



Informatik I

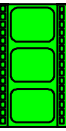
Prof. Dr. Christian Pape

Kapitel 10 Rekursion / Backtracking

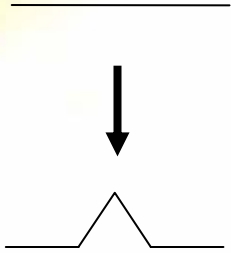


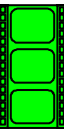
Inhalt

- Rekursion
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - Verzweigende Rekursion
 - Verschachtelte Rekursion
 - Türme von Hanoi
- Backtracking (Rückverfolgung)
 - Wegsuche
 - Springerproblem

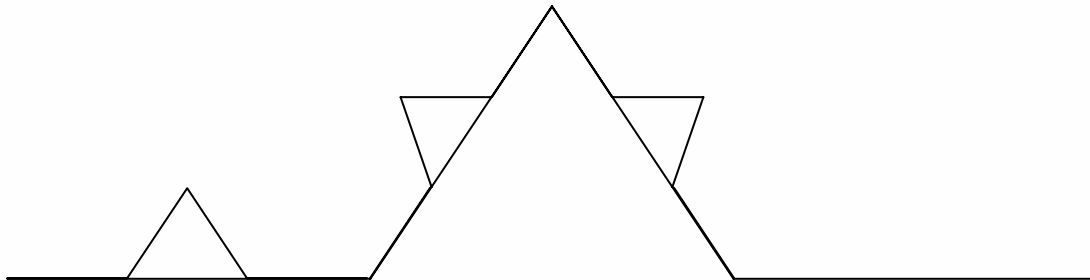
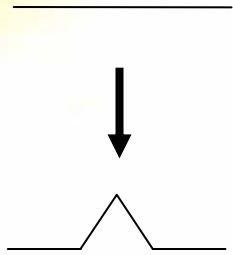


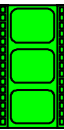
Rekursion / Kochkurve



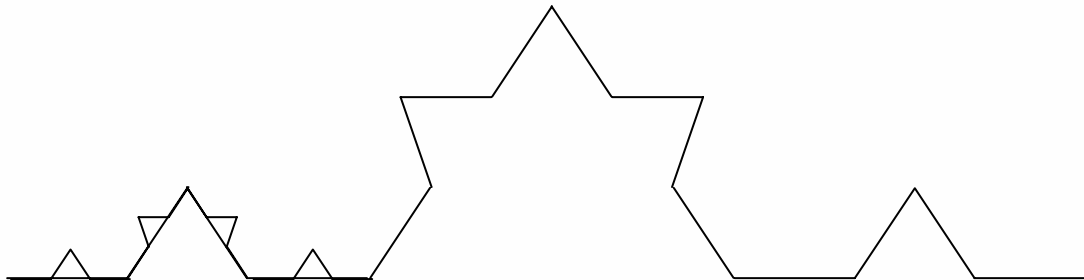
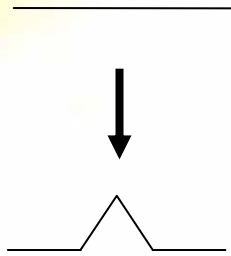


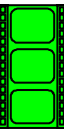
Rekursion / Kochkurve



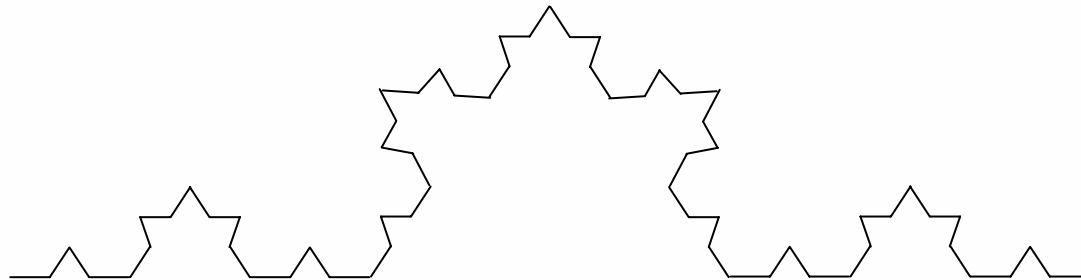
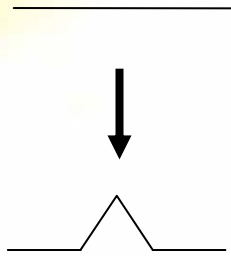


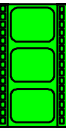
Rekursion / Kochkurve



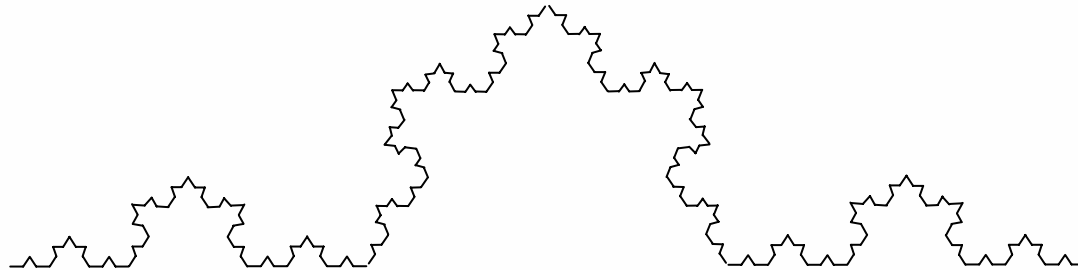
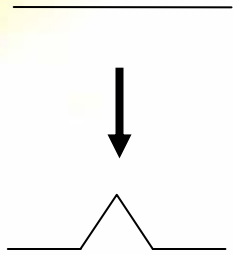


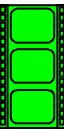
Rekursion / Kochkurve



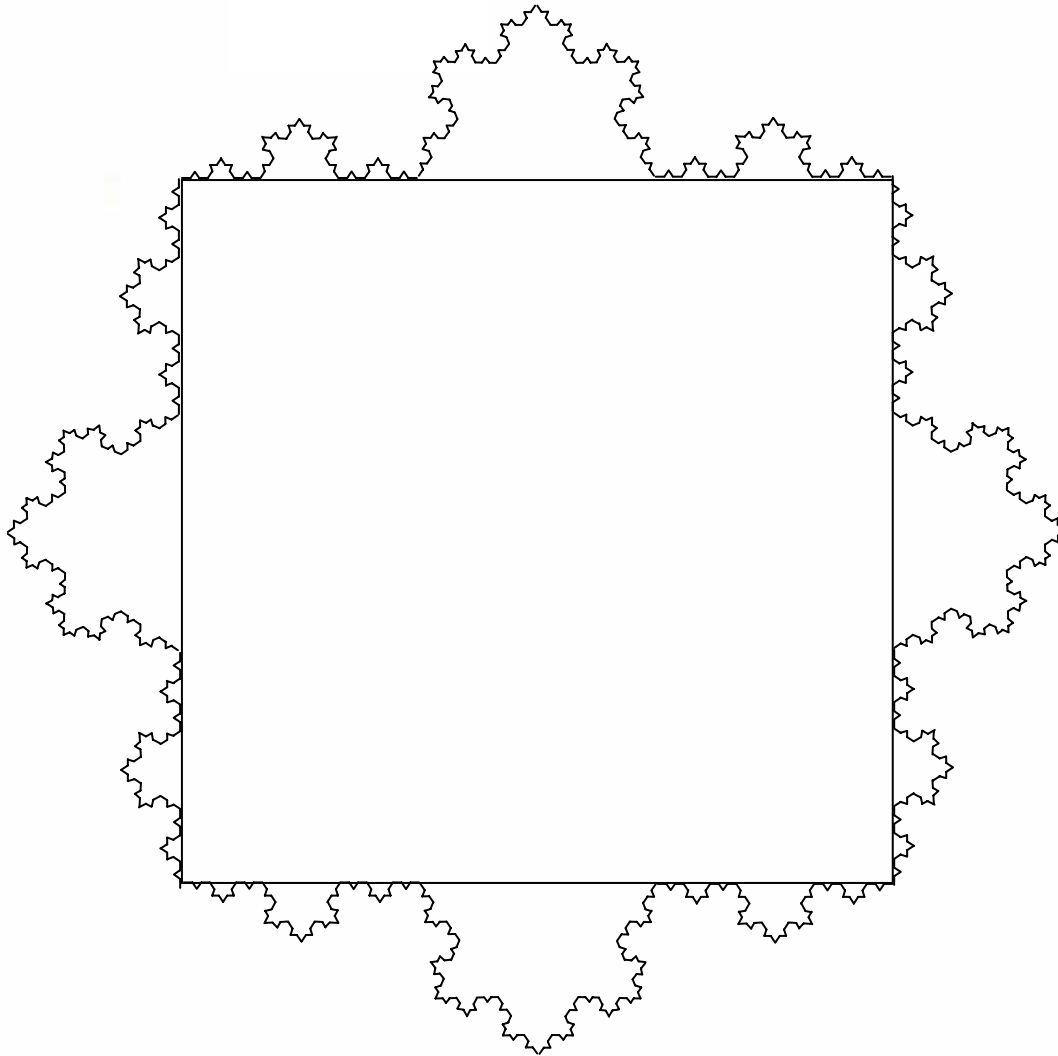


Rekursion / Kochkurve





Rekursion / Kochkurve

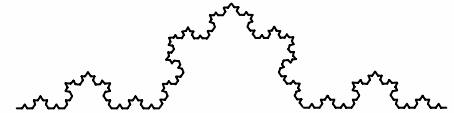


Rekursion, Backtracking



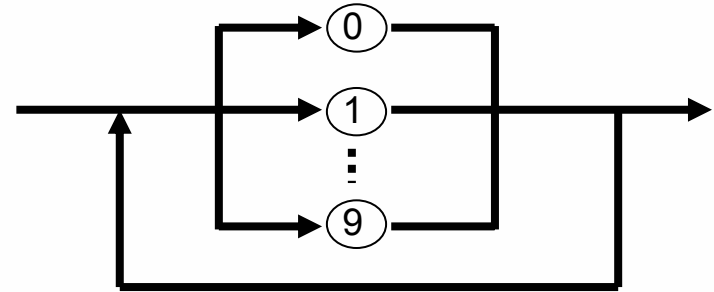
Rekursion

- lat. *recurrere*, zurücklaufen
- Etwas verweist auf sich selbst
 - „Dieser Satz ist unwahr“
 - „Dieser Satz kein Verb“
 - „Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen“
- Rekursion in der Informatik
 - Strukturen oder Probleme mit Selbstbezug
 - Algorithmen, die Berechnungen auf sich selbst zurückführen



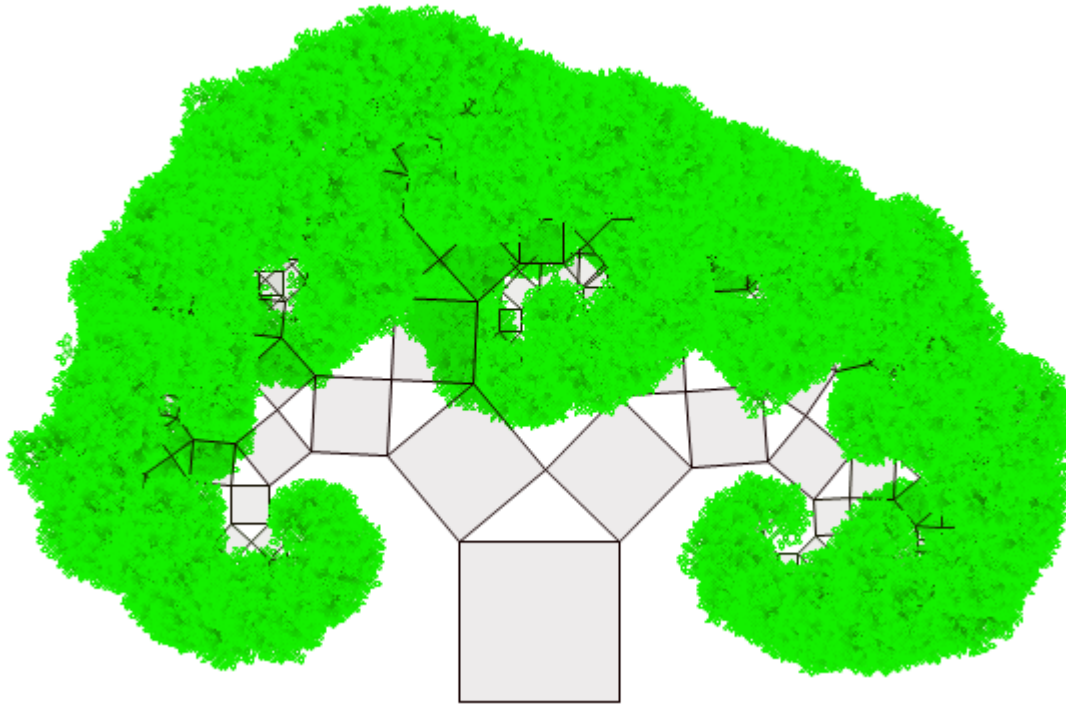


Rekursion / Beispiele

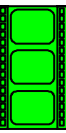




Rekursion / Pythagoräische Baum

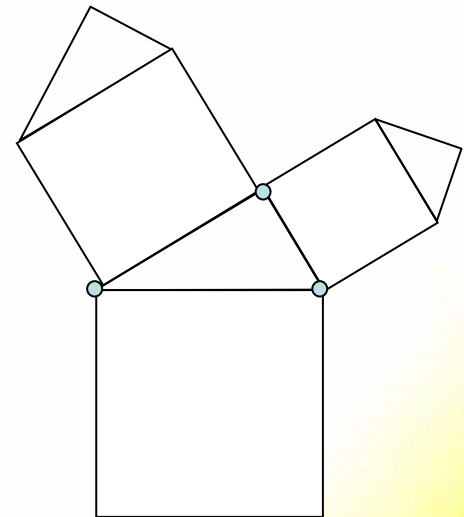
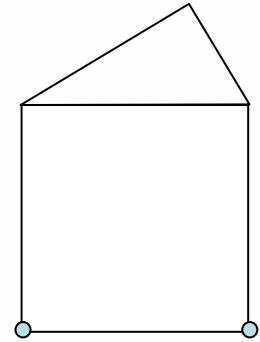


[aus http://de.wikipedia.org/wiki/Bild:Recursive_Pythagoras.png]



Rekursion / Pythagoräische Baum

- Rekursiver Algorithmus
 - Eingabe: zwei Punkte und eine Höhe
 - Errichte über zwei Punkten ein Quadrat
 - Auf der Oberseite zeichne ein rechtwinkliges Dreieck mit einer bestimmten Höhe
- Rufe *diesen Algorithmus* jeweils für die beiden Punkte auf, die die Schenkel dieses Dreieckes definieren, und einer neuen Höhe
- Rekursiver Aufruf, bis man genug hat:
Rekursionsabbruch
 - Bis zum Beispiel Abstand beider Punkte eine Schwelle unterschreiten





Rekursion / Pythagoräische Baum

- In Java (fiktiv)

```
public void zeichneBaum(Punkt p1, Punkt p2, double hoehe) {  
    Punkt d1, d2, d3; // Punkte für das Dreieck  
    if ( abstand(p1,p2) > MINIMALER_ABSTAND ) {  
        zeichneQuadrat(p1, p2);  
        // Berechne Punkt des Dreiecks mit d2 als Spitze  
        zeichneDreieck(d1, d2, d3)  
        // Berechne neue hoehe1 für d1 und d2  
        zeichneBaum(d1, d2, hoehe1);  
        // Berechne neue hoehe2 für d1 und d3  
        zeichneBaum(d2, d3, hoehe2);  
    }  
}
```



Rekursion

- Rekursion entspricht dem mathematischen Prinzip der vollständigen Induktion
 - Reduktion eines Problems der Grösse n auf ein Problem der Grösse $n-1$ (Induktionsschritt)
 - Einfache Lösung eines Problems der Grösse 1 (Induktionsanfang)
- Alternative zu Iterationen / Schleifen
- Probleme mit rekursiven Lösungen können auch immer mit Iteration gelöst werden
 - Viele Lösungen sind mit Rekursion einfacher zu verstehen und/oder einfacher zu programmieren



Rekursion / Beispiel 1

- **Problem: Berechne Summe**

$$\text{sum}(n) = 0 + 1 + 2 + 3 + \dots + n$$

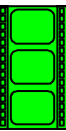
rekursiv

- **Rekursive Definition**

$$\text{sum}(0) := 0$$

$$\text{sum}(n) := \text{sum}(n-1) + n \text{ für } n > 0$$

- **Rekursive Definition von `sum` mit Bezug auf sich selbst.**



Rekursion / Beispiel 1

$$\text{sum}(n) := \text{sum}(n-1) + n$$

$$\begin{aligned}\text{sum}(5) &= \text{sum}(5-1) + 5 \\ &= \text{sum}(4) + 5 \\ &= \text{sum}(3) + 4 + 5 \\ &= \text{sum}(2) + 3 + 4 + 5 \\ &= \text{sum}(1) + 2 + 3 + 4 + 5 \\ &= \text{sum}(0) + 1 + 2 + 3 + 4 + 5 \\ &= 0 + 1 + 2 + 3 + 4 + 5\end{aligned}$$

Rekursion / Beispiel 1

```

public static long sum(long n) {
    if (n == 0) {
        return 0;
    } else {
        return sum(n-1) + n;
    }
}

```

Rekursionsabbruch $\text{sum}(0) = 0$
bzw. „Induktionsanfang“

Rekursion $\text{sum}(n) = \text{sum}(n-1) + n$
bzw. „Induktionsschritt“

- Ausführung / Implementierung Rekursion

- Bei Aufruf einer Methode wird der Speicher für alle lokale Variablen / Parameter auf den Laufzeitkeller (engl. *Stack*, Stapel) angelegt, d.h. die übergebenen Parameterwerte werden auf den Laufzeitkeller geschoben (*to push*).
- Nach Ende der Methode werden die lokalen Variablen / Parameter des Aufrufs vom Laufzeitkeller wieder „heruntergeworfen“ (*to pop*): die lokalen Variablen vom vorherigen Aufruf sind wieder sichtbar
- Der Bereich für einen Programmaufruf wird *Rahmen* (*Frame*) genannt. Nur ein Frame ist immer „sichtbar“: der oberste, der alle aktuellen lokalen Variablen und Parameter enthält. (Dazu gehören aber auch noch Rückgabewerte, Rücksprungadresse, Stack für Operanden und eine nicht gefangene Exception.)



Laufzeitkeller

```
1: public static long sum(long n) {  
2:     if (n == 0) {  
3:         return 0;  
4:     } else {  
5:         return  
6:             sum(n-1)  
7:             + n;  
8:     }  
9: }
```

```
23: long summe = sum(5);  
24: summe =  
→ 25:     sum(2); // <- Aufruf  
26: summe = sum(7);
```

Laufzeitkeller vor Aufruf

Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

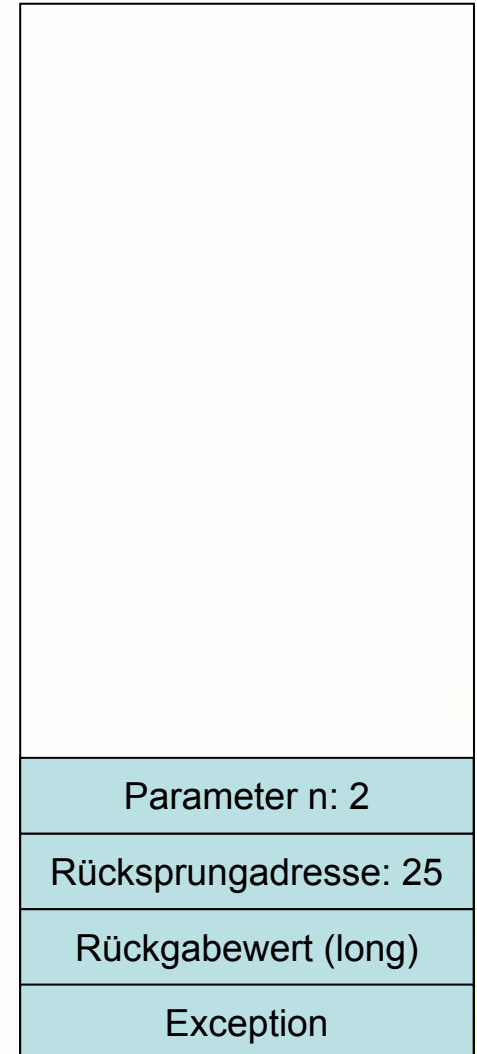
```

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- Aufruf
26: summe = sum(7);

```

Laufzeitkeller bei n=2



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

Laufzeitkeller bei n=1
(1. Rekursion)

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```

Parameter n: 1
Rücksprungadresse: 6
Rückgabewert (long)
Exception
Parameter n: 2
Rücksprungadresse: 25
Rückgabewert (long)
Exception

Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

→

Laufzeitkeller bei n=0
(2. Rekursion)

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```

Parameter n: 0
Rücksprungadresse: 6
Rückgabewert 0
Exception
Parameter n: 1
Rücksprungadresse: 6
Rückgabewert (long)
Exception
Parameter n: 2
Rücksprungadresse: 25
Rückgabewert (long)
Exception



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }
    
```



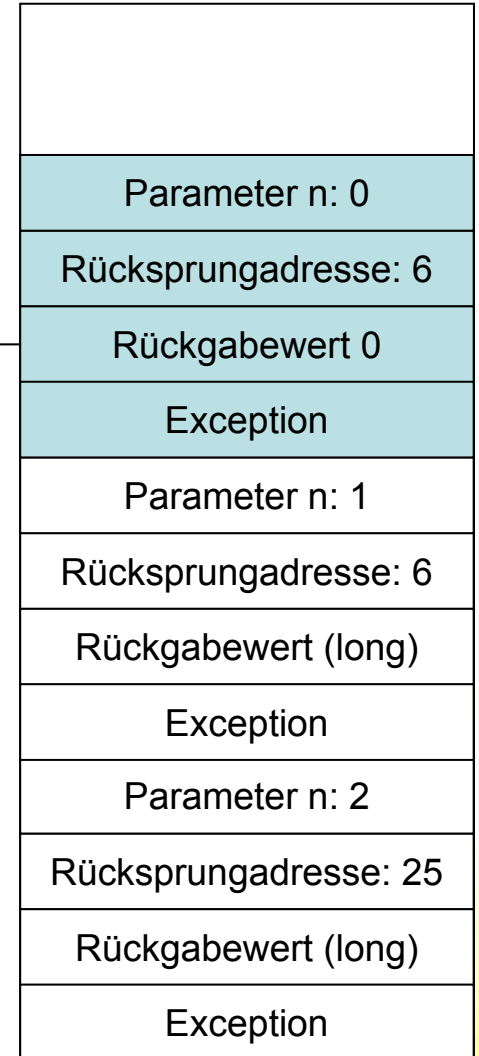
Rücksprung von 2. Rekursion zu Zeile 6

spezielles Register für Rückgabewert

0

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);
    
```



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

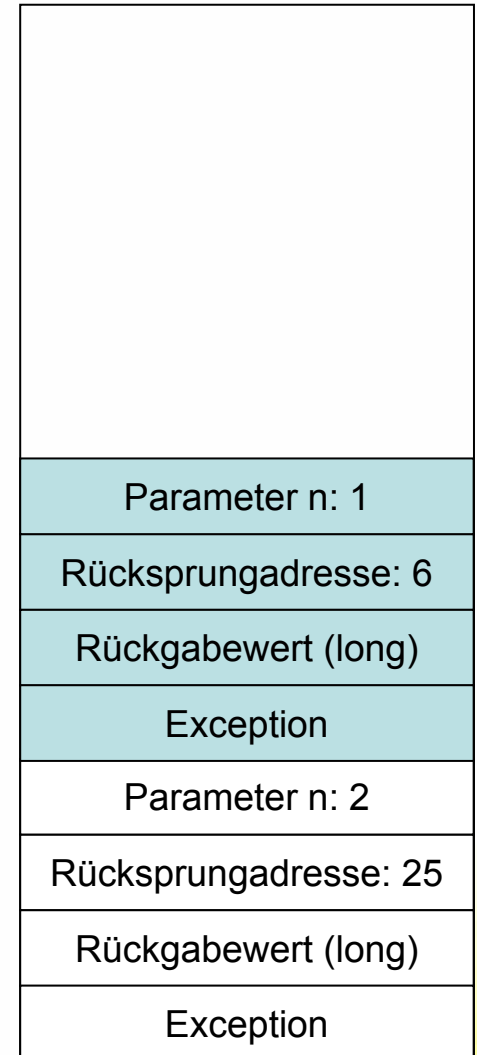
spezielles Register
für Rückgabewert

0

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

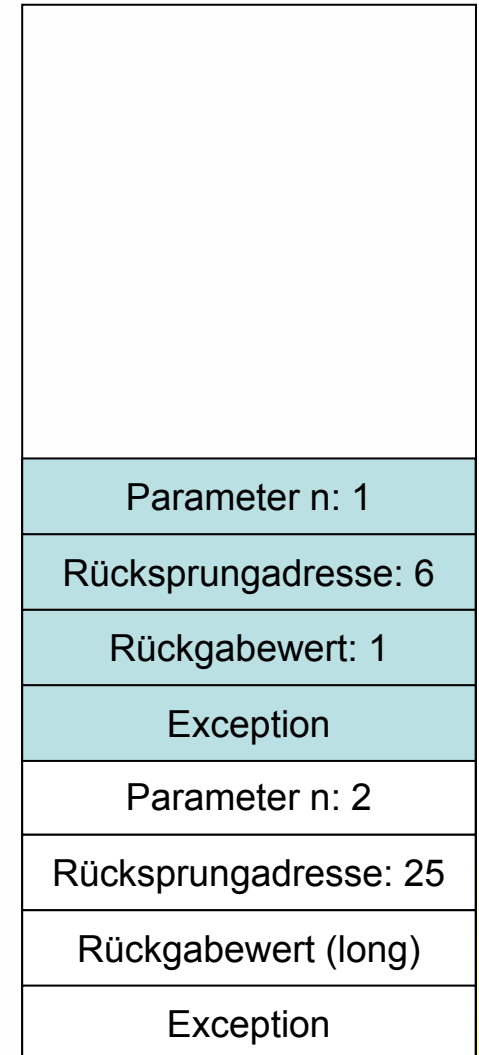
spezielles Register
für Rückgabewert

0

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

Rücksprung von 1. Rekursion
zu Zeile 6

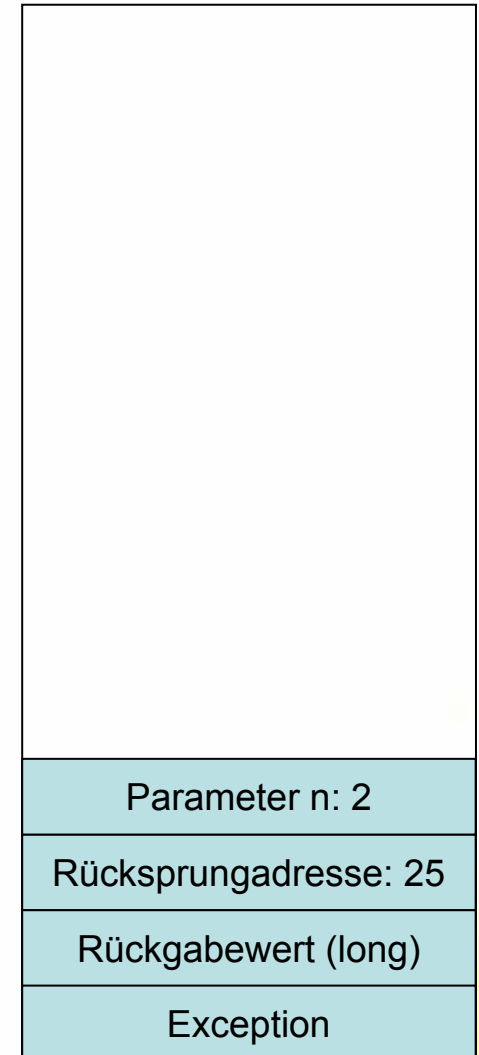
spezielles Register
für Rückgabewert

1

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

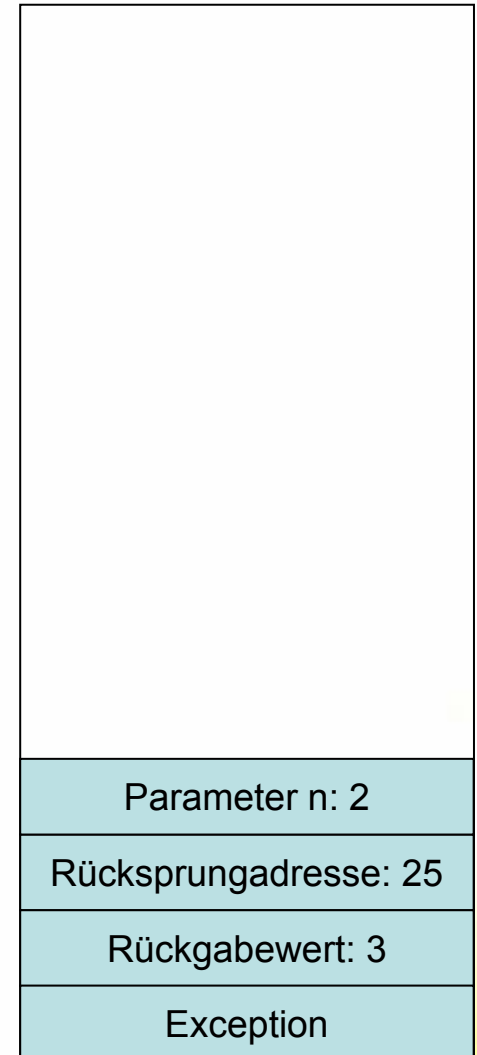
spezielles Register
für Rückgabewert

1

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```



Laufzeitkeller

```

1: public static long sum(long n) {
2:     if (n == 0) {
3:         return 0;
4:     } else {
5:         return
6:             sum(n-1)
7:             + n;
8:     }
9: }

```

Rücksprung von 1. Rekursion
zu Zeile 25

spezielles Register
für Rückgabewert

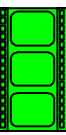
3

```

23: long summe = sum(5);
24: summe =
25:     sum(2); // <- 1. Aufruf
26: summe = sum(7);

```

Laufzeitkeller



- Daten des Laufzeitkellers sind eine öfter vorkommende Datenstruktur
 - **Stapel** (stack)
 - **Datenstruktur**: Daten + Methoden zur Manipulation der Daten
 - Zwei Methoden: push(x) und pop()
 - Kann mit einer Klasse und einem Feld implementiert werden
- Push: schiebt x als oberstes Element auf den Stapel
- Pop: gibt oberstes Element des Stapels zurück

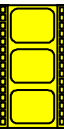
push(7)
push(12)
pop()
push(1)
push(42)
pop()
pop()

Stapel

42
1
7

JDK 1.5

```
import java.util.Stack;  
  
Stack stack = new Stack();  
stack.push(7);  
stack.push(12);  
stack.pop();  
System.out.println(stack.pop());  
// 7 wird ausgegeben
```



Laufzeitkeller

- Laufzeitkeller ist eine öfter vorkommende Datenstruktur
 - **Stapel** (stack)
 - **Datenstruktur**: Daten + Methoden zur Manipulation der Daten
 - Zwei Methoden: `push(x)` und `pop()`
 - Kann mit einer Klasse und einem Feld implementiert werden
- Push: schiebt `x` als oberstes Element auf den Stapel
- Pop: gibt oberstes Element des Stapels zurück

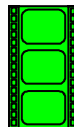
push(7)
push(12)
pop()
push(1)
push(42)
pop()
pop()

Stapel



JDK 1.5

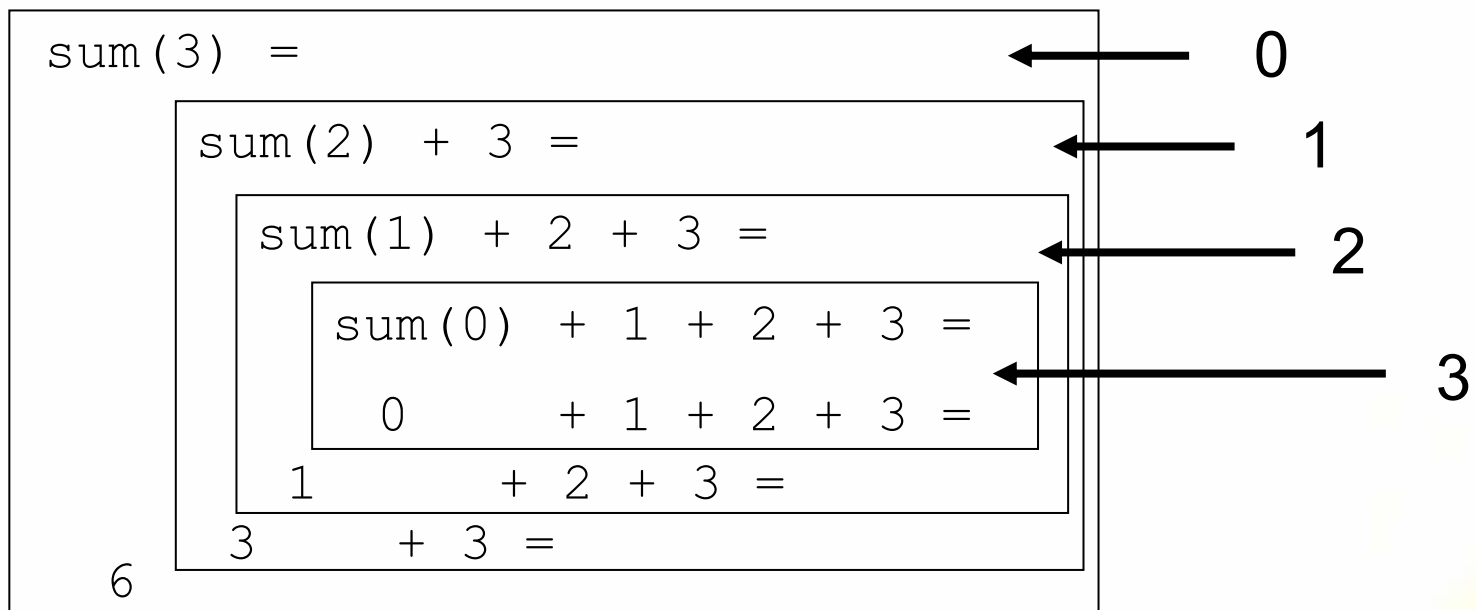
```
import java.util.Stack;  
  
Stack stack = new Stack();  
stack.push(7);  
stack.push(12);  
stack.pop();  
System.out.println(stack.pop());  
// 7 wird ausgegeben
```



Rekursion / Beispiel 1

```
public static long sum(long n) {
    if (n == 0) {
        return 0;
    } else {
        return sum(n-1) + n;
    }
}
```

Rekursionstiefe

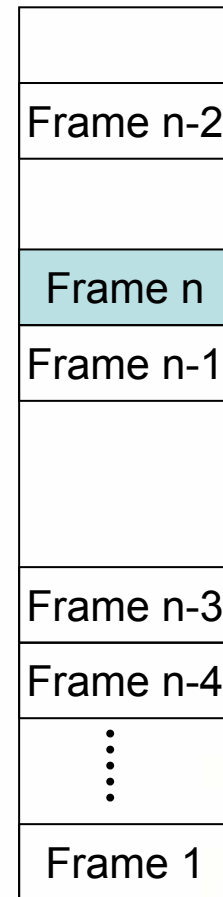




Laufzeitkeller

- Laufzeitkeller in Java (genauer in der Java Virtual Machine, JVM)
 - „The memory for a Java virtual machine stack does not need to be contiguous“
 - „The Java virtual machine specification permits Java virtual machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation“ [Abschnitt 3.5.2 der JVM Spezifikation]
- Größe Kellerspeicher bei Suns Java-Interpretern dynamisch bis zu einer maximalen Obergrenze, Stackspeicher ist nicht kontinuierlich
- Bei zu vielen rekursiven Aufrufen
 - StackOverflowError, falls Obergrenze Stack erreicht
 - OutOfMemoryError, falls kein passender Speicher für dynamische Erweiterung des Stacks fehlt
- Obergrenze bei Start der JVM definierbar
 - „java -Xss 400KB“ bei Windows Version

Hauptspeicher



Nicht kontinuierlicher Laufzeitkeller im Hauptspeicher



Rekursion

- **Lineare Rekursion**
 - Jeder Aufruf löst maximal einen rekursiven Aufruf auf
- **Verzweigende Rekursion**
 - Jeder Aufruf kann mehr als einen rekursiven Aufruf auslösen
- **Verschachtelte Rekursion**
 - Argument in rekursivem Aufruf ist zusätzlich ein rekursiver Aufruf



Inhalt

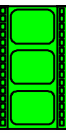
- Rekursion
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - **Verzweigende Rekursion**
 - Verschachtelte Rekursion
 - Türme von Hanoi
- Backtracking (Rückverfolgung)
 - Wegsuche
 - Springerproblem



Verzweigende Rekursion

- **Problem:**
 - Berechne Fibonacci Zahlen rekursiv
- **Rekursive Definition**

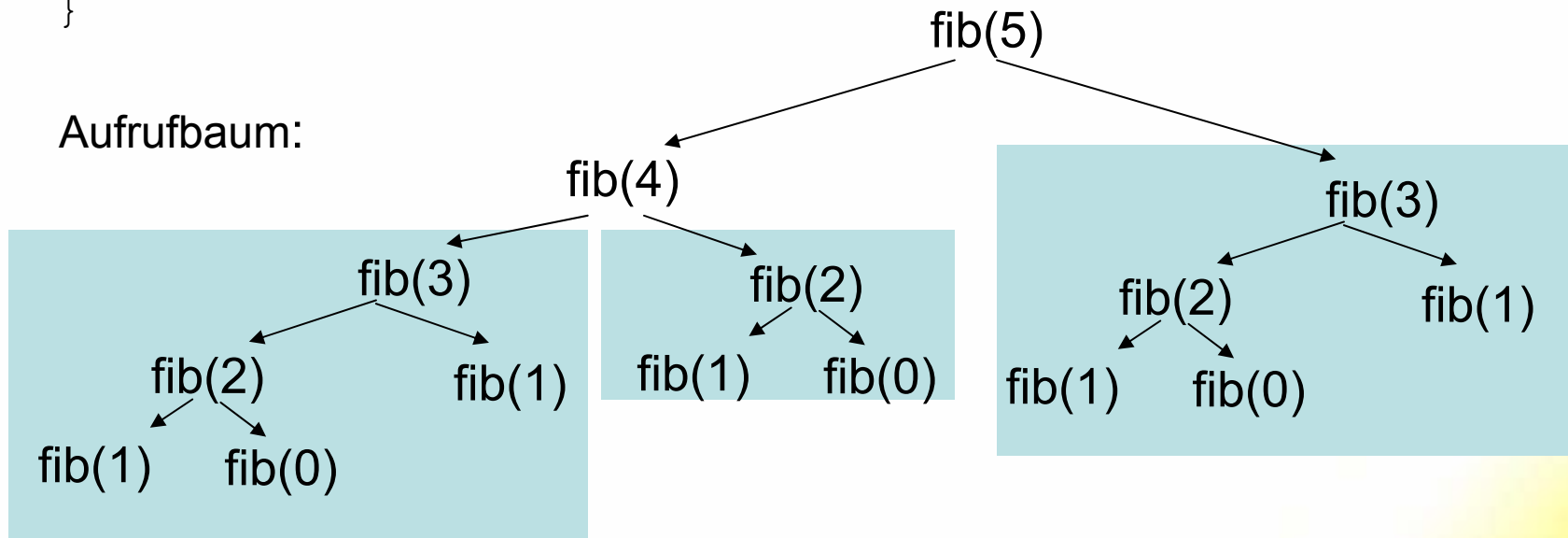
$$\text{fib}(0) := 1, \text{fib}(1) := 1$$
$$\text{fib}(n) := \text{fib}(n-1) + \text{fib}(n-2) \text{ für } n > 1$$



Verzweigende Rekursion

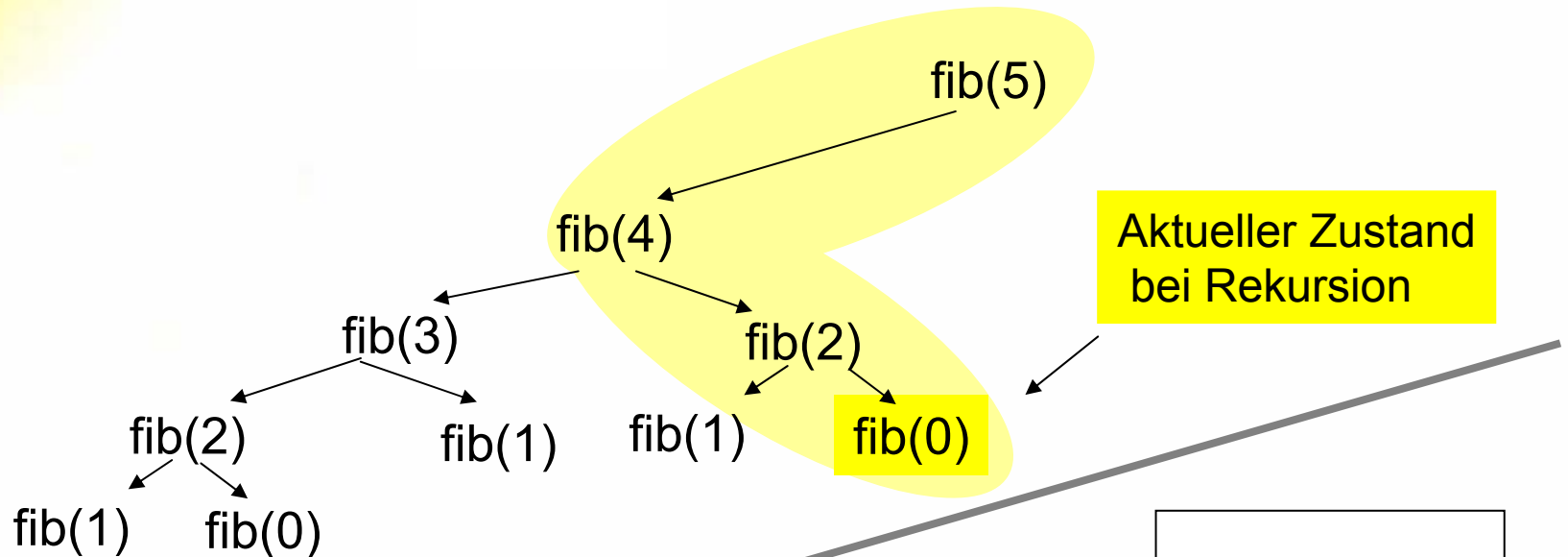
```
public static long fib(long n) {  
    if (n <= 1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}
```

Aufrufbaum:





Verzweigende Rekursion



Aktueller Zustand bei Rekursion

0
2
4
5

Laufzeitkeller enthält alle lokalen Variablen vom „aktuellen“ Knoten bis zur Wurzel



Fibonacci-Zahlen

- **Lösung ineffizient**
- $\text{fib}(n)$ kann einfach berechnet werden, wenn zwei Vorgänger $\text{fib}(n-1)$ und $\text{fib}(n-2)$ bekannt sind
- **Dynamisches Programmieren** anwenden

Schrittweise Verfeinerung

```
public static long fib(long n) {  
  
    // fib(k) schrittweise berechnen bis n = k  
  
}
```



Fibonacci-Zahlen

- **Lösung ineffizient**
- `fib(n)` kann einfach berechnet werden, wenn zwei Vorgänger `fib(n-1)` und `fib(n-2)` bekannt sind
- **Dynamisches Programmieren** anwenden

Schrittweise Verfeinerung

```
public static long fib(long n) {  
    // lokale Variablen für beiden Vorgänger  
    // lokale Variable für Fibonacci Zahl von k  
  
    // Fibonacci Zahl fib(k) schrittweise aus Vorgänger  
    // berechnen bis n = k ist  
  
    // fib(k) zurückgeben  
}
```



Fibonacci-Zahlen

- Lösung ineffizient
- $\text{fib}(n)$ kann einfach berechnet werden, wenn zwei Vorgänger $\text{fib}(n-1)$ und $\text{fib}(n-2)$ bekannt sind
- **Dynamisches Programmieren** anwenden

Schrittweise Verfeinerung

```
public static long fib(long n) {  
    long fibKMinus2 = 1;  
    long fibKMinus1 = 1;  
    long fibK = 1;  
  
    // Fibonacci Zahl fib(k) schrittweise aus Vorgänger  
    // berechnen bis von k = 2 bis n ist  
  
    return fibK;  
}
```

Fibonacci-Zahlen

- Lösung ineffizient
- $\text{fib}(n)$ kann einfach berechnet werden, wenn zwei Vorgänger $\text{fib}(n-1)$ und $\text{fib}(n-2)$ bekannt sind
- **Dynamisches Programmieren** anwenden

Schrittweise Verfeinerung

```
public static long fib(long n) {
    long fibKMinus2 = 1;
    long fibKMinus1 = 1;
    long fibK = 1;

    for (int k = 2; k <= n; k++) {
        // fib(k) schrittweise aus Vorgänger berechnen
        // Vorgänger jeweils eins "weiterschalten"
    }
    return fibK;
}
```

Fibonacci-Zahlen

- Lösung ineffizient
- $\text{fib}(n)$ kann einfach berechnet werden, wenn zwei Vorgänger $\text{fib}(n-1)$ und $\text{fib}(n-2)$ bekannt sind
- **Dynamisches Programmieren** anwenden

Schrittweise Verfeinerung

```
public static long fib(long n) {  
    long fibKMinus2 = 1;  
    long fibKMinus1 = 1;  
    long fibK = 1;  
  
    for (int k = 2; k <= n; k++) {  
        fibK = fibKMinus1 + fibKMinus2;  
        // Vorgänger jeweils eins "weitchalten"  
    }  
    return fibK;  
}
```



Fibonacci-Zahlen

- ***Lösung nun wesentlich effizienter***
 - fib(n) kann einfach berechnet werden, wenn zwei Vorgänger fib(n-1) und fib(n-2) bekannt sind
 - **Dynamisches Programmieren** anwenden
-

```
public static long fib(long n) {
    long fibKMinus2 = 1;
    long fibKMinus1 = 1;
    long fibK = 1;

    for (int k = 2; k <= n; k++) {
        fibK = fibKMinus1 + fibKMinus2;
        fibKMinus1 = fibKMinus2;
        fibKMinus2 = fibK;
    }

    return fibK;
}
```



Inhalt

- Rekursion
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - Verzweigende Rekursion
 - **Verschachtelte Rekursion**
 - Türme von Hanoi
- Backtracking (Rückverfolgung)
 - Wegsuche
 - Springerproblem



Verschachtelte Rekursion

- Parameter eines rekursiven Aufrufs sind selbst wieder rekursive Aufrufe
- Beispiel: *Ackermann-Funktion*

$$\text{ack}(x, 0) := x + 1$$

$$\text{ack}(0, y) := \text{ack}(1, y-1) \text{ für } y > 0$$

$$\text{ack}(x, y) := \text{ack}(\mathbf{ack}(x-1, y), y-1) \text{ für } x, y > 0$$

Verschachtelte Rekursion

- Werte der Ackermann Funktion

ack	x=0	x=1	x=2	x=3	x=4	x=5
y=0	1	2	3	4	5	6
y=1	2	3	4	5	6	7
y=2	3	5	7	9	11	13
y=3	5	13	29	61	125	253
y=4	13	??				

java.lang.StackOverflowError

Verschachtelte Rekursion

- Anzahl rekursive Aufrufe Ackermann Funktion

ack	x=0	x=1	x=2	x=3	x=4	x=5
y=0	1	1	1	1	1	1
y=1	2	4	6	8	10	12
y=2	5	14	27	44	65	90
y=3	15	106	541	2432	10307	42438
y=4	107	??				

← Zu viele rekursive Aufrufe
(Laufzeitkeller voll)



Rekursion

- Verschachtelte Rekursion
 - vermeiden
 - von rein theoretischem Interesse
- Lineare Rekursion
 - Iterativen Problemlösung vorziehen, falls rekursives Programm schwieriger zu verstehen
 - Lineare Rekursion etwas weniger effizient (durch Laufzeitkeller, Methodenaufrufe)
- Verzweigende Rekursion
 - vermeiden, wenn Berechnungen wiederholt werden
 - Dynamisches Programmieren anwenden



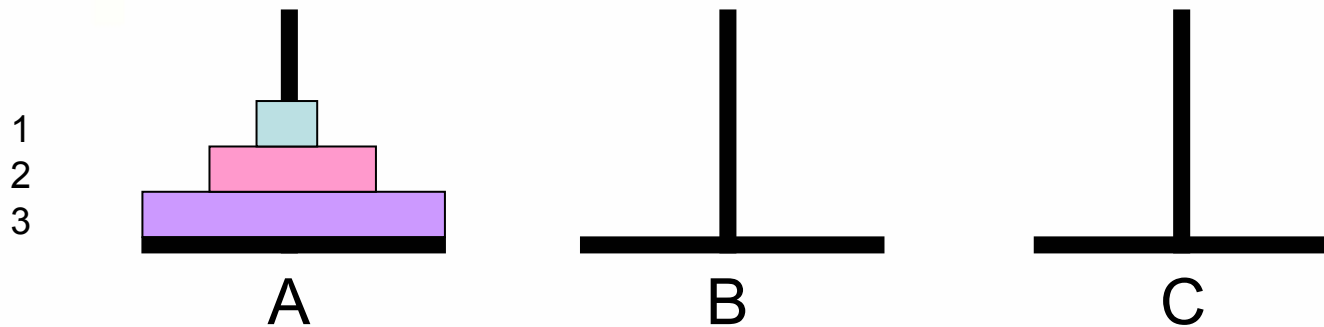
Inhalt

- Rekursion
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - Verzweigende Rekursion
 - Verschachtelte Rekursion
 - **Türme von Hanoi**
- Backtracking (Rückverfolgung)
 - Wegsuche
 - Springerproblem

Rekursion / Türme von Hanoi

Gegeben

- Drei Stäbe und n unterschiedlich grosse Scheiben
- Anfangssituation mit pyramidenförmigen Turm aller n Scheiben auf ersten Stab

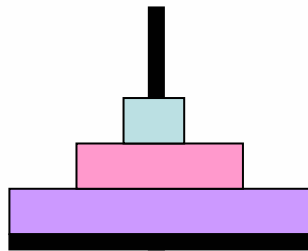


Gesucht: Umbau des ersten Turm als Turm auf zweiten Stab

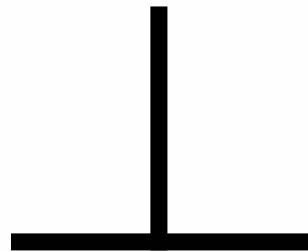
1. Nur oberste Scheibe darf von einem Stab auf einen anderen versetzt werden
2. Eine größere Scheibe darf nicht auf einer kleineren liegen



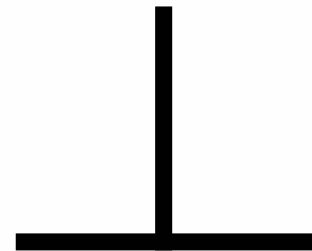
Rekursion / Türme von Hanoi



A



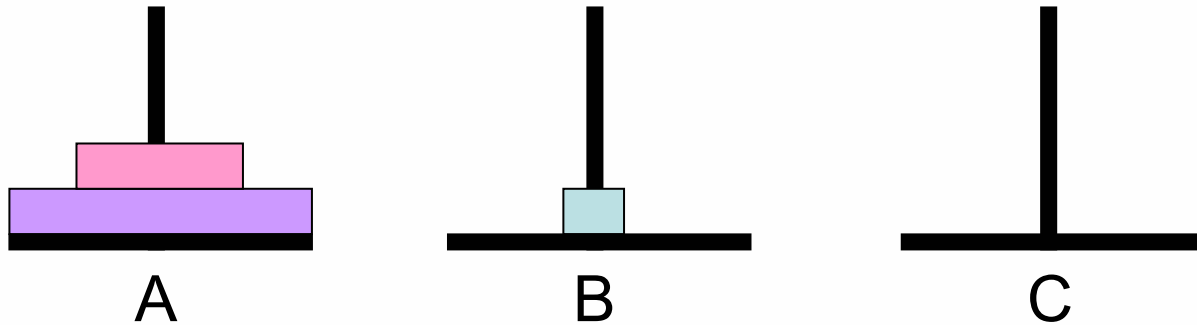
B



C



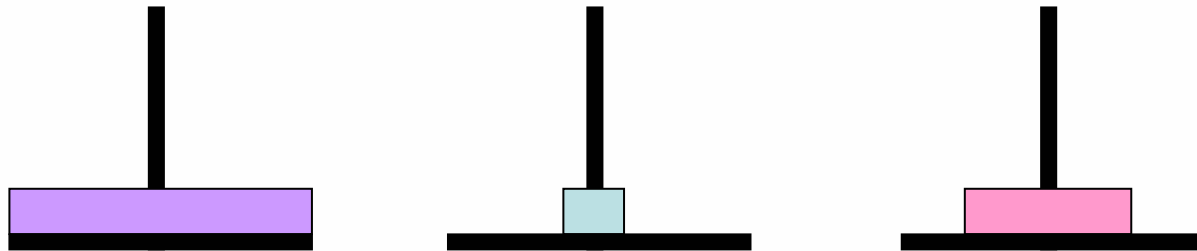
Rekursion / Türme von Hanoi



Scheibe 1 von A auf B



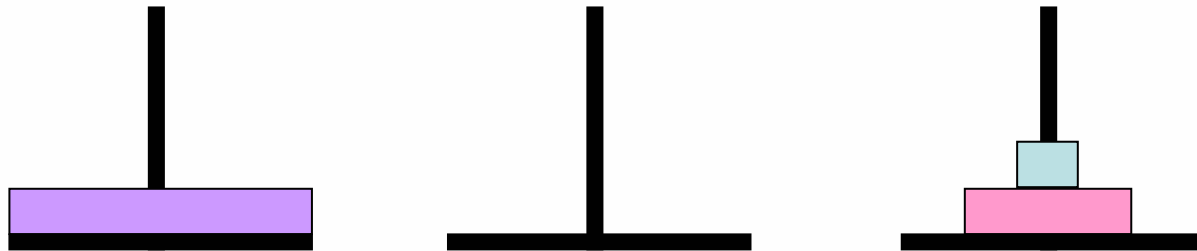
Rekursion / Türme von Hanoi



Scheibe 2 von A auf C



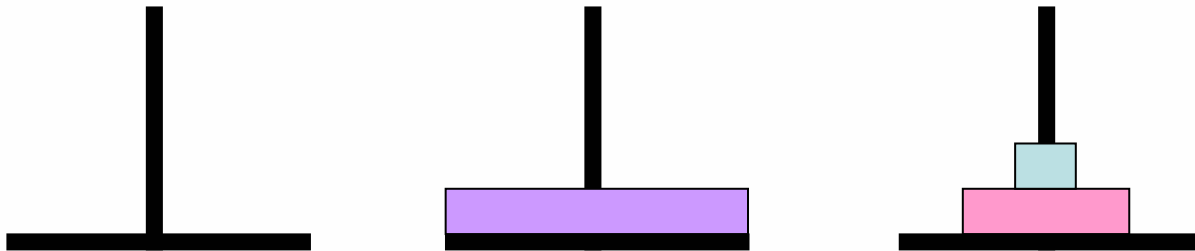
Rekursion / Türme von Hanoi



Scheibe von B auf C



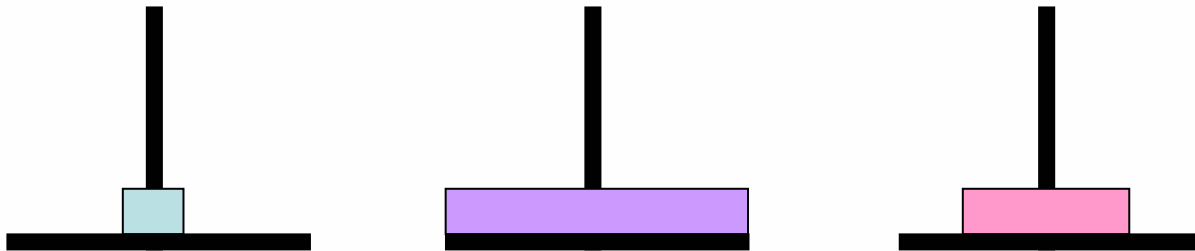
Rekursion / Türme von Hanoi



Scheibe 1 von A auf B



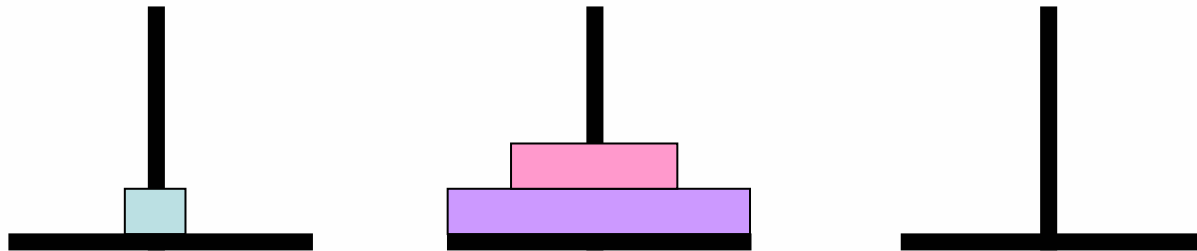
Rekursion / Türme von Hanoi



Scheibe 1 von C auf A



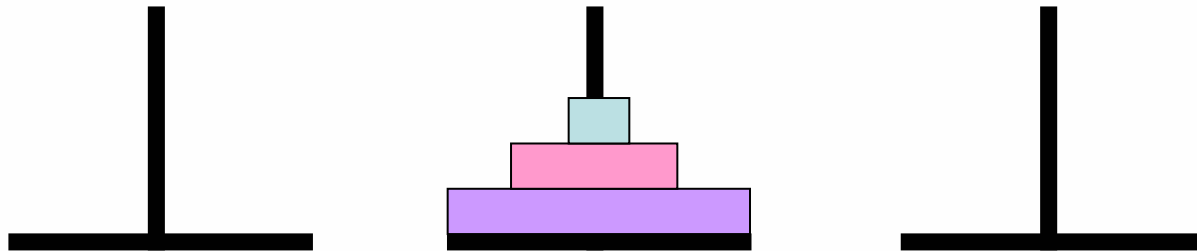
Rekursion / Türme von Hanoi



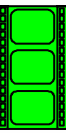
Scheibe 2 von C auf B



Rekursion / Türme von Hanoi

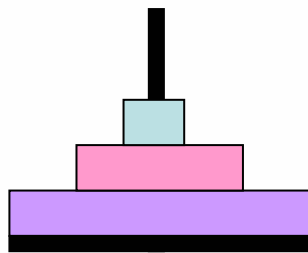


Scheibe 1 von A auf B

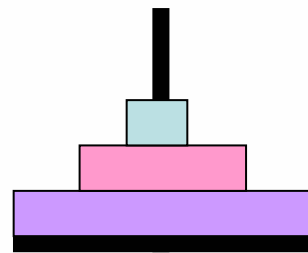


Rekursion / Türme von Hanoi

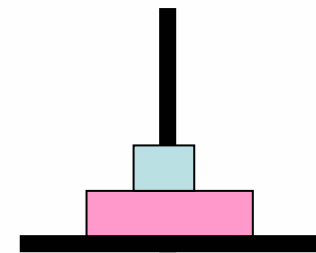
- Oberen zwei Scheiben bilden eine (kleinere) Pyramide
 - Kleinere Pyramide (rekursiv) verschieben
- Idee für rekursive Lösung
 - Obere Pyramide auf Stab C verschieben (rekursiv)
 - Unterste Schreibe von A nach B
 - Obere Pyramide von C auf B verschieben (rekursiv)
- Rekursionsabbruch
 - Pyramide enthält nur eine Scheibe



A



B

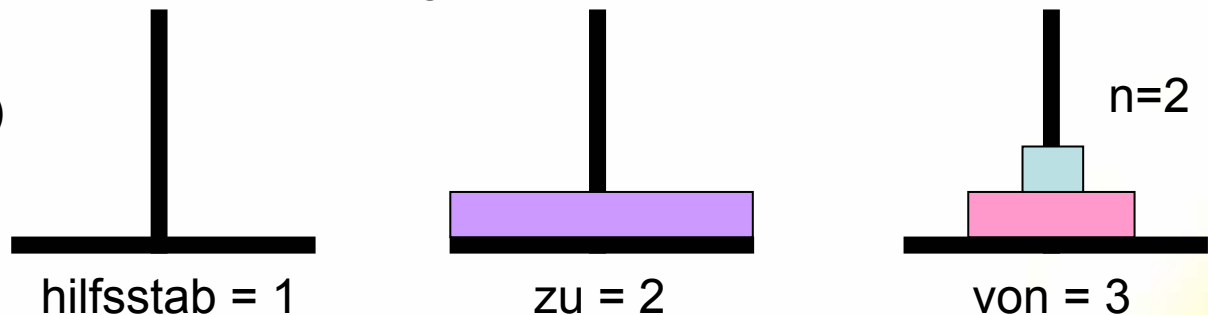


C

Rekursion / Türme von Hanoi

- Implementierung
 - Methode `hanoi(int n, int von, int zu, int hilfsstab)`
- Vier Parameter
 - `n`: Anzahl der Scheiben (oberste hat Nummer `n`)
 - `von`: Nummer Stab von dem Pyramide verschoben wird (A=1, B=2, C=3)
 - `zu`: Nummer Stab zu dem Pyramide verschoben wird
 - `hilfsstab`: Nummer des freien Hilfsstabs
- **Textuelle Ausgabe der Verschiebeoperation**
 - Keine explizite Implementierung der Stäbe und Scheiben

`hanoi(2, 3, 2, 1)`





hanoi (3, 1, 2, 3)

1. (Rekursion)

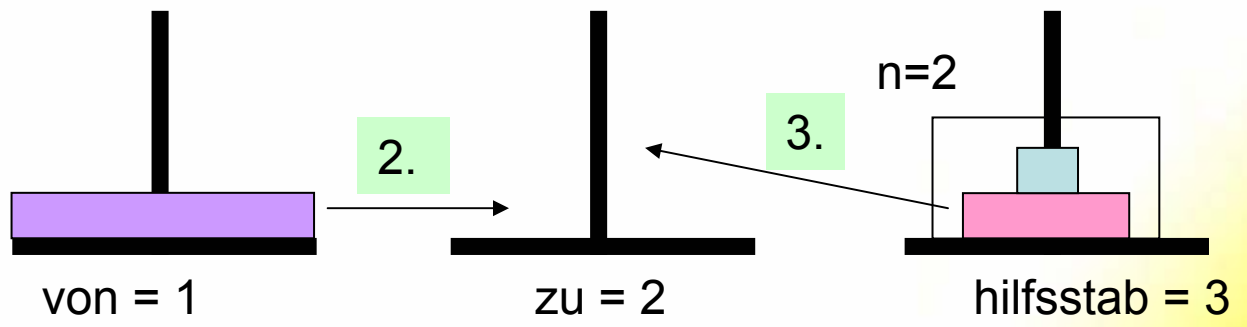
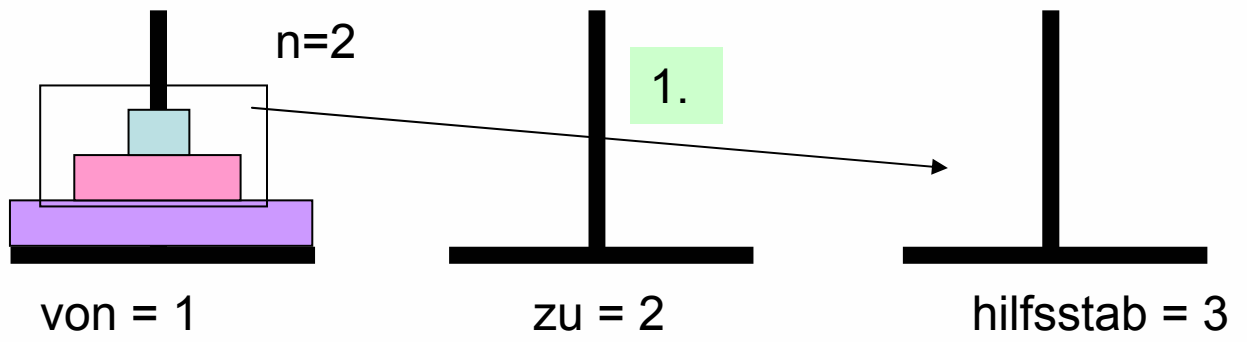
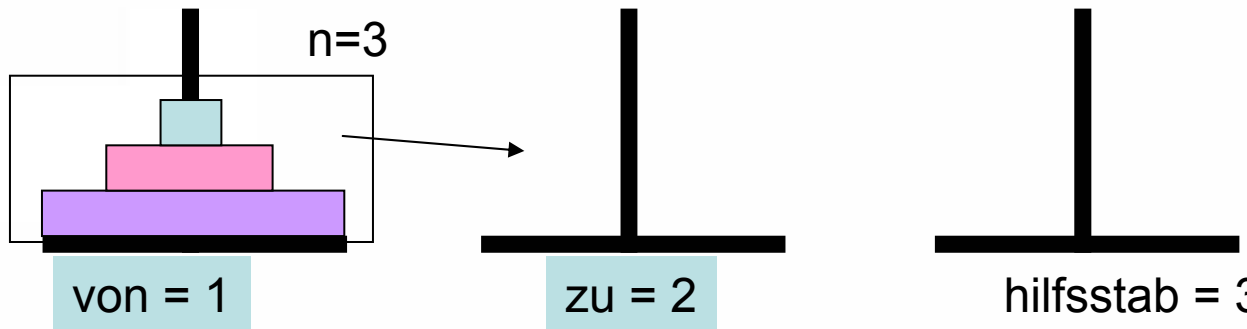
hanoi (2, 1, 3, 2)

2.

direkt von=1 zu=2

3. (Rekursion)

hanoi (2, 3, 2, 1)





Rekursion / Türme von Hanoi

- Rekursionsabbruch
 - Nur eine (kleinste) Scheibe ($n=1$). Diese direkt verschieben.
- Rekursion
 - Obere Pyramide von $n-1$ Scheiben verschieben
 - Unterste Scheibe verschieben
 - Obere Pyramide von $n-1$ auf untere Scheibe verschieben

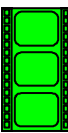
```
void hanoi(int n, int von, int zu, int hilfe) {  
    if (n == 1) {  
        System.out.println(n + ": " + von + "-->" + zu);  
    } else {  
        hanoi(n - 1, von, hilfe, zu);  
        System.out.println(n + ": " + von + "-->" + zu);  
        hanoi(n - 1, hilfe, zu, von);  
    }  
}
```



Rekursion / Türme von Hanoi

- Kürzere Variante:

```
void hanoi(int n, int von, int zu, int hilfe) {  
    if (n > 0) {  
        hanoi(n - 1, von, hilfe, zu);  
        System.out.println(n + ": " + von + "-->" + zu);  
        hanoi(n - 1, hilfe, zu, von);  
    }  
}
```

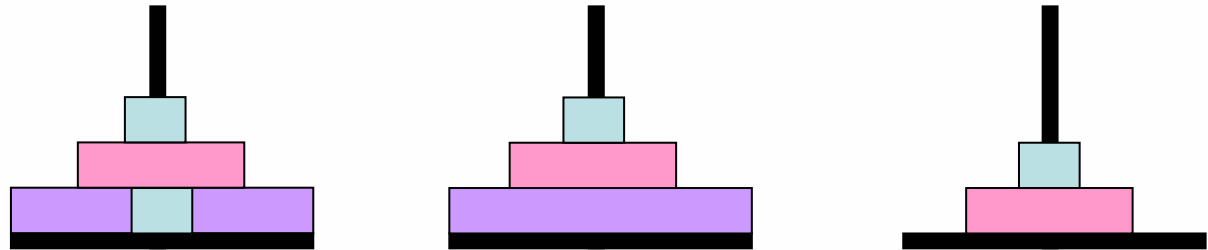


Rekursion / Türme von Hanoi

Ausgabe bei Aufruf hanoi(3,1,2,3)

(Pyramide mit 3 Schreibe von A auf B unter Hilfe von C verschieben).

- 1: 1-->2
- 2: 1-->3
- 1: 2-->3
- 3: 1-->2
- 1: 3-->1
- 2: 3-->2
- 1: 1-->2



hanoi(3, 1, 2, 3)

hanoi(2, 1, 3, 2)

hanoi(1, 1, 2, 3)

hanoi(1, 2, 3, 1)

hanoi(2, 3, 2, 1)

hanoi(1, 3, 1, 2)

hanoi(1, 1, 2, 3)

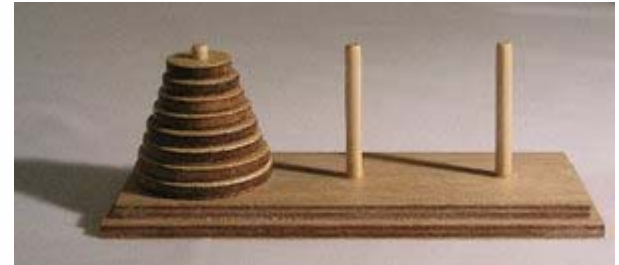
```

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

Rekursion / Türme von Hanoi

- Francois Eduoard Anatole Lucas

- * 4. April 1842 in Amiens
- † 3. November 1891 in Paris
- französischer Mathematiker



- Erfand vermutlich „Türme von Hanoi“ als Geduldsspiel und folgende Geschichte dazu

- Indische Mönche im Tempel zu Benares müssen einen Turm aus 64 goldenen Scheiben versetzen
- Wenn ihnen das gelungen sei, wäre das Ende der Welt gekommen.



- Minimale Anzahl Züge bei n Scheiben: $2^n - 1$

- Bei 64 Scheiben: viele Milliarden Jahre nötig



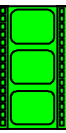
Inhalt

- **Rekursion**
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - Verzweigende Rekursion
 - Verschachtelte Rekursion
 - Türme von Hanoi
- **Backtracking (Rückverfolgung)**
 - Wegsuche
 - Springerproblem



Rekursion / Backtracking

- Backtracking (Rückverfolgung)
 - Rekursive Lösungsstrategie nach Prinzip „Versuch und Irrtum“ (*trial and error*)
- Solange Lösung noch nicht gefunden
 1. Teillösung um einen Schritt erweitern und rekursiv zu Gesamtlösung erweitern (trial)
 2. Falls keine Gesamtlösung gefunden (error)
Teillösungen verwerfen und weiter mit 1.
(Teillösung um anderen Schritt erweitern)

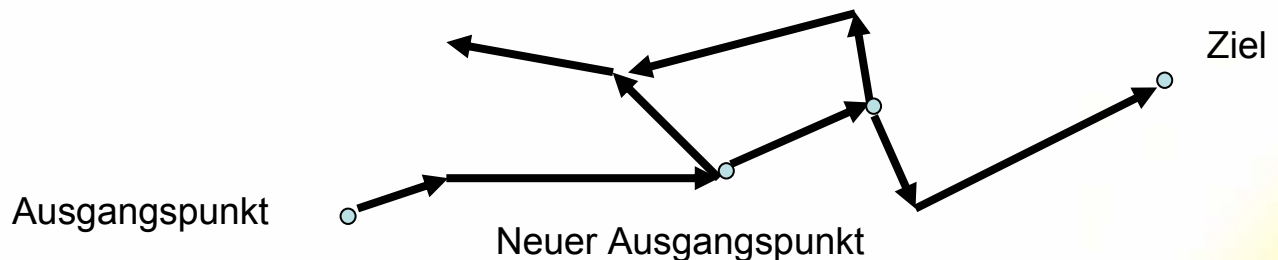


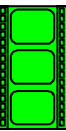
Rekursion / Backtracking

Im Alltag: Suche einer bestimmten Strasse in einer fremden Stadt (ohne Navigationssystem)

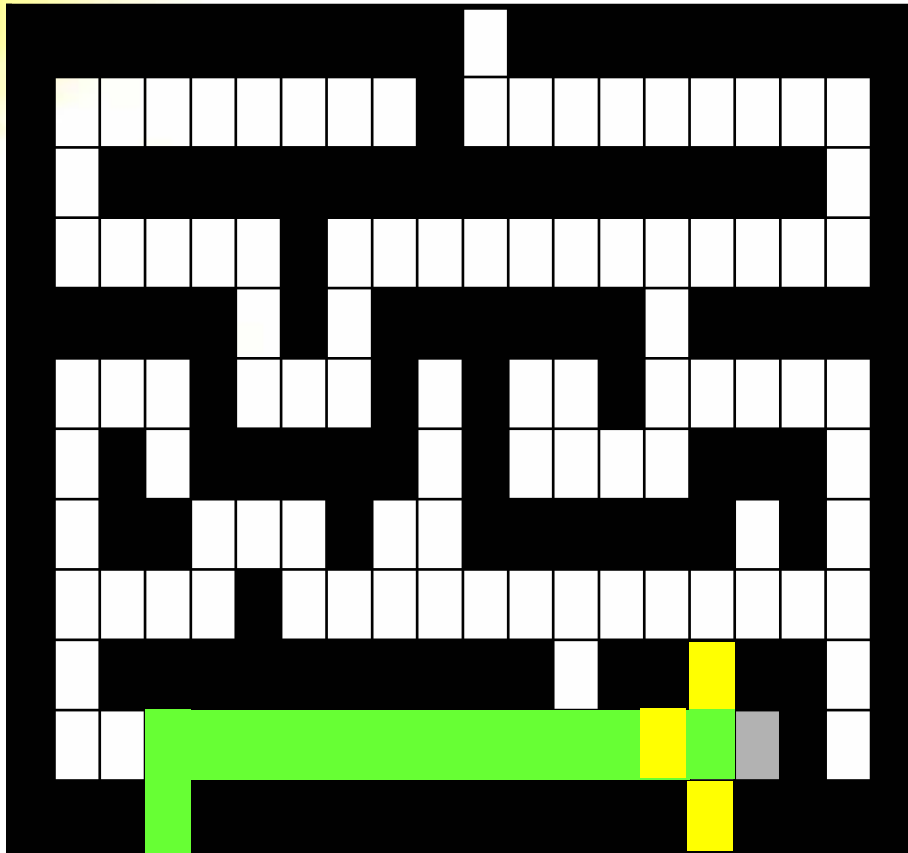
Solange Ziel (Strasse) noch nicht gefunden:

1. An derzeitigem Ausgangspunkt:
Frage nach richtigem Weg (Heuristik),
dem beschriebenen Weg folgen (Teillösung)
2. Falls Weg nicht zum Ziel führte
(Sackgasse, Verlaufen,
Wegbeschreibung vergessen oder falsch)
 - ein Stück zurückgehen (Backtrack, Teillösung verwerfen)
 - Weiter mit Schritt 1 (Rekursion)





Backtracking / Wegsuche im Labyrinth



1: Gehe einen Schritt in eine Richtung

2: Suche Ausgang von dort

3: Falls Ausgang gefunden

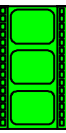
- fertig

Falls nicht

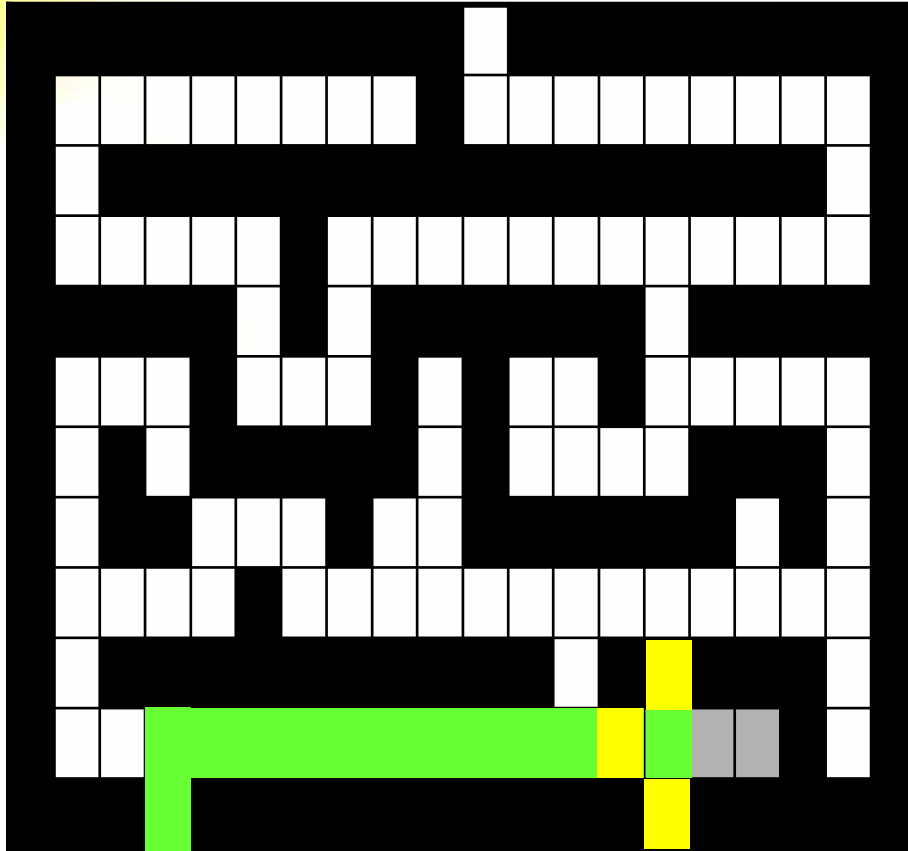
- Verwerfe Lösung von Schritt 2

- Gehe Schritt in eine andere

Richtung und weiter mit 2



Backtracking / Wegsuche im Labyrinth



1: Gehe einen Schritt in eine Richtung

2: Suche Ausgang von dort

3: Falls Ausgang gefunden

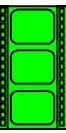
- fertig

Falls nicht

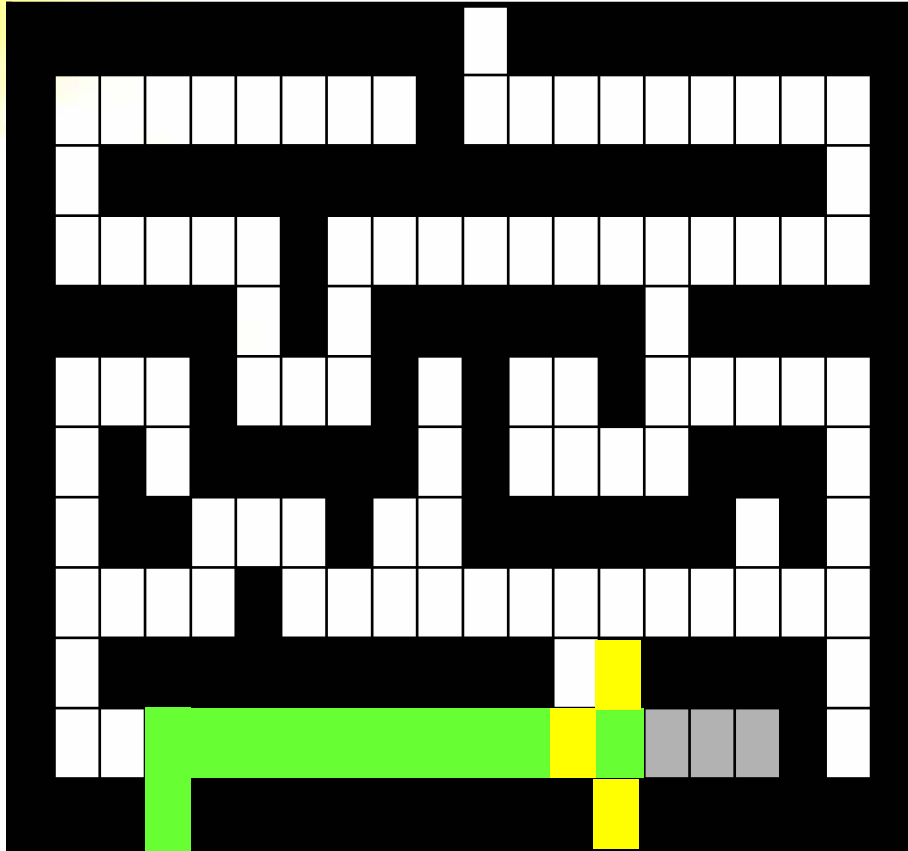
- Verwerfe Lösung von Schritt 2

- Gehe Schritt in eine andere

Richtung und weiter mit 2



Backtracking / Wegsuche im Labyrinth



1: Gehe einen Schritt in eine Richtung

2: Suche Ausgang von dort

3: Falls Ausgang gefunden

- fertig

Falls nicht

- Verwerfe Lösung von Schritt 2

- Gehe Schritt in eine andere

Richtung und weiter mit 2

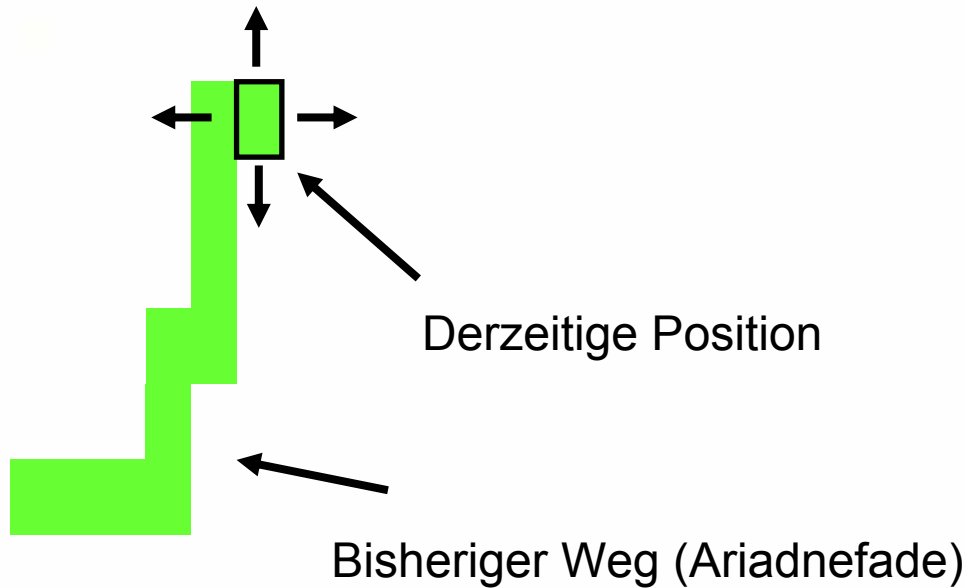
Wegsuche im Labyrinth / Ariadne

- Ariadne (griech. Mythologie)
 - Tochter des Minos (von Kreta)
 - Minos forderte von Athener regelmässige Menschenopfer als Vergeltung für den Tod an seinem Sohn und als Futter für den Minotaurus
 - Theseus wurde (als Opfer) in das Labyrinth des Minos gesendet
 - Ariadne verliebte sich in Theseus und gab ihm einen roten Faden (und ein Schwert), damit er wieder aus dem Labyrinth zurückfindet
 - Theseus erschlug den Minotaurus und floh (ohne Ariadne) von Kreta





Backtracking / Wegsuche im Labyrinth





Backtracking / Wegsuche im Labyrinth

- Zwei dimensionales Feld, Typ char
 - 'W' = Wand
 - ' ' = Gang (Konstante WEG)
 - '.' = bisher gefundener Weg
 - '-' = markierte Sackgasse (Konstante MARKIERT)
- labyrinth[y][x] = Position x,y im Labyrinth

- Eingang
 - Eine Startposition x, y

- Ausgang
 - int ausgangX
 - int ausgangY

Labyrinth

-labyrinth : char [] []

-ausgangX : int

-ausgangY : int

// Konstanten

// ggf. weitere Eigenschaften

+print() : void

+sucheAusgang(x : int, y : int) : boolean

// weitere Methoden

Backtracking / Allgemeiner Algorithmus

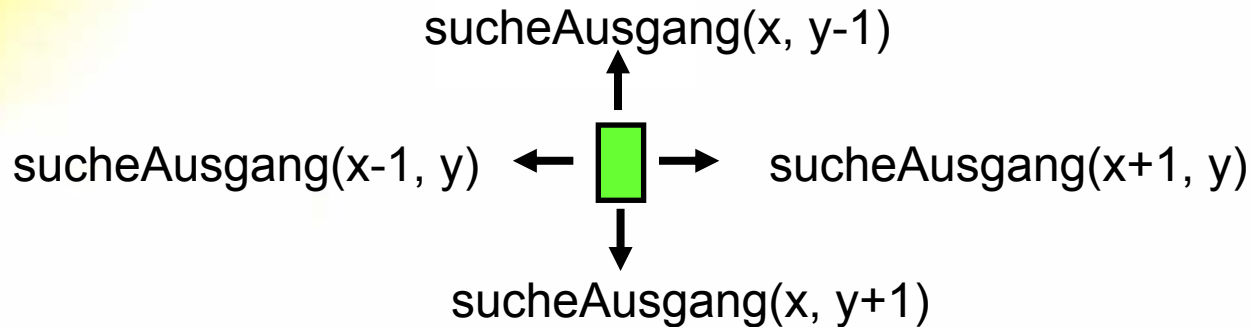
```

boolean sucheLoesung( /* Parameter, um einen Schritt zu merken */ ) {
    if ( /* Noch nicht fertig (Sonderfall) */ )
        while (ex. noch Teillösungsschritte */ ) {
            // wähle nächsten Schritt
            if ( /* Schritt ist gültig */ ) {
                // Erweitere Lösung
                if ( /* Lösung noch nicht vollständig */ ) {
                    if ( sucheLoesung( /* neuer Schritt */ ) ) {
                        return true;
                    } else {
                        // Mache Schritt rückgängig
                    }
                } else {
                    return true;
                }
            }
        }
    }
    return false;
} else {
    return true;
}
}

```



Backtracking / Wegsuche im Labyrinth



Rekursionsabbruch: wenn `x == ausgangX && y == ausgangY` (true zurück)

Ansonsten nächsten Schritt wählen:

zum Beispiel `x1 = x - 1, y1 = y` für Schritt nach links

Gültigkeit des Schritts überprüfen:

`y1, x1` gültiger Index und `labyrinth[y1][x1] == WEG`

Lösung um nächsten Schritt erweitern:

`labyrinth[x1][y1] = '.';`

Rekursion:

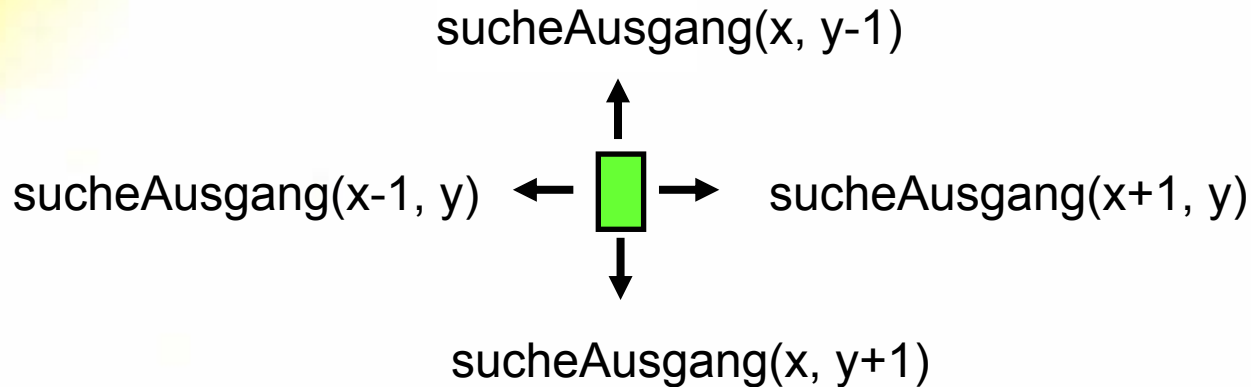
`sucheAusgang(x1, y1)` aufrufen

Falls rekursiv keine Lösung gefunden: **Schritt rückgängig machen**

`labyrinth[y1][x1] = MARKIERT;`



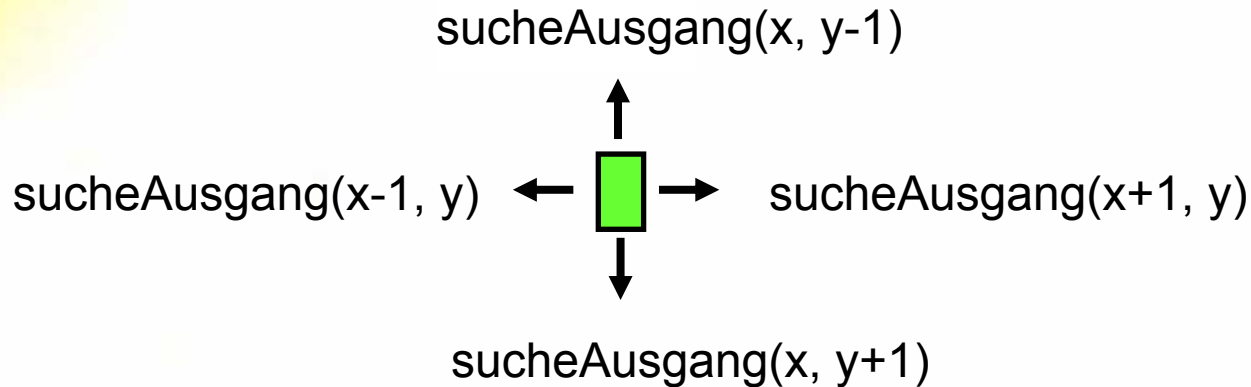
Backtracking / Wegsuche im Labyrinth



Rekursionsabbruch: wenn $x == \text{ausgangX} \ \&\& \ y == \text{ausgangY}$ (true zurück)

```
public boolean ausgangGefunden(int x, int y) {  
    return x == ausgangX && y == ausgangY;  
}
```

Backtracking / Wegsuche im Labyrinth



Ansonsten **nächsten Schritt wählen:**

zum Beispiel $x_1 = x - 1$, $y_1 = y$ für Schritt nach links

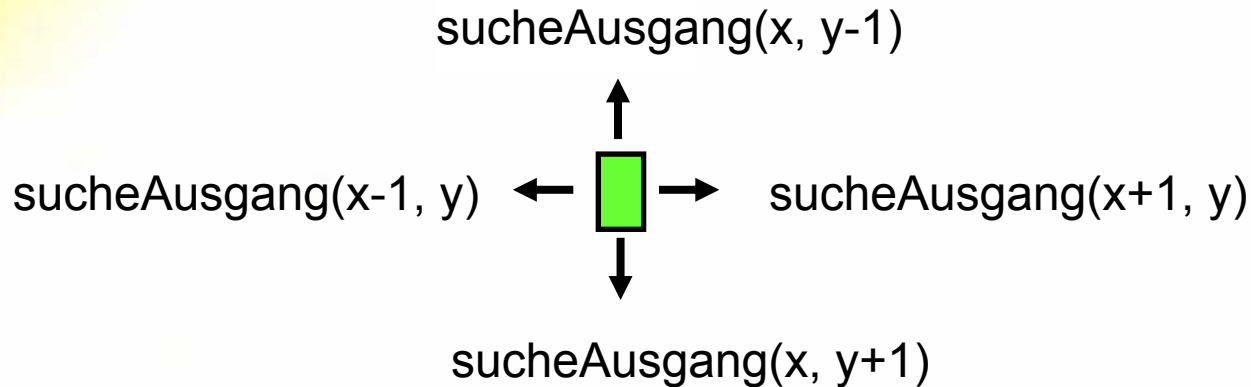
```
public static final int DX[] = {1, 0, -1, 0};
```

```
public static final int DY[] = {0, -1, 0, 1};
```

```
for (int i=0; ! ausgangGefunden(x, y) && i<4; i++) {
    int y1 = y + DY[i];
    int x1 = x + DX[i];
    // schritt überprüfen, Lösung erweitern, usw.
}
```



Backtracking / Wegsuche im Labyrinth



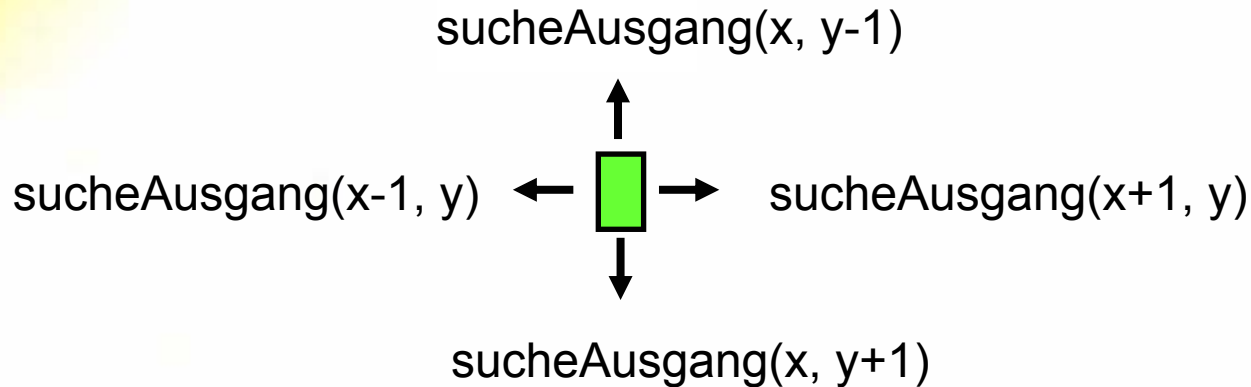
Gültigkeit des Schritts überprüfen:

$y1, x1$ gültiger Index und `labyrinth[y1][x1] == ' '`

```
public boolean freierPlatz(int x, int y) {
    return 0 <= y && y < labyrinth.length
        && 0 <= x && x < labyrinth[y].length
        && ( labyrinth[y][x] == WEG);
}
```



Backtracking / Wegsuche im Labyrinth



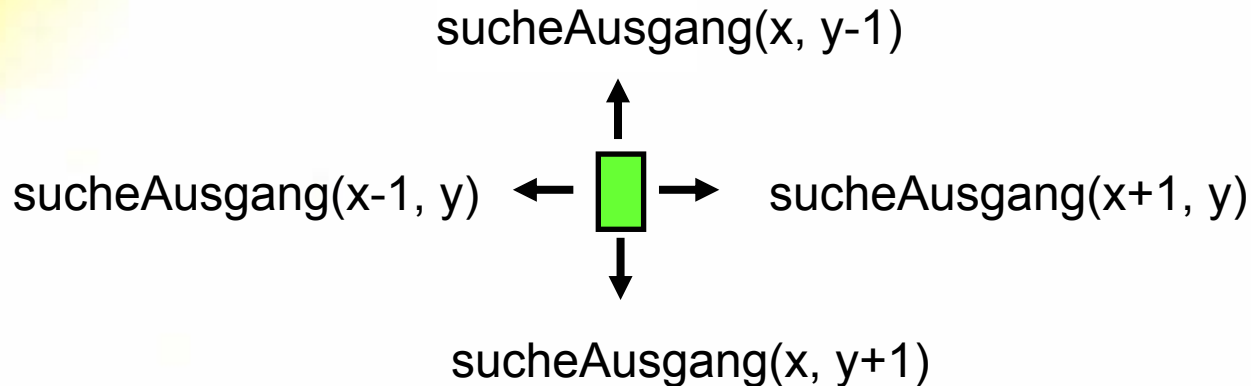
Lösung um nächsten Schritt erweitern:

```
labyrinth[x1][y1] = '.';
```

```
public void erweitereLoesung(int x, int y) {  
    labyrinth[y][x] = '.';  
}
```



Backtracking / Wegsuche im Labyrinth



```
public void alsSackgasseMarkieren(int y1, int x1) {  
    labyrinth[y1][x1] = MARKIERT;  
}
```

Falls rekursiv keine Lösung gefunden: **Schritt rückgängig machen**
labyrinth[y1][x1] = MARKIERT;

Backtracking / Wegsuche im Labyrinth

```
public boolean sucheAusgang(int x, int y) {
    for (int i=0; i<4; i++) {
        // wähle neuen Schritt
        int y1 = y + richtungenDY[i];
        int x1 = x + richtungenDX[i];
        // Schritt gültig?
        if ( freierPlatz(x1, y1) ) {
            erweitereLoesung(x1, y1);
            // überprüfe Lösung
            if ( ! ausgangGefunden(x1, y1) ) {
                // rekursiver Aufruf
                if ( sucheAusgang(x1, y1) ) {
                    return true;
                } else {
                    // mache Schritt rückgängig
                    alsSackgasseMarkieren(y1,x1);
                }
            } else {
                return true;
            }
        }
    }
    return false;
}
```

Backtracking / Wegsuche im Labyrinth

```

Labyrinth labyrinth = new Labyrinth(
    new char[][] {
        {'W', 'W', 'W', 'W', 'W', ' ', 'W', 'W', 'W', 'W', 'W'},
        {'W', ' ', ' ', ' ', ' ', 'W', ' ', 'W', ' ', ' ', ' ', 'W'},
        {'W', ' ', ' ', 'W', 'W', 'W', ' ', ' ', ' ', 'W', ' ', 'W'},
        {'W', ' ', ' ', 'W', ' ', 'W', 'W', 'W', 'W', 'W', ' ', 'W'},
        {'W', ' ', ' ', 'W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'W'},
        {'W', ' ', ' ', 'W', 'W', ' ', 'W', 'W', ' ', ' ', 'W', 'W', 'W'},
        {'W', ' ', ' ', ' ', 'W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'W'},
        {'W', ' ', ' ', 'W', 'W', ' ', 'W', 'W', 'W', 'W', ' ', 'W'},
        {'W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'W', ' ', ' ', 'W', ' },
        {'W', ' ', ' ', 'W', 'W', 'W', ' ', ' ', 'W', ' ', ' ', 'W', ' },
        {'W', ' ', ' ', 'W', ' ', ' ', ' ', ' ', 'W', ' ', ' ', 'W', ' },
        {'W', 'W', 'W', 'W', 'W', ' ', ' ', 'W', 'W', 'W', 'W', 'W'}
    },
    0, 5);

```

Ausgang

Eingang

```

labyrinth.print();
labyrinth.sucheAusgang(5,12);
labyrinth.print();

```



Backtracking / Wegsuche im Labyrinth

```

WWWWWW  WWWWWW
W      W  W      W
W  WWW      W  W
W  W  WWWWWW  W
W  W              W
W  WW  WW  WWW
W  W              W
W  WW  WWWWW  W
W              W  W  W
W  WWW  W  W  W
W  W      W  W  W
W  W  W  W      W
WWWWWW  WWWWWW
    
```

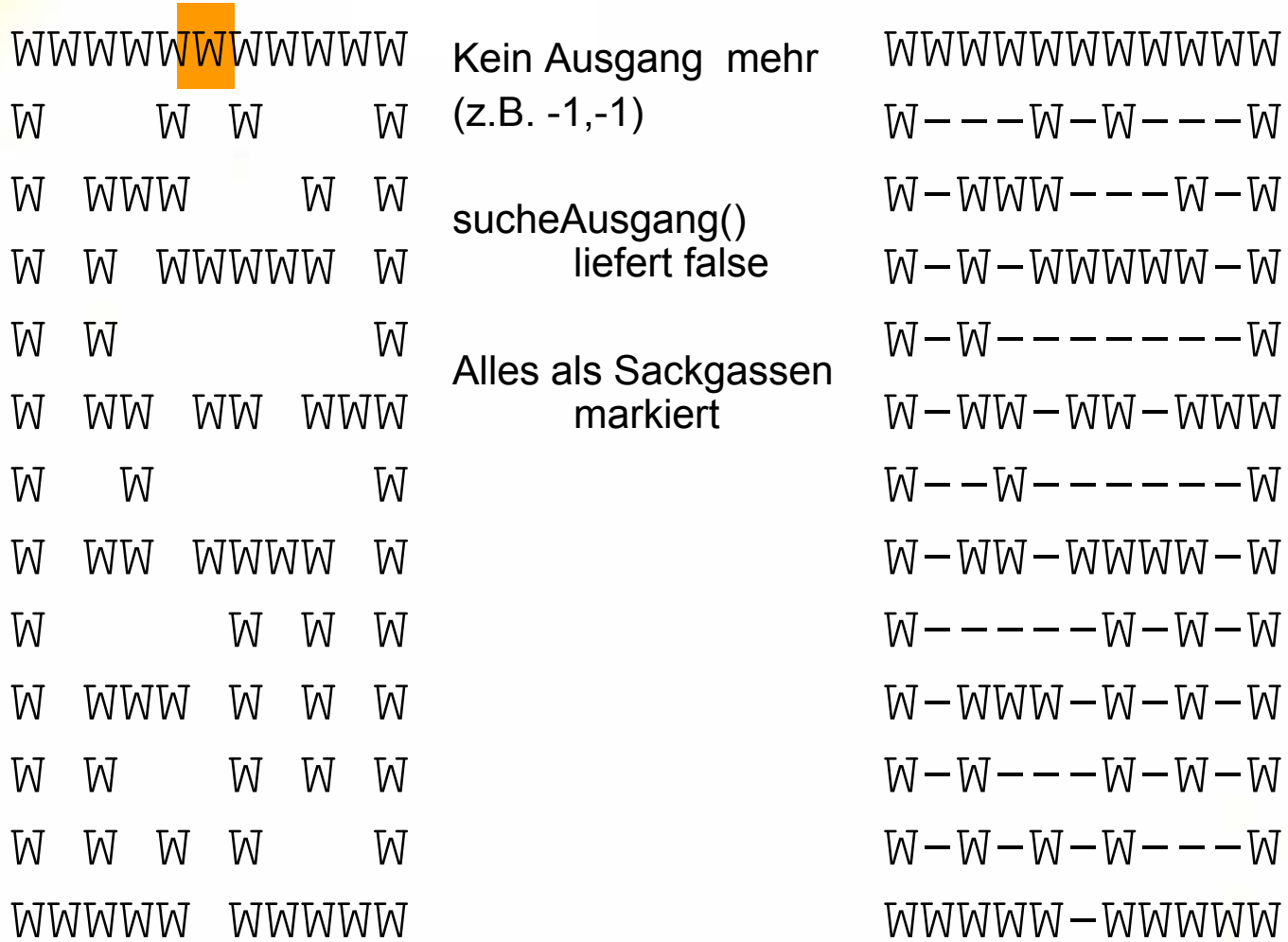


```

WWWWWW  WWWWWW
W      W.W...W
W  WWW...W.W
W  W  WWWWWW.W
W  W              . . . W
W  WW  WW.WWW
W  W...--W
W  WW.WWWW-W
W      . . W-W-W
W  WWW.W-W-W
W  W      .W-W-W
W  W  W.W---W
WWWWWW . WWWWWW
    
```

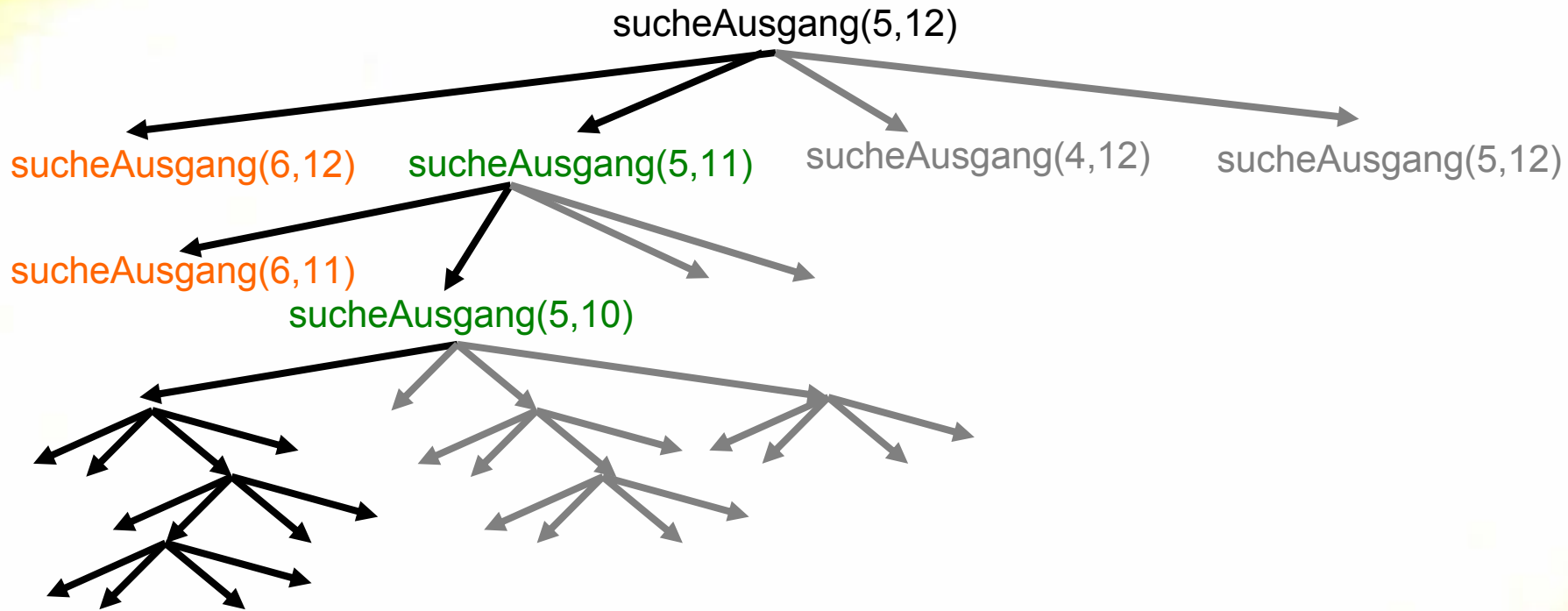


Backtracking / Wegsuche im Labyrinth





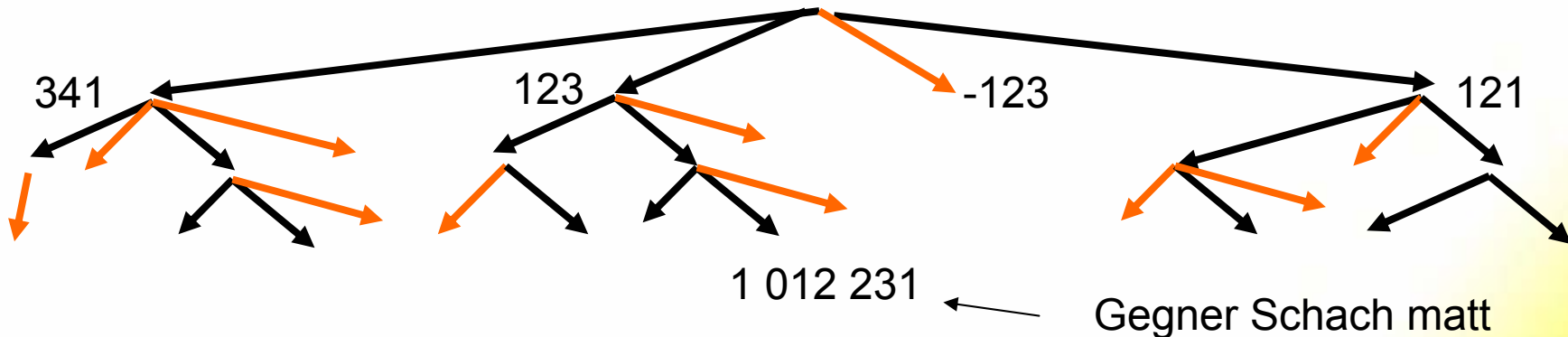
Backtracking / Wegsuche im Labyrinth



Lösung mit tiefem Rekursionsabstieg: „Tiefensuche“

Breitensuche

- Breitensuche (keine Backtracking)
 - Z.B. bei Schachprogrammen, Gesellschaftsspielimplementierungen
 - Menge von Teillösungen berechnen (Knoten, Enden der Pfeile)
 - Teilmenge davon behalten (schwarzen Pfeile) und Lösungen erweitern
 - Lösungen, die nicht gut genug sind, werden verworfen
 - Güte einer Lösung wird mit einer Bewertungsfunktion berechnet
 - Berechnung wird nach einer bestimmten Tiefe / Anzahl Teillösungen gestoppt und die beste Lösung wird durchgeführt
- Schachprogramm: Teillösung ist die nach einem Zug entstehenden Situation
 - Schlechte Lösungen: Doppelbauer (-10), Springer am Rand (-20)
 - Gute Lösungen: Gegner Schach gesetzt (+30) oder gar Matt (+1 000 000), Bauer gegen Dame eingetauscht (+100)



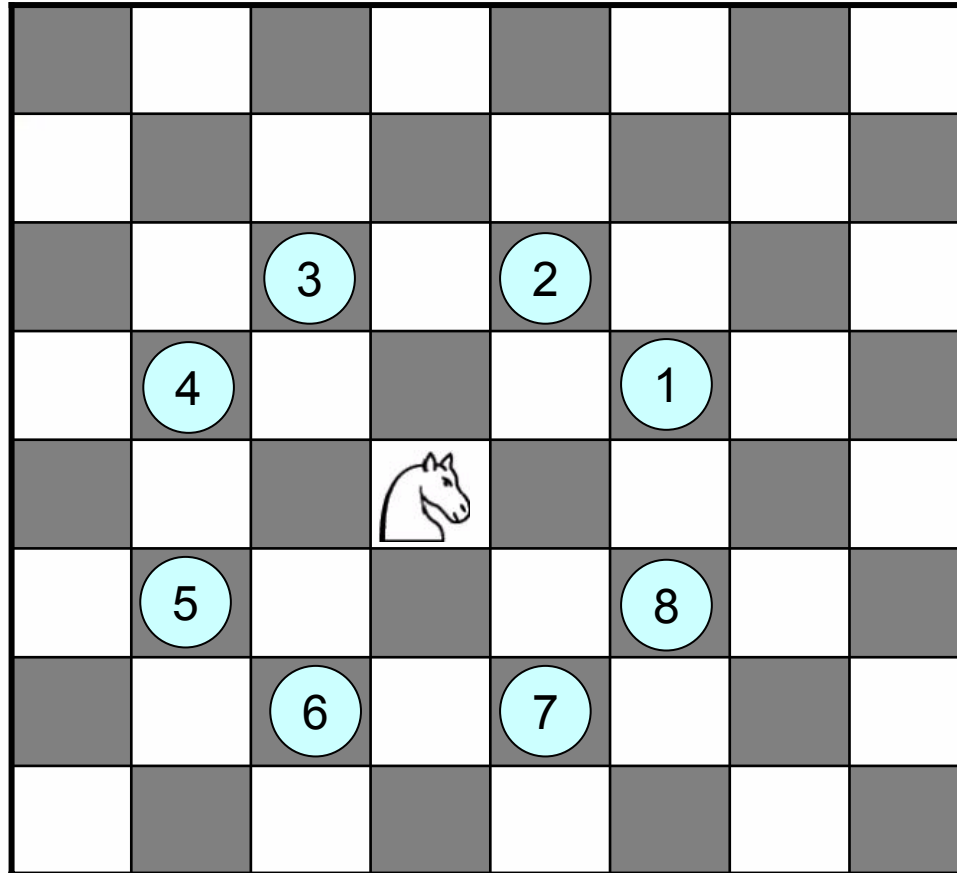


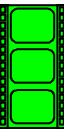
Inhalt

- Rekursion
 - Begriff
 - Mathematische Beispiele
 - Lineare Rekursion
 - Verzweigende Rekursion
 - Verschachtelte Rekursion
 - Türme von Hanoi
- Backtracking (Rückverfolgung)
 - Wegsuche
 - Springerproblem

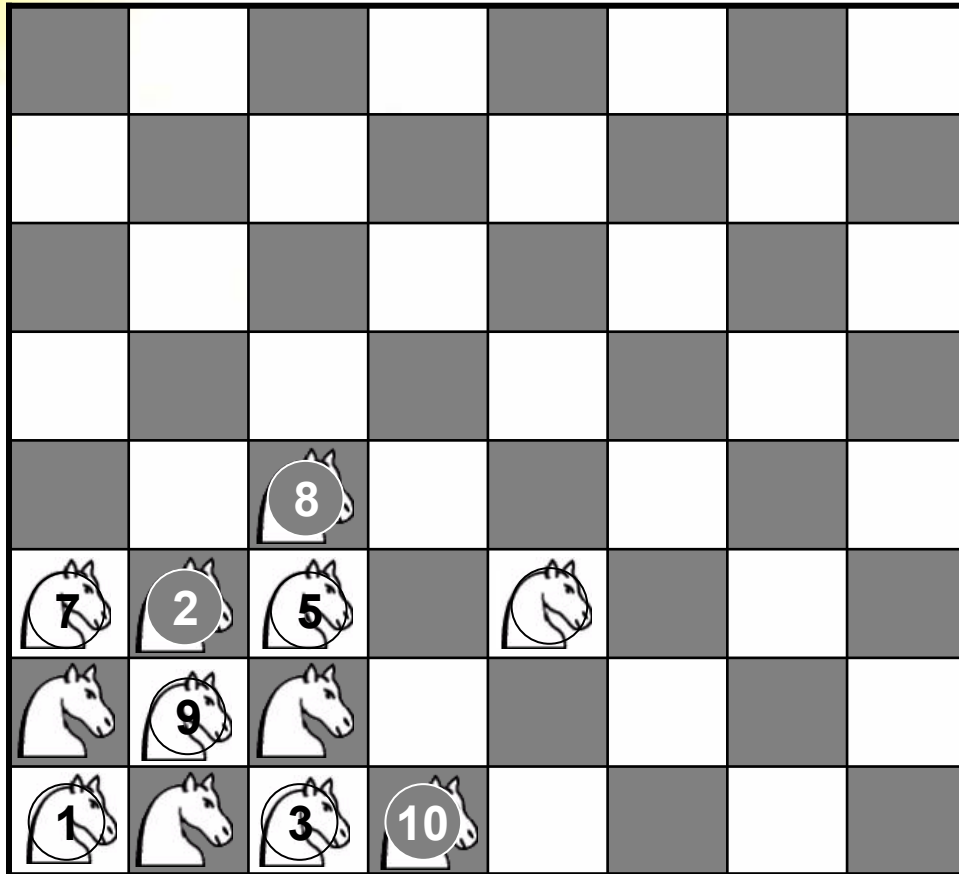


Backtracking / Springerproblem





Backtracking / Springerproblem



- 64 Felder (8x8)
- Springer muss jedes Feld genau einmal besuchen
- Kein Feld zweimal besuchen
- Variante: Letzter Zug muss wieder auf Ausgangsfeld führen



Backtracking / Springerproblem

3	12	7	18	25
6	17	4	13	8
11	2	19	24	21
16	5	22	9	14
1	10	15	20	23

Backtracking / Springerproblem

- Leonhard Euler

- * 15. April 1707 in Basel

- † 18. September 1783 in St. Petersburg

- Mathematiker, Physiker

- Hunderte von Arbeiten

- Notationen und Symbole: \sum , i , $f(x)$, ...

- Königsberger Brückenproblem (Euler Kreis)

- Springerproblem, 1759

- ähnlich Euler Kreis (Knoten statt Kanten)

- als Hamiltonkreis bekannt





Backtracking / Springerproblem

SpringerProblem

loesung : int [] []

n : int

SpringerProblem(n : int)

print() : void

sucheSpringerFolge(x : int, y : int, zaehler : int)
: boolean

3	12	7	18	25
6	17	4	13	8
11	2	19	24	21
16	5	22	9	14
1	10	15	20	23

- Problem für beliebige Brettgrösse n
- Brett zwei-dimensionales Feld „loesung“
 - Feldwerte sind gesuchte Nummern der Springerfolge
 - 0, falls Feld unbesucht

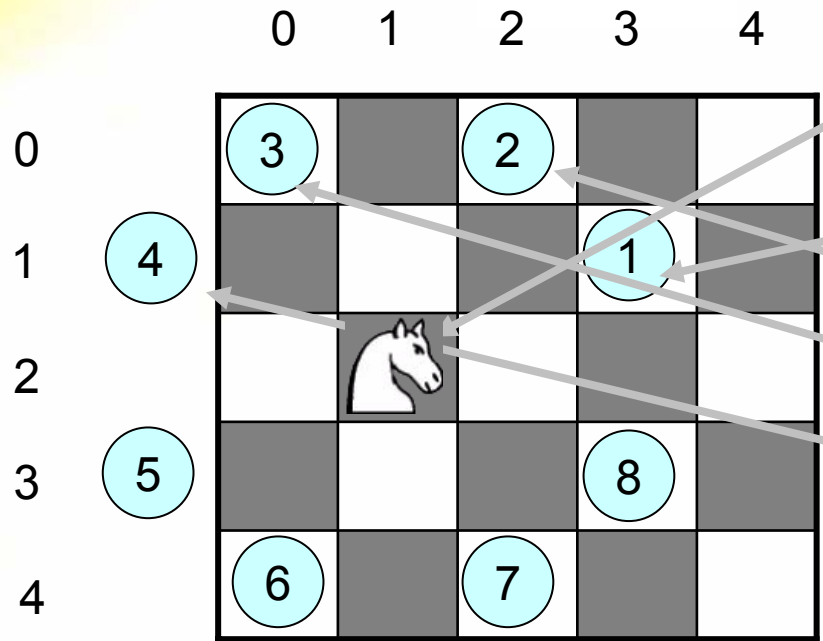
Backtracking / Springerproblem

```

boolean sucheSpringerFolge (int x, int y, int zaehler) {
  while ( /* ex. noch Teillösungsschritte */ ) {
    // wähle nächsten Schritt
    int x1 = ??
    int y1 = ??
    if ( istGueltigerSprung(x1,y1) ) {
      // Erweitere Lösung
      loesung[x1][y1] = zaehler;
      if ( zaehler < n * n ) {
        if ( sucheSpringerFolge (x1,y1, zaehler + 1) ) {
          return true;
        } else {
          // mache Schritt rückgängig
          loesung[x1][y1] = 0;
        }
      } else {
        return true;
      }
    }
  }
  return false;
}

```

Backtracking / Springerproblem



Springer an Position x, y

Mögliche neuen Positionen $x1, y1$:

$x+2, y-1$

$x+1, y-2$

$x-1, y-2$

$x-2, y-1$

```
int offset[][] = {
    { 2, -1},
    { 1, -2},
    {-1, -2},
    {-2, -1},
    {-2, 1},
    {-1, 2},
    { 1, 2},
    { 2, 1}
};
```

```
boolean istGeltigerSprung(int x, int y) {
    return (0 <= x && x < n)
        && (0 <= y && y < n)
        && loesung[x][y] == 0;
}
```

```
x1=x+offset[i][0];
y1=y+offset[i][1];
```

Backtracking / Springerproblem

```

boolean sucheSpringerFolge (int x, int y, int zaehler) {
  for (int i = 0; i < offset.length; i++ ) {
    // wähle nächsten Schritt
    int x1 = x + offset[i][0];
    int y1 = y + offset[i][1];
    if ( istGueltigerSprung(x1,y1) ) {
      // Erweitere Lösung
      loesung[x1][y1] = zaehler;
      if ( zaehler < n * n ) {
        if ( sucheSpringerFolge (x1,y1, zaehler + 1) ) {
          return true;
        } else {
          // mache Schritt rückgängig
          loesung[x1][y1] = 0;
        }
      } else {
        return true;
      }
    }
  }
  return false;
}

```



Backtracking / Springerproblem

```
public static void main(String argv[]) {  
    SpringerProblem sp = new SpringerProblem(n);  
  
    sp.loesung[n-1][0] = 1;           // 1. Schritt notieren  
    sp.sucheSpringerFolge(n-1,0,2); // Methode sucht nächsten Schritt  
    sp.print();  
}
```



Backtracking / Springerproblem

Ausgabe für $n=5$

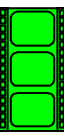
3	12	7	18	25
6	17	4	13	8
11	2	19	24	21
16	5	22	9	14
1	10	15	20	23

n	Zeit	Sackgassen
5	<1 sek	263
6	<1 sek	177012
7	<1 sek	132967
8	Stunden, Tage???	



Backtracking / Springerproblem

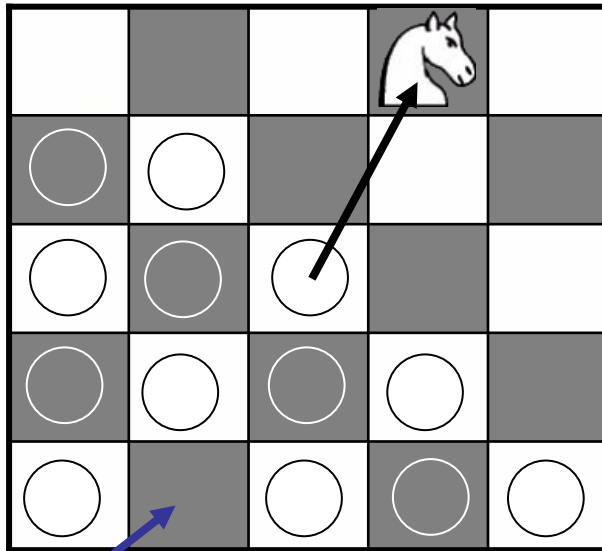
- Lösung zwar korrekt, aber
 - zu viele Teillösungen werden verworfen und
 - Zeitverbrauch steigt explosionsartig mit Grösse von n .
- Optimierung nötig (**Heuristiken**)
 - Sprünge vermeiden, die nie zu einer Lösung führen können
 - Sprünge bevorzugen, die vermutlich eher zu einer Lösung führen
- „Heuristik“, (griech. *heurísko*, ich finde):
Strategie, ein Ziel planmässig zu finden



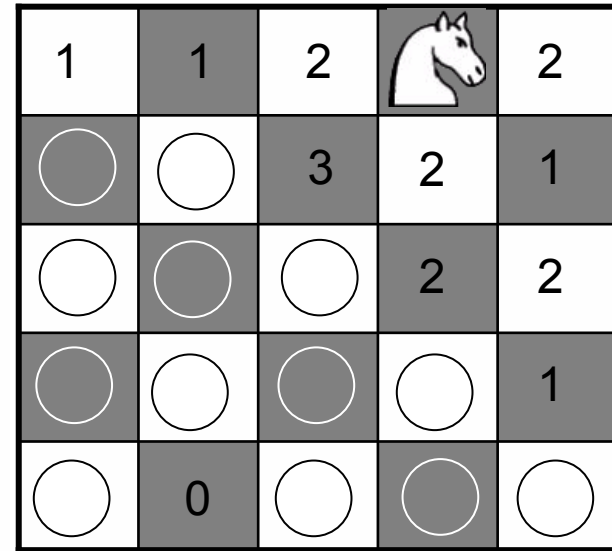
Backtracking / Springerproblem

Heuristik 1:

Vermeiden, dass nicht mehr erreichbare Felder entstehen



Feld nicht mehr vom Springer erreichbar



Führe pro Feld Buch über Anzahl Felder, von denen Springer das Feld erreichen kann

Backtracking / Springerproblem

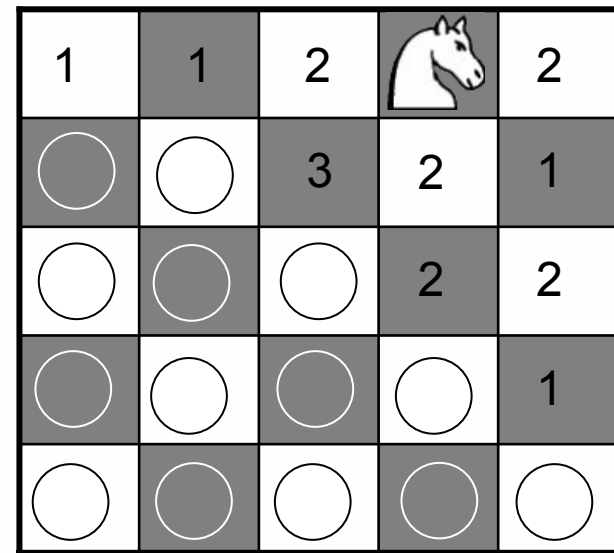
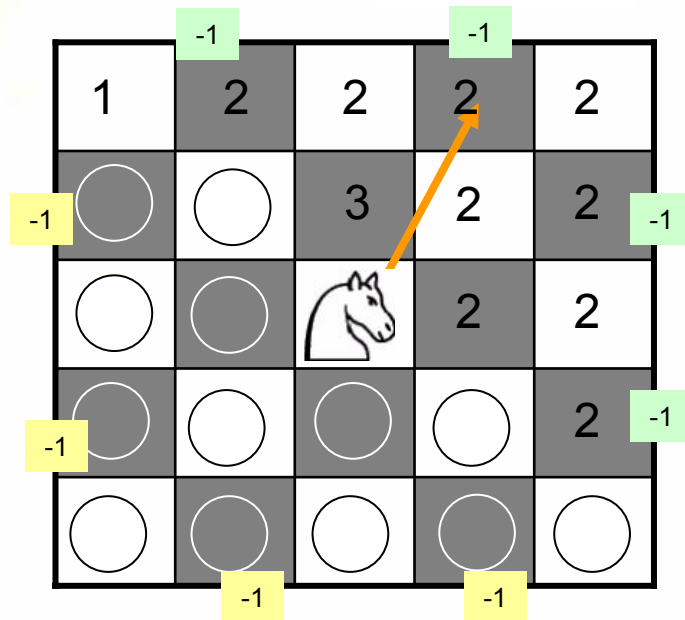
2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

2	3	4	4	3	2
3	4	6	6	4	3
4	6	8	8	6	4
4	6	8	8	6	4
3	4	6	6	4	3
2	3	4	4	3	2

```
int [][] erreichbarkeit= new int[n][n];
```

Initialwerte der Anzahl für erreichbare Felder

Backtracking / Springerproblem



Änderungen nach Springerzug berechnen:

Anzahl aller vom Springer erreichbaren Felder um Eins reduzieren
(beim Verwerfen einer Lösung, wieder rückgängig machen!)

Negative Werte für besuchte Felder, um beim Verwerfen von
Lösungen korrekte Werte zu berechnen.



2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2



2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2



2	3	4	3	2
3	4	6	4	3
4	5	8	6	4
3	4	5	4	3
1	3	4	3	2



1	2	3	3	2
3	3	6	2	3
3	4	8	6	3
3	4	4	3	2
0	3	3	3	1



1	2	3	3	2
3	4	6	3	3
4	4	8	6	4
3	4	4	3	2
1	3	3	3	2



1	3	3	3	2
3	4	6	3	3
4	5	8	6	4
3	4	5	3	3
1	3	3	3	2



1	1	3	3	2
3	3	5	2	3
3	4	8	6	3
3	4	3	3	2
0	2	3	3	1



0	1	3	3	1
3	3	5	2	3
2	4	8	6	2
3	3	3	2	2
0	2	3	3	1

Nicht mehr erreichbar:
 • Rekursion abbrechen,
 • false zurückgeben.

Backtracking / Springerproblem

```

boolean sucheSpringerFolge (int x, int y, int zaehler) {
    for (int i=0; zaehler < n*n && i<offset.length; i++ ) {
        int x1 = x + offset[i][0];
        int y1 = y + offset[i][1];
        // Heuristik 1
        if ( istGueltigerSprung(x1,y1) ) {
            loesung[x1][y1] = zaehler;
            if ( zaehler < n*n ) {
                if ( sucheSpringerFolge (x1,y1, zaehler+1) ) {
                    return true;
                } else {
                    // Berechnung für Heuristik 1 rückgängig machen
                    loesung[x1][y1] = 0;
                }
            } else {
                return true;
            }
        } else {
            // Berechnung für Heuristik 1 rückgängig machen
        }
    }
    return false;
}

```

















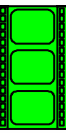
Backtracking / Springerproblem

n	Normal:		Optimiert: Heuristik 1	
	Zeit	Sackgassen	Zeit	Sackgassen
5	<1 sek	263	< 1 sec	99
6	<1 sek	177012	< 1 sec	41966
7	<1 sek	132967	< 1 sec	12 006
8	???		< 5 sec	3 451 066
9	???		???	
10	???		???	

Backtracking / Springerproblem

- Heuristik 2:
 - Die Suche abbrechen, wenn mehr als ein Feld nur noch über genau ein anderes Feld aus erreichbar ist
 - Diese Felder müssen Endfelder der Folge sein (wenn man sie erreicht hat, kann man von ihnen aus nirgendwo mehr hin springen)

1	1	2		2
		4	2	1
1			2	3
				1
				



Springerproblem / Warnsdorff Regel

- Heuristik 3:
- Auf am schlechtesten erreichbare Feld springen
 - Springer findet sonst immer schlechter Weg dorthin
 - auf Feld mit den wenigsten Nachfolgefeldern springen

3	10	21	16	5
20	15	4	11	22
9	2	25	6	17
14	19	8	23	12
1	24	13	18	7

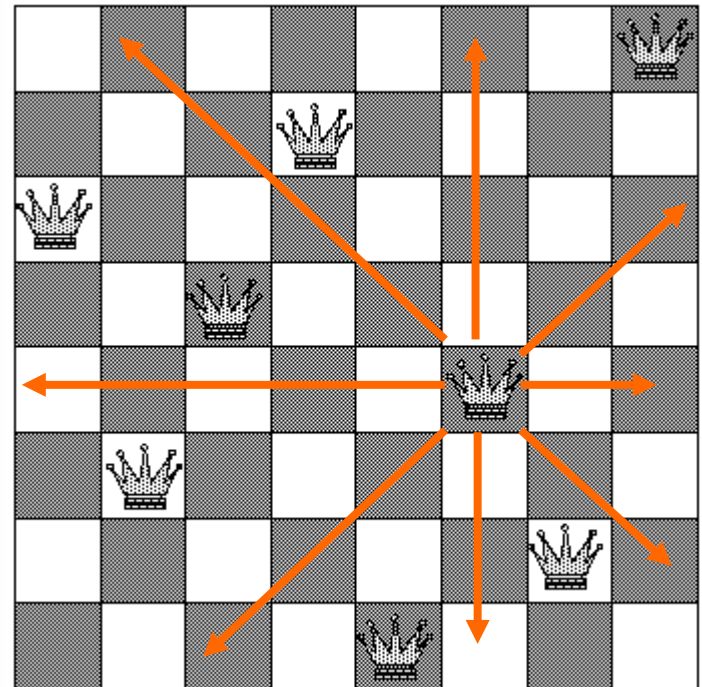


Backtracking / Springerproblem

n	Normal:		Optimiert: Heuristik 3	
	Zeit / Versuche		Zeit	Sackgassen
5	<1 sek	288	< 1 sec	0
6	<1 sek	177 048	< 1 sec	0
7	<1 sek	133 016	< 1 sec	0
8	???		< 1 sec	0
9	???		< 1 sec	0
51	???		< 1 sec	0
52	???		Stack Overflow	

Backtracking / Acht-Damen-Problem

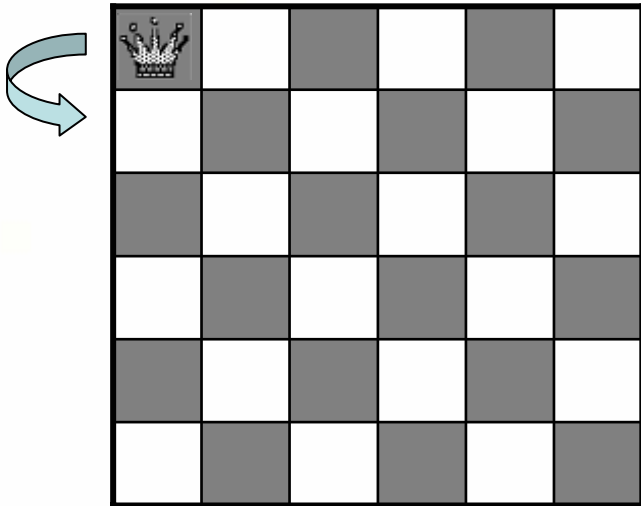
- Wie können acht Damen auf dem Schachbrett positioniert werden, so daß sie sich nach den Schachregeln nicht schlagen können?
- F. Gauß, 1850
- 92 Lösungen
- Lösung mit Backtracking:
 - Nächsten Schritt auswählen
 - Schritt überprüfen
 - Teillösung erweitern
 - Lösung überprüfen
 - Rekursiv weitersuchen
 - Teillösung/Schritt zurücknehmen





Acht-Damen-Problem

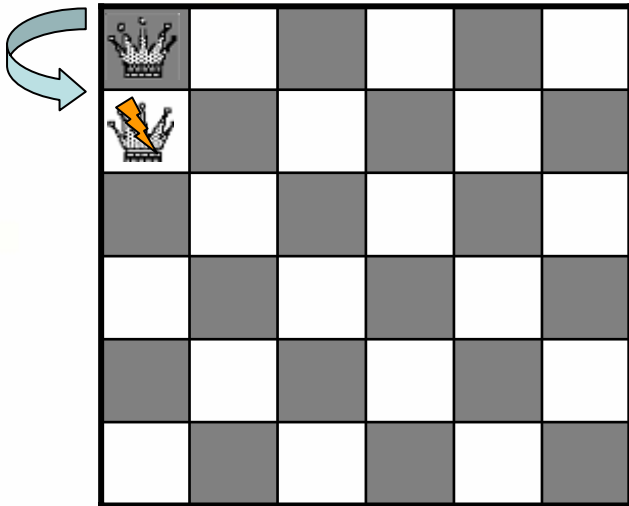
- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren





Acht-Damen-Problem

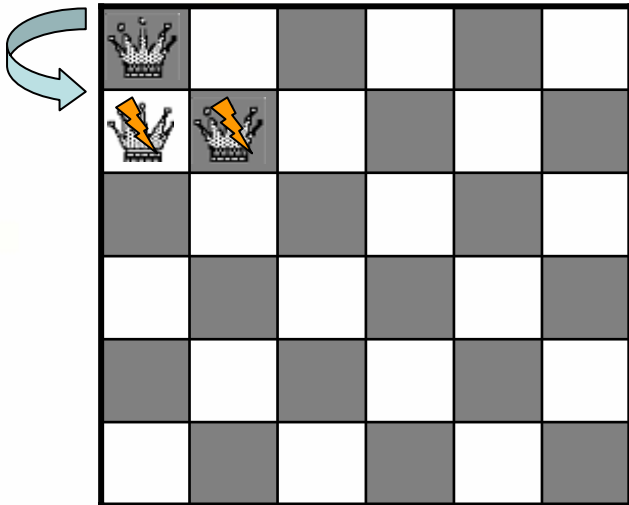
- Nächsten Schritt auswählen
 - Dame in j -ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren





Acht-Damen-Problem

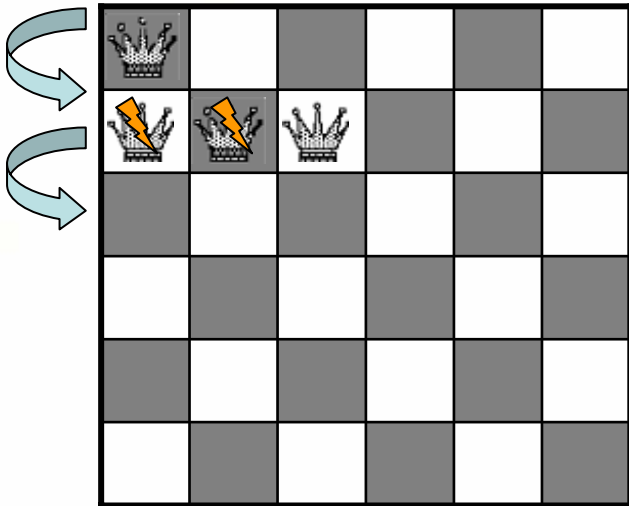
- Nächsten Schritt auswählen
 - Dame in j -ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren





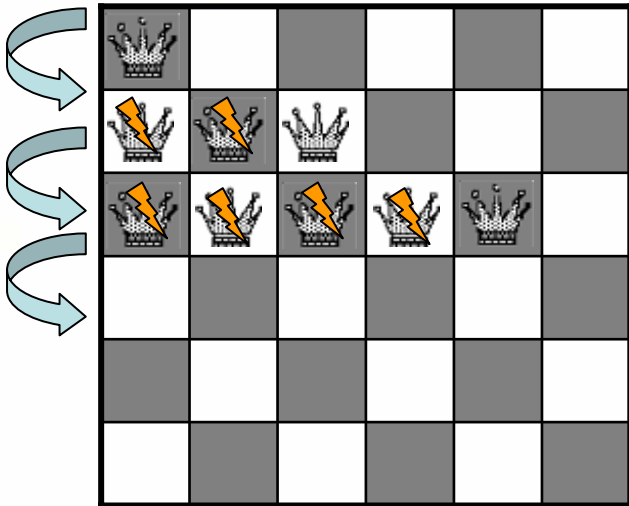
Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren



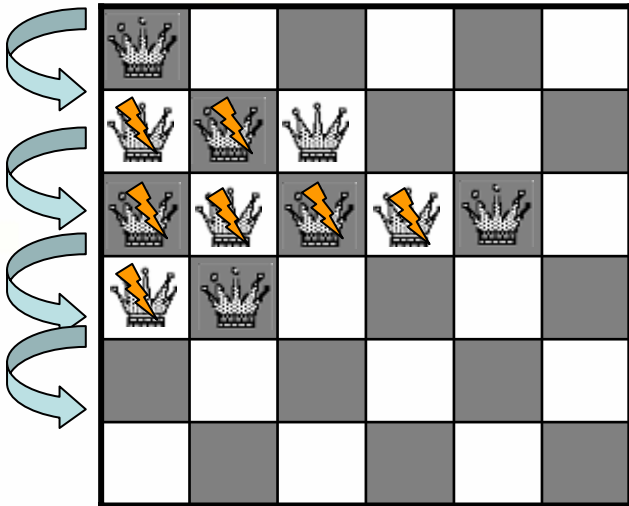
Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren



Acht-Damen-Problem

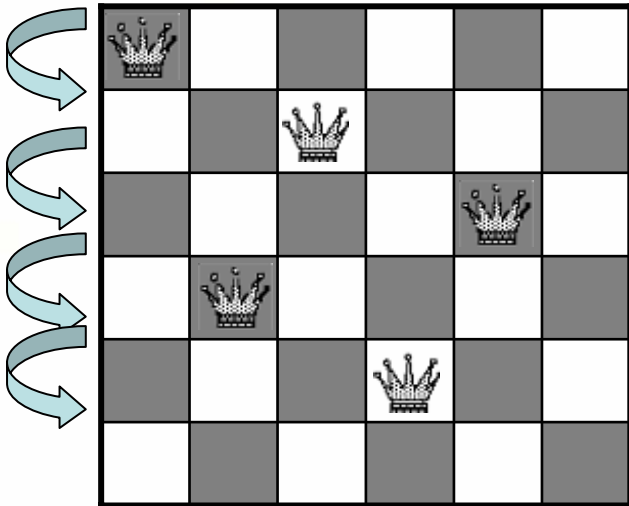
- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren





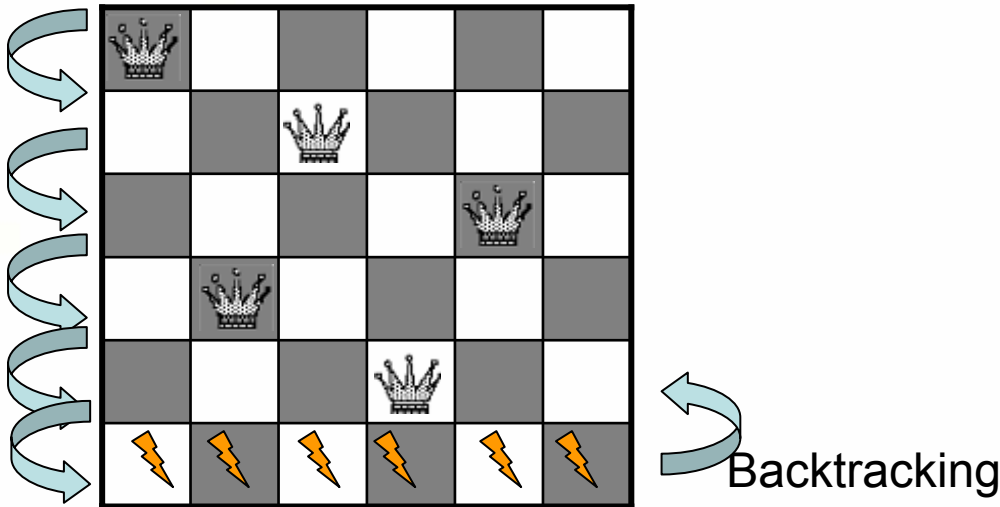
Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren



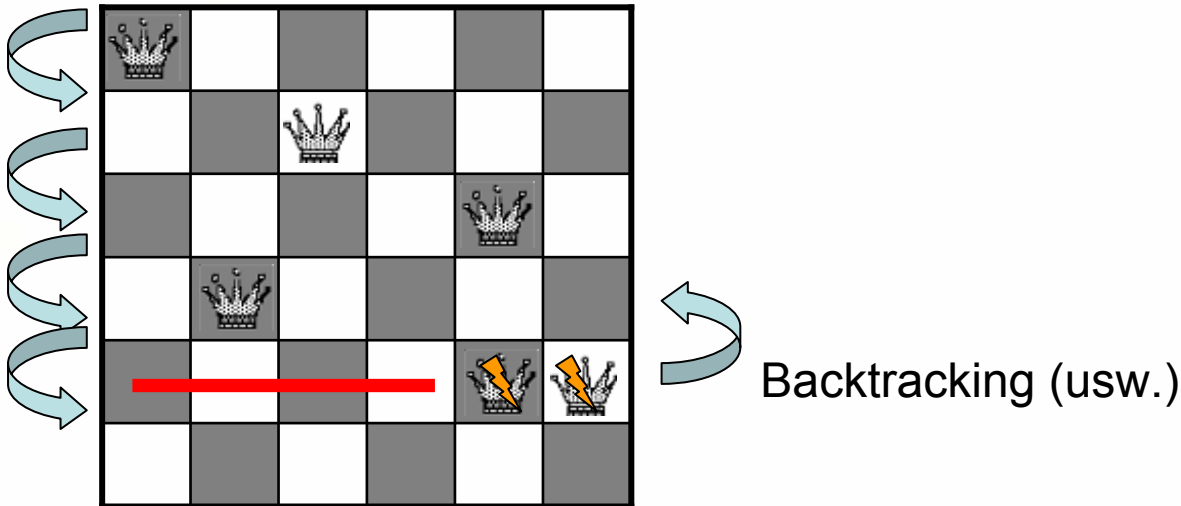
Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren



Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren





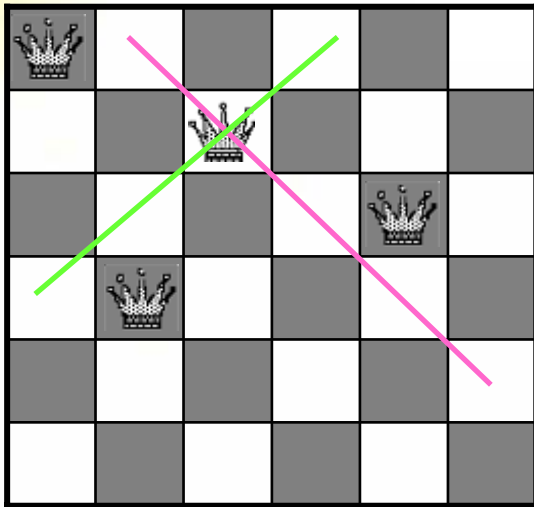
Acht-Damen-Problem

- Nächsten Schritt auswählen
 - Dame in j-ter Zeile alle $i=0, \dots, 7$ Positionen durchlaufen lassen
- Schritt überprüfen
- Rekursiv weitersuchen
 - Nächste Dame in Zeile $j+1$ positionieren

```
public boolean sucheLoesung(int j) {
    for (int i = 0; i < 8; i++) {
        if ( /* schritt ist gültig */ ) {
            // Erweitere Lösung
            if ( j < 8 ) {
                if ( sucheLoesung(j+1) ) {
                    return true;
                } else {
                    // Mache schritt rückgängig
                }
            } else {
                return true;
            }
        }
    }
    return false;
}
```



Acht-Damen-Problem



- Schritt überprüfen
 - Ist keine Dame in Spalte i ?
 - Ist keine Dame in Zeile j ?
 - Ist keine Dame in Diagonalen 1?
 - Ist keine Dame in Diagonalen 2?

```
boolean [] spalte = new boolean[8];
```

$n=8$

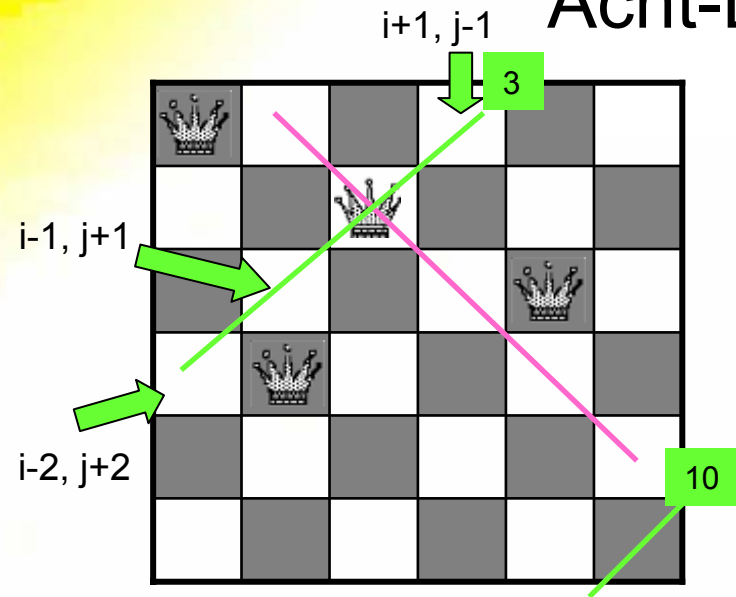
```
boolean [] zeile = new boolean[8];
```

```
spalte[i] == false
```

```
zeile[j] == false
```



Acht-Damen-Problem



Schritt überprüfen

- Ist keine Dame in Spalte i ?
- Ist keine Dame in Zeile j ?
- Ist keine Dame in **Diagonalen 1**?
- Ist keine Dame in **Diagonalen 2**?

```
boolean [] spalte = new boolean[8];    n=8
```

```
boolean [] zeile = new boolean[8];
```

```
boolean [] diagonale1 = new boolean[15];
```

↑
 $2*n-1$

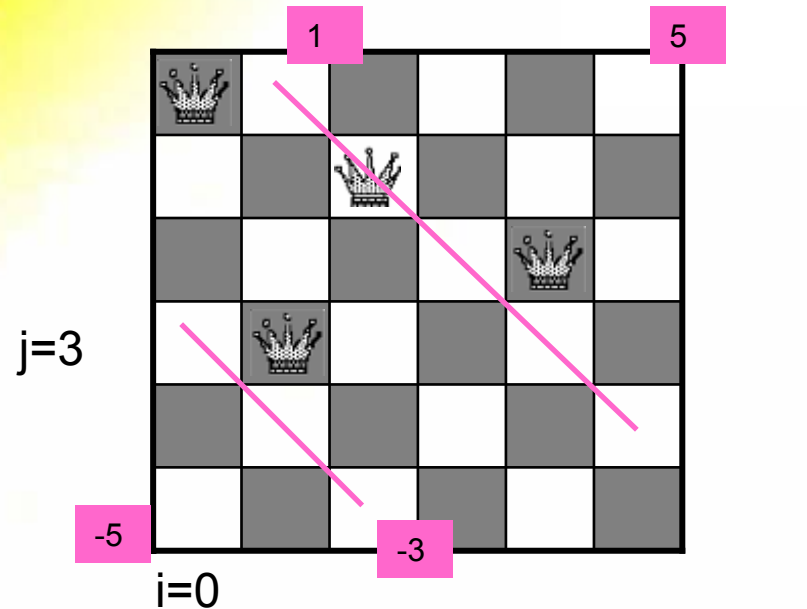
```
spalte[i] == false
```

```
zeile[j] == false
```

```
diagonale1[i+j] == false
```

Summe der Indizes
Bei Diagonale 1 ist
immer gleich
0 bis 14

Acht-Damen-Problem



- Schritt überprüfen
 - Ist keine Dame in Spalte i ?
 - Ist keine Dame in Zeile j ?
 - Ist keine Dame in Diagonalen 1?
 - Ist keine Dame in **Diagonalen 2?**

```
boolean [] spalte = new boolean[8];    n=8
```

```
boolean [] zeile = new boolean[8];
```

```
boolean [] diagonale1 = new boolean[15];
```

```
boolean [] diagonale2 = new boolean[15];
```

Differenz der Indizes
Immer gleich, können
aber negativ werden
-7 bis +7

```
spalte[i] == false
```

```
zeile[j] == false
```

```
diagonale1[i+j] == false
```

```
diagonale2[i-j+7] == false
```

Acht-Damen-Problem

```
public boolean sucheLoesung(int j) {
    for (int i = 0; i < 8; i++) {
        if ( ! (spalte[i] || zeile[i] || diagonale1[i+j] || diagonale2[i-j+8])) {
            spalte[i] = true;
            zeile[j] = true;
            diagonale1[i+j] = true;
            diagonale2[i-j+8] = true;
            if ( j < 8) {
                if ( sucheLoesung(j+1) ) {
                    return true;
                } else {
                    spalte[i] = false;
                    zeile[j] = false;
                    diagonale1[i+j] = false;
                    diagonale2[i-j+7] = false;
                }
            } else {
                return true;
            }
        }
    }
    return false;
}
```

Wo ist aber nun die Lösung zu finden?

Acht-Damen-Problem

```

public boolean sucheLoesung(int j) {
    for (int i = 0; i < 8; i++) {
        if ( ! (spalte[i] != 0 || zeile[j] || diagonale1[i+j]
                || diagonale2[i-j+8])) {
            spalte[i] = j;
            zeile[j] = true;
            diagonale1[i+j] = true;
            diagonale2[i-j+8] = true;
            if ( j < 8) {
                if ( sucheLoesung(j+1) ) {
                    return true;
                } else {
                    spalte[i] = 0;
                    zeile[j] = false;
                    diagonale1[i+j] = false;
                    diagonale2[i-j+7] = false;
                }
            } else {
                return true;
            }
        }
    }
    return false;
}

```

Statt

boolean [] spalte = new boolean[8];
 verwende
 int [] spalte = new int[8];
 und schreibe dort
 die Zeile der Dame in Spalte i hinein



Backtracking

- Je nach Problemart Laufzeit effizient oder sehr ineffizient (exponentiell)
- Effizient
 - Wenn – wie beim Labyrinth – Sackgassen nicht noch einmal durchlaufen werden müssen
- Ineffizient
 - Wenn immer wieder neue Teillösungen berechnet aber verworfen werden: Springerproblem
 - *Richtige* Heuristiken nötig, um Suche zu optimieren