Overview

- Debriefing Exercise 10
- Briefing Exercise 11
U10.A1a Merge Sort
U10.1b Merge Sort – Divide and Conquer

```java
public ArrayList<T> sort(ArrayList<T> items) {
    return sortRec(items, 0, items.size());
}
```

```java
private ArrayList<T> sortRec(ArrayList<T> items, int begin, int end) {
    if (begin == end) {
        return new ArrayList<T>();
    }
    if (begin + 1 == end) {
        ArrayList<T> result = new ArrayList<T>();
        result.add(items.get(begin));
        return result;
    }

    int middle = begin + (end-begin) / 2;
    ArrayList<T> left = sortRec(items, begin, middle);
    ArrayList<T> right = sortRec(items, middle, end);
    return merge(left, right);
}
```
U10.1b Merging two sorted arrays

```java
int leftIdx = 0;
int rightIdx = 0;
ArrayList<T> sorted = new ArrayList<T>(end-begin);
while(true) {
    if (leftIdx == left.size()) {
        sorted.addAll(right.subList(rightIdx, right.size()));
        return sorted;
    }
    if (rightIdx == right.size()) {
        sorted.addAll(left.subList(leftIdx, left.size()));
        return sorted;
    }
    if (left.get(leftIdx).compareTo(right.get(rightIdx)) < 0) {
        sorted.add(left.get(leftIdx));
        leftIdx+=1;
    } else {
        sorted.add(right.get(rightIdx));
        rightIdx+=1;
    }
}
```

If all the elements from the left sublist have been processed, add all the remaining elements from the right sublist.

The same as the above if the other case holds.

Otherwise choose the smallest between left and right and add the item to the sorted list.
U10.1c,d Runtime analysis

![Graph showing runtime analysis]

- $f(n)$ vs. $n$
- $n \times \log(n)$ and Measurement lines on a log-log scale.

**Messung**
U10.A2 Towers of Hanoi

Summary:
Number of discs (n): 4
Number of steps \((2^n - 1)\): 15
Not used towers:
3 2 1 3 2 1 3 2 1 3 2 1 3 2 1
How does it look like with 5 discs?
19. 

20. 

21. 

22. 

23. 

24. 

1 \rightarrow 3 \quad 2 

2 \rightarrow 1 \quad 3 

3 \rightarrow 2 \quad 1 

3 \rightarrow 1 \quad 2 

2 \rightarrow 1 \quad 3 

2 \rightarrow 3 \quad 1
Summary:

Number of discs (n): 5

Number of steps \((2^n-1)\): 31

Sequence of not used towers:

\[2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \]
U10.A2 Towers of Hanoi (n discs)

Summary:

5 Discs (31 Steps):
2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2

4 Discs (15 Steps):
3 2 1 3 2 1 3 2 1 3 2 1 3 2 1

3 Discs (7 Steps):
2 3 1 2 3 1 2
U10.A2 Towers of Hanoi - Pseudocode

\[
\begin{align*}
moves &= 2^{n-1}; \\
counter &= 0; \\
\text{if } n \text{ even then} & \text{ while } (counter < moves) \\
& \quad \text{make possible move between tower 1 and tower 2} \\
& \quad \text{make possible move between tower 1 and tower 3} \\
& \quad \text{make possible move between tower 2 and tower 3} \\
& \quad \text{increment counter by 3 units} \\
\text{else } [n \text{ is odd}] & \text{ while } (counter < moves-1) \\
& \quad \text{make possible move between tower 1 and tower 3} \\
& \quad \text{make possible move between tower 1 and tower 2} \\
& \quad \text{make possible move between tower 3 and tower 2} \\
& \quad \text{increment counter by 3 units} \\
& \quad \text{make available move between tower 1 and tower 3}
\end{align*}
\]

make possible move $\Rightarrow$ there is always only one possible way (the smaller disc, or the only disc)
U10.A3 Reversi [Part 4]

- Implement an evaluation function that operates on the α-β-method, but the final outcome is as the pure min-max method of the last exercise series.

- The simplest way to do it is by:
  - 2 functions: min and max, which are alternately called
  - One changes the Beta-bound and the other changes the Alpha-bound
BestMove max (int maxDepth, long timeout, GameBoard gb,
   int depth, int alpha, int beta) throws Timeout
   if (System.currentTimeMillis() > timeout) throw new Timeout();
   if (depth==maxDepth) return new BestMove(eval(gb),null,true);

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

   for (int x = 1; x <= gb.getSize(); x++)
      for (int y = 1; y <= gb.getSize(); y++) {
         Coordinates coord = new Coordinates(x, y);
         if (gb.checkMove(myColor, coord))
            availableMoves.add(coord);
      }

   if (availableMoves.isEmpty())
      if (gb.isMoveAvailable(otherColor)) {
         BestMove result =
            min(maxDepth, timeout, gb, depth+1, alpha, beta);
         return new BestMove(result.value, null);
      } else
         return new BestMove(finalResult(gb), null);

   [...]

Recursive call when MAX has no possible move
BestMove max (int maxDepth, long timeout, GameBoard gb,  
   int depth, int alpha, int beta) throws Timeout

[...]
boolean cut = false;
Coordinates bestCoord = null;
for (Coordinates coord : availableMoves) {
   GameBoard hypothetical = gb.clone();
   hypothetical.checkMove(myColor, coord);
   hypothetical.makeMove(myColor, coord);
   BestMove result = min(maxDepth, timeout, hypothetical,  
      depth+1, alpha, beta);
   if (result.value > alpha) {
      alpha = result.value; // update the value of alpha
      bestCoord = coord;
   }
   if (alpha >= beta) { // pruning: ignore siblings of same node!
      return new BestMove(alpha, null);
   }
}
return new BestMove(alpha, bestCoord);
Reversi Tournament

- Wednesday, May 30th, 2018, 12:30, CABinett (Stuz2).

Submission:
- **Deadline:** Wednesday, May 23rd, 2018, 23:59 (Zürich Time)
- You can work alone or in groups of two
- Upload your player on the online platform
- Mark it as a tournament player

https://www.vs.inf.ethz.ch/edu/I2/reversi/

- **16.05.2018:** Ab sofort gibt es auch ein Eclipse und ein IntelliJ Projekt was die *.jar, javadoc, HumanPlayer und AlphaBetaPlayer sowie die nötigen configuration fürs Ausführen und *.jar erstellen enthalten: Eclipse Project und IntelliJ Project
Reversi tournament

- Mi, 30.05.2018, 12:30
- Room: Stuz2 (CABinett)
- Cool prizes: Everybody who reaches the quarterfinals(!)
- Catering
Reversi
Overview

- Debriefing Exercise 10
- Briefing Exercise 11
Algorithm Complexity

- Problem scope n
  - Often: Number of input values

- The complexity of a problem
  - Minimum cost, that the algorithm can be solved with.
  - Often the cost of an algorithm is not only determined by the problem scope n, but also depends on the input value or the order of the input values.
  - Then the following cases can be specified:
    - Best cost („best case“)
    - Middle cost („average case“)
    - Worst cost („worst case“)
U11 Algorithm cost

- Ex: bubble sort

```haskell
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
        until not swapped
    end procedure
```

First Pass:

(51428) → (15428),
(15428) → (14528),
(14528) → (14258),
(14258) → (14258),

Second Pass:

(14258) → (14258),
(14258) → (12458),
(12458) → (12458),
(12458) → (12458),

Third Pass:

(12458) → (12458),
(12458) → (12458),
(12458) → (12458),
(12458) → (12458)
3 Theoretical questions

- Describe how one could use BSTs for sorting

- How about inserting pre-sorted elements into a BST? Will this harm/benefit the BST? Three cases: sorted asc., sorted desc., random

- Complexity of sorting using BSTs in the best, worst, and average case, in Landau-Notation
U11.A2 Complexity analysis

- Analyze the code fragments in terms of their runtime complexity and enter the result in O notation

- Valid solution
  - Calculation steps and resulting O notation!
U11.A2 Complexity analysis

// Fragment 1
for (int i=0; i<n; i++)
    a++;

// Fragment 2
for (int i=0; i<2n; i++) a++;
for (int j=0; j<n; j++) a++;

// Fragment 3
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a++;

// Fragment 4
for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
        a++;

// Fragment 5
while(n >=1 )
    n = n/2;

// Fragment 6
for (int i=0; i<n; i++)
    for (int j=0; j<n*n; j++)
        for (int k=0; k<j; k++)
            a++;
U11.A2 Example

// Fragment 1

for (int i=0; i<n; i++)
a++;

c_{0} + c_{1}n \sim O(n)
U11.A3 Complexity (1)

\[ t_{op} = \frac{1}{3} t_{op} \]

- \( t_{op} \): Time per Operation
- \( M' \): Input Size
- \( T_{tot}' \): Total run time
U11.A3 Complexity (2)

What is the new input size (M), if time per operation ($t_{op}$) is 3x faster?

<table>
<thead>
<tr>
<th>O(...)</th>
<th>$T_{tot}$</th>
<th>$T'_{tot}$</th>
<th>$T_{tot} = T'_{tot}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$T_{tot} = t_{op} \times M_1$</td>
<td>$T'<em>{tot} = t'</em>{op} \times M'_1$</td>
<td>$t'<em>{op} \times M'<em>1 = t</em>{op} \times M_1 \Rightarrow \frac{1}{3} t</em>{op} \times M'<em>1 = t</em>{op} \times M_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\Rightarrow M'_1 = 3M_1$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(\log_2 n)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
U11.A4 A knight on a chessboard
U11.A4a Reachable fields

- Find the set of fields:
  - Reachable by \( n \) moves,
  - Given: start position
U11.A4a Knight’s tour

- Class Position
  - \( p = \text{new Position}(0,0); \)
  - Position next = \( p.\text{add}(\text{new Position}(\text{offX}, \text{offY})); \)
  - Implement compareTo, equals, etc.

- Method getReachableSet
  - \( \text{ArrayList<Position>} \) getReachableSet(Position \( p \), int \( n \))
    - \( p \): start position
    - \( n \): number of hops
    - returns: nodes in the set
U11.A4b Backtracking

- Find a way (Implement `findCompletePath`: Returns, for a given starting position, the path that will touch every position on the board exactly once)
  - Visit all the fields and each field only once
  - Special case of the ‘Hamiltonian Path Problem’

- Early termination
  - In case all the fields are visited
  - Backtracking: delete last moves until the termination condition is not met
Have Fun!