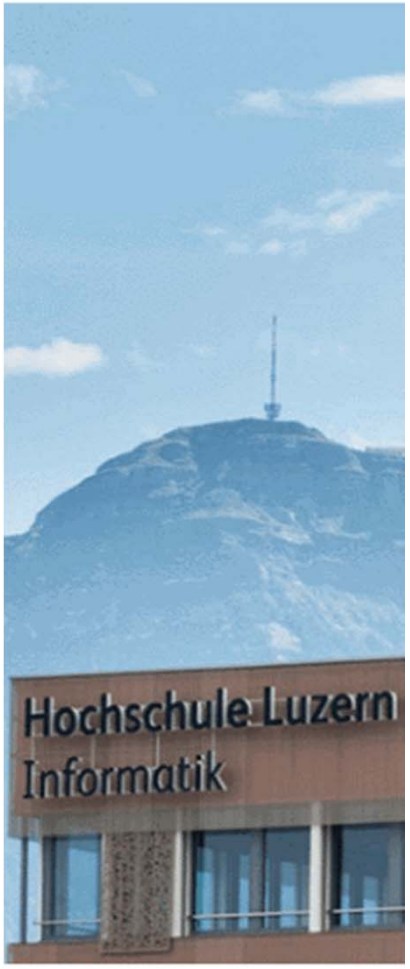
Algorithmen & Datenstrukturen

Rekursion

Hansjörg Diethelm

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN
INFORMATIK



Inhalt

- Grundlegendes zur Rekursion
- Iteration vs. Rekursion am Beispiel der Fakultätsberechnung
- Beispiel Fibonacci-Zahlen
- Rekursions-Typen
- Beispiel Permutationen
- Beispiel «Türme von Hanoi»

Lernziele

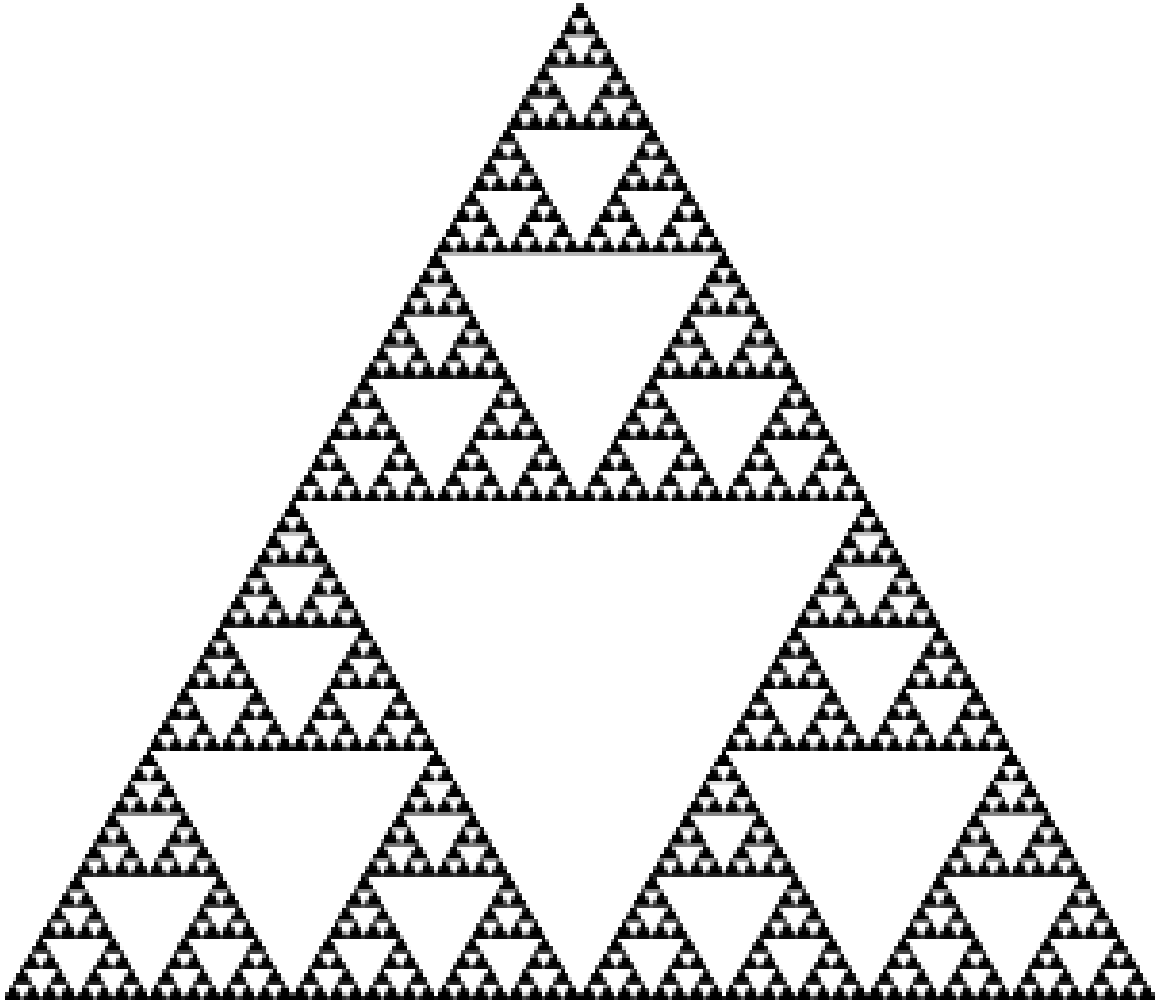
Sie …

- können beschreiben, was Algorithmen und Datenstrukturen mit Selbstähnlichkeit und Selbstbezug zu tun haben.
- können bei einer rekursiven Methode Rekursionsbasis und Rekursionsvorschrift identifizieren.
- können gut nachvollziehbar aufzeichnen, wie eine rekursive Methode abgearbeitet wird.
- können beschreiben, wozu Heap und Call Stack dienen.
- können die Eigenheiten der Rekursion (vs. Iteration) beschreiben.
- können einfache rekursive Methoden implementieren.

Grundlegendes zur Rekursion

https://www.youtube.com/watch?v=qqRiZWGwk-A

Was fällt auf?



Sierpinski-Dreieck in der Natur

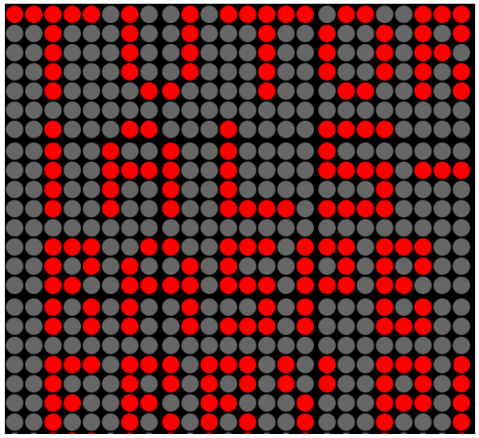
Muschel
Quelle: Max-Planck-Institut für Entwicklungsbiologie

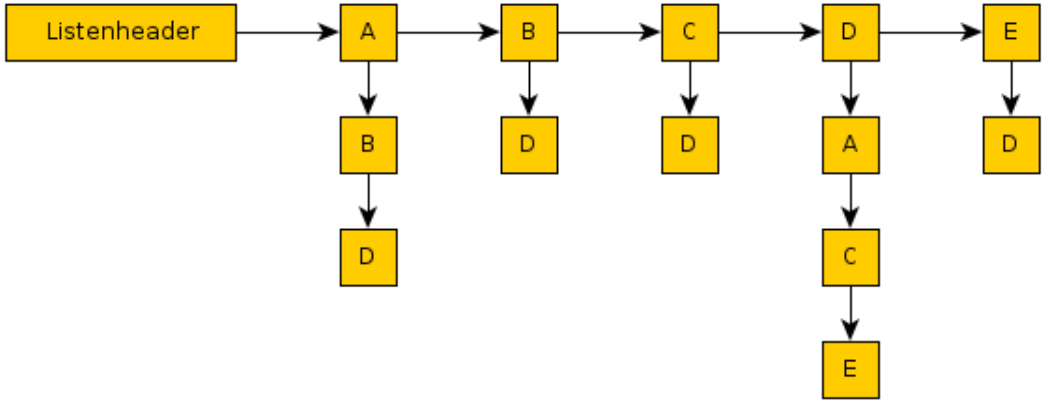
Romanesco
http://de.wikipedia.org/wiki/Datei:Romanesco.jpg

Grundlegendes zur Rekursion

Viele Algorithmen und Datenstrukturen sind von Natur aus selbstähnlich bzw. selbstbezüglich:

- Der ggT von (21, 15) ist gleich dem ggT von (21-15, 15).
- Ein Verzeichnis enthält Dateien und andere Verzeichnisse.
- Ein Ausschnitt einer Matrix, einer Liste, eines Baumes, eines Graphen ist wieder eine Matrix, eine Liste, ein Baum, ein Graph.





Iteration vs. Rekursion
am Beispiel der Fakultätsberechnung

Iterative Fakultätsberechnung

- Iterative Definition der Fakultät n!:

  n! = 1 · 2 · 3 · 4 · ... · n (wiederholtes Multiplizieren)

```
public static int factorialIter(final int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

- Die Fakultät wird hier mit Hilfe einer Schleife

  ➔ **iterativ** berechnet.

Rekursive Fakultätsberechnung

Rekursive Definition der Fakultät n!:

- Für mindestens ein einfaches Problem wird eine direkte Lösung angegeben; hier für 0! und 1!:

  0! = 1

  1! = 1 ➔ **Rekursionsbasis**

  Nur so kann die Rekursion terminieren.

- Es wird ein Lösungsweg aufgezeigt, wie das allgemein schwierige Problem auf ein gleichartiges, aber etwas einfacheres Problem zurückgeführt werden kann; hier n! auf (n-1)!:

  n! = n · **(n-1)!** ➔ **Rekursionsvorschrift**

Vgl. Selbstähnlichkeit von n! mit (n-1)! sowie lateinisch: recurrere = zurücklaufen

Rekursive Berechnung von 5!

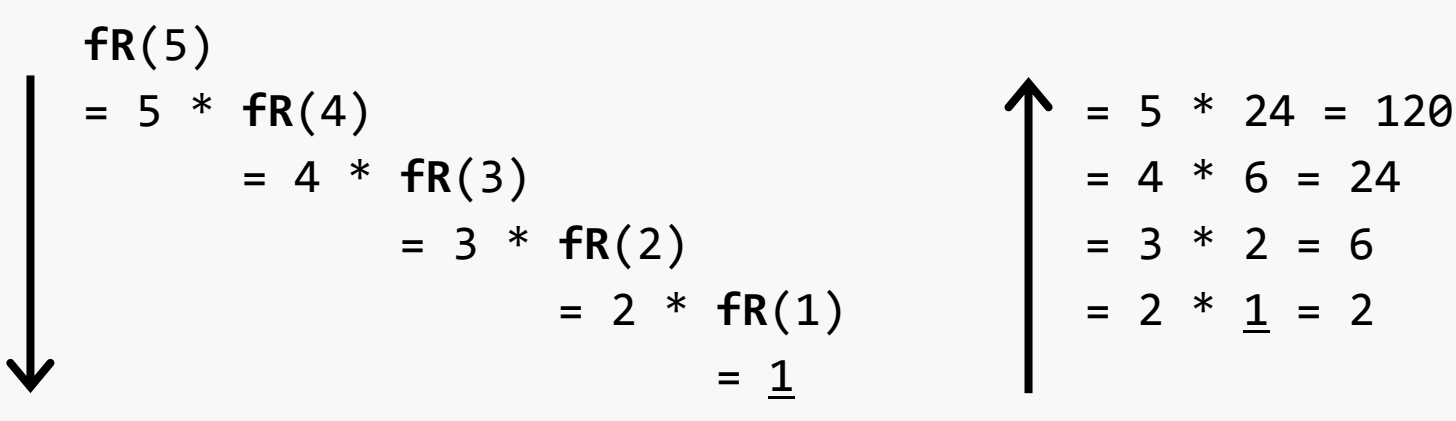
- Gesucht: **5!** = ?
- Wird zurückgeführt auf 5 · **4!**
- Wird zurückgeführt auf 5 · (4 · **3!**)
- Wird zurückgeführt auf 5 · (4 · (3 · **2!**))
- Wird zurückgeführt auf 5 · (4 · (3 · (2 · **1!**)))
- Es ist definiert: 1! = 1

- Damit berechnet sich:
  5! = 5 · (4 · (3 · (2 · 1))) = 5 · 4 · 3 · 2 · 1 = 120
  Also auch durch wiederholtes Multiplizieren, aber implizit via Rekursion.

Rekursive Methode

```
public static int factorialRec(final int n) {
    if ((n == 0) || (n == 1)) {  // Rekursionsbasis
        return 1;                // Rekursionsbasis
    } else {                                 // Rek'Vorschrift
        return (n * factorialRec(n - 1));  // Rek'Vorschrift
    }
}
```

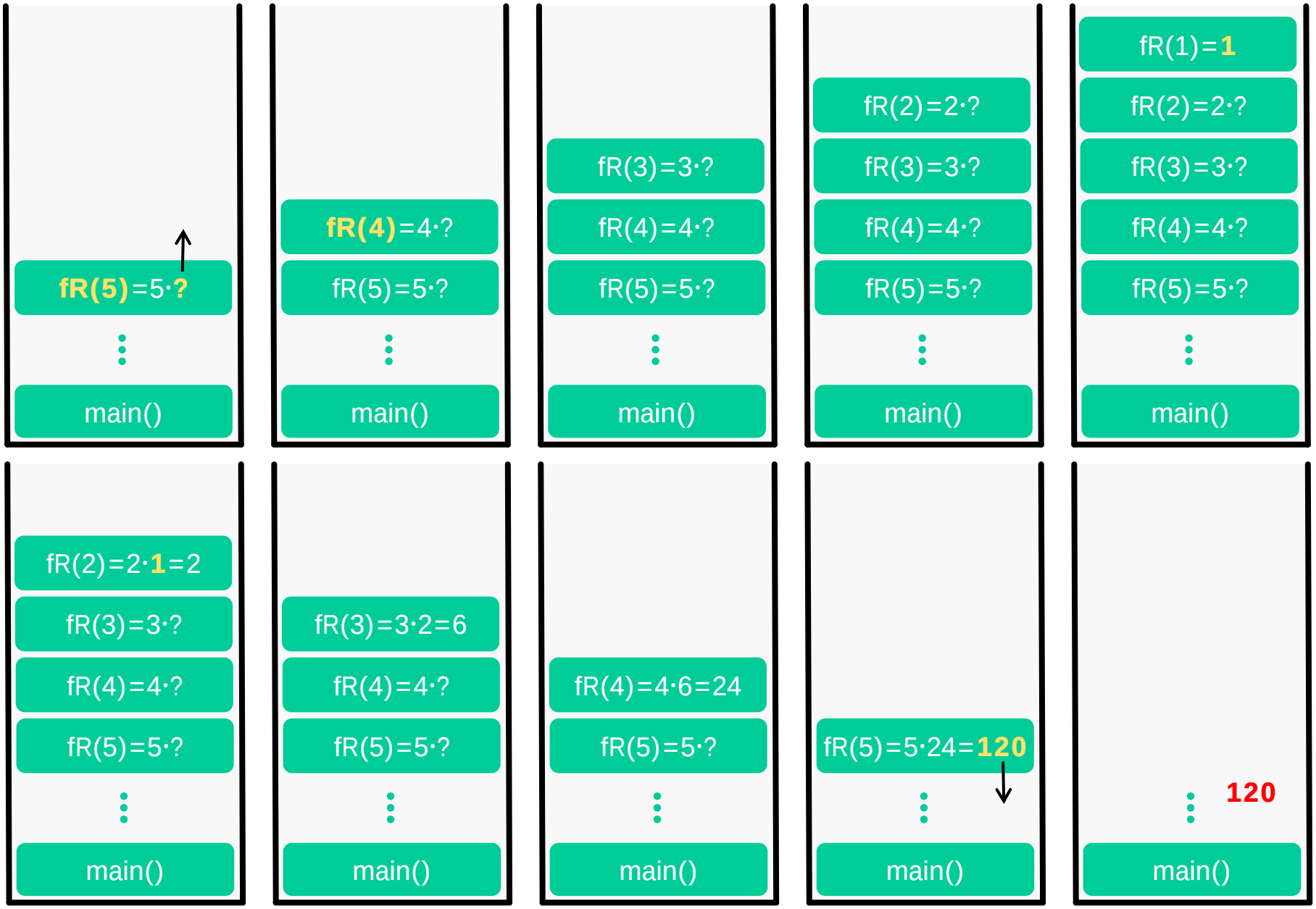
- Die Fakultät wird hier ➔ **rekursiv** berechnet, indem sich die Methode selber wieder aufruft.



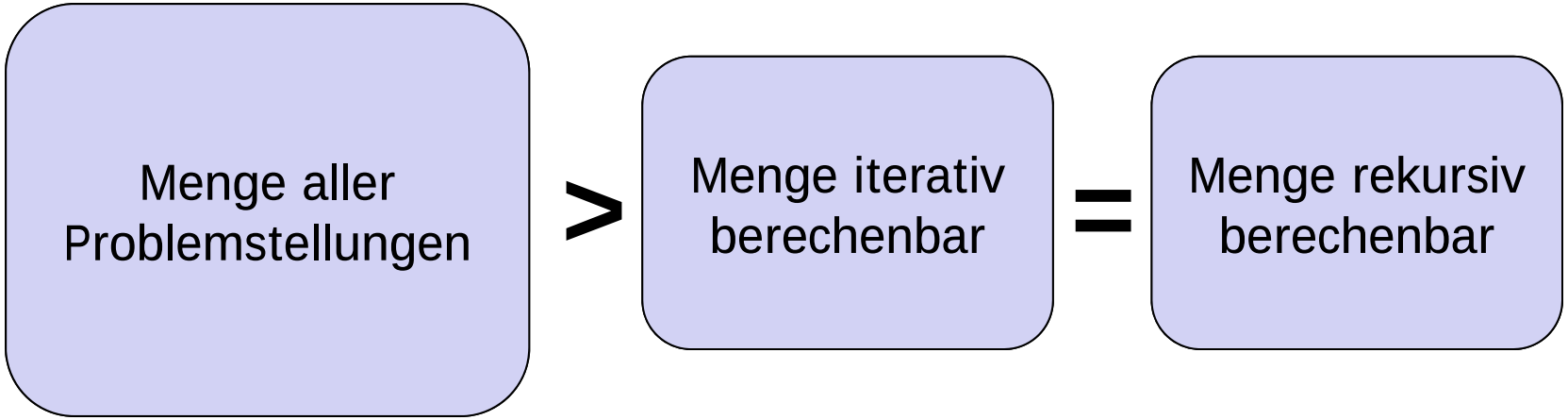
Call Stack https://www.youtube.com/watch?v=450maTzSIvA

- Für die Ausführung eines Programmes verwendet die Java Virtual Machine (JVM) zwei wichtige Speicher: ➔ **Heap** und ➔ **Call Stack**
- **Heap**: In diesem Speicherbereich werden die Objekte gespeichert, d.h. deren Instanzvariablen bzw. Zustände. Nicht mehr referenzierbare Objekte werden durch den Garbage Collector (GC) automatisch gelöscht.
- **Call Stack**: Letztendlich wird bei der Ausführung eines Java-Programms «eine Kette» von Methoden aufgerufen bzw. abgearbeitet. Ursprung ist die `main()`-Methode. Jeder Methodenaufruf bedingt gewissen Speicher, insbesondere für ihre aktuellen Parameter und lokalen Variablen. Dazu dient der Call Stack. Ein neuer Methodenaufruf bewirkt, dass der Call Stack wächst bzw. darauf ein zusätzlicher ➔ **Stack Frame** angelegt wird.

Der Call Stack in Aktion



Mächtigkeit der Rekursion



- Rekursion und Iteration sind ➔ **gleich mächtig**.
- D.h. die Menge der berechenbaren Problemstellungen bei Verwendung der Rekursion und Verwendung der Iteration ist gleich.
- D.h. eine rekursive Implementation lässt sich grundsätzlich immer in eine gleichwertige iterative Implementation umprogrammieren und umgekehrt.

Hinweis: Dies gilt exakt nur für sogenannte primitiv-rekursive Probleme, d.h. bei linearer und nicht geschachtelter Rekursion bzw. bei reinen Zählschleifen.

Motivation für die Rekursion

- mächtiges Lösungskonzept
- häufig einfache und elegante Problemlösungen
- weniger Quellcode
- Korrektheit häufig einfacher zu zeigen
- z.B. kennen die Programmiersprachen
  - **Lisp** (List Processing) und
  - **Prolog** (Programming in Logic)

  nur Rekursion!

Tücken der Rekursion

- schnell sehr viele Methodenaufrufe
- tendenziell langsamere Programmausführung
- grosser Speicherbedarf auf dem Call Stack
- Gefahr eines ➔ **Stack Overflow**!

→ Bei Wahlmöglichkeit eine iterative Implementierung bevorzugen!

Beispiel Fibonacci-Zahlen

https://www.youtube.com/watch?v=iPKUe-69PdA

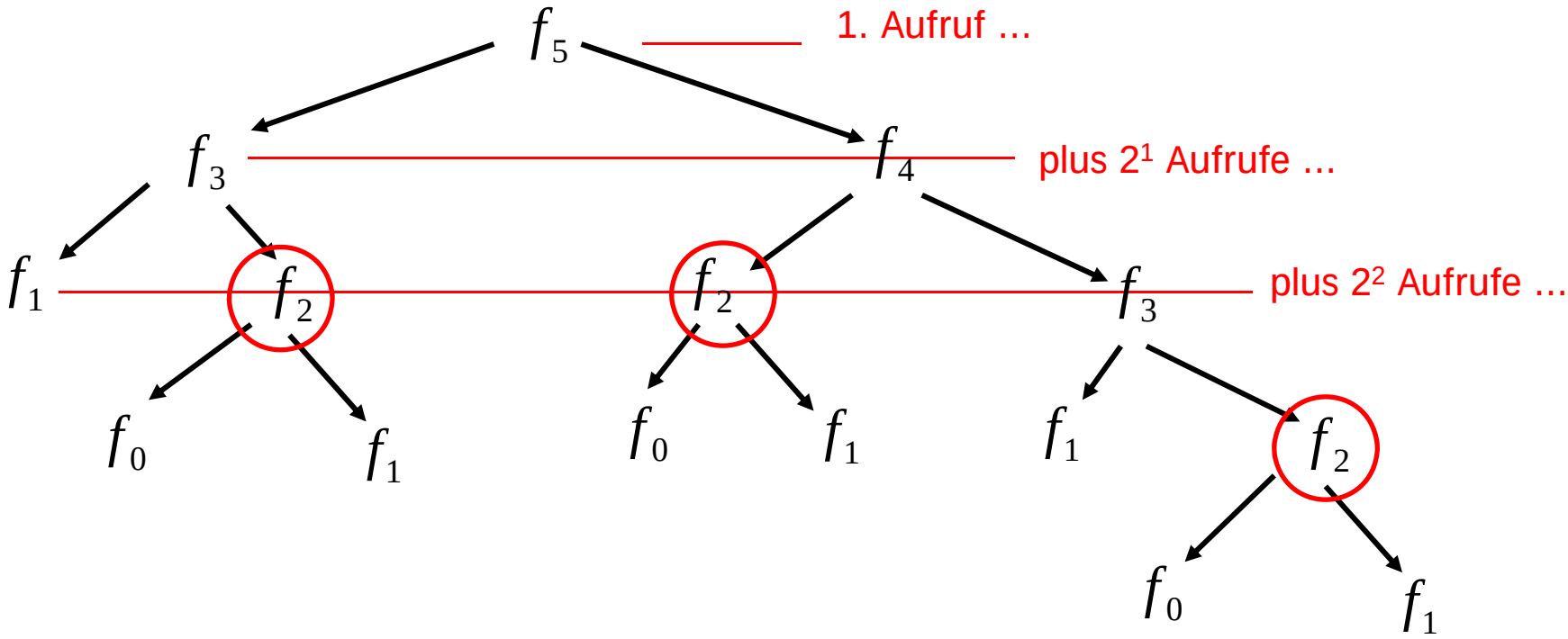
Fibonacci-Zahlen

- Fibonacci-Zahlen sind in der Mathematik wie folgt rekursiv definiert:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{für} \quad n \geq 2$$

- Erste Idee: Implementation einer **rekursiven Methode**, welche die mathematische Definition 1 zu 1 übernimmt.
- Aber: Eine genauere Betrachtung des Algorithmus zeigt, dass diese rekursive Berechnung **komplex** ist!

Fibonacci-Zahlen: Beispiel f_5



- Mehrfache Berechnung von denselben Fibonacci-Zahlen.
- Viele rekursive Aufrufe: Komplexitätsklasse **$O(2^n)$**
- Suche nach iterativen Lösungen notwendig.
- **Allgemein**: Nicht jedes Problem, das sich rekursiv präsentiert, sollte rekursiv umgesetzt werden!

Rekursions-Typen

Rekursion-Typen (1)

- **Lineare Rekursion:** Die Ausführung der Methode `m(...)` führt zu **höchstens einem** rekursiven Aufruf: `m(...)` → `m(...)`
  z.B. Fakultät: 5! → 4! → 3! → 3! → 2! → 1! → 1

- **Nichtlineare Rekursion:** Die Ausführung der Methode `m(...)` führt zu **mehr als einem** rekursiven Aufruf.

  - **nicht geschachtelt** , d.h. ➔ **primitiv rekursiv**:
    `m(...)` → `m(...), m(...)`
    z.B. Fibonacci (vgl. oben): $f_5 \rightarrow f_3, f_4$ ....

  - **geschachtelt** , d.h. **nicht primitiv rekursiv**:
    `m(...)` → `m(m(...)...)` (vgl. «Ackermann-Funktion»)

Rekursion-Typen (2)

- **Direkte Rekursion**: Eine rek. Methode ruft sich **direkt** selbst auf.
- **Indirekte Rekursion**: Eine rekursive Methode ruft sich **indirekt** selbst auf, .B.

```
public static boolean isEven(final int n) {
    if (n == 0) {
        return true;
    } else {
        return isOdd(n - 1);
    }
}
 public static boolean isOdd(final int n) {
    if (n == 0) {
        return false;
    } else {
        return isEven(n - 1);
    }
}
```

Beispiel Permutationen

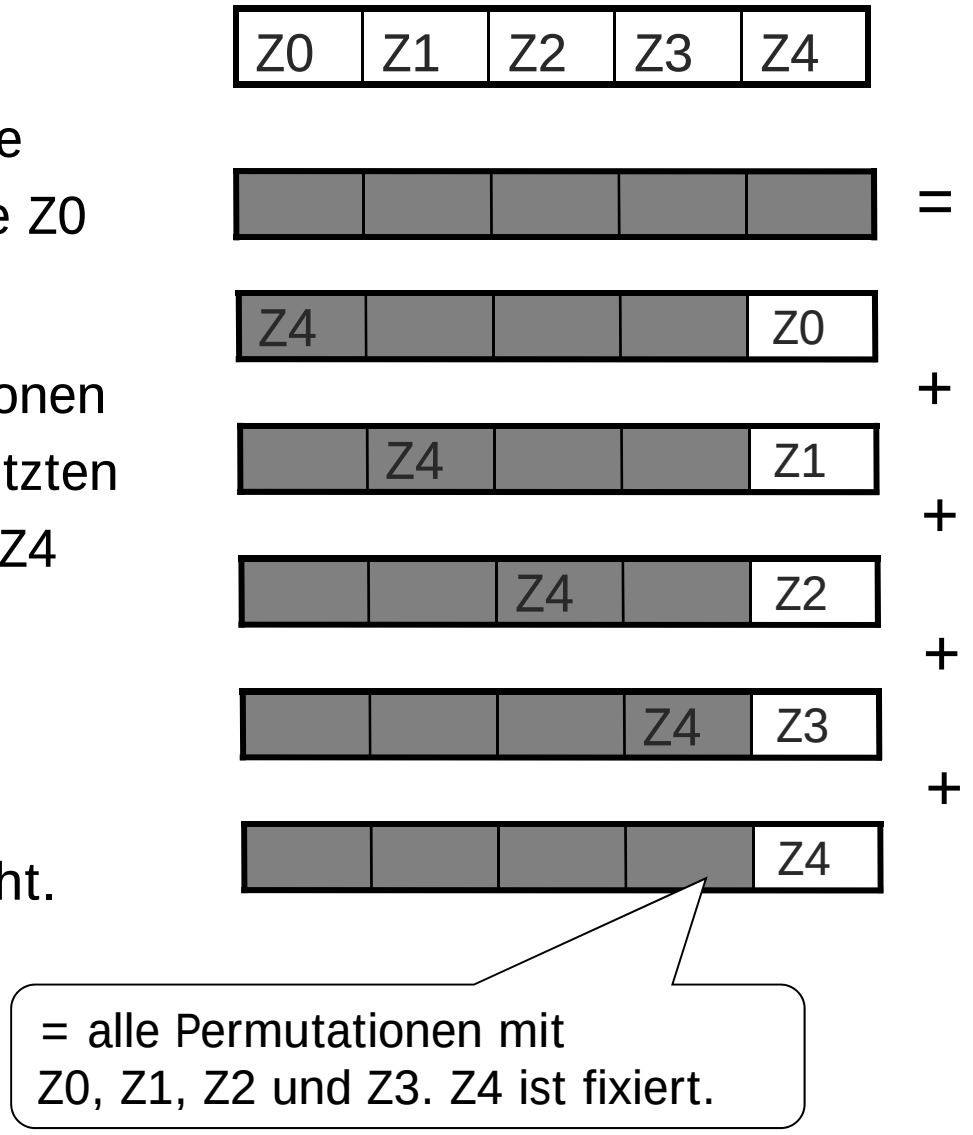
Passwort Cracking

- Jemand hat sein Passwort vergessen. Es war sechs Zeichen lang. Im Passwort kamen nur die Buchstaben von A bis F vor und jeder Buchstabe nur einmal. Wie viele verschiedene Passwörter bzw. Buchstaben-Kombinationen muss er/sie durchprobieren?
- Mit anderen Worten: Gefragt sind alle **Permutationen** der Buchstaben A bis F?
- Also $6 \cdot 5 \cdot 4 \cdot 3 \cdot \ldots \cdot 1 =$ **6! Permuationen**
- Im Falle von nur drei Buchstaben A, B und C ergeben sich diese Permutationen: ABC, BAC, CBA, BCA, ACB, CAB.

- Eine entsprechende Methode soll alle Permutationen ausgeben.

Rekursive Lösungsstrategie

- Die 5 grauen Felder stellen alle Permutationen der 5 Elemente Z0 ... Z4 dar.
- Dies entspricht den Permutationen über 4 Felder, wobei an der letzten Position der Reihe nach Z0 ... Z4 eingesetzt werden.
- Nach diesem Schema geht es weiter, bis es nur noch um Permutationen über 1 Feld geht. Dies ist dann trivial (vgl. Rekursionsbasis).



Implementation

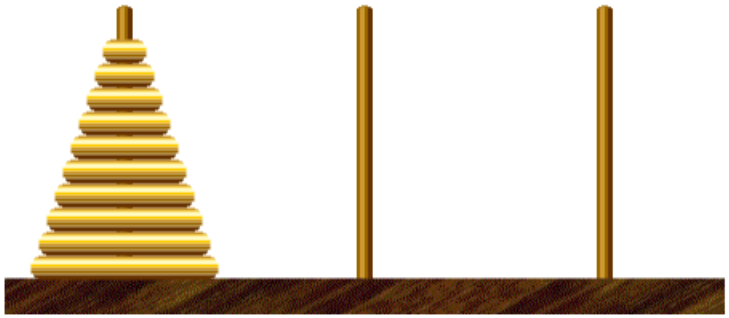
```
private static void exchange(final char[] a, final int i, final int j) {
    char tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

```
public static void permute(final char[] a, final int endIndex) {
    if (endIndex == 0) {   // Rekursionsbasis
        ausgabe(a);
    } else {    // Rekursionsvorschrift
        for (int i = 0; i <= endIndex - 1; i++) {
            exchange(a, i, endIndex);
            permute(a, endIndex - 1);
            exchange(a, i, endIndex);
        }
        permute(a, endIndex - 1);
    }
}
```

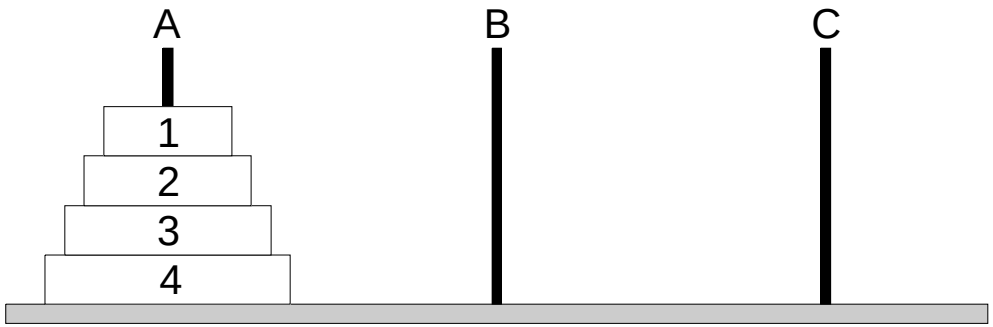
```
public static void main(String[] args) {
    char[] feld = {'A', 'B', 'C'};
    permute(feld, feld.length - 1);
}
```

Beispiel «Türme von Hanoi»

https://www.mathematik.ch/spiele/hanoi_mit_grafik



Problemstellung

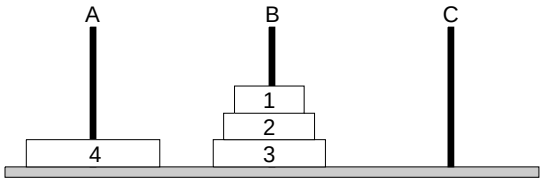


- Gemäss einer alten Geschichte:
  - Mönche müssen 64 goldene Scheiben (oben sind es nur 4) vom linken Ausgangspfosten **A** zum rechten Zielpfosten **C** verschieben.
  - Der mittlere Pfosten **B** dient als Hilfspfosten.
  - Aufs Mal darf nur 1 Scheibe zu einem anderen Pfosten verschoben werden.
  - Nie darf eine grössere Scheibe auf einer kleineren zu liegen kommen.
  - Nach getaner Arbeit ist das Ende der Welt gekommen …

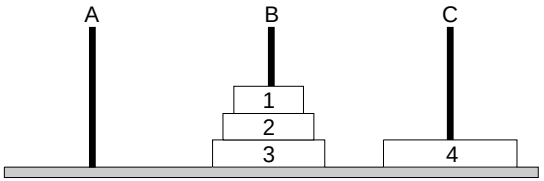
Rekursive Lösungsstrategie – eigentlich ganz einfach

moveDisks(...):

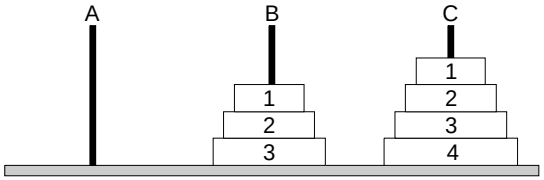
- **Rekursionsbasis**: Falls die Turmhöhe nur n = 1 ist, kann die Scheibe einfach von A nach C verschoben werden – fertig!
- **Rekursionsvorschrift**: Falls die Turmhöhe grösser 1 ist, dann …
  1. Verschiebe mit **moveDisks(...)** n-1 Scheiben von A zum Hilfspfosten B.
  2. Verschiebe die letzte Scheibe (d.h. die verbliebene grösste) vom Ausgangspfosten A direkt zum leeren Zielpfosten C.
  3. Verschiebe mit **moveDisks(...)** die n-1 Scheiben vom Hilfspfosten B auch noch zum Zielpfosten C.



1.



2.



3.

Implementation

```
public static void moveDisks(String from, String via, String to, int n)
{
    if (n == 1) {
        System.out.println("move disk from " + from + " to " + to);
    } else {
        moveDisks(from, to, via, n - 1);
        System.out.println("move disk from " + from + " to " + to);
        moveDisks(via, from, to, n - 1);
    }
}

public static void main(String[] args) {
    moveDisks("A", "B", "C", 4);
}
```

```
move disk from A to B
move disk from A to C
move disk from B to C
move disk from A to B
usw.
```

Zusammenfassung

- Selbstähnlichkeit bzw. Rekursion findet sich bei Algorithmen und Datenstrukturen.
- Iteration und Rekursion sind zentrale und praktisch gleich mächtige Konzepte in der Programmierung.
- Rekursive Lösungen sind häufig elegant und einfach, haben aber ihre Tücken!
- Bei einer rekursiven Methode unterscheidet man zwischen Rekursionsbasis und Rekursionsvorschrift.
- Erstere ist für den Rekursionsabbruch da, letztere beinhaltet die eigentliche Rekursion.
- Auf dem Heap werden die Objekte gespeichert und auf dem Call Stack die Methoden gespeichert bzw. abgearbeitet.

Fragen?