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 understand that the increase in the number of input data will always cause an increase in the execution time of the program. This is why the time complexity is particularly important for applications that provide an interactive mode of the program, or for complex control tasks in real time.

A time complexity should not be dependent on the technical specification of a computer. It should be measured in relative units; typically, the number of performed operations estimates the time complexity. Here are some examples of basic operations: one arithmetic operation, one assignment, one test (e.g., x>0), one reading or writing of a primitive type. However, the number of elementary operations is often a useless measure for algorithm complexity; not always is clear which operations should be considered as elementary. In addition, different operations may require for their performance a different amount of time, and furthermore, the transfer of operations used in the description of the algorithm into the operations performed by a computer depends on the properties of the compiler and on such unpredictable factors as programmer skill. The assertion that a certain algorithm requires *N* operations does not mean anything in practice. Practitioners are most interested in the answer to the question of how the program execution time will increase with the increase in the number of input data.

Consider an algorithm that adds the *n* numbers $x_1, x_2, ..., x_n$. Here it is:

```
Sum = 0;
for i=1 to n do
  { Sum = Sum + x[i] };
```

If the numbers are not large and their sum can be calculated using standard addition, it is possible to measure the input data volume as the number *n* of summands. Assuming that the basic operation is addition, we can conclude that the complexity of the algorithm is equal to n. The time for addition is linear in the number of items (summands); if the problem size doubles, the number of operations also doubles.
Consider another example – a sequential search algorithm. It is looking for an element *y* in the *n*-elements set *x*, scanning sequentially all the elements of the set $x_1, x_2, ..., x_n$. The search result is the value of the variable *k;* if the desired element is found $y = x_i$, it is equal *i*, otherwise $k = n + 1$.

```
k = n + 1;
for i = 1 to n do
  { if y = x[i] then
    { k = i }
  };
```

Here, the basic operation is a conditional statement *if*. The number of comparisons made in the best case will be equal to *1*, and in the worst case *n*. It may be noted that on average there will be about n/2 basic operations made. In practice, the two most common measures of complexity of algorithms that are used are the complexity in the worse case and the average complexity.

Here are a few comments on the practical treatment of the complexity of algorithms. Firstly, a programmer should often find a compromise between computation accuracy and execution time; generally, the increase of accuracy requires an increase in time (an increase of time complexity of algorithm). Secondly, spatial complexity of the program becomes critical when the volume of data to be processed reaches the limit of the amount of RAM of a computer. On modern computers the intensity of this problem is reduced both by increasing the volume of random access memory, and by the efficient use of multilevel systems of storage devices (swapping, caching and buffering processes). Thirdly, the time complexity of the algorithm may be dependent both on the number of data and of their values. If we denote the value of the time complexity of the algorithm α by symbol T_α, and designate by V a numerical parameter that characterizes the original data, the time complexity can be represented formally as a function $T_\alpha(V)$. Selecting V will depend on the problem or the type of algorithm used for solving this problem. Here is an example of calculating the factorial of a positive integer *n*:

```
f = 1;
for i = 2 to n do
  { f = f * i };
  factorial =  f;
```

In this case, it can be said that the time complexity depends linearly on the data parameter *n* – the value of the factorial function argument; here the basic operation is multiplication.

In computer science, the characteristic of an algorithm which relates to the amount of resources used by the algorithm, is called *algorithmic efficiency*; it can be considered as an equivalent to engineering productivity for repeating or continuous processes; for maximum efficiency, one should to minimize resource usage. However, processor time and memory space as different resources cannot be compared directly, so which of the algorithms when compared is considered more efficient often depends on which measure of efficiency is being considered as the more important. In practice, efficiency of an algorithm or piece of code covers not only CPU time usage and RAM usage; it may include the usage of lots of resources, e.g. disks, network.

Computational Complexity Theory

In a wider perspective, the concept of the complexity of the algorithm can be used to characterize computational problems, namely the parametric evaluation of their intrinsic difficulty. A branch of the theory of computation, called *computational complexity theory*, focuses on classifying problems according to their difficulty (by means of computational complexity). This work assumes that the more difficult problem will have the more complex algorithm describing its solution. Here is a five-step process to solving computational problems:

1. Build a clear declaration of the problem.
2. Find a mathematical and logical model for the problem; usually, it is associated with making some simplifications to shift from the problem to the model.
3. Using the model, settle upon a method for solving the problem on a computer; remember what a computer can and cannot do.
4. Implement the method (carry it out with a computer).
5. Assess the implementation; estimate obtained answers to see if they make sense, and try to identify errors or unjustified assumptions. In case of a relatively large disparity between the correct and the obtained result, reassess the problem and try the process again.

The essential parts of the problem solving process should include understanding the problem and good intuition. Informaticians should deal with multidisciplinary problems; as computer professionals, they must be able to take not theoretical but real problems that involve specific information from the different areas (physics, chemistry, biology, psychology, politics, etc.) and find solutions, which will be finally rendered as instructions operating on data. Obviously, computational complexity theory studies also its own problems, but there are primary problems, which engineers and scientists from other disciplines might be interested in. Here are some examples of such problems:

– A decision problem, which is one that can be formulated as a polar question (i.e., with "yes" or "no" answer) in some formal system.
– A search problem, which is not only about existence of solution but about finding the actual solution as well.
– Counting problems, which are requests for the number of solutions of a given instance.
– An optimization problem, which has more solutions but requires the best one.

The notion of efficiency is one of the most important things in computational complexity theory. According to convention, computation is efficient if its execution time is bounded by some polynomial function of the size of input; it is said that a calculating machine runs in polynomial time. This is possible when the algorithm uses some vital knowledge about the structure of the problem; it will result in proceeding with much better efficiency than "brute force", which corresponds to the exponential time.

The theory of algorithms introduces *classes of complexity*; these are sets of calculation tasks characterized by approximately the same height of resource requirements. In every class, there is a category of problems, which are most difficult; this means that any problem in the class can be mapped to those problems. Such difficult problems are called *complete problems* for this class; the most famous are NP-complete problems. Complete problems are used to prove the equality of classes; instead of comparing sets of tasks just compare the complete problems belonging to these sets. The previously mentioned NP-complete problems, in some sense, form a subset of the typical problems in NP class; if for some of them a polynomial fast solution algorithm is found, then any other task in the NP class can be solved in the same fast tempo. Here, NP class can be defined as containing tasks that can be solved in real time on the non-deterministic Turing machine. Unfortunately, a more detailed presentation of this issue requires a very specialized knowledge of mathematics and computer science especially in the area of theoretical modeling of computer systems, which is used in automata theory.

2.1.3 Algorithm Analysis

Analysis of algorithms is a branch of applied informatics that provides tools to examine the efficiency of different methods of problem solutions. The analysis of the algorithm is intended to discover its properties, which enable the evaluation of its suitability for solving various problems; by analyzing different algorithms for the same application area, they can be compared with each other (ranking of algorithms). The practical goal of *algorithm analysis* is to foresee the efficiency of diverse algorithms in order to conduct a software project. In addition, an analysis of the algorithm may be the reason for its reengineering and improvements to simplify a solution, shorten code, and improve its readability.

A naive approach to comparison of algorithms suggests implementing these algorithms in the programming language, compiling codes and running them to compare their time requirements. In fact, this approach suggests comparing the programs instead of the algorithms; such an approach has some difficulties. Firstly, implementations are sensitive to programming style that may obscure the matter of which algorithm is more effective. Secondly, the efficiency of the algorithms will be dependent on a particular computer. Lastly, the analysis will be dependent on specific data used throughout the runtime. That is why, to analyze algorithms, one should utilize mathematical techniques that will analyze algorithms independently of detailed implementations, hardware platform, or data. This explanation has repeated evidence that was previously mentioned while explaining the complexity of algorithms. This is not a coincidence, because a professional approach to the analysis of algorithms provides an assessment of their complexity, effectiveness and efficiency through objective and independent determination of their categories or class.

Indeed, software engineers measure such parameters as the actual time and space requirements of a program, the frequency and the duration of subroutine calls, or even the usage of specified instructions. The results of these measurements serve to aid program (not algorithm) optimization; this activity is called *program profiling* and is a form of *dynamic program analysis*.

According to the strict language of mathematics, informaticians express the complexity of algorithms using big-*O* notation [16] that is a symbolism used in mathematics to describe the asymptotic behavior of functions. Generally, it tells us how fast a function grows. The letter *O* is used because the rate of growth of a function is also called its *order*. Formally, a function $T_\alpha(N)$ is $O(F(N))$ if for some constant c and for all values of N greater than some value n_0: $T_\alpha(N) \le c{*}F(N)$. For a problem of size N a constant-time method is order 1: $O(1)$ and a linear-time method is order N: $O(N)$. Of course, $T(N)$ is the precise complexity of algorithm as a function of the problem size N, and selected *growth function* $F(N)$ is an upper limit of that complexity. Algorithms can be categorized based on their efficiency; the following categories are often used in order of decrease of efficiency and increase of complexity:

Complexity	**Growth function**	**Comments**
O(1)	Constant	It is irrespective of size of data
O(log n)	Logarithmic	Binary search in a sorted array
O($\log^2$ n)	Log-squared	
O(n)	Linear	Finding an item in an unsorted array
O(n log n)	Linearithmic	Popular sorting algorithms
O(n^2)	Quadratic	Polynomial time algorithms
O(n^3)	Cubic	
O(a^n)	Exponential	Generally, brute-force algorithms
O(n!)	Factorial	

Let us see how to analyze an example algorithm. In the beginning, we need to count the number of important operations in a particular solution to assess its efficiency. Each operation in an algorithm has a cost (take a certain amount of time) c_i. The total cost of a sequence of operations is $C=c_1+c_2+\ldots+c_n$; loop statements cost should be calculated from the cost of a block of operations multiple times, for all iterations:

Nr	**Instruction**	**Cost**
00	// Sum of sequential numbers from 1 to n	
01	i = 1;	c_1
02	sum = 0;	c_2
03	while (i <= n) {	$c_3\ (n+1)$
04	i = i + 1;	$c_4\ n$
05	sum = sum + i;	$c_5\ n$
06	}	

Total cost $C = (c_3 + c_4 + c_5)\ n + (c_1 + c_2 + c_3)$

It is easy to notice that the time required for this algorithm is proportional to n. If a nested loop is added to this algorithm to change the logic of sum counting, the time required for a new algorithm will be proportional to n^2:

Nr	Instruction	Cost
00	// Double sum of sequential numbers from 1 to n	
01	i = 1;	c_1
02	sum = 0;	c_2
03	while (i <= n) {	$c_3\ (n + 1)$
04	j = 1;	$c_4\ n$
05	while (j <= n) {	$c_5\ (n + 1)$
06	sum = sum + i;	$c_6\ n^2$
07	j = j + 1;	$c_7\ n^2$
08	}	
09	i = i + 1;	$c_8\ n$
10	} //	

Total cost $C = (c_6 + c_7)\ n^2 + (c_3 + c_4 + c_5 + c_8)\ n + (c_1+c_2+c_3+c_5)$

When determining the cost of the operation, it should be taken into account that most arithmetic and indexing operations are constant time (runtime does not depend on the magnitude of the operands). Addition and subtraction are the shortest time operations, multiplication usually takes longer, and division takes even longer than they do. Sometimes the operands are very large integers; in such a situation, the runtime increases with the number of digits. If the algorithm contains conditional statement (e.g., *if condition then s1 else s2*), the cost of this fragment should be counted by adding the condition test running time and the larger running time of *s1* and *s2*. In general, the built-in functions tend to be faster than user-defined functions because they are implemented more efficiently.

After counting the number of important operations, we should express the efficiency of the complete algorithm using growth functions. It is important to know how rapidly the algorithm's time requirement grows as a function of the problem size. The efficiency of two algorithms can be compared via juxtaposition of their growth rates. Typically, two algorithms with the same growth rate are dissimilar only according to constant coefficients, but the different growth rate explicitly indicates a good algorithm and a bad algorithm. Informaticians who care about performance of algorithms often have an aversion to this kind of conclusion; sometimes the coefficients and the non-leading conditions make a real difference (e.g., the details of the hardware, the programming language, and the characteristics of the input). Furthermore, for small problems, asymptotic approximation is irrelevant; it is useful at least for big computational problems.

2.2 Data Science (Datalogy)

Data are the basis for any analytical or explorative business. Data science is a new interdisciplinary area in which informatics plays an essential role beside mathematics and statistics. Data science deals with large amounts of data and facilitates the acquisition of knowledge from such large collections. The morphology and semantic of terms *data science* and *datalogy* suggests that both can be used interchangeably. Historically, in connection with the central role of data in the construction of a computing system, Peter Naur has suggested the name datalogy for the whole of computer science. The design technique, data driven design, which was very popular in the last century, was evidence of the advantage of using this term. The rapid development of technology and the theory of relational databases also pointed to the dominant role of data in computer systems. In the new century, the vast change of the idea of large data centers into the new idea of distributed computing network systems caused the need of new data structures and new methods of data analysis. Currently, the demand for services related to the processing of huge data sets is increasing incredibly; professionals need theoretical support in this area – they need new models and methodologies as well.

2.2.1 Raw Data

We will call *raw data* (sometime, unprocessed data) every kind of data (e.g., commercial data, experimental data, statistical data), which have been collected in the process of observation, research, measuring, and monitoring, but have not been transformed or structured artificially (this means, they are not processed and behave like a natural structure). In order for such data to be available to computers, they must be digitized; a digital data source (generally) can be any of database, computer file, or even a live data feed like a network data stream. Digital data are stored on hard drives, on CD (DVD) or on flash memory; to be processed, they always should get to RAM. The data might be located on the same computer as the software that processes them, or on another computer (server) anywhere on a network.

Data does not magically come into view – somebody must have gathered it. Information and knowledge comes from data through processing them; the place where data are obtained from, is called the *data source*. The *original source*, which supplies *primary data*, is distinguished from the *secondary source*, which supplies secondary or *indirect data*. Usually, primary data are gathered (some time even developed or generated) by the informatics-analyst exclusively for the research project. The advantages of such data include the control over the process of data collection, the formal specification of data, and the dynamic management of data collection based on the previously stored values. Building an original data source requires a lot of time; very often in practice, the original source will be smaller than the secondary sources. Therefore, analysts often use secondary data sources; they contain data that has previously been gathered by somebody else and/or for some other purpose. One can obtain from secondary sources a huge series of data; compared to primary data gathering this approach is usually less expensive. The disadvantage of indirect data may be its lower quality. It may be inaccurate, not current or biased.

To be workable, data sources should hold not only raw data themselves but also information on *metadata* and *connectivity settings* such as the server address, table name, login, etc. Metadata is a data about data; there are catalogs, directories, registry, which contain the information about the composition of data, content, status, source, location, quality, format and presentation, access conditions, the acquisition and use, copyright, property and related rights to data. Metadata are the information that describes the content of the database. Metadata inform users when a piece of data has been updated for the last time, its format and what is supposed to apply it. In a practical sense, this information can serve as a reference for a user while working with the data source, and helps him to understand the meaning and context of data. Metadata are used to improve the quality of search results; a data source enables relatively complex operations for simultaneous search and filtering by informing the computer about the relationships that exist between its data elements. This approach (called *knowledge representation*) is in the interests of artificial intelligence and the semantic web. IT professionals call a structured metadata the *ontology* or the *schema* (for example, XML-schema). A metadata schema announces a group of terms, their definitions and relationships; the terms are often referred to as *elements*, *attributes* and *qualifiers*, the definitions make available the semantics of data elements. High-quality schema makes the data source understandable both for machine and for human through presentation of ontology.

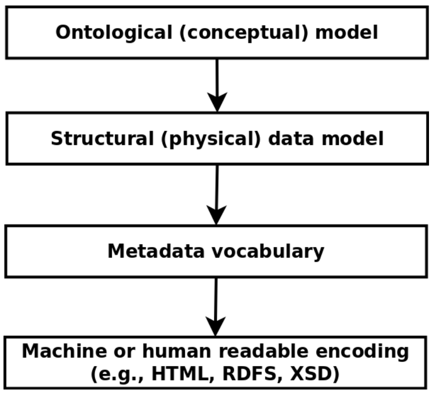
Ontological (conceptual) model
Structural (physical) data model
Metadata vocabulary
Machine or human readable encoding (e.g., HTML, RDFS, XSD)

Figure 4: The role of metadata in data set management.

Very often, the big volume of obtainable data, the variety of their formats and the high velocity of data actualization makes it more difficult to carry out large analytical projects. IT professionals use a special technical process, called *data integration*, to combine disparate sources into one significant and valuable data source. Data integration provides standardized access to data collected from heterogeneous and distributed data sources; it includes locating, monitoring, purifying, transformation and delivery of data from dissimilar data sources.

2.2.2 Data Structures

In a computer memory, all data are represented by binary codes, which cannot be distinguished directly. To recognize the different types of data (like text and numbers or like vectors and scalars), computer programmers use data type classification. Before using the data, a programmer should declare their types; after declaration, in order to ensure the faultless results of calculations, data types should be referenced and used properly. Data types serve to explicitly process and store data in a particular way; understanding data types allows programmers to design programs efficiently. Unfortunately, for historical reasons, such declarations and the list of possible data types are dependent on the programming language. However, there are some common notions and senses; all programmers make use of technical terms *primitive*, *built-in* and *compound* data types.

Some data types are primitive in the sense that they are basic and cannot be decomposed into simpler data types. From these basic types, one can define new compound types. Common examples of such basic data types are integer numbers, floating-point numbers, characters, pointers, and Booleans. Some categories of compound types are arrays, structures, unions, objects and alphanumeric strings. A built-in type is distinguished by the fact that the programming language (compiler) provides built-in support for it. In general, all primitive data should be supported by programming languages, not only when they fall within a group of built-in data types within a given programming language. For example, the alphanumeric string can be regarded as built-in data in some programming languages, in some others it is regarded as array of characters.

As there are only a few very important ways of organizing non-primitive data and data sets in computer memory or in storage, let us analyze them. The first approach is based on the declaration of *data structures* – the collection of elements (datum) of possibly different types. The architecture of a specific kind of data structure is matched to the solved problem, and the data items contained within such a structure are provided to be accessible and secure. The process of creating and using data structures is based on pointers – basic computer data that keep the memory addresses of structures; instead of handling the entire complicated data structure, it is often sufficient to perform operations only on these primitive pointers. In informatics, certain sorts of data structures that have similar behavior are represented by a mathematical model called an *abstract data type* (ADT), which is used by programmers to manage data during calculation. Data items have both a logical and a physical form – the meaning of the data item within an ADT, and the implementation of the data item within a data structure respectively.

The oldest data structures are *arrays*, they hold a specified number of homogeneous elements (data of the same data type); each element of the array is identified by an array index (pointer) and can be accessed individually. The entire advantage the arrays show in mass calculation, e.g. in loops. The most clear in a linear data structures, into which all elements have an order (each element has a predecessor and a successor); in the linear data structure one element is first, and one is last. The examples of *linear data structure* are a list (an ordered collection of nodes), a stack (last-in-first-out data structure), and a queue (first-in-first-out data structure). *Hierarchical data structures* are more complicated. They are organized like a tree, in which the first element (the root) can have more than one successor (the leave), and all leaves can have one or many successors as well. A completely disordered collection might seem to be a *graph data structure*; its nodes may have "many to many" relationships with other nodes of data sets, and there are no constraints on the numbers of predecessors or successors.

An alternative approach to the management of data items is provided by an object-oriented paradigm. In this approach, data are attributes of *object*, they can be represented by primitive or compound data types, or by another objects. Objects consists of attributes and data and are responsible for themselves – the attributes (data) have to know what state the object is in, the methods (operations and functions possible on this data) have to work on data properly. By packing of data and functions into a single component, data are encapsulated (closed in the separate memory area); that means, programmers can effectively manage them.

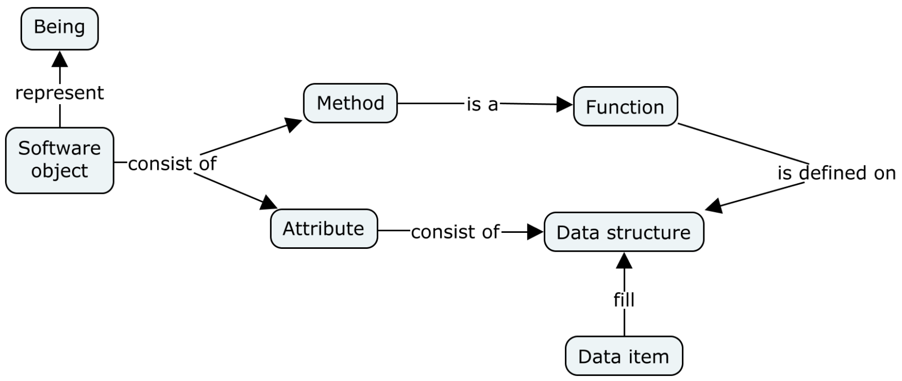

Figure 5: Software object idea (concept map).

Database specialists prefer a slightly different approach; they offer *records* (also called row or tuple) as a basic data structure for all storage medium, including the main memory of computers. Records are a single, implicitly structured complete set of information; the main goal of this specific data structure is the ease and speed of search and retrieval of data representing of the entity. Other database components are fields and files; a record is a collection of related fields, a file is a collection of records. The database is stored logically as a collection of files.

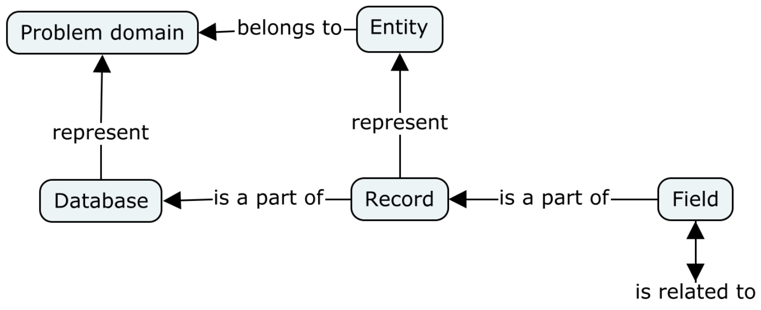

Figure 6: The record as a set of information (concept map).

2.2.3 Data Analysis (Data Analytics)

Data do not have their own meaning; data are facts (details), but the informatician-analyst has to see truth (regularity) behind facts by discovering rules hidden in the data. Even the greatest amount of the best quality data means nothing if they have not been analyzed. The primary activity of the analyst is to turn raw data into useful information. The aim of data analysis should be to search for the answers to questions that are asked by software engineers and domain specialists. Then the information and knowledge gained from the data will be useful in the context of the development of new software and will contribute to progress in the domains that are problematic for informaticians. It should be noted that the relatively new term *data analysis* is not successful, since the word "analysis" in mathematics has an established meaning and is the name of many of the classical sections (e.g., mathematical analysis, functional analysis, complex analysis, discrete analysis). In data analysis, there is not the study of the mathematical apparatus, which is based on some fundamental results. There is not a finite set of basic facts, from which follows how to solve the data problems especially that many such problems are needed to develop an individual mathematical apparatus. The science that studies raw data in order to draw conclusions about the information contained in them is called data analytics.

Data analysis is a field of informatics, engaged in the design and study of the most common computational algorithms (of course, based on mathematical and statistical methods) for extracting knowledge from raw data. In practice, data analyzing means transformation, filtering, mapping, and modeling of data in order to extract useful information and to aid decision-making. There are three traditional data analysis approaches: classical, exploratory, and Bayesian. They all deal with engineering and science problems. They gather big collections of relevant data; the difference lies in the sequence and focal point of their milestones. In the case of classical analysis, data collection is followed by modeling; on the next stage, the analysis is focused on the parameters of created model. For an exploratory analysis [17], data collection is followed directly by analysis. The goal of analysis is to arrive at a model, which would be appropriate. In the last case, that of a Bayesian analysis, data-independent distributions are imposed on the parameters of the selected model. The analysis consists of formally combining both the prior distribution on the parameters and the collected data, to draw conclusions about the model parameters. The sequences of activities shown in the table are performed in the context of these approaches. In addition, the following paragraphs provide more details of these approaches.

Table 2: Model of the data analysis process.

Data analysis	The sequence of activity
Classical	Problem → Data → Model → Analysis → Conclusions
Exploratory	Problem → Data → Analysis → Model → Conclusions
Bayesian	Problem → Data → Model → Prior distribution → Analysis → Conclusions

The classical approach is inherently quantitative and deals with mathematical models that are imposed on the data collections. Deterministic and probabilistic models which are built as a part of this approach can include regression analysis and analysis of variance, Student‘s t-test, chi-square tests, and F-tests. The outcomes of deterministic models are precisely determined through known relationships, and can be used to generate predicted values. Based on available data, probabilistic models are used to estimate the probability of a data related event or state occurring again. Research based on classical techniques is generally very sensitive, but depends on assumptions. Unfortunately, the proper assumptions may be unknown or impossible to verify.

The exploratory data analysis approach is based generally on graphical techniques, which allows the data set to suggest acceptable models (those that best match the data). For a visual assessment of the situation, histograms, scatter plots, box plots, probability plots, partial residual plots, and mean plots are created. Unfortunately, it follows that exploratory techniques may depend on subjective interpretation. Nevertheless, the advantage is that this approach makes use of all of the available data, without mapping the data into a few estimates as a classical approach does.

Bayesian data analysis is based on so-called subjective probability. It enables reasoning with hypotheses whose truth or falsity is uncertain. The difference between classical and Bayesian interpretation of probability plays an important role in practical statistics. For example, when comparing two hypotheses (models) on the same data, the theory of testing statistical hypotheses based on the classical interpretation, allows the rejection of a potentially adequate model. Bayesian techniques, in contrast, depending on the input data will give a posteriori probability chance to be adequate for each hypotheses-model. In fact, Bayesian theory allows the adaptation of existing probabilities to the newly obtained experimental data. It is useful for building intelligent information filters, such as a spam filter for email.

Besides the previous approaches for data analysis, there are a number of methods and styles tailored to the requirements of a particular sector of engineering and science. For example, biological and medical data analysis, social data analysis, business analytics and so on. Particularly well known for its extensive research is *business intelligence*. Most often, only final products fall under this concept – the software created to help managers to analyze information about the company and its environment. However, there is a wider understanding of term business intelligence as the methods and tools used for the transformation, storage, analysis, modeling, delivery and tracking of information for the period of work on tasks related to decision-making based on actual data. Business intelligence technology allows the analysis of large amounts of information, guiding the user's attention only to their effectiveness by simulating the outcome of various options for action, and tracking the results of the adoption of certain decisions. Data mining and data warehousing are typical tools in this area and are directly related to data analysis.

2.2.4 Data Mining

Nowadays, data are produced at a phenomenal rate. The continuous growth of the economy means that more and more data are generated by business transactions, scientific experiments, publishing, monitoring, etc.; more and more data is captured because of faster and cheaper data storage technologies, yet the capabilities of storages are limited. These trends lead to the phenomenon of the data flood. According to Moore's law, computer CPU speed doubles every 1.5 years, and total storage doubles twice as fast; in consequence, only a small part of gathered data will be looked at by a human. Society needs some kind of knowledge discovery in data [18] to make sense of data collected.

Data mining is the analytical process which is used to search large amounts of business or market related data (typically known as *big data*) for patterns. The final goal of data mining is the prediction or anticipation of live commercial data streams using those patterns. Data mining is associated generally with knowledge discovery in business. By uncovering hidden information, data mining performs a central role in the knowledge discovery process. The potential result of data mining is meta-information that may not be obvious when looking at raw data. It should be noted that the term data mining is a bit fuzzy in the public eye. Even experts cannot define the strict boundaries between data mining and exploratory data analysis or data-driven discovery because certain terms for similar activities are sometimes misleading. Here are several alternative names for data mining: data pattern analysis, data archeology, knowledge extraction, business intelligence, and knowledge discovery in databases (KDD).

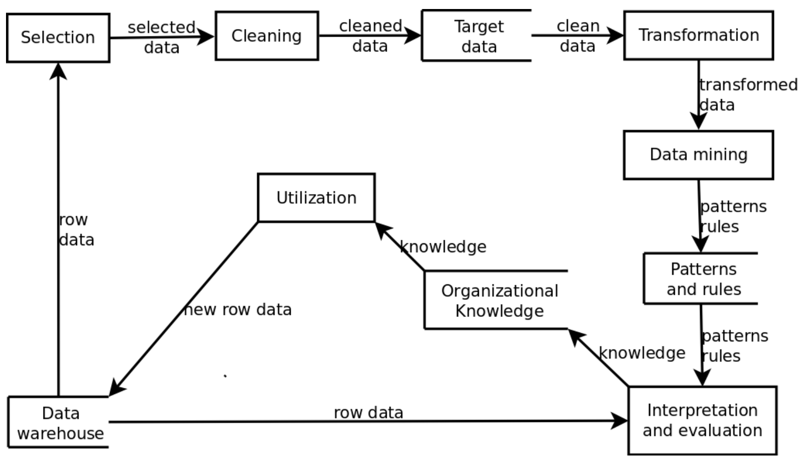

Figure 7: Knowledge discovery process (data flow diagram notation).

Here are the seven most important data mining tasks:
1. Visualize a data set to facilitate data regularity discovery by a human.
2. Clustering: finding natural explicit groups in data collection.
3. Classification: predicting an item class based on a learned method from pre-labeled instances.
4. Uncover association rules (in the form of if-then statements) that will support data relationships identification.
5. Link analysis: finding relationships between data based on uncovered earlier association rules.
6. Estimation: predicting a continuous data value based on historical records.
7. Deviation detection, this means finding irregular changes in data.

One can notice the similarity between database processing and data mining. Both activities are based on the formulation of queries to the collection of data. Data mining queries are not as well defined. In addition, data mining does not have precise-defined query languages. The result is that data mining processing cannot output the subset of a database, but only some fuzzy information e.g., about data clusters or about association rules between them. One important note, it makes sense to apply data mining techniques only for sufficiently large databases; patterns that have been found in small data collections may be accidental.

A giant amount of data like terabytes will disqualify traditional data analysis. Big data algorithms are highly scalable to handle such collections. High-dimensionality and high complexity of data are the reason to choose data mining processing before data analysis as well, especially in case of such dynamic data sources as data streams and sensor data. Contrary to expectation, many commercial and non-commercial organizations such as pharmacies, retail chains, international airports, and universities only hold large amounts of data and do nothing with it. These organizations will need data analysts specializing in data mining.

Some application areas for data mining solutions are advertising, bioinformatics, customer relationship management, database marketing, fraud detection, ecommerce, health care, investment, manufacturing, and process control. These are separate areas in engineering and science, where specialists deal with big or unstructured data. The most familiar is *text mining*.

Text Data Mining

Text data mining (also called text mining or *text analytics*) is the process of deriving information from collections of texts, e.g., from business or legal documents, newspaper or scientific articles and all kind of books. This is an area in artificial intelligence, the purpose of which is to obtain information from text sources based on machine learning and natural language processing. There is a complete similarity of goals between text data mining and data mining in approaches to information processing and applications. A difference appears only in the final methods that are used. Key groups of tasks for text data mining are text categorization, information extraction and information retrieval, processing changes in the collections of texts, as well as the development of tools for presenting information to the end user. The text is not a random collection of characters, so a text mining procedure should distinguish the words, phrases and sentences in special way to become aware of the meaning. These pieces of text are then encoded in the form of numeric variables, which are subjected to applied statistical methods and data mining in order to discover the relationship between them, and indirectly between the meanings hidden in the text.

Machine Learning

Machine learning is an area of artificial intelligence which is the study of algorithms that can learn from data. The main objective of machine learning is the practical application of such algorithms in automated systems so that they are able to know how to improve themselves by means of accumulated experience. Learning (the training of an algorithm) can be seen in this context as a concretization of the algorithm, i.e. an optimal selection of algorithm parameters called knowledge or skill. The criterion for this selection may be increasing efficiency, productivity, uptime and reduction of the costs of an automated system. There are a few approaches to training an automated system; e.g., training on precedents provide an inductive learning based on identifying patterns in empirical data gathered by an automated system. Alternatively, deductive learning involves formalized expert knowledge in the form of rules. Machine learning technologies contribute to the field of natural language processing, stock market analysis, computer vision, brain-machine interfaces, speech and handwriting recognition, robot locomotion and others.

The Use of Data Mining
One might become familiar with the areas of application of data mining from examples of big data sources. Here are some popular web based collections:
– The European union open data portal, http://open-data.europa.eu/en/data/
– The home of the U.S. Government's open data, http://data.gov
– A federal government website managed by the U.S. Department of Health & Human Services, https://www.healthdata.gov/
– A community-curated database of well-known people, places, and things, http://www.freebase.com/
– The New York Times articles, http://developer.nytimes.com/docs
– Amazon public data sets, http://aws.amazon.com/datasets
– Google Trends, http://www.google.com/trends/explore
– DBpedia, http://wiki.dbpedia.org

Data mining means also using specialized, complex computer software; there are many commercial and non-commercial programs for data miners. Here are some examples of such tools:

Name	App Description	URL
Cubist	The tool analyzes data and generates rule-based piecewise linear models – collections of rules, each with an associated linear expression for computing a target value	https://www.rulequest.com/
Mercer's Internal Labor Market Mapping Tool	The tool constructs classification models in the form of rules, which represent knowledge about relations hidden in data	http://www.imercer.com/
Magnum Opus	The data mining software tool for association discovery finds association rules providing competitive advantage by revealing underlying interactions between factors within the data.	http://www.giwebb.com/

Name	App Description	URL
Orange	This is an open source data visualization and analysis tool, which realize data mining through visual programming or Python scripting. It has components for machine learning, add-ons for bioinformatics and text mining, and it is packed with features for data analytics.	http://orange.biolab.si/
Weka	This is a collection of machine learning algorithms for data mining tasks.	http://sourceforge.net/projects/weka/
RapidMiner	It provides an integrated environment for machine learning, data mining, text mining, predictive analytics and business analytics.	https://rapidminer.com/
Apache Mahout	This is an Apache project to produce free implementations of distributed or otherwise scalable machine learning algorithms on the Hadoop platform.	https://mahout.apache.org/
KNIME	This is a user friendly, intelligible and comprehensive open-source data integration, processing, analysis, and exploration platform.	http://www.knime.org/
SCaVis	This is an environment for scientific computation, data analysis and data visualization designed for scientists, engineers and students. The program incorporates many open-source software packages into a coherent interface using the concept of dynamic scripting.	http://jwork.org/scavis/
Rattle	It presents statistical and visual summaries of data, transforms data into forms that can be readily modeled, builds both unsupervised and supervised models from the data, presents the performance of models graphically, and scores new datasets.	https://code.google.com/p/rattle/
TANAGRA	This is free data mining software for academic and research purposes. It proposes several data mining methods from exploratory data analysis, statistical learning, machine learning and databases area.	http://eric.univ-lyon2.fr/~ricco/tanagra/