Rekursion

- Sie wissen wie man Programme rekursiv entwickelt
- Sie kennen typische Beispiele von rekursiven Algorithmen
- Sie kennen die Vor-/Nachteile von rekursiven Algorithmen

Einführung

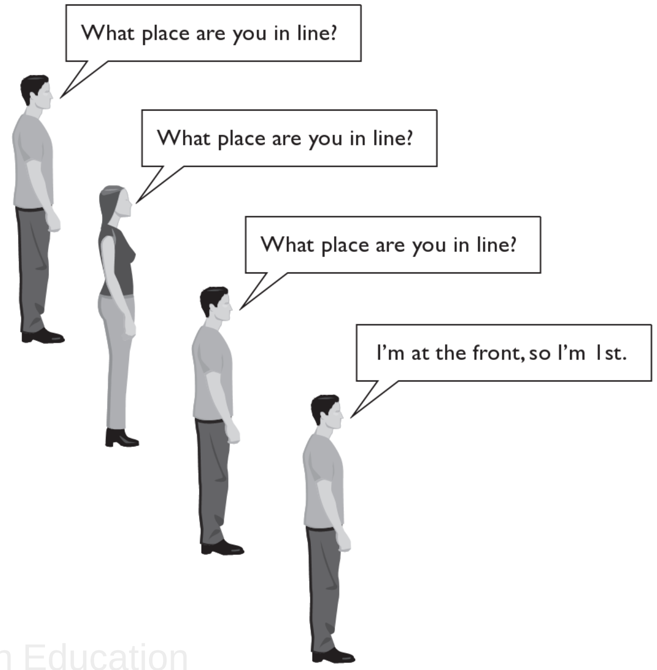
Rekursiver Algorithmus

- **Rekursiver Algorithmus**: Lösungsbeschrieb, der sich selber enthält
  - z.B. in der Mathematik sehr beliebt: Fakultät, Algorithmus nach Euklid

- Beispiel: an welcher Position in der Schlange stehe ich?
  - den Anfang der Schlange sieht man nicht

Rekursiver Algorithmus

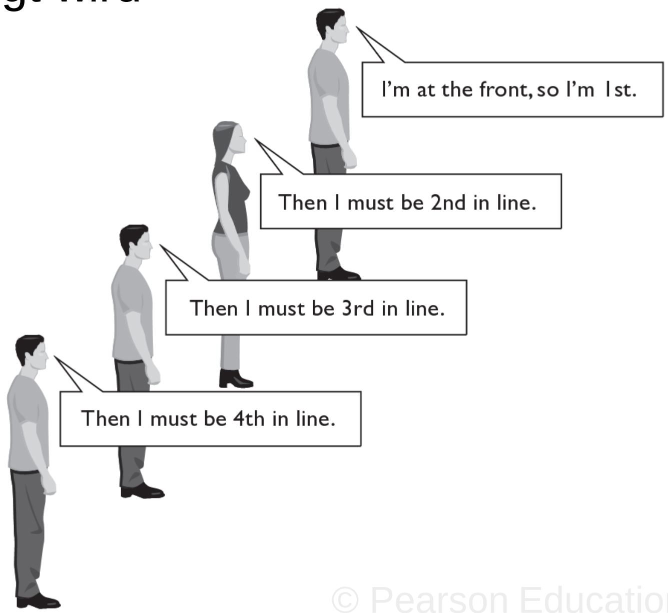
- 1. Frage Person vor dir, welche Position sie hat
  - falls sie zuvorderst steht, wird sie direkt antworten können
  - sonst fragt sie einfach die Person vor sich



Rekursiver Algorithmus

- 2. Sobald die Person an der ersten Stelle geantwortet hat
  - wird der zweitvordersten geantwortet usw.

- Essenz eines Rekursiven Algorithmus: Wiederholter Aufruf desselben Algorithmus/Methode, welche das Problem zum Teil löst und dann zu einem Ganzen zusammengefügt wird



iTempel vs Tempel

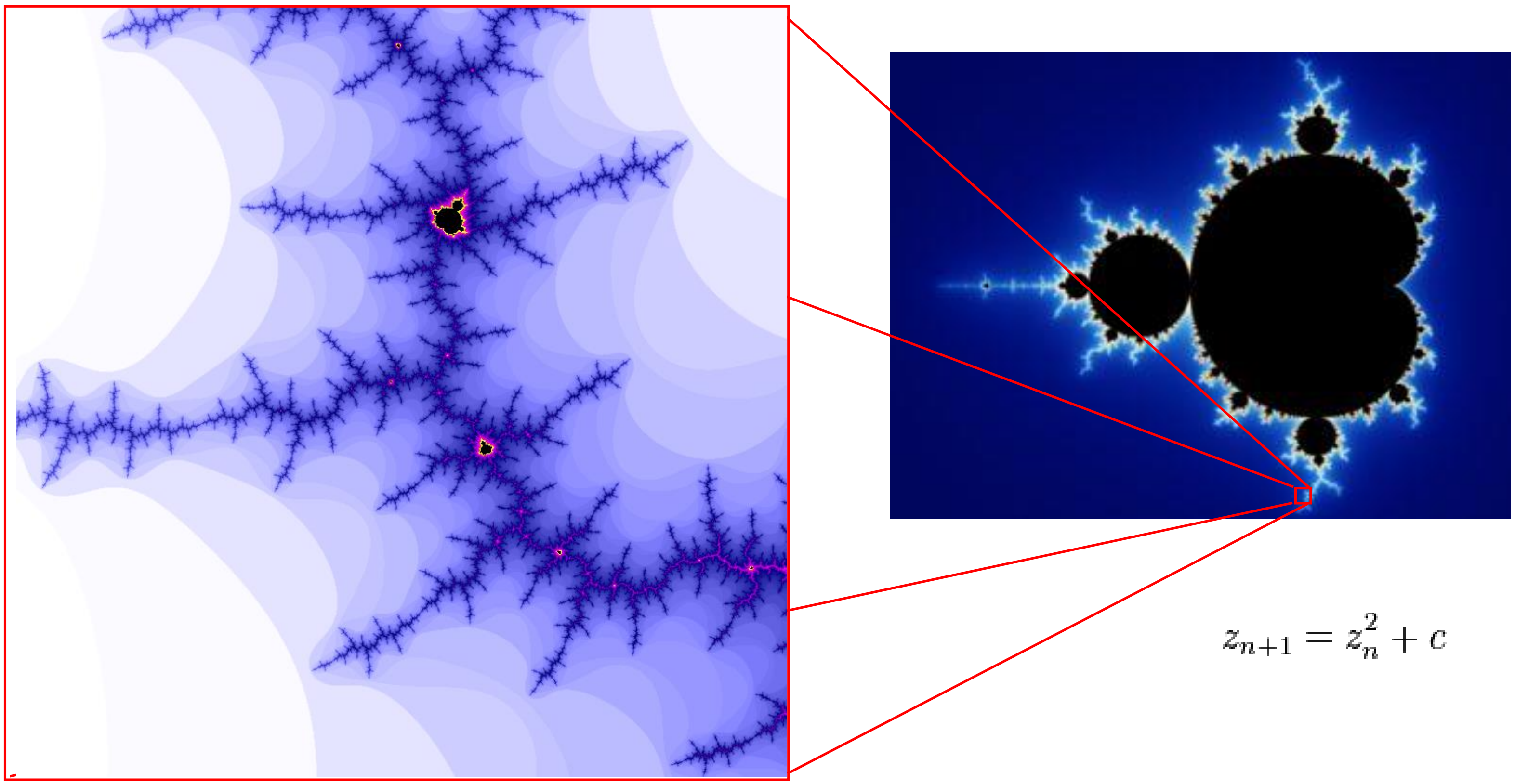
- Übereinstimmung der aktivierten Hirn-Regionen bei *überzeugten (!)* Apple-Benutzer und stark religiösen Menschen festgestellt

Beispiele

- Natur
  - Blätter des Farnstrauches, Küstenlinie, Fraktale Kurven
  - Schneeflocken
- Mathematik
  - positive Ganzzahl
    - *1 sei eine positive Ganzzahl*
    - *Der Nachfolger einer positive Ganzzahl ist wieder eine positive Ganzzahl*
  - Fakultät fak(n)
    - *fak(0) = 1*
    - *Wenn n >0, dann gilt fak(n) = n*fak(n -1)*
- Informatik
  - Liste kann als Sequenz oder rekursiv definiert werden
  - Baumstrukturen
    - *Ein Baum ist entweder leer*
    - *oder besteht aus einer Wurzel und zwei disjunkten Teilbäumen.*

Beispiel: Fraktale Kurven

- Figuren bei denen man beliebig hinein zoomen kann und immer wieder *ähnliche* Muster entdeckt: z.B. Mandelbrot's "Apfelmännchen"



$$z_{n+1} = z_n^2 + c$$

Beispiel der Fakultät

- n! = 1*2*3…(n-1)*n   : 1, 1, 2, 6, 24, 125, 720, 5040, 40320,

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

```
fak(3) = 3 * fak(2)
             2 * fak(1)
                 1 * fak(0)
                       1
                 1 * 1
             2 * 1
         3 * 2
           6
```

Fakultätsberechnung mit Rekursion

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

Abbruch

rekursiver Aufruf

Rekursive Algorithmen und Datenstrukturen

Rekursion

Definition:

> Ein Algorithmus/Datenstruktur heisst rekursiv definiert, wenn er/sie sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist.

- Vorteil der rekursiven Beschreibung ist die Möglichkeit, eine unendliche Menge durch eine endliche Aussage zu beschreiben
  - z.B. Objekt x enthält wieder Objekt x, Algorithmus a ruft sich selber auf
- In Java Programmen wird Rekursion durch Methoden implementiert, die sich selbst aufrufen
  - z.B. Methode p ruft Methode p auf

Eine rekursiv definierten Datenstruktur

Liste nicht rekursiv

- Liste = (ListNode)*

```
p = first;
while (p != null) {
        p = p.next;
}
```

Liste rekursiv definiert

- Liste = leer
- Liste = ListNode (Liste)?

```
void traverse(ListNode p) {
        if (p == null) // Abbruch
        else traverse(p.next);
};
```

RegEx Notation

- = definiert
- ()* : beliebig oft, 0..∞
- ()? : optional, 0..1

Übung

- Schreiben Sie eine rekursive Methode, welche die Elemente einer einfach verketteten Liste der Reihe nach ausgibt. Sie können direkt auf die Felder zugreifen.

```
ListNode
    int val;
    ListNode next;
}
```

- Schreiben Sie eine rekursive Methode, welche die Elemente einer **einfach** verketteten Liste in **umgekehrter** Reihenfolge ausgibt.

Generelle Vorlage für rekursive Programme

- Rekursive Programme sind das Programmäquivalent der **vollständigen Induktion**. Wesentlich ist somit, dass man zwischen zwei Fällen unterscheidet (wie bei den Beweisen):

- **1. Basis Fall** ("Verankerung"):
  - Man weiss z.B., dass fak(0) = 1 ist.

- **2. Allgemeiner Fall** ("Induktionsschritt"):
  - Für alle anderen Fälle (z.B. n>0) weiss man, dass sich die Lösung des Problems X n zusammensetzt aus einigen Operationen und einem Problem X n1 was **eine Dimension kleiner** als X n ist, z.B. fak(n) = n * fak(n1), n>0. Man zerlegt also das Problem für den allgemeinen Fall so lange, bis man auf den Basis Fall kommt.

Vorlage für rekursive Programme

- Damit muss eine allgemeine Vorlage für rekursive Programme diese **beiden Fälle unterscheiden.**
- Der **Basis Fall** stellt sicher, dass die rekursiven Programme endlich sind und **terminieren** (d.h. die Anzahl der rekursiven Aufrufe ist begrenzt).
- Vergisst man den Basis Fall, so werden im allgemeinen so viele rekursive Aufrufe durchgeführt, bis der Stack überläuft (Abbruch mit StackOverflow).

```
public int p(int n) {
     if (basecase) {
     // behandeln Basis Fall
     } else {
        p(n1); // behandeln allg. Fall
     }

}
```

führt sicher zum
Basis Fall, da jetzt
ein
kleineres Problem
gelöst werden soll

Übung

- Schleifen: Operationen werden endlich oft wiederholt

```
public void p()  {
    int i = 0;
    while (i < 10);
       System.out.println(i++);
    }
}
```

output

- Rekursion: Aufruf **p(0)**

```
public void p(int i)  {
    if(i < 10) {
       System.out.println(i);
       p(i+1)
    }
}
```

```
public void p(int i)  {
    if(i < 10) {
       p(i+1);
       System.out.println(i);
    }
}
```

Direkte/indirekte Rekursion

- Direkte Rekursion:
  - Bei der direkten Rekursion ruft eine Methode sich selber wieder auf.

```
public int p(int a) {

        int x = p(a1);

}
```

- Indirekte Rekursion:
  - Bei der indirekten Rekursion rufen sich 2 oder mehrere Methoden gegenseitig auf (ungewollte Fehlerquelle beim Programmieren!)

```
public int p(int a) {

        int x = q(a1);

}
```

```
public int q(int a) {

        int x = p(a-1);

}
```

Endrekursion (tail rekusion)-> Schleife

- Programm mit Rekursion

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

- Programm mit Iteration

```
int fak(int n) {
   if (n == 0) return 1;
   else {
     int res = n;
     while (n > 1) {
        n--;res = n* res;
     }
     return res;
   }
}
```

- Programme, bei denen der rekursive Aufruf die letzte Aktion im Else-Zweig bzw. allgemeinen Fall ist werden *endrekursiv* bezeichnet

- Endrekursive Programme lassen sich einfach in iterative Form überführen.

- Frage: lässt sich jedes Programm in eine nichtrekursive Form überführen?

Schleife -> Endrekursion

- Schleifen (Iterationen) lassen sich in Endrekursion überführen

```
void p(int i) {
   while ( <Bedingung>; i++ )
      <Anweisung>
}
```

```
void p1(int i) {
   if ( <Bedingung> ) {
      <Anweisung>; i++;
      while ( <Bedingung>, i++
)
         <Anweisung>
   }
}
```

```
void p2(int i) {
   if ( <Bedingung> ) {
      <Anweisung>; i++;
      if ( <Bedingung> ) {
         <Anweisung>; i++;
         while ( <Bedingung>;i++
)
            <Anweisung>
      }
   }
}
```

```
void pR(int i) {
   if ( <Bedingung> ) {
      <Anweisung>
      pR(i+1);
   }
}
```

Entwicklung von rekursiven Algorithmen

Hamster Beispiel 1

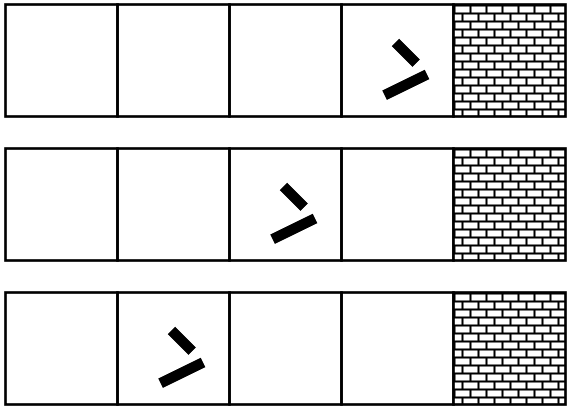
- Der Hamster soll bis zur nächsten Wand laufen!

Iterative Lösung:

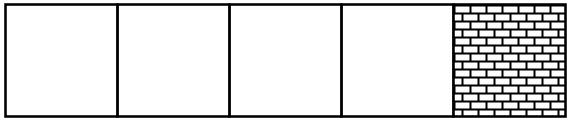
```
void zurMauer() {
  while (vorn_frei())
    vor();
}
```

Direkt rekursive Lösung:

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



Hamster Beispiel 2



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

Direkt rekursive Lösung:

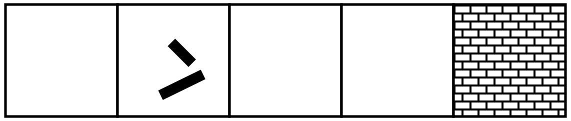
```
void hinUndZurueckR() {
  if (!vorn_frei()) {
    kehrt();
  }
  else{
    vor();
    hinUndZurueckR();
    vor();
  }
}
```

Hamster Beispiel 2 im Detail

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

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

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



Hamster Beispiel 3

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

Iterative Lösung:

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

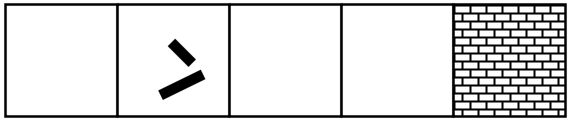
Rekursive Lösung:

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

Hamster Beispiel 3 im Detail

```
main:           aSR (1.)           aSR (2.)           aSR (3.)

i=aSR();        vorn_frei -> t
                vor();
                aSR()   -----> vorn_frei -> t
                               vor();
                               aSR()   -----> vorn_frei -> f
                                              return 0;
                                          0
                                       <-----
                               return 0 + 1;
                            1
                        <-----
                return 1 + 1;
              2
          <-----
i=2;
```

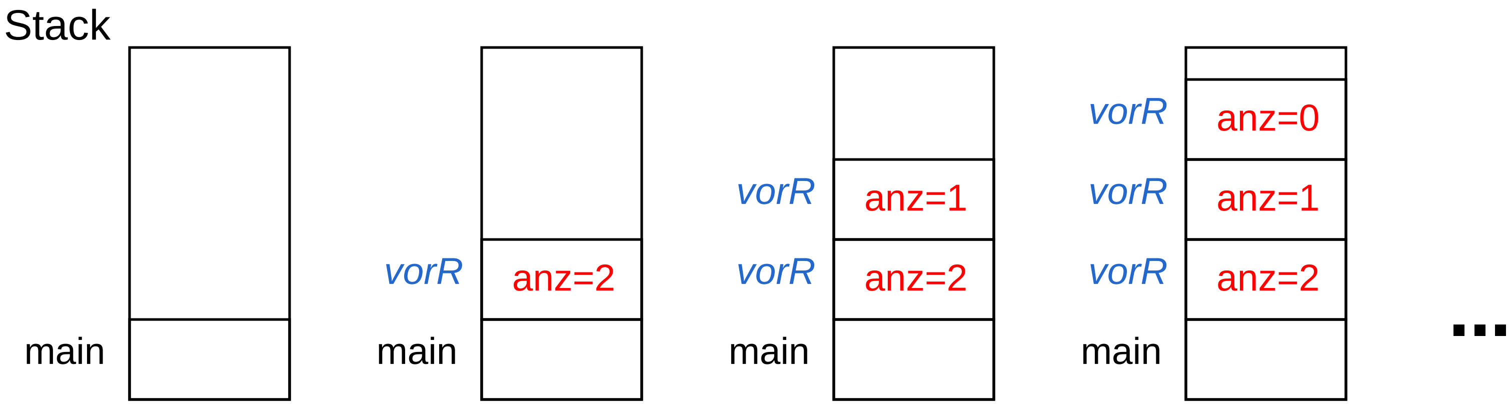


Hamster Beispiel 4

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

Parameter

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



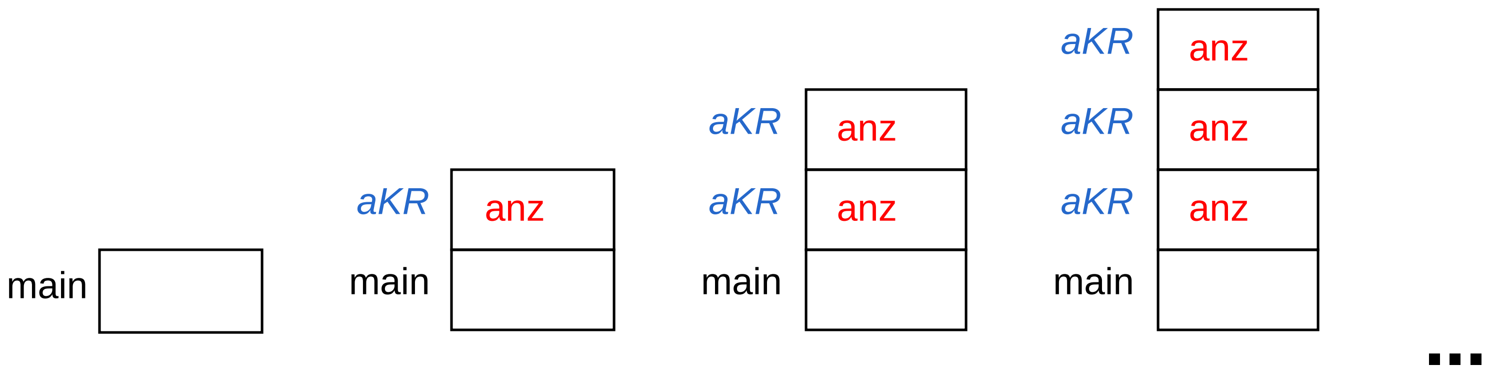
Hamster Beispiel 5

- Der Hamster soll die Anzahl Körner zählen

```
int anzahlKörnerR() {
   int anz;
   anz = körner();
   if (vorn_frei()) {
      vor();
      return anz + anzahlKörnerR();
   } else
      return anz;
}
```

lokale Variable

Stack



Hamster Beispiel 6: Endlosrekursion

- Der Hamster soll die Anzahl Körner zählen

falsch

```
int anz = 0;
void anzahlKörnerR() {
  if (vorn_frei()) {
    anzahlKörnerR();
    anz += körner();
    vor();
  }
}
```

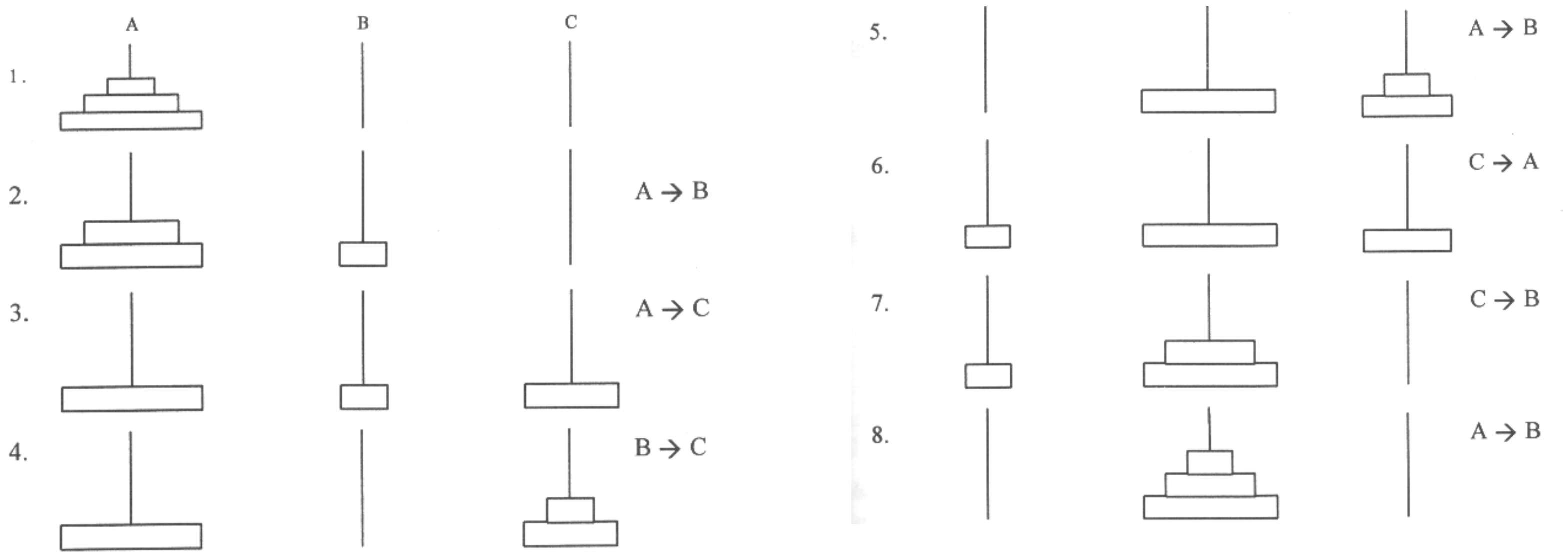
korrekt

```
int anz = 0;
void anzahlKörnerR() {
  anz += körner();
  if (vorn_frei()) {
    vor();
    anzahlKörnerR();
  }
}
```

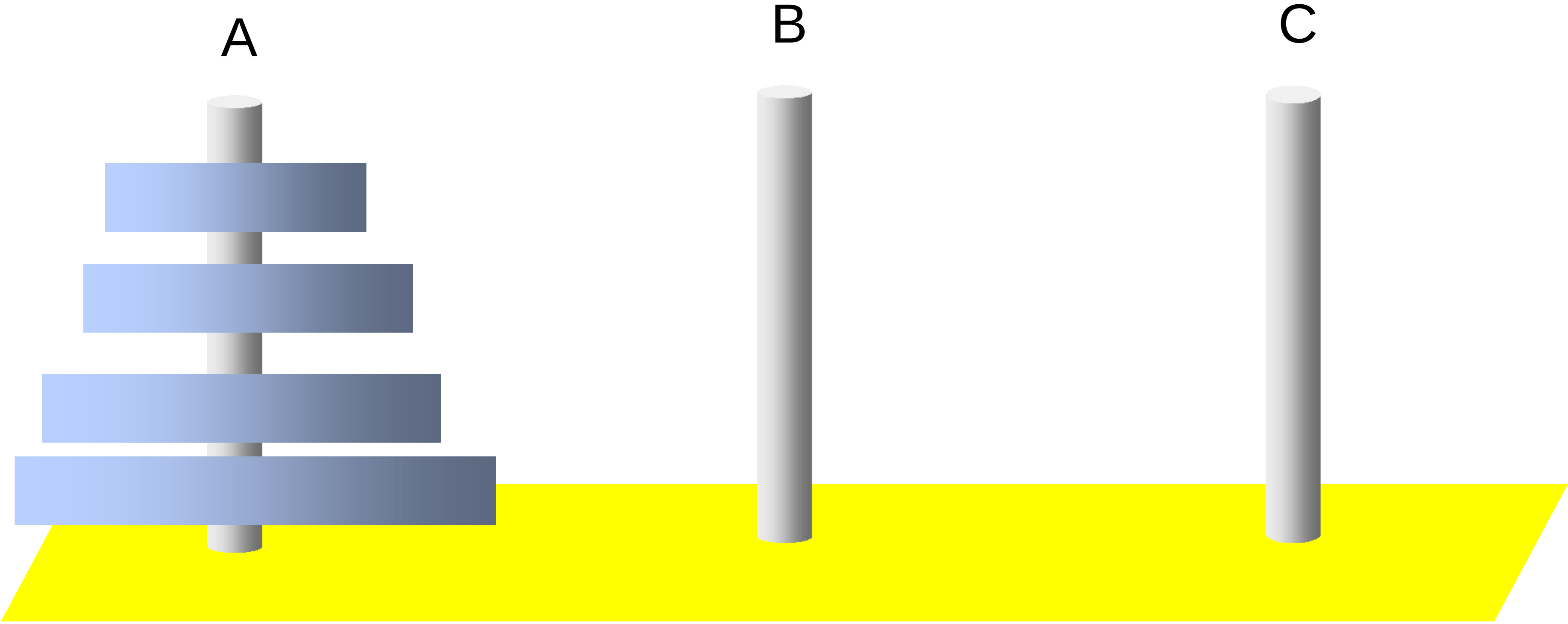
- Rekursionstiefe "unendlich"! erzeugt einen Laufzeitfehler: Stack Overflow!

Turm von Hanoi

- eine gegebene Anzahl von Scheiben unterschiedlicher Grösse soll von der Stange A nach B bewegt werden, ohne dass eine grössere auf eine kleinere zu liegen kommt.



Demo



Vorgehen

- Basisfall (n = 1)
  - bewege Scheibe von A nach B

- Lösung für (n = 2);
  - bewege kleinere Scheibe von A nach C (Hilfsstange)
  - bewege grössere Scheibe von A nach B
  - bewege kleinere Scheibe von C (Hilfsstange) nach B

- Lösung für allgemeine n
  - bewege Stapel (n-1) von A nach C
  - bewege grösste Scheibe von A nach B
  - bewege Stapel (n-1) von C nach B

Programm Beschreibung

```
void hanoi (int n, char from, char to, char help) {
     if (n == 1) {
         // bewege von from nach to
     }
     else {
         // bewege Stapel n-1 von from auf help
         // bewege von from nach to
         // bewege Stapel n-1 von help auf to
     }
}
```

weitere Vereinfachung

```
void hanoi (int n, char from, char to, char help) {
     if (n > 0) {
         // bewege Stapel n-1 von from auf help
         // bewege von from nach to
         // bewege Stapel n-1 von help auf to
     }
}
```

Java Programm

```
void hanoi (int n, char from, char to, char help) {
     if (n > 0) {
         // bewege Stapel n-1 von from auf help
         hanoi(n-1,from,help,to);
         // bewege von from nach to
         System.out.println("bewege " + from + " nach " + to);
         // bewege Stapel n-1 von help auf to
         hanoi(n-1,help,to,from);
     }
}


main {
   hanoi (3,"A","B","C");
}
```

Rekursionstiefe, Speicherkomplexität, Zeitkomplexität

- **Rekursionstiefe**:
  - Maximale "Tiefe" der Aufrufe einer Methode minus 1
  - hanoi(3) -> hanoi(2) -> hanoi(1) -> hanoi(0) : Rekusionstiefe 3
- **Zeitkomplexität:** Rechenaufwand
  - Aufwand für n: 1 + 2 * (n-1)
  - n-1: 1 + 2 * (n-2)
  - n-2: 1 + 2 * (n-3)
  - ……
  - 2 * 2 * 2 * (n mal)… .
  - d.h. Verdoppelung mit jedem Schritt bestimmt -> ~ 2^n
- $O(2^n)$ -> **Aufwand ist exponentiell**
- **Speicherkomplexität:** benötigter Speicher

Beispiel: Fibonacci-Zahlen

$$\text{fib(n)} = \begin{cases} 0 & \text{falls n = 0} \\ 1 & \text{falls n = 1} \\ \text{fib(n-1) + fib(n-2)} & \text{sonst} \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	..
fn	0	1	1	2	3	5	8	13	21	34	55	89	144	..

```
public int fib(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

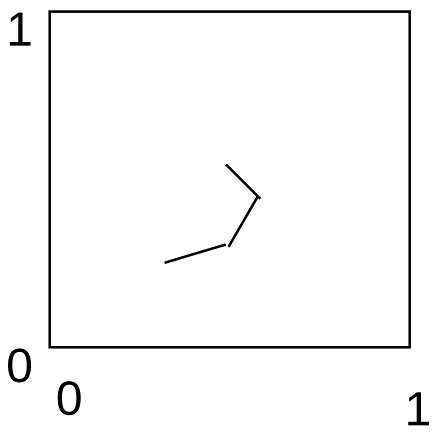
Übung

- Von welcher Ordnung ist dieser Algorithmus?

- Berechnen Sie die Fibonacci Zahlen iterativ

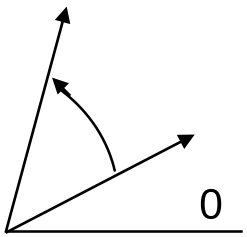
Beispiel: Rekursive Kurven

Einschub: Schildkröten-Graphik



"Turtle" bewegt sich vorwärts

"Turtle" dreht sich um Winkel



in Grad

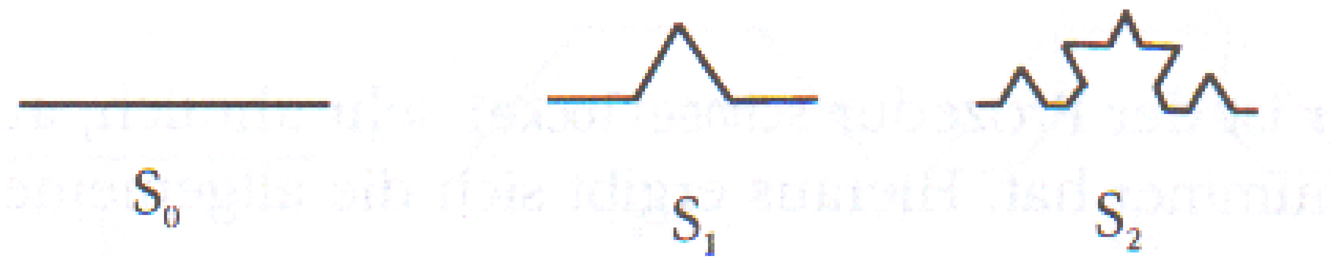
+ Gegenuhrzeiger

0 bei 3 Uhr

```
class Turtle
    double x, y;
    bewege(double distanz);
    drehe(double winkel):
}
```

Schneeflockenkurve

- die Strecke wird dreigeteilt; der mittlere Teil wird durch die zwei Seiten eines gleichseitigen Dreiecks ersetzt



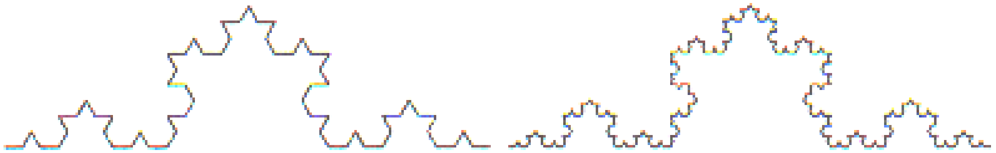

```
turte.strecke(dist);
```

```
dist = dist/3;
turtle.bewege(dist);
turtle.drehe(60);
turtle.bewege(dist);
turtle.drehe(-120);
turtle.bewege(dist);
turtle.drehe(60);
turtle.bewege(dist);
```

Java Programm für Schneeflocke

```
void schneeflocke(int stufe, double dist) {
if (stufe == 0) {
     turtle.bewege(dist)
} else {
     stufe--;
     dist = dist/3;
     schneeflocke(stufe,dist);
     turtle.drehe(60);
     schneeflocke(stufe,dist);
     turtle.drehe(-120);
     schneeflocke(stufe,dist);
     turtle.drehe(60);
     schneeflocke(stufe,dist);

}
```



Rekursive Methoden Zusammenfassung

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

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

- Nachteile rekursiver Algorithmen:
  - z.T. weniger effizientes Laufzeitverhalten (Overhead beim Methodenaufruf)
  - Konstruktion rekursiver Algorithmen "gewöhnungsbedürftig"