

# Praktische Informatik 2

## Algorithmische Grundkonzepte

Thomas Röfer

Cyber-Physical Systems  
Deutsches Forschungszentrum für  
Künstliche Intelligenz

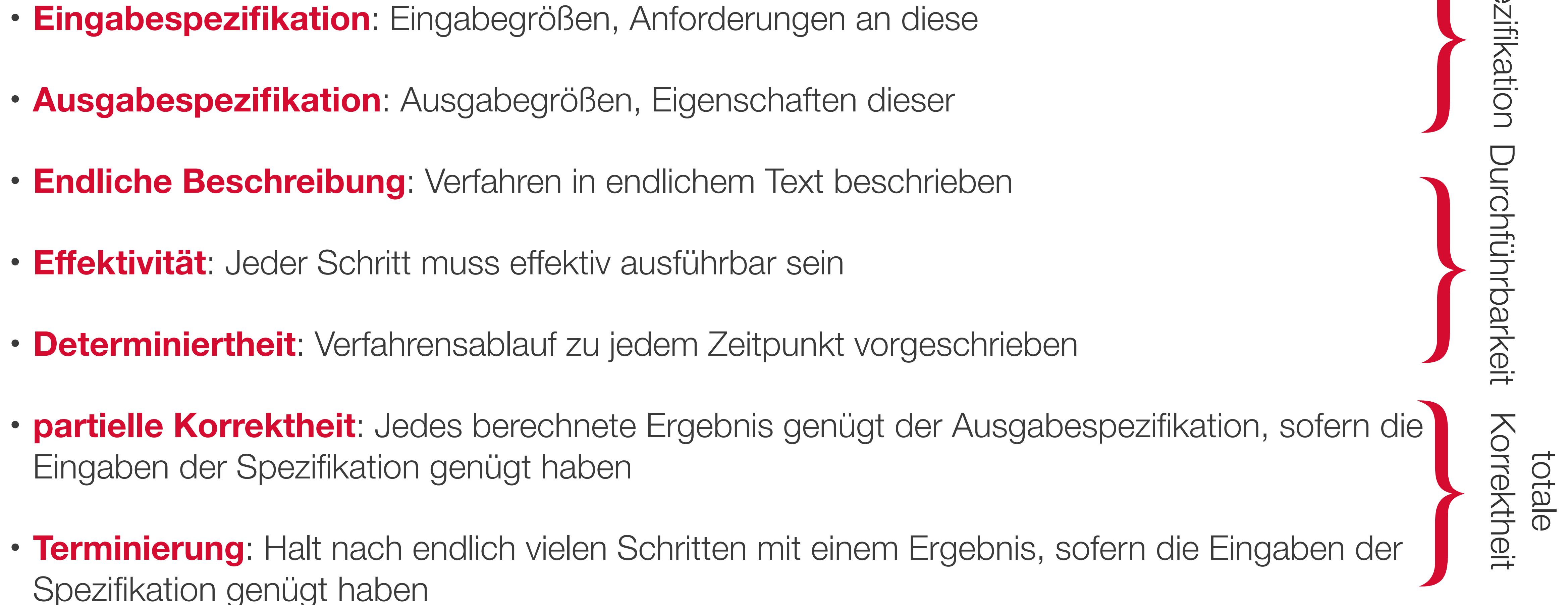
Multisensorische Interaktive Systeme  
Fachbereich 3, Universität Bremen



# 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 Feldbeschreibung eine Bitmaske für die Erreichbarkeit von Nachbarzellen erstellen

# Algorithmus: Definition

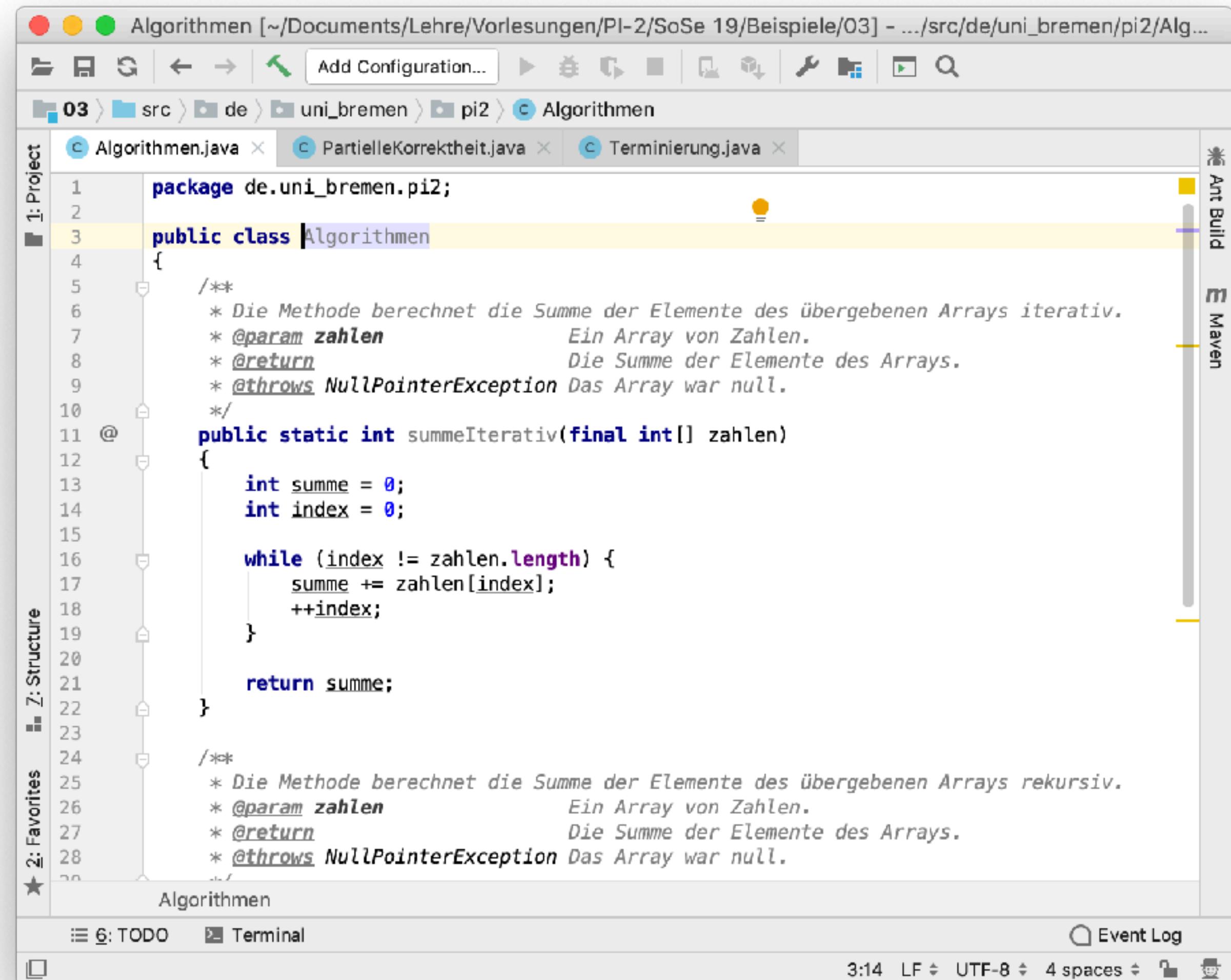
- **Eingabespezifikation**: Eingabegrößen, Anforderungen an diese
  - **Ausgabespezifikation**: Ausgabegrößen, Eigenschaften dieser
  - **Endliche Beschreibung**: Verfahren in endlichem Text beschrieben
  - **Effektivität**: Jeder Schritt muss effektiv ausführbar sein
  - **Determiniertheit**: Verfahrensablauf 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
- 
- The diagram illustrates the 7 properties of an algorithm. The first four properties (Eingabespezifikation, Ausgabespezifikation, Endliche Beschreibung, and Effektivität) are grouped under the heading 'Spezifikation'. The next three properties (Determiniertheit, partielle Korrektheit, and Terminierung) are grouped under the heading 'Durchführbarkeit'. The property 'partielle Korrektheit' is bracketed separately under the heading 'Korrekttheit'.

# 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)

# Grundschema des Algorithmenaufbaus: Demo



The screenshot shows a Java code editor with the following code:

```
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     @
12     public static int summeIterativ(final int[] zahlen)
13     {
14         int summe = 0;
15         int index = 0;
16
17         while (index != zahlen.length) {
18             summe += zahlen[index];
19             ++index;
20         }
21
22         return summe;
23     }
24
25     /**
26      * Die Methode berechnet die Summe der Elemente des übergebenen Arrays rekursiv.
27      * @param zahlen Ein Array von Zahlen.
28      * @return Die Summe der Elemente des Arrays.
29      * @throws NullPointerException Das Array war null.
30  }
```

The code defines a class `Algorithmen` with two static methods: `summeIterativ` and `summeRekursiv`. The `summeIterativ` method calculates the sum of an array iteratively, while the `summeRekursiv` method calculates it recursively. Both methods take an array of integers as a parameter and return the sum. The `summeIterativ` method uses a `while` loop to iterate through the array, adding each element to a `summe` variable and incrementing the `index`. The `summeRekursiv` method is defined but not yet implemented.

## Algorithmenmuster

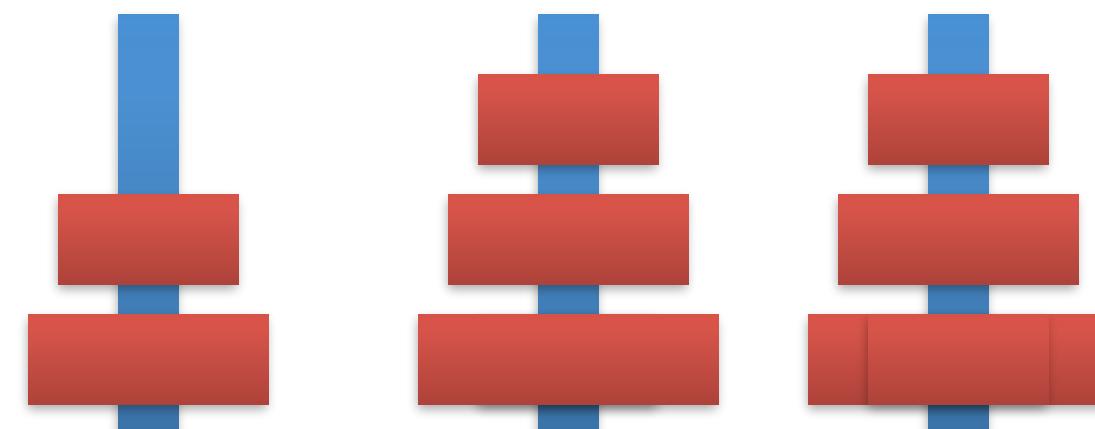
- **Greedy** (gierig): Reduziere das Problem in einer Richtung
- **Divide and Conquer** (Teile und herrsche): Reduziere das Problem in zwei oder mehrere Teilprobleme
- **Backtracking**: Generiere mögliche Lösungen, wenn es nicht weiter geht, mache beim Vorgänger weiter
- **Dynamische Programmierung**: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

## Greedy (gierig)

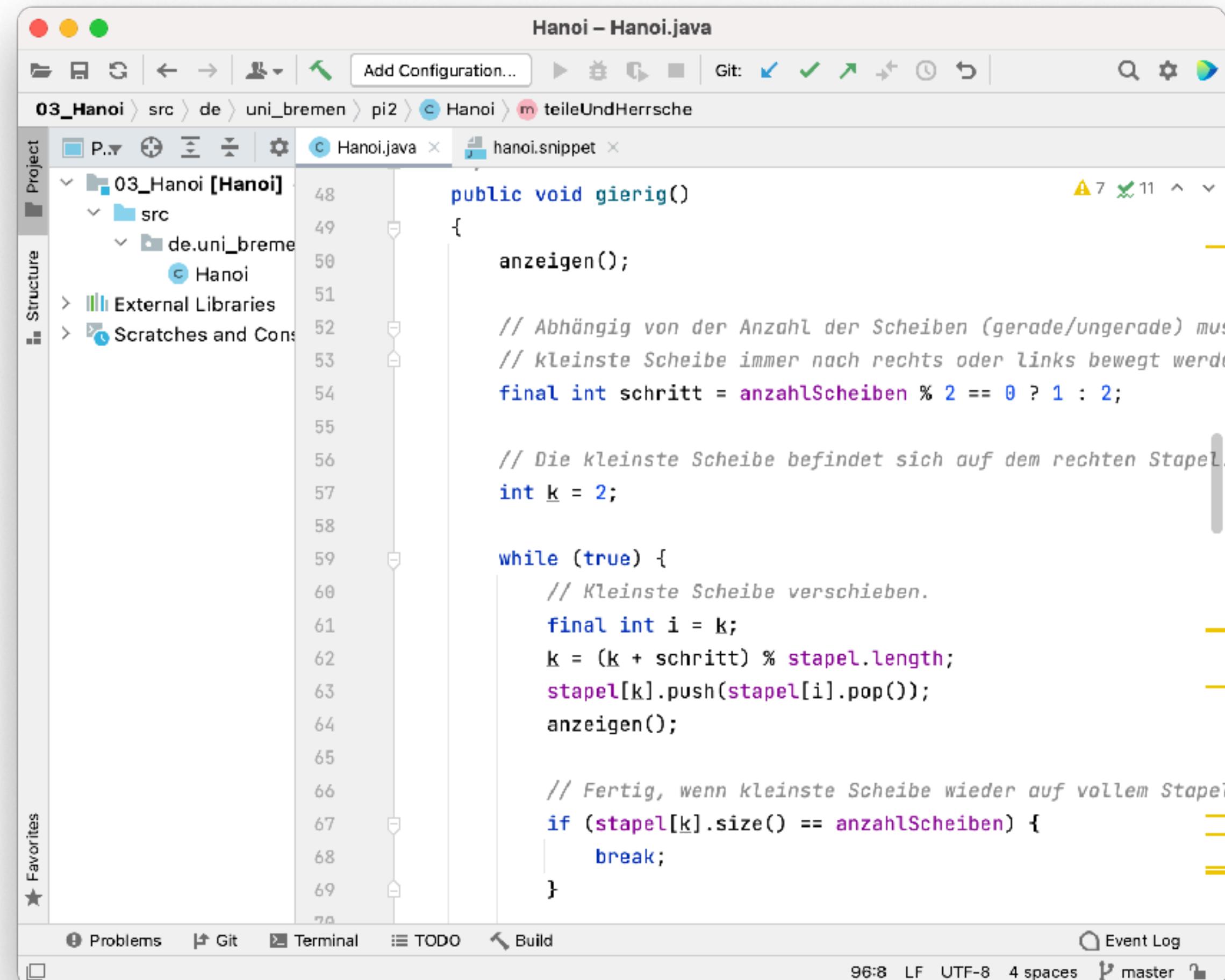
- Behandle einfache und triviale Fälle
- Reduziere das Problem in **einer** Richtung
- Rekursiver Aufruf oder Iteration
- **Beispiel:** Wechselgeld mit minimaler Anzahl von Münzen, wobei genug Münzen verfügbar sind
  - Lösung: So lange die größte Münze zurückgeben, deren Wert kleiner oder gleich der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



The screenshot shows a Java code editor within an IDE. The project structure on the left shows a package named `03_Hanoi` containing a `src` folder, which in turn contains a `de.uni_bremen` folder and a `Hanoi` class. The code itself is a Java file named `Hanoi.java`. The code implements a greedy algorithm for the Tower of Hanoi problem, specifically for the case where the number of disks is even. It uses a stack `stapel` to represent the towers and a variable `k` to track the current disk being moved. The code is annotated with German comments explaining the logic.

```
public void gierig()
{
    anzeigen();

    // Abhängig von der Anzahl der Scheiben (gerade/ungerade) muss
    // kleinste Scheibe immer nach rechts oder links bewegt werden
    final int schritt = anzahlScheiben % 2 == 0 ? 1 : 2;

    // Die kleinste Scheibe befindet sich auf dem rechten Stapel.
    int k = 2;

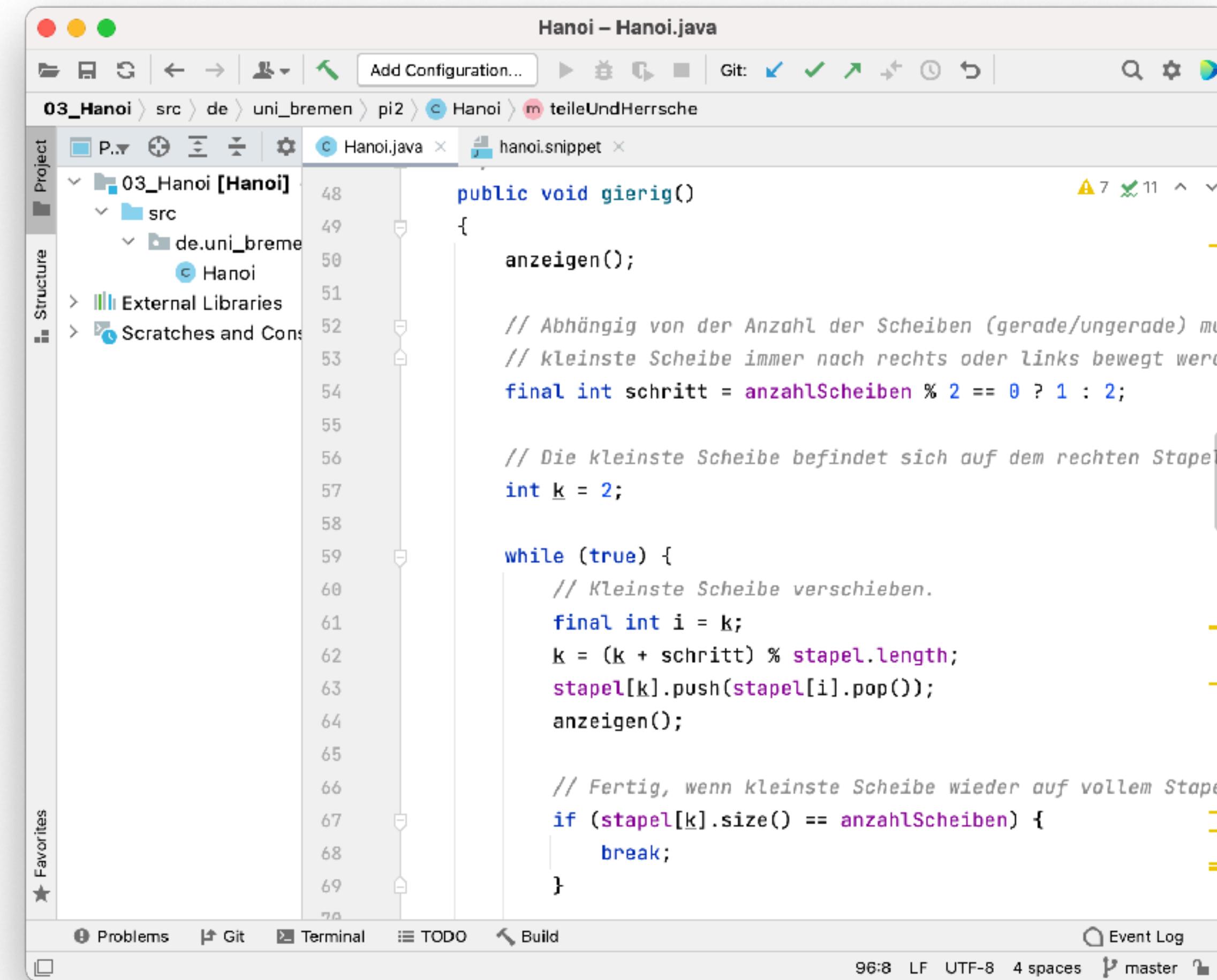
    while (true) {
        // Kleinste Scheibe verschieben.
        final int i = k;
        k = (k + schritt) % stapel.length;
        stapel[k].push(stapel[i].pop());
        anzeigen();

        // Fertig, wenn kleinste Scheibe wieder auf vollem Stapel
        if (stapel[k].size() == anzahlScheiben) {
            break;
        }
    }
}
```

## Divide and Conquer (Teile und herrsche)

- Behandle 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



The screenshot shows an IDE interface with the following details:

- Title Bar:** Hanoi – Hanoi.java
- Toolbar:** Includes standard file operations (New, Open, Save, Print, Find, Replace, Git), a search icon, and a settings icon.
- Project Bar:** Shows the current project structure: 03\_Hanoi > src > de > uni\_bremen > pi2 > Hanoi > teileUndHerrsche. The Hanoi.java file is the active file.
- Code Editor:** Displays the Java code for the Hanoi problem. The code uses a divide and conquer approach to solve the Tower of Hanoi problem.
- Toolbars:** Includes Project, Structure, and Favorites.
- Bottom Bar:** Shows tabs for Problems, Git, Terminal, TODO, Build, and Event Log. The Event Log tab is selected, showing the message "96:8 LF UTF-8 4 spaces master".

```
public void gierig() {
    anzeigen();

    // Abhängig von der Anzahl der Scheiben (gerade/ungerade) muss
    // kleinste Scheibe immer nach rechts oder links bewegt werden
    final int schritt = anzahlScheiben % 2 == 0 ? 1 : 2;

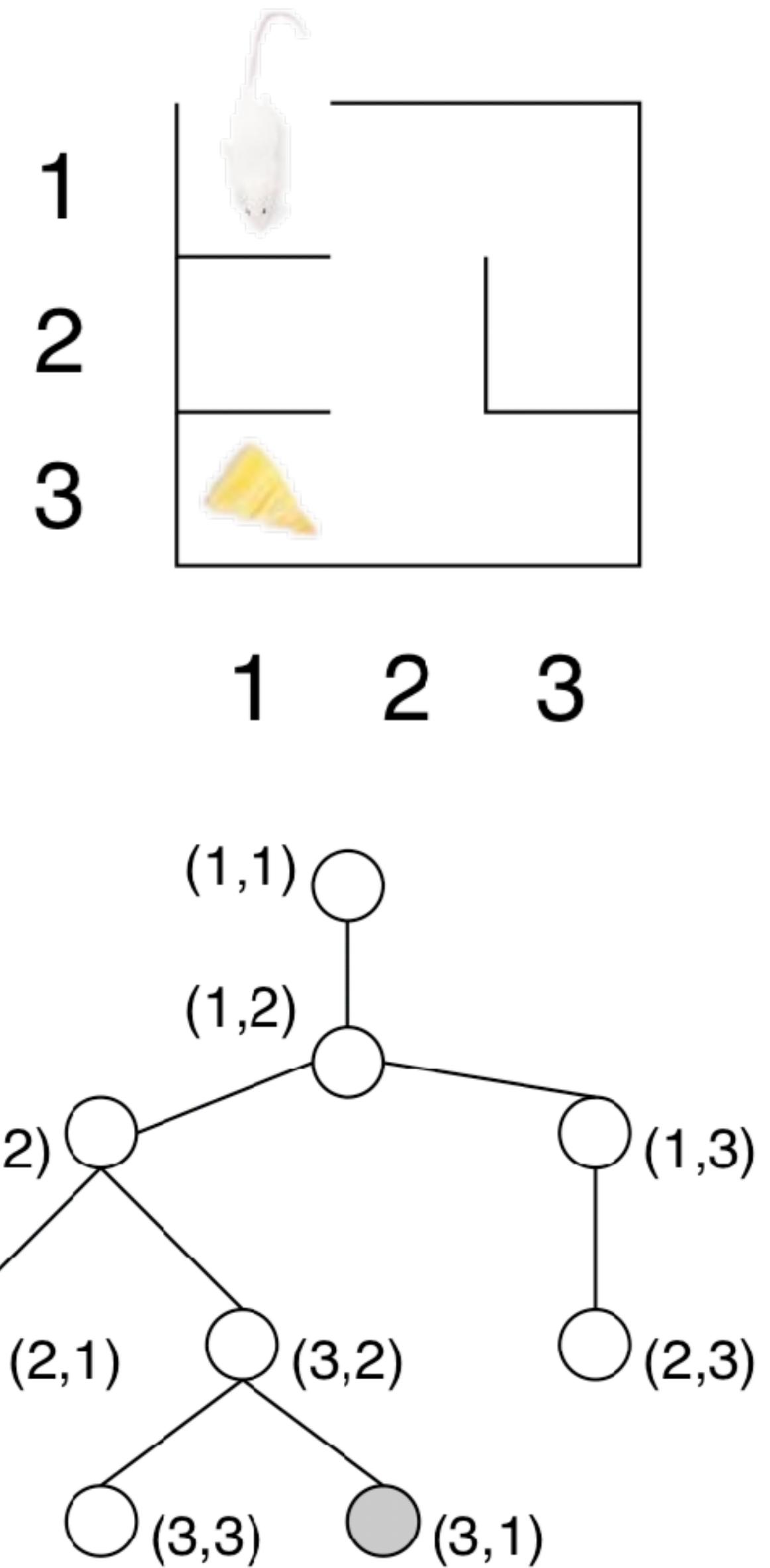
    // Die kleinste Scheibe befindet sich auf dem rechten Stapel.
    int k = 2;

    while (true) {
        // Kleinste Scheibe verschieben.
        final int i = k;
        k = (k + schritt) % stapel.length;
        stapel[k].push(stapel[i].pop());
        anzeigen();

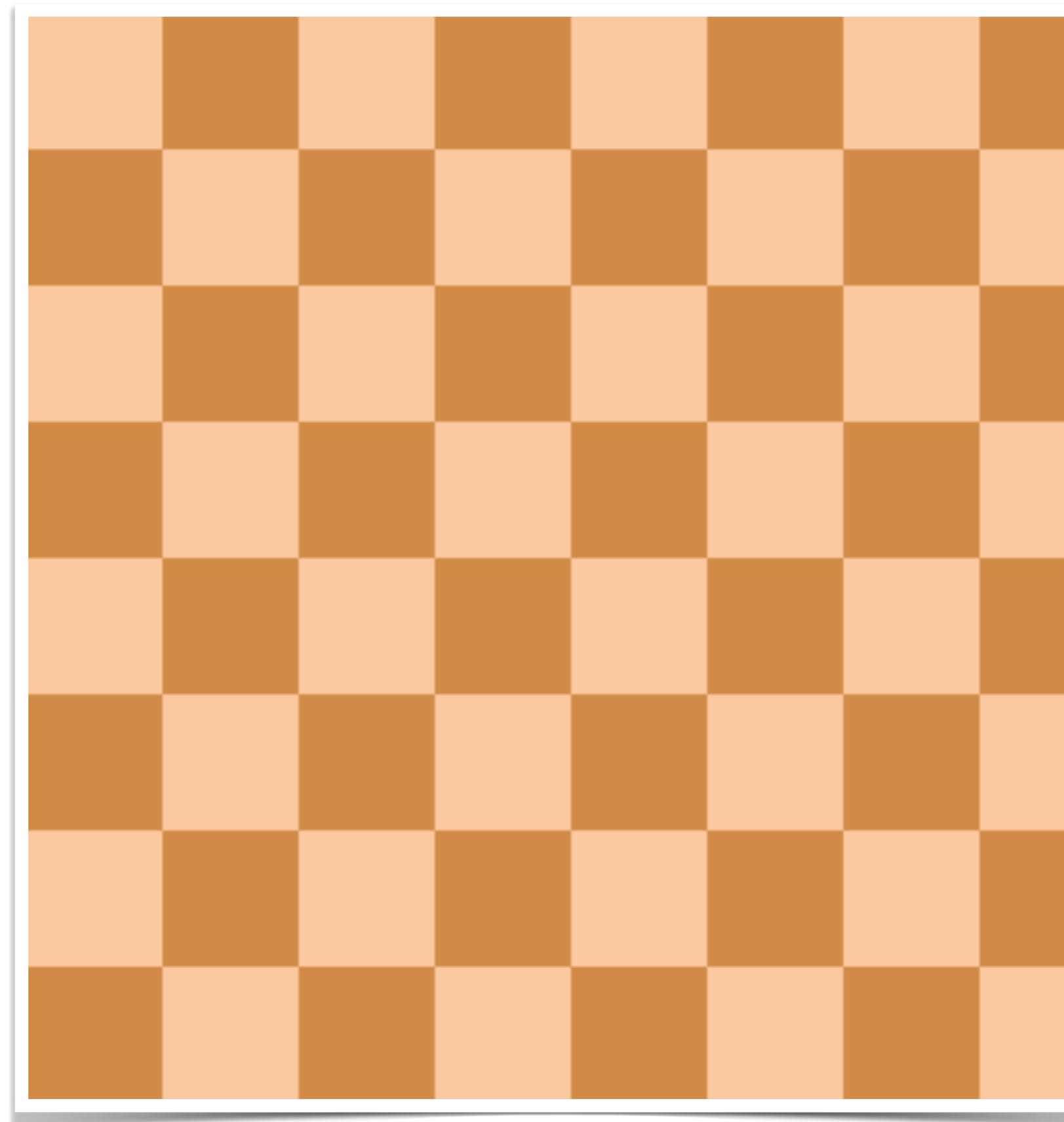
        // Fertig, wenn kleinste Scheibe wieder auf vollem Stapel
        if (stapel[k].size() == anzahlScheiben) {
            break;
        }
    }
}
```

# 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



Queens [~/Documents/Lehre/Vorlesungen/PI-2/SoSe 19/Beispiele/05/Queens] - .../src/de/uni\_bremen/Queens.java

Project: Queens

Queens.java

```
25     }
26     }
27     }
28     }
29     @
30     private static boolean isSafe(final int[] columnInRows)
31     {
32         for (int other = 0; other < row; ++other) {
33             if (columnInRows[row] == columnInRows[other]
34                 || columnInRows[row] + other - row
35                 || columnInRows[row] + row - other
36             )
37             return false;
38         }
39         return true;
40     }
41
42     public static void draw(final int[] columnInRows)
43     {
44         if (columnInRows == null) {
45             System.out.println("Keine Lösung!");
46         }
47         else {
48             ...
49         }
50     }
51 }
```

Ant Build

Maven

Scratches and Consoles

JShell Console

Event Log

36:14 LF UTF-8 4 spaces

# 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

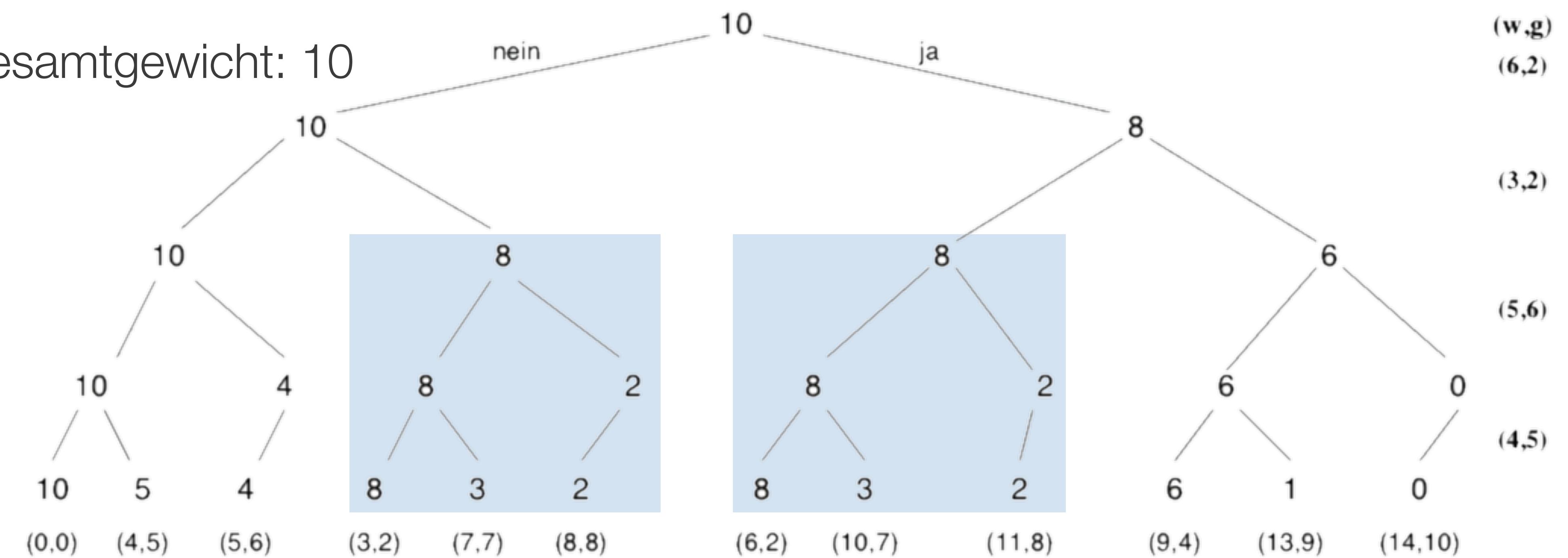


## Dynamische Programmierung: Rucksackproblem – Beispiel

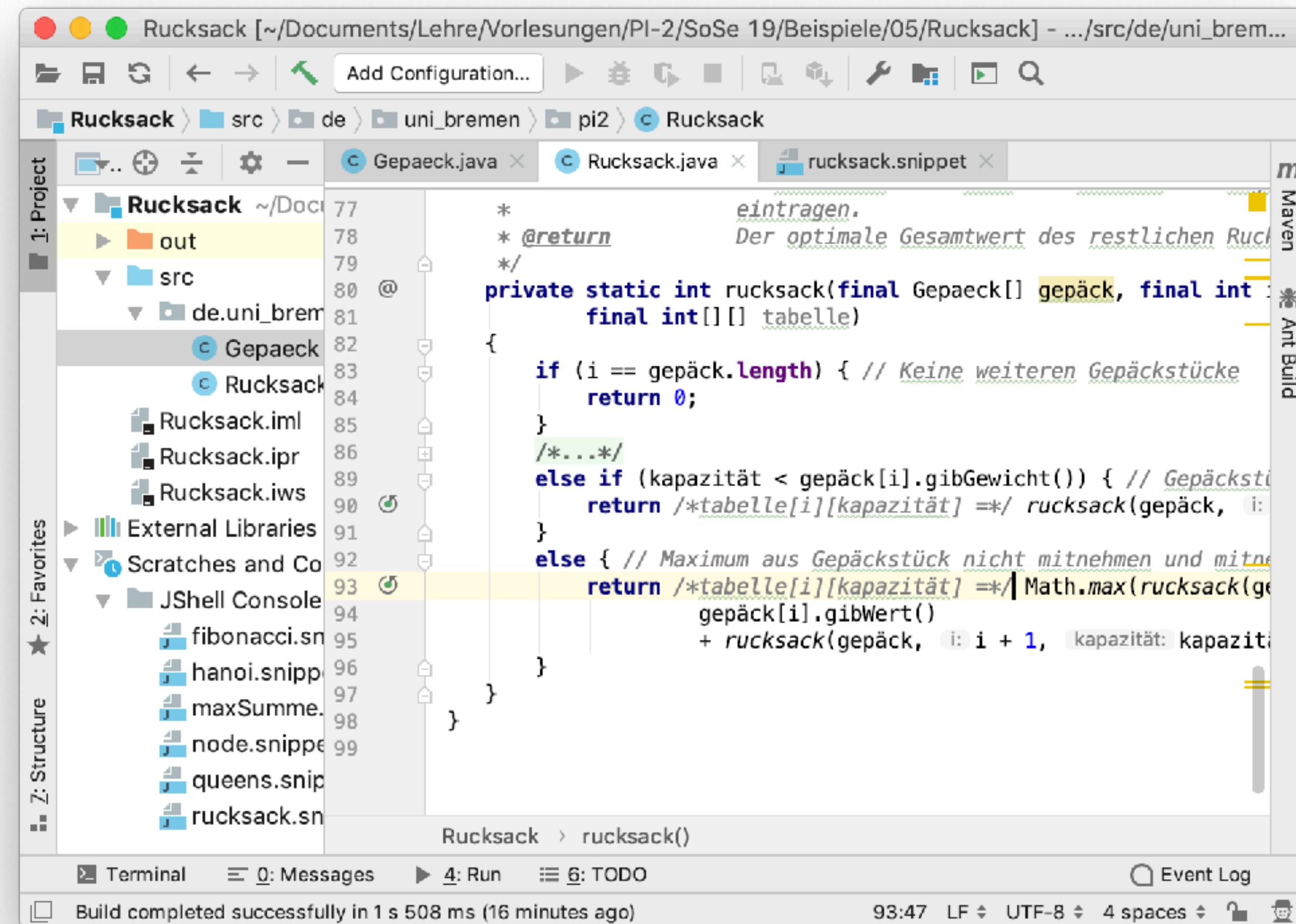
- Gewichte  $g_1 = 2, g_2 = 2, g_3 = 6, g_4 = 5$

- Werte  $w_1 = 6, w_2 = 3, w_3 = 5, w_4 = 4$

- Maximales Gesamtgewicht: 10



# Dynamische Programmierung: Rucksackproblem



The screenshot shows an IDE interface with the following details:

- Project:** Rucksack
- File:** Rucksack.java
- Code (Java):**

```

    /*
     * @return Der optimale Gesamtwert des restlichen Rucksacks
     */
    private static int rucksack(final Gepaeck[] gepaechk, final int[] kapazitaet, final int i, final int[][] tabelle) {
        if (i == gepaechk.length) { // Keine weiteren Gepäckstücke
            return 0;
        }
        /*...*/
        else if (kapazitaet < gepaechk[i].gibGewicht()) { // Gepäckstück kann nicht mitgenommen werden
            return /*tabelle[i][kapazitaet] */ rucksack(gepaechk, kapazitaet, i);
        }
        else { // Maximum aus Gepäckstück nicht mitnehmen und mitnehmen
            return /*tabelle[i][kapazitaet] */ Math.max(rucksack(gepaechk, kapazitaet, i, tabelle), rucksack(gepaechk, kapazitaet - gepaechk[i].gibWert(), i + 1, tabelle));
        }
    }

```

# Zusammenfassung der Konzepte

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