Overview

- Debriefing Exercise 8
- Briefing Exercise 9
U8.A1 Binary Search (a & b)

mi = (re-li)/2 + li;
Conclusion:
Small numbers? Faster!
Large numbers? Slower?

Overall?
Better to split in half!
U8.A1 Binary Search: How to implement?

The interface defines the method like this:

```java
public Value find(ArrayList<Unit<Key, Value>> haystack, Key needle)
{
}
```

But, we can do better:

```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)
{
    ...
}
```
U8.A1 Binary Search: find

```java
private Value findRec(List<Unit<Key, Value>> haystack, Key needle, int begin, int end) {
    numberOfCalls++; // IMeasure
    if (begin == end) {
        return null;
    }
    int middle = begin + (end - begin) / factor;
    Unit<Key, Value> middleThing = haystack.get(middle);
    int match = needle.compareTo(middleThing.key);
    if (match == 0) {
        return middleThing.value;
    } else if (match < 0) {
        return findRec(haystack, needle, begin, middle);
    } else {
        return findRec(haystack, needle, middle + 1, end);
    }
}
```

Find division point
Go left on the tree
Or right
U8.A2 Knapsack problem

- Does the simple strategy deliver the optimal solution?
  - Yes, why?
  - We go through all the possible solutions
  - However, this is quite unlikely in most scenarios (time, memory, etc.)

- Is there only one optimal solution?
  - No
  - How do you prove this?
  - By counter-example
    - Item <Weight, Value>
      - [ <1,1>, <2,1>, <3,2> ]
    - \( W_{\text{max}} = 3 \)
    - Selection_1 = [1, 1, 0]
    - Selection_2 = [0, 0, 1]
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;
}
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;
}
U8.A2 Backtracking (2)

// 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);
    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);
        FindResult resultWith = // recursion with current item
            find(with, weight + weights.get(depth), values, weights, maxWeight);

        if(resultWith.value > resultWithout.value)
            return resultWith;
    }
    return resultWithout;
}
U8.A2 Some conclusions

- Backtracking vs. Brute force
  - Backtracking is much faster
  - Backtracking: cuts early many solutions that are not possible
  - Both find the optimal solution
U8.A3 Reversi [Part II] – Some useful commands

- 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! Use it!
  https://www.vs.inf.ethz.ch/edu/I2/reversi/javadoc/
 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 )
 Follow this new direction
 If it’s a stone from the other player, call follow again on the same direction
 If it’s my stone, we stop. The move is valid!
public boolean checkMove(GameBoard gb, int player, Coordinates coord) {
    try {
        if (gb.getOccupation(coord) != GameBoard.EMPTY) return false;
    } catch (OutOfBoundsException e) {
        return false;
    }

    for (int x = -1; x <= 1; x++) {
        for (int y = -1; y <= 1; y++) {
            if (x == 0 && y == 0) continue;
            if (checkDirection(gb, player, coord, x, y)) return true;
        }
    }

    return false;
}
U8.A3 checkMove - checkDirection

```java
private boolean checkDirection(GameBoard gb, int player, Coordinates coord, int x, int y)
{
    Coordinates c = new Coordinates(coord.getRow() + x, coord.getCol() + y);
    try {
        if (gb.getOccupation(c) != Utils.other(player)) return false;
    } catch (OutOfBoundsException e) {
        return false;
    }

    return follow(gb, Utils.other(player), c, x, y);
}
```

follow: move along a certain direction, given the rules from the previous slides
private boolean follow(GameBoard gb, int player, Coordinates coord, int x, int y) {
    Coordinates c = new Coordinates(coord.getRow() + x, coord.getCol() + y);

    int occupation;
    try {
        occupation = gb.getOccupation(c);
    } catch (OutOfBoundsException e) {
        return false;
    }

    if (occupation == player) return follow(gb, player, c, x, y);
    if (occupation == Utils.other(player)) return true;
    return false;
}
U8.A3 Reversi – Greedy player

- For all possible moves

- Simulate the move
  - Make a copy of the gameboard
  - Make the move
  - Evaluate your move (how?)

- Choose the best move
```java
public Coordinates nextMove(GameBoard gb) {
    Coordinates bestMove = null;
    int bestValue = Integer.MIN_VALUE; // minEval(gb);

    for (int x = 1; x <= gb.getSize(); x++) {
        for (int y = 1; y <= gb.getSize(); y++) {
            Coordinates c = new Coordinates(x, y);
            if (gb.checkMove(myColor, c)) {
                GameBoard hypotheticalBoard = gb.clone();
                hypotheticalBoard.checkMove(myColor, c);
                hypotheticalBoard.makeMove(myColor, c);
                int value = eval(hypotheticalBoard);
                if (value > bestValue) {
                    bestValue = value;
                    bestMove = c;
                }
            }
        }
    }

    return bestMove;
}

/**
 * Evaluate a situation on a board
 * @param gb the situation in question
 * @return the value of the situation. Larger means better.
 */
private int eval(GameBoard gb) {
    return gb.countStones(myColor) - gb.countStones(Utills.other(myColor));
}
```
Overview

- Debriefing Exercise 8
- Briefing Exercise 9
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.A1 Minmax Algorithm

MAX

MIN

0

+1

0

-1

+1

-1

0

-1

+1

0

0
Beta-Cut: MAX will give a value of at least 20. It does not matter what is in the rest of the tree as the MIN player will select 10.
Alpha-Cut: MIN will choose 8 or less. Since on the other branch we have a 10, MAX will always choose the larger value.
<table>
<thead>
<tr>
<th>Player</th>
<th>nextMove()</th>
<th>Function 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>MinMaxPlayer</td>
<td>nextMove()</td>
<td>Chooses the next move by means of a Minimax analysis through a new evaluation function</td>
</tr>
</tbody>
</table>
U9.A2a Reversi MinMax Player

- Evaluation of the game trees

- Implement a method that evaluates the game tree through Minmax 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 $d$
  - Minmax on the obtained evaluation yields the strategy
  - Incorporate all special cases (e.g. pass)!
U9.A2b Reversi Timeout

- The timeout value is an input parameter (initialize method)

- 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!

- Possible approach: throw an out-of-time 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) (optional)

- 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!