O-Calculus: Rules

• Important computation rules:

\[ f(n) \in O(r(n)) \text{ and } g(n) \in O(s(n)) \Rightarrow f(n) + g(n) \in O(r(n) + s(n)) \]

\[ f(n) \in O(r(n)) \text{ and } g(n) \in O(s(n)) \Rightarrow f(n) \cdot g(n) \in O(r(n) \cdot s(n)) \]

\[ f(n) \in O(r(n)) \text{ and } k \in \mathbb{N} \Rightarrow f(n) \pm k \in O(r(n)) \]

\[ f(n) \in O(r(n)) \text{ and } k \in \mathbb{N} \Rightarrow f(n) \cdot k \in O(r(n)) \]

• Functions with different growth rate (ascending order):

\[ c, \log n, \sqrt{n}, \frac{n}{\log n}, \frac{n^n}{\log n}, n, n \log n, n (\log n)^2, n (\log n)^p, \]

\[ \frac{n^2}{(\log n)^q}, \frac{n^2}{\log n}, n^2, n^2 \log n, n^2 (\log n)^p, \ldots, n^3, \ldots, n^p, \ldots, \]

\[ 2^n, e^n, 3^n, \ldots, p^n, n^n \]
<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>O-Calculus</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
<td>3, 10, 23</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log n)$</td>
<td>2 \cdot \log n, 5 + \log(2 + 3n)</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
<td>1 + n, n + \log n, 10n</td>
</tr>
<tr>
<td>Logarithmic-Linear</td>
<td>$O(n \log n)$</td>
<td>2n \log n, n \log(3 + n)</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^k)$</td>
<td>2n^2 + 3n, 16n^3 + 5</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(a^n)$</td>
<td>2^{4n}, 3^n + n^3</td>
</tr>
</tbody>
</table>
Different complexity functions with respect to „small“ values of n.
Example:

Problem area: add all numbers between 1 and 100

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sum = 0;</td>
<td>int sum = 0;</td>
</tr>
<tr>
<td>for (int i = 1; i &lt;= 100; i++)</td>
<td>for (int i = 1; i &lt;= 50; i++)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>sum = sum + i;</td>
<td>sum = sum + i;</td>
</tr>
<tr>
<td>}</td>
<td>sum = sum + (101 - i);</td>
</tr>
</tbody>
</table>

**Same complexity classes!**

run time = 2 * n + 2
(2 * n + 2) ∈ O(n)

run time = 4 * n/2 + 2
(4 * n/2 + 2) ∈ O(n)

You see that both algorithms show the same run time behavior. They behave in the same way according to the input size of n.
The following table shows an overview to the **access possibilities** and the **run time behavior** of different operations to several data structures from chapter 4.

<table>
<thead>
<tr>
<th></th>
<th>indicated access</th>
<th>List Operations</th>
<th>Front Operations</th>
<th>Back Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search tree</td>
<td>O(log(n))</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Simple linked list</td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
<td>O(n)</td>
</tr>
<tr>
<td>Double linked list</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Array</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- **Indicated access**: access to an element on a previously given position
- **List operations**: insert or delete elements from a specific position
- **Front operation**: insert or delete from the beginning of the data structure
- **Back operation**: insert or delete at the end of the data structure
If $f$ and $g$ are functions, and $c$ is a constant.

**Rules:**

\[
\begin{align*}
f &= O(f) \\
O(O(f)) &= O(f) \\
O(c \cdot f) &= O(f) \\
O(f + c) &= O(f) \\
O(f) + O(g) &= O(f + g)
\end{align*}
\]

**Examples:**

\[
\begin{align*}
x^2 + 2x + 10 &= O(x^2 + 2x + 10) \\
O(O(n^2)) &= O(n^2) \\
O(230n^2) &= O(n^2) \\
O(x^3 + 4) &= O(x^3) \\
O(x^3) + O(x) &= O(x^3 + x)
\end{align*}
\]
(a) Sum of $n$ numbers: $f(n) = 5n + 3$

For: $f \in O(g)$, with $g(n) = n$

Notation: $f \in O(n)$

- $f(n) = n^2 + 1000n$
  
  $f \in O(n^2)$ then choose $c = 2$, $n_0 = 1000$

  $$n^2 + 1000n \leq c \cdot n^2 \quad \forall \quad n \geq n_0$$
product = 1;

while (n > 1) {
    if (n mod 2 == 1) {
        product = product * x;
    }
    x = x * x;
    n = n div 2;
}

product = product * x;

- Before the loop, we have a constant operation, thus O(1).
- Inside the loop, the operation is constant, whether the IF-case happens or not; thus, also O(1).
- After the loop, the operation is also constant, thus O(1).
- Question: How often is the loop run?
Note:

- The O-Notation gives an „upper limit“. That doesn’t mean that the algorithm indeed requires this amount of time. It just means that in no case the algorithm requires more time.

- The O-Notation estimates the run time for an infinite input. But no real input is infinite, and therefore the actual dimension of the input should be considered, too.
Note:

\[ O(f) = \{ g \in R_+^N \mid \exists c_{1,2} > 0 : \forall n \in N : g(n) \leq c_1 \cdot f(n) + c_2 \} \]

- The system dependent constants \( c_{1,2} \) are usually unknown. They must not necessarily be small though. If they are large, this can be a significant risk: suppose an algorithm needs \( n^2 \) nanoseconds (quadratic complexity), and another one “only” needs \( n \) centuries (linear complexity). With respect to the O-notation a linear complexity appears to be better than a quadratic one! Therefore the O-notation here isn’t suitable for the decision.

- O-notation is a theoretical estimation! For a qualitative statements the system, too, must be taken into consideration, i.e. processor, usage (data amount), etc.
Requirements for algorithm design are:

• **Systematic and reproducible design**
  - allows for systematic testing and verification
  - simplifies care, maintenance, and advancement of programs

• **Division of labor in the design (in large problem areas)**
  - Early structuring
  - Distribution into smaller problems

• **Efficiency of the designed algorithm**
  - Time complexity
  - Storage complexity
So far, we only considered short and simple algorithms. The design of such algorithms is relatively **problem free**, since they are manageable and the operation sequence can quickly be derived from the mathematical definition of the problem area.

**In practice, often extensive and rather complex** algorithms are necessary. The design of such algorithms is **complicated**.

Algorithm design is a part of **Software Engineering** and is today partially supported by computers.

The most important **concepts** are:

- Stepwise refining
- Modularizing
Stepwise Refining

- The elementary operations that describe an algorithm in its final form are generally relatively simple.
- If an algorithm is considered to be a complex operation, that is comprised of many elementary operations and run structures, the question arises how we can get from the complex operation to the elementary operation in a simple and manageable manner.
Principles of stepwise refining:

1. Sketch a raw algorithm with **abstract operations and data types**

2. Refine the operations in the first step, that means: **implementation with fewer abstract operations and data types**

3. Repeat the refining step till you have an **algorithm**, that only contains the run structures and elementary operations that are available.
Example: Sorting with direct insertion

Problem area:
A given list \( F = (k_1, k_2, k_3, \ldots, k_n) \) of values shall be sorted in an order <.

step 1:
\[ \text{sort}(k_1, k_2, k_3, \ldots, k_n); \]

The algorithm is first defined as an abstract operation.
step 2:
The algorithm will be refined through multiple abstract or concrete operations, in which the sorting problem is reduced to a series of insert-operations.

```c
for (i = 2; i <= n; i++) {
    insert i-th element a[i] into the list (a[1], a[2], … a[n]) at its correct position
}
```
Stepwise Refining - Example (3)

step 3:

The insert operation is refined through the **input of concrete insert-positions**. At this position the elementary operations are already abstractly described.

```c
for (i = 2; i <= n; i++) {
    int j = i - 1;
    note the value of a[i]
    while((j != 0) && (a[i] < a[j])) {
        push a[j] one spot to the right
        j = j - 1;
    }
    insert the noted value in the position here (j + 1)
}
```
**Step 4:**

The abstract descriptions of the elementary operations are translated into **concrete operations**. Thus, the finished algorithm appears in JAVA-code:

```java
for (i = 2; i <= n; i++) {
    int j = i - 1;
    int k = a[i];
    while((j != 0) && (a[i] < a[j])) {
        a[j + 1] = a[j];
        j = j - 1;
    }
    a[j + 1] = k;
}
```
• Stepwise refining requires that simultaneous refining of the algorithm component parts matches with one another. That means, that single component parts of the algorithm should be refined to the same level.

• This leads to the disadvantage, that the refining of large and complex algorithms can be very complex.

• A division of the problem into smaller parts can help. The parts can then be individually refined by stepwise refining and later be put together again. This is called modularizing.
Modularization (1)

- Modularization decomposes the problem into parts…

  - that are clearly separated from one another: 
    
    *a partial problem solves a specific problem that can’t be further subdivided.*

  - that are largely independent from one another, 
    
    *The solutions to single partial problems don’t influence each other, but can make use of one another.*

  - whose solutions are **interchangeable** to alternative solutions of the particular partial problem without side effects.
• The creation of algorithms and software follows the "building block principle". Single solution building blocks and partial solutions are called modules.

• The largely independent modules are provided with a clear-cut specified interface. This allows for the single modules to be easily combined later on.

• Single modules, esp. algorithms, can be independently tested from one another, verified, or advanced.

• Modularization allows for an unproblematic and good transition to successor versions of a software product by the exchange of modules.
Possibilities for classifying modules:

- **Problem oriented modularization**,  
  *The module is subdivided into closed processing units, so that each module fulfills a problem specific task.*

- **Data oriented modularization**,  
  *The module is split according to the data on which the module will work on.*

- **Function oriented modularization**  
  *Algorithms and esp. modules with similar functions are combined together.*
• The methods “stepwise refining” and “modularizing” are generally valid and recognized design concepts. They are universally used.

• In contrast to this, design techniques include solution ideas and concepts that are designed for specific problem areas. For example, the following:

  - Systematic Trying and Backtracking
  
  - Divide and Conquer.
Assumption:

- An exact or direct solution of the problem is not possible in appropriate time.
- It is possible to generate all possible solutions or partial solutions of the solution space.

Procedure:

- Create the solution space (e.g. with a solution tree)
- **Systematic (recursive) search** of the solution space.
- If a partial solution leads to an invalid solution, the last step is un-done. Now try to lead the reduced partial solution to a valid general solution by trying another way.
**Procedure:**

1. Starting with the node of the solution tree that represents the current partial solution, we test if the next (succeeding) node leads to a success (what success means depends on the problem)

2. If not: try the next node

3. If no succeeding node leads to a positive result, go back to the ancestor of the current node, and continue the search there.

![Solution tree diagram](image)
• Backtracking algorithms usually have a **exponential time complexity**.

• This procedure is only recommendable if no other algorithm is known, or developing a better algorithm is too expensive.

• Problems suitable for Backtracking are:
  - **Jigsaw – or game problems:**
    search for the best move, search for positions with certain properties
  - **Graph problems:**
    search for certain paths (round trip)
  - **Optimization problems and combinatorics**
  - etc.
The basic idea of the „Divide and Conquer“ approach is to split problems into \textbf{smaller parts} so that it is easier to handle small problems and jobs, and not the entire problem at once.

The solutions to the parts are in the end used for solving the entire problem.

In general, this method is used \textbf{recursively} until small problem sizes are achieved.

This approach can have a better efficiency if the time for solving the smaller problems and the time for combining the solutions is less than solving the problem all at once.
The *Divide-and-Conquer-Strategy* for the solution of a problem consists of 3 steps:

- **Divide**: The problem is divided into parts
- **Conquer**: The single parts are solved
- **Combine**: The solutions of the single parts are re-combined to the solution of the original problem.
Example: Sorting algorithm „Mergesort“ (alphabetical order)

Dividing by 2

Symbol chain

Merging

Source: http://de.wikipedia.org/wiki/Mergesort
"We already have quite a few people who know how to divide. So essentially, we're now looking for people who know how to conquer."
[Schn98] H.-J.Schneider (Hrsg.), Lexikon der Informatik, Oldenbourg Verlag, 1998
Principle:

- Bubblesort is one of the simplest sorting algorithms.
- In the first passage, the smallest element is sought for, in the second passage, the second smallest, etc.
- The array is traversed from back to front. First, the last element is compared to the second last element. If the last element is smaller than the second last, they switch. The same operation is performed with the second last and the third last element. By this, the larger elements stay in place, while the smaller elements are forwarded.
- Depending on the sorting direction (upwards or downwards), the bigger or the smaller elements ascent like bubbles in the water.
**Principle:**

- Quicksort **chooses an element** (pivot element) from the list to be sorted and **divides the list into two sublists**, a lower one, that contains **all elements smaller** and an upper one, that contains **all elements bigger or equal** to the pivot element.

- For this purpose, an element is sought in the lower list that is bigger than (or equal to) the pivot element. This element is too big for the lower list and mustn’t stay there. Accordingly, an element smaller than the pivot element is sought in the upper list. These elements switch places and are thus placed in the correct sublist. The procedure is repeated, **until the upper and the lower search meet**. Thus two correct sublists (see above) are created in one run.
**Mergesort**

**Principle:**
- Similar to quicksort, the method is based on the **Divide-and-Conquer-strategy**.
- The list to be sorted is subdivided into **two sublists of equal size**.
- The corresponding lists are subdivided continuously, until they reach a **length of 1**.
- **Merging** is performed by comparing the first elements of two sublists: the smaller one is deleted and added to the new (sorted) list.
Example: Mergesort (1)

Unsorted Data Set

Part A

Part B

Part A (sorted)

Part B (sorted)

Sorted Data Set
Procedure: Divide the list into two lists (of the same size) and divide these again and again until the sub-lists contain only one single element.
Now, two sublists are sorted into one list that is twice as long. For this, the first elements of each list are compared and the smaller one is transferred into the new list, until all the elements are sorted.
The last step consists of merging two lists of length $n/2$ to one list of length $n$ by sorting them.

- Compare both halves with an index $i$ and an index $j$ for each element and copy the next largest element in the results list.
Example: Mergesort (5)

- Compare the first element of the first list with the first element of the second list.
- Copy the smallest element into the results list.

List 1

```
2 3 4 5
```

List 2

```
0 1 6 7
```

Results list

```
0
```
• Compare the first element of the first list with the second element of the second list.
• Copy the smallest element into the results list.
Example: Mergesort (7)

• Compare the first element of the first list with the third element of the second list.
• Copy the smallest element into the results list.

List 1

2 3 4 5

List 2

0 1 6 7

Results list

0 1 2
• Compare the second element of the first list with the third element of the second list.
• Copy the smallest element into the results list.
Example: Mergesort (9)

- Compare the third element of the first list with the third element of the second list.
- Copy the smallest element into the results list.

List 1

2 3 4 5

List 2

0 1 6 7

Results list

0 1 2 3 4

index i

index j
Example: Mergesort (10)

- Compare the fourth element of the first list with the third element of the second list.
- Copy the smallest element into the results list.
Example: Mergesort (11)

- Copy the next element from the second list into the results list.

```
<table>
<thead>
<tr>
<th>List 1</th>
<th>List 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 4 5</td>
<td>0 1 6 7</td>
</tr>
</tbody>
</table>

Results list: 0 1 2 3 4 5 6
```
Example: Mergesort (12)

- Copy the last element from the second list in the results list.

List 1

```
2 3 4 5
```

List 2

```
0 1 6 7
```

Results list

```
0 1 2 3 4 5 6 7
```
Example: Mergesort (13)
public class MergeSorter {
    private static int[] a, b; // Help array b
    private static void merge(int lo, int m, int hi) {
        int i, j, k;
        // copy both halves of a in the help array b
        for (i=lo; i<=hi; i++)
            b[i]=a[i];
        i=lo; j=m+1; k=lo;
        // copy back the next largest element
        while (i<=m && & j<=hi)
            if (b[i]<=b[j])
                a[k++]=b[i++];
            else
                a[k++]=b[j++];
        // copy the rest of the first half if still there
        while (i<=m)
            a[k++]=b[i++];
        while (j<=hi)
            a[k++]=b[j++];
    }
    public static void sort(int[] a0) {
        a=a0;
        System.out.println("unsorted order: ");
        System.out.println();
        int n=a.length;
        b=new int[n];
        mergesort(0, n-1);
        System.out.println("\nsorted order: ");
        System.out.println();
    }
}

public class MergeSorter {
    private static int[] a, b; // Help array b
    private static void merge(int lo, int m, int hi) {
        int i, j, k;
        // copy both halves of a in the help array b
        for (i=lo; i<=hi; i++)
            b[i]=a[i];
        i=lo; j=m+1; k=lo;
        // copy back the next largest element
        while (i<=m && & j<=hi)
            if (b[i]<=b[j])
                a[k++]=b[i++];
            else
                a[k++]=b[j++];
        // copy the rest of the first half if still there
        while (i<=m)
            a[k++]=b[i++];
        while (j<=hi)
            a[k++]=b[j++];
    }
    public static void sort(int[] a0) {
        a=a0;
        System.out.println("unsorted order: ");
        System.out.println();
        int n=a.length;
        b=new int[n];
        mergesort(0, n-1);
        System.out.println("\nsorted order: ");
        System.out.println();
    }
}
// The TestMergeSorter is a test class
public class TestMergeSorter {
    public static void main(String[] args) {
        int array[] = {2,5,4,3,1,7,0,6};
        MergeSorter.sort(array);
    }
}

Output on the screen:
The unsorted order:
2 5 4 3 1 7 0 6
The sorted order:
0 1 2 3 4 5 6 7
• Bubble sort
  
  - Expense:
    - Best case: field is already sorted: $T(n) = O(n)$
    - Worst case: field is decreasingly sorted: $T(n) = O(n^2)$
  
  - Advantages and disadvantages:
    - Easy to program and understand
    - Can easily be modified to be very fast by sorting data
    - Doesn’t need additional storage, stable
    - Slow with random data, because there are many comparisons
    - Still slow when the data is partially sorted.
• Quick sort
  - Expense:
    - Best case: $T(n) = \Theta(n \log(n))$
    - Worst case: $T(n) = \Theta(n^2)$
  - Advantages and disadvantages:
    ☺ Doesn’t need additional storage
    ☺ As a rule, generally faster than Merge sort by a factor of 2
    ☺ In practice, often the fastest algorithm
    ☹ Not stable
    ☹ Worst case $\Theta(n^2)$
• Merge Sort

- Expense: \( T(n) = \Theta(n \log(n)) \)

- Advantages and disadvantages:
  - ☺ Can be bettered in the sense that you can separately treat simple cases with 2 or 3 elements.
  - ☺ Expense for all cases is \( T(n) = \Theta(n \log(n)) \)
  - ☹ Algorithm requires additional storage space.
  - ☹ Not stable

Note: The consistent generation of help fields is ineffective. Therefore, an algorithm is faster when you create a large help field and apply this over and over again.
• As an interdisciplinary subject area of technical computer science, robotics deals with the navigation processes for a mobile and sensor-led robot, among others.

• The navigation of a mobile robot is outlined in the following steps:
  1. **Mapping and localization**
     Generate a map of the environment and determine your position
  2. **Path design**
     Calculate the optimal path from your current point to the goal point
  3. **Rules of motion**
     Follow the previously calculated path with help from sensors

*Source: Fraunhofer IPA*
• As an example to the practical application of algorithms, we will take a closer look at the path design.

• The goal of path design in robotics is to give robots the ability to independently design their own movements and actions starting from the start situation to the target situation.

• This is essential for being able to develop autonomous robots.

• Even when such autonomy is not required, these processes can serve as programming relief for users, in that he only has to define the task, since the robot can then carry out the rest of the task.

• Complex tasks in a dynamic environment make manual path specification almost impossible.
• Exploratory robots
  - Steering exploratory robots on Mars is not possible due to the long signal run time between Earth and Mars.
  - If a controller were to send a steering signal to the vehicle, he would have to wait 40 minutes until he sees from which cliff it is going to fall down.

source: NASA
• Service robots

- Service robots perform tasks for people that are, for instance, not able to be performed by people or are too dangerous.

- This type of robot is still being developed, and it just now starting to be seen as a possibility for performing household and office tasks.

- Possible application areas are: assistance systems, museum robots, autonomous road sweepers, and production assistants.

(source: Fraunhofer IPA)
Example (3)

• Game robots
  - The fast progressive development and potential of robots is shown every year in RoboCup
  - The latest processes and constructions are demonstrated in a fun and competitive manner against each other.

source: RoboCup 2005, Noriaki Mitsunaga
• Problem of path design:

Starting from a start position and orientation, a target position and orientation should be achieved. Determine the path $\pi$, that defines the complete order of positions and orientations, so that no barrier is touched and the goal is reached.

**Example:** The parallel parking problem

*source: Marco Loh, TU München*
• As a possible path design strategy, a Voronoi diagram can be used.

• In the context of path design, the Voronoi diagram shows those paths that maintain a maximum distance to the barriers (points).
• One possibility for constructing Voronoi Diagrams is the following recursive algorithm, based on the principle of the “divide and conquer“ approach.

Requirements
A number $P$ of $n$ points are given in a plane.

Algorithm
1. Divide $P$ into 2 equally sized heaps $P_1$ and $P_2$
2. Calculate the Voronoi Diagram for $P_1$ and $P_2$
3. Merge both Voronoi Diagrams for $P_1$ and $P_2$ to the Voronoi Diagram for $P$
Given heap of points $P$
1. Divide $P$ into 2 equally sized heaps $P_1$ and $P_2$.
2. Calculate the Voronoi Diagram for \( P_1 \) and \( P_2 \)

By the recursive break down of the point heap \( P \), the construction problem area of a Voronoi Diagram can be reduced to 2 simple cases.

**case 1:** Voronoi Diagram construction with **two points**

The **vertical line** sets up a Voronoi Diagram with 2 points
2. Calculate the Voronoi Diagram for $P_1$ and $P_2$

By the recursive break down of the point heap $P$, the construction problem area of a Voronoi Diagram can be reduced to 2 simple cases.

**Case 2:** Voronoi Diagram construction with **three points**

The **vertical lines of all point pairs** are cut at a common intersection. They set up a Voronoi diagram with 3 points.
2. Calculate the Voronoi Diagram for $P_1$ and $P_2$
3. Merge both Voronoi Diagrams for $P_1$ and $P_2$

3.1. Connect the closests neighbour along the dividing line
3. Merge both Voronoi Diagrams for $P_1$ and $P_2$

3.2. Construct the new vertical lines
3. Merge both Voronoi Diagrams for $P_1$ and $P_2$

3.3. Cut the vertical lines at the intersecting points
Result: The finished Voronoi Diagram for \( P \)
• Example of a complex environment and the corresponding Voronoi Diagram and the chosen path of a mobile robot

Voronoi Diagrams: Advantages and Disadvantages

• Advantages:
  - Maximum distance to the barriers
  - The robot can easily test- with help from the distancing sensors - if the correct path will be followed and if it’s not to be followed, correct itself.

• Disadvantages:
  - The path is normally not the shortest way
  - When there are only a few barriers, then only a few paths can be generated