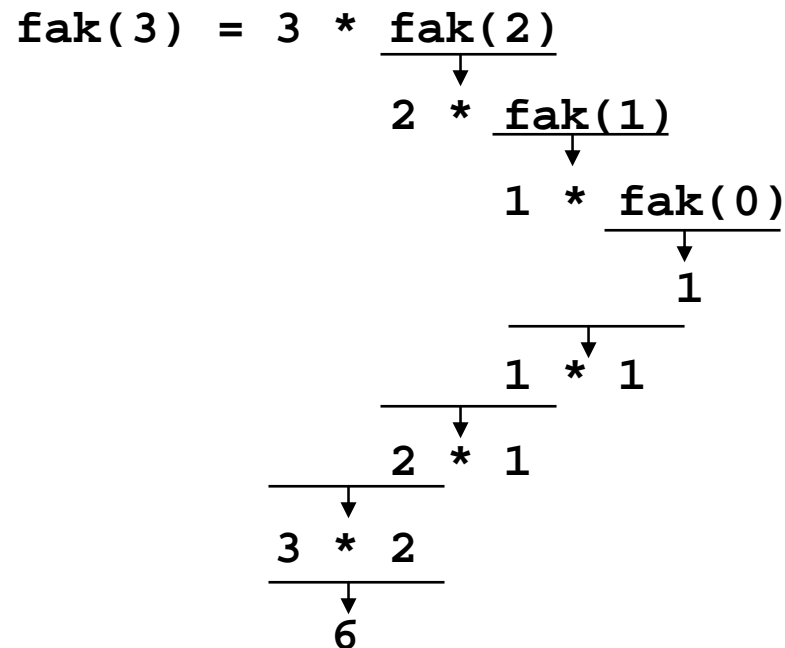
Anzahl an Ziffern einer Zahl ermitteln:

```
static int length(int zahl) { // iterativ
    if (zahl == 0) return 1;
    int laenge = 0;
    while (zahl != 0) {
        zahl /= 10;
        laenge++;
    }
    return laenge;
}
```

```
static int lengthR(int zahl) { // rekursiv
    if (zahl >= -9 && zahl <= 9) return 1;
    return lengthR(zahl/10) + 1;
}
```

Berechnung der Fakultätsfunktion:
$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

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



Berechnung einer Fibonacci-Zahl:

$$\text{fib}(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 1 & \text{falls } n = 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{sonst} \end{cases}$$

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


Beispiel 4

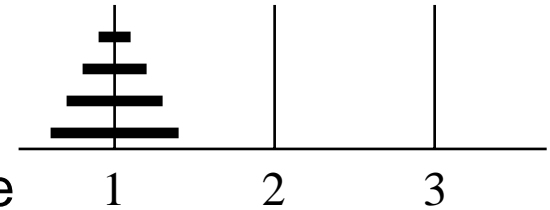
Türme von Hanoi:

Gegeben: 3 Pfosten mit n Scheiben

Ziel: Lege alle n Scheiben von 1 nach 3

Restriktion 1: immer nur eine Scheibe bewegen

Restriktion 2: niemals größere auf kleinere Scheibe



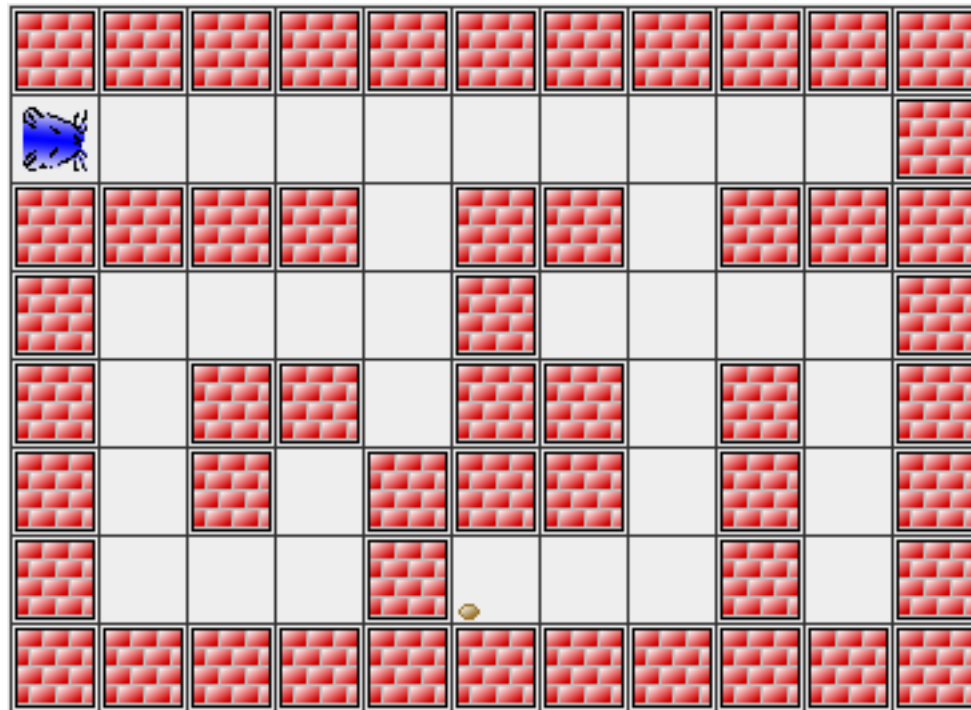
<http://thinks.com/java/hanoi/hanoi.htm>

```
class Hanoi {  
    public static void main(String[] args) {  
        int hoehe = IO.readInt("Hoehe: ");  
        verlegeTurm(hoehe, 1, 3, 2);  
    }  
    static void verlegeTurm(int hoehe, int von,  
                             int nach, int ueber) {  
        if (hoehe > 0) {  
            verlegeTurm(hoehe-1, von, ueber, nach);  
            IO.println(von + "-" + nach);  
            verlegeTurm(hoehe-1, ueber, nach, von);  
        } } }
```

Demo

- Prinzip:
 - Versuch, eine Teillösung eines gegebenen Problems systematisch zu einer Gesamtlösung auszubauen
 - falls in einer gewissen Situation ein weiterer Ausbau einer vorliegenden Teillösung nicht mehr möglich ist ("Sackgasse"), werden eine oder mehrere der letzten Teilschritte rückgängig gemacht
 - die dann erhaltene reduzierte Teillösung versucht man auf einem anderen Weg wieder auszubauen
 - Wiederholung des Verfahrens, bis Lösung gefunden wird oder man erkennt, dass keine Lösung existiert
- Grundlage der Programmiersprache PROLOG!
- Bekannte Probleme:
 - Springerproblem
 - Acht-Damenproblem
 - Labyrinthsuche

Aufgabe: Der Hamster steht am Eingang eines zyklensfreien Labyrinths, in dem er ein Korn finden und auf dem schnellsten Weg zurücktransportieren soll!



Demo

```
void main() {  
    durchsucheLabyrinth();  
}
```

```
boolean durchsucheLabyrinth() {  
    if (kornDa()) {  
        nimm();  
        kehrt(); // mach dich auf den Heimweg  
        return true;  
    }
```

```
    if (linksFrei() && durchsucheTeilLabyrinthLinks()) {  
        kehrt(); // mach dich auf den Heimweg  
        return true;  
    }
```

```
    if (rechtsFrei() && durchsucheTeilLabyrinthRechts()) {  
        kehrt(); // mach dich auf den Heimweg  
        return true;  
    }
```

```
    if (!vornFrei()) {
        kehrt(); // Sackgasse
        return false;
    }

    return durchsucheTeilLabyrinthVorne();
}

boolean durchsucheTeilLabyrinthLinks() {
    linksUm();
    vor();
    boolean gefunden = durchsucheLabyrinth();
    // Ausgangsposition einnehmen
    vor();
    linksUm();
    return gefunden;
}
```

```
boolean durchsucheTeilLabyrinthRechts() {  
    rechtsUm();  
    vor();  
    boolean gefunden = durchsucheLabyrinth();  
    // Ausgangsposition einnehmen  
    vor();  
    rechtsUm();  
    return gefunden;  
}
```

```
boolean durchsucheTeilLabyrinthVorne() {  
    vor();  
    boolean gefunden = durchsucheLabyrinth();  
    vor();  
    // Seiteneffekt: Hamster schaut in andere Richtung  
    return gefunden;  
}
```

```
// + Hilfsfunktionen
```

- Rekursive Funktionen: Funktionen, die während ihrer Abarbeitung erneut aufgerufen werden
- Backtracking-Verfahren: Problemlösungsverfahren, das rekursiv Teilwege durchforstet, um eine Lösung zu finden