2 Algorithmics

2.1 The Science of Algorithms

The *algorithm* is a fundamental notion to mathematics and informatics; it was created far before the invention of modern computers. Originally, an algorithm referred to a procedure of arithmetic operations on decimal numbers; later the same term began to be used to designate any series of actions that leads to a solution for a problem. In the field of informatics, an algorithm is understood to be the exact and finite list of instructions that defines the content and the order of enforcement actions, which are done by the executor on certain objects (source and in-between data sets) in order to obtain the expected result (final data sets).

Any algorithm depends on the executor; the description of the algorithm is performed using the executor's commands; objects which can be acted on by executor must belong to its environment, so that the input and output data of the algorithm must belong to the environment of a particular executor. The meaning of the word algorithm is similar to the meaning of words such as rule, method, technique or the way. In contrast to rule and methods, it is necessary that an algorithm have the following characteristics:

- *Discreteness* (discontinuity). An algorithm consists of a limited number of finite actions (steps); only after finishing the current step, the executor can proceed to the next step. The executor identifies each successive action solely based on statements recorded in the algorithm; such instruction is called a command.
- *Determinacy*. The way to solve the problem is unequivocally defined as a sequence of steps; this means that the algorithm used several times for the same input data will always result in the same set of output data.
- *Understandability*. An algorithm should not contain ambiguous instructions and prescriptions; the executor should not undertake any independent decisions.
- *Effectiveness*. If each step of an algorithm is executed precisely, the computation should be completed in a real time by delivering a solution to the problem; one possible solution is no result (an empty set of result data).
- *Mass character*. An algorithm should work correctly for some types of problems (not for a single problem); an algorithm's usefulness area includes a number of tasks belonging to the defined category of the problem.

There is a fundamental difference between executing and developing algorithms. To execute an algorithm, you have to have an executor (a performing machine), for which the algorithm has been developed and saved. To develop the algorithm, you have to have a storage medium, onto which the contents of the algorithm can be saved. There are several indirect forms of describing the same algorithm; you can describe it:

- In text form, using a natural language or a specially defined algorithmic language;
- In the graphic form, e.g. using a flow chart;
- In analytical form as a sequence of formulas;
- In the form of a computer program, more precisely, in a programming language.

Regardless of the form of representation, algorithms can be translated into common control and data structures provided by most high-level programming languages. Different forms of presentation of algorithms are preferred to precisely analyze the temporal and spatial requirements. Regardless of the form of description, their structure comprises steps connected into serial-chained, branched or cyclic fragments. Edsger W. Dijkstra has shown that these three structural units are the only ones needed to write any algorithm.

The science of algorithms (sometimes called *algorithmics*) is classified as a branch of computer science (in a historical perspective it was a branch of cybernetics) and occurs as a sector in the majority of natural sciences, economics and technology. The algorithmization or the art of building algorithms is called *algorithm design*; the result of algorithmization is a procedure (defined process), which solves efficiently a class of problem. Also within the scope of algorithmics, there is the study of the difficulty of solved problems; *algorithmic complexity theory* deals with this. A branch of algorithmics, called *algorithm analysis*, studies the properties of solved problem. Analysis defines resources needed by the algorithm to solve this problem.

2.1.1 Algorithm Design

From a practical point of view, an algorithm is a set of steps to be adopted by the computer code to reach specific informational goals; it is based on the idea of a solution to a problem. Therefore, the development of algorithms is the most important structural component of programming; the algorithm should not depend on the syntax of programming languages and the specifics of a particular computer, to be reusable. One can say that a program is a specific implementation of an algorithm, when the algorithm is the idea of the program. In a certain sense, the creation of algorithms is not engineering, because this activity contains elements of art; nevertheless, there are several different engineering approaches for algorithm design [14]. Sometimes, this general approaches are called *algorithmic paradigms*; they were formulated to constructing of efficient solutions to solving a broad range of diverse problems.

Operational Approach

Approaches and requirements for the development of algorithms have changed significantly during the evolution of computers. During the first generations, when computer time was expensive, and their ability was modest in terms of today's

achievements, the basic requirement for algorithms was narrowly understood as their effectiveness. The criteria were:

- Use the smallest number of memory cells when the program is executed;
- Achieve minimum execution time or the minimum number of operations.

In this case, the processor has executed the program commands nearly directly; the most frequent commands were an assignment statement, simple arithmetic operations, comparisons of numbers, unconditional and conditional jumps, subroutine calls. Such an approach to the creation of algorithms and to programming, focused on an operation directly executed by a computer is called an *operational approach*. Let us consider the basic steps of algorithms that are performed by a computer assuming the operational approach.

An *assignment statement* copies the value into a variable memory cell; this cell may belong to the main computer's memory or be one of the processor's registers. After an assignment, specified value is stored in the memory cell, where it is located until it will be replaced by another assignment. The memory cell, which houses the value, is indicated in the computer program by a name (identifier). As the variables and their values can be of different types, and values of these types are coded and represented in computer memory in different ways, they must match each other.

A set of *simple arithmetic operations* allows us to record arithmetic expressions using the numeric constants and variable names.

Comparison of numbers operations are actually reduced to the determination of the sign of the difference, which is displayed by a special memory (flag of the result) of a computing device and can be used in the performance of *conditional jumps* between the step of an algorithm. A conditional jump changes the order of command execution depending on some condition, most often the conditions of a comparison of the numeric types. In contrast, an *unconditional jump* changes the order of the commands independent from any conditions.

A *subroutine call* operation interrupt the normal order of execution steps and jumps into a separate sequence of program instructions (steps) to perform a specific task; this separate sequence is called subroutine and is packaged as a unit of code.
The use of an operational approach provokes certain drawbacks in the resulting code; the misuse of conditional and unconditional transitions often leads to a confusing structure of the program. A large number of jumps in combination with treatments (to increase the efficiency of the code) lead to the fact that the code may become incomprehensible, very difficult to develop and to maintain.

Structural Approach

Since the mid-1960s, computer professionals have obtained a deeper understanding of the role of subroutines as a means of abstraction and as units of code [15]. New programming languages have been developed to support a variety of mechanisms of parameter transmission. These have laid the foundation of *structural* and *procedural*

programming. Structural programming is a software development methodology, based on the idea of program architecture as a hierarchical structure of units or blocks (this idea was formulated by Edsger W. Dijkstra and Niklaus Wirth). This new programming paradigm was able to:

- Provide the programming discipline that programmers impose themselves in the process of developing of software;
- Improve the understandability of programs;
- Increase the effectiveness of programs;
- Improve the reliability of programs;
- Reduce the time and cost of software elaboration.

The structural approach methodology enabled the development of large and complex software systems. At the same time, it gave birth to structure mechanisms for algorithms and program codes, and a control mechanism for proving the correctness of calculations. A routines mechanism was implemented in form of procedures and functions, which are powerful programming tools.

There are four basic principles of structural methodology:

1. *Formalization* of the development process guarantees the adherence to a strict methodological approach; usually, programming should be engineering, not art.
2. The hierarchy of levels of *abstraction* requires building an algorithm divided into units, which differ in the degree of detail and approximation to the problem being solved.
3. In accordance with the maxim "*Divide and conquer*", the splitting of a whole problem into separate sub-problems, allowing the independent creation of smaller algorithms is a method for complex projects. The fragments of solution code can be compiling, debugging and testing separate as well.
4. The *hierarchical ordering* (hierarchical structuring) should cover relationships between units and modules of software package; this means respect for the hierarchical approach up to the largest modules of the software system.

A structural approach can be applied step by step, detailing of parts of an algorithm, until they are quite simple and easily programmable. Another way is the developing of lower-level units, and further combining them into units with higher levels of abstraction. In practice, both methods are used.

Modular Approach

In computer programming, a module is a set of related subroutines together with the data that these subroutines are treated. The terminology of different programming paradigms may provide another name for subroutine; it may be called a subprogram, a routine, a procedure, a function or a method. Gradually, software engineers formed the concept of a *module* as a mechanism of abstraction; a dedicated syntax was developed for modules to be used on par with subprograms. The construction of modules hides

the implementation details of subprograms, as opposed to subroutines, because they are under the effect of global variables. The prohibition of access to data from outside the module has a positive effect on a program; it prevents accidental changes and therefore a violation of the program. To ensure the cooperation of modules, a programmer needs only to consider the construction of an interface and a style of interaction for the designated modules in the whole program.

Any subroutine or group of subroutines can constitute the module: the ability to determine the structure and functions of the program modules depends on the qualification and the level of training of the programmer; in larger projects such decisions are taken by a more experienced specialist, called *software architect*. According to recommendations, a module should implement only one aspect of the desired functionality. Module code should have a header, which will include comments explaining its assignment, the assignment of variables passed to the module and out of it, the names of modules that call it and modules that are called from it.

Not all of the popular programming languages such as Pascal, C, C++, and PHP originally had mechanisms to support modular programming. It is worth noting that a modular approach can be performed even if the programming language lacks the explicit support capabilities for named modules. Modules can be based on data structures, function, libraries, classes, services, and other program units that implement desired functionality and provide an interface to it. Modularity is often a means to simplify the process of software design and to distribute task between developers.

The choice of a design approach is an important aspect of algorithm production. More than one technique could be valid for a specific problem; frequently an algorithm constructed by a certain approach is definitely better than alternative solutions.

2.1.2 Algorithmic Complexity Theory

Many algorithms are easy to understand, but there are some which are very difficult to follow. However, when professionals argue about the complexity of algorithms, they usually have something else in mind. When an algorithm is implemented as a program and the program is executed, it consumes computational resources; the most important resources are processor time and random access memory (RAM) space. These two parameters characterize the complexity of an algorithm in the conventional sense today. The time and memory consumed in the solution of the problem, are called the *time complexity* and the *space complexity*.

The development of industrial technology has led to cheap and compact RAM. Nowadays, RAM is not a critical resource; generally, there is enough memory to solve a problem. Therefore, most often the complexity of the algorithm should be understood as time complexity. Ordinary time T (seconds) is not suitable to be the measure for the complexity of the algorithms; the same program at the same input data may be performed in different time on different hardware platforms. It is easy to