Informatik II - Übung 8

Pascal Schärli

knapsack@pascscha.ch

13.11.2020

ETH zürich

Nachbesprechung

Bild: look again by corpus delicti from the Noun Project

Array-Listen und Generics

```java
private boolean filter(Student student) {
    return student.getPoints() >= (IFilter.criteria / 100 * IFilter.maxNumberofPoints);
}

public ArrayList filterRaw(ArrayList groups) {
    ArrayList result = new ArrayList();
    for (int i = 0; i < groups.size(); i++) {
        ArrayList group = (ArrayList) groups.get(i);
        for (int j = 0; j < group.size(); j++) {
            Student student = (Student) group.get(j);
            if (filter(student)) {
                result.add(student);
            }
        }
    }
    return result;
}

public ArrayList<Student> filterGeneric(ArrayList<ArrayList<Student>> groups) {
    ArrayList<Student> result = new ArrayList<Student>();
    for (int i = 0; i < groups.size(); i++) {
        ArrayList<Student> group = groups.get(i);
        for (int j = 0; j < group.size(); j++) {
            Student student = group.get(j);
            if (filter(student)) {
                result.add(student);
            }
        }
    }
    return result;
}
```

Reversi Teil 1

```java
public Coordinates nextMove(GameBoard gb) {
  ArrayList<Coordinates> validMoves = new ArrayList<Coordinates>(gb.getSize() * gb.getSize());

  for (int row = 1; row <= gb.getSize(); row++) {
    for (int col = 1; col <= gb.getSize(); col++) {
      Coordinates coord = new Coordinates(row, col);
      if (gb.checkMove(color, coord)) {
        validMoves.add(coord);
      }
    }
  }

  if (validMoves.size() > 0) {
    int randIndex = rand.nextInt(validMoves.size());
    return validMoves.get(randIndex);
  }
  else {
    return null;
  }
}
```

Best of Vorlesung

Binäre Suche

Denkt an eine Zahl zwischen 0 und 1023

https://pascscha.ch/info2/binary_search.html

Binäre Suche

- Gegeben wird ein sortierter Array [1,2,4,6,9,10,14,16,17,20,23,26]

- Welcher Index hat 16?
  1. Alle Elemente durchprobieren.
     $\rightarrow$ nicht sehr effizient

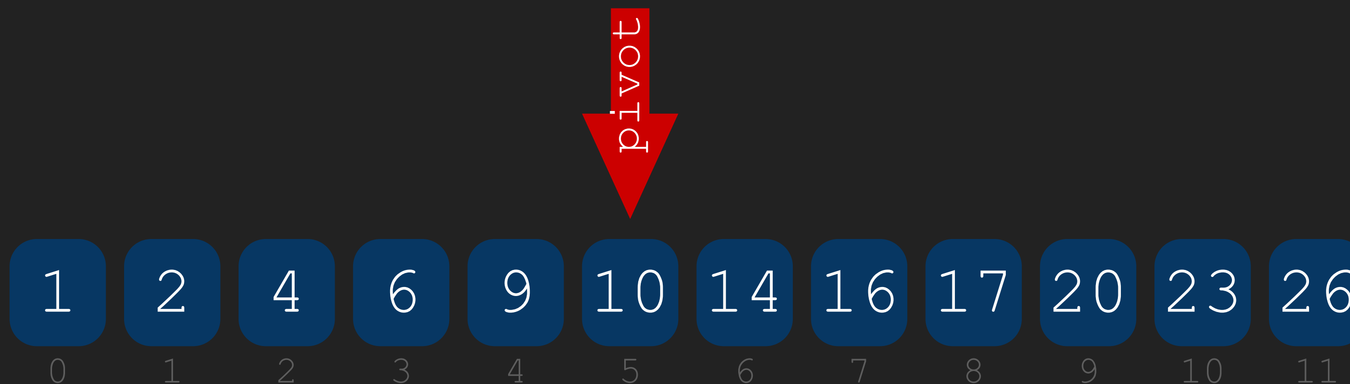
  2. Eventuell kann man ausnutzen, dass der Array bereits sortiert ist?
     $\rightarrow$ Binäre Suche!

Binäre Suche

```
1  Binäre Suche:
2
3  Gesuchter Wert mit der Mitte/Pivot vergleichen.
4
5  Gleich wie die Mitte?
6          -> Gefunden!
7
8  Grösser als die Mitte?
9          -> rechts weitersuchen
10
11 Kleiner als die Mitte?
12     -> links weitersuchen
```

Vergleich:

16 > 10

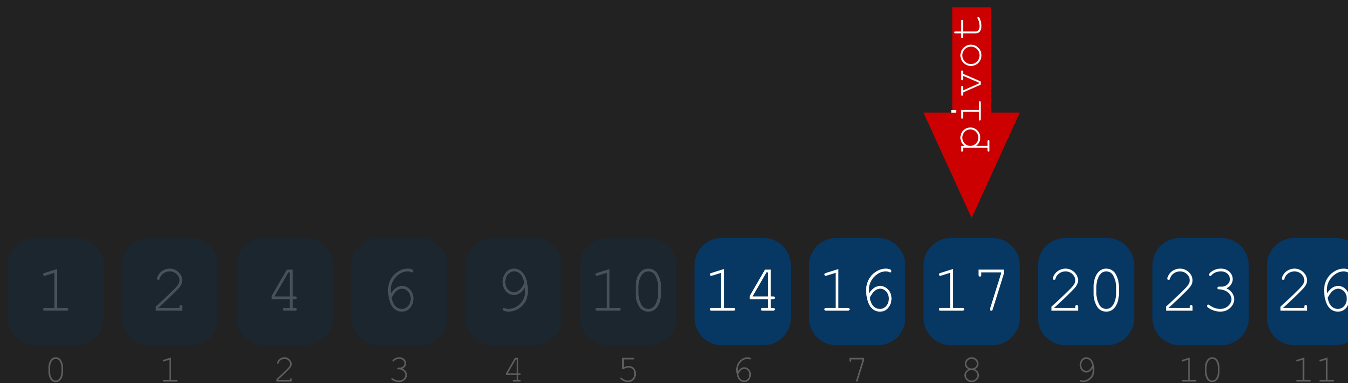


Binäre Suche

```
1  Binäre Suche:
2
3  Gesuchter Wert mit der Mitte/Pivot vergleichen.
4
5  Gleich wie die Mitte?
6          -> Gefunden!
7
8  Grösser als die Mitte?
9          -> rechts weitersuchen
10
11 Kleiner als die Mitte?
12     -> links weitersuchen
```

Vergleich:

16 < 17

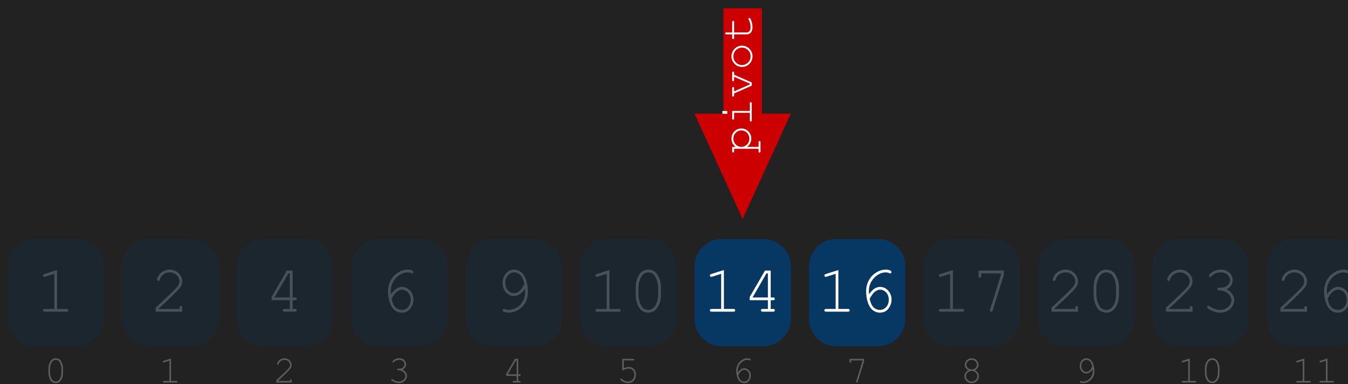


Binäre Suche

```
 1 Binäre Suche:
 2
 3 Gesuchter Wert mit der Mitte/Pivot vergleichen.
 4
 5 Gleich wie die Mitte?
 6         -> Gefunden!
 7
 8 Grösser als die Mitte?
 9         -> rechts weitersuchen
10
11 Kleiner als die Mitte?
12     -> links weitersuchen
```

Vergleich:

16 > 14

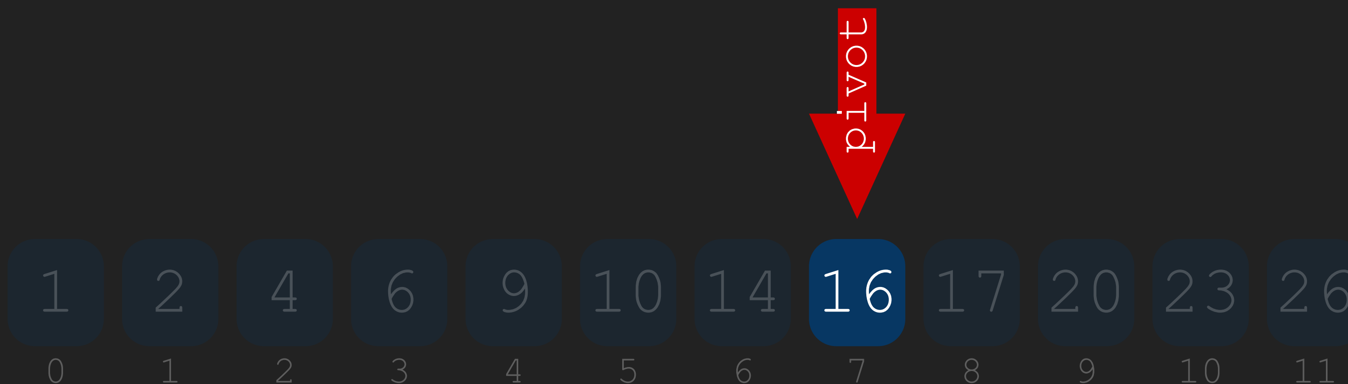


Binäre Suche

```
1  Binäre Suche:
2
3  Gesuchter Wert mit der Mitte/Pivot vergleichen.
4
5  Gleich wie die Mitte?
6          -> Gefunden!
7
8  Grösser als die Mitte?
9          -> rechts weitersuchen
10
11 Kleiner als die Mitte?
12     -> links weitersuchen
```

Vergleich:

16 = 16



Backtracking

- In Informatik I musstet ihr einen Sudoku-Solver Programmieren
- Die einfachste Lösung wäre alle möglichen Zahlenkombinationen auszuprobieren.
  - Code ist kurz und überschaubar
  - Die Laufzeit ist dafür unmenschlich lange
- Darum habt also nur Positionen geprüft, welche die Sudoku Regeln nicht verletzen.
  - Euer Programm wurde komplexer
  - Die Laufzeit war jedoch viel schneller.

Backtracking - Damen

- Annahme: 10'000 Positionen/Sekunde, 8x8 Spielbrett

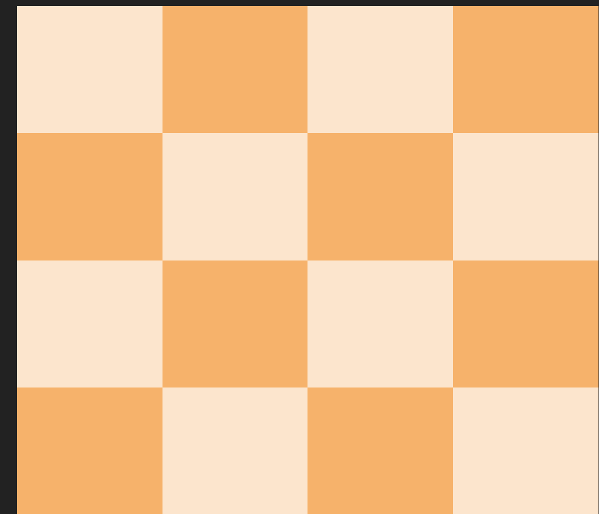
- Brute Force:
  - $8^8 = 16'777'216$ Pos.
  - ≈ 28 min

- Backtracking:
  - 15'720 Pos.
  - ≈ 1.5 sek

Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

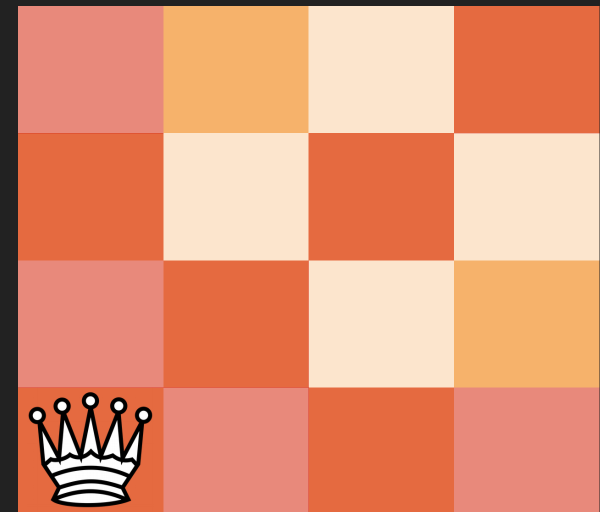
```
 1  Falls in der ersten unbesetzten Zeile
    Platz vorhanden ist
 2      -> setze möglichst weit unten
 3
 4  Falls kein Platz vorhanden ist
 5      -> Backtracking
 6
 7  Die Zeile vorher machte Backtracking
 8       Falls es oberhalb noch platz hat
 9           -> Dame nach oben verschieben
10       Sonst
11           -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

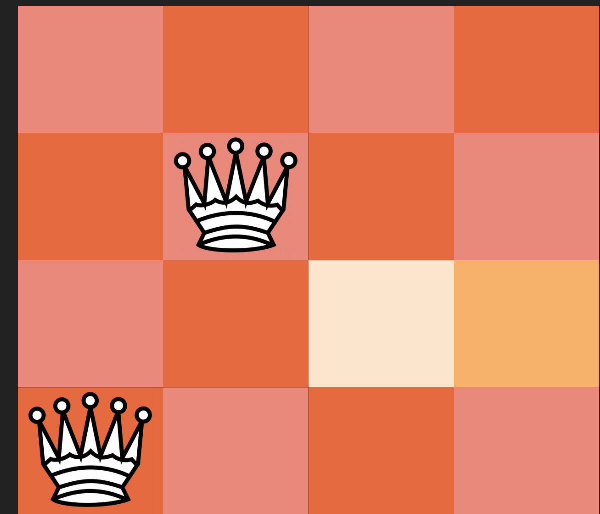
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

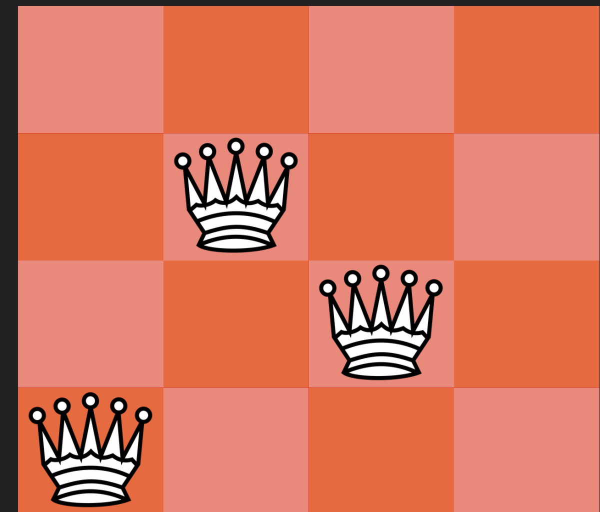
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

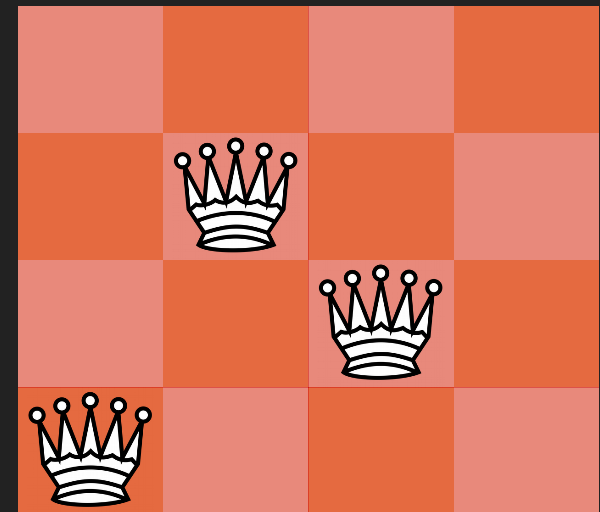
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```

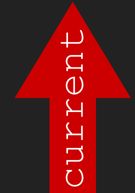


Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```

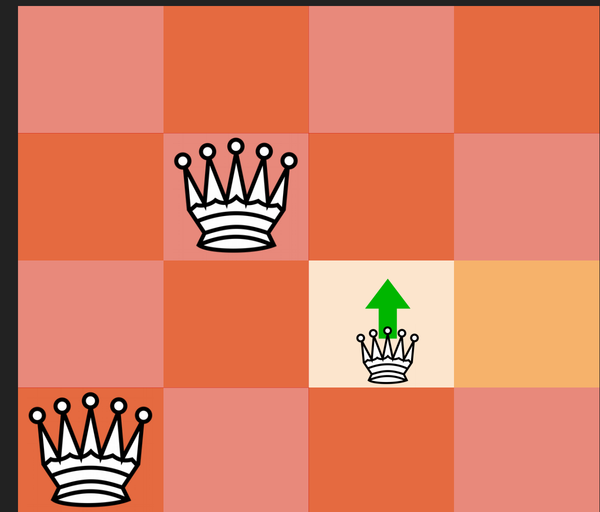




Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

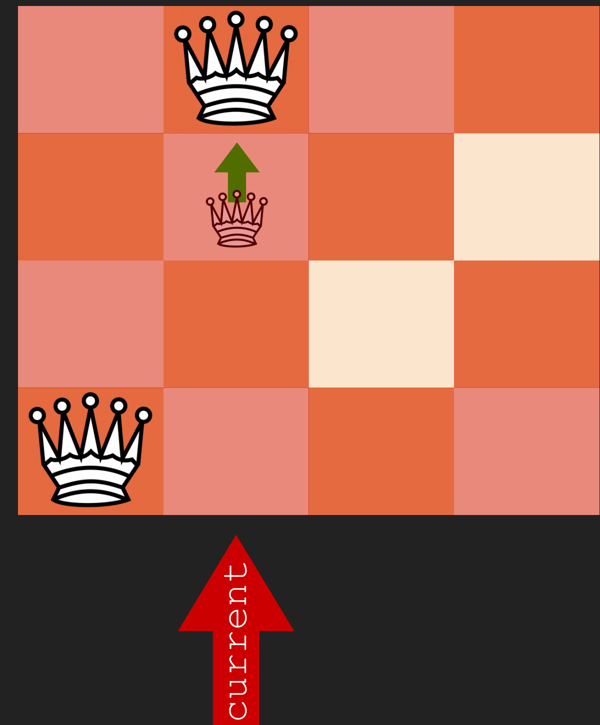
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

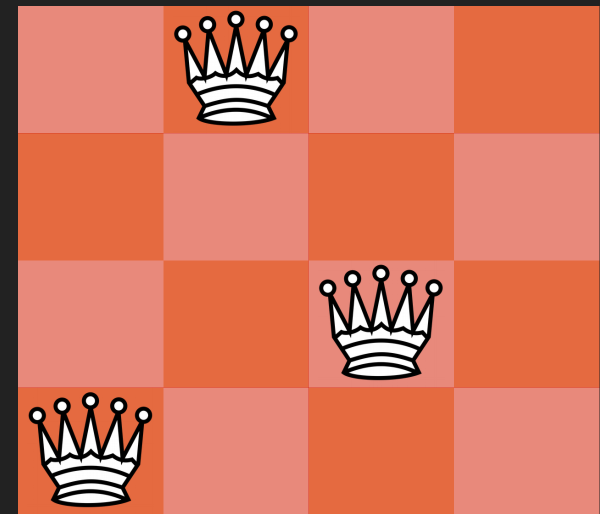
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

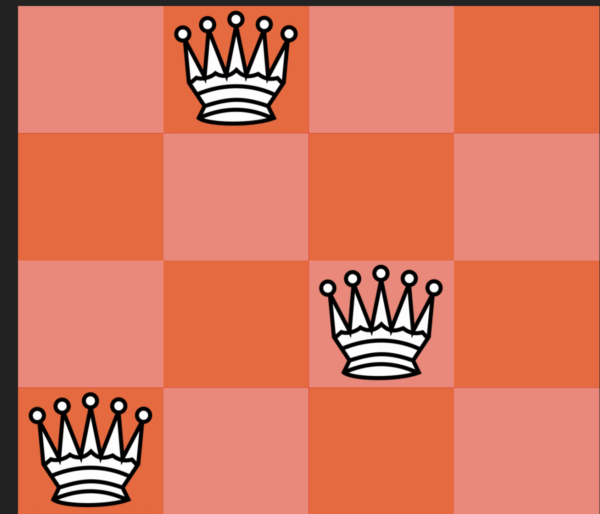
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

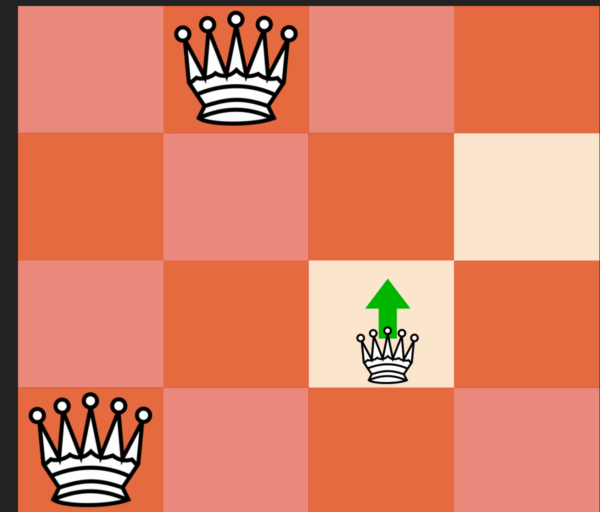
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

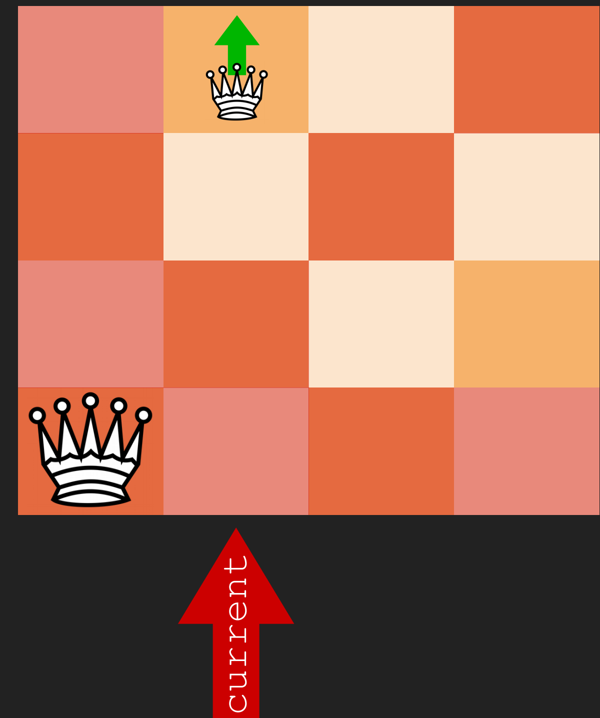
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

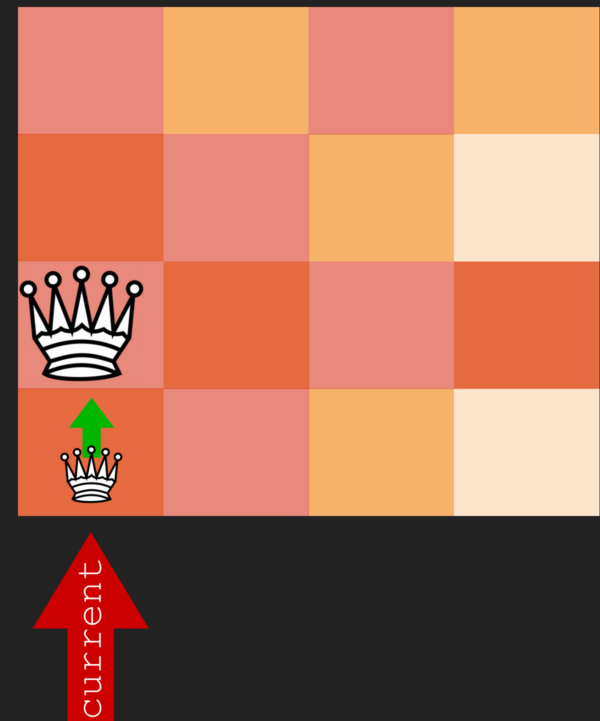
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

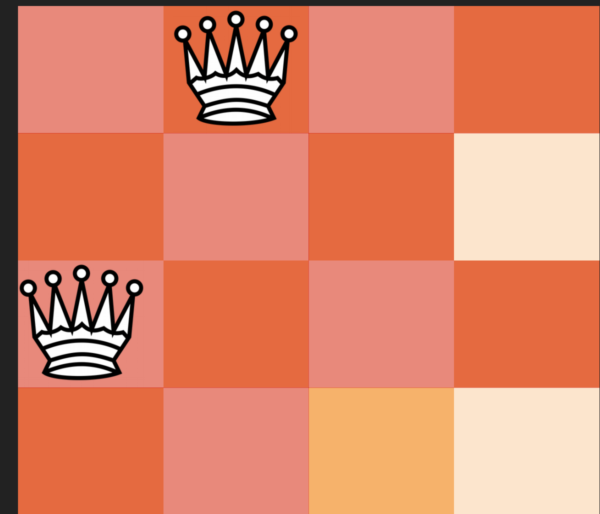
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

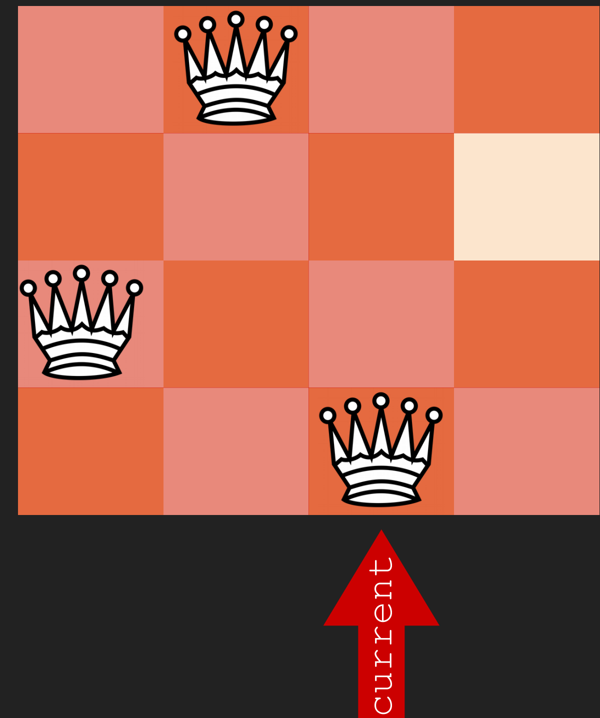
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

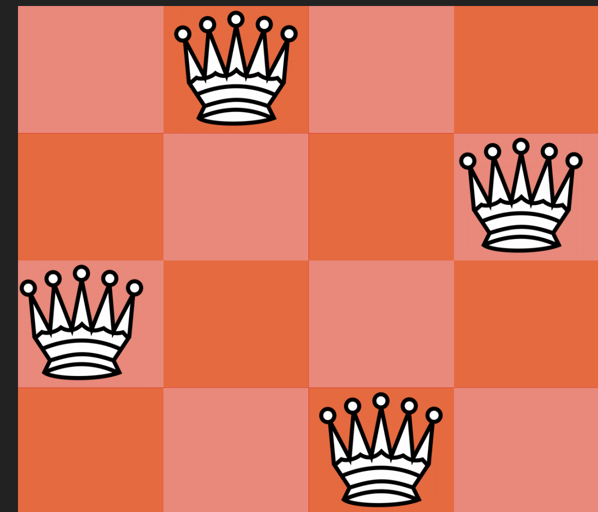
```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

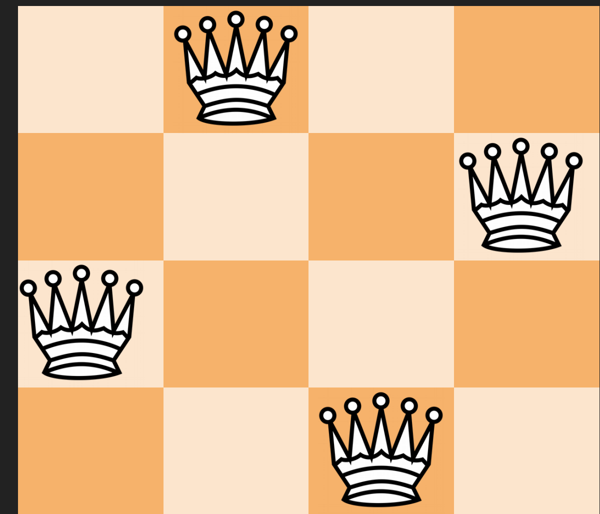
```
 1  Falls in der ersten unbesetzten Zeile
    Platz vorhanden ist
 2      -> setze möglichst weit unten
 3
 4  Falls kein Platz vorhanden ist
 5      -> Backtracking
 6
 7  Die Zeile vorher machte Backtracking
 8       Falls es oberhalb noch platz hat
 9           -> Dame nach oben verschieben
10       Sonst
11           -> Backtracking
```



Backtracking - Damen

Setze in jede Spalte eine Dame, so dass sich keine zwei Damen in einem Zug erreichen können:

```
1  Falls in der ersten unbesetzten Zeile
   Platz vorhanden ist
2      -> setze möglichst weit unten
3
4  Falls kein Platz vorhanden ist
5      -> Backtracking
6
7  Die Zeile vorher machte Backtracking
8       Falls es oberhalb noch platz hat
9           -> Dame nach oben verschieben
10      Sonst
11          -> Backtracking
```

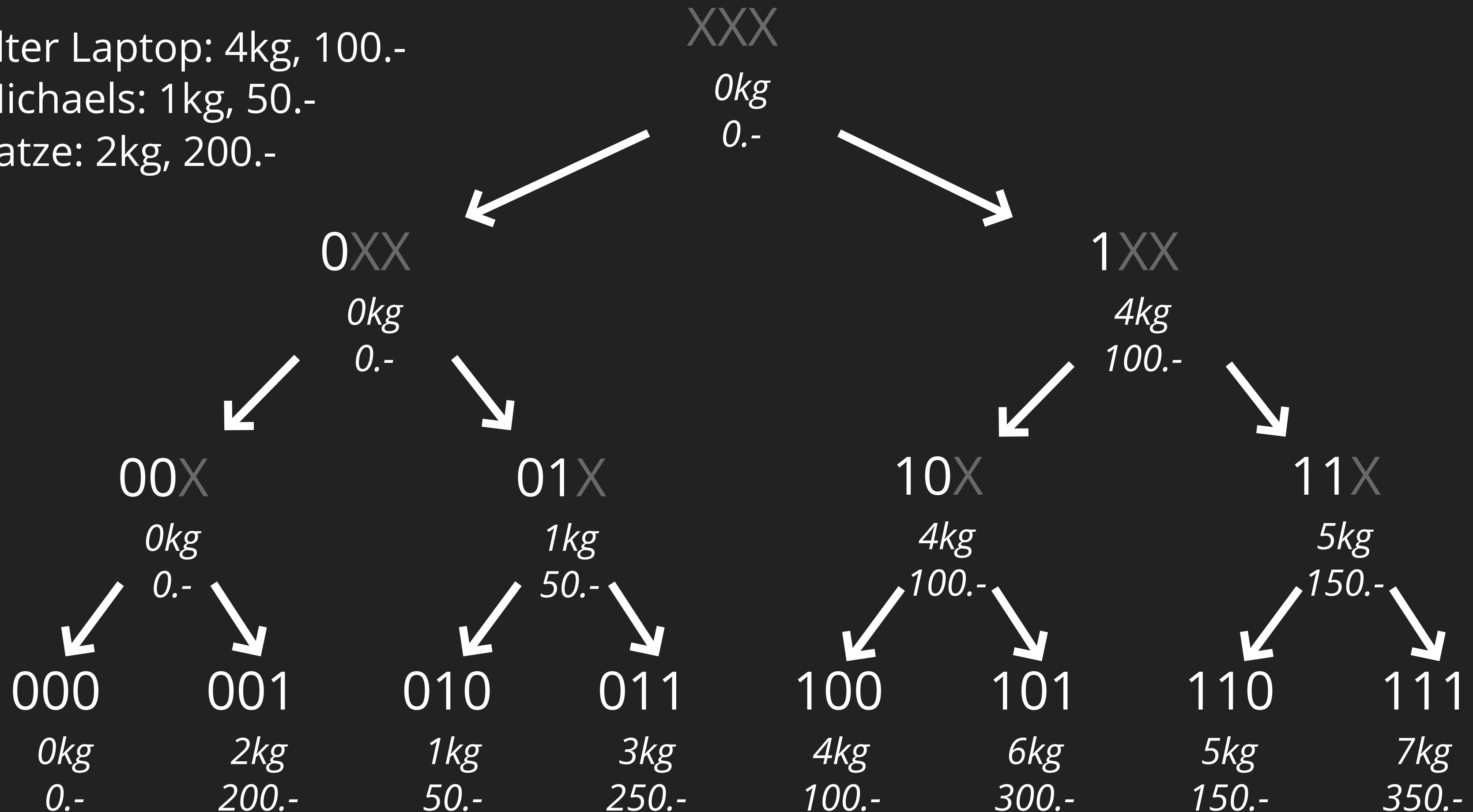


→ Lösung gefunden!

Rucksack Problem

- Ein Dieb ist in ein Haus eingebrochen
- Der Dieb kennt von jedem Gegenstand dessen Wert $v_i \geq 0$ sowie dessen Gewicht $g_i \geq 0$.
- Er hat eine Tasche, in welche ein Gewicht G passt.
- Welche der Gegenstände $x_1, ..., x_K$ soll er einpacken um den Wert der Beute zu maximieren.

Rucksack - Brute Force



Rucksack - Brute Force

Kapazität: 3kg

Alter Laptop: 4kg, 100.-
Michaels: 1kg, 50.-
Katze: 2kg, 200.-

Beste Lösung:

Michaels & Katze mit Wert 250.-

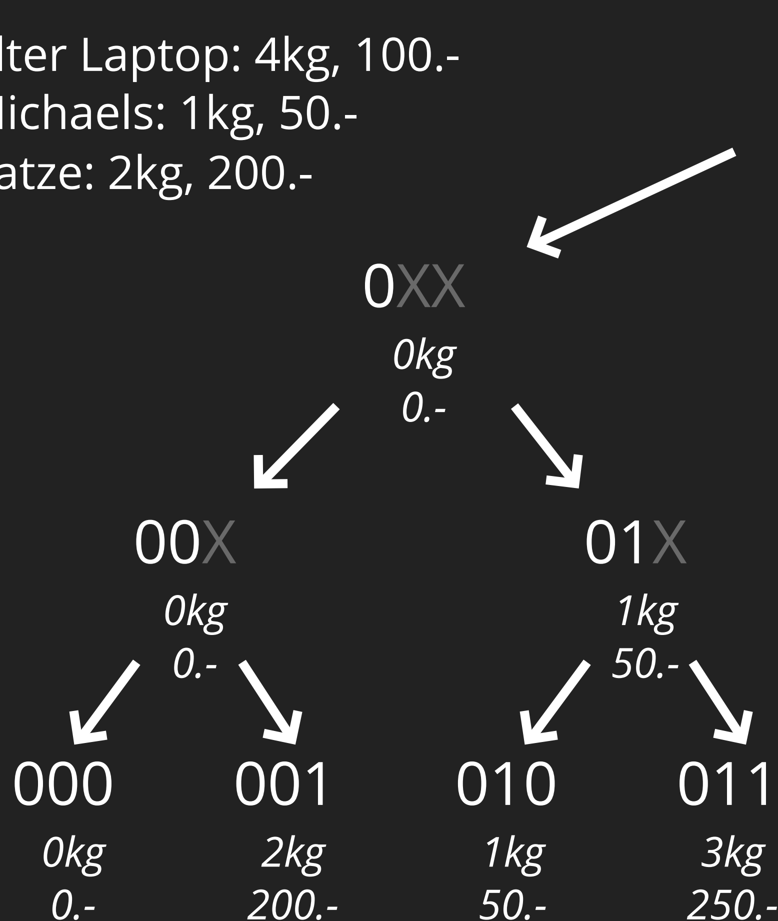
Zu schwer

100
4kg
101
300.-
110
150.-
7kg

Rucksack - Backtracking

Kapazität: 3kg

Alter Laptop: 4kg, 100.-
Michaels: 1kg, 50.-
Katze: 2kg, 200.-



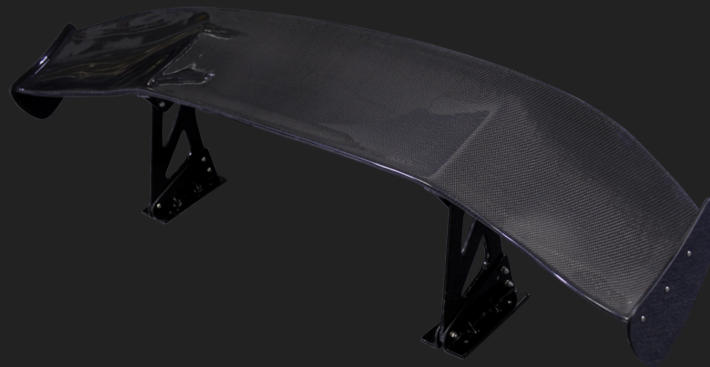
1XX
100.-

Zu schwer

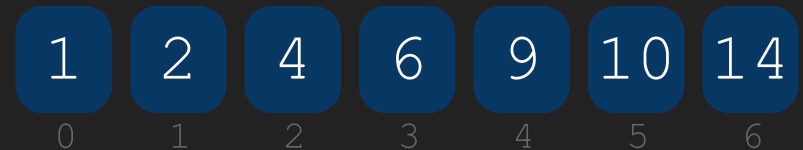
Viel Arbeit erspart

K!

Vorbesprechung



Binäre Suche Textaufgaben



Zeichnet "Entscheidungsbaum" für Binäre Suche.

Beispiel: Entscheidungsbaum für 9

Binäre Suche Programmieren

- Gegeben wird ein Interface für einen Binären Suchbaum mit zwei generischen Typen:
  - Key extends Comparable<Key>
    - Einen "Key", also das Element nach welchem ihr sortieren müsst.
    - Keys könnt Ihr mit key1.compareTo(key2) vergleichen
      - 0 $\rightarrow$ key1 = key2
      - negativ $\rightarrow$ key1 < key2
      - positiv $\rightarrow$ key1 > key2
  - Value
    - Ein "Value", das ist der Wert welcher ihr nach der Suche zurückgeben müsst

Rucksackproblem und Backtracking
Textaufgaben

- Gibt es beim Rucksackproblem immer nur eine Lösung?
- Falls Ja $\rightarrow$ Beweis
- Falls Nein $\rightarrow$ Gegenbeispiel

Rucksackproblem und Backtracking Programmieren

- Klasse Selection:
  - Effizientere Art Booleans zu speichern:
    - anstatt [true, false, true, true, false, false]
    - bits = 0b101100 (= 44)
    - Bits werden hier in einem Integer gespeichert
  - `Selection s = new Selection(size, bits)`
    - $\rightarrow$ Neue Selektion
  - `s.set(index,value)`
    - $\rightarrow$ Bestimmtes Bit auf *value* (true/false) setzen
  - s.sum(werteliste)
    - $\rightarrow$ bestimme den Wert eine Selektion aufgrund von einer Werteliste (ArrayList<Integer>)

Rucksackproblem und Backtracking Programmieren

- Selection Beispiel:

```
1  ArrayList<Integer> values = new ArrayList(4);
2  values.add(10);
3  values.add(20);
4  values.add(30);
5  values.add(40);
6
7  // Initialize selection as 0b0101
8  Selection selection = new Selection(values.size(), 5);
9  // 0*40 + 1*30 + 0*20 + 1*10 = 40
10 System.out.println(selection.sum(values));      → 40
11
12 // Change selection to 0b1101
13 selection.set(3, true);
14 // 1*40 + 1*30 + 0*20 + 1*10 = 80
15 System.out.println(selection.sum(values));      → 80
16
17 // Change selection to 0b1001
18 selection.set(2, false);
19 // 1*40 + 0*30 + 0*20 + 1*10 = 50
20 System.out.println(selection.sum(values));      → 50
```

Rucksackproblem und Backtracking

Programmieren

- Brute-Fore
  - Probiert alle Selections von 0 bis $2^{values.size()}$ aus und findet die beste.
- Backtracking
  - Vom Interface gegeben:
    ```
    public Selection findBest(ArrayList<Integer> values,
    ArrayList<Integer> weights, int maxWeight);
    ```
  - Helferfunktion:
    ```
    public Selection findBest(ArrayList<Integer>
    values,ArrayList<Integer> weights,int maxWeight,
    int ptr, Selection s);
    ```
    - ptr $\rightarrow$ Pointer, äquivalent zu der Höhe im Baum
    - s $\rightarrow$ Slection, die bis zu diesem Pointer bereits gemacht wurde

Reversi [Teil 2]

1. Check Move
    - Gut um ein Gespür für das Gameboard zu bekommen
    - Gar nicht relevant für das Turnier
    - Besser ihr investiert mehr Zeit in andere Reversi Projekte
2. Greedy Player
    - Macht einen Spieler, welcher immer so spielt, dass er im nächsten Zug möglichst viele Steine drehen kann
    - Kopiert das Spielbrett und probiert die möglichen Züge auf einem "hypotetischen" Gameboard aus
    - ```
      GameBoard hypothetical = gb.clone();
      hypothetical.makeMove(...);
      ```
      ```

Viel Spass!