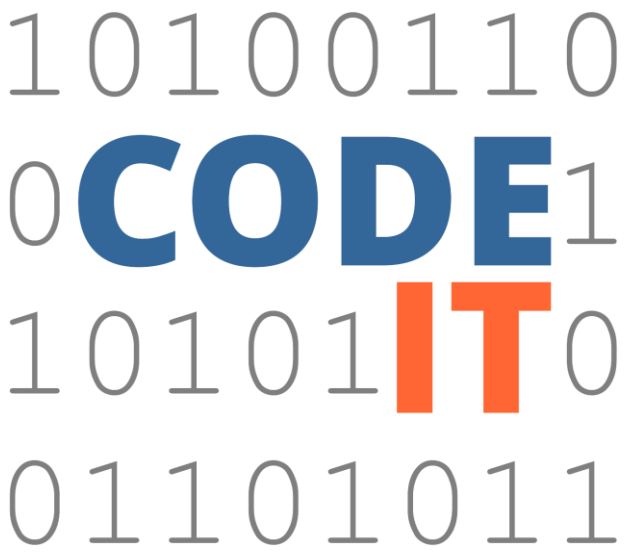


Algorithmic and Programming

Training materials for Teachers

MARIA CHRISTODOULOU

ELŻBIETA SZCZYGIEŁ

ŁUKASZ KŁAPA

WOJCIECH KOLARZ

Erasmus+

This project has been funded with support from the European Commission. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Algorithmic and Programming

Training materials for Teachers

MARIA CHRISTODOULOU

ELŻBIETA SZCZYGIEŁ

ŁUKASZ KŁAPA

WOJCIECH KOLARZ

Krosno, 2018

The Authors:

Maria Christodoulou, Elżbieta Szczygieł, Łukasz Kłapa, Wojciech Kolarz

Scientific reviewer: *Marek Sobolewski PhD, Rzeszow University of Technology*

Publishing house:

P.T.E.A. Wszechnica Sp. z o.o.
ul. Rzeszowska 10,
38-404 Krosno
Phone: +48 13 436 57 57
https://wszechnica.com/

Krosno, 2018

ISBN 978-83-951529-0-0

Table of contents

Instead of the introduction

Ability to using algorithmic and programming is recognized by the European authorities as one of the important, nowadays skill forming part of "digital competence" which is one from eight key competences. In EURYDICE report (2012) a following statement has been made: "The need to improve the quality and relevance of the skills and competences with which young Europeans leave school has been recognised at EU and national level. The urgency of addressing this issue is further underlined by the current situation in which Europe faces high youth unemployment and, in some cases, serious skills mismatches (…). The European Policy Network on the Implementation of the Key Competences (KeyCoNet) analyses emerging initiatives for the development of the key competences (…). One of them relates to the need for a more strategic approach in supporting the key competences approach at school. A second one is related to the efforts to enhance the status of the transversal competences (digital, civic and entrepreneurship) as compared to the traditional subject-based competences."

Publication entitled: *Algorithmic and Programming - Training materials for Teachers. Algorithmic. Programming. Didactics.* meets these recommendations. The main aim of the publication is presenting the teachers of an idea of algorithmic and programming along with their practical application in didactics. The publication is the first Intellectual Output of the project entitled **"CodeIT: Enhancing Teachers' professional development through algorithmic and programming"**, which is realised by the international consortium consist of six partners from five countries: P.T.E.A. Wszechnica Sp. z o.o. (Krosno, Poland), M.K. INNOVATIONS LTD (Nicosia, Cyprus), Danmar Computers Sp. z o.o. (Rzeszów, Poland), Istituto Superiore E. Mattei (Fiorenzuola d'Arda, Italy), Liceul Pedagogic "Mircea Scarlat" Alexandria (Alexandria, Romania) and Kekavas vidusskola (Kekava, Latvia). The project is realised under the frame of Erasmus+, Strategic Partnership Programme.

The main aim of the project is to help teachers enhance their professional development by raising programming competences through the development of innovative resources. Primary target group are non-IT teachers from elementary schools (grades 4 and higher) and gymnasiums (lower-secondary schools) with special attention to teachers of Chemistry, Geography, Math and Physics. Secondary target group is students abovementioned schools.

The publication consist of three chapters. The first chapter is devoted to the algorithmic and it presents the idea of programs and programming languages, the importance of algorithms and its design. The second chapter presents the definition of programming, its history as well as programmers skills and the process of developing them. In this chapter the information about principles of programming are also

consisted. The last one chapter is devoted to presentation of didactic elements with the use of algorithmic and programming. In this chapter the information about basic assumptions of algorithmic and programming in school teaching, as well as computational thinking concept are presented. The final part of the publication is devoted to presentation of practical application of computational thinking.

The Authors hope that the publication meets meet with interest from teachers and brings them the useful knowledge in algorithmic and programming. The publication initialized the set of educational materials for teachers and students, which will also include:

- Virtual Learning Environment for Teachers containing training materials in algorithmic and programming and its didactic in other than IT subjects,
- Model lesson plans incorporating programming for Chemistry, Geography, Maths and Physics,
- Handbook entitled “Advance your teaching skills with the use of algorithmic and programming”.

The Authors

1 Introduction to algorithmic

(Maria Christodoulou)

An algorithm is a description of how a specific problem should be solved.

If you have ever baked brownies, you know that first you have to gather ingredients, then measure them, mix them together then prepare the pan, heat the oven and cook them.

Figure 1 – Algorithmic description is like a recipe

Source: momsrecipesandmore.blogspot.gr

If you forget the sugar, they do not taste good and you have to start over.

Determining the right steps, following them correctly and completely and learning from mistakes are all part of the process of algorithm design.

1.1 Computer Programs

In order to be executed by computers, algorithms need to be in the form of a 'program'. A program is written in a programming language, and the activity of expressing an algorithm as a program is called programming.

In algorithms, steps are expressed in the form of an **instruction** or **statement**. As a consequence, a computer program comprises a series of statements which indicate to the computer which operation to perform.

The programming language used will dictate the nature of the statements in a program.

1.2 Algorithms and their importance

To use a computer for the purpose of executing processes, it is necessary to:

- design the algorithm to describe how the process will be performed;
- use a programming language to express the algorithm into a program;
- run the program on the computer.

To this end, it is important to understand that **algorithms are independent of the programming language used** and each algorithm can be expressed in different programming languages and executed on different computers. This is the reason why the design of algorithms is a fundamental aspect of computer science. The design of an algorithm is a demanding intellectual activity, significantly more difficult than expressing the algorithm as a program.

Among the **skills needed to design algorithms are creativity and insight** (Goldschlager and Lister, 1988) while there is no general rule, meaning there is no algorithm for algorithm design!

1.3 Algorithmic design

Algorithm Design:

- comprises a set of instructions for completing a task,
- moves the problem from the modelling phase to the operation stage,
- the set of instructions should be sequential, complete, accurate and have a clear end point,

- if intended for a computer the algorithm must comprise a series of tasks written in a way that the computer is able to perform.

In this chapter we will look into algorithms, considering their structure, their composition and their expression in an executable form.

1.4 Algorithms, programs and programming languages

As said an algorithm is a description of how to carry out a task or process and there are algorithms for carrying out pretty much all kinds of tasks/processes. From building a model plane to guiding an excavation machine.

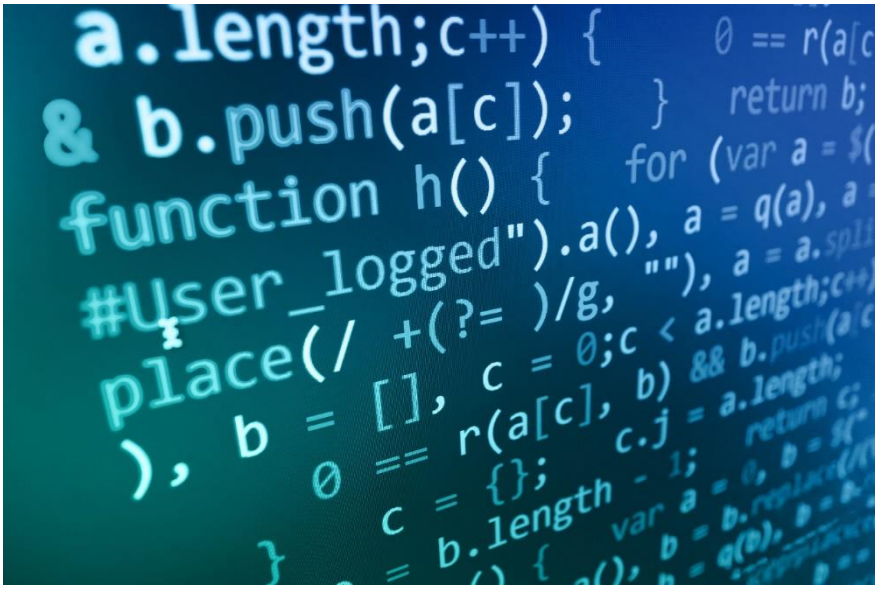


Figure 2 – Algorithms, programs and programming languages

Source: shutterstock.com

More often than not, the process described by **an algorithm interacts with its environment, accepting input(s) and producing output(s)** (Goldschlager and Lister, 1988). Processes executed on the computer more often require input in the form of data and very often the output produced will also be in the form of data. As an example, think about the process of calculating salaries which will require inputs such as daily cost, days worked, etc. and will produce outputs such as salaries to be paid, employees/employers contributions and taxes to be deducted.

Inputs and outputs are part of the specification of a process but are still independent of the processor which carries out the process.

Another important aspect of processes is the **termination**. Process may terminate or may never terminate. This is the reason that the **termination or nonterminating of a process is one of its important characteristics.**

1.4.1 Syntax and semantics

A computer needs to be able to interpret an algorithm in order to execute the process described by the algorithm. As such, the computer must be able to:

- understand in which **form** an algorithm is expressed;
- execute the **operations** described by the algorithm.

Figure 3 – Syntax and semantics

Source: shutterstock.com

In the present section we will look into the **form in which algorithms are expressed**. The set of grammatical rules which govern how the symbols in a language may be legitimately used is called the **syntax** of the language (Goldschlager and Lister, 1988).

A program which adheres to the syntax of the language in which it is expressed is said to be **syntactically correct**. A deviation from the legitimate syntax of a language is called a **syntax error**. Syntactic correctness is normally a prerequisite for a computer to be in position to execute a program.

The meaning of particular forms of expression in a language is called the **semantics** of the language. Detection of semantic inconsistencies relies on knowledge of the objects being referred to and in particular on knowledge of the attributes of those objects, and of the relationships between them. Consider for example (Goldschlager and Lister, 1988) a computer processor faced with the following command:

write down the name of the 13th month of the year

If the processor knows that there are only 12 months in the year it can detect the semantic inconsistency in the command before trying to execute it. If it does not know this, then it will attempt to execute it and most likely an exception will be raised.

More often, semantic inconsistencies and more subtle, being the result of executing a previous part of the algorithm:

Think of a number from 1 to 13

Call this number N

Write down the name of the Nth month of the year

This algorithm contains a potential inconsistency which emerges only if execution of the first line results to number 13. When an inconsistency is a result of executing an algorithm there is in general no chance of detecting it before-hand.

In addition to **syntacti**c and **semantic** errors, there are also **logical** errors. It is possible that a program may be syntactically correct and contain no semantic inconsistencies, but may still not lead to the intended process.

As an example (Goldschlager and Lister, 1988), consider the algorithm for computing the circumference of a circle:

Compute the circumference by multiplying the radius by π

The algorithm is syntactically and semantically correct but it produces a wrong result due to the logical error which is the omission of a factor of 2.

Unfortunately, logical errors are hard to be detected by computer processors prior to executing the process and comparing the result with the desired outcome.

1.4.2 Algorithmic Design: Stepwise refinement of algorithms

The design of algorithms describing non-trivial processes is usually very difficult. Very frequently the errors appearing in algorithms have to do with the process described where the described process is very close to the intended process but not exactly.

Another common failure is when execution results in the intended process being carried out, but in certain circumstances (unforeseen or overlooked by the designer) it does not. Here is such an example (Goldschlager and Lister, 1988) which describes how to calculate the flight time of an aircraft from an arrival timetable:

1. Look up departure time
2. Look up arrival time
3. Subtract departure time from arrival time

This algorithm will produce the correct result in most cases, but will fail to do so if the departure point and the destination are in different time zones.

The conclusion is that the designer of an algorithm should ensure the algorithm describes precisely the process that needs to be carried out while all possible

circumstances have been accounted for. If the process to be carried out is too complex then the designer's task is difficult.

This is why a methodical approach is needed. One such approach is **the stepwise refinement** (or **top-down design**).

Stepwise refinement is a variation of the divide and conquer where the process to be carried out are broken down into a number of steps, each of which can be described by an algorithm which is smaller and simpler. Because each such sub-algorithm is simpler than the entire process the designer usually has a clearer idea of how to efficiently construct it, and can therefore sketch it in more detail than if he tried to handle the whole algorithm at once. The sub-algorithms can themselves be broken into smaller pieces which are even simpler and can again be expressed in even more detail and precision. Refinement of the algorithm continues this way until each of its steps is sufficiently detailed and precise to allow execution by the computer processor.

EXAMPLE

Design an algorithm for a domestic servant robot to make a cup of instant coffee.

The initial version of the algorithm can be the following (Goldschlager and Lister, 1988):

```
(1) boil water
(2) put coffee in cup
(3) add water to cup
```

The steps in this algorithm are not detailed enough for the robot to be able to execute them. Each step must therefore be refined into a sequence of simpler steps, each specified in more detail than the original. Thus the step:

```
(1) boil water
```

might be refined into

```
(1.1) fill boiler with water
(1.2) switch on boiler
(1.3) wait until the water boils
(1.4) wait for the boiler to switch off
```

Similarly,

```
(2) put coffee in cup
```

might be refined into

```
(2.1) open coffee jar
(2.2) dip spoon and fill with coffee
(2.3) drop spoonful into cup
(2.4) close coffee jar
```

and

```
(3) add water to cup
```

might be refined into

```
(3.1) pour water from boiler into cup until cup is full
```

The last refinement does not actually increase the number steps in the algorithm, but simply re-expresses an existing step in more detail.

At this stage the original algorithm has been refined into three sub-algorithms, to be executed in sequence. If the robot can interpret all the steps in each sub-algorithm, then the process of refinement can stop and the design of the algorithm is complete. However, some steps may still be too complex for the robot to interpret and these steps must be refined further. Thus the step:

```
(1.1) fill boiler with water
```

may need further refinement into

```
(1.1.1) put boiler under tap
(1.1.2) turn on water tap
(1.1.3) wait until boiler is full
(1.1.4) turn off water tap
```

Other steps may need similar refinement, though there may be some, such as:

```
(1.2) switch on boiler
```

which can already by executed by the robot without the need for further refinement.

Finally, after a number of refinements, every step of the algorithm will be understood and executed by the robot. At this stage the algorithm is complete. The successive refinements are shown in the Figure 4 below.

Original	First refinement	Second refinement
(1) boil water	(1.1) fill boiler with water	(1.1.1) put boiler under tap
		(1.1.2) turn on water tap
		(1.1.3) wait until boiler is full
		(1.1.4) turn off water tap
	(1.2) switch on boiler	
	(1.3) wait until water boils	(1.3.1) wait until boiler switches off
	(1.4) Wait for boiler to switch off	
(2) put coffee in cup	(2.1) open coffee jar lid	(2.1.1) take coffee jar from shelf
		(2.1.2) remove lid from jar
	(2.2) dip spoon and fill with coffee	
	(2.3) drop spoon of coffee into cup	
	(2.4) close coffee jar lid	(2.4.1) put lid on coffee jar
		(2.4.2) replace coffee jar on shelf
(3) add water to cup	(3.1) pour water form boiler into cup until cup is full	

Figure 4 - Refinement of a coffee-making algorithm

Source: example based on (Goldschlager and Lister, 1988)

The final version of the algorithm is obtained by taking the last refinement of each step, as shown in Figure 5.

```
{boil the water}
(1.1.1)  put boiler under water tap
(1.1.2)  turn on the water tap
(1.1.3)  fill boiler until full
(1.1.4)  turn off water tap
(1.2)    switch on the boiler
         wait for boiler to boil the
(1.3.1)  water
(1.4)    switch off the boiler
{drop a spoon of coffee to a cup}
(2.1.1)  pick up coffee jar
(2.1.2)  remove lid
(2.2)    fill in spoon with coffee
(2.3)    drop spoon into cup
(2.4.1)  close lid
(2.4.2)  put coffee jar back
{poor water into cup}
         poor boiled water from boiler
(3.1)    into the cup until full
```

Figure 5 – Final version of the coffee-making algorithm

Source: example based on (Goldschlager and Lister, 1988)

Using stepwise refinement implies that the algorithm designer knows where to stop. The designer must know when a specific step in the algorithm is sufficiently primitive to be left without any further refinement. This necessitates some knowledge by the designer about the sort of steps the computer processor can execute.

In our example, the designer knew that the robot can interpret *switch on boiler* and no further refinement was applied but the robot cannot interpret *fill boiler* and additional refinement was applied.

The conclusion is that stepwise refinement of an algorithm cannot take place in a vacuum. The designer must be aware of the interpretive capabilities of the intended processor so that he can push the refinement in particular directions and know when to terminate the refinement of each part.

The good news is that the interpretive capabilities of computers are precisely defined: **a computer can interpret anything which is properly expressed in a programming language.** Thus, the designer refines the algorithm in such a way that the steps can be expressed in an appropriate programming language, and terminates the refinement when every step is expressed in the language of choice.

Each refinement implies a number of design decisions based upon a set of design criteria. Among these criteria are **efficiency**, **storage economy**, **clarity**, and **regularity of structure**. Designers must be taught to be conscious of the involved decisions and

to critically examine and to reject solutions, sometimes even if they are correct as far as the result is concerned; they must learn to weigh the various aspects of design alternatives in the light of these criteria. In particular, they must be taught to revoke earlier decisions, and to back up, if necessary even to the top. Relatively short sample problems will often suffice to illustrate this important point; it is not necessary to construct an operating system for this purpose.

1.4.3 Algorithmic Design: Sequence

The coffee-making algorithm of the previous section involves simple steps to be executed sequentially:

- steps are executed one at a time and not in parallel,
- each step is executed only once,
- execution order is the order in which steps are written,
- execution of the last step terminates the algorithm.

Such an algorithm is not flexible as it cannot be adapted to respond to different circumstances. For example think about the unavoidable situation that at some point the coffee jar is empty or the situation when the robot needs to handle several requests for coffee or custom requests for milk or sugar.

The conclusion is that an algorithm which is merely a combination of steps in a sequence will not get us far and more flexible structures are needed in order to design algorithms capable of realistically depicting real life situations.

1.4.4 Algorithmic Design: Selection

A more advanced structure which provides flexibility is the **selection**. Using the selection, we can refine step 2.1 of the previous coffee-maker algorithm example as follows (Goldschlager and Lister, 1988):

```
(2.1.1) take coffee jar off the shelf
(2.1.2) if jar is empty
            then get new jar from cupboard
(2.1.3) remove lid from jar
```

We see the introduction of a **condition** in step 2.1.2. The condition is *"jar is empty"*. If the condition is met, then a conditional instruction applies which is "*get new jar from cupboard".*

This is the general form in which the selection structure is expressed:

```
If condition
then step
```

The condition can be used to specify any kind of circumstance which when true needs the execution of a certain step.

Off course, in real life there are alternatives in case a particular circumstance emerges. To cope with such situations, the selection structure can be extended so as to provide alternative steps to be executed.

Assume a simple algorithm for driving with a car to work which needs to cater for the occasion that the car needs fuel:

```
(1) start car
(1.1) if fuel indicator on
then drive to the nearest gas station
else drive to work
```

In this case we can select between two alternative steps in which case the condition (fuel indicator on or off) dictates which step is to be executed based on the situation we are faced with.

Here is another example of how refinements and the use of selection can produce a much more realistic algorithm capable of handling most circumstances without unexpected outcomes:

```
(1) wash car
(1.1) if feeling lazy
            then drive to the car wash to get it washed
(1.2) wash it by hand
```

Step 1.1 can be further refined:

```
(1.1.1) buy a token
(1.1.2) wait in line
(1.1.3) have the car washed
```

Step 1.1.3 can be further refined:

```
(1.1.3.1) drive into the car wash
(1.1.3.2) check that all doors and windows are closed
(1.1.3.3) get out of the car
(1.1.3.4) put token to the machine
(1.1.3.5) wait until washing cycle and drying is finished
(1.1.3.6) get into the car
(1.1.3.7) drive away
```

More flexibility can be introduced to algorithms by taking advantage of **nested** selection. Consider the following algorithm for a pedestrian crossing a street at the zebra crossing.

```
if light is green
      then proceed
      else stop
```

This example contains one selection and can be further improved as:

```
if no light or light is blinking green
      then proceed with caution
      else if light is red
            then stop
else proceed
```

This later example contains two selections. The second selection is nested inside the first and is executed only if the light is red.

1.4.5 Algorithmic Design: Iteration

Iteration is the repetition of a process in a computer program. Iterations of functions are common in computer programming, since they allow multiple blocks of data to be processed in sequence. This is typically done using a "while loop" or "for loop". These loops will repeat a process until a certain number or case is reached.

Simply put, **iteration** means repeating the same step several times.

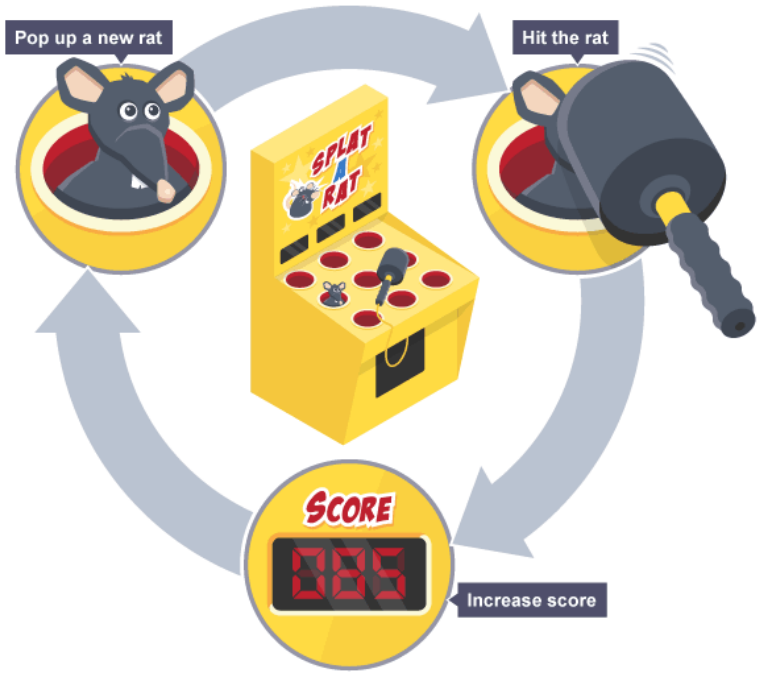


Figure 6 – Iteration: Repeating the same step several times

Source: okclipart.com

Iterative algorithms should obey three important principles:

1. An iterative process repeats (iterates) a certain sub-process.
2. Each iteration should change at least one value.
3. There should be some condition under which the iteration terminates. And the iteration should reach that state.

A very simple algorithm for eating breakfast cereal might consist of these steps (BBC Bitesize KS3 Subjects: *Iteration*):

```
(1) put cereal in bowl
(2) add milk to cereal
(3) spoon cereal and milk into mouth
(3.1) repeat step 3 until all cereal and milk is eaten
(4) rinse bowl and spoon
```

What we see in step 3.1 is the introduction of a **condition** which is a situation that is checked every time an iteration occurs. The condition is introduced with the words **repeat** and **until**. Without the condition, the algorithm would not know when to stop. The condition, in this case, will be to check if all the milk and cereals is eaten. If that condition is False (there is still milk and cereals in the bowl), then another iteration occurs. If the condition is True (there is no more milk and cereals in the bowl), then no more iterations occur.

The general form is:

```
repeat
      part of the algorithm
until condition
```

This means that the part of the algorithm between *repeat* and *until* is executed repeatedly until the condition specified after *until* holds true. The condition which comes after *until* is called a **terminating condition**.

The occurrence of iteration is called a loop.

Iteration allows algorithms to be simplified by stating that certain steps will repeat until told otherwise. This makes designing algorithms quicker and simpler because they don't need to include lots of unnecessary steps (BBC Bitesize KS3 Subjects: Iteration).

A more advanced use of iteration is the use of **nested loops**. A nested loop is a loop within a loop, an **inner loop** within the body of an **outer** one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then, the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes.

When one loop is nested within another, several iterations of the inner loop are performed for every single iteration of the outer loop.

Example: Suppose we have data in the form below, involving several ID strings. For each ID string, a variable number of readings have been recorded; the number of readings for each ID is shown in the `howMany` column:

ID	howMany	Readings
200	3	110 30 10
201	5	2 46 109 57 216
202	2	10 500

Our task is to read the data and display a summary chart showing the average of each reading per ID as follows**:**

```
ID          AVERAGE
200         50
201         86
202         255
```

This is how the algorithm could look like:

```
read first ID and howMany
     repeat
            display ID
                  repeat
                        read and sum up this ID’s howMany readings
                        calculate and display average for ID
                  until end of howMany readings
                  read next ID
     until end of ID
```

Looking at the algorithm we easily observe that the outer loop controls the number of lines and the inner loop controls the content of each line. So, 3 IDs leads to 3 rows and each row will display just one average calculated by adding all readings to get their sum and dividing by the number of readings to get the average.

1.4.6 Summary of most important algorithmic constructs

Sequence: A series of steps that are carried out sequentially in the order in which they are written. Each step is carried out only once

Selection: One of two or more alternatives should be chosen.

Iteration: Part of the algorithm should be capable for repetition, either a defined number of times or until a certain condition has been met.

1.4.7 Algorithmic Design: Recursion

Recursion is a powerful tool based on which the algorithm can be expressed in terms of itself. It provides a simple, powerful way of approaching a variety of problems. It is often hard to think about how a problem can be approached recursively. It is also very easy to write a recursive algorithm that either takes too long to run or doesn't properly terminate.

In the present sub-section we will look into the basics of recursion in order to provide the reader with the opportunity to understand how complex problems may be solved easily using recursive thinking.

The first thing to do is to answer the question: “what is recursion?”

Figure 7 – Recursion

Source: Shutterstock.com

Russian dolls offer a helpful analogy for understanding **recursion**. Suppose you have an algorithm for opening each doll. Then, to get to the last doll which cannot be opened any more to get a smaller version, you would execute the same algorithm each time until there are no more dolls to open.

In algorithmic design, **an algorithm is said to be recursive if it calls itself**. Obviously when dealing with an algorithm which calls itself there is a challenge in ensuring that the algorithm will eventually terminate producing the correct result. The circularity of recursive algorithms is avoided by ensuring that each time the input to the successive execution is decreased by some number so that at some point it will eventually become zero and the algorithm will terminate.

To implement a recursive algorithm, the problem must satisfy the following conditions:

- It can be broken down into a simpler version of the same problem
- It has one or more base cases whose values are known

Here is a real world example of recursion: Consider you are seated in the last row of a cinema and you want to count the number of rows. You ask the person seated

in front of you to tell you how many rows are there in front of him. This person does the same thing and asks the person who is seated in front of him and so on till the person who is seated first row is asked the question. This person can answer zero to the person behind him because he can see that there are no more rows in front of him. This is the base case. The person in the second row adds 1 to the answer and tells it to the person behind him in the third row and so on until the answer reaches you. Notice that at each step the problem is reduced to a simpler version of the same problem.

Let's see now an example using an algorithm. First, consider a **module** (in programming languages it is called procedure or function or process or routine) which prints the phrase "Hello CodeIT Enthusiasts" a total of `X` times:

```
module Hello(X)
if(X<1)
      return print("Hello CodeIT Enthusiasts!")
      Hello(X - 1)
```

Let's simulate the execution by calling the module Hello with a value for `X = 10`. Since `X` is not less than `1`, we do nothing on the first line. Next, we print "Hello CodeIT Enthusiasts!" once. At this point we need to print our phrase 9 more times. Since we now have a Hello module that can do just that, we simply call Hello (this time with `X` set to 9) to print the remaining copies. That instance of Hello will print the phrase once, and then call another copy of Hello to print the remaining 8. This will continue until finally we call Hello with `X` set to zero. `Hello(0)` does nothing; it just returns. Once `Hello(0)` has finished, `Hello(1)` is done too, and it returns. This continues all the way back to our original call of `Hello(10)`, which finishes executing having printed out a total of 10 "Hello CodeIT Enthusiasts!".

These are some key considerations in designing a recursive algorithm:

- It handles a simple generic situation without using recursion: In the example above, the generic situation is `Hello(0)`. If the module is asked to print zero times then it returns without spawning any more "Hello CodeIT enthusiasts".
- It avoids cycles. Imagine if `Hello(10)` called `Hello(10)` which called `Hello(10)`. You'd end up with an infinite cycle of calls and this usually would result in a "stack overflow" error while running on a computer. In many recursive programs, you can avoid cycles by having each module call be for a problem that is somehow smaller or simpler than the original problem. In this case, for example, X will be smaller and smaller with each call. As the problem gets simpler and simpler (in this case, we'll consider it "simpler" to print something zero times rather than printing it 5 times) eventually it will arrive at the generic situation and stop recursion. There are many ways to avoid infinite cycles, but making sure that we're dealing with progressively smaller or simpler problems is a good rule of thumb.

- Each call of the module represents a complete handling of the given task. Sometimes recursion seems to breaks down big problems in a magical way but this is not what happens in reality. When our module is given an argument of 10, we print "Hello CodeIT enthusiasts!" once and then we print it 9 more times. We can pass a part of the job along to a recursive call, but the original function still has to account for all 10 copies somehow.

One reason why recursion is a bit confusing is that it does not follow a direct natural way of thinking as is the case with iteration.

A recursive algorithm implies that the recursive module part of the algorithm needs to be invoked initially from outside the module, but then it keeps calling itself from inside the module until it reaches a terminating condition. From a computer's perspective if a module calls itself or calls another module it is still a regular module call as long as the required input parameters are provided.

Recursion follows the divide and conquer algorithm design technique which means breaking a problem into sub problems of the same or related type until the sub problem becomes simple enough to be solved directly. The solutions to the sub problems are then combined to give a solution to the original problem. So, how divide and conquer relates to a module calling itself? In the first call to the recursive module the full input value is provided, the moment the module starts invoking itself, a lesser input is provided each time in each call unit the input is simple enough to allow for a solution to be provided in a single call.

As it turns out, for any algorithm which uses recursion there can be an equivalent algorithm which uses iteration. So, recursion can be also seen as a different form of iteration. Nevertheless, a recursive algorithm for describing a particular process is often far more concise than an iterative one.

Perhaps the most famous example of a problem which can be solved with recursion more efficiently than it can be understood and solved with iteration is the Towers of Hanoi.

Lastly, as general knowledge, recursion is very big in game development. Here are some frequent uses in game development (Quora.com):

- *Body part destruction*: Evaluate voxels around a sword cut to determine what is still connected, and what should fall off
- *Maze hallway creation*: Use recursion to randomly wander a maze, adding doorways at set intervals.
- *Pathfinding*: Construct a distance dictionary around a grid position to compare path lengths between movement alternatives.

1.4.8 Differences between iterative and recursive algorithms

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure. An **Iterative algorithm** will use looping statements such as for loop, while loop or do-while loop to repeat the same steps while a **Recursive algorithm,** a module (function) calls itself again and again till the *base condition*(stopping condition) is satisfied.

An Iterative algorithm will be faster than the Recursive algorithm because of overheads like calling functions and registering stacks repeatedly. Many times the recursive algorithms are not efficient as they take more space and time.

Recursive algorithms are mostly used to solve complicated problems when their application is easy and effective. For example Tower of Hannoi algorithm is made easy by recursion while iterations are widely used, efficient and popular.

Recursive vs Iterative Algorithms:

- **Approach:** In recursive approach, the function calls itself until the condition is met, whereas, in iterative approach, a function repeats until the condition fails.
- **Programming Construct Usage:** Recursive algorithm uses a branching structure, while iterative algorithm uses a looping construct.
- **Time & Space Effectiveness:** Recursive solutions are often less efficient in terms of time and space, when compared to iterative solutions.
- **Termination Test:** Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- **Infinite Call:** An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.

A practical problem: The birthday guy cuts the birthday cake and has to ensure that everyone in the room gets a slice.

Solution 1 - Iterative: The birthday guy uses a tray and goes around giving everyone a slice.

`serveslice` (while there are people to serve or slices left give slice)

Solution 2 - Recursive: Take a slice of cake from the tray and pass the tray to the next person who takes a slice from the tray and passes the tray to the next person, who takes a slice from the tray and passes the tray to the next person…

Note that the same function is being performed every time.

```
takeslice(tray)
If there are people to serve or slices left
     take slice and call takeslice(tray)
else return
```

1.4.9 Algorithmic Design: Data structures

Until now, focus was on how to control algorithms by looking into constructs which are used for controlling the order and circumstances in which the individual steps of an algorithm will be executed. The reason is that the choice of the most appropriate control structures is important for designing an efficient and effective algorithm. However, algorithms are designed to handle inputs in the form of data and these inputs need to be considered when designing an algorithm.

Usually, the data which will be processed by an algorithm comprises items which are related in some way to each other rather than arbitrary collections of unrelated items. Consider for example the metadata which makes up a user profile, such as name, age, address, function, contact number. Etc. These items are logically connected to one another and cannot be processed as a collection of unrelated items by an algorithm but need to be processed as a **record** where each record comprises information about a specific user.

Data which is organised in such a way so as to capture logical relationships between its elements is referred to as **structured data** and its elements together with their interrelationships form a **data structure**.

The most common type of data structure is perhaps the **sequence** which comprise a set of items sorted in a way that every item apart from the last item in the sequence has a successor and every item apart from the first one in the sequence has a predecessor. These are some common examples of easily understood sequences:

- A number presented as a sequence of digits, e.g. *1512000 (1 is first, 0 is last)*
- A word presented as a sequence of letters, e.g. *sequence (s is first, e is last)*
- A chain
- A denture
- A rolodex

Obviously, a sequence is an ideal structure for items which are to be processed sequentially, one after another. The most common way of progressing through a sequence is through the use of a looping structure such as a while loop or a repeat loop with the difference being that a while loop is used when the number of items in the sequence is not known while the repeat loop is used for a definitive iteration when the number of items in the sequence is known.

Example algorithmic construct for navigating through a sequence is the following:

```
enter sequence
while not end of sequence do
      process next item in the sequence
```

Some sequences are so common and occur so often that have been given specific names:

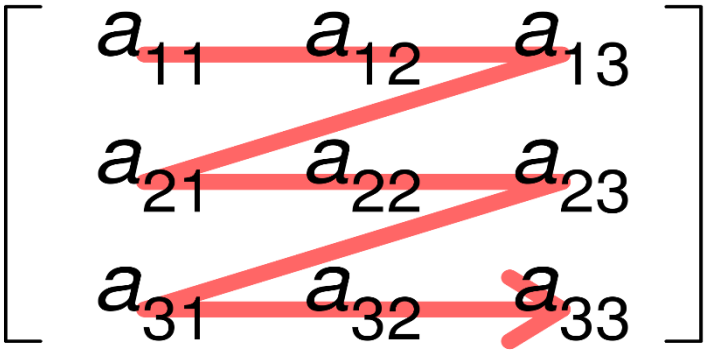


Figure 8 – Array

Source: wikipedia.org, Row- and column-major order

Arrays: A sequence of fixed length where each element is identified by its position in the sequence (Goldschlager and Lister, 1988). It is an indexed collection of homogeneous elements. An Array sequence has the following properties:

- All elements of the array are of the same data type (homogeneous),
- Dimension: One dimensioned arrays have one dimension, two dimensioned arrays have two dimensions, etc.,
- Each dimension has a size,
- The size of a particular dimension is constant. Consider a matrix. It has rows and columns. The size of each row (number of columns), is the same for all rows,
- Each element of an array is identified by its position along each dimension (index, e.g.: num[1], num[4], etc.).

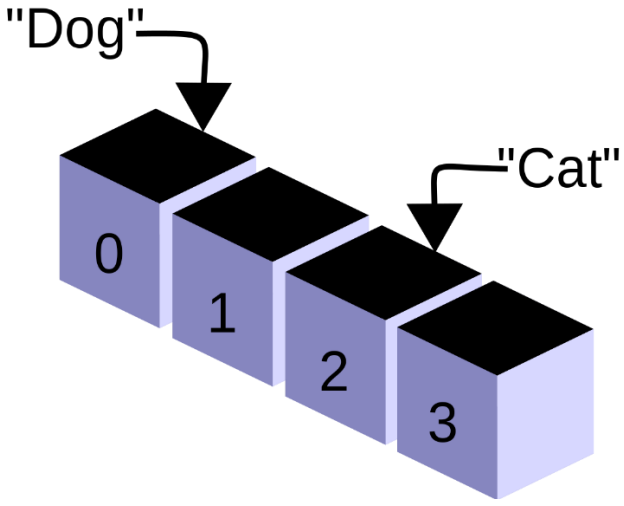


Figure 9 – Vector

Source: wikibooks.org, A-level_Computing

Vectors: A Vector is similar to an array but with added functionality. Vectors keep track of their size and they resize automatically when you add or delete an item. On the positive side, vectors are easy to use, keep track of their size, they are resizable and offer simple access like the normal arrays. On the down side, resizable vectors can be slow (lots of new items and deletion of old items) so it is important to ensure an efficient implementation.

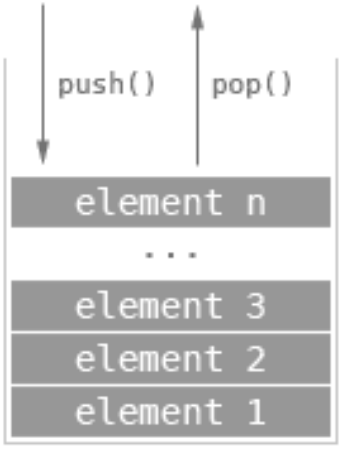


Figure 10 – Stack

Source: Creative Commons Image

Stacks: A sequence of variable length in which items are added and removed only at one end. Think of it as a container of elements that are inserted and removed according to the last-in first-out (LIFO) principle because the order in which elements are removed is the revers of that in which they are added:

- Two operations are allowed, ***push*** the item into the stack, and ***pop*** the item out of the stack,
- Elements can be added and removed from the stack only at the top (think of a stack of plates in a cupboard),
- Push adds an item to the top of the stack, pop removes the item from the top,
- It is a recursive data structure,
- It is either empty or it has a top followed by a stack.

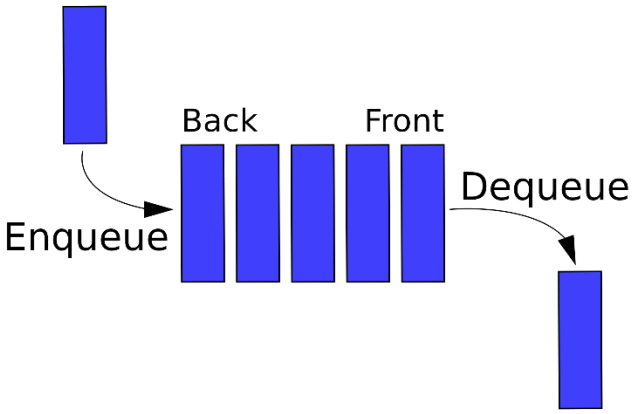


Figure 11 –Queue

Source: Creative Commons Image

Queues: A sequence of variable length where items are always added on one end and removed from the other end. Unlike stacks, a queue is open at both its ends and follows the First-In-First-Out principle (FIFO), i.e., the data item stored first will be accessed first because the order in which items are removed is the same of that in which they are added. A queue comprises a:

- **Rear**, also referred to as **tail** which is where the first element is inserted from one end.
- **Front**, also referred to as **head** which is where the first element is deleted from the other end.

A real-world example of queue is a single-lane one-way road, where the vehicle which enters the road first, exits first. Another simple example is any kind of service point, such as a ticket counter where the first person to queue is the first person to get their ticket from the counter.

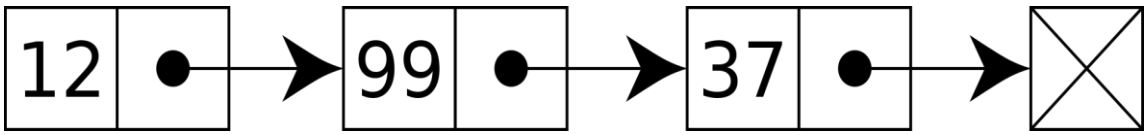


Figure 12 –Linked List

Source: Creative Commons Image

Linked lists: A linked list is a data structure which consists of a group of nodes in a sequence and is used to store elements when we don't know how many elements we are going to store. A linked list is basically a sequence of structures where each element in the structure has a **Node** and a **Link**. The node contains the items stored which may well be of different type and the link is a pointer which points to the next node in the sequence. A linked list is a data structure capable of storing an arbitrarily large ordered collection of items with a minimum of overhead because a great advantage of linked lists is that insertion and deletion operations can be easily implemented.

A common use of linked lists is for implementing stacks and queues with the advantage of allowing the insertion of elements at the beginning and end of the list while the size of the linked list is not necessary to know in advance. So, they are easier to edit than arrays and can be split and manipulated easily.

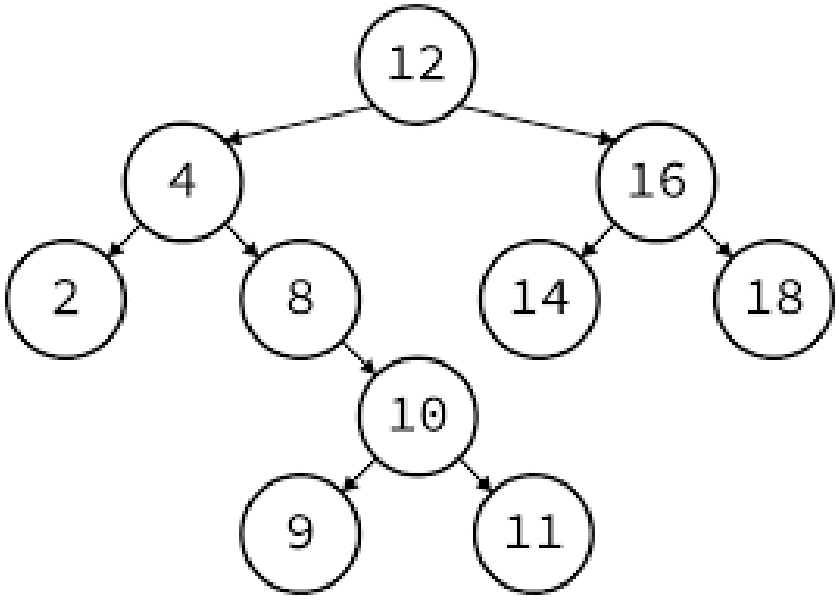


Figure 13 –Tree

Source: Creative Commons Image

Trees: A tree is a powerful data structure for organising items. Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure. A tree is hierarchically arranged and is used for expressing hierarchical relationships, such as an organisational structure. Its ***nodes*** are used for representing its data or branching points. A tree node has a data part and references to its left and right child nodes. The node at the highest level of the hierarchy is called a ***root*** while the nodes at the lowest level, where the tree ends, are called ***leaves***. The ***branches*** of the tree represent logical relationships between items at one level of the hierarchy and other items at the next level. A tree is structured in a way that every node is at the same time the root of another tree (subtrees). So, hopefully by now, it has become apparent that **a tree is a recursive data structure** as each tree can be defined in terms of other trees. In terms of properties, a tree is either:

- empty (a tree with no nodes is called the null or empty tree), or
- a node and a set of branches each belonging to a tree (a tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy)

The PDF is a tree based format. It has a root node followed by a catalogue node, followed by a pages node which has several child page nodes.

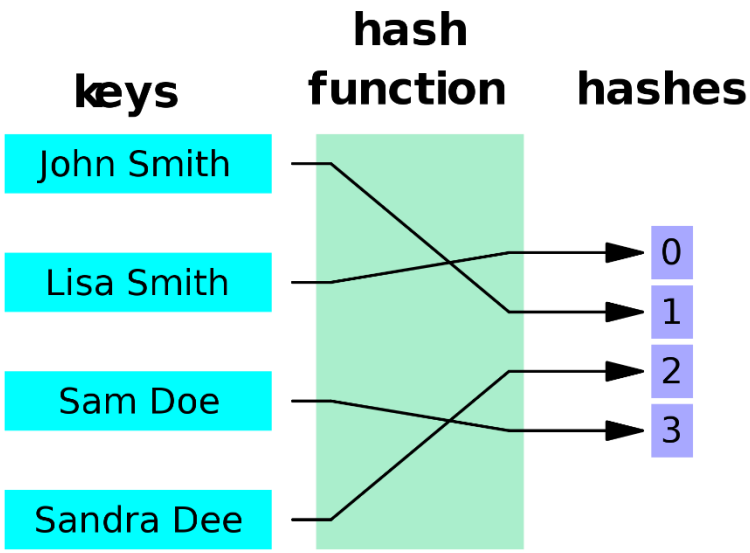


Figure 14 – Hash Table

Source: Creative Commons Image

Hash Tables: Hash tables are key/value pairs. A hash table (also referred to as hash map) is a data structure used to implement an associative array, a structure that can map keys to values. It files items according to some property that is easy to find, even if not directly relevant to the item, and not a complete description. The Hash table helps reduce the possible places where an item can be found. A hash table is populated through a 'Hashing function'. A hashing function is a way of mapping elements to locations. As an example consider as a hashing function a function that will help you decide (and later search/find) the shelf in which you are going to place a book. A hashing function is a mathematical function which is given an item and provides the location to store the item while at the same time it maps each item to the corresponding location. Hash tables allow insertion/search and deletion of items by providing the associated keys for each item. The keys are transformed into integers by a hash function and the items are stored in an array using indexes corresponding to these integers (hashed keys).

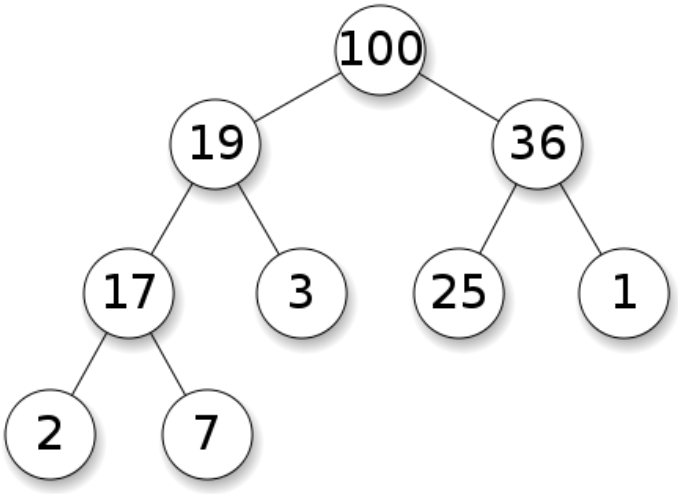


Figure 15 – Heaps

Source: Creative Commons Image

Heaps: A heap is a collection that allows items to be inserted and the smallest item to be found and/or removed. Think of a heap data structure as an implementation of a "priority queue" where instead of just joining a queue at its tail, a person or object may be inserted further up the queue depending on their priority. Heaps are especially useful when needing to query for the minimum value multiple times from a dynamic collection of values.

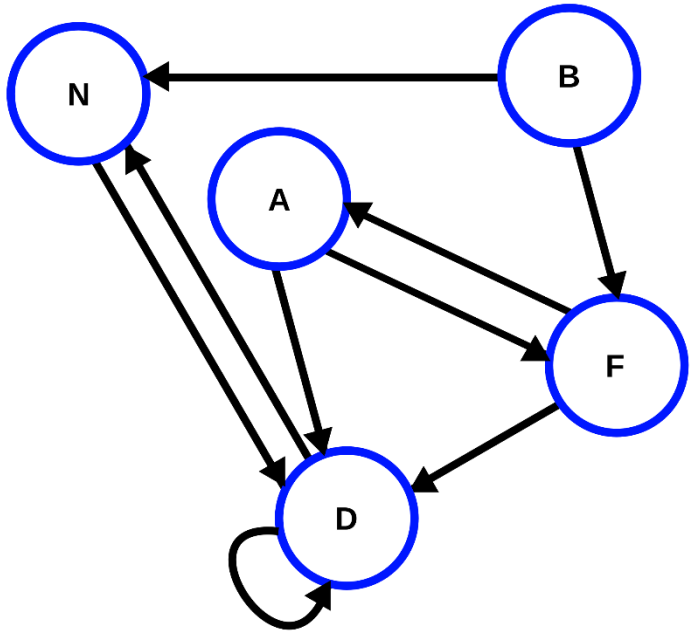


Figure 16 – Graphs

Source: Creative Commons Image

Graphs: Graph data structures provide for an easy representation of real life relationships between different types of data (nodes) and are thus used for representing networks. A graph is made up of ***vertices***, ***nodes***, or ***points*** which are connected

by ***edges***, ***arcs***, or ***lines.*** A graph may be ***undirected*** (the relationship exists in both directions, i.e. John is a friend of Paul means Paul is a friend of John), meaning that there is no distinction between the two vertices associated with each edge, or its edges may be ***directed*** (it can be a one way relationship or a two way relationship but it must be explicitly stated) from one vertex to another. The link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. There are different ways to construct graphs in algorithmic design. The data structure used depends on both the graph structure and the method used for manipulating the graph. Usually, graphs are constructed by using a combination of list and matrix structures. The following operations can be performed on graphs:

- Additions:
    - Add vertices to the graph
    - Create edges between two given vertices in the graph
- Removals:
    - Remove vertices from the graph
    - Remove edges between two given vertices in the graph
- Search
    - Check if the graph contains a given value
    - Check if a connection exists between two given nodes in the graph

1.5 Bibliography

BBC Bitesize KS3 Computer Science Algorithms [On-line source: https://www.bbc.co.uk/education/topics/z7d634j] (available on: 8th February 2018)

Goldschlager L, Lister A., (1988), *Computer Science: A Modern Introduction*, Second edition, Prentice Hall International Series in Computer Science.

Quora.com, Questions and Answers on “Data Structures”, “Algorithms”, “Programming” [On-line source: http://quora.com] (available on: 8th February 2018)

2 Introduction to programming

(Elżbieta Szczygieł, Łukasz Kłapa)

2.1 The definition of programming

The intuitive definition of programming does not cause much difficulty and brings to mind the introduction of commands in the form of a code, expressed in a specific programming language. However, defining the exact essence of programming is a bit more difficult. This results from the necessity of using several terms, which should be defined beforehand (among others: algorithm, program, programming language) and the degree of formalization of the transferred content.

The following definition of **programming** was adopted for the purposes of this publication:

> *A set of activities related to the preparation of a program for a digital machine*
> (*Encyclopedia*, 1988).

Since the computer is the most common machine, the term will be used in the rest of this handbook to indicate any kind of electronic digital machine. Under the term **program** is meant the **algorithm** record of a specific task in the form of a sequence of declarations and instructions in one of the **programming languages** (Encyclopedia, 1988). The **algorithm** is defined as a description of the solution to a **problem (task)** expressed by means of such operations that the algorithm performer understands and is able to perform (Encyclopedia, 1988). Therefore, programming has to do with a task that needs to be solved in a certain way. This method is expressed in a specific form, which is the **programming language**. This language is intended for the recording of data processing programs by computers (Encyclopedia, 1988). Bearing these assumptions in mind, the concepts related to programming can be arranged in the following way (Figure 17).

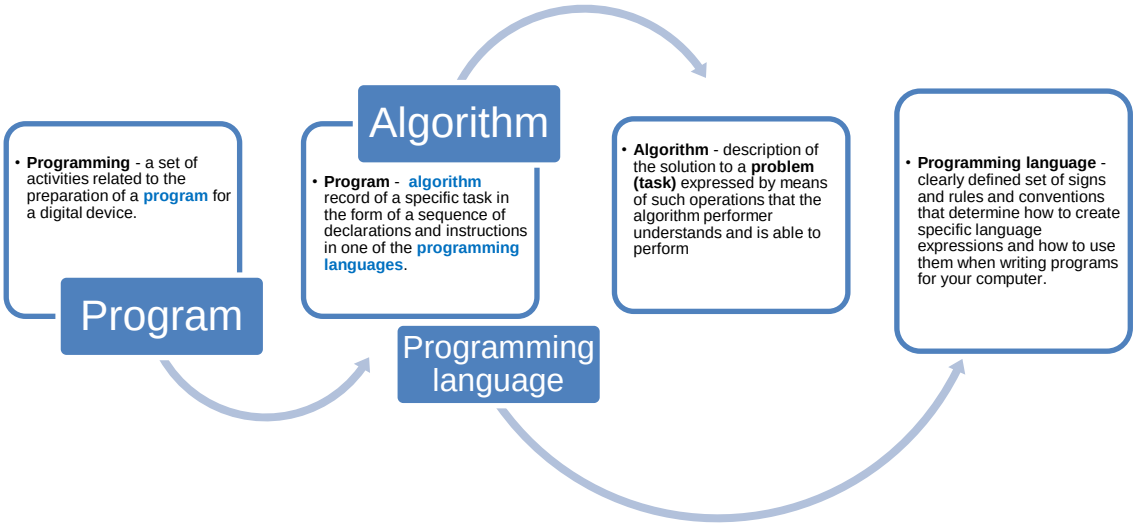


Figure 17 - Links between the terms of program, programming, algorithm and programming language

Source: own elaboration based on (Encyclopedia, 1988)

Programming will therefore be the recording of an algorithm, conveying the description of a solution to a given problem, with the use of a description content medium – the programming language. **Programming languages** will be introduced further on in this handbook. It is worth mentioning, however, that like any language, the programming language has its own alphabet, which are most often letters of the Latin alphabet, Arabic numerals and signs of arithmetic and logic operations, as well as special characters. Commands recorded by the programming language alphabet are translated by the computer into its internal language, which takes the form of a binary code. In such a record there are only two numbers "0" and "1" and it is through them that information is transferred to the computer. Because with two symbols - i.e. "0" and "1" – binary digits, you can pass on only two pieces of information, the binary system combines binary digits into groups to transmit more detailed messages (Wałaszek, 2018). In this way, by means of a group of binary digits, expressed by a compound numerical symbol, the computer can perform the requested operation. For example, single letters of the alphabet will be saved as follows (BiMatrix, 2018):

„a” - 01100001

„b” - 01100010

An example word „hello” – will be marked „01001000 01100101 01101100 01101100 01101111”, and the phrase „hello there!” – „01001000 01100101 01101100 01101100 01101111 00100000 01110100 01101000 01100101 01110010 01100101 00100001”.

Using the programming language you can create any command, so that a computer can proceed according to that instruction. It is important that the commands are algorithms with (*Encyclopedia,* 1988):

- a clearly defined beginning,
- an indicated operation – action where to start using the algorithm,
- precise instructions for performing further actions, given in exact order,
- ending date (deadline for algorithm realization).

It should be noted, however, that the introduction of a problem solving recording to your computer is just **coding**, not **programming** (Soukup, 2015). Programming is a much broader term and includes the analysis of the problem to be solved and the idea to use available digital technologies in finding this solution.

2.2 History of programming

The history of programming goes back long before the first computer was built. Key to its creation was the figure of a lady brought up in the home of one the greatest poets of Romanticism, while the first programming language was recorded in a plain diary. The history of programming and computers is related to the history of mathematics, which does not seem so far away. Already in antiquity, various types of abacus or plaques were created to facilitate calculations or measurements. In the 17th century, J. Neper, the creator of the logarithm developed the so-called Napier's bones used to calculate logarithms and significantly shortening the time of making calculations (Encyclopedia, 1988). The following years brought the development of the first calculating machines, constructed separately by W. Schickhard (in 1623), B. Pascal (in 1645) and G. W. Leibniz (in 1671). Unfortunately, none of these machines could be called automatic. J. M. Jacquard returned to the idea of constructing such a machine at the beginning of the 19th century, presenting punch cards for controlling the weaving loom at the exhibition in Paris (Heath, 1972). This technique was used by Ch. Babbage, who is considered the inventor of an automatic machine. In 1833 he designed a punch card-based analytical machine, which was de facto not created due to technical and financial difficulties. Nevertheless, his ideas prompted A. A. Lovelance, the daughter of the poet G. G. Byron, to develop the idea of creating interchangeable programs in this type of machines. For this reason, on the basis of her notes, A. A. Lovelace is perceived as the author of the first computer program. She herself speculated on the possibilities of playing chess with digital machines, or the possibility of singing through them (*The history of computer programming*). Further work on the development of digital machines led to the creation of an electronic tabulation system, thanks to which machines could read data. This was thanks to H. Hollerith, who developed and applied this system in calculating machines. After successes in using his machines during the census in the USA, he founded in 1996 the Tabulating Company Machine, which dealt with the mass production of calculating machines. Many years later, this company together with others gave birth to the IBM® concern. The subsequent development of computer science and programming itself is multithreaded, although to a large extent the story is related to the course of the Second World War and activities undertaken

in the scientific field to solve logistical and military problems. The development of the operational research trend and the need to conduct increasingly complex and extensive calculations caused that in 1942 the first digital computer (Atanasoff-Berry Computer - ABC) was developed to calculate linear equations and - a year later - the Colossus computer, used for decrypting German messages (*The history of computer programming*). Subsequent programming works took place on the EDSAC device, designed and created in 1949, the continuation and improvement of ENIAC and EDVAC computer projects, developed just after the end of military actions (*Encyclopedia*, 1988). A breakthrough event in programming history was the creation of the first high-class programming language - Fortran, which was created in 1954 by J. Backus working for IBM®. The development of programming theory and practice has introduced an increasing number of languages in use. After the success of the Fortran language, it was time for languages with higher degrees of universality, such as ALGOL 60, or with specific implementation goals as COBOL. The popularization of personal computers in the early 1980s turned out to be a breakthrough in this respect. The use of data transfer floppy disks provoked very large changes in this respect, also as to the development of their protection. In 1983, F. Cohen created the first computer virus transferred by means of floppy disks, only to show this possibility. Not everything in programming has been or currently is associated with positive goals. Any development, also in terms of programming, is exposed to the impact of adverse or even harmful factors. This is particularly visible nowadays, when computers and various types of digital devices have massively entered everyday life, not necessarily directly, but also as parts of other devices. Also the creation and development of the Internet network forces constant changes in the area of programming, and hence - in the way of thinking and looking at the emerging problems in the digital world.

2.3 Programmers skills and the process of developing them

Defining **skills** requires distinguishing them from the concept of **competence** with which they tend to be identified. The latter is defined as **the ability to do something**, depending on knowledge, skills, abilities and the degree of belief in the need to make use of this ability (Jurgielewicz-Wojtaszek, 2012). It is a broad piece of terminology, encompassing the concept of **skills**. They can be defined as **practical knowledge and proficiency in something** (*Dictionary*, 2018) or a **coherent set of abilities to do something** (Routier, Mathieu, Secq, 2001). **Skills** are one of the components of **competence** and affect the ability to complete an action based on the received task (Figure 18).

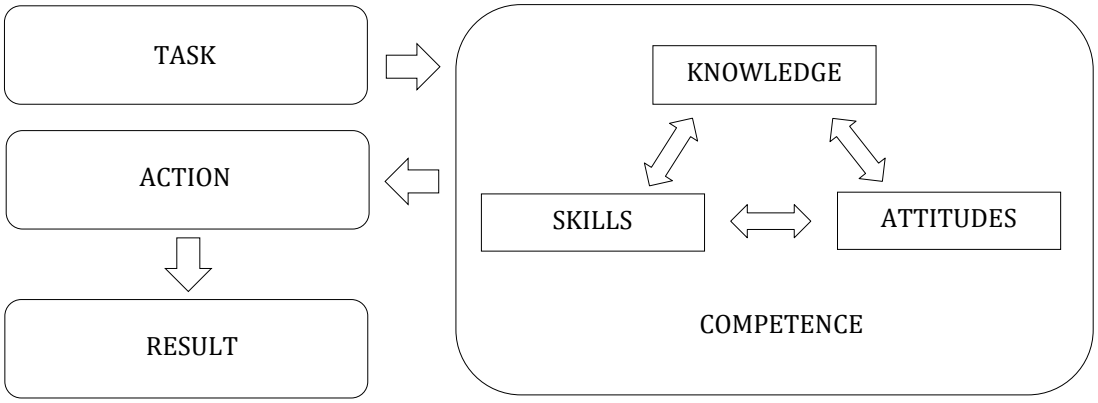


Figure 18 - Competence definition in the behavioural approach

Source: (Adamczyk, 2014)

As R. Boyatzis (1982) wrote that a programmer working in the comfort of his home will need different skills than a neurosurgeon who performing complicated surgery together with other doctors. Although both should have the ability to diagnose a problem, logical thinking or to seek additional information in problem solving, the first should have the ability to experiment and communicate with the program user, while the second must be able to make quick decisions and communicate well with his team. This means that each of them will have in addition to common skills, also those specific ones that they should develop. Figure 19 illustrates the links between a programmer and the product of is work, the device and the user.

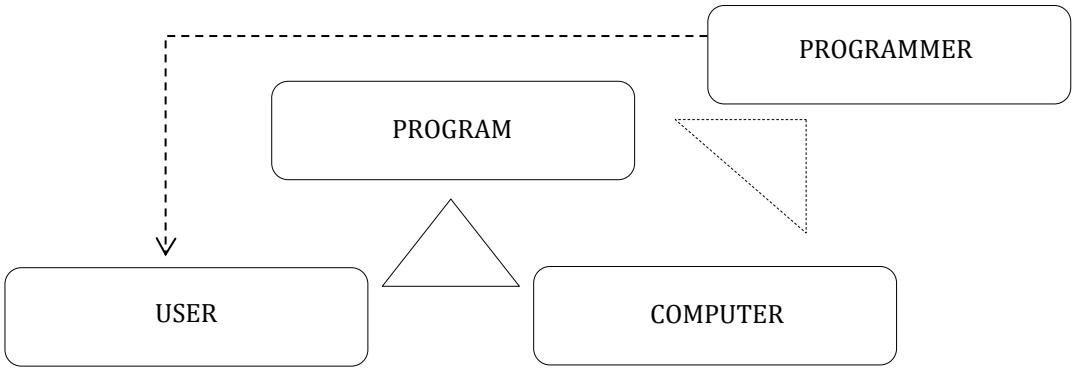


Figure 19 - Links between a programmer and program user

Source: own elaboration based on (Rogalski, Samurçay, 1990)

Defining the skills that a programmer needs, you can indicate the areas (competencies) that these skills will apply to. In the literature on the subject, a number of classifications of both competences and specific skills within each of them, was developed. Taking the current state of knowledge as a basis, research is often conducted to verify which of them are the most important in the programmer's work and which should be developed. Table 1 presents a summary of the most popular competencies among programmers and their groups of skills. For example, one of the last studies (Manawadu, Johar, Perera, 2015) indicated that out of the seven areas of competence, programmers

will have to primarily develop: *User Requirements, Software Development Process, and System Analysis and Design.*

Table 1. List of programmers' competencies and skills

Adamczyk, 2014	Turley, Bieman, 1994	Manawadu, Johar, Perera, 2015
Source code management (*Using proper naming and comments, using ready-made libraries and striving to achieve the highest possible portability*): – Applying patterns – The use of libraries – The use of algorithms – Using the IDE (Integrated Development Environment) – Use of portability **Knowledge management** (*Both effectively acquiring knowledge yourself and sharing it with others*): – Applying code writing rules – Learning from others – Effective knowledge acquisition – Sharing knowledge **Managing your own work** (*Self-organization of work, manifesting itself in meeting the established deadlines*): – Organization of own work – The use of versioning – Using tests **Requirements management** (*Customer orientation and creation of all solutions guided by the expectations of the end user*): – Customer orientation	**Task Accomplishment:** – Leverages/Reuses Code – Methodical Problem Solving, – Skills/Techniques, – Writes/Automates Tests with Code, – Experience, – Obtains Necessary, – Training/Learning, – Uses Code Reading, – Use of New Methods or Tools, – Schedules and Estimates Well, – Use of Prototypes, – Knowledge, – Communication/Uses Structured Techniques for Communication, **Personal Attributes:** – Driven by Desire to Contribute – Pride in Quality and Productivity – Sense of Fun – Lack of Ego – Perseverance – Desire to Improve Things – Pro-active/Initiator/Driver – Breadth of View & Influence – Desire to Do – Thoroughness – Sense of Mission – Strength of Convictions – Mixes Personal and Work Goals – Pro-active Role with Management **Situational Skills:** – Quality – Focus on User or Customer Needs – Thinking – Emphasizes Elegant and Simple Solutions – Innovation – Attention to Detail – Design Style – Response to Schedule Pressure. **Interpersonal Skills:** – Seeks Help – Helps Others – Team Oriented: – Willingness to Confront Others	**Programming** *The ability of writing computer programs for multi platforms, devices and channels with ability to adapt with any programming language.* **Computer Science** *Ability to integrate the principles of computer science in order to produce tangible, physical artefacts.* **Systems Analysis & Design** *Ability to examines complicated industrial and business operations in order to find ways of improving or solving them systematically* **Software Development Process** *Ability to effectively use a software development process or life cycle which is a structure imposed on the development of a software product.* **User Requirements** *Ability to understand the expectations of the users of software and deliver them as expected.* **Software Tools Usage** *Ability to use an array of software tools or build your own tools to bring productivity into tasks done in software engineering.* **Delivering Quality Code** *Ability to deliver quality code adhering to best practices and principles with abstraction in mind and ensuring defects are not injected.*

Source: own elaboration based on (Adamczyk, 2014; Turley, Bieman, 1994; Manawadu, Johar, Perera, 2015)

S. Goel (2010) indicates that currently employees working in the IT sector should have skills in the following areas:

1) Problem solving,
2) Analysis/Methodological skills,
3) Basic engineering proficiency,
4) Development know-how,
5) Teamwork skills,
6) English language skills,
7) Presentation skills,
8) Practical engineering experience,
9) Leadership skills,
10) Communication.

It is worth paying attention to the knowledge of English, a non-technical skill, but a key factor when working in the IT environment. This author of *Curriculum Guidelines for Degree Undergraduate Programs in Software Engineering* (2004) points out that a programmer should be able to:

1) show mastery of the software engineering knowledge and skills, and professional issues necessary to begin practice as a software engineer,
2) work as an individual and as part of a team to develop and deliver quality software artefacts,
3) reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations,
4) design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns,
5) demonstrate an understanding of and apply current theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation,
6) demonstrate an understanding and appreciation for the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment,
7) learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development.

The indicated areas of competence of programmers and specific skills that they should possess relate to various aspects of the work of software developers. It is clear that the work of a professional programmer is far from the stereotypical image of a man leaning over the keyboard, working alone, who only communicates with the computer. In this respect, the development of the skills of programmers is indispensable, all the more that their work environment changes very quickly. At the moment, the skill of writing codes (colloquially, translating commands related to the task into computer language) is not enough, but strong **communication skills** are required to understand

the need and be able to transmit a specific command to a program responding to it. Similarly, due to the turbulences of the environment, the multitude of tasks and needs, the programmer should have good **time management skills** and be **flexible** in regards to the tasks being performed. Emphasizing soft skills in the work of a programmer does not equal the reduction in relevance of technical skills, but indicates an increasing need for employees with good programming skills.

The already mentioned document: *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (2004) and its later version (2014), as well as *Curriculum Guidelines for Graduate Degree Programs in Software Engineering* (2010), are interesting proposals of programs educating future programmers and are developed by IEEE Computer Society (https://www.computer.org/) together with the Association for Computing Machinery (https://www.acm.org/). These programs are based on *The Guide to the Software Engineering Body of Knowledge* (SWEBOK Guide) and describe the current state of knowledge in the 15 areas of competence related to software engineering. They can be the basis for supplementing the already existing qualifications by programmers or - due to their construction - should be taken into account in the process of formal education of programmers.

2.4 Variables and constants

The programming can be seen as a way of "translating" the real world cases to a form of a software code. For this process, however, there is a need to model and mimic what is known from the surrounding environment. There is a need for the code to be able to name specific things and perform various operations on these characteristics. Let us assume for a moment, that we are dealing with the application that is expected to store basic information about your friends including their first name, last name and birthday date. Not focusing too much on details, we would need appropriate places in the code that are able to store this information and be able to perform various operations on a set of data. If we are considering the aforementioned three characteristics, it is safe to assume that they will vary from person to person. In other words, they will have a variable characters, and this is precisely what variables are used for.

In computer programming, *a variable is a location capable of storing temporary data within a program. This data can then be modified, stored, or displayed whenever needed* (https://www.computerhope.com/jargon/v/variable.htm).

Coming back to our example, variables can therefore be used to store first name, last name and birthdate as explained above. As a result, if we were to enter a new person, the software would need to ask us for this information, storing them internally as variables. Perhaps you are wondering, why are we considering things such as a birthday date as something that is variable? You should not be mistaken by the fact

that the birth date cannot change in real life. The "variability" here relates strictly to the way the code handles the information.

Variables can be named differently, but it is a general rule that their names should start with a letter, and not a digit (there are few exceptions but since it is beyond the scope of this handbook, feel free to discover this subject on your own!). In addition to that, each programming language might have different conventions regarding how the variables can be named, which names are reserved (and therefore cannot be used), if the variables are preceded by any special character (like $) and other useful information in this respect.

For now, and also for future chapters of this handbook, we will use a so-called pseudocode, which is the code that is not the actual programming language, but rather mimics its semantics. Going back to our example with people and their birth dates, we might conclude that our code (or rather pseudocode) that does the assignment of the variables could be structured as follows:

```
firstname = "John"
lastname = "Doe"
birthday = "1998-02-20"
```

From the above example we can see, that an equal sign was used to perform the actual assignment. Starting from the left, we have a variable name, then the equal sign and then the value. Doing so will allow us to use `firstname` in other places of the code, for example to show a message that `<firstname> <lastname> has birthday at <birthday>.`

At this stage it is worth to mention, that variables can be named very differently, as there are different conventions imposed by the programming language, but also different approaches towards naming the variables. Other examples of `firstname` variable could be `$firstname`, `firstName`, `first_name`, `FirstName` and so on. In any case, it is important that the variable does inform what information it is holding. The variables in the above example could have been named as a, b and c, but it would be then difficult to figure it out in other part of the code.

The other important aspect regarding the variables is that in some of the programming languages you are required to specify their type as well. The type can be understand for example as text, number, date or Boolean. The Boolean variables can store only two values: `true` or `false`. It is one of the most frequently used type of variables. For example, we could have a variable named `birthdaytoday`, which would be true if someone from our list have their birthday today, or `false` otherwise.

There is also a special variable that is called an array. It represents a collection of various elements, and can be very simple such as `fruits = [apple, pear, plum]`, or more complex:

```
food = [
```

```
    fruits => [apple, pear, plum],
    vegetables => [asparagus, potato, tomato]
]
```

With the use of arrays we can store elements in hierarchical ways, and then use them in our code by writing `food[fruits][0]` which has the value apple. This is because we have referred to the first element from the fruits array contained in our food array. You might have noticed that the index we have used is equal to 0. This is because in programming we usually keep the numbering that is referred to as 0-based indexing. At this stage it is enough for you to remember about arrays and their basic structure. For more information, make sure to have a look at our "Further reading" section.

Now that we have touched briefly the variables, it is time for us to move to constants. The constants can be seen as variables with one important exception. Although they share similar concepts (naming conventions, types and so on), their purpose is to hold the same value across the whole code execution. In other words, they do not change. The constants are usually written in capital letters, so if we were to limit the maximum number of birthday entries our code can store, we could have used a `MAX_RECORDS = 100` constant for that. The use of constants simplifies the process of writing the code. Taking the example of `MAX_RECORDS`, this constant will be used probably many times across our code. If, at any time in the future, we decide to raise the limit of the records to 200, we only need to change the value of that constant, and the required changes will be immediately available in all other places of the code.

2.5 Objects

When we are talking about objects in the programming, we are referring to them in the scope of a so-called object oriented programming. Put simply, a programming object carries information about object from reality, and it can be virtually anything. In the subchapter where we discussed variables, we were only talking about things such as first name, last name and birthday date. These obviously describe a person. Can it then be, that a person can become an object when we are talking about programming? Well, definitely! A person can become an object that has its own *properties*. These *properties* of an object are kind of variables that are associated with this specific object. We can add more properties, such as for example height, weight, gender of a specific person. Whichever characteristics of a person are needed in our software can be implemented with the use of properties. This brings us to a question – why can't we simply use variables? What is the reason for creating objects?

Without a doubt, the object-oriented programming revolutionised the way we write the code today. This concept dates back to the 1960s and Simula 67 programming language created by Ole-Johan Dahl and Kristen Nygaard from the Norwegian Computing Center in Oslo. These researchers needed a way to simulate multiple objects from real-life in a way that these objects can be responsible for their own behaviour. While the expected

outcome could have been achieved simply by using variables, that proved to be highly inefficient. This brings us back to the very core of object-oriented programming in which objects can not only have their own properties, but, unlike variables, are able to take specific **actions**.

Let us move back to our example, when a person is turned to an object in a programming sense. What actions can you think of when you are considering interacting with a person in real life situation? What about something as simple as introducing your name to other people (other objects)? The simplest way of explaining the concept is to add an action (also referred to as method) named `introduce` that would enable the object to say `Hello! My name is <firstname>`. Now you can probably see the connection between the property of an object (firstname) and the method (`introduce`). Inside our code, if we had an object representing a person, we are able to interact with it with the use of methods. More examples of methods? Sure! How about `whenIsYourBirthday` that would say `My birthday is at <birthday>`. Too easy? Well, now we can also add a new method `isYouBirthdayToday`? This method would need to perform one additional operation. It would sort of interact with the environment by first checking what date it is today. Then it would need to compare this date with object's own property birthday. Depending on the resulting match, the object could say `Today is my birthday!`, `It is not by birthday today`, or, to complicate things, `My birthday will be in X days from today`, `X` being the calculated difference between today's date and object's birthday property.

Object oriented programming does bring an extra layer of complexity to the code, but as a reward it gives a great flexibility on how we can reflect the real world objects in our code. There is also a number of other concepts connected with object oriented programming that we will not cover in this handbook, but if you are interested, please make sure to have a look at "Further reading" section which contains top picks of books on this subject.

2.6 Operators

It is time for us to move to a bit more serious code. In this subchapter we will discuss operators that are used to compare two elements. Is 4 greater than 5? Is 3 greater than or equal to 1? What is the outcome if we multiply 5 by 4, and what if we divide 10 by 2? Is the today's date equal to my birthday date? All these questions (and an infinite number of other examples) can be answered with the use of operators.

The operators can generally be divided into three groups that we will briefly describe. We have arithmetic, relational and logical operators.

One more important clarification before we move forward relates to the operands. The operators are in the middle between two elements, as such: `a operator b`. In this case, `a` and `b` are so-called operands, and the operator is in the middle. For the purpose of clarity, the two operands are referred to as the left operand (in this case: `a`) and the right operand (in this case: `b`). What is this operator that sits between two operands? Let us find out!

The first type, arithmetic operator, is a simple mathematical operation that we all know from our early education, such as addition, subtraction, multiplication, division and the remainder. These are the most popular arithmetic operators that are additionally presented in table 2 below.

Table 2. List of the most popular arithmetic operators – part I

Operator	Description	Left operand (a)	Right operand (b)	Outcome
`+`	Adds two operands	`4`	`5`	`4 + 5 = 9`
`-`	Subtracts right operand from the left operand	`4`	`5`	`4 – 5 = -1`
`*`	Multiplies both operands	`4`	`5`	`4 * 5 = 20`
`/`	Divides left operand by the right operand	`18`	`3`	`18 / 3 = 6`
`%`	Calculates the remainder of dividing the left operand by the right operand	`20`	`3`	`20 % 3 = 2`

Source: own elaboration

Where these arithmetic operators can be used? In plenty of cases! If I decide to read 10 pages of a book on object-oriented programming each day, how many pages am I going to read in one month? And what about in one year? What is the sum of all expenses I made last week? All of these operators can be used together in one equation, just like you would do on a paper. We will get back to this later, but first let us go through the remaining operators.

The next group considers relational operators, and, as the name suggests, these are used to compare two operands. By using relational operators you are able to tell what is the relation between two operands. These can be treated as asking a question to get a yes/no answer, which in programming is expressed as a Boolean value (just as a reminder, they return either `true` or `false`). Is person `a` older than person `b`? Is the number of people from our code whose name starts with letter `C` greater than those whose name starts with letter `D`? Let us first look at the table 3 below which summarises the most frequently used operators.

Table 3. List of the most popular relational operators – part II

Operator	Description	Left operand (a)	Right operand (b)	Outcome
`==`	Checks if both operands are equal	4	5	`4 == 5` `false`
`!= or <>`	Checks if both operands differ from each other	4	5	`4 != 5` `true`
`>`	Checks if the left operand is greater than the right operand	4	5	`4 > 5` `false`
`<`	Checks if the left operand is smaller than the right operand	4	5	`4 < 5` `true`
`>=`	Checks if the left operand is greater than or equal to the right operand	4	4	`4 >= 4` `true`
`<=`	Checks if the left operand is smaller than or equal to the right operand	3	4	`3 <= 4` `true`

Source: own elaboration

The last type of operators are logical ones. In this category we can include three most popular logical operators that are present in almost every programming code. We have also promised to get back to more complex operations that the operators are capable of, so there we go! First, let us have a look at the table 4.

Table 4. List of the most popular logical operators – part III

Operator	Description	Left operand (a)	Right operand (b)	Outcome
`&&`	Checks if both operands are true (logical AND)	`true`	`false`	`true && false false`
`||`	Checks if at least one of the operands is true (logical OR)	`true`	`false`	`true || false true`
`!`	Reverses the logical value of the operand, so true becomes false and false becomes true (logical NOT)	`true`	`false`	`!(true && false)` `true`

Source: own elaboration

Please pay special attention to the last example, in which we have used `true && false` at the first place. That would, according to the first example, result in false. However, with the use of a logical operator `NOT`, we changed `false` to `true`. Coming back to our example, that could be useful in case we are looking for people who are older than 20 years old, but do not have their birthday today. Such people would be marked as true Boolean value. How could we possibly write such code using the operators we know so far? Let us try!

```
!(olderthan20 && birthdaytoday)
```

Of course we would need to include two Boolean variables here, named as `olderthan20` and `birthdaytoday`. Now you can also see that we are using round brackets to group operators. This allows us to precisely evaluate even more complex examples. How about selecting people who are older than 20 AND their first name starts with A, OR are older than 25 AND their first name starts with B?

```
(olderthan20 && startswithA) || (olderthan25 && startswithB)
```

Using round brackets gives us unlimited possibilities of playing with different Boolean values to achieve the condition that is required. That would be more or less it when it comes to very basic introduction to various operators. Please note, however, that the results of using the operators is a different aspect. We do not calculate or evaluate the statements just for the sake of doing it. Instead, we use them to tell our code what should be done given specific circumstances. This is what is referred to as decision statements, described in the next subchapter.

2.7 Decision statements

We make various decisions on a regular basis. When it is raining, we take our umbrella with us before going out. In other words, based on different information that we have, we might act differently. This is precisely what decision statements are used for in our code. If the person A has their birthday today, we should wish them happy birthday, right? In the previous sentence we have already used a conditional statement, which in most programming languages is expressed by IF. This IF statement is evaluated logically, so in our code we could write something such as:

```
if (personAHasBirthday)
    happybirthday()
```

What does it say? It can actually be read as-is, so if person A has birthday, "do" `happybirthday`. Do not worry about the round brackets after `happybirthday`. For now just note that it is a function that is called and is expected to do something. We will get back to functions soon. The code also carries important information. The conditional `IF` statement assumes, that the `happybirthday` will only be "executed" if the condition that is evaluated is true. In other words, if person A does not have their birthday today, the code will not wish happy birthday. That should be logical. Usually only the next line of code after the `if` statement is executed. But wait, what if we really wanted to give this person a small gift? That would mean that we need to not only wish them happy birthday, but also give a small gift. This can be achieved using curly brackets, like shown on the sample code below.

```
if (personAHasBirthday) {
    happybirthday()
    giveSmallGift()
}
```

Now you can clearly see, that the conditional if statement, and hence our code, is indeed going to make these two actions. Not only the wishes will be passed on, but also a small gift.

Should our code be that impolite to all those people, who are not lucky to celebrate their birthday today? In its current shape, the code cares only about birthday people. The others are left with nothing. This can be changed, with the use of `ELSE` instruction. Let us have a look at our updated code.

```
if (personAHasBirthday) {
    happybirthday()
    giveSmallGift()
} else {
    howareyou()
}
```

Now it looks better! If the person that we (or rather our code) are considering does not have their birthday, simply ask them how they are doing today. Hopefully that is polite enough!

Although usually the `ELSE` keyword does not need the curly brackets right now (as it has only one instruction to execute), it is usually a good practice to include it for clarity.

Let us now have a look at more complex case. We would like to make groups of people we considered earlier based on their age. There will be four groups based on the age:

- Group A – people aged 21
- Group B – people aged 22
- Group C – people aged 23
- Group D – all the other people

So far we have learned about operators and now we are learning about conditional statements. We have a list of people, and now we need to make the proper assignments to groups based on their age. Please have a look at the below sample code for details.

```
if (age == 21) {
    assignToGroupA()
} else if (age == 22) {
    assignToGroupB()
} else if (age == 23) {
    assignToGroupC()
} else {
    assignToGroupD()
}
```

It would be a good idea to stop at this point for a while. What you see above is the set of 4 conditional statements. When the code is being executed, it starts from the top and evaluates the age of a person against the conditions expressed as relational and logical operators. If the first condition is not met (in other words the person is for example 23 years old), the code moves to the next statement. Please note, that because we wanted to consider all these conditional statements as one big statement, we used else if. This instruction, unlike simple else, allows us to give more statements to be evaluated. The "single" else condition has been moved to the very end of the code in order to match all the people who were not falling in the age ranges defined in previous conditional statements.

We have to agree however, that even though there are only four statements, the code does not look too clear. What if we had even more complex case to be solved? What if we had 10, or 20 cases? Great, we have just introduced another very useful word: a case! Cases can also be used to organise your code if these are connected with a so-called switch instruction!

The switch-case construction is used specifically for this purpose when we have to evaluate a number of conditions. The equivalent piece of code to the one we discussed above could be written as presented below.

```
switch(age) {
    case 21:
        assignToGroupA()
        break
    case 22:
        assignToGroupB()
        break
    case 23:
        assignToGroupC()
        break
    default:
        assignToGroupD()
}
```

Now we are telling our code to perform the required assignments using a switch-case construction. At the very top we are explaining that we are going to "switch" the code based on the age. Then, below, there are cases. If the age matches given case, this case code (here – the assignment to the group) is executed. You probably noticed the break

instruction. This instruction is telling our code to stop there and forget about other cases. Why is that so? How would you understand the following piece of code?

```
case 21:
case 22:
case 23:
assignToGroupX()
break
```

With cases constructed in such a way, our code will assign to group X people, who are aged 21, 22 or 23. This is because we have not provided any other instructions under the first two cases, and we have not used the break instruction either. This might be useful as well, depending on what we are trying to achieve. In our case, however, we do need to use the break instruction to be able to assign people to the required groups.

The conditional statements are relatively simple to understand, but only through hands-on experience you will get more confidence even with the most complex code. It is time for us to move further, starting with loops described in the next subchapter.

2.8 Loops

Loops in programming can greatly simplify the code. These instructions are used to run the same block of code multiple times, depending on the needs. If you are inviting all your friends to your place, you would need to individually greet all those who arrive. Our code could do it very easily, by calling a function (please be patient, functions are described in the next subchapter) shown below.

```
greetAPerson()
```

This is great, but can you imagine the code if you had 30 people to greet?

```
greetAPerson()
greetAPerson()
(27 more lines hidden)
greetAPerson()
```

If you think, that there must be a better way to do it, you are right! This is a perfect example where we can use loops.

In programming languages there are different types of loops. They differ from each other when it comes to the construction, but in principle they are designed to do the same – repeat execution of the same block of code. Each such execution is referred to as iteration. As a start, let us consider a simple `FOR` loop.

A `FOR` loop is used to execute a specific code as long as the condition that is evaluated equals to Boolean true. Please consult the example shown below for details.

```
for (i = 0; i < 30; i++) {
greetAPerson()
}
```

As there are new elements, we will go through them one by one. In the first line you can see that this loop is defined by three blocks separated by semicolons. The first one is evaluated when the loop is about to be executed. In this case, the variable `i` will get value equal to `0`. This variable will be used to evaluate whether the loop should continue or stop. According to the code, the loop will continue until `i` reaches `30`. The evaluated expression, `i < 30` will be evaluated as true for `i` equal to `0, 1, …, 29`, but when `i` reaches `30` it will become `false`, as `30 < 30` statement is not `true`. In other words, thirty is not less than thirty. How does the loop know about the value of `i`? It is the third block which says `i++`. This is a post-increment expression which increases value of `i` by `1`, and, coming back to arithmetic expressions, is equivalent to `i = i + 1`.

Important information regarding the for loop and also other loops is that in many cases so-called steering variables are used, and it is a common practice to name them as short, one letter variables like `i`, `j` and `k`.

The `FOR` loop is useful if we precisely know how many iterations do we need. In our example we knew that we were expecting 30 people who should be greeted. The loops, however, can also be combined with other instructions. What if we wanted to iterate over the list of people, and **stop** at certain point that is not known before we actually run the code? Let us assume, that we want to iterate again through the list of 30 people, and stop if we find a specific person whose name is Jane.

```
for (i = 0; i < 30; i++) {
    if (firstName == “Jane”) {
        break;
    }
}
```

The `break` instruction is the same that we discusses while explaining the `switch-case` statement. The code will stop at the first person whose first name is Jane. This will allow us to stop iterating unnecessarily through other people, as we have already found Jane.

Another example might again consider Jane. We are still greeting all the people, but when we are greeting Jane, we want to give her a small gift, as she has her birthday today. A sample code might look as the one presented below.

```
for (i = 0; i < 30; i++) {
    greetAPerson()
    if (firstName == “Jane”) {
        giveSmallGift()
    }
}
```

This code greets anyone, but for Jane another action is expected – presenting her a small gift.

Last example will illustrate the situation in which we know that Jane is unfortunately not coming. In this case, we would like to be able to skip greeting her.

```
for (i = 0; i < 30; i++) {
    if (firstName == “Jane”) {
        continue
    }
    greetAPerson()
}
```

As you can see inside the piece of code that is executed at each iteration, we have a conditional statement. If the person that we are considering right now is indeed Jane, we tell our loop to continue. This is the way to tell the loop to forget the remaining code and move on to another person. If the first name is not Jane, this conditional statement will be evaluated as Boolean `false`, and as a result, `greetAPerson` will be executed, and this is precisely what we wanted to achieve.

You might be wondering if `FOR` is the only loop you can include in your code. The answer is no! It should be pointed out, however, that not all types of loops are present in each programming language. Anyway, let us look at two more loops: `WHILE` and `FOREACH`.

The `WHILE` loop will execute the same block of code for as long as the condition that is being checked evaluates to Boolean `true`. In other words, we are telling our code to act in a following way: `IF <condition> REPEAT <code> AND go back to beginning`.

As the condition that is to be evaluated is of Boolean type, it means that we need to get either true or false value that will tell us if the code block inside the `WHILE` loop needs to be executed. For example something like that:

```
while (! allPeopleArrived) {
    keepGreetingPeople()
}
```

Assume that we have a `allPeopleArrived` variable that is set to `false`. New people are coming to our place, but this variable will be false up to a moment where all guests have arrived. So basically our simple loop is executing the `keepGreetingPeople` function UNTIL all people have arrived – and that would be reflected by `allPeopleArrived` changing its Boolean value from `false` to `true`. You must have also noticed the exclamation mark before the variable name. As you remember, this is a `NOT` logical operator which reverses (or flips) the value from `true` to `false` and from `false` to `true`. When we are starting our loop and the `allPeopleArrived` variable is `false`, we need to negate it in order to be `true`, as only this will make the block of the code being executed. Finally, when `allPeopleArrived` changed its value to `true`, along with the negation the whole expression will evaluate to `false`. This will result in `WHILE` loop not being executed any longer – so precisely what was needed.

There are also other ways of writing the WHILE loop, but it depends on the specific programming language. Some of them follow the structure WHILE <condition> DO <code>, so there is one extra "do" word that explicitly says how the code should behave. From this point we can also mention very briefly the other construction, which, instead of WHILE <condition> DO <code> inverses its behaviour, so it becomes DO <code> WHILE <condition>. Our example could therefore be modified slightly as such:

```
do {
    keepGreetingPeople()
} while (! allPeopleArrived)
```

Principally, the behaviour is very similar to the one we discussed above. We would like to greet our guests up to the time when all of them have arrived, as it is indicated by our expression that is being evaluated to be either true or false. There is one important exception. The do-while construction will be executed at least one time, as the condition is evaluated only after the block of the code! This might be confusing. Imagine, that all our guests have arrived. Now, accidentally, the same code is being executed. Remember, that the allPeopleArrived variable has the value true, as all our guests are there. Looking at the first while example, what is going to happen? The expression will be evaluated to false (allPeopleArrived is true, but together with the NOT logical operation we are going to get false). False value will therefore prevent the while loop to execute the block of the code, so we are not going to greet people for the second time.

The second example, however, will actually DO greet people, only to notice after the first greeting that it was not necessary. So this is the main difference between two while loops types.

The DO-WHILE construction might be useful in other cases. Let us assume that you know that there is one person who has birthday today, and you would like to ask all your guests, one by one, if this is their special day or not. We will talk to each person one by one. The following code could illustrate this situation:

```
do {
    askTheQuestion()
} while (! personFound)
```

Before we even start talking to people, our variable personFound will be false. We ask the first person if this is their birthday today. If not, we are not changing the value of the personFound variable, but instead we are going to the next person. At some point we will get the positive answer to our question, and we will change the value of personFound from false to true. As a result, right after we have the answer (and the person we were looking for), the while statement will be evaluated to false (personFound with value of true together with NOT logical operator will give us false), and that would stop the while loop from next iterations. It might also be, that the first person we ask is the one we are looking for. In that case, the question will

be asked and immediately after the very first iteration we will be able to stop asking other people the same question.

The last type of loop we would like to have a quick look at is the `FOREACH` loop. This loop is meant to be executed against a collection of elements, and the code is expected to do something for each element found. We are still with our guests, and we would like to iterate through all of them just to see if the phone number we have in our address book is still valid or needs to be updated. Let us assume that our guests will be present in one `allGuests` variable. The sample code is presented below.

```
foreach (allGuests as aGuest) {
    verifyPhoneNumber()
}
```

What can be seen here is that we are indeed going through all our guests, but in each iteration we are looking only at one specific guest that will be assigned to the `aGuest` variable. It means that out of a bigger collection of all guests we can focus on one specific person. For now we will simply assume that the `verifyPhoneNumber` function checks if the phone we have is correct and update it if necessary. If someone does not have a phone number, we can exclude such people with the use of continue statement as you have already learned. Please consult the example presented below.

```
foreach (allGuests as aGuest) {
    if (aGuest hasPhoneNumber) {
        verifyPhoneNumber()
    } else {
        continue
    }
}
```

In the above example someone has a phone number, we are going to verify it, and continue to the next person otherwise. If you think that the continue statement is not quite needed here, you are right. It would be enough to wrap the `verifyPhoneNumber` function in an if statement and the code would work in the same way. In some cases though it is needed to continue immediately to the next iteration of the `foreach` loop, and this is what the continue statement can be used for.

The beauty of the programming is that the same goal can be achieved in a number of ways. If we go back for example to the sample code in which we asked people if they have their birthday today using a do-while construction, we could approach it in a similar way using different loops, such as `foreach`.

```
foreach (allGuests as aGuest) {
    if (aGuest hasTheirBirthday) {
        personFound = true
        break
    }
}
```

In this case we are iterating through all our guests, but once the person we are checking in the current iteration has their birthday today, we will set `personFound` variable to `true` and quit the whole `foreach` loop using `break` statement.

This is something that you should keep in mind. The programming gives you the ability to solve the same problem using a number of methods. Some of them are better suited for specific purposes, some of them can be executed faster, and obviously they have their own advantages and disadvantages. Only through devoting your time to learning the very nitty gritty of the programming and actually experimenting on your own you are able to become fluent in the programming language of your choice.

2.9 Functions

In this subchapter we will focus on functions that were mentioned above a few times. Functions are isolated parts of the code that are meant to be executed to perform specific action. They may take some data as input parameters, process them and finally return a result. In other words, we can have the input and the output to and from the function. If it is not immediately clear, we can use an example.

One of the functions we have used in the previous subchapters was `greetAPerson()`. How could we possibly write such function?

```
function greetAPerson() {
    say Hello!
    say How are you doing?
    say Come on in!
}
```

Now we can easily say what is hidden in this function. Each time this function is invoked, it will say these three phrases.

Functions are used mainly to organise the code, and to divide it into logical pieces that can be invoked from different places of our code. Functions can also take input parameters that are called arguments. What does it mean? It means that we can pass a variable to the function and use it internally. How about greeting people by their first name? Surely this is what would normally be done, but in its current shape, our function is not aware of the person we are greeting – let us change that!

```
function greetAPerson(aPerson) {
    say Hello aPerson->firstName!
    say How are you doing?
    say Come on in!
}
```

In the above example, we are passing an argument as `aPerson` variable. Thanks to that, inside our function we can refer to our guests by their first name. Please not that we have used `->` sign that is expected to get the first name of the person that our

function just got as the parameter. At this point we could mention that there are different ways of accessing the first name, depending on the variable we are dealing with. Usually, the `->` sign will be used to access properties of an object. In other cases it might be that we will use `aPerson['firstname']`, or `aPerson::firstname` – it all depends on the programming language you are working on and also on the type of variable.

Functions can also return values, which is of great help to the programmers. Let's have a look at our `verifyPhoneNumber()` function, with the following implementation:

```
function verifyPhoneNumber(aGuest) {
    if (aGuest->hasPhoneNumber) {
        currentNumber = aGuest->phoneNumber
        if (currentNumber != theNumberIHave) {
            updateEntry()
            return true
        }
    }
    return false
}
```

The new thing that you probably spotted is that there is a new word named `return`. The `return` statement is used to return the value from the function to whatever piece of code called this specific function. In this case we are only returning `true` or `false` Boolean variables, and these are used to let our code know if we updated the entry (true) or not (`false`). Please note, however, that in order to use the value that is being returned, you would need to assign the called function to a variable. Having in mind the above example, let us consider the next sample code.

```
wasUpdateNeeded = verifyPhoneNumber(aGuest)
```

In this case, we are getting the return variable back from the function. Once the function is executed, the `wasUpdateNeeded` variable will tell us if we needed to update our address book (the variable will have the Boolean value `true`), or our entry did not need any amendments (the variable will have the Boolean value `false`).

To wrap up, functions can be used to save our time and increase the clarity of our code. If you go through the presented examples once again, you would probably notice that all of these instructions contained within `verifyPhoneNumber()` function could have been implemented directly inside the code as we were going through `allGuests` in our `foreach` loop. Then, however, the code would become too clumsy and not easy to be read. In programming you should always focus on your code, as it has to be clear not only to you, but also to other people. As this is very broad topic that unfortunately goes beyond the scope of this handbook, please be invited to explore this area on your own. A good starting point might be the "Further reading" section.

2.10 Bibliography

Adamczyk M., (2014), *Rozwój kompetencji zawodowych programistów w gospodarce opartej na wiedzy*, [‘*Development of professional competence of programmers in a knowledge-based economy’*], „Zeszyty Studenckiego Towarzystwa Naukowego” Akademia Górniczo-Hutnicza, w Krakowie, ISSN: 1732-0925

Boyatzis R.E., (1982), *The Competent Manager: A Model for Effective Performance*, New Jersey: John Wiley & Sons, Hoboken,.

BiMatrix, (2018), *Tłumacz tekstu i kodu binarnego,* [http://dbm.org.pl/strony/kod_binarny.php]

Goel S., (2010), *Design of interventions for instructional reform in software development education for competency enhancement*, Jaypee Institute of Information Technology,

Heath F. G., (1972), *Origins of the Binary Code*, “Scientific American”, Vol. 227, No. 2.

IEEE & ACM, (2004), *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*.

IEEE & ACM (2014), *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.*

Jurgielewicz-Wojtaszek M., (2012), *Ocena poziomu kompetencji trenerów zarządzania w zakresie nauczania osób dorosłych*, [‘*Assessment of the level of competences of management trainers in the field of adult education*’] [in:] Kuźniak A. (ed.), *Vedemecum Trenera II. Tożsamość Zawodu Trenera Zarządzania*, Kraków: Księgarnia Akademicka.

Encyklopedia szkolna – Matematyka. [‘*School Encyclopedia - Mathematics*‘] (1988), Waliszewski W. (ed.), Warszawa: Wydawnictwa Szkolne i Pedagogiczne, ISBN 83-02-02551-8.

Manawadu C.D. Johar M.G.M., Perera S.S.N, (2015), *Essential Technical Competencies for Software Engineers: Perspectives from Sri Lankan Undergraduates*, “International Journal of Computer Applications” Vol. 113, No. 17.

Rogalski J., Samurçay R., (1990), *Acquisition of Programming Knowledge and Skills*, [in:] Hoc J.M., Green T.R.G., Samurçay R., Gilmore D.J. (ed.), *Psychology of Programming,* New York: Academic Press Ltd., doi:10.1016/B978-0-12-350772-3.50015-X

Routier J.C., Mathieu P., Secq Y., (2001), *Dynamic Skills Learning: a Support to Agent Evolution*, [w:] *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1,* New York: ACM (Association for Computing Machinery).

Słownik Języka Polskiego [‘*The Polish language dictionary*’]. (2018), Warszawa: Wydawnictwa Naukowe PWN, (on-line: https://sjp.pwn.pl/)

Soukup B, (2015), *Nauka programowania zamiast przedmiotu informatyka w szkołach Podstawowych* [‘*Learning of programming instead of IT subject in primary schools*’], [in:]

J. Morbitzer, D. Morańska, E. Musiał (ed.) *Człowiek –- Media – Edukacja*, Dąbrowa Górnicza: Wyższa Szkoła Biznesu w Dąbrowie Górniczej, ISBN 978-83-64927-39-3.

The history of computer programming, graphic material [https://visual.ly/community/infographic/technology/history-computer-programming]

Wałaszek J., (2018), *Binarne Kodowanie Liczb* ['*Coding of Binary Numbers*'], [Online source: http://eduinf.waw.pl/inf/alg/006_bin/0009.php] (available on: 8th February 2018) GNU Free Documentation License

3 Didactics with the use of algorithmic and programming

(Wojciech Kolarz)

3.1 Basic assumptions of algorithmic and programming in school teaching

Modern school should, among other things, prepare a young person for life in a changing technological world, at the same time in a world based on the unchanging laws of nature. The ability to function in society is also important. In connection with the above, it will be necessary to be able to communicate, cooperate in a team, as well as the ability to describe phenomena, the ability to solve problems, including through the creation and use of models.

Today it is not a problem to get to the information. The ubiquitous technology gives you the opportunity to search information quickly and effectively. The problem is its creative use, the ability to build knowledge, and thus the ability to use it in solving problems, including atypical ones, deviating from standards.

An extremely important task of the school is to develop students' learning skills, creative thinking skills, overcoming the barrier of "intellectual laziness".

New technologies enable the use of new means of describing reality, and IT provides tools for computer modelling of reality and for solving problems. Knowledge of algorithms, basics of programming, programming language has become almost as important as knowledge of a foreign language.

In the current teaching model, two main trends clash: behaviourism and constructivism.

The creator of **behaviourism** was John B. Watson. This trend has dominated the pedagogical sciences. Many behaviourist views deviate far from the requirements of today's school, today's education. However, this trend cannot be completely ignored because some of its assumptions are still effective, of course often in a modified form, adequate to today's reality. Above all, the question is raised by the fact that behaviourism treats the student as a blank sheet of paper, on which the teacher writes down, in order that he has planned, content. Through appropriate selection of the consequences of behaviour, i.e. rewards and punishments, strengthens or weakens the student's behaviour. As a result, this approach leads to programmed teaching (based on an unchanging, rigid curriculum) and directive (frequent use of the method of giving, criticizing student's failures, appealing to authorities, giving guidance).

The basic assumption of **constructivism** is to treat the student as an active person who is the creator of his own knowledge. Knowledge cannot be passed on to the student (information can be provided), the student must build his / her own knowledge. The student's activity in the process of building one's own knowledge is important. Constructivism indicates that a student is not an empty vessel, which is filled by a teacher, but a person actively constructing his knowledge.

Constructionism is one of the trends of constructivism (it is a learning strategy as well as an educational strategy). Constructionism assumes that in the learning process, students actively engage in creating their own objects, events, ideas, ideas, which within the content they teach can be shared with others for the purpose of joint analysis and reflection. For constructionism, the social aspect is not insignificant - learning through team cooperation, discussion, exchange of views. As constructionism envisages the construction of material, external (existing outside of reason) representation of abstraction, it is particularly important that the external form of knowledge representation, proof of understanding issues and phenomena, is a computer program, and in principle an algorithm. In addition, by including the above in the reverse cause-and-effect hierarchy, the very process of creating an algorithm (computer program) requires deepening and understanding the problem (phenomenon). The process of creating an algorithm (program) is a process in which a person who creates an algorithm also creates his / her knowledge of the analysed problem (phenomenon), so we can talk about learning by creating algorithms, learning through programming.

Eight great ideas of constructionism according to Seymour Papert, South African mathematician and computer scientist, among others author of the LOGO programming language:

"The first great idea is learning through creation. We learn better when learning is part of doing something that really interests us. We learn the most effectively when we can use what we have learned to satisfy our needs or desires.

The second big idea is technology as a material. With technology you can create much more interesting things and create them you can learn a lot more. This particularly applies to digital technology.

The third idea is the idea of a lot of fun. We learn and work best when we enjoy it. But "we are happy", it does not mean "it is easy". The hardest fun gives you the most satisfaction. Our sport heroes work very hard to be the best in their discipline. The best carpenter finds joy in carpentry. The most effective businessman enjoys the difficult whipping of business.

The fourth big idea is the idea of learning how to learn. Many students believe that the only way to learn is that someone has to teach you. This is the reason for failures at school and in life. No one can teach you everything you need to know. You must take responsibility for your learning yourself.

The fifth great idea - give yourself time to do the job. Many students get the habit out of school that someone speaks every five minutes or every hour: do it, do that and now. If someone does not dictate to them what to do, they start to get bored. In life, it is completely different to create something really important; you have to learn how to manage your own time. This is the most difficult lesson for many students.

The sixth idea is the most important of all: there is no success without failures. Nothing really important works right away. The only way to success is to carefully analyse what and why it does not work properly. To succeed, you must free yourself from the fear of mistakes.

The seventh great idea - practice it yourself, what you recommend to students. We've been learning all our lives. Although we have extensive experience of working on projects, everyone is different and usually realizing the next one we cannot predict with all details in advance how it will work. We enjoy what we do, but we know that hard work awaits us. Every difficulty is an opportunity to learn. The best lesson we can give to our students is to teach them how we learn.

The eighth great idea: we are entering a digital world in which knowledge of digital technology is as important as reading and writing. So learning about computers is crucial for the future of our students. But the most important goal is to use them NOW to learn other things." (Walat, 2007b)

The use of algorithms in teaching undoubtedly fits in with the constructionist educational strategy. The algorithm itself is inextricably linked with technology, as well as stimulates creative thinking and the need to search for optimal solutions.

3.2 Computational thinking concept in teaching of algorithmic thinking

Algorithms and programming in teaching are also the implementation of the concept of teaching algorithmic thinking, computational thinking.

In 2006, Jeannette Wing, Avanessians Director of Data Sciences Institute at Columbia University and Professor of Computer Science, began to promote his idea of computational thinking (Sysło, 2012):

"Computer-related thinking, accompanying the processes of solving problems using computers, can be characterized by the following features:

- the problem is formulated in a form that allows and enables using IT methods and a computer or other devices for automated information processing;
- the problem lies in the logical organization of data and drawing conclusions from them;
- abstract the data representation, for example in the form of a model or simulation;

- the solution to the problem is in the form of a sequence of steps, so it can be obtained as a result of an algorithmic approach;
- design, analysis and computer implementation (implementation) of the solution to the problem lead to obtaining the most effective solution and the best use of the computer's capabilities and resources;
- the experience gained in solving one problem can be used to solve other problems, related as well as other areas."

Computational thinking, as understood by the Center for Citizenship Education, consists of the following "skills" and "attitudes and habits" (*Computational thinking*):

Skills

1. Formulating problems. Identifying, naming problems, asking the right questions;
2. Data collection. Determining the reliability of data and the reliability of information sources;
3. Unloading into parts. Organizing data, dividing tasks into smaller ones;
4. Pattern recognition. Classification (creation of sets), recognition of similarities, finding significant and irrelevant differences, generalization;
5. Abstraction and creation of models. Removing unnecessary information, simplifying, creating models;
6. Creating algorithms. Setting the next steps and creating rules, sequence, recursion (repeatability of procedures and activities);
7. Detecting and diagnosing errors. Searching, finding and analysing errors;
8. Comprehensible and effective communication. Formulating understandable messages, adapted to the recipient (computer or other people), coding, depicting (symbols and signs).

Evaluation

Recognition of evaluation criteria, determination of priorities, evaluation of prototypes and solutions.

Logical thinking

Drawing conclusions, recognizing logic errors, submitting.

Attitudes and habits

1. Search. Experimenting, free and open search for solutions, playing with solutions;
2. Creativity and ingenuity. Developing and using imagination, coming up with new solutions;
3. Improving. A critical approach to the effects of your work and focusing on their continuous improvement and improvement;
4. Perseverance and patience. Enduring the pursuit of the goal, mastering in anticipation of the effects, awareness of the need to make an effort;

5. Cooperation. Work in a group and in pairs;
6. A healthy distance to technology. Reflecting on the limitations of technology and the critical attitude towards it.

The failures and problems of today's school cause many factors, both resulting from the negative luggage behaviourism that the teacher is burdened with, as well as being the result of changes in the way of thinking of new generations, often caused by the negative impact of technology. The negative impact of technology usually has its source in the teaching and upbringing process itself.

One of the typical problems we encounter in teaching, especially mathematics and other exact science is the student's excessive expectations for the teacher to indicate specific provisions to solve the problem - waiting for a step-by-step guide. Today's students, people belonging to the Z generation, expect immediate information that indicates how to solve the problem, expect quick results, are not prone to mental effort, have a problem with referring to the reality surrounding objective tasks. On the other hand, the teacher, who in most of his work uses a behavioural learning model, boils down to providing information and practicing the discussed content on typical tasks. What often results in the fact that in the case of difficulties in understanding the material, the teacher too quickly takes steps to indicate to the student further steps, ways to solve the problem, "telling" students what to do, how they should act. In addition, teachers prefer to exercise and master small, isolated skills and facts, use tasks in which one or two simple skills should be applied. The nature of the Z generation and the traditional teaching model strengthen the student's habits to such a state of affairs hat when he finds himself in a difficult situation - then the teacher will show the way to the solution. Students are more and more often showing reluctance to mental effort, reluctance to seek solutions, analyse, make attempts to explore the topic and are increasingly inclined to often search for ready-made solutions. This is because, as already mentioned, the exercise of their skills consisted in solving very typical, uniform problems (tasks), so when they do not find adequate, ready-made patterns, they come to the conclusion that tasks cannot be solved.

If, on the other hand, the students are forced to follow the necessity of approaching the solution themselves, they often fall into the extreme of mindless application of any previously known models or models.

The technology gives the possibility of trial and error. Although this method is one of the basic methods of creative thinking, it should be remembered that in the healthy application of this method - after selecting the solution variant there is an in-depth analysis, validation of the method, and reaching constructive conclusions. Today, the student expects and through the new technology he is accustomed to an immediate response to its operation (trial). Computer educational games give the answer immediately, the use of the trial and error method is painless for the student, simple, because he is able to check hundreds of variants of solutions in a relatively short time.

Students' activities are deprived of basic and important elements in the process of building knowledge such as: hypothesis, checking the hypothesis, drawing conclusions (including gaining experience) and setting a new hypothesis that takes into account previous conclusions. This is because new technologies provide the possibility of instant confirmation or denial of a given hypothesis, the student ceases to consider the premises of correctness of the hypothesis, does not analyse failures, mindlessly tests another idea to solve the problem.

3.3 Application of computational thinking in educational practice

In schools, however, it does not really learn to use new technologies to solve problems - new technologies are usually used for quick information retrieval, supporting the didactic process - as a modern medium of information transfer as a tool supporting typical office work. In most cases, the use of new technologies boils down to passively using their functionality.

Thus, in today's education, the ways of teachers' actions resulting from the traditional behavioural approach to the expectations and capabilities of the Z generation clash. Not without reason, the generation of the young is said to be "digital natives", while in the teachers we meet people who are so-called. "Digital immigrants". And even if the teacher is a young person, he was often taught the methods of "digital immigrants".

The opinion of Andrzej Walat (2007b) also seems important. “A lot of students, not only in Poland, are convinced that if they cannot recall the algorithm - a formula for solving a task, they will not come up with it themselves, and further dealing with the problem is just a waste of time. Many also think that school tasks usually "do not make sense", so they do not even try to use reason - You need to use learned methods, thinking can only harm. It is difficult not to ask the question: *Why is this so?* and *How to change it?* Undoubtedly, the formation and consolidation of beliefs, in particular the belief that:

- you can come up with an algorithmic solution to the problem yourself,
- many tasks have many different correct solutions and usually there are many fundamentally different approaches to the problem,
- it is worth first looking for a solution to the problem in your own head, and only then in the book,
- do not give up after the first failures, usually finding a solution to an interesting task requires many attempts.”

With the teacher's appropriate strategy, learning by creating algorithms (through algorithmic thinking) can overcome the above drawbacks, because even the adoption of the trial and error method can be targeted so that already at the stage of making the hypothesis enter the analysis of the problem based on already acquired knowledge. In addition, just checking the hypothesis (analysis of the finished algorithm, checking its

logical and substantive correctness) entails the necessity of re-reading information and knowledge about the subject of the problem. Another analysis of the topic, occurring in the process of testing (testing) the solution is extremely important, because it is an element that strengthens the already acquired knowledge, as well as contributes to the acquisition of further elements of experience related to the topic of the problem. It is also a moment when you can, and sometimes even need to, expand the information acquired so far, because testing the solution may indicate lack of knowledge or incorrect, incomplete understanding of the subject. Table 5 presents the linkage between activities, computational thinking and elements of constructionist ideas.

Table 5. Linkage between activities, computational thinking and elements of constructionist ideas

Teacher / student activities	Elements of computational thinking	Elements of great constructionist ideas
Introduction 1. Clearly presented problem (task) to be solved, paying special attention to: a) *input data* - that is, a clear description of the initial situation, the initial data set, b) *output data* - defining the expected final situation, final data set, determining what should be the solution. It is also necessary to define the limits and remind the elements of basic knowledge related to the topic.	1. Formulating the problem. 2. Determination of input data - initial state. 3. Determination of output data - final state, effect. 4. Possible identification of information sources.	The implementation of the problem solution requires the creation of an algorithm and program. The problem formulated in a way that meets the student's needs - play, competition, the content of the problem is in the interests of students, is a challenge for them. To solve the problem it is necessary to use technology so that thinking is no longer a chore, it has become easier and more effective. We use technology to learn - learning by creating an algorithm (program).
Main phase 1. Directing students' actions aimed at solving the task - building an algorithm, sequence of actions solving the problem (task)[1]. 2. Finding/investigating a solution. 3. Creating the solution.	1. Breakdown of the problem into smaller parts, easy and possible to solve. Search for similarities to existing patterns. If the task (problem) is solved by the project method or in student groups, it is also possible to suggest the division of duties (scope of work) into	1. Introducing the idea of *hard fun*[3]. 2. Giving the students of a time[4].

[1] It should be emphasized here that the teacher's work should focus only on directing students' actions in situations where their thinking is far from expected. It should be remembered not to build artificial limitations of the students 'thought process, the teacher should interfere when students' actions - their thinking begins to be inappropriate due to non-use, deviating from the knowledge gained so far. However, it is not always necessary for the teacher to intervene when the reasoning of students is incorrect due to lack of knowledge (the process of solution creation still applies to learners, eg. lack of information), in such cases the information may be supplemented when the pupils themselves will find that their strategy for solving the problem has been stuck because of too little knowledge, possibly a return to the initial state and re-creative work on solving the problem - in cases when the necessary teacher interference would concern information (knowledge) far beyond the scope provided for a given educational stage. You cannot put a hard border here, a teacher who knows his students, their abilities and capabilities decide how to divide the problem into smaller parts, easy and possible to solve. Search for similarities to existing patterns.

	individual persons. 2. Creative investigation into a solution[2]. 3. Creating models and simulations, simplifying.	
Closing phase 1. Presentation of solutions, discussion on their correctness. 2. Testing, checking on different data sets, in different situations. 3. Doing possible correction (improvement) of the algorithm (program).	Searching for the optimal solution, testing, searching, finding and analysing errors.	There is no success without failures. Independent search, creativity and creativity in solving tasks having many different variants of solutions, error helps to better understand the core of the phenomenon, problem, tasks.

Source: own elaboration

The method of interpretation and action presented above gives the opportunity to formulate problems, tasks whose solving requires the use of many different skills, different ideas. It supports the construction of reasoning and building meanings and connections.

The use of algorithms and programming (computational thinking) can be an important and fascinating addition to subject activities. The approach consisting of decomposing the problem (creating algorithms, procedures for sub-problems) is conducive to finding solutions based on the knowledge that we have so far. It allows recognizing patterns, finding similarities and differences, developing the ability to predict a solution. On the other hand, the generalizations we use when creating algorithms (programs) allow to learn and assimilate general principles, assertions.

Algorithmic thinking, understood not as the ability to perform an algorithm, but as an ability to analyse a problem, to analyse a task in order to develop a solution described as a set of steps favours an in-depth understanding of the problem. Creating an algorithm is a way of learning, it helps to know and understand many areas from various fields of mathematics and other science.

The knowledge acquired "by the way" of creating an algorithmic solution (program writing) is usually deeper and more durable. Learning to get involved in other goals - like creating a program - is more effective.

---

[3] If the problem is formulated with regard to satisfying the student's needs, it will not be a boring problem. Usually, pupils do not manage at school, because something is too difficult, students cannot cope at school, because the way the content is transferred is boring. If we are in a state of determination, fascinating with the problem, with interest in the problem, because it is in the area of satisfying our needs, then working on its solution does not cause us any problems. We are ready to give more. Effort during exciting, addictive fun is not good, it's fun. The road to the destination begins to be more exciting than the effect of achieving it.

[4] Let them come to solutions at their own pace. Success in solving a task requires time, often many trials, exploring the problem, sought after, drilling from various directions.

[2] It is the most important moment, primarily because during the process of creating a solution (algorithm), students effectively acquire knowledge (combine information with context and experience).

In addition, Kazimierz Mikulski (2017) wrote: “Today's language of creativity is programming that gives children and young people the opportunity to creatively approach IT goods and develop positive traits of activities, helping them in their future careers. Programming teaches young people logical thinking, problem solving and above all, working in a group.”

Tomasz Kopczyński (2016) wrote also: "The role of the modern teacher is to prepare students to acquire key competences and master them to the extent that they can cope in the real social world in the future. This programming helps to learn and understand many interesting and important areas from various fields of knowledge that the student could learn in the process of learning at school.

The use of new technologies at school is not an end in itself. Digitization aims to support the process of learning and teaching, in which the student is not only a participant, but also a creator and an important link. It enables individualisation of the education process and preparation for independent use of educational resources, and above all in a longer perspective to prepare for adult life, in which also students will have to constantly develop, acquire new competences and skills, continually education to perform professionally chosen occupation or even get a new one. Therefore, what we really want to teach children is not the programming itself, but the skills that it requires, among others logical thinking, solving tasks. There is no the need to educate all future IT specialists or programmers, but to develop the habits of thought that facilitate functioning in the modern world."

Several areas of algorithmic and programming can be distinguished:

- as one of the elements from the assumption supporting the acquisition of new knowledge, in cases when new knowledge and skills are a priori algorithmic,
- as an activity supporting the introduction of new content, new concepts, where computational thinking and algorithmic is one of the tools,
- as an activity during which students "by the way" practice the application of acquired knowledge in various problems, including atypical ones, and acquire new ones or broaden the already acquired knowledge and skills.

It is also worth paying attention to a slightly different division of the application of programming:

- a tool for solving computational tasks
- tool for solving control tasks (simulations, creating a model of reality)

The practical introduction of computational thinking, algorithms and programming often raises many fears and doubts, mainly because teachers of non-IT subjects are not always able to see such a possibility in the currently processed material. Often you have to get through many problems yourself in order to learn to recognize areas of your own subject in which algorithmic and programming can be used.

It should also be borne in mind that there should be close cooperation between a teacher of non-IT subject and an IT teacher, because many problems will also contain purely IT sub-problems. On the other hand, the IT teacher should also be aware that the use of algorithms and programming on other objects shows students that the digital world is not detached from other disciplines, that IT really surrounds us everywhere.

The first algorithmic problems with which students will meet on non-IT subjects will be difficult for them at the beginning. Realization of constructionism assumptions will not occur automatically, in the first phase the teacher should choose very easy problems, target the students more, make them gain some experience, learn some patterns of "approach to the subject".

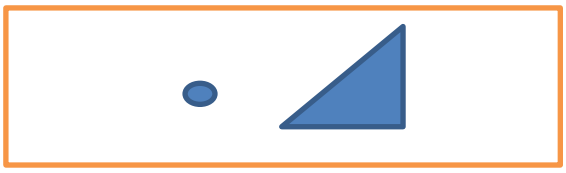
3.4 Practical exercises of using of algorithmic and programming

In a few examples, let's look at how you can use algorithmic and programming in teaching of non-IT subjects.

EXAMPLE 1.[5]

Write a program that meets the following assumption:

There is a point in the center of the screen. In the right upper part of the screen, the program randomly draws a geometrical figure (triangle, quadrilateral). The player's task is to draw a figure in the lower left part of the screen so that after turning 180 degrees it will cover as accurately as possible with the figure drawn by the computer.



After decomposing the problem, we get a set of sub-problems related to the graphic part. These sub-problems will not be more difficult (eg. in Scratch they are quite simple to solve).

From the point of view of non-formal learning, we will be interested in the mathematical sub-problem - finding the coordinates of the vertices of the figure after rotation on a semi-solid angle. This problem can be used by the mathematician to introduce the concept of axial symmetry and central symmetry (turning over a semi-double angle is the middle symmetry).

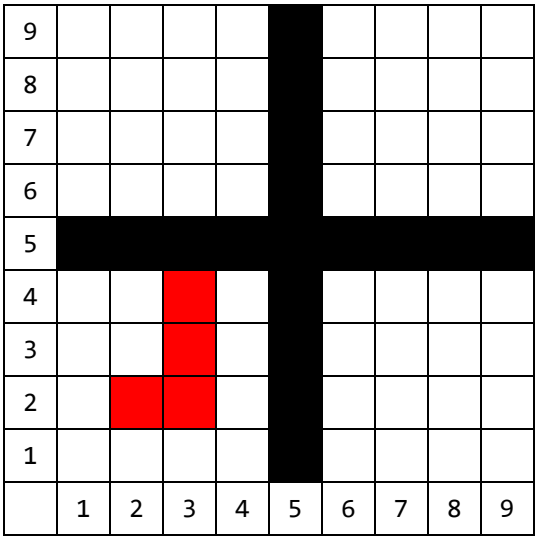
[5] All examples were prepared on the base on: Kolarz & Tluczykont, 2018.

The mathematical sub-problem can be further decomposed until the set of activities is understandable and easily possible for the student to perform. We assume that the student does not yet know the concepts of symmetry.

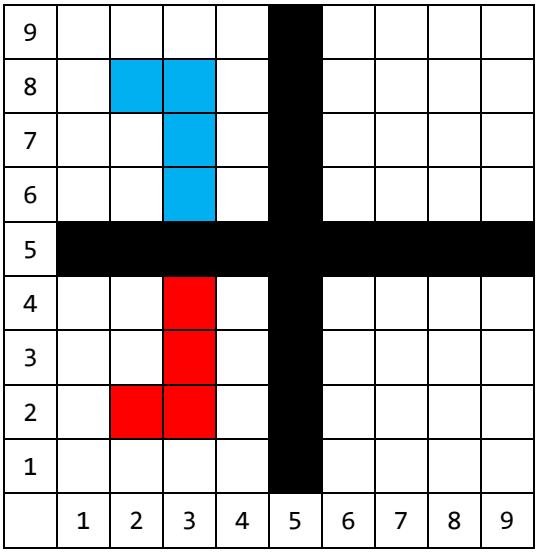
So let's consider the following problem - how to create an image that mirrors the original image. In addition, the issue will be simplified by "enlarging" the plane, so that it can operate on individual pixels, analyzing the arrangement of four pixels. It is also an important moment of our activity with the students because our considerations will be carried out by modeling the reality in a simplified way. Imagine that on the plane we have four elements and visible edges of mirrors aligned at right angles.

The plane consists of individual pixels, whose position can be determined by means of coordinates (traditionally, the horizontal axis ox, vertical oy). The thickness of the mirror coincides with dimension of the pixel. In the picture, thick black lines are the edges of mirrors. The red figure on which we will be experimenting consists of four pixels.

In the first place, let's deal with the mirror image relative to the horizontal line.



Students will easily arrange the image that will be created after mirroring the red figure.



Let us now try to find regularities and to save in an algorithmic way the transformation of the red figure into its mirror image (blue figure). We will use a table for analysis (the first two lines contain the coordinates of the pixels that make up the red figure:

X	Y	X	Y	X	Y
2	2				
3	2				
3	3				
3	4				

After arranging the mirror image of the red figure, the students supplement the coordinates of the blue elements in the table. Then they try to analyze and find regularities, in order to algorithmically save the way of transformation (mirror reflection).

X	Y	X	Y	X	Y
2	2	2	8		
3	2	3	8		
3	3	3	7		
3	4	3	6		

```
1) Select the item.
2) Leave the ox coordinates of the selected element unchanged.
3) Calculate the oy coordinate as follows:
y blue = (y edge – y red) + y edge = 2 × y edge - y red
4) If there is any irrelevant element go to point 1)
```

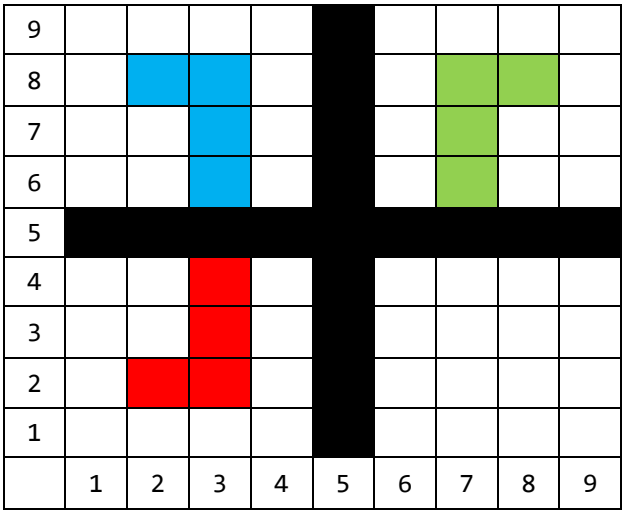
Let's try to experiment with the "mirror", the edge of which runs vertically, we will be particularly interested in the image of the image created in the previous reflection.

We already have a ready algorithm, just swap the x places with y.

```
Select the item.
Leave the oy coordinate of the selected element unchanged.
Calculate the ox coordinate as follows:
x green = (x edge - x blue) + x edge = 2 × x edge - x blue
If there is any irrelevant element go to point 1)
```

Now the students should realize the above algorithm first by completing the coordinates in the table, and then, according to the "green" coordinates, arrange the elements on the plane.

X	Y	X	Y	X	Y
2	2	2	8	8	8
3	2	3	8	7	8
3	3	3	7	7	7
3	4	3	6	7	6



Time for analysis and tested solutions.

What is, for the selected red element, the value derived from the calculation of the expression (*y* `edge` - *y* `red`), of course the distance of this element from the edge of the mirror expressed in pixels. Thus, increasing the resolution of the dependencies used will also be true. Is increasing the resolution to infinity, and so really switching to the Cartesian system, in which, you can say, the size of the pixel is infinitesimal (real numbers) also will be possible?

Mirror reflection is, after all, an axial symmetry whose properties were discovered by the students. And the assembly of two axial symmetries with axes perpendicular to each other is nothing more than a central symmetry - a rotation by a semi-solid angle.

The idea behind this exercise is that the pupils themselves experience certain regularities and dependencies, and only then find out that their "discoveries" are already defined.

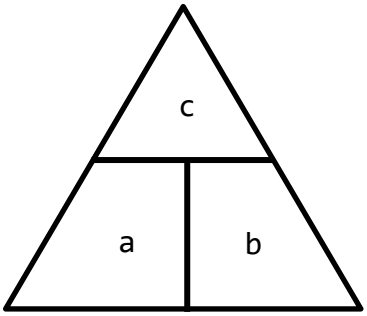
EXAMPLE 2.

The equation is the following:

```
a = b/c
```

you must calculate c.

It is possible for students to present a method, often called a magic triangle, by writing the arguments of the equation as follows:



Covering the searched one, we get a transformed pattern. The student will learn this method very easily, even without the teacher's help he will find it on the Internet. The method can be written in the form of an algorithm. But what will be the effect? The student will get the information (recipe) to transform the pattern, but it has nothing to do with gaining knowledge. Information on how to do it (recipe) will be used only for designs in such and no other form. The student will transform the pattern without thinking. We can handle unusual situations (more complicated pattern).

Let's analyze the transformation of the pattern in the form:

```
a = bc + d
```

Before attempting to solve the problem, students should be reminded of the rules:

1) you can add (subtract) any expression to both sides of the equation,
2) both sides of the equation can be multiplied (divided) by any expression (in the case of division, of course, dividing by zero should be excluded).

Step A.1.: Let's decompose the problem, the equation will be easier:

a = x + d

Our input data is the above form of the pattern. The output is a pattern in the form x=?

Step A.2.: What can you do using rules 1) or 2) to make x appear on the left and "disappear" on the right? It is necessary to x deduct both sides of the equation:

```
a – x = x + d - x
```

which will give us:

```
a – x = d
```

Step A.3.: On the left side of the equation is unnecessary a. What can you do to get rid of the equation on the left? You must use the method from *Step A.1.* (this is an example for looking for patterns, similarities) and subtracting from both sides the equation:

```
a – x – a = d - a
```

which will give us:

```
(– x) = d - a
```

Step A.4.: How to eliminate *minus* from x?

You need to multiply (-x) by (-1), remembering that multiply by (-1) you need both sides of the equation:

```
(-x) × (- 1) = (d - a) × (- 1)
```

which will give us:

```
x = (-d) + a [i.e.: x = d – a]
```

Step A.5.: Let us analyze the course of action (Steps 2 to 4) for the optimization of activities. Let's put together the original form of patterns (*Step A.1.*) With the figure obtained in *Step A.2.*:

```
a = x + d
a - x = d
```

Step A.6.: Conclusion - by moving the expression from one side to the other, we change the sign to the opposite one.

We create an algorithm after optimizing activities.

Step B.1.: Input data - equation in the form:

```
a = x + d
```

The output in the form:

```
x = ?
```

Step B.2.: Move x to the left, changing the sign to the opposite:

```
a - x = d
```

Step B.3.: Move a to the right side by changing the sign to the opposite:

```
(-x) = d - a
```

Step B.4.: Multiply both sides of the equation by (-1):

```
(-x) × (- 1) = (d - a) × (- 1)
```

which will give us:

```
x = a – d
```

Steps from *B.1.* to *B.4.* (algorithm) we can call: `TRANSFORM(x, a, d)`. In this step, we have defined a procedure that you can use from now on.

Let's get back to the right problem. By creating an algorithm for transforming a formula:

```
a = b/c + d
```

we are doing the same in the case of the above-mentioned sub-problem, discussing each step with the students, each case reaching the final form of the algorithm:

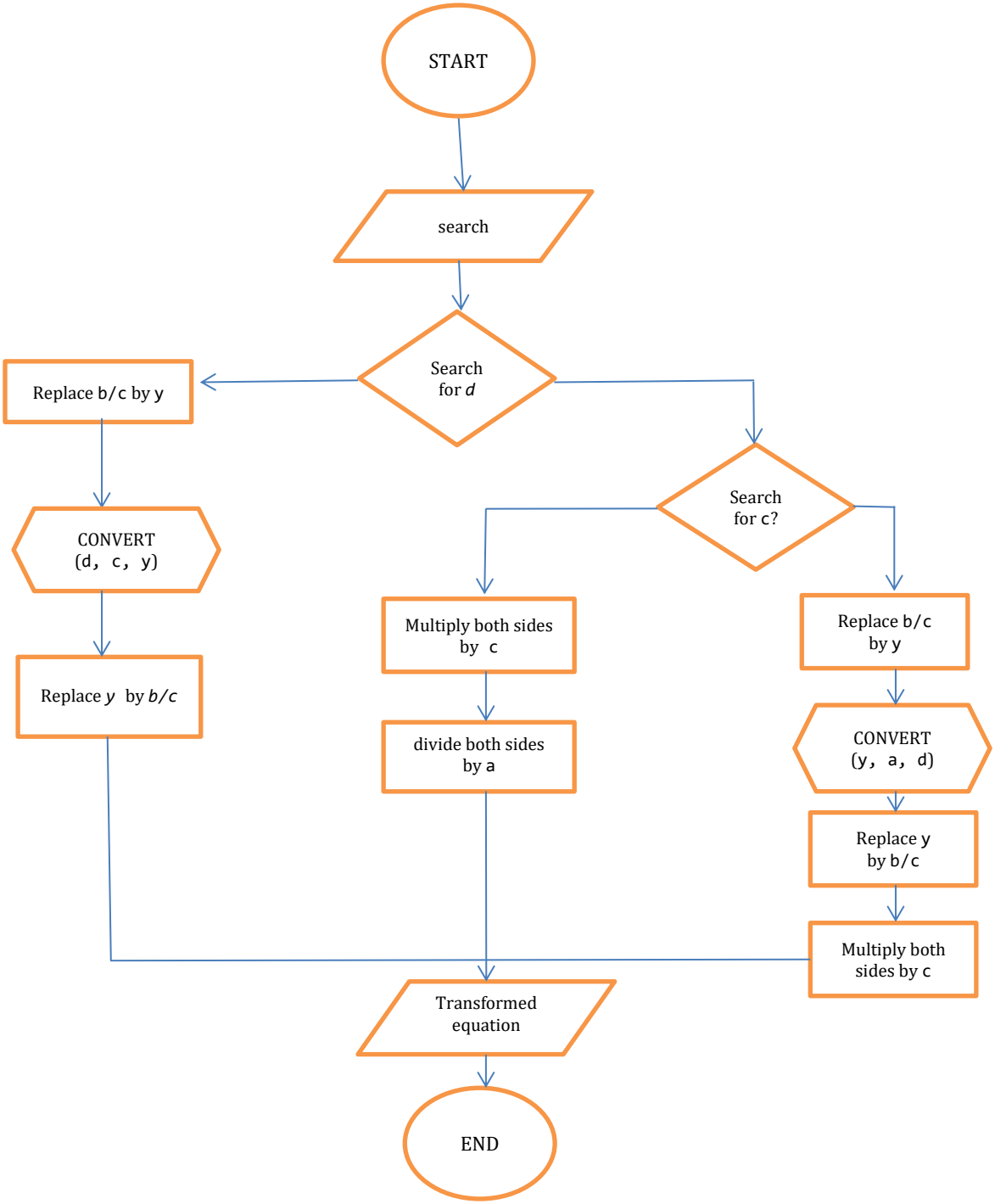


Figure 20 – An algorithm for example 2

Source: own elaboration

Please, note that the above solution is obviously correct, but you can discuss further optimization, further exploring the topic. A careful reader will probably pay attention to the fact that during the transformation of the equation:

```
a = x + d
```

you can first move `d` to the left to get:

```
a – d = x
```

and then write the equation in the form:

```
x = a - d
```

Paying attention to this fact is an opportunity to discuss that since `a – d = x` is `x = a - d`, but also the above can be broken down into.

Step C.1.:

```
(-x) = -(a - d)
```

Step C.2.: We also multiply on both sides by `(-1)` getting:

```
x = a - d
```

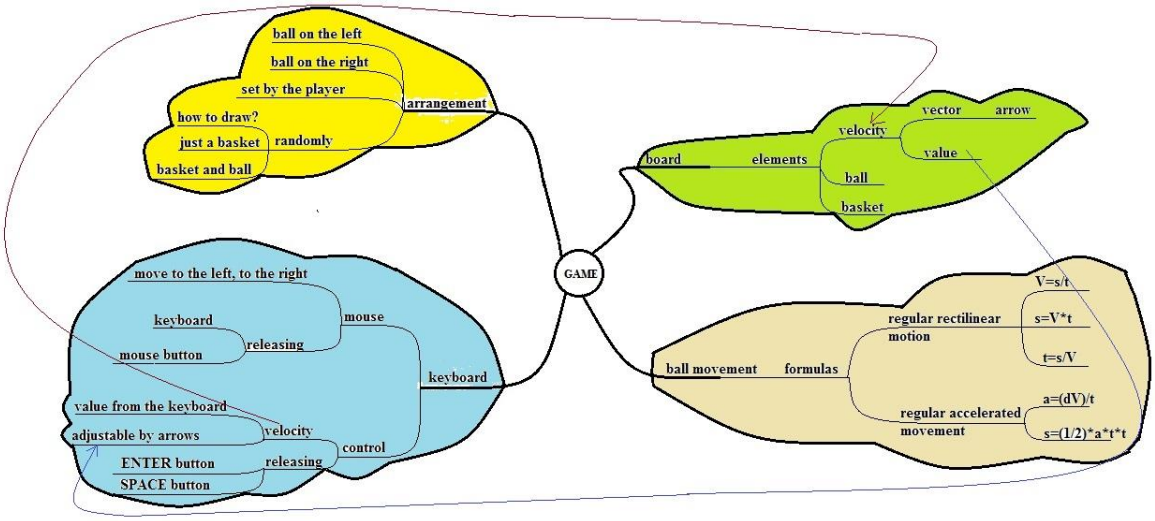
The presented example shows how sometimes a problem, trivial for the teacher, can be very profoundly filtered. The actions aimed at the algorithmic approach to the solution provide the possibility of a thorough discussion, and thus a proper understanding of the topic. Testing the solution, trying to optimize, become an important element that fosters the creation of proper cause-and-effect relationships while building generalizations and simplifications. Patterns and methods are built correctly in the correct way, because acquiring them is associated with the process of understanding.

EXAMPLE 3.

Task for students

Create a game that will consist in the fact that the player adjusting the starting speed of the ball (speed in the horizontal direction) will cause the ball to fall into the basket. The movement of the ball should reflect reality.

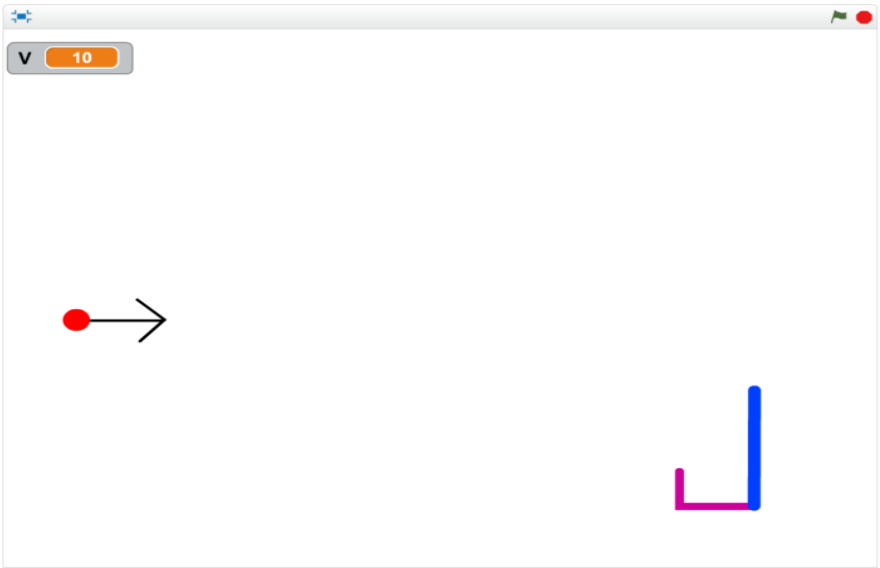
The task is formulated very generally, it is a typical divergent problem. The task can be implemented in groups. Give students time to think about and work out an overall strategy. Suggest that they write different ideas of the game, ideally like in groups, using brainstorming, students can write down ideas using a mind map.



If necessary, the teacher should guide the students indicating what physical laws should be applied.

Let us assume that as a result of the analysis of ideas, the students chose the following option:

On the left side of the screen, there is a ball at random height (oy coordinate). On the other side of the screen, there is a basket at random height. The player with a choice of different speed balls (horizontally) tries to hit the basket. The forces of gravity act on the ball - they must be predicted so that the movement of the ball is a reflection of the movement of the ball in reality. Any frictional resistance will be neglected.



It should be noted what will be the input data and what is the output:

- *input data* - position (oy coordinate) of the ball, and the basket,

- *output data* - the final position of the ball, assuming that the program will end its operation when the ball:
    - falls out of the screen;
    - falls into the basket (according to the design, it touches the right edge of the basket);
    - hits the basket, but it doesn't fall into it (according to the design it touches the left edge of the basket).

Decomposing the problem

Students should consider and define sub-problems. Apart from trivial and perhaps insignificant, from the point of view of the subject being taught, sub-problems, such as graphical development of the ball, basket, arrow representing the velocity vector, control method, should pay special attention to the movement of the ball (trajectory of movement). The key will be independent, possibly with the help of a teacher, acting only as a guide, to conclude that the movement of the ball will consist of horizontal and vertical movement. The teacher can achieve the right effect by asking the students the right questions.

The first question from the teacher

Let's try to ask for the problem if we only consider the movement of the ball on a flat surface. Assuming that the ball at the beginning has some speed set by the player, neglecting the friction resistance, what would the movement be like?
Expected students' response: straight linear motion.

The second question from the teacher

If such a move had to be presented in the form of animation, what would it consist in?
Expected students' response: at the performance of successive phases of movement at short intervals.

The third question from the teacher

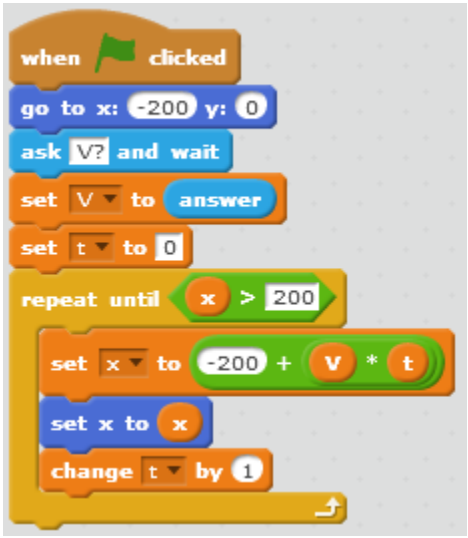
Can we calculate the position of the ball after the end of the time? How to draw next phases of movement on a piece of paper? What do you need to calculate? What could the algorithm look like, every now and again, on the screen object symbolizing the ball, moving in a uniformly rectilinear motion? What would a simple program implementing such an algorithm look like? What equation to use for this?
Expected students' response: the equation of the straight linear motion described by the equations: $s = V \times t$, or on the ox axis: $x = V \times t$. Therefore, the algorithm and the example program in Scratch would be as follows:

```
1) Enter V
2) t = 0
3) x = V × t
4) set the object to position x
5) increase t by 1 point
6) Repeat from 3)
```

The fourth question from the teacher

What to do to make the ball always move from the left side of the screen? What should the ball do to stop at the right side of the screen?
Expected students' response: the algorithm and program will be the following:

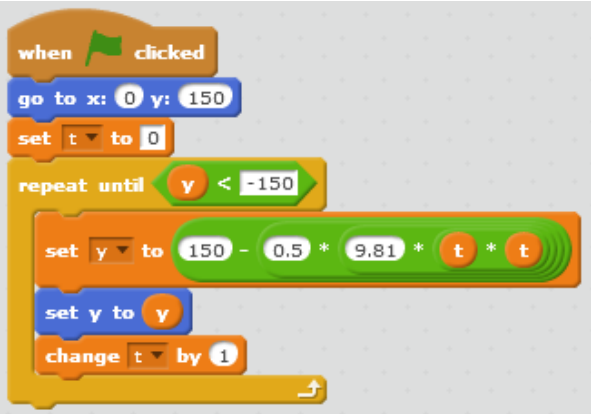
```
1) Set the object to position x = -200, y = 0
2) Enter V
3) t = 0
4) x = -200 + V × t
5) set the object to position x
6) increase t by 1 point
7) Repeat from 4) until x is greater than 200
```



The fifth question from the teacher

Let's put aside the problem of horizontal motion. What causes the ball to move down? What is the type of traffic? Do we know the equations describing this movement? Will the experience gained in considering horizontal traffic be useful? Can vertical movement animations (falling down) be carried out in a similar way, what differences will be possible? How could the algorithm and program look like?
Expected students' response: the algorithm and program will be the following:

```
1) Set the object to position x = 0, y = 150
2) t = 0
3) y = 150 – 0,5 × 9.81 × t²
4) set the object to position y
5) increase t by 1 point
6) Repeat from 3) until y is less than 150
```



The only thing that remains is to experiment with the above algorithms to create a final version, though perhaps not final:

```
1) Set the object to position x = -200, y = 150
2) Enter V
3) t = 0
4) x = -200 + V × t
5) y = 150 – 0.5 × 9.81 × t²
6) set the object to position x, y
7) increase t by 1 point
8) Repeat from 4) as long as y is not less than 150 or x will not be greater
than 200
```

In the above example, you can find important elements of commutative thinking and the idea of constructionism. The problem has been formulated in a very general way, which promotes the development of creative thinking skills in order to seek solutions to differential problems. When solving the problem, we try to determine what the input data is (the status at the beginning) and the output data (state at the end, the goal to achieve). We determine possible limitations. By speaking in a computer language, we define the input specification, output specifications and boundary conditions (restrictions). The problem is divided into sub-problems. When looking for solutions,

we use patterns, we discover patterns (correctness, a pattern for the implementation of point 5 will be points 3 and 4). The solution is a list of steps (algorithm, program). The solution (including also sub-problems' solutions) is tested and improved. We teach by creating using technology as a material. We use the idea of harsh play, of course, hoping that the problem of programming the game will involve more students than considering often torn away from reality, often boring to students typical tasks. In fact, we create simulations of a physical phenomenon, and by the way, we learn how to learn - the implementation of even simple issues requires getting into the topic, analyzing and effectively helps to understand the phenomenon. At first glance, the presented algorithms may not cover many aspects that will occur when creating and testing the program. One can get the impression that the problem is limited to the use of appropriate equations physical. Attention should be paid to several other aspects that will come to light when testing the presented programs. At this stage of creating the algorithm, we did not say anything about units, however, the students will notice in the testing process that "something is wrong", taking a change of time by 1 gives a good visual effect in the case of horizontal movement, while in the case of vertical movement it will turn out that the effect is too short - the time step is too big. This is an excellent opportunity to discuss the units, scale, selection of the time step - what actually means `t = 1`. It is also an excellent opportunity to familiarize students with the subject of computer simulation, pointing out that in the majority of simulations we consider changing values in time, that time "does not run" continuously, but is subject to discretization. In addition, students gain valuable experience related to typically programming skills. It is also worth noting that we are building a certain area of knowledge that did not occur until the program was implemented. In this example, students did not have to know anything before the program was realized that the movement, considered in the coordinate system, can be broken down into components. Knowing the dependencies associated with horizontal and vertical movement, students can experiment by combining both algorithms, thus realizing that motion along any trajectory can be represented as the assembly of motions along the axis of the reference system.

The investigation into the correct solution will not happen without many failures, the seemingly correct algorithm, the correct program, does not give the expected effect, which in turn prompts further analysis of the problem. Keep in mind to give students time, let them experiment, use guides and guiding questions skillfully when they are really stuck (Walat, 2007b). "The student writing simulation programs undoubtedly deepens his knowledge about the nature of matters controlling complex processes. This activity is of great importance for the personalization of knowledge. "In addition, when looking for solutions to interesting mathematical problems, as well as problems in other subjects, you can learn programming much more effectively than in lessons whose only goal is to learn programming".

The use of algorithmic thinking, commutative thinking and programming in the teaching of mathematics and other science subjects gives the opportunity to effectively acquire

knowledge and learn how to learn. Important is the fact that such a way of working with the student gives the opportunity to use the relevant, from the point of view of education, ways to deal with the problem (Kopczyński, 2016):

1. Experimental approach (trial and error method, containing hypotheses, analysis of solutions) - teaches the consequences in proceedings, builds the experience of "causes and effects".
2. Designing and formulating - includes elements of form selection, evaluation of things and methods necessary for solution, it is a process of active implementation of set goals.
3. Correcting (learning from mistakes) - it often takes more time than creating an algorithm or program, but it is an important part of deepening the knowledge. It requires determining the initial state, diagnosing errors (misunderstandings, knowledge gaps), locating the cause to the current situation, the attempt to fix or eliminate errors, checking the correctness after removing the errors.
4. Consequence - it carries an important element of perseverance, which is necessary when searching for and fixing errors of the algorithm (program). It contributes to the systematization of knowledge, creates good habits.
5. Cooperation - develops the ability to work in a group, promotes one of the elements of learning - students learn from themselves, enables efficient implementation of more complex tasks, is an essential element of the work.

Education with the use of algorithms and programming is an effective way of teaching and learning, it is a response to the educational requirements of today's school, inscribing into modern paradigms and educational strategies, and it is an indispensable element of the interdisciplinary preparation of a young person to functioning in the digital world.

3.5 Bibliography

Computational thinking within the meaning of the Center for Citizenship Education. [Online source: http://www.ceo.org.pl/sites/default/files/news-files/elementy_myslenia_komputacyjnego_wedlug_ceo.pdf] (available on: 8th February 2018)

Jelińska A., (2017). *Kompetencje przyszłości. Po co nauczycielom przedmiotów nieinformatycznych programowanie?* [‘*The competences of the future. For what purpose teachers of non-IT subject needs programming?*’] [in:] Kwiatkowska A. B., Sysło M. M. (ed.) *Information Technology in Education*. Toruń: Scientific Publisher of the Mikołaj Kopernik University.

Kolarz W., Tluczykont K., *Kodowanie z matą* [‘Coding with mat’]. Katowice: Computer and Business Association "KISS", in preparation.

Kopczyński T., (2016), *Myślenie komputacyjne jako imperatyw XXI wieku w kontekście nadmiaru łatwej do pozyskania informacji* [‘Computational thinking as the imperative of the 21st century in the context of an excess of easy-to-obtain information’] [in:] Mitasa A. W. (ed.) *System komplementarnego nauczania algorytmiki w aspekcie myślenia komputacyjnego* [‘A system of complementary teaching of algorithms in the aspect of computational thinking’], Goleszów: “Galeria na Gojach” A.B.K. Heczko.

Mikulski K., (2017), *Programowanie elementem kreatywnej pedagogiki,* [‘Programming as an element of creative pedagogy’] [in:] Kwiatkowska A. B., Sysło M. M. (ed.) *Information Technology in Education*. Toruń: Scientific Publisher of the Mikołaj Kopernik University.

Sysło M. M., (2014). Myślenie komputacyjne. Nowe spojrzenie na kompetencje informatyczne [‘Computational thinking. A new look at IT competences’] [in:] Kwiatkowska A. B., Sysło M. M. (ed.) *Information Technology in Education*. Toruń: Scientific Publisher of the Mikołaj Kopernik University.

Walat A., (2007a). *O konstrukcjonizmie i ośmiu zasadach skutecznego uczenia się według Seymoura Paperta* [‘About the constructionism and eight principles of effective learning according to Seymour Papert’] “Meritum” Vol. 4(7).

Walat A., (2007b). *Zarys dydaktyki informatyki* [‘Outline of IT didactics’]. Warsaw: Ośrodek Edukacji Informatycznej i Zastosowań Komputerów [‘Center for IT Education and Computer Applications’].

Ability to using algorithmic and programming is recognized by the European authorities as one of the important, nowadays skill forming part of "digital competence" which is one from eight key competences. Publication entitled: Algorithmic and Programming - Training materials for Teachers. Algorithmic. Programming. Didactics. meets EURYDICE recommendations in this scope. The main aim of the publication is presenting the teachers of an idea of algorithmic and programming along with their practical application in didactics.

[excerpt of Introduction]

ISBN 978-83-951529-0-0