Computer Science for Engineers

Lecture 11

Algorithms – part 1

Prof. Dr. Dr.-Ing. Jivka Ovtcharova
Dipl. Wi.-Ing. Dan Gutu
29th of January 2010
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation
4. Data Structures
5. Algorithms
   5.1. Introduction
   5.2. Characteristics
   5.3. Complexity
   5.4. Design Methods
   5.5. Examples of Algorithms
• The term „algorithm” means, generally speaking, an exact instruction for the solution of a problem or a specific type of problems.

• Algorithms are one of the central topics in computer science and mathematics. As computer programs or electric circuits, they steer the actions of computers or other machines.

• There exist various representations for algorithms. They range from algorithms being the abstract complement to the concrete programs tailored for a machine, up to the idea to see algorithms as programs for ideal mathematical machines.

• Theoretical computer science uses the abstraction of these mathematical machines as a basis for calculation models. Thus definitions and conclusions about specific problem classes can be formalized.
In chapter 1, the term „algorithm“ was defined as follows:

**Definition 1.2:** An algorithm is a precise, e.g., drafted in an acceptable language, finite description of a general process with performing of executable elementary (processing) steps, whose task can be accomplished with one input \[\text{[Bieh00]}\].
Examples of Algorithms

Changing the tire on a car:

1. Loosen the wheel nut
2. Lift the vehicle
3. Unscrew the wheel nuts
4. Exchange the tire with another tire
5. Screw the nuts on again
6. Lower the vehicle to the ground
7. Tighten the wheel nuts

Other everyday examples:
Recipes, repair manuals, etc.
1. Operations

- For the **formal representation** of an algorithm, a **set of elementary operations** must be defined, which can be understood and performed by the user.

  for example: mechanic
  - (car) lift
  - (wheel nut) loosen
  - (wheel nut) unscrew
  - (tire) change
  - etc.

- The instructions of an algorithm are performed in **a specific order**.
2. Objects

- **According to the instruction, objects** are handled. Objects can be of **abstract** or of **concrete nature**.

  for example: car tires - car
  - tire
  - wheel nuts
  - etc.

- The involved objects are formalized in a „computer comprehensible“ way by describing their **characteristics through data**.

  for example: a single tire is described by its characteristics like manufacturer, size, type, etc.
1. Formal description of the problem area and boundary conditions

example:

A rational number \( a \) shall be multiplied with a natural number \( n \)

input: \( a \in \mathbb{R}, n \in \mathbb{N} \)

output: \( x = n \cdot a \)

boundary conditions:

- Elementary operations:
  
  assignment (\( := \)), Addition (\( + \)),
  Subtraction (\( - \)), equal (\( = \))

- run structures:
  
  Sequence, loop (WHILE)
2. Formulation of the algorithm with the help of existing elementary operations.

example:

input: \( a \in \mathbb{R}, n \in \mathbb{N} \)

\[ x := 0 \]

as long as \( n > 0 \)

\{ \[
  x := x + a \\
  n := n - 1
\] \}

output: \( x \)

Here the multiplication is represented by multiple addition. We need the elementary operations assignment and addition. Subtraction and equal operations are also needed in order to steer the While-loop of the algorithm and to test the abort conditions for the multiple addition.
3. The general and formally described algorithm is translated by the implementation into an understandable and executable language for the computer.

**example:** (Java)

```java
double multiply(double a, int n) {
    double x = 0;
    while (n > 0) {
        x += a;
        n--;
    }
    return x;
}
```

Simplifies the addition and the assignment with the operator `+=`

Decrements the loop variable `n` and steers the number of repetitions of the loop
The investigation and analysis of algorithms is one of the main tasks in computer science, and is mostly theoretically performed (without concrete transition into a programming language).

Possible criteria for classifying an algorithm are (in order of importance):

1. complexity of calculating time
2. complexity of storage space for cache
3. clarity and maintainability
4. easy verification of correctness
5. easy implementation
Classification of the characteristics of algorithms:

- **Problem independent characteristics:**
  They refer exclusively to the considered algorithm, but not to the problem or the specification.

  **Examples:**
  - Finiteness
  - Termination
  - Determination
  - Determinism
  - Recursivity
  - Parallelism

- **Problem related characteristics:**
  They refer both to the algorithm and to the problem or specification.

  **Examples:**
  - correctness
  - efficiency
Finiteness Condition:
An algorithm must be finitely describable - that means, it must be possible to formulate it with (finite) text.

- The condition for finiteness refers to the description of the algorithm, not to its execution.
- The description consists of a finite number of elementary operations. But when the algorithm is executed, an arbitrary number of operations can be executed – e.g. caused by loops. This can lead to the situation that the algorithm does not end (not terminate).
Finiteness Condition (2)

A counter example:

Pi \( \pi \) shall be shown on the screen:

```java
System.out.print("3");
System.out.print٬
System.out.print("1");
System.out.print("4");
```

Since \( \pi \) has an \textbf{infinite number of positions} after the decimal point, our program (when it’s done as shown on the left) must consist of an infinite number of lines (1 line per position), too.

The sequential output of single numbers of \( \pi \) leads to a \textbf{program code that is just as long}.

\( \Rightarrow \) The condition for finiteness is not fulfilled.
Termination:

An algorithm is **terminating** when it reaches a result for each valid input after a finite number of steps.

- An algorithm must terminate after a finite amount of time in a controlled way.

- The actual **number of performed steps** can be **arbitrarily large**.

- Generally, it's not possible to decide for each arbitrary algorithm if it terminates or not. **Many algorithms are too complex** even to be described by a mathematical set of rules.
Note: an algorithm that is described with a final source text (program text), can still have an infinite run time.

Example: (Java)

```java
double multiply(double a, int n)
{
    double x = 0;
    while (n > 0)
    {
        x += a;
        n--;  // Decrements the loop variable to control the loop iterations
    }
    return x;
}
```

This is an infinite loop, because \( n \) is not decremented in the loop: the aborting conditions in the while instruction are never fulfilled. This is a typical beginner error in programming with Java (and C++).
Termination (2)

Counter example:

input: \( n \)

repeat
  \( n := n + 1 \)
  till \( n = 50 \)

output: \( n \)

With an input of \( n < 50 \) the loop is exited after \( n = 50 \) and the algorithm then terminates.

With an input of \( n \geq 50 \) the aborting condition will never be fulfilled.

\( \Rightarrow \) The condition for termination is not fulfilled.
Determination

An algorithm is determined if the same parameters and starting values always lead to the same result.

- The determination properties guarantee a clear dependence of the output data from the input data. Thus it’s possible to describe functions with algorithms.

- An algorithm is not determined if it’s output is partly based on coincidence.
Determinism

An algorithm is deterministic when in each step of execution there exists only exactly one possibility for the continuation of the program.

• **In practice**, non-deterministic algorithms are not of big importance. Non-deterministic machines cannot be realized practically.

• Non-deterministic algorithms find use in **theoretical computer science**, in order to estimate the complexity of problem areas or to describe **quantum computers**.

http://de.wikipedia.org/wiki/Quantencomputer

A **quantum computer** is any device for **computation** that makes direct use of distinctively **quantum mechanical** phenomena, such as **superposition** and **entanglement**, to perform operations on data. In a classical (or conventional) computer, the amount of **data** is measured by bits; in a quantum computer, it is measured by **qubits**. The basic principle of quantum computation is that the quantum properties of particles can be used to represent and structure data, and that quantum mechanisms can be devised and built to perform **operations** with this data.
Recursivity

An algorithm is considered *recursive* when it uses itself again.

They are realised as follows:

**example:**
Calculate the factorial n!

```c
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return factorial(n-1)*n;
}
```

**Recursion anchor** (without reference to itself)
If the recursion anchor is reached, the result will be immediately returned.

factorial(0) = 1

**Recursion step** (with self reference)
This defines the function with help of a recursive call.

factorial(n) = factorial(n-1) * n

*Recursion anchor: Contains the aborting conditions of the recursion.*
There are 2 types of recursion:

**Direct Recursion:**
An algorithm calls itself again.

*example:*
calculate the factorial \( n! \)
factorial(0) = 1
factorial(n) = factorial(n-1) * n

**Indirect Recursion:**
Multiple different algorithms alternatively call each other.

*example:*
Test on even or odd numbers

\[
\begin{align*}
even(0) & = \text{true} \\
even(n) & = \text{odd}(n-1) \\
odd(0) & = \text{false} \\
odd(n) & = \text{even}(n-1)
\end{align*}
\]

The „even“ function calls the „odd“ function. In this call, the parameter is decremented by 1. The functions alternately call each other until the parameter is decremented to 0.
Everyday Recursion:
- Mirror image between 2 mirrors
- Audio- or Video return coupling

Mathematical examples:
- Factorial functions
  \[0! = 1\]
  \[n! = n \times (n - 1)\]
- Fibonacci-order \(fib(0), fib(1), fib(2), \ldots\)
  \[fib(0) = 0\]
  \[fib(1) = 1\]
  \[fib(n) = fib(n-1) + fib(n-2)\]
Recursivity (4)

- Recursion is a popular solution strategy, since it allows to formulate **complex problems** through a **small, „elegant“ description**.
- Recursive algorithms are frequently **shorter** than non-recursive algorithms, but are **not always more efficient**.
- **Each iterative (= loop based) algorithm can be expressed as a recursive algorithm.** On the other hand, an arbitrary recursive algorithm can **not be transferred** into an iterative algorithm **without further algorithms**.

```c
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return factorial(n-1)*n;
}
```

```c
int factorial(int n) {
    int a = 1; int i = 0;
    while (i < n) {
        ++i;
        a = a*i;
    }
    return a;
}
```
Parallelism

An algorithm is considered **parallel** if it is divided in sub-tasks that can be simultaneously run by different processors.

- A single processor allows only the **strict sequential execution** of elementary operations.
- Multiple processors allow for the **execution of different parts at the same time** in parallel algorithms.

<table>
<thead>
<tr>
<th>sequential</th>
<th>parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor 1:</strong></td>
<td><strong>Processor 1:</strong></td>
</tr>
<tr>
<td>( x := 3; )</td>
<td>( x := 3; )</td>
</tr>
<tr>
<td>( y := 4; )</td>
<td></td>
</tr>
</tbody>
</table>
• **Data dependence**
  The simultaneous execution of operations that use the same data leads to undefined results.

**Example:**
Different bookings in a bank can be performed by using different processors at the same time:

**Algorithms** for single bookings:

1. Read account balance $S$ from account $K$
2. Add the amount $B$ to the read-in account
3. insert $X$ as the new account balance of $K$

Both processors return different account balances. That means, depending on the execution time, one booking is lost.

<table>
<thead>
<tr>
<th>Processor 1:</th>
<th>Processor 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K = 3089; B = 1000$</td>
<td>$K = 3089; B = 5000$</td>
</tr>
<tr>
<td>$X := S(K);$  (1.)</td>
<td>$X := S(K);$  (1.)</td>
</tr>
<tr>
<td>$X := X + B;$  (2.)</td>
<td>$X := X + B;$  (2.)</td>
</tr>
<tr>
<td>$S := X;$  (3.)</td>
<td>$S := X;$  (3.)</td>
</tr>
</tbody>
</table>

[Algo04]
Parallelism: Limitations (2)

- If the dependencies of the parallel algorithms on different processors cannot be avoided, a **deadlock** can occur. The processors then wait respectively for a calculation result or for a resource that the other processor must provide.

- Each parallel algorithm can also be run sequentially. Computer science is today still unclear of whether or not all algorithms are able to be parallelized or not.

[Algo04]
Example for Deadlock – Dining Philosophers (1)

- 5 philosophers are sitting at a round table. Each one is alternatively eating and thinking. When a philosopher is hungry, he grabs the two forks (on his left and on his right) and starts eating. After he finishes eating, he puts the forks back on the table and continues his thinking (Dijkstra 1965).

- Abstract program for the philosophers:

```plaintext
repeat(
    think
    get right fork
    get left fork
    eat
    release right fork
    release left fork
)```
Possible dead-lock:

For all processes $i \in \{1 \ldots 5\}$ we have:

process $i$ has the right fork and waits for the left fork

If all philosophers grab the right fork simultaneously, no one can grab his left fork and waits. \textbf{\large Deadlock}
Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation
4. Data Structures
5. Algorithms
   5.1. Introduction
   5.2. Characteristics
   5.3. Complexity
   5.4. Design Methods
   5.5. Examples of Algorithms
• 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.
Introduction: Example

Example:
Problem area: add all numbers between 1 and 100.

Algorithm 1

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

Algorithm 2

```java
int sum = 0;
for (int i = 1; i <= 50; i++)
{
    sum = sum + i;
    sum = sum + (101 - i);
}
```

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 durations 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:

\[ t_{\text{assignment}} = 1 \mu s \quad t_{\text{Addition}} = 2 \mu s \quad t_{\text{Subtraction}} = 2 \mu s \quad t_{\text{equals}} = 4 \mu s \]

Algorithm 1 calculation time/lines

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

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 \(n = 100\).

Algorithm 2

```c
int sum = 0;
for (i = 1; i <= 50; i++)
{
    sum = sum + i;
    sum = sum + (101 - i);
}
```
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:**

```plaintext
Algorithm 1

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

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

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 $g(n)$.
(c) \( \Theta(g) \coloneqq O(g) \cap \Omega(g) \).

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.

\[
\Theta(g) \coloneqq \left\{ f \in P \mid \exists n_0 \in \mathbb{N} \ \exists c > 0 \ \forall n > n_0 : |f(n)| = c \cdot |g(n)| \right\}
\]

\( O(g) \), \( \Omega(g) \) and \( \Theta(g) \) thus describe sets of functions which for large \( n \) grow maximally or minimally or exactly as strong as the function \( g \).
Complexity: Examples to \( O, \Omega, \Theta \) Functions (1)

- Informal: \( g \) is "of order \( f \)" (notation \( O(f) \)), if:

- For an \( n \) big enough, \( 0 \leq g(n) \leq c \cdot f(n) \), for \( c > 0 \) constant
• With $f \in \Theta(g(n))$ it is expressed that $f(n)$ grows \textit{exactly} as fast as $g(n)$. 

\[ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \]
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):

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