Outlook

- Exercise 10: Solution discussion
- Exercise 11: Overview (Sort Search tree, complexity, Knight’s tour)
L10.A1a – Merge sort
L10.A1b – Merge sort pseudocode: sort()

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

```java
ArrayList sort (ArrayList unsorted, int begin, int end )
    if ( end - begin == 0 )
        return new ArrayList ( 0 )

    if ( end - begin == 1 ){
        ArrayList result = new ArrayList ( 1 )
        result.add ( unsorted[begin] )
        return result
    }
    // divide..
    ArrayList lhs = sort ( unsorted, begin, (begin+end) / 2 )
    ArrayList rhs = sort ( unsorted, (begin+end) / 2, end )

    // ..et impera
    return merge ( lhs, rhs )
```
L10.A1b – Merge sort pseudocode: merge()

- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

```java
ArrayList merge (ArrayList lhs, ArrayList rhs )
int left = 0, right = 0
ArrayList result = new ArrayList ( lhs.size + rhs.size )
loop
    if ( left == lhs.size )
        result.addAll ( rhs.subList ( right, rhs.size ) )
        break
    if ( right == rhs.size )
        result.addAll ( lhs.subList ( left, lhs.size ) )
        break
return result
```
L10.A1c,d – Merge sort

The graph shows the relationship between the function $f(n) = n \cdot \log(n)$ and the measurement values for different values of $n$. The graph includes a logarithmic scale for $f(n)$ and a linear scale for $n$. The graph demonstrates how the growth rate of $n \cdot \log(n)$ compares to the actual measurement data as $n$ increases.

- **$f(n)$** represents the function $n \cdot \log(n)$.
- **Measurement** represents the actual measurement data.

The graph illustrates that as $n$ increases, the growth of $f(n) = n \cdot \log(n)$ follows a pattern that is closely matched by the measurement data, indicating that the theoretical model is a good fit for the observed data.
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
How does it look like with 5 discs?

1. $1 \to 3$
2. $1 \to 2$
3. $3 \to 2$
4. $1 \to 3$
5. $2 \to 1$
6. $2 \to 3$
<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 → 3</td>
<td>2</td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 → 2</td>
<td>3</td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 → 2</td>
<td>1</td>
</tr>
<tr>
<td>10.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 → 1</td>
<td>2</td>
</tr>
<tr>
<td>11.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 → 1</td>
<td>3</td>
</tr>
<tr>
<td>12.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 → 2</td>
<td>1</td>
</tr>
</tbody>
</table>
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:

\[ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \]

\[ \rightarrow \ 3 \]

\[ 2 \]
L10.A2 – Tower 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 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
L10.A2b – Pseudocode

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

make possible move \(\rightarrow\) there is always only one possible way (the smaller disc, or the only disc)
L10.A3 – Reversi (Part 4)

- Implement an evaluation function that operates on the $\alpha$-$\beta$-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
L10.A3 – Reversi (Part 4)

```java
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, false);
        } else
            return new BestMove(finalResult(gb), null, false);
[...]
```
L10.A3 – Reversi (Part 4)

```java
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;
            bestCoord = coord;
        }
        if (alpha >= beta) {
            return new BestMove(alpha, null, false);
        }
        cut = cut || result.cut;
    }

    return new BestMove(alpha, bestCoord, cut);
```
Reversi Tournament

- Tournament: Wednesday 31.05.2017, 12:30, CABinett (Stuz2).
- Submission:
  - **Deadline: Wednesday, May 23, 2017, 23:59 (Zürich Time)**
  - Submit your player to the Reversi-Platform.
  - You can work alone or in groups of two
Outlook

- Exercise 10: Solution discussion
- Exercise 11: Overview (Sort Search tree, complexity, Knight’s tour)
U11.A2 Tree Sort

- Describe briefly how to use binary search trees for sorting.
- Specify the run-time complexity of a sorting operation with binary search tree for the best, average, and worst case in $O$-notation.
Asymptotic Complexity

- An important question is: How efficient is an algorithm or piece of code?
  - Efficiency covers lots of resources, including:
    - CPU (time) Usage
    - Memory Usage
    - Disk Usage
    - Network Usage

- Asymptotic time complexity and Asymptotic space complexity
  - When analyzing the running time or space usage of programs, we usually try to estimate the time or space as function of the input size.
  - The asymptotic behavior of a function f(n) refers to the growth of f(n) as n gets large.
U11 – 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 only, but also depends on the input value or the order of the input values.
- Then the following cases can be specified:
  - „best case“
  - „average case“
  - „worst case“
## U11 – Algorithm Complexity

<table>
<thead>
<tr>
<th>Name</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Cocktail sort</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Comb sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Gnome sort</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n + d)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>-</td>
<td>-</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary tree sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Library sort</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>In-place merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Smoothsort</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Introsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Patience sorting</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Quelle: wikipedia.org
Complexity Classes

- $\theta(1)$: Constant Complexity
- $\theta(\log n)$: Logarithmic Complexity
- $\theta(n)$: Linear Complexity
- $\theta(n \log n)$: $n \log n$ Complexity
- $\theta(n^b)$: Polynomial Complexity
- $\theta(b^n)$: Exponential Complexity
- $\theta(n!)$: Factorial Complexity
U11.A1 – Complexity

Big-O Complexity

Operations vs. Elements

- $O(1)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(2^n)$
- $O(n!)$

http://bigocheatsheet.com/
# U11.A1 – Complexity

<table>
<thead>
<tr>
<th>order of growth</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td><code>a = b + c;</code></td>
<td>statement</td>
<td>add two numbers</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td><code>while (N &gt; 1) { N = N / 2; ... }</code></td>
<td>divide in half</td>
<td>binary search</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td><code>for (int i = 0; i &lt; N; i++) { ... }</code></td>
<td>loop</td>
<td>find the maximum</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>linearithmic</td>
<td>[see mergesort lecture]</td>
<td>divide and conquer</td>
<td>mergesort</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td><code>for (int i = 0; i &lt; N; i++) { ... }</code></td>
<td>double loop</td>
<td>check all pairs</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td><code>for (int i = 0; i &lt; N; i++) { ... }</code></td>
<td>triple loop</td>
<td>check all triples</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>[see combinatorial search lecture]</td>
<td>exhaustive search</td>
<td>check all subsets</td>
</tr>
</tbody>
</table>

https://class.coursera.org/algs4partI-007/lecture
U11.A2 – Complexity Analysis

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

- Valid solution
  - Calculation steps and resulting O-Notation!
// 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++;
// Fragment 1

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

\[ c_0 + c_1 n \sim O(n) \]
U11.A3 – Complexity (I)

\[ t'_{op} = \frac{1}{3} t_{op} \]

- \( M' \)
- \( T'_{tot} \)

Time per Operation
Input Size
Total run time
### U11.A3 – Complexity (II)

<table>
<thead>
<tr>
<th>$O(\ldots)$</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>$t_{op} \times M'<em>1 = t</em>{op} \times M_1 \Rightarrow \frac{1}{3} t_{op} \times M'<em>1 = t</em>{op} \times M_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$M'_1 = 3M_1$</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 move
  - Given: startPosition
U11.A4a – Knight’s tour

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

- Method getReachableSet
  - 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
  - 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!