CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Großübung

CARL VON OSSIETZKY universität OLDENBURG

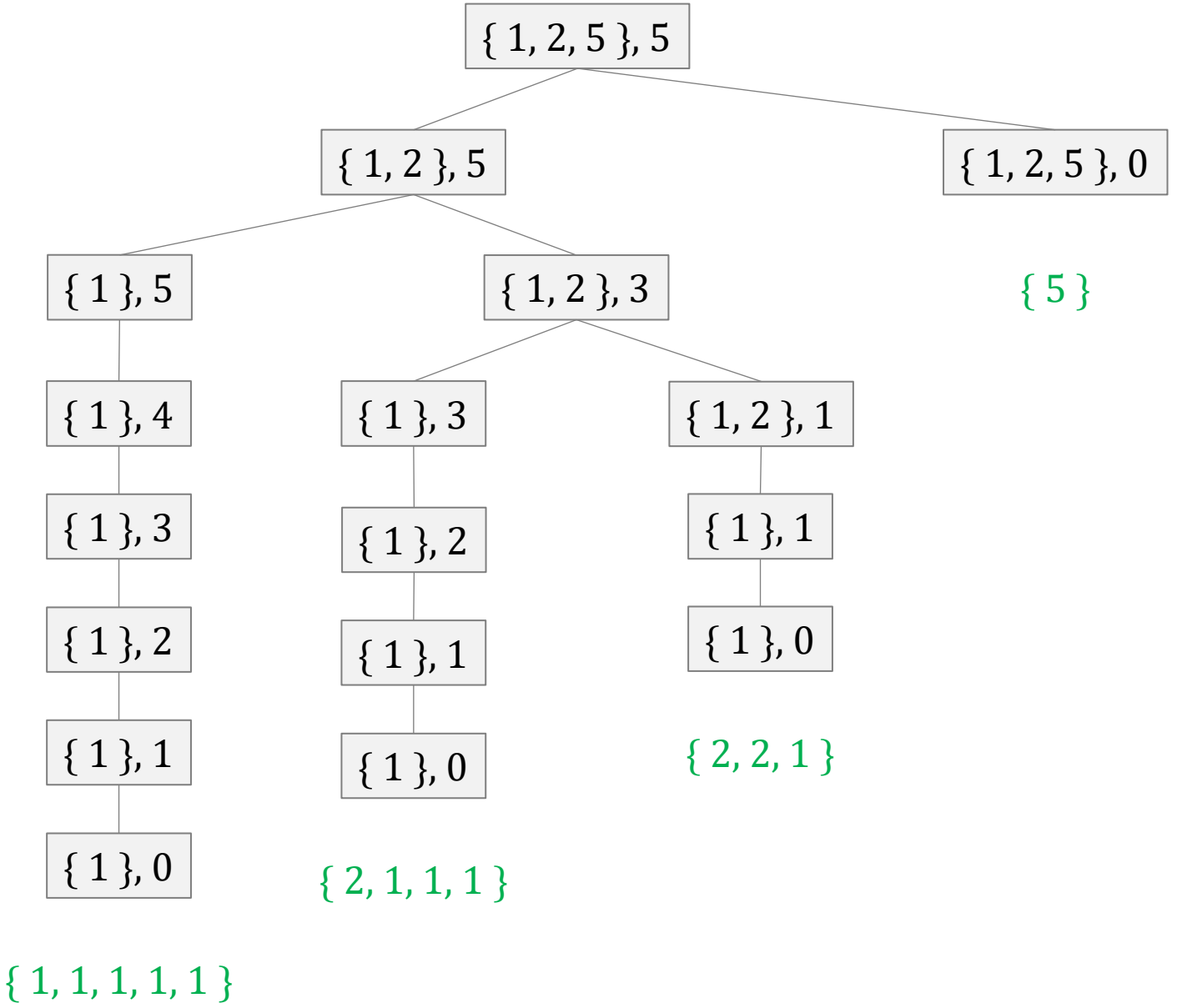
Wechselgeld

Problemstellung 1

Wechselgeld

- Wechselgeld im Wert von m Cent
- Wie viele (welche?) Möglichkeiten gibt es, unter Verwendung von Münzen aus $\mathrm{C} = \{ c_0, c_1, \ldots, c_n \}$ auf die Summe von m Cent zu kommen?
- Beispiel:
  - $\mathrm{C} = \{ 1, 2, 5 \}$
  - $\mathrm{m} = 5$
  - vier mögliche Lösungen:
    $\{ 1, 1, 1, 1, 1 \}$
    $\{ 1, 1, 1, 2 \}$
    $\{ 1, 2, 2 \}$
    $\{ 5 \}$

Rekursion


{ 1, 2, 5 }, 5
{ 1, 2 }, 5
{ 1, 2, 5 }, 0
{ 1 }, 5
{ 1, 2 }, 3
{ 5 }
{ 1 }, 4
{ 1 }, 3
{ 1, 2 }, 1
{ 1 }, 3
{ 1 }, 2
{ 1 }, 1
{ 1 }, 2
{ 1 }, 1
{ 1 }, 0
{ 1 }, 1
{ 1 }, 0
{ 2, 2, 1 }
{ 1 }, 0
{ 2, 1, 1, 1 }
{ 1, 1, 1, 1, 1 }

Programmrahmen

```
public abstract class Change {

    /**
     * Calculates the number of solutions for the coin-change-problem, given
     * a set of coins <code>coins</code> and an amount of money <code>money</code>
     * @param coins the set of coins
     * @param money the amount of money
     * @return the number of possible solutions
     */
    public abstract int count(int[] coins, int money);

    public static void main(String[] args) {
        int[] coins1 = new int[] { 1, 2, 5 };
        int[] coins2 = new int[] { 1, 2, 5, 10, 20, 50 };
        Change c = new Change???();
        System.out.println(c.count(coins1, 5));   System.out.println(c.count(coins1, 12));
        System.out.println(c.count(coins1, 123)); System.out.println(c.count(coins2, 123));
        System.out.println(c.count(coins2, 213)); System.out.println(c.count(coins2, 321));
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Aufgabe 1a

- Schreiben Sie eine Klasse **`ChangeRecursive`**, die von **`Change`** erbt und die Methode
  `int count(int[] coins, int money)`
  rekursiv implementiert.
- Hinweis: Schreiben Sie dazu eine Hilfsmethode
  `int count(int[] coins, int money, int index)`
  die über den **`index`** das **`coins`**-Array schrittweise verkleinern kann.
- Hinweis: Denken Sie an die **Basisfälle**!

CARL VON OSSIETZKY universität OLDENBURG

Mögliche Lösung

```
4
13
806
9838
89798
529566
```

```java
public class ChangeRecursive extends Change {

    @Override
    public int count(int[] coins, int money) {
        return count(coins, money, coins.length - 1);
    }

    private int count(int[] coins, int money, int index) {
        if (money < 0) {
            return 0;
        }
        if (money == 0) {
            return 1;
        }
        if (index < 0) {
            return 0;
        }
        return count(coins, money, index - 1) + count(coins, money - coins[index], index);
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Aufgabe 1b

- Schreiben Sie eine Klasse **`ChangeGreedy`**, die von **`Change`** erbt und die Methode
  `int count(int[] coins, int money)`
  greedy implementiert.
- Hinweis: Sie dürfen annehmen, dass das **`coins`**-Array aufsteigend sortiert ist (also z.B. $\{ 1, 2, 5 \}$).

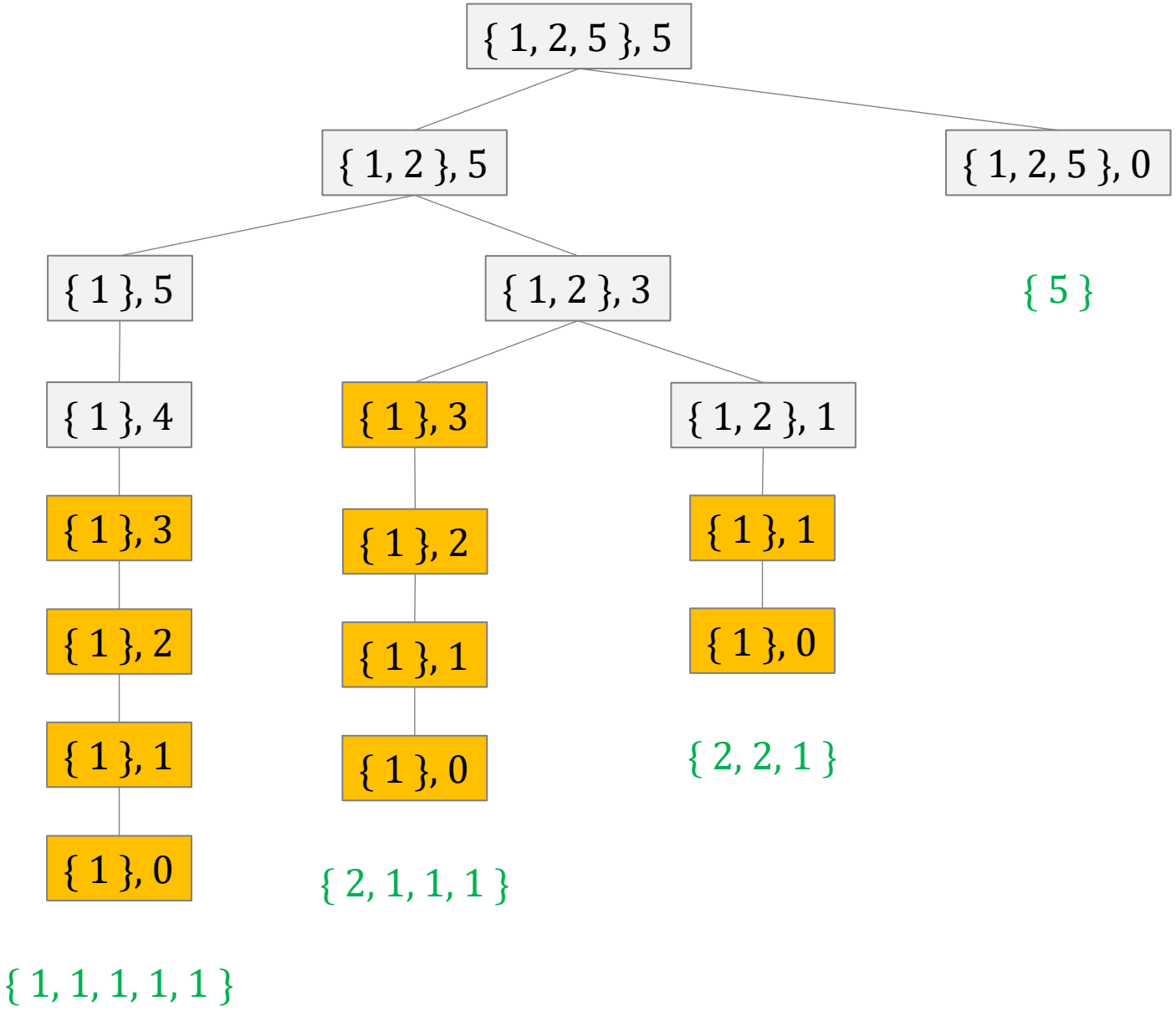
CARL VON OSSIETZKY universität OLDENBURG

Mögliche Lösung

```
1
1
1
1
1
1
```

```
private int count(int[] coins, int money, int index) {
    if (money < 0) {
        return 0;
    }
    if (money == 0) {
        return 1;
    }
    if (index < 0) {
        return 0;
    }
    if (coins[index] <= money) {
        return count(coins, money - coins[index], index);
    } else {
        return count(coins, money, index - 1);
    }
}
```

Überlappende Teilprobleme



Aufgabe 1c

- Schreiben Sie eine Klasse **`ChangeDynamic`**, die von **`Change`** erbt und die Methode
  `int count(int[] coins, int money)`
  mit **dynamischer Programmierung** und **Memoisation** implementiert.
- Hinweis: Sie dürfen die Klasse **`Pair<T>`** verwenden, die Paare von Werten kapselt und die die **`equals`** und **`hashCode`**-Methoden implementiert.

```
public class Pair<T> {
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
}}
```

Mögliche Lösung

```
private Map<Pair<Integer>, Integer> cache = new HashMap<>();

private int count(int[] coins, int money, int index) {
    if (money < 0) {
        return 0;
    }
    if (money == 0) {
        return 1;
    }
    if (index < 0) {
        return 0;
    }
    Pair<Integer> pair = new Pair<>(money, index);
    Integer result = cache.get(pair);
    if (result != null) {
        return result;
    }
    result = count(coins, money, index - 1) + count(coins, money - coins[index], index);
    cache.put(pair, result);
    return result;
}
```

```
     4
    13
   806
  9838
 89798
529566
```

CARL VON OSSIETZKY universität OLDENBURG

Alternativlösung

```
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

`table[i]` enthält die Anzahl von Lösungen für den Geldbetrag i, für $0 \leq i \leq \text{money}$.

Für jede Münze wird die Tabelle an den Stellen, welche diese Münze enthalten können, aktualisiert:

- das sind die Stellen, an denen der Geldbetrag mindestens so groß ist wie der Betrag der Münze
- zu den bisher gefundenen Lösungen werden die Lösungen für den aktuellen Betrag ohne die aktuelle Münze hinzugezählt

CARL VON OSSIETZKY universität OLDENBURG

Alternativlösung

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	0	0	0	0	0

Alternativlösung

```
coin = 1
amount = 1
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	0	0	0	0

Alternativlösung

coin = 1
amount = 2

```
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **count({ 1, 2, 5 }, 5)**

Index	0	1	2	3	4	5
Wert	1	1	1	0	0	0

Alternativlösung

```
coin = 1
amount = 3
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	1	1	0	0

Alternativlösung

```
coin = 1
amount = 4
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	1	1	1	0

CARL VON OSSIETZKY universität OLDENBURG

Alternativlösung

coin = 1
amount = 5

```
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **count({ 1, 2, 5 }, 5)**

Index	0	1	2	3	4	5
Wert	1	1	1	1	1	1

CARL VON OSSIETZKY universität OLDENBURG

Alternativlösung

coin = 2
amount = 2

```
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **count({ 1, 2, 5 }, 5)**

Index	0	1	2	3	4	5
Wert	1	1	2	1	1	1

Alternativlösung

```
coin = 2
amount = 3
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	2	2	1	1

Alternativlösung

```
coin = 2
amount = 4
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	2	2	3	1

Alternativlösung

```
coin = 2
amount = 5
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	2	2	3	**3**

Alternativlösung

```
coin = 5
amount = 5
```

```java
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	2	2	3	**4**

CARL VON OSSIETZKY universität OLDENBURG

Alternativlösung

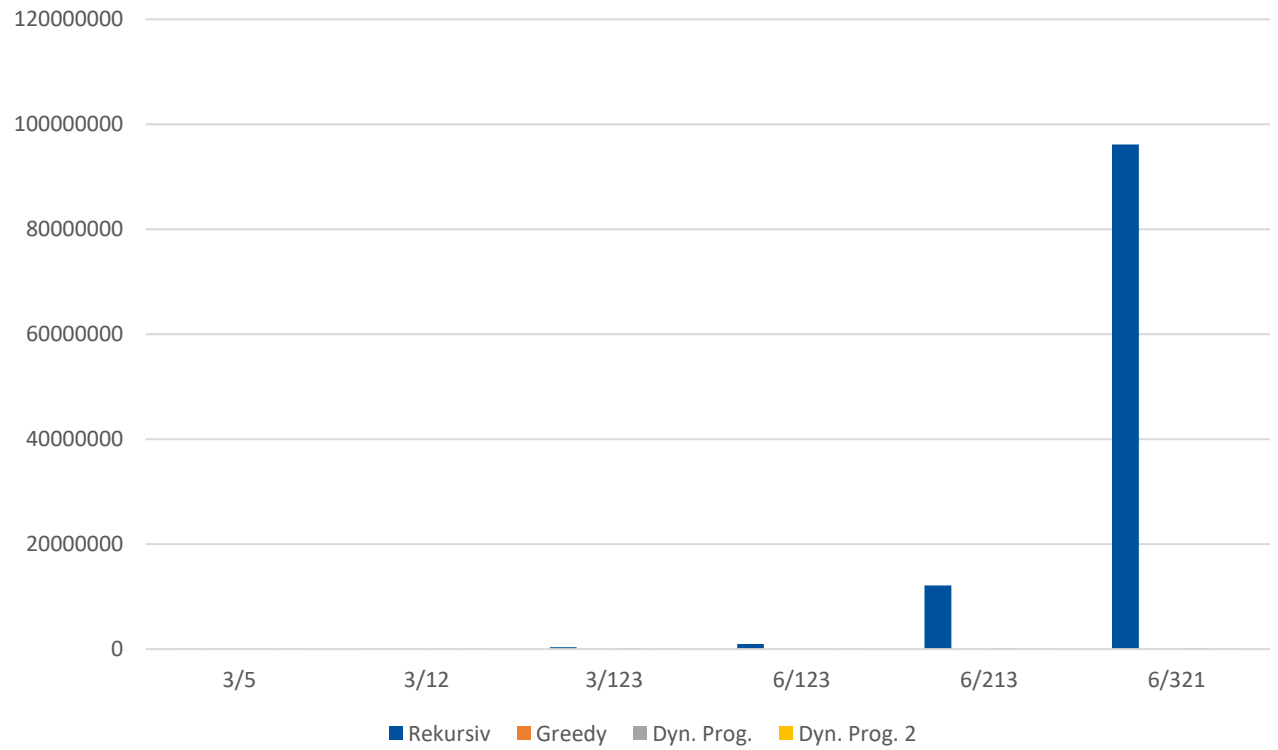
```
@Override
public int count(int[] coins, int money) {
    int[] table = new int[money + 1];
    table[0] = 1;
    for (int coin : coins) {
        for (int amount = coin; amount <= money; amount++) {
            table[amount] += table[amount - coin];
        }
    }
    return table[money];
}
```

Tabelle für **`count({ 1, 2, 5 }, 5)`**

Index	0	1	2	3	4	5
Wert	1	1	2	2	3	4

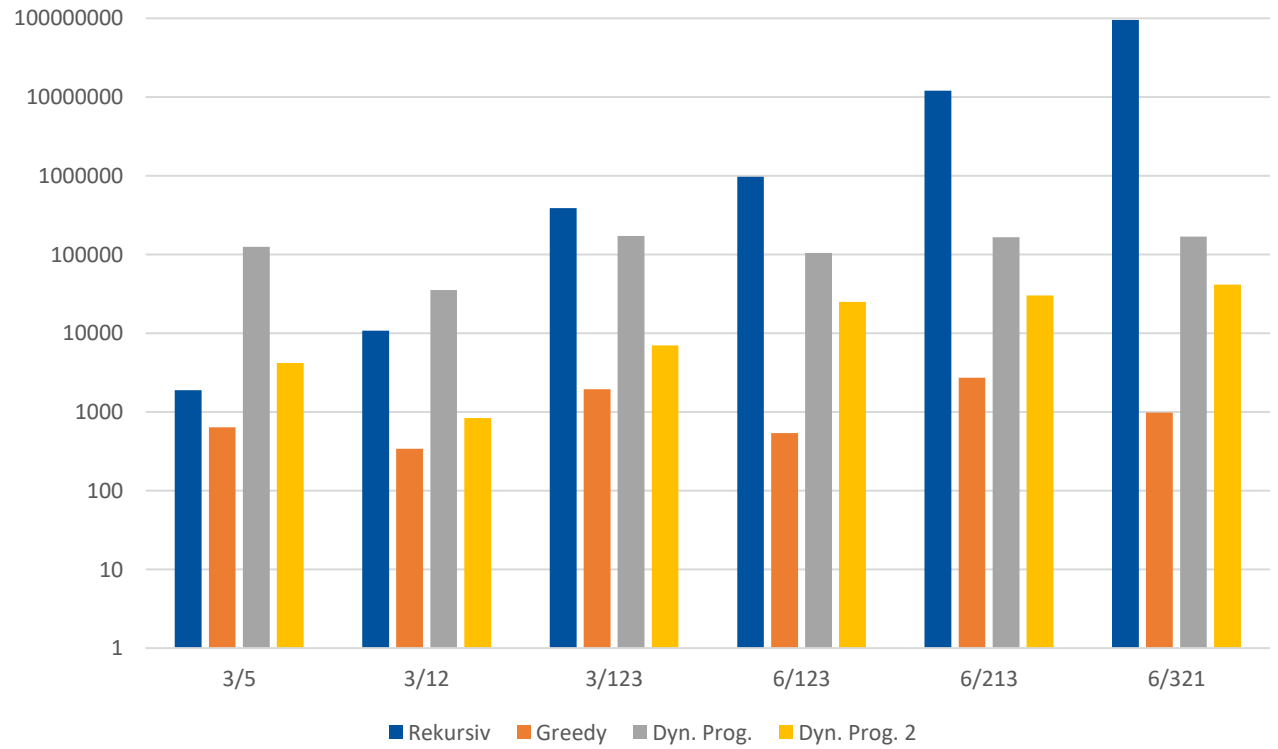
Messungen

	3/5	3/12	3/123	6/123	6/213	6/321
Rekursiv	0,001 ms	0,011 ms	0,391 ms	0,980 ms	12,133 ms	96,186 ms
Greedy	0,001 ms	0,000 ms	0,001 ms	0,001 ms	0,003 ms	0,001 ms
Dyn. Prog.	0,125 ms	0,035 ms	0,172 ms	0,105 ms	0,167 ms	0,169 ms
Dyn. Prog. 2	0,004 ms	0,001 ms	0,007 ms	0,025 ms	0,030 ms	0,042 ms



Messungen (log. Skalierung)

	3/5	3/12	3/123	6/123	6/213	6/321
Rekursiv	0,001 ms	0,011 ms	0,391 ms	0,980 ms	12,133 ms	96,186 ms
Greedy	0,001 ms	0,000 ms	0,001 ms	0,001 ms	0,003 ms	0,001 ms
Dyn. Prog.	0,125 ms	0,035 ms	0,172 ms	0,105 ms	0,167 ms	0,169 ms
Dyn. Prog. 2	0,004 ms	0,001 ms	0,007 ms	0,025 ms	0,030 ms	0,042 ms



CARL VON OSSIETZKY universität OLDENBURG

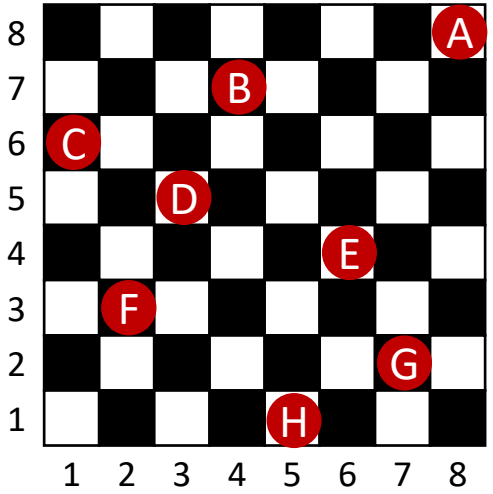
n-Damenproblem

Problemstellung 2

n-Damenproblem

- Wie können auf einem $n \times n$ Schachbrett n Damen positioniert werden, so dass keine Dame eine andere schlagen kann?
  - jede Dame darf jede andere schlagen (keine Farben)
  - übliche Bewegungsmuster (horizontal, vertikal, diagonal)

➢keine zwei Damen in der gleichen Zeile, Spalte oder Diagonale



Backtracking-Strategie

- Beginne mit trivialem Teilproblem mit Lösung s_n
- Wähle die erste Entscheidung d_n und erweitere s_n durch d_n zu s_{n-1}
- Fahre fort, bis durch die resultierende Entscheidungsfolge $(d_n, \ldots, d_1)$ eine vollständige Lösung s_0 konstruiert wurde
- Falls auf dem Weg keine Lösung konstruiert werden kann oder falls alle Lösungen konstruiert werden sollen:
  - revidiere die letzte Entscheidung
    (kehre von s_k zurück zu s_{k+1})
  - falls nun eine andere Entscheidung möglich ist
    → wähle nächste mögliche Entscheidung
    sonst: wiederhole Revidierungsschritt

Backtracking-Strategie für das n-Damenproblem

- Beginne mit leerem Feld
- Für alle Zeilen row:
  - platziere Dame an Position col, sofern col nicht bedroht wird
  - wenn die Dame platziert wurde, setze mit der nächsten Zeile fort
  - wenn die Dame nicht platziert wurde, gehe eine Zeile zurück und setze die Dame dort um (Revidierungsschritt/Backtracking)
  - wenn in dieser Zeile keine weitere Position möglich ist, gehe weiter zurück

CARL VON OSSIETZKY universität OLDENBURG

Backtracking-Strategie für das n-Damenproblem

```
Queens solution = new Queens(size)
boolean searchBT(int row)
    if row < size
        for col from 0 to size
            if isSafe(row, col))
                place(row, col)
                    if (searchBT(row + 1))
                        return true
                    else
                        remove(row, col)
        return false
    else
        return true
```

CARL VON OSSIETZKY universität OLDENBURG

Programmrahmen

```
public class Queens {

    /** Create an empty board of size x size */
    public Queens(int size) { … }

    /** Create a copy of the given board */
    public Queens(Queens other) { … }

    /** Create new board of size x size with a random placement of queens */
    public Queens(int size, Random random) { … }

    public boolean isSet(int row, int col) { … }
    public void place(int row, int col) { … }
    public void remove(int row, int col) { … }
    public int getSize() { … }
    public double getQuality() { … }
    public boolean isSafe(int row, int col) { … }
```

CARL VON OSSIETZKY universität OLDENBURG

Programmrahmen

```
    public static void main(String[] args) {
        Queens solution;

        System.out.println("Backtracking");
        BacktrackingQueens btr = new BacktrackingQueens();
        solution = btr.search(8);
        System.out.println(solution.getQuality());
        System.out.println(solution);
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Aufgabe 2a

- Schreiben Sie eine Klasse **`BacktrackingQueens`**, die die Methode **`Queens search(int size)`** per **Backtracking** implementiert.

CARL VON OSSIETZKY universität OLDENBURG

Mögliche Lösung

```
public class BacktrackingQueens {

    private Queens solution;

    public Queens search(int size) {
        solution = new Queens(size);
        if (searchBT(0)) {
            return solution;
        } else {
            return new Queens(0);
        }
    }
```

CARL VON OSSIETZKY universität OLDENBURG

Mögliche Lösung

```
    private boolean searchBT(int row) {
        if (row < solution.getSize()) {
            for (int col = 0; col < solution.getSize(); col++) {
                if (solution.isSafe(row, col)) {
                    solution.place(row, col);
                    if (searchBT(row + 1)) {
                        return true;
                    } else {
                        solution.remove(row, col);
                    }
                }
            }
            return false;
        } else {
            return true;
        }
    }
}
```

```
Backtracking
28.0
X-------
----X---
-------X
-----X--
--X-----
------X-
-X------
---X----
```

Voraussetzungen für Lokale Suche

- Initiale Startlösung?
  - Berechnung einer zulässigen suboptimalen Lösung, z.B. durch einfaches Greedy-Verfahren
- Nachbarschaft $N(x)$ einer Lösung x?
  - Menge der zulässigen Lösungen, die sich „wenig“ von x unterscheiden
- Abbruchbedingung?
  - kein Nachbar mit besserer Güte vorhanden?
  - Lösungsgüte erreicht?
  - maximale Anzahl an Iterationen oder Rechenzeit

➢ alle problemabhängig zu definieren

CARL VON OSSIETZKY universität OLDENBURG

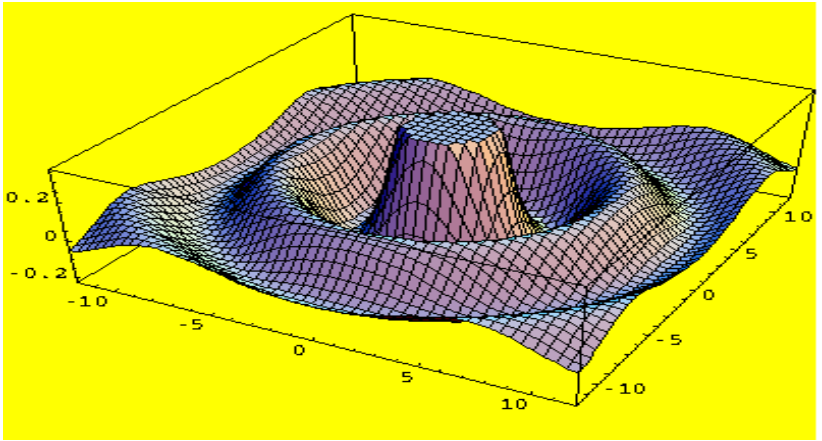
Programmrahmen

```
    public static void main(String[] args) {
        Queens solution;

        System.out.println("Local Search");
        LocalSearchQueens lsc = new LocalSearchQueens(8);
        solution = lsc.execute();
        System.out.println(solution.getQuality());
        System.out.println(solution);
    }

}
```

Lokale Suche

- Lokal: nächste Zwischenlösung wird nur lokal in der Nachbarschaft der aktuellen Lösung gesucht

```
x = startLoesung;
while (!abbruch) {
    waehle y aus Nachbarschaft N(x)
    if q(y) besser als q(x)
        x = y
}
```



CARL VON OSSIETZKY universität OLDENBURG

Programmrahmen

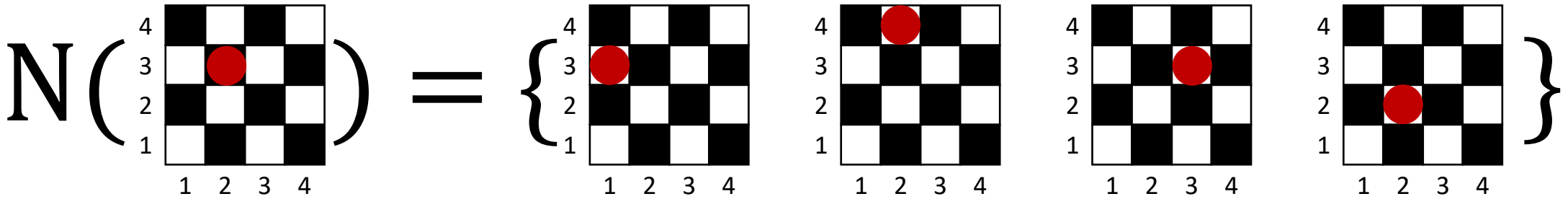
```
public abstract class LocalSearch<T> {

    protected int iterationCount;

    public abstract T getInitialSolution();

    public abstract boolean isTerminated();

    public abstract Collection<T> getNeighborhood(T current);

    public abstract double getQuality(T solution);
```

Programmrahmen

```
public T execute() {
    iterationCount = 0;
    T currentSolution = getInitialSolution();
    boolean change = true;
    while (change && !isTerminated()) {
        change = false;
        double qualityOfCurrentSolution = getQuality(currentSolution);
        Collection<T> neighborhood = getNeighborhood(currentSolution);
        for (T otherSolution : neighborhood) {
            double qualityOfOtherSolution = getQuality(otherSolution);
            if (qualityOfOtherSolution > qualityOfCurrentSolution) {
                currentSolution = otherSolution;
                qualityOfCurrentSolution = qualityOfOtherSolution;
                change = true;
                //break;
            }
        }
        iterationCount++;
    }
    return currentSolution;
}}
```

Aufgabe 2b

- Schreiben Sie eine Klasse **`LocalSearchQueens`**, die von **`LocalSearch<Queens>`** erbt und diese Methoden implementiert:
  - **`Queens getInitialSolution()`**
  - **`boolean isTerminated()`**
  - **`Collection<Queens> getNeighborhood(Queens current)`**
  - **`double getQuality(Queens solution)`**
- Hinweis: Die Nachbarschaft einer Damen-Belegung kann z.B. darin bestehen, jede Dame um ein Feld in jede mögliche Richtung zu verschieben.



CARL VON OSSIETZKY universität OLDENBURG

Mögliche Lösung

```
public class LocalSearchQueens extends LocalSearch<Queens> {

    private int size;

    public LocalSearchQueens(int size) { this.size = size; }

    @Override
    public Queens getInitialSolution() {
        return new Queens(size, new Random());
    }

    @Override
    public boolean isTerminated() {
        return iterationCount > 1000;
    }

    @Override
    public double getQuality(Queens solution) {
        return solution.getQuality();
    }
```

Mögliche Lösung

```
    @Override
    public Collection<Queens> getNeighborhood(Queens current) {
        Collection<Queens> result = new HashSet<>();
        for (int row = 0; row < size; row++) {
            for (int col = 0; col < size; col++) {
                if (current.isSet(row, col)) {
                    if (row > 0) {
                        Queens other = new Queens(current);
                        other.remove(row, col);
                        other.place(row - 1, col);
                        result.add(other);
                    }
                    if (row < size - 1) { … }
                    if (col > 0) { … }
                    if (col < size - 1) { … }
                }
            }
        }
        return result;
    }
}
```

```
Local Search
25.0
------X-
----X---
------X-
X-------
-----X--
-------X
----X---
--X-----
```