Informatik 1

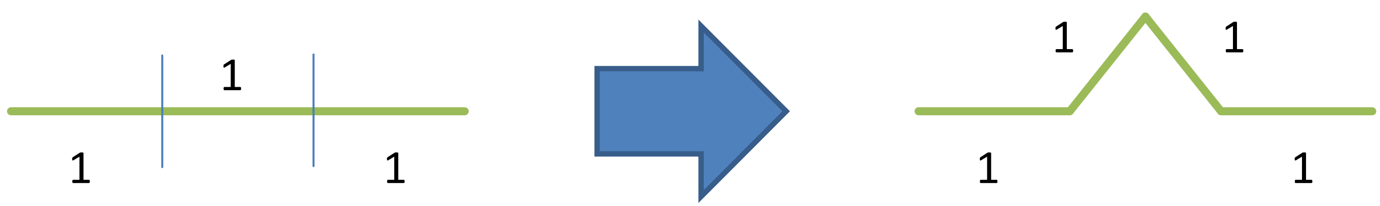
Vorlesungsfolien

Rekursion

- Definition
- Rekursive Programme
  - Fakultät
  - Fibonaccizahlen
- Problemlösung mit Rekursion
  - Türme von Hanoi
  - Rekursives Backtracking
    - Springerproblem

Definition

- "Zurücklaufen" (lat. recurrere)
- Rekursive Strukturen:
  - Strukturen, die im kleinen sich selbst enthalten oder ähnlich sind.
- Rekursiver Algorithmus:
  - Algorithmus, der sich selbst aufruft.
- Beispiel: **Kochkurve**
  - Rekursive Struktur, die mit einem rekursivem Algorithmus erzeugt werden kann
  - Zu Beginn: Eine Kante vorhanden
  - Algorithmus: Ersetzte jede Kante in der Struktur durch vier neue Kanten wie unten skizziert. Wiederhole dies unendlich oft.



In drei gleichlange Strecken teilen

Teilstrecke in der Mitte durch zwei Strecken, die ein Dreieck bilden ersetzen

Definition

- Ergebnis: Wenn mit einem Quadrat angefangen wird
  - Eine Kurve im zwei-dimensionalen Raum mit unendlicher Länge, aber endlicher Ausdehnung in der Fläche
  - Teile der Kurve sind identisch mit der ganzen Kurve
- Rekursiver Algorithmus:

```
Start mit  K := { ___ }

berechneKochkurve( K ):
    K' := {};
    Für jedes ___ aus K füge _/\_ in K' ein
        berechneKochkurve( K' )
```

- Dieser Algorithmus terminiert nicht.
- Rekursive Algorithmen sollten immer abbrechen: <u>Rekursionsabbruch</u> nötig.

Definition

- Rekursive Funktionen in der Mathematik:
  - Funktion, deren Definition die Funktion selbst enthält.
- Ein <u>Funktion</u> von einer Menge A (Definitionsbereich) zu einer Menge B ist eine Zuordnungsvorschrift, die jedem Wert von A genau einem Wert von B zuordnet.
- Beispiel: **Fakultätsfunktion**

$$n! := \begin{cases} 1 & n = 0 \\ (n-1)!\, n & n > 0 \end{cases}$$

- Ist n! <u>wohldefiniert</u> auf den natürlichen Zahlen?
  - Ergibt die Definition wirklich eine Zuordnungsvorschrift?
  - Ja: Für jedes n > 0 wird der Funktionsparameter auf der rechten Seite um genau Eins kleiner und damit irgendwann 0. Die Rekursion bricht dann ab, der Funktionswert ist damit als endliches Produkt natürlicher Zahlen wieder eine natürliche Zahl.

Rekursive Programme

- Rekursiver Aufruf einer Methode: Eine Methode wird während Ausführung dieser Methode wieder aufgerufen.
- Direkte Rekursion: Der Methodenaufruf ist direkt im Methodenrumpf enthalten.
- Indirekte Rekursion: Die Methode wird indirekt über andere im Methodenrumpf enthaltenen Methode aufgerufen.
- Indirekte Rekursion vermeiden!
- Rekursive Java-Funktion, die die Fakultät berechnet:

```
public long getFakultaet(long n) {
  if (n == 0) { // Rekursionsabbruch
    return 1;
  } else {
    return getFakultaet(n - 1) * n;
  }
}
```

Rekursive Programme

- Wieso funktioniert die Berechnung?
- Wir betrachten dazu die Aufrufe und den Laufzeitkeller während der Ausführung.
- Aufrufbaum:
  - Darstellung von Methodenaufrufe
  - Ein Aufruf einer Methode g aus einer Methode f wird mit einem Pfeil oder Strich von oben nach unten von f zu g notiert
  - Bei mehreren Aufrufen in einer Methode f werden die Pfeile von links nach rechts in zeitlicher Aufrufreihenfolge notiert
- Rekursionstiefe:
  - Maximale Anzahl rekursiver Aufrufe einer Funktion
  - Wert ist abhängig von einer Eingabe

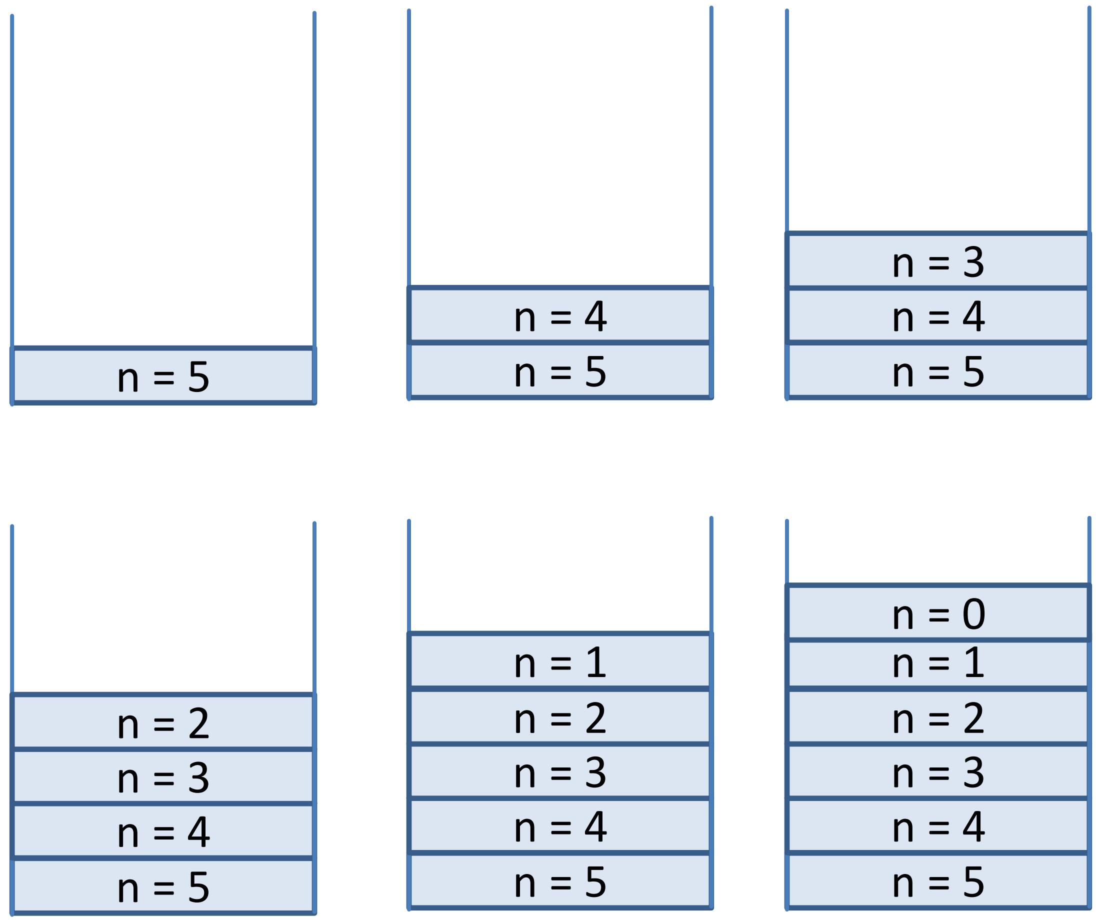
Rekursive Programme

Aufrufbaum

getFakultaet(5)
↓
getFakultaet(4)
↓
getFakultaet(3)
↓
getFakultaet(2)
↓
getFakultaet(1)
↓
getFakultaet(0)

Maximale Rekursionstiefe: 5

Entwicklung des Laufzeitkellers bis zum letztem rekursivem Aufruf



Rekursive Programme

- Verschiedene **Rekursionsarten**
  - Lineare Rekursion
    - Ein Aufruf einer Methode hat höchstens einen rekursiven Aufruf zur Folge wie bei dem Fakultätsbeispiel
  - Endrekursion
    - Lineare Rekursion, bei dem der Aufruf als letztes ausgeführt wird. Beim Fakultätsbeispiel durch n * getFakultaet(n – 1) erreichbar.
  - Verzweigende Rekursion
    - Ein Aufruf hat mehr als einen rekursiven Aufruf zur Folge
  - Indirekte Rekursion
    - Die Methode wird indirekt über andere im Methodenrumpf enthaltenen Methode aufgerufen
  - Verschachtelte Rekursion
    - Der rekursive Aufruf enthält als Parameter mindestens einen weiteren rekursiven Aufruf.
      Etwa: f(n) := f(f(n – 3))

Rekursive Programme

- Beispiel einer verzweigenden Rekursion: **Fibonaccizahlen**

$$fib(n) := \begin{cases} 1, & n = 1, n = 2 \\ fib(n-2) + fib(n-1), & n > 2 \end{cases}$$

$$\begin{aligned} fib(4) = \ & fib(2) + \ fib(3) = 1 + fib(3) \\ & = 1 + fib(1) + fib(2) = 1 + 1 + 1 \\ & = 1 + 2 = 3 \end{aligned}$$

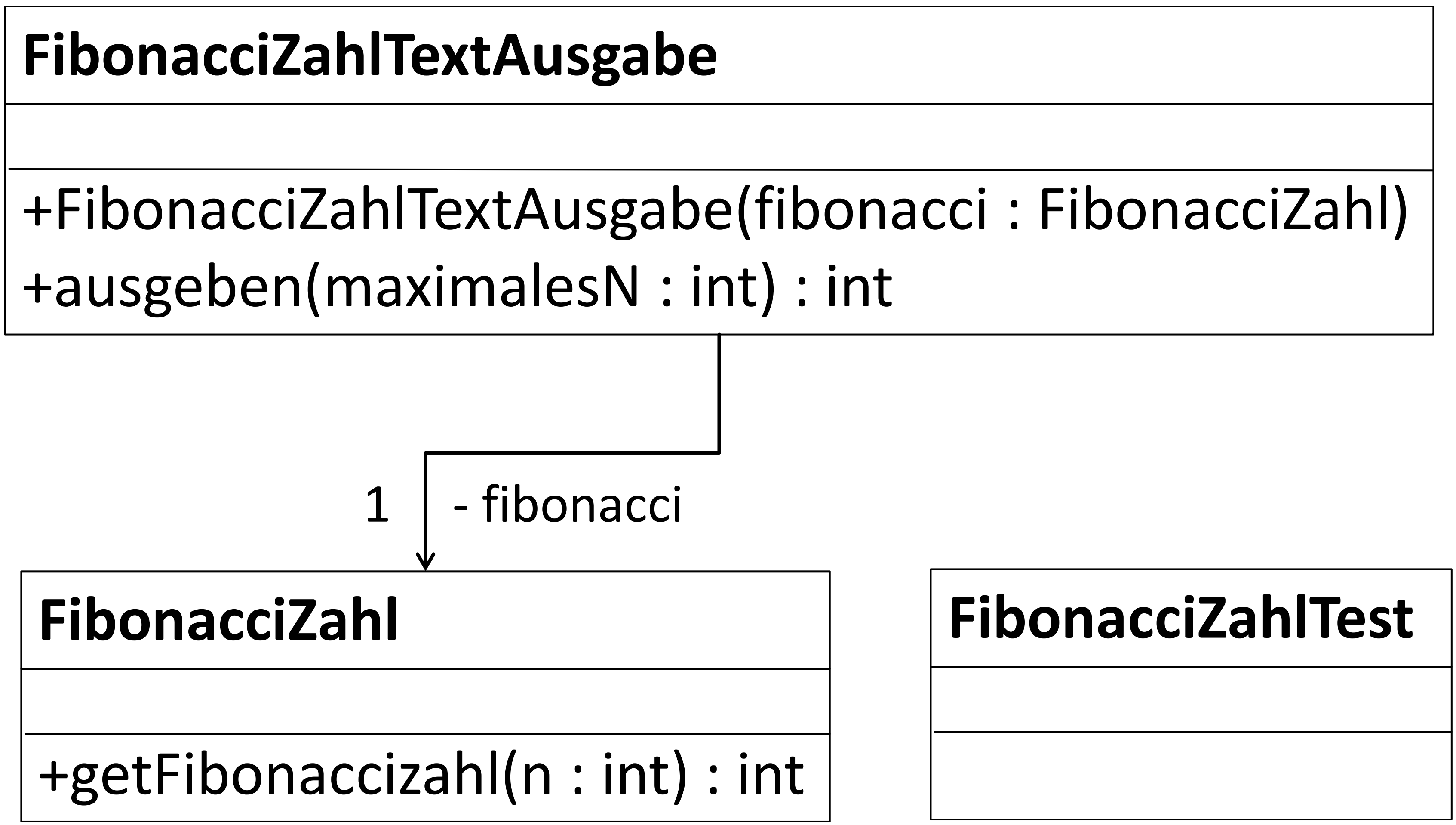
n	1	2	3	4	5	6	7
fib(n)	1	1	2	3	5	8	13

Rekursive Programme

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

- Aufrufbaum und Laufzeitkeller für getFibonaccizahl(5)?
- Maximale Rekursionstiefe in Abhängigkeit von n?
- Programm, das alle Fibonaccizahlen bis zu einer Obergrenze berechnet und auf dem Bildschirm ausgibt

Rekursive Programme



Rekursive Programme

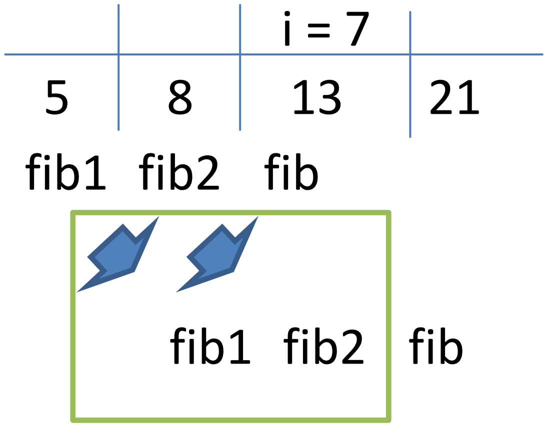
- Dynamisches Programmieren anwenden, um bereits berechnete Werte nur einmal zu berechnen
  - jeden Wert fib(i) für 0 < i < n berechnen und speichern
  - z.B. in einem Feld für allen Fibonaccizahlen von 1 bis n

```
public int getFibonacciZahl(int n) {
   int [] fib = new int[n + 2];
   fib[1] = 1;
   fib[2] = 1;

   for (int i = 3; i <= n; i++) {
      fib[i] = fib[i - 2] + fib[i - 1];
   }

   return fib[n];
}
```

Rekursive Programme

- Verbesserung
  - Nur die zwei vorangehenden Werte werden zur Berechnung des nächsten Werts benötigt
  - Feld durch drei Variablen ersetzen

```
public int getFibonacciZahl(int n) {
   int fib = 1
   int fib1 = 1;
   int fib2 = 1;
   for (int i = 3; i <= n; i++) {
       fib = fib1 + fib2;
       fib1 = fib2;
       fib2 = fib;
   }
   return fib;
}
```



Rekursive Programme

- Dynamisches Programmieren kann oft eingesetzt werden, um Rekursion vollständig zu vermeiden
- Kosten:
  - zusätzlicher Speicherplatz für die Teillösungen
- Nutzen:
  - Speicherverbrauch des Laufzeitkellers reduziert
  - Erhöhte Ausführungsgeschwindigkeit, wenn unnötige rekursive Aufrufe vermieden werden
- Implementierung Fakultät und Fibonaccizahlen in der Praxis?
  - Hinweis: Werte- und Definitionsbereich sind wenige, ganze Zahlen, die ab 0 oder 1 beginnen

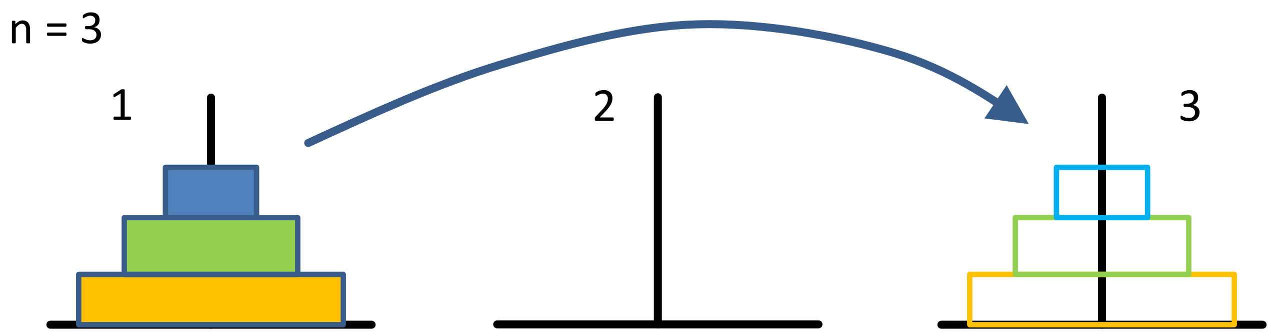
Problemlösung mit Rekursion
(analog zur vollständigen Induktion)

Induktionsbeweis	Rekursion
Rekursive Struktur der zu beweisenden **Aussage** erkennen und Induktionshypothese aufstellen	Rekursive Struktur des gegebenen **Problems** erkennen
Induktionsanfang: Für kleinsten Aussagen (Zahlen oder Strukturen) einen Beweis finden	**Rekursionsabbruch**: Für die kleinsten Probleme eine direkte Lösung finden
Induktionsschluss: 1. Aussage A auf kleinere Aussagen reduzieren 2. Induktionshypothese auf kleinere Aussagen anwenden 3. Aus bewiesenen kleineren Aussage, Beweis von a konstruieren	Rekursive Lösung konstruieren: 1. Problem p in kleinere Probleme aufteilen, 2. diese rekursiv lösen und 3. aus den Lösungen eine Lösung für p konstruieren

Ähnelt Dynamischen Programmieren, aber Teillösungen werden nicht explizit zwischengespeichert.

Problemlösung mit Rekursion

Türme von Hanoi	
Gegeben:	Drei Stäbe (1, 2 und 3). Eine Pyramide aus n Scheiben auf Stab 1.
Gesucht:	Reihenfolge von Verschiebungen der Scheiben, so dass die Pyramide auf Stab 3 zu liegen kommt.

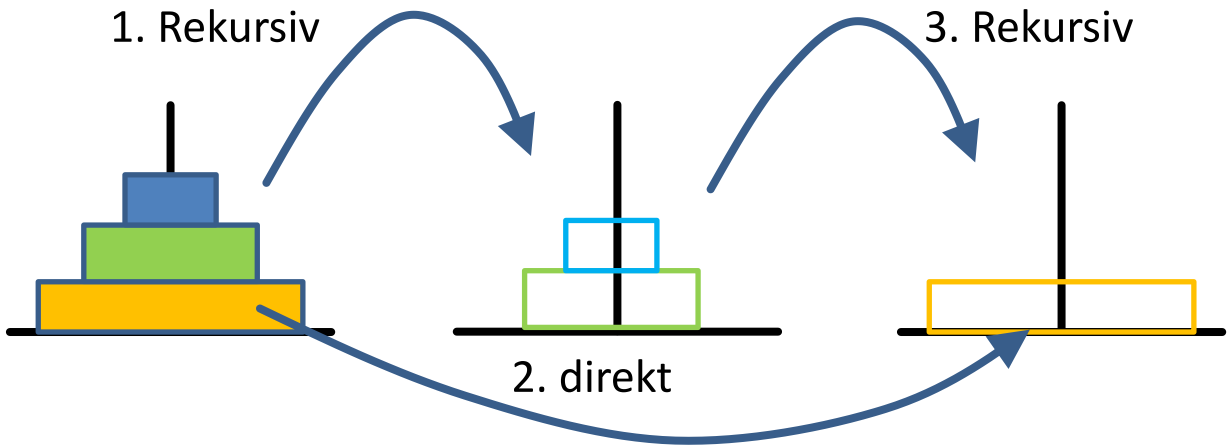


Reihenfolge:
1 -> 3
1 -> 2
....

- Nur die oberste Scheibe einer Pyramide darf verschoben werden
- Ein Scheibe darf nur auf einer größeren Scheibe liegen

Problemlösung mit Rekursion

- Rekursive Struktur
  – Pyramide mit n Scheiben enthält eine Pyramide mit n – 1 Scheiben (auf der untersten Scheibe)
- Idee für Rekursion
  – Pyramide mit n – 1 Scheiben rekursiv verschieben
  – Rekursion bricht bei n = 1 ab. Eine Pyramide mit einer Scheibe kann direkt verschoben werden.



Problemlösung mit Rekursion

- Annahme so eine rekursive Funktion existiert
  - Bei Aufruf muss n angegeben werden
  - Der Funktion muss bei Aufruf auch mitgeteilt werden, von und zu welchem Stab die Pyramide verschoben wird
  - Ein Stab bleibt als Information übrig
- Diese Informationen könnte als Parameter bei Aufruf übergeben werden

```
fib( n, von, zu, frei)
```

- Wie würden die rekursiven Aufruf dieser Funktion für n = 3 aussehen?
- Diese Aufrufe verallgemeinern.
- Anschließend fib programmieren.

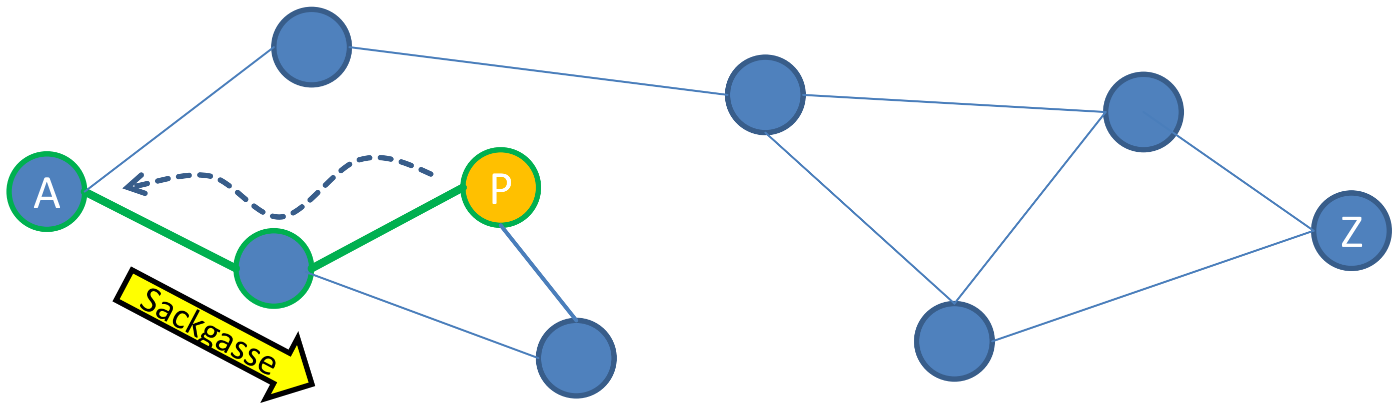
Problemlösung mit Rekursion

```
public void hanoiBerechnen(int n, int von, int zu, int frei)
{
  if (n == 1) {
    System.out.println(von + " -> " + zu);
  } else {
    hanoiBerechnen(n - 1, von, frei, zu);
    System.out.println(von + " -> " + zu);
    hanoiBerechnen(n - 1, frei, zu, von);
  }
}
```

```
public void hanoiBerechnen(int n, int von, int zu, int frei) {
  if (n >= 1) {
    hanoiBerechnen(n - 1, von, frei, zu);
    System.out.println(von + " -> " + zu);
    hanoiBerechnen(n - 1, frei, zu, von);
  }
}
```

Backtracking

- Backtracking: Problemlösungsmethode
  1. schrittweise eine Teillösung mit einem Teilschritt zu einer Gesamtlösung erweitern
  2. Teilschritte verwerfen und zu einer vorherigen Teillösung zurückgehen (to backtrack), wenn sich im Laufe des Verfahrens imt der Teillösung keine Gesamtlösung konstruieren lässt
- Rekursives Backtracking: Algorithmus ist rekursiv implementiert
- Einige Probleme, die mit Backtracking gut lösbar sind:
  - Wegsuche in einem Labyrinth
  - Kombinatorische Spiele wie Sudoku
  - Packprobleme: einen LKW möglichst voll mit Kisten bepacken

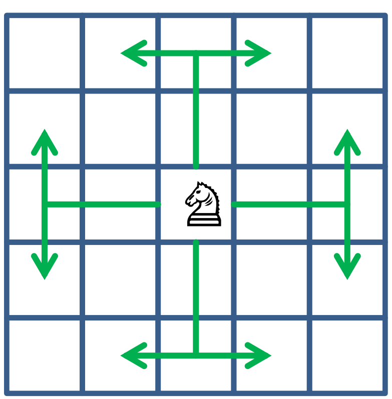


Backtracking

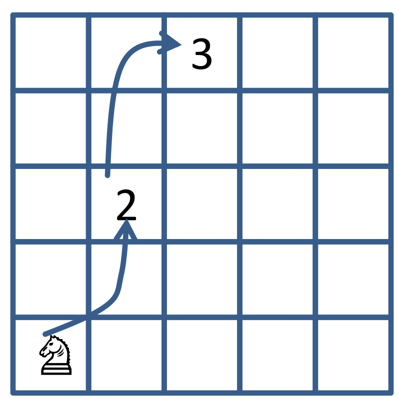
Springerproblem	
Gegeben:	$n \times n$ Schachbrett mit $n \geq 5$, eine Springerfigur auf einem Eckfeld
Gesucht:	Zugreihenfolge, so dass der Springer jedes Feld genau einmal besucht

- Für $n < 5$ gibt es keine Lösungen
- Variante: Springer landet mit dem letzten Zug wieder auf dem Ausgangsfeld

Erlaubten Springerzüge:



Lösungsnotation:
Sprünge nummerieren



Eine Sackgasse:

19	8	3	12	17
	11	18	9	4
7	2		16	13
	15	10	5	
1	6		14	

Backtracking

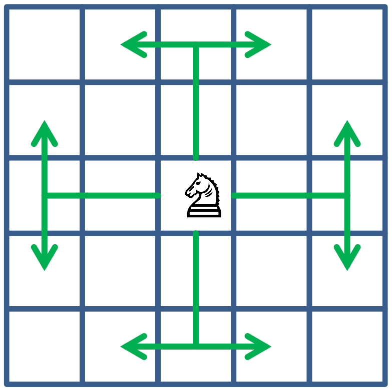
- Rekursive Struktur?
  - Nach einem Zug, das Feld von dem gezogen wurde „wegnehmen“
  - Der Rest ist wieder ein „Springerzugproblem“
- Kleinste einfache Teilproblem:
  - Nur ein letztes Feld vorhanden
- Programm muss alle Möglichkeiten systematisch aufzählen
  - Jeden der 8 möglichen Züge aufzählen und überprüfen
  - Nur gültigen Zug in ein leeres Feld (innerhalb Spielfeld) ausführen
  - Ist eine Gesamtlösung noch nicht erreicht:
    - Rekursiv ab nächster Position weitersuchen
    - Falls rekursiver Aufruf keine Lösung brachte: letzten Zug zurücknehmen (backtrack) und nächsten möglichen der 8 Züge probieren
    - Fallsalle Züge ab einer Position erfolglos ausprobiert wurden, gibt es keine zu einer vollständigen Lösung erweiterbare Teillösung: die Rekursion muss abbrechen

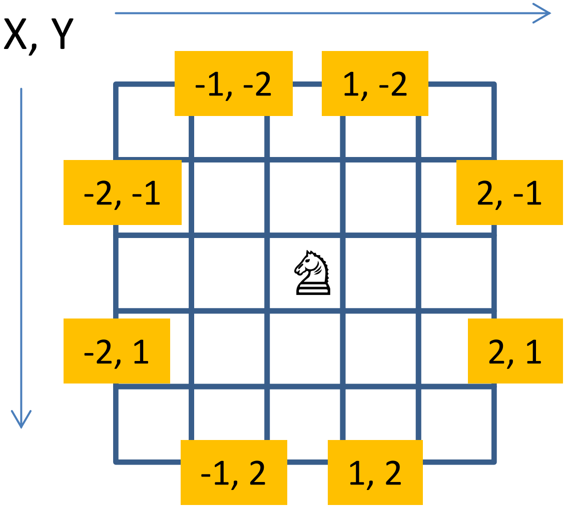
Backtracking

```
private boolean sucheLoesung( /* Position .... */ ) {
  while ( /* es existiert noch ein Teilschritt */ ) {
    // Teilschritt auswählen
    if ( /* ausgewählter Teilschritt gültig */ ) {
      // Teilschritt durchführen
      // Teillösung erweitern
      if( /* Losung gefunden */ ) {
        return true;
      } else {
        if ( sucheLoesung( /* neue position ... */  )  ) {
          return true;
        } else {
          // Teillösung wieder zurücknehmen (Backtracking)
        }
      }
    }
  }
  return false;
}
```

Backtracking

- Möglichen Züge berechnen
  - Schleife von 0 bis 7 laufen lassen
  - Änderung des x- und y-Werts auf bestehende Koordinate addieren (offset)
  - Die offset-Werte jeweils für Zeile und Spalte in einem Feld speichern





Backtracking

- Codierung:
  - zwei-dimensionales int-Feld
  - 0 bedeutet, dass das Feld nicht besucht ist
  - >0 bedeutet, dass Feld ist bereits besucht (markiert)
- Position mit Zeilen- und Spaltenindex
- Aktuelle Zugnummer ist ein int-Wert

Springerproblem
-feld : int [] [] -richtungX : int [8] -richtungY : int[8]
+Springerproblem(n : int) +sucheLoesung(x : int, y : int, zug : int) : boolean