Solution U8.A1a,b – Binary Search

mi = (re-li)/2 + li;
Solution U8.A1c – Binary Search (with factor 3)

Conclusion: when I search for small numbers that are present in the array, this method is faster. But worse in the general case (lower tree)
Solution U8.A1d – Binary Search

- Always copying sub-arrays is not a good idea
  → Better to create a separate method that takes additional two int values: `begin` and `end` as parameters:

```java
public Value find(ArrayList<Unit<Key, Value>> haystack, Key needle)
{
    return findRecursive(haystack, needle, 0, haystack.size());
}

private Value findRecursive( ArrayList<Unit<Key, Value>> haystack,
                              Key needle, int begin, int end)
{
    ...
}
```

Eclipse DEMO
Solution U8.A2a – Simple Thief Strategy

- Does the simple thief strategy always deliver the optimal solution?
  - Yes, because it is going through all configurations

- Is there always one single optimal solution?
  - No, there can be multiple optimal solutions
  - Proof by counter-example:
    - Item <Weight, Value>
      - \[ <1,1>, <2,1>, <3,2> \]
    - \( W_{\text{max}} = 3 \)
    - Solution 1 = [true, true, false]
    - Solution 2 = [false, false, true]
public Selection findBest( ArrayList<Integer> values, 
                        ArrayList<Integer> weights, int maxWeight) 
{
    int last = (int) Math.pow(2, values.size());
    int bestsum = 0;
    Selection bestsel = new Selection(values.size());
    Selection sel = new Selection(values.size());

    for(int i = 0; i < last; i++)
    {
        sel.setBits(i);

        if( sel.sum(weights) <= maxWeight ){
            if( sel.sum(values) > bestsum ){
                bestsum = sel.sum(values);
                bestsel.setBits(i);
            }
        }
    }

    return bestsel;
}
Solution U8.A2c – Backtracking.java

```java
private class FindResult {
    public int value;
    public Selection selection;

    FindResult(int val, Selection sel) {
        value = val;
        selection = sel;
    }
}

public Selection findBest(ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
    if (values.size() != weights.size())
        throw new IllegalArgumentException("sizes of values and weights vectors are not equal");

    // give initial selection and weight 0
    FindResult result = find(new Selection(0), 0, values, weights, maxWeight);

    return result.selection;
}
```
// find(new Selection(0), 0, values, weights, maxWeight);
private FindResult find(Selection selection, int weight, ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
    final int depth = selection.size();
    if (depth == values.size())
        return new FindResult(selection.sum(values), selection);

    Selection without = new Selection(depth + 1, selection.bits());
    without.set(depth, false); // set left-most element to 0
    FindResult resultWithout = // recursion without current item
        find(without, weight, values, weights, maxWeight);

    if (weight + weights.get(depth) <= maxWeight) {
        Selection with = new Selection(depth + 1, selection.bits());
        with.set(depth, true); // set left-most element to 1
        FindResult resultWith = // recursion with current item
            find(with, weight + weights.get(depth), values, weights, maxWeight);

        if (resultWith.value > resultWithout.value)
            return resultWith;
    }
    return resultWithout;
}
Solution U8.A2d – Runtime

- Measurement (2GHz Intel)
  - Brute Force: ~4.0s
  - Backtracking: ~0.08s
- Backtracking requires about 2% of the time spent on the Brute Force approach
  - $W$ accounts for about 5x the average weight
  - On average only 5 items out of 20 are chosen (20%)
  - A lot of premature aborts
Solution U8.A3a – checkMove: how to

Invalid!

Valid?
Solution U8.A3a – checkMove

- boolean checkMove(GameBoard ..., Coordinates c)
  - Field must be free!
  - Check all directions
  - Until at least one direction is “valid”…
    
    for( int i = -1; i <= 1; ++i )
    for( int j = -1; j <= 1; ++j )
    if( i != 0 || j != 0 )
      if( checkDirection(gb, player, c, new Coordinates(i, j) )
        return true; //would be a possible move
    return false; //not a possible move
  
- follow( gb, player, pos, dir )
  - sum := #number of opponent (player) stones in your direction
  - Gain:= sum > 0 && last stone belongs to the player
  - Last stone is located at coordinates:
    (pos.x+sum*dir.x, pos.y+sum*dir.y)

Similar Solution: Eclipse DEMO
Solution U8.A3a – Reversi Tips & Tricks

- checkMove is declared in reversi.GameBoard
  - boolean reversi.GameBoard.checkMove(int, Coordinates)

- Other useful methods
  - boolean reversi.GameBoard.isMoveAvailable(int)
  - boolean reversi.GameBoard.validCoordinates(Coordinates)
  - int reversi.GameBoard.countStones(int)
  - int reversi.Utils.other(int)
  - …

- Summary: JavaDoc is cool…
Solution U8.A3b – Greedy Player

- Player-AI
  - For all possible moves
    - Simulate the move on a copy of the current board
    - Review the resulting situation
    - Save move and gain(evaluation) in a list
  - Sort the list / search the maximum / maxima
  - Select a random / the maximum move
  - Evaluation function (so far)
    - Ratio of own stones vs. opponent’s stones

- Data Structures
  - MoveInfo: stores evaluated move information
    - Coordinates and Review
  - List<MoveInfo> (e.g. an ArrayList)
    - More efficient than a Vector
    - As practical as an Array

Similar solution without a list: Eclipse DEMO
HINTS ON U9

A1 – Game Tree Evaluation
A2 – Reversi (Part 3)
Hints on U9.A1

Game theory/ Game tree analysis

a) A little bit of theory

b) Minimax-Algorithm

c) Optimal Strategy for MAX-Player

d) Alpha/Beta-Algorithm
U9.A1 – Game Theory

- Components of a game tree
  - Root → Beginning (state before any move)
  - Node → Possible state of the game
  - Edge → Move
  - Leaf → End of the game (final state)
U9.A1b – Minimax-Algorithm

- Algorithm to determine the optimal game strategy for zero-sum (-1,0,1)
- Secures the highest possible gain when the opponent plays to win
- For zero-sum games, better algorithms exist

\[\begin{align*}
&\text{MAX} \\
&\text{MIN} \\
&\text{MAX} \\
&\text{MIN} \\
&\text{MIN}
\end{align*}\]
U9.A1c – Strategy for Max

- Strategy

  - A strategy (for Max) is a graph, which derives from a game tree, where all edges are stricken out and only edges coming out of Max nodes are kept.

  - The best move for MAX given all possible moves for MIN...

  By proxy a set of nodes/edges, not just a path!
U9.A1d – The α-β-Algorithm

- The α-β-Algorithm
  - Prunes the game tree, delivers the MiniMax value of the root in the same way as the MiniMax algorithm
  - The MiniMax algorithm evaluates the whole search tree. In this case, nodes that don't influence the outcome (choice of the branch at the root) are also evaluated.
  - The Alpha-Beta search ignores those nodes.

- α will be the value of the best possible move you can make, that you have computed so far.
- β will be the value of the best possible move your opponent can make, that you have computed so far.
- If at any time, α >= β, then your opponent's best move can force a worse position than your best move so far, and so there is no need to further evaluate this move.
U9.A1d – The α-β-Algorithm

alpha-beta(player, board, alpha, beta)
    if(game over in current board position)
        return winner

    children = all legal moves for player from this board
    if(max's turn)
        for each child
            score = alpha-beta(other player, child, alpha, beta)
            if score > alpha then alpha = score (we have found a better best move)
            if alpha >= beta then return alpha (cut off)
        return alpha (this is our best move)
    else (min's turn)
        for each child
            score = alpha-beta(other player, child, alpha, beta)
            if score < beta then beta = score (opponent has found a better worse move)
            if alpha >= beta then return beta (cut off)
        return beta (this is the opponent's best move)

https://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
U9.A1d – The α-β-Algorithm

- Remember the operation tree from lecture 9
  - MAX nodes are OR nodes
  - MIN nodes are AND nodes
  - Leaves can only take 0 and 1 as values
Compute the value of the root in the following game tree using the $\alpha$-$\beta$ method. Initially, $\alpha=\infty$ and $\beta=\infty$.

$\beta$-cut: MIN has already nodes with $\beta < 7 \rightarrow$ abort before end!
U9.A1d – The α-β-Algorithm

Online example:
(user: i2 password: i22012)

Online JAVA Applet:
http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
# Hints U9.A2 – Reversi (Teil 3)

<table>
<thead>
<tr>
<th>Player</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HumanPlayer</td>
<td>nextMove()</td>
<td>Waits for entry from command line</td>
</tr>
<tr>
<td>RandomPlayer</td>
<td>nextMove()</td>
<td>Chooses a random (but valid) next move</td>
</tr>
<tr>
<td>GreedyPlayer</td>
<td>nextMove()</td>
<td>Chooses the next move by means of an easy and non-recursive evaluation function</td>
</tr>
<tr>
<td>MiniMaxPlayer</td>
<td>nextMove()</td>
<td>Chooses the next move by means of a Minimax analysis through a new evaluation function</td>
</tr>
</tbody>
</table>

Download | Übung 7 | Übung 8 | Übung 9
Evaluation of the game trees
- Implement a method that evaluates the game tree through Minimax until the depth $d$ is reached (alternating Max and Min)
- Depth of research can be configured
- Recursive approach
  - Build the game tree recursively
  - Evaluate the state at depth $d$
  - Minimax on the obtained evaluation yields the strategy
  - Incorporate all special cases (e.g. pass)!

No time-out yet
**U9.A2b – timeLimit**

- **Time limit per move:**
  - Your `nextMove()` method should return a valid move before the time-out in milliseconds
  - Plan a time buffer (in the order of 10 ms) so that cancellation and delivery of result do not happen straight away!
    - e.g., `long timeout = System.currentTimeMillis() + timeLimit - 10;`
  - Possible approach: throw a Timeout exception
U9.A2c – Evaluation function (I) (optional)

- Improve game state evaluation

- You can find a “source of inspiration” in the following article:
  - „The Development of a World Class Othello Program“, Kai-Fu Lee and Sanjoy Mahajan, 1990

- To download from the Reversi website
  - username: i2bib
  - password: reversi

- Artificial Intelligence: A Modern Approach
U9.A2c – Evaluation function (II)

- Possible „evaluation functions“
  - How many counters are flipped?
  - Where are the flipped counters located (center/border)?
  - ....

- A few pieces of advice concerning the tournament
  - Start with writing the idea for the evaluation function in the pseudo-code
  - Keep developing the pseudo-code
  - The pseudo-code yields hints about how the information about the next move should be computed
  - Keep implementing the different versions of the pseudo-code for the tournament player
...have fun!