

Praktische Informatik 2

Algorithmische Grundkonzepte

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für
Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen



Algorithmus: Definition

- Beschreibung einer Methode zur Lösung einer gegebenen Aufgabenstellung
 - Begriff sehr allgemein fassbar: auch Kochrezepte und Aufbauanleitungen sind Algorithmen
 - In der Informatik: Algorithmen, die programmierbar sind
- Beispiele
 - Schriftliches Addieren, Subtrahieren, Multiplizieren und Dividieren
 - Aus der Position und einer Felddescription eine Bitmaske für die Erreichbarkeit von Nachbarzellen erstellen

Algorithmus: Definition

- **Eingabespezifikation**: Eingabegrößen, Anforderungen an diese
- **Ausgabespezifikation**: Ausgabegrößen, Eigenschaften dieser
- **Endliche Beschreibung**: Verfahren in endlichem Text beschrieben
- **Effektivität**: Jeder Schritt muss effektiv ausführbar sein
- **Determiniertheit**: Verfahrensablauf zu jedem Zeitpunkt vorgeschrieben
- **partielle Korrektheit**: Jedes berechnete Ergebnis genügt der Ausgabespezifikation, sofern die Eingaben der Spezifikation genügt haben
- **Terminierung**: Halt nach endlich vielen Schritten mit einem Ergebnis, sofern die Eingaben der Spezifikation genügt haben

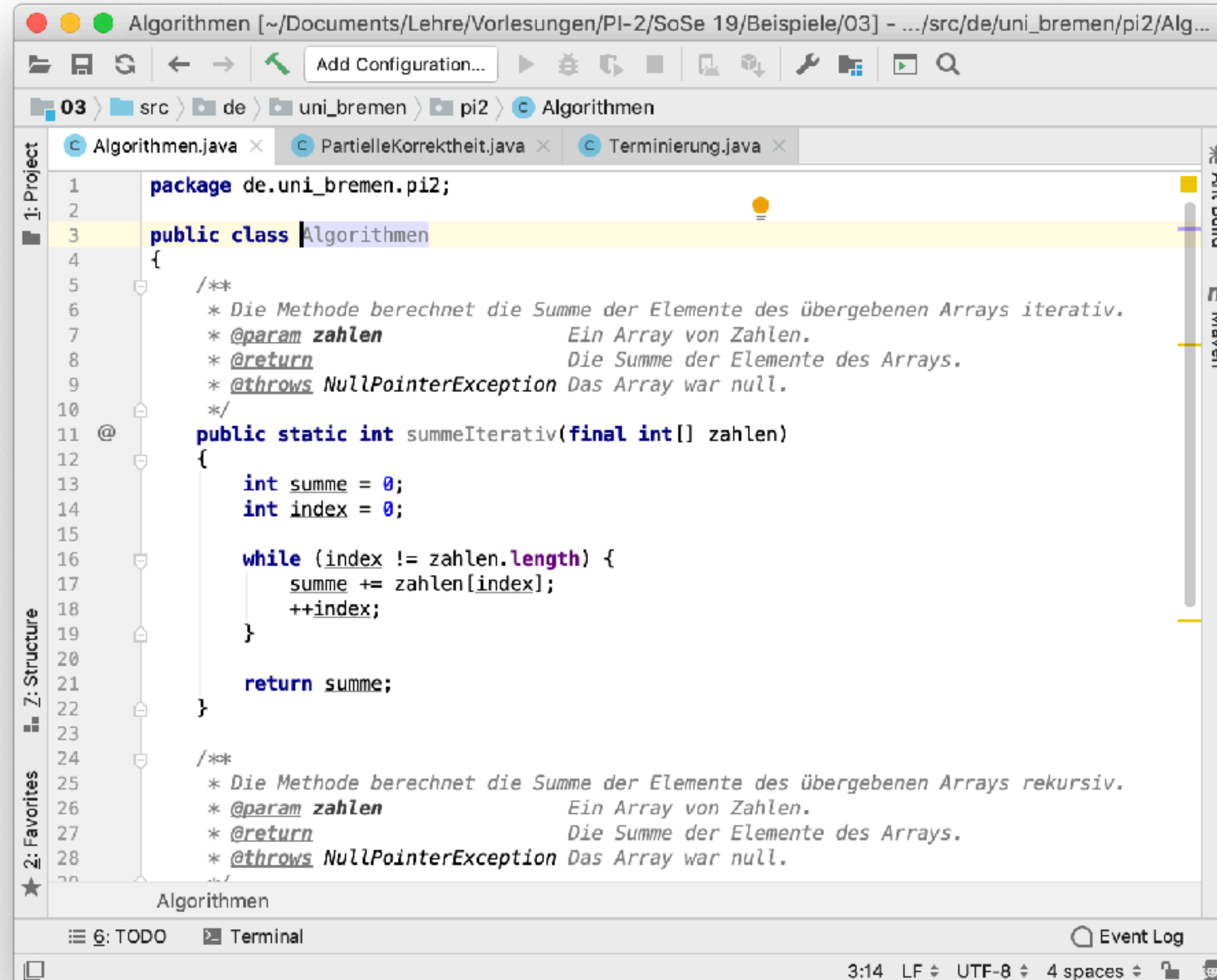
Spezifikation Durchführbarkeit Korrektheit
totale

Grundschema des Algorithmenaufbaus

Name des Algorithmus und Liste der Parameter, Spezifikationen des Ein-/Ausgabeverhaltens

1. **Vorbereitung**: Einführung von Hilfsgrößen etc.
2. **Trivialfall?** Falls einfacher Fall vorliegt, Beendigung mit Ergebnis
3. **Arbeit** (Problemreduktion, Ergebnisaufbau)
 - Reduziere Problemstellung **X** auf eine einfachere Form **X'**, mit **X > X'**
 - Baue entsprechend der Reduktion einen Teil des Ergebnisses auf
4. **Rekursion** bzw. **Iteration**: Rufe zur Weiterverarbeitung Algorithmus mit reduziertem **X'** erneut auf (Rekursion), bzw. fahre mit **X'** bei 2. fort (Iteration)

Grundschemata des Algorithmenaufbaus: Demo



```
1 package de.uni_bremen.pi2;
2
3 public class Algorithmen
4 {
5     /**
6      * Die Methode berechnet die Summe der Elemente des übergebenen Arrays iterativ.
7      * @param zahlen Ein Array von Zahlen.
8      * @return Die Summe der Elemente des Arrays.
9      * @throws NullPointerException Das Array war null.
10     */
11     @ public static int summeIterativ(final int[] zahlen)
12     {
13         int summe = 0;
14         int index = 0;
15
16         while (index != zahlen.length) {
17             summe += zahlen[index];
18             ++index;
19         }
20
21         return summe;
22     }
23
24     /**
25      * Die Methode berechnet die Summe der Elemente des übergebenen Arrays rekursiv.
26      * @param zahlen Ein Array von Zahlen.
27      * @return Die Summe der Elemente des Arrays.
28      * @throws NullPointerException Das Array war null.
29     */
30 }
```

Algorithmenmuster

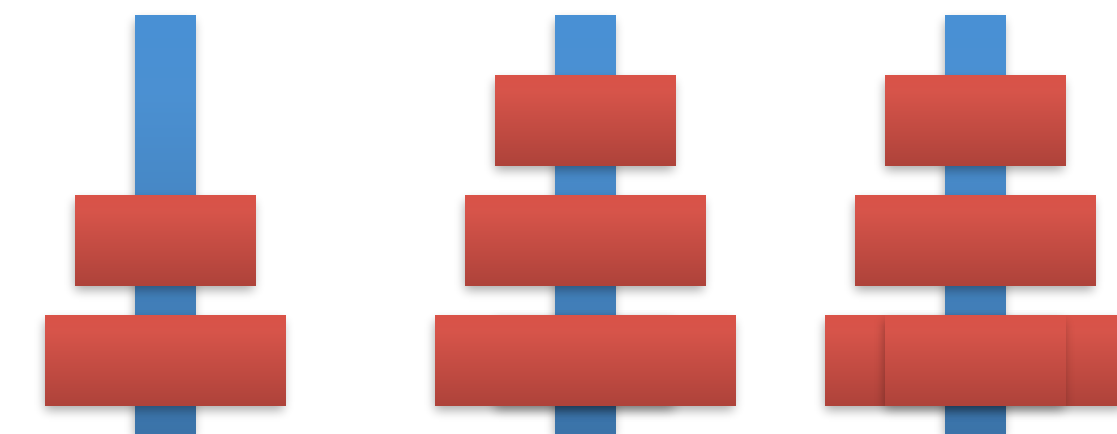
- **Greedy** (gierig): Reduziere das Problem in einer Richtung
- **Divide and Conquer** (Teile und herrsche): Reduziere das Problem in zwei oder mehrere Teilprobleme
- **Backtracking**: Generiere mögliche Lösungen, wenn es nicht weiter geht, mache beim Vorgänger weiter
- **Dynamische Programmierung**: Werden Teillösungen mehrfach benötigt, merke sie dir in einer Tabelle und lies diese aus, wenn sie wieder benötigt werden

Greedy (gierig)

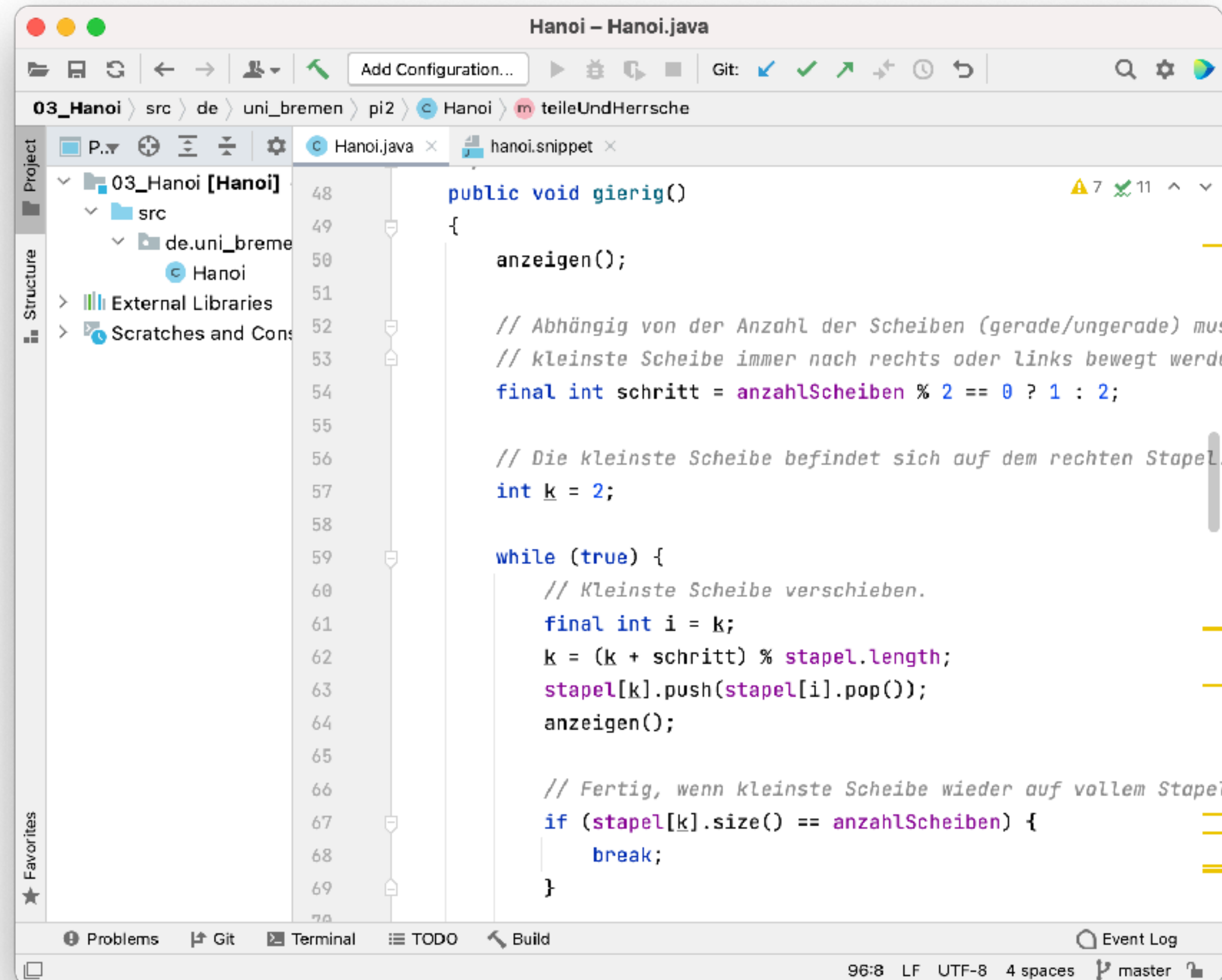
- Handle einfache und triviale Fälle
- Reduziere das Problem in **einer** Richtung
- Rekursiver Aufruf oder Iteration
- **Beispiel:** Wechselgeld mit minimaler Anzahl von Münzen, wobei genug Münzen verfügbar sind
 - Lösung: So lange die größte Münze zurückgeben, deren Wert kleiner oder gleich der noch offenen Restsumme ist, bis die Restsumme 0 ist

Greedy: Türme von Hanoi

- Bewege Turm von einer Stange zu einer anderen, ohne dass jemals eine größere Scheibe auf einer kleineren liegt
- Lösung
 - Bewege kleinste Scheibe nach links (zyklisch)
 - Fertig, falls Turm vollständig bewegt
 - Ansonsten bewege einzig mögliche nicht-kleinste Scheibe
 - Beginne von vorne



Greedy: Türme von Hanoi

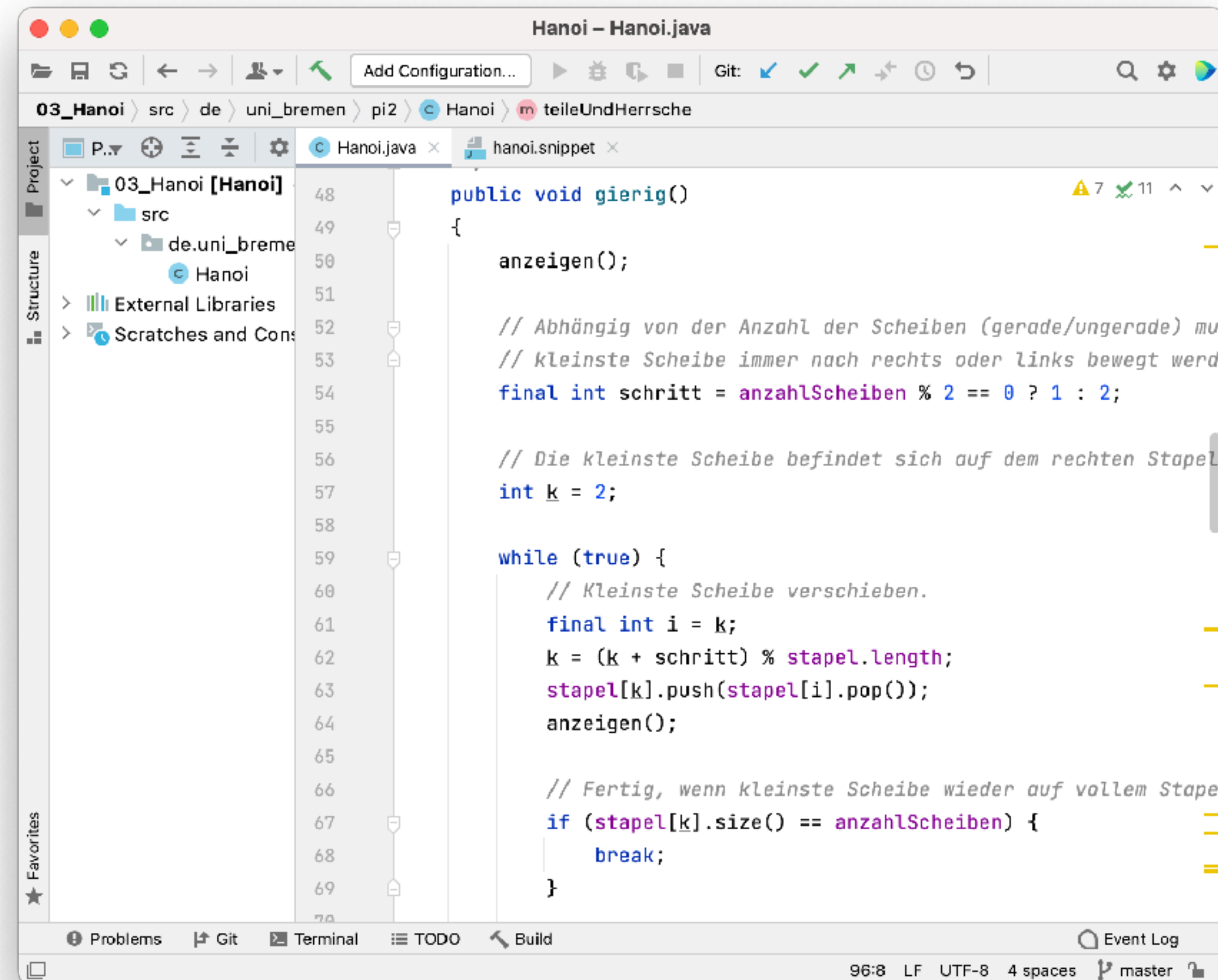


```
Hanoi – Hanoi.java
Add Configuration...
03_Hanoi > src > de > uni_bremen > pi2 > Hanoi > m teileUndHerrsche
Hanoi.java x hanoi.snippet x
Project
  03_Hanoi [Hanoi]
    src
      de.uni_breme
        Hanoi
      External Libraries
      Scratches and Cons
Structure
Favorites
48 public void gierig()
49 {
50     anzeigen();
51
52     // Abhängig von der Anzahl der Scheiben (gerade/ungerade) mus
53     // kleinste Scheibe immer nach rechts oder links bewegt werde
54     final int schritt = anzahlScheiben % 2 == 0 ? 1 : 2;
55
56     // Die kleinste Scheibe befindet sich auf dem rechten Stapel
57     int k = 2;
58
59     while (true) {
60         // Kleinste Scheibe verschieben.
61         final int i = k;
62         k = (k + schritt) % stapel.length;
63         stapel[k].push(stapel[i].pop());
64         anzeigen();
65
66         // Fertig, wenn kleinste Scheibe wieder auf vollem Stapel
67         if (stapel[k].size() == anzahlScheiben) {
68             break;
69         }
70     }
Problems Git Terminal TODO Build Event Log
96:8 LF UTF-8 4 spaces master
```

Divide and Conquer (Teile und herrsche)

- Handle einfache und triviale Fälle
- **Teile**: Reduziere das Problem in zwei oder mehrere Teilprobleme
- **Herrsche**: Löse die Teilprobleme (typischerweise rekursiv)
- **Kombiniere**: Setze Teillösungen zur Gesamtlösung zusammen
- Teilprobleme sollten **von derselben Art** wie das Ursprungsproblem sein, damit sie mit **demselben Algorithmus** gelöst werden können

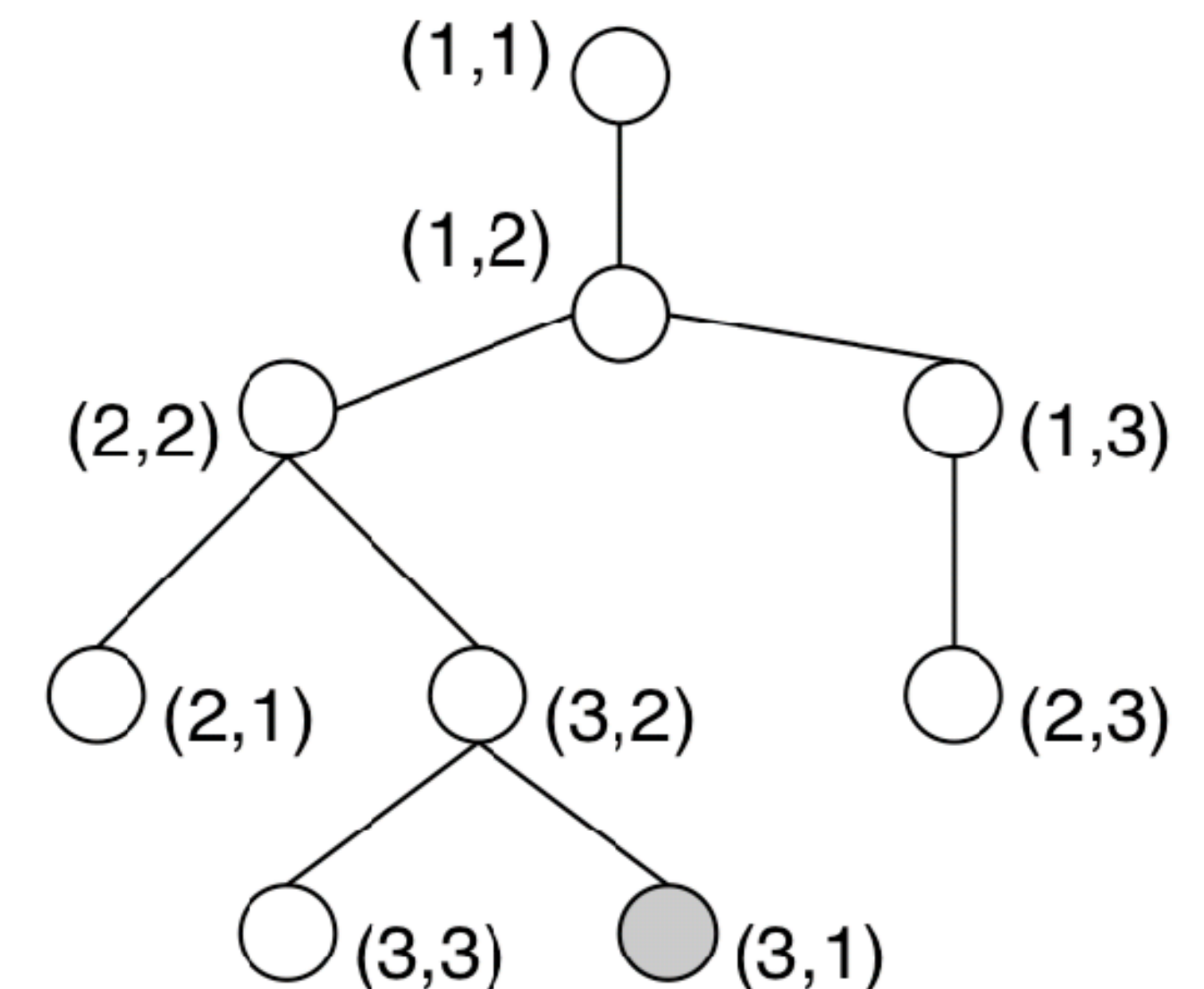
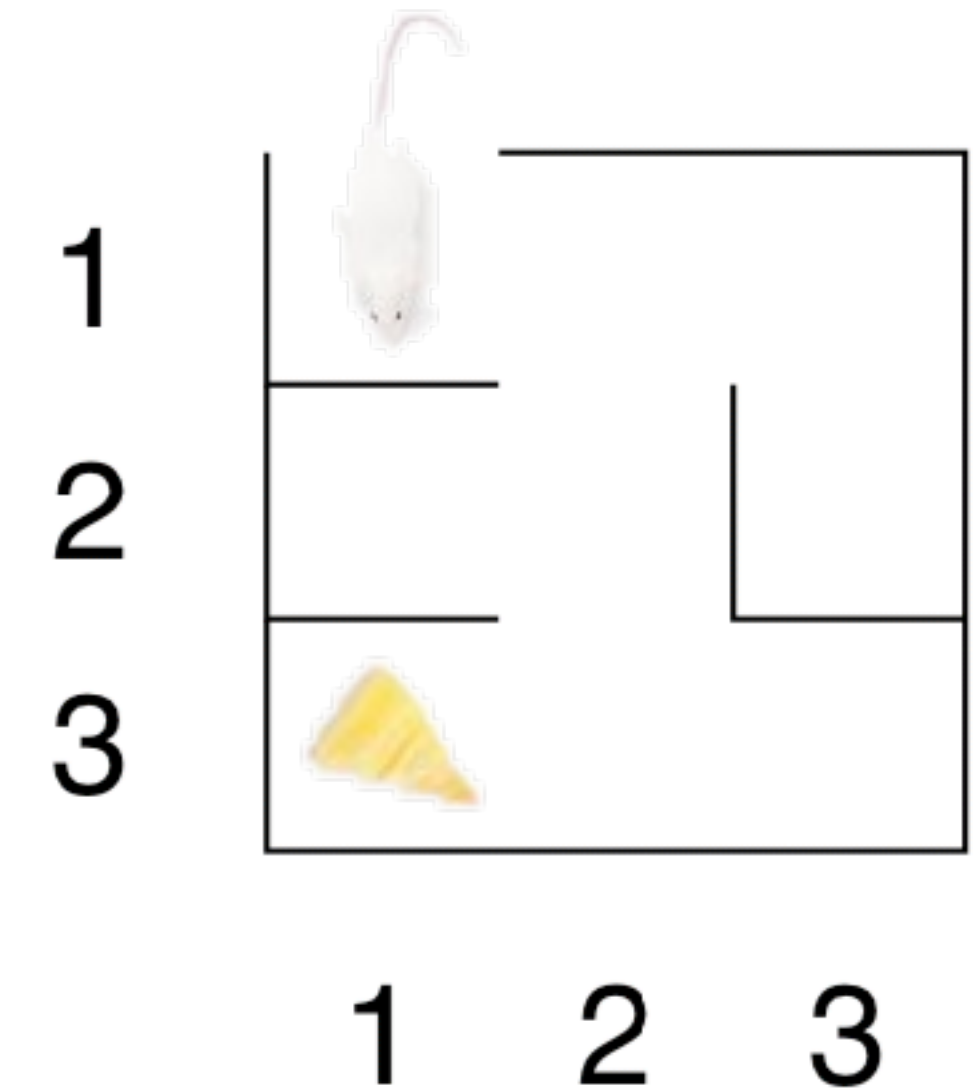
Divide and Conquer: Türme von Hanoi



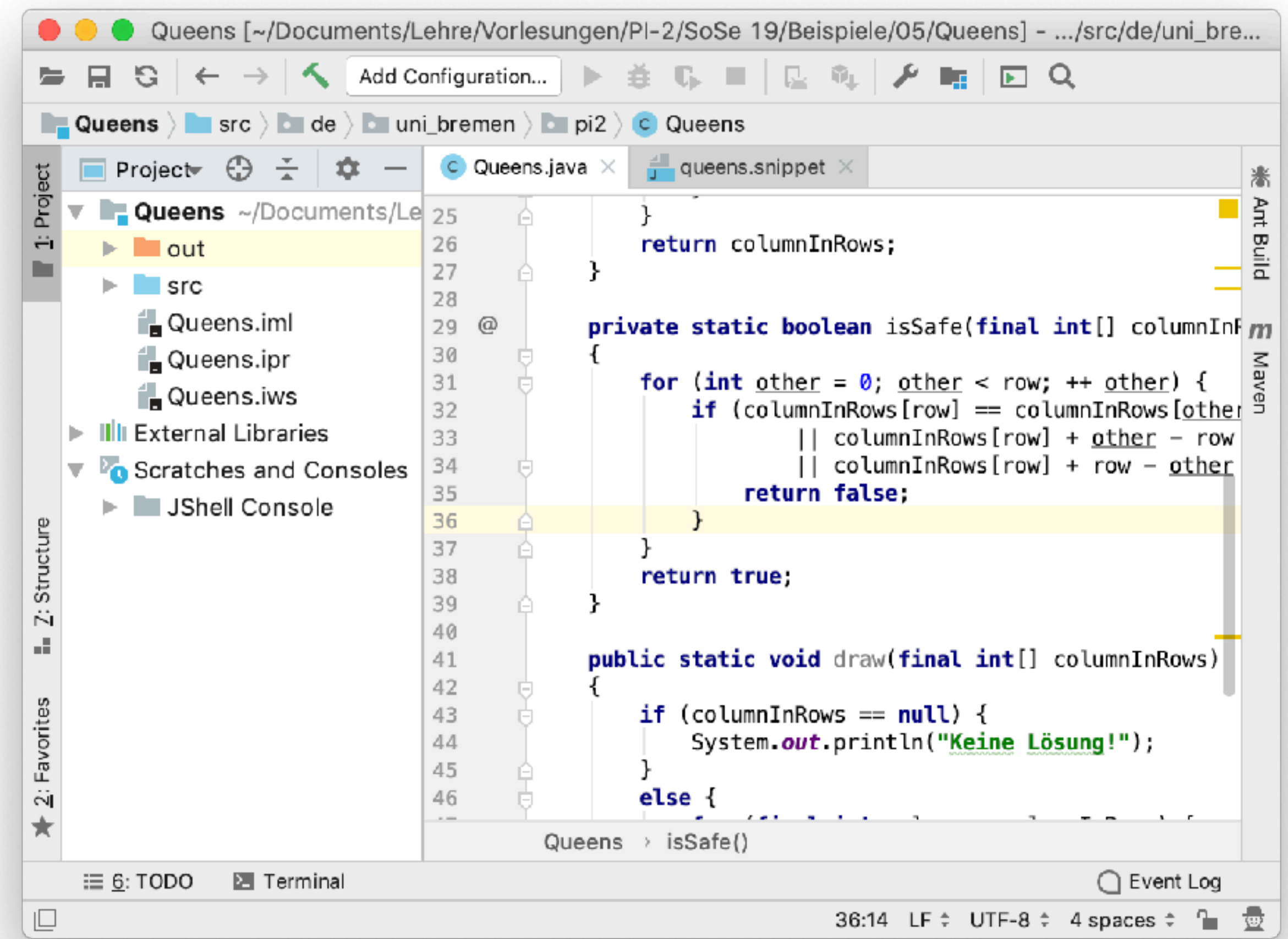
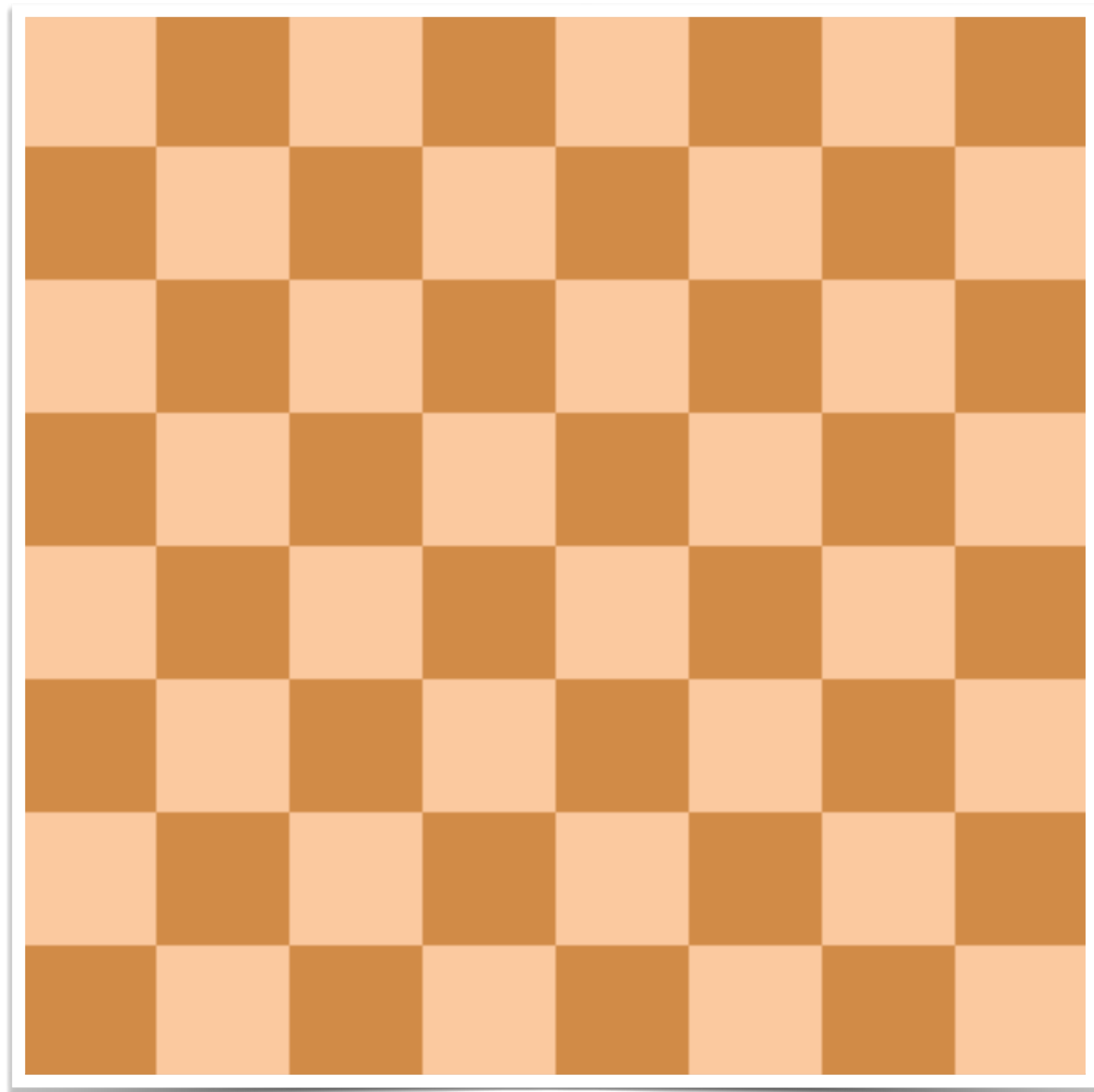
```
48 public void gierig()
49 {
50     anzeigen();
51
52     // Abhängig von der Anzahl der Scheiben (gerade/ungerade) muss
53     // kleinste Scheibe immer nach rechts oder links bewegt werden
54     final int schritt = anzahlScheiben % 2 == 0 ? 1 : 2;
55
56     // Die kleinste Scheibe befindet sich auf dem rechten Stapel
57     int k = 2;
58
59     while (true) {
60         // Kleinste Scheibe verschieben.
61         final int i = k;
62         k = (k + schritt) % stapel.length;
63         stapel[k].push(stapel[i].pop());
64         anzeigen();
65
66         // Fertig, wenn kleinste Scheibe wieder auf vollem Stapel
67         if (stapel[k].size() == anzahlScheiben) {
68             break;
69         }
70     }
```


Backtracking

- Es gibt eine Menge von Konfigurationen K
- K_0 ist die Anfangskonfiguration
- Für jede Konfiguration K_i kann Menge der direkten Erweiterungen $K_{i, 1 \dots n}$ bestimmt werden
- Es ist entscheidbar, ob eine Konfiguration Lösung ist
- Von jeder Konfiguration aus werden alle Erweiterungen rekursiv durchprobiert
- Hat eine Konfiguration keine Erweiterungen, wird zum Vorgänger zurückgekehrt (**backtrack**)



Backtracking: n-Damen-Problem



```
25 }
26 return columnInRows;
27 }
28
29 @
30
31 private static boolean isSafe(final int[] columnInRows, final int row, final int column) {
32     for (int other = 0; other < row; ++ other) {
33         if (columnInRows[other] == column ||
34             columnInRows[other] + other - row == column ||
35             columnInRows[other] - other + row == column) {
36             return false;
37         }
38     }
39     return true;
40 }
41
42 public static void draw(final int[] columnInRows) {
43     if (columnInRows == null) {
44         System.out.println("Keine Lösung!");
45     }
46     else {
47         // ...
48     }
49 }
```

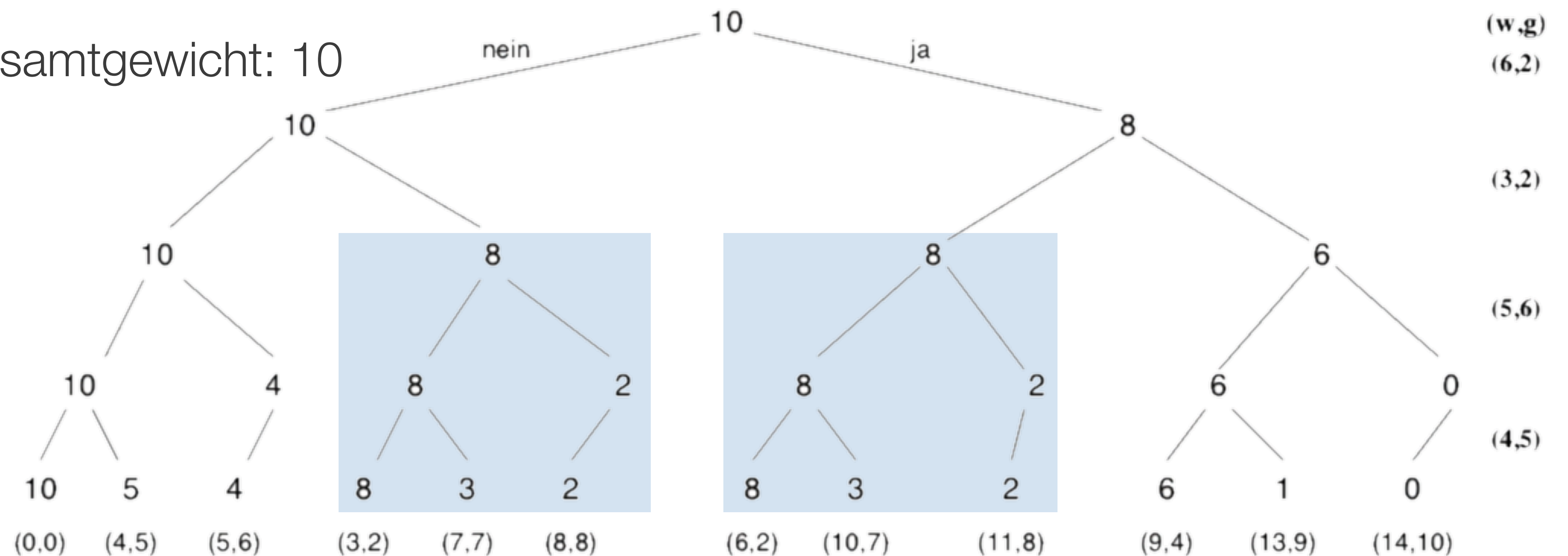
Dynamische Programmierung

- Werden Teillösungen mehrfach benötigt, speichere sie in einer **Tabelle**
- Lies die Tabelle später aus, wenn die Teillösungen wieder benötigt werden
- **Beispiel:** Rucksackproblem
 - Fülle Rucksack so mit Gegenständen, dass das zulässige Gesamtgewicht nicht überschritten wird und ihr Wert maximal ist

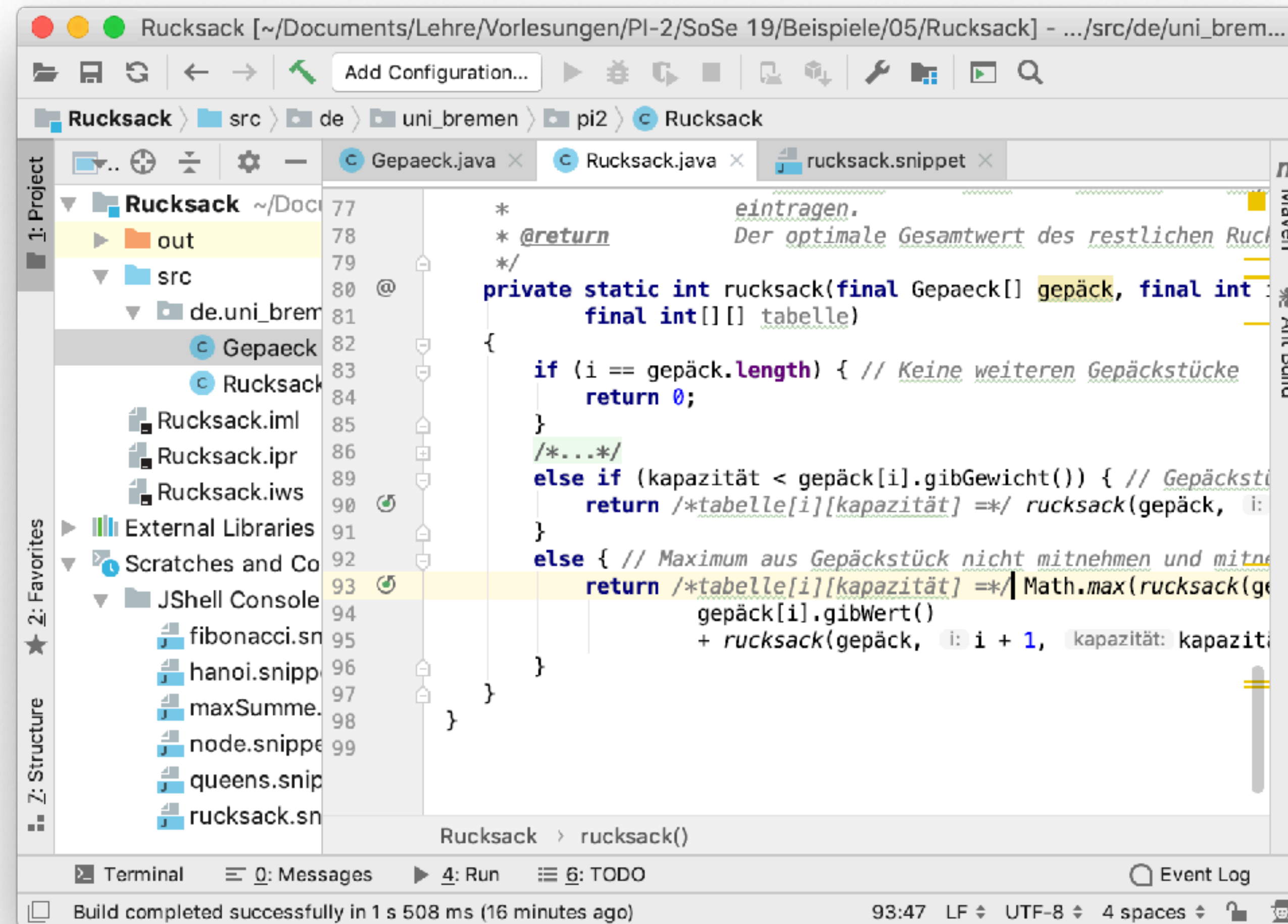


Dynamische Programmierung: Rucksackproblem – Beispiel

- Gewichte $g_1 = 2, g_2 = 2, g_3 = 6, g_4 = 5$
- Werte $w_1 = 6, w_2 = 3, w_3 = 5, w_4 = 4$
- Maximales Gesamtgewicht: 10



Dynamische Programmierung: Rucksackproblem



The screenshot shows an IDE window titled "Rucksack [~/Documents/Lehre/Vorlesungen/PI-2/SoSe 19/Beispiele/05/Rucksack] - .../src/de/uni_brem...". The left sidebar displays the project structure for "Rucksack", including folders "out" and "src", and files "Gepaeck.java", "Rucksack.java", "Rucksack.iml", "Rucksack.ipr", "Rucksack.iws", "External Libraries", "Scratches and Console", and "JShell Console". The main editor window shows the "Rucksack.java" file with the following code:

```
77      *      eintragen.  
78      * @return      Der optimale Gesamtwert des restlichen Ruck  
79      */  
80      @  
81      private static int rucksack(final Gepaeck[] gepäck, final int  
82      final int[][] tabelle)  
83      {  
84          if (i == gepäck.length) { // Keine weiteren Gepäckstücke  
85              return 0;  
86          }  
87          /*...*/  
88          else if (kapazität < gepäck[i].gibGewicht()) { // Gepäckst  
89              return /*tabelle[i][kapazität] ==*/ rucksack(gepäck, i:  
90          }  
91          else { // Maximum aus Gepäckstück nicht mitnehmen und mitne  
92              return /*tabelle[i][kapazität] ==*/ Math.max(rucksack(ge  
93              gepäck[i].gibWert()  
94              + rucksack(gepäck, i: i + 1, kapazität: kapazität  
95          }  
96      }  
97  }  
98  }  
99  }
```

The bottom status bar indicates "Build completed successfully in 1 s 508 ms (16 minutes ago)" and "93:47 LF UTF-8 4 spaces".

Zusammenfassung der Konzepte

- **Algorithmus**
- **Greedy**
- **Divide and Conquer**
- **Backtracking**
- **Dynamische Programmierung**