• In order to be able to **compare** algorithms with one another and **judge** their efficiency with respect to the problem to be solved a method is needed to **measure and calculate the efficiency**.

• **The most important evaluation criteria** in this respect is the **complexity**. The complexity theory of computer science tries to **classify** algorithmic problems according to their **needs for calculation resources**.

• Calculation resources are the **calculating time** and the **storage space** the algorithm needs for solving the problem.
**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++) { sum = sum + i; }</td>
<td>for (int i = 1; i &lt;= 50; i++) { sum = sum + i; sum = sum + (101 - i); }</td>
</tr>
</tbody>
</table>

While algorithm 1 must run the `for-loop` 100 times, the `for-loop` in algorithm 2 only has to run 50 times. Hence, this clearly requires fewer comparisons, but correspondingly more additions/subtractions.

**Question:** which algorithm is more efficient (runs faster)?
This question shows which influencing factors must be considered when trying to judge the efficiency:

- **The number and size of the input data (problem size)**
  The addition of all numbers up to 100 is obviously faster than the addition of all numbers up to 10000.

- **the algorithm itself**
  Algorithms can use different solution approaches to solve a problem. This can lead - despite the same in- and output data - to different times until termination.

- **Characteristics of the computer on which the algorithm is performed**
  The implementation of the algorithm on a specific computer determines the time needed to calculate the elementary operations. Addition or multiplication require different amounts of time, according to the computer type.
Assuming **algorithm 1** runs on a computer, on which the elementary operations have been defined as the following from the calculation time:

\[
\begin{align*}
 t_{\text{assignment}} & = 1 \mu s \\
 t_{\text{Addition}} & = 2 \mu s \\
 t_{\text{Subtraction}} & = 2 \mu s \\
 t_{\text{equals}} & = 4 \mu s
\end{align*}
\]

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>calculation time/lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sum = 0;</td>
<td>1\mu s</td>
</tr>
<tr>
<td>for (i = 1;</td>
<td>1\mu s</td>
</tr>
<tr>
<td>i &lt;= 100;</td>
<td>4\mu s</td>
</tr>
<tr>
<td>i++) {</td>
<td>2\mu s</td>
</tr>
<tr>
<td>sum = sum + i;</td>
<td>2\mu s</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

The computer will require **802\mu s** in algorithm 1 when „100” is the input. For the input of \( n \) (in this case, 100) this means that the algorithm has a run time of \( t = 8\mu s \times n + 2\mu s \)

**Note:** Although algorithm 2 appears to be longer, it requires only **602\mu s** when \( n = 100 \).
The consideration of the execution times of single operations on specific computers proves to be of little use. Identical algorithms require different amounts of time on different computers. Thus they had to be compared depending on the computer on which they are performed.

Complexity theory makes the following idealized assumptions:
- Each elementary operation (assignment, addition, subtraction, multiplication, comparing 2 numbers, etc.) requires exactly one computing step.
- Complex operations (i.e. matrix multiplication) are not available.
- The data is viewed as one uniform size, that means, a cell contains always exactly one datum.
Furthermore, the specific run times are no longer measured. Instead, the run times are divided into **complexity classes**:

**Example:**

**Algorithm 1**

```c
int sum = 0;
for (i = 1; i <= 100; i++)
{
    sum = sum + i;
}
```

The loop runs with an input of $n = 100$ exactly 100 times.

This means, dependent on the input size, the loop will run $n$-times. The algorithm 1 has therefore a **linear run time behavior**.

In order to formalize this classification in the complexity class, the term **O-Notation** (meaning: „large-O-Notation“) is established.
Definition:

Let $P$ be the set of all functions $f : \mathbb{N} \rightarrow \mathbb{R}$, and let $g$ belong to $P$. Then the following will be determined:

(a) $O(g) := \{ f \in P \mid \exists n_0 \in \mathbb{N} \ \exists c > 0 \ \forall n > n_0 : |f(n)| \leq c \cdot |g(n)| \}$

for $f \in O(g)$ it is said: "$f$ grows maximally as strong as $g"$, or "$f$ has maximally the order $g"$, or "$f$ is of „O-Big“ of $g"$.

(b) $\Omega(g) := \{ f \in P \mid \exists n_0 \in \mathbb{N} \ \exists c > 0 \ \forall n > n_0 : |f(n)| \geq c \cdot |g(n)| \}$

for $f \in \Omega(g)$ it is said: "$f$ grows at least as strong as $g"$, or "$f$ has at least the order $g"$.}
• With \( f \in O(g(n)) \) it is expressed that \( f(n) \) grows \textit{maximally} as fast as \( g(n) \).
for \( f \in \Theta(g) \) it is said: \( f \) grows exactly as strong as \( g \), or \( f \) has exactly the order \( g \). In this case, the reversing of \( g \in \Theta(f) \) is also true.

\[ \Theta(g) \subseteq \bigcap \left( \Omega(g) \right) \]

\( O(g) \), \( \Omega(g) \) and \( \Theta(g) \) thus describe sets of functions which for large \( n \) grow maximally or minimally or exactly as strong as the function \( g \).
• Informal: \( g \) is "of order \( f \)" (notation \( O(f) \)), if:

- For an \( n \) big enough, \( 0 \leq g(n) \leq c \cdot f(n), \text{ for } c > 0 \) constant
• With $f \in \Theta(g(n))$ it is expressed that $f(n)$ grows exactly as fast as $g(n)$.
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):
  \[ \log n, \sqrt{n}, \frac{n}{\log n}, n, n \log n, n(\log n)^2, n^2, n^3, 2^n, e^n, 3^n \]
### Complexity Classes

<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 + 3$n$)</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
<td>1 + $n$, $n + \log n$, 10$n$</td>
</tr>
<tr>
<td>Logarithmic-Linear</td>
<td>$O(n \log n)$</td>
<td>2$n$ log $n$, $n$ log(3 + $n$)</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^k)$</td>
<td>2$n^2$ + 3$n$, 16$n^3$ + 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 $n$.

- $O(n)$: Linear
- $O(\log n)$: Logarithmic
- $O(n\log n)$: Logarithmic-Linear
- $O(n^k)$: Polynomic
- $O(a^n)$: Exponential

[Bieh00]
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>

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 3.

<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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple linked list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</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:

\[ f = O(f) \]
\[ O(O(f')) = O(f') \]
\[ O(c \cdot f) = O(f) \]
\[ O(f + c) = O(f) \]
\[ O(f') + O(g) = O(f + g) \]

Examples:

\[ x^2 + 2x + 10 = O(x^2 + 2x + 10) \]
\[ O(O(n^2)) = O(n^2) \]
\[ O(230n^2) = O(n^2) \]
\[ O(x^3 + 4) = O(x^3) \]
\[ O(x^3) + O(x) = O(x^3 + x) \]
(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 $c = 2$, $n_0 = 1000$

  $n^2 + 1000n \leq c \cdot n^2 \forall n \geq n_0$
O-Calculus: Example (2)

```java
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 application, thus $O(1)$.
- Inside the loop, the application is constant, whether the IF-case happens or not; thus, also $O(1)$.
- After the loop, the application 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 **free of problems**, 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.

```java
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
}
```
step 3:

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

```java
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

- Stepwise refining requires that **simultaneous refining of the algorithm 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.
• Backtracking algorithms usually have an **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 **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 **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)

Symbol chain

Dividing
by 2

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."
Recommended Literature


[Schn98] H.-J. Schneider (Hrsg.), Lexikon der Informatik, Oldenbourg Verlag, 1998