Outlook

- Exercise 7: Solution discussion
- Exercise 8: Overview (Binary Search, Backtracking, Reversi)
Solution Ex7.Q1 – Tic-Tac-Toe

MAX strategy

MAX

MIN

MAX

MIN

0

+1

-1

+1

-1

0
Solution Ex7.Q2 – IBinarySearchTreeUtils<T>

(a) Löschen: 15

(b) Löschen: 12

(c) Löschen: 20

(d) Resultierender Baum
Solution Ex7.Q2 – IBinarySearchTreeUtils<T>

Confusion because of Generics?
The type of T does not matter for us.

```java
public class TreeUtils<T> extends IBinarySearchTreeUtils<T> {
  //...

  public int height(BinarySearchTree<T> tree) {
    if (tree == null) return 0;
    return 1 + Math.max(height(tree.left), height(tree.right));
  }

  //...
}
```
public boolean isLeaf(BinarySearchTree<T> tree) {
    return (tree.left == null && tree.right == null);
}

public boolean hasOneChild(BinarySearchTree<T> tree) {
    return (tree.left != null) ^ (tree.right != null);
}

/* Post-order: L-R-W */
private ArrayList<T> postOrder(BinarySearchTree<T> tree, ArrayList<T> arrayList) {
    if (tree == null) return arrayList;
    arrayList = postOrder(tree.left, arrayList);
    arrayList = postOrder(tree.right, arrayList);
    arrayList.add(tree.thing);
    return arrayList;
}

public ArrayList<T> postOrder(BinarySearchTree<T> tree) {
    return postOrder(tree, new ArrayList<T>());
}
Solution Ex7.Q2 – IBinarySearchTreeUtils<T>

```java
public BinarySearchTree<T> insert(BinarySearchTree<T> tree, int key, T thing) {
    if (tree == null) {
        return new BinarySearchTree<T>(key, thing);
    }

    if (tree.key == key) {
        tree.thing = thing;
    } else if (key < tree.key) {
        tree.left = insert(tree.left, key, thing);
    } else {
        tree.right = insert(tree.right, key, thing);
    }
    return tree;
}

public T find(BinarySearchTree<T> tree, int key) {
    if (tree == null) return null;
    if (key == tree.key) {
        return tree.thing;
    }
    if (key < tree.key) {
        return find(tree.left, key);
    }
    return find(tree.right, key);
}
```
public BinarySearchTree<T> remove(BinarySearchTree<T> tree, int key) {
    if (tree == null) return null;
    if (tree.key == key) {
        if (isLeaf(tree)) {
            return null;
        }
        if (hasOneChild(tree)) {
            if (tree.left != null) {
                return tree.left;
            }
            return tree.right;
        }
        UnlinkSmallestResult<T> result = unlinkSmallest(tree.right);
        result.smallest.right = result.tree;
        result.smallest.left = tree.left;
        return result.smallest;
    }
    if (key < tree.key) {
        tree.left = remove(tree.left, key);
    } else {
        tree.right = remove(tree.right, key);
    }
    return tree;
}
private Random rand = new Random(); //outside nextMove!!!

public Coordinates nextMove( GameBoard gb ){
    Coordinates coord = null;

    ArrayList<Coordinates> validMoves =
        new ArrayList<Coordinates>( gb.getSize() * gb.getSize() );

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

    if( validMoves.isEmpty() ) return null;

    int randIndex = rand.nextInt( validMoves.size() );
    return validMoves.elementAt( randIndex );
}
Outlook

- Exercise 7: Solution discussion
- Exercise 8: Overview (Binary Search, Backtracking, Reversi)
Hints to Sheet 8

- Binary Search
- Knapsack problem and Backtracking
- Reversi (Part 2)
Hints to Ex8.Q1 – Binary Search

- Binary Search Algorithm (illustrated)

- Decision Tree

query: 16
sub-array-length: 1
return: null

query: 23
sub-array-length: 1
return: "23"
Hints to Ex8.Q1 – Binary Search

- Draw the decision tree and make some thoughts
  - Superposition, factors

- Implementation:
  - find(List<Unit<Key, Value>> haystack, Key needle)
  - setFactor(int factor)
    - Generalize the search → unbalanced search trees
  - getNumberOfCalls()
    - Benchmarking with various factors
    - Average # of recursive calls to various factors
Ex8.Q2 – Knapsack problem and Backtracking

x1 \ g1, w1
x2 \ g2, w2
x3 \ g3, w3
x4 \ g4, w4
x5 \ g5, w5
Hints Ex8.Q2

The general Knapsack problem

- $k$ items $x_1, ..., x_k$ and each has known value and weight
- Choice of items, such that total weight is not exceeded
- Optimization problem: Maximize the value of the chosen items

a) Theory
b) Bruteforce approach
c) Backtracking approach
d) Comparison of Bruteforce und Backtracking approaches
Ex8.Q2 – Subset

- How many different possibilities does our thief have?
  - $S =$ Set of items at our disposal
  - The thief can only take a subset home
  - The thief can also choose the empty subset $\emptyset$ (lazy thief) or the whole set $S$ (strong thief with big bag)!

- $\#\text{items} := \#\text{elements in the power set of } S$

- Example:
  - $S = \{x_1, x_2\}$, $|S| = K = 2$
  - 4 Subsets: $\emptyset, \{x_1\}, \{x_2\}, \{x_1, x_2\}$
Ex8.Q2 – Backtracking

- What does "Backtracking" mean?
  - Principle: "trial and error"

- Example: Looking for a maze exit
  - Decide upon a direction
  - Continue in this direction
  - If eventually unsuccessful
    - Return and choose another direction
  - If eventually successful
    - Done…

In case all directions were tried → keep going back.
Ex8.Q2 – A simple strategy for the thief

- **Algorithm Implementation**

\[
W = \max \left( \sum_{i=0}^{K-1} b_i w_i \right), \quad \sum_{i=0}^{K-1} b_i g_i \leq G
\]

\[
mit: \quad b_i = \begin{cases} 
0, & \text{wenn Gegenstand } x_i \text{ nicht } "\text{eingepackt}" \\
1, & \text{wenn Gegenstand } x_i \text{ } "\text{eingepackt}" 
\end{cases}
\]

- **Reminder**
  - A set S with |S|=k contains $2^k$ subsets
Ex8.Q2 – A simple strategy for the thief

- Apply the implementation in pseudo-code

1. Initialization

2. Take the next configuration (how exactly?)

3. Compute the overall weight
   
   if (overall weight < W)
   
   Compute overall value
   
   if (new overall value > overall value of current optimal solution)
   
   Current configuration becomes optimal solution

4. If more configurations remain ,
   
   Go to point 2
   
   else
   
   Computation done
Ex8.Q2.a – A simple strategy for the thief

- Does the thief’s strategy always yield the optimal results?
  - Yes/No
  - Why?...

- Is there always exactly one optimal solution?
  - Yes/No
  - Why?...
Ex8.Q2b,c – Bit value

- Configuration as a sequence of bits: class Selection
- The bit value describes the value of single bits, which depends on their position in the overall binary form of a number.

**MSB - Most Significant Bit/Byte**
- The most significant bit (MSB, also called the high-order bit) is the bit position in a binary number having the greatest value.

**LSB - Least Significant Bit/Byte**
- The least significant bit (LSB) is the bit position in a binary integer having the smallest value.
Ex8.Q2b,c – Hints for the implementation

- `class Selection` is well documented
- Beware: If you increase the configuration (if you put a new item in the bag, A1c), the new status has to be updated
- Example of selections for $S$

\[ M = \{x_1, x_2, x_3, x_4\}, \quad |M| = K = 4 \]

\[ \text{Teilmengen: } \emptyset, \{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_1, x_2\}, \{x_1, x_3\}, \ldots \]

\[ (b_1, b_2, b_3, b_4) = (0, 0, 0, 0) \]

\[ (b_1, b_2, b_3, b_4) = (1, 0, 1, 0) \]

\[ (b_1, b_2, b_3, b_4) = (0, 1, 0, 0) \]
Ex8.Q2b,c – Hints for the implementation

Bruteforce approach

```java
public Selection findBest(ArrayList<Integer> values,
                         ArrayList<Integer> weights,
                         int maxWeight)
{
    ...

    int last = java.Math.pow(2, values.size()); //Number of subsets
    for( int i = 0; i < last; i++ )
    {
        new Selection(values.size(), i); //Selection bit field with value i
    }
    ...
}
```
Ex8.Q2b,c – Hints for the implementation

- Backtracking approach:
  - **FindResult** Class (Selection and Value together)
  - Recursive method:
    ```
    FindResult fr = find(currSelection, currWeight, values, weights, maxWeight);
    ```
  - **End condition**: `selection.size()==values.size(); //all things considered`
  - In the method: two directions to continue
    ```
    //Leave item
    Selection without = new Selection(...); //Increase by one, set bit to 0
    //then continue down the tree

    //Check whether weight is OK, then take item
    ...
    Selection with = new Selection(...); //Increase by one, set bit to 1
    //then continue down the tree
    ```
Hints to Ex8.Q3 – Reversi (Part 2)

HumanPlayer

```
nextMove()
Wait for input from The command line
```

RandomPlayer

```
nextMove()
Random selection (but valid move!)
Next move
```

GreedyPlayer

```
nextMove()
Select the next move using a simple, None-recursive Evaluation function
```

Download

Excercise 7

Excercise 8
Hints: Reversi (Part 2)

a. Implementing `ICheckMove` without Framework-Function.
   Ideas?
   Learn from GameBoard class. What methods available what could be useful?

b. Implement a player that selects the best move among all possible moves
   
   *Best move:* Move, after it's execution one owns max. more stones than the opponent: «Depth = 1»
   
   **Given:** Game Tree is not needed!

   *Determining the best move:* Copy Board (clone), run the move, count...
Hints Ex8.Q3a – checkMove()

```java
boolean checkMove(GameBoard gb, int player, Coordinates c) {

    //Check all directions
    
    //Unless at least one direction is valid
    //...

    //GameBoard.checkMove is not allowed to be used!

}
```
Hints Ex8.Q3a – greedyPlayer()

- Simple computer opponents
  - Move selection: Best first
  - Search depth: 1 (my move)
  - Evaluation Function: The difference of stone numbers after the move

- Tips
  - A GameBoard can be copied with gb.clone()
  - You are allowed to use GameBoard.checkMove() here