Department of Computer Science
Faculty of Science
Palacký University Olomouc

BACHELOR THESIS

Visualization of Sorting Algorithms

2019

Mykhailo Klunko

Supervisor: Mgr. Tomáš Kühr, Ph.D.

Study field: Computer Science, full-time form

Bibliografické údaje

Autor:	Mykhailo Klunko
Název práce:	Vizualizace třídicích algoritmů
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Mgr. Tomáš Kühr, Ph.D.
Počet stran:	30
Přílohy:	1 CD/DVD
Jazyk práce:	anglický

Bibliograhic info

Author:	Mykhailo Klunko
Title:	Visualization of Sorting Algorithms
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Computer Science, full-time form
Supervisor:	Mgr. Tomáš Kühr, Ph.D.
Page count:	30
Supplements:	1 CD/DVD
Thesis language:	English

Anotace

Cílem práce bylo vytvořit software pro podporu výuky třídících algoritmů pomocí vizualizace průběhu třídění nejznámějšími algoritmy a jejich variantami. Program podporuje názornou vizualizaci vybraných algoritmů na zadaném či vygenerovaném vstupním poli a krokování průběhu výpočtu se souběžným zobrazením pseudokódu použitého algoritmu a aktuálních hodnot použitých proměnných.

Synopsis

The main goal of the thesis was to create a teaching support software with visualization of the most known sorting algorithms and their variations. The application supports a graphic visualization of selected algorithms on randomly generated or manually created array, step-by-step execution possibility, pseudocode and current state of variables.

Klíčová slova: třídící algoritmus; třídění; vizualizace; výukový program

Keywords: sorting algorithm; sorting; visualization; educational software

I would like to acknowledge my sincere gratitude to my supervisor Mgr. Tomáš Kühr, Ph.D. for his patience and guidance throughout my bachelor thesis. Also, I would like to thank my parents for constant motivation and support from their side. Special thanks go to my friends and colleagues.

I hereby declare that I have completed this thesis including its appendices on my own and used solely the sources cited in the text and included in the bibliography list.

date of thesis submission author's signature

Contents

List of Figures

List of Tables

1 Introduction

Nowadays sorting algorithms are widely used in computer software. For example, if you open file explorer on your PC, you may see files sorted in different ways. Searching in sorted data is more efficient than in not sorted ones. Students of computer science start learning different algorithms in the first year of studies and sorting algorithms are among them.

Since I faced the problems of sorting during the course of algorithm design in the first year of my studies, there is an understanding that the visual representation is a vital part of the studying process. During working on the thesis it was very exciting to learn different techniques of sorting algorithms into the depth.

The main goal of the thesis was to create a program which would serve as a tool for understanding how most known sorting algorithms work. There was an attempt to make the best possible user experience. The demonstration software is made in a user-friendly and easy-to-use style. To gain maximal benefit from learning you can try each sorting algorithm on your data.

The text of the thesis describes principles of the most known sorting algorithms which are demonstrated in the computer program. It might be used as a source for learning algorithms by students. Also, the program might be easily used as a demonstration by lecturers and tutors during classes. Besides, there is programmer documentation and user guide to the provided software.

Readers of this text are expected to have some programming experience to know basic data structures such as arrays, lists, trees and understand recursive procedures. Also, knowledge of some simple algorithms and their implementations could be helpful. In order to understand the topic better, knowledge of linear algebra and calculus is involved.

2 About algorithms

2.1 What is an algorithm?

Before moving on to the main part of the text which describes algorithms and the software, we need to assure that there is an understanding of the basics. We shall begin with the algorithm definition.

What is an algorithm? How can we define it? We may say simply that an algorithm is a sequence of steps or instructions that solves some kind of problem. Although it may be a bit imprecise because we have not defined what does problem mean and what does instruction mean.

Problem is a kind of task that we need to solve. We are facing different problems every day: finding the fastest route to work or home, etc. However, not all of them are problems that may fit our algorithm definition. A suitable definition of a problem has some limitations: problem should be specified by its inputs and all inputs have to be mapped to some outputs. Step or instruction is some action that is clear to its executor. In our case, it could be a PC. To solve a problem means to find a solution for each input.

Here is more precise definition by Thomas Cormen: "Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output." [1]

5 features of an algorithm according to D. Knuth[2]:

- *finiteness* – an algorithm should end in a finite number of steps.
- *definiteness* – each step of an algorithm should have precise definition. And it means that for the same inputs we will obtain the same results.
- *input* – an algorithm may have inputs, they are taken from some set of objects.
- *output* – an algorithm may have outputs which should be in some relation with inputs.
- *effectiveness* – we may expect an algorithm to be effective. It means "its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper."[2]

2.2 Algorithm description

To execute or understand an algorithm, we may need somehow to describe it. There are several ways of algorithm representation.

- *Natural language*. An algorithm described in the natural language is clear to everyone. However, it may be imprecise and somehow longer than other methods here.
- *Programming language*. This kind of description is unambiguous. It may be used directly to create a computer program. A programming language may contain a lot of implementation details.
- *Pseudocode* looks similar to a programming language, but it is more general, without deep details. It can be easily rewritten to most of the programming languages, and it is understandable to all programmers without regard to the exact language.
- *Visual representation*. An algorithm may be described by many other methods, including graphical representation like flowcharts (Figure 1).

As an example, we will consider the algorithm for finding factorial representing it in pseudocode (Code 1) and by the flowchart (Figure 1).

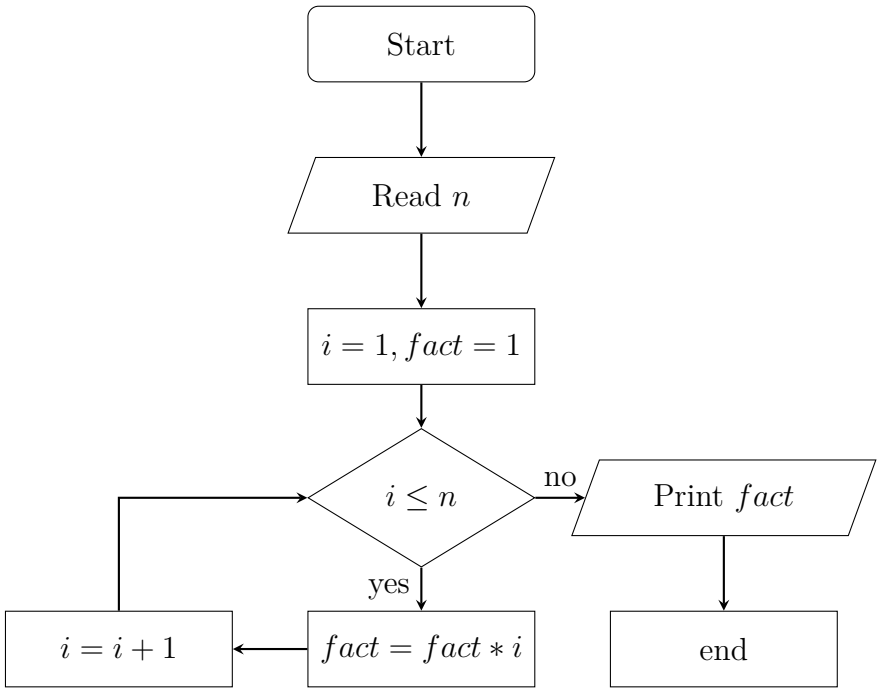


Figure 1: Flowchart example: factorial

Algorithm 1 Factorial algorithm

```
1: procedure FACTORIAL(n)
2:     fact ← 1
3:     for i ← 1 to n do
4:         fact ← fact * i
5:     return fact
```

2.3 Types of algorithms

Nowadays there exist a large number of different algorithms. They are divided into different groups by different criteria. Often one algorithm may belong to several groups. Here are some of them [1].

- *Recursive algorithms* – algorithms that call itself until the limiting condition is reached.
- *Probabilistic algorithms* – algorithms that make some random decisions.
- *Parallel algorithms* – algorithms that divide a task, for example, into threads or between processors, etc.
- *Sequential algorithms* – unlike the parallel algorithms, every step of the task is done in sequence.
- *Divide and conquer* – algorithms that divide a problem into smaller parts until they become indivisible. Then merge solutions in some way.
- *Greedy algorithms* – these algorithms make the best choice in a given situation. They never change their previous choice.
- *Dynamic programming* – algorithms that start solving a problem from the simplest to more complicated parts. They use outcomes from previous solutions.
- *Heuristic algorithms* – algorithms which are used for finding solution among all possible. However, there is no guarantee that a found one will be the best.

2.4 Complexity

Let us consider some problem that requires a solution. Then, suppose that this problem is possible to solve by three or more different algorithms. For the most suitable choice, we compare the algorithms. To compare these algorithms we need to measure the performance of each. Besides, we might need to measure the time for an algorithm to run with some input. Finally, we are moving to the concept of the algorithm complexity [2], [3], [4].

There is a time complexity and a space complexity. In general, complexity gives the amount of time or space needed to run an algorithm. To be more precise: time complexity is a function $f : \mathbb{N} \to \mathbb{N}$ that maps each input to the maximum number of steps for this input needed for an algorithm to complete a task. We will be talking here only about the time complexity and will call it just complexity.

Analyzing algorithms we might want to investigate different sides of the algorithm performance. Usually, it could be a *worst-case analysis* or an *average-case analysis*. The worst-case analysis implies determining the longest running time of all inputs with the same length. In the average-case analysis, we take the average running time of inputs with the same length.

Often complexity of an algorithm is measured on the large inputs. Since complexity is expressed by a polynomial, we might want to have it in a more convenient form. This type of estimation is called an *asymptotic analysis*. Such form takes only the highest order term of the polynomial.

To be more clear, let us have an example. $f(n) = 5n^3 + 10n^2 + 20n + 4$, the highest degree term here is $5n^3$. In some cases we can omit the coefficient 5, now f is asymptotically at most n^3.

For such approximations there exist different notations [2], [3]:

1. *Asymptotic upper bound* (or Big O notation). For the function $f(n)$ with an input size of n, $O(f(n))$ means that the running time or complexity of an algorithm grows as much as the $f(n)$ but may grow more slowly. Sometimes we may write $f(n) = O(g(n))$. To be more precise, it means that there exists some constant $c > 0$ such that $f(n) \leq c * g(n)$ for some large enough n (Figure 2).

2. *Asymptotic lower bound* (or Big Ω[1] notation). If the complexity of some algorithm is in $\Omega(f(n))$, it means that there exists some large enough n such that the function $f(n)$ is a lower bound for the algorithm running time. $f(n) = \Omega(g(n))$ means that there exists some constant $c > 0$ such that $c * g(n) \leq f(n)$ for some large enough n.

3. *Asymptotic tight bound* (or Big Θ[2] notation). This notation combines both lower and upper bounds.

All the functions that we have here we will assume to take only non-negative values. Also, there is a *small o* notation. It slightly differs from the *Big O* notation. "Big-O notation says that one function is asymptotically no more than another. To say that one function is asymptotically less than another we use small-o notation."[3]

[1] Omega
[2] Theta

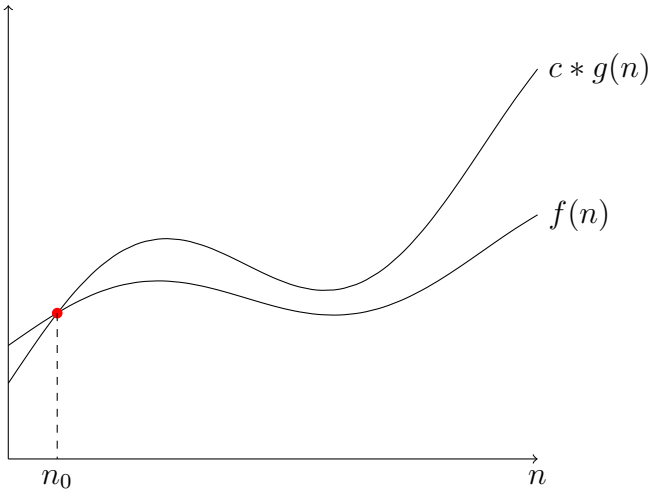


Figure 2: $f(n)$ in $O(g(n))$

Algorithm is considered for practically solvable if its running time is polynomial. It means that the complexity of the algorithm is in $O(n^c)$, c is usually a small constant [1].

Here are some complexities (from the slowest to the fastest growing):

1. $O(1)$ – constant complexity
2. $O(\log(n))$ – logarithmic
3. $O(n)$ – linear
4. $O(n \log(n))$ – linear-logarithmic
5. $O(n^2)$ – quadratic
6. $O(2^n)$ – exponential
7. $O(n!)$ – factorial

2.5 Sorting

As was already said that sorting is used for solving a wide range of problems. It may be used for further searching or, for example, as part of different complex tasks. We were talking about sorting. But what actually sorting is?

Simply said, sorting is a process of rearranging of items, which are possible to compare, in ascending or descending order. In the text we are meaning only ascending order if not stated in a different way.

Ascending order means that items in a sequence are arranged from the smallest to the largest item. On the contrary, descending order means positioning from the largest to the smallest item.

Sorting algorithms are divided into two main types [4]:

1. *Algorithms of internal sorting* – all the data to sort is stored in the internal memory during the sorting process. It is used when the amount of data to sort is known.

2. *Algorithms of external sorting* – all the data to sort is stored outside the internal memory (e.g. on a hard disk). These algorithms usually combine sorting in the internal memory, merging of sorted parts and saving them to the external memory.

In the text we are talking only about the algorithms of internal sorting. Here are five main techniques which are usually used with the algorithms which use these techniques and are included in the software [5]:

1. *Sorting by Insertion* – single items from the sequence are put into the right place of the sorted part. Here belongs Insertion Sort.

2. *Sorting by Exchanging* – swap elements of each pair that are out of order till no more such pairs exist. Here we have Bubble Sort, Cocktail Sort, Quick Sort.

3. *Sorting by Selection* – method that uses repeated selection. Selection Sort and Heap Sort use this technique.

4. *Sorting by Merging* – merging smaller parts with the right order. And Merge Sort uses it.

5. *Sorting by Distribution* – technique which does not use comparisons to sort. It works relying on the knowledge about the set from where data to sort is taken. Data is distributed to some intermediate structures according to values. Radix Sort, Bucket Sort and Counting Sort belong here.

3 Sorting algorithms included

This section describes algorithms included to the software. [1], [4], [5], [6] and [7] served as sources of information for this section.

3.1 Insertion Sort

Insertion Sort algorithm has a simple idea. Assume an array with items to be sorted. We divide the array into two parts: sorted one and unsorted one. At the beginning sorted part consists of the first element (Figure 6). Then, for each item that we have in the unsorted part, we take element and insert it into the right place among the sorted items.

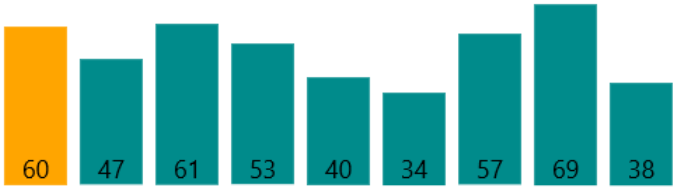


Figure 3: Insertion Sort: sorted and unsorted parts at the beginning

In order to insert element into the right place in the sorted part, we compare selected item from the unsorted part with each item from the sorted part in the direction from right to left. Comparing continues until smaller or equal element is found or no elements to compare left. After each comparison, if current item in the sorted part is greater, we move that current item one position right. Finally, when the right position is found, we insert an item into the sorted part. Complexity of Insertion Sort is $\Theta(n^2)$.

3.2 Selection Sort

Selection Sort algorithm is based on the repeated selection. Here we consider finding minimal key from the unsorted part and swapping it with the first unsorted key. As well as in the Insertion Sort, sorted part grows from the beginning of the sequence.

Assume an array of items to sort. At the beginning of the sorting process unsorted part is represented by the whole array. Then, the first item of the unsorted part is set as the smallest item and is compared with the follow-up elements. When smaller item is found, it is set as a new smallest key. After the end of the array is reached the smallest item is swapped with the first element of the unsorted part and it becomes the sorted part of the array. This step is repeated till the array is sorted. Complexity of this sorting algorithm is $\Theta(n^2)$.

3.3 Bubble Sort

Bubble Sort is based on the idea of exchanging two adjacent elements if they have the wrong order. The algorithm works stepping through all elements from left to right, so the largest elements tend to move or "bubble" to the right (Figure 4). That is why the algorithm is called Bubble Sort.

Now we are going to the details. Let us have an unsorted array. The algorithm does iterations through the unsorted part which is the whole array at the beginning. And with each iteration through the array the range of inspected items is decreased by one till only two elements left. After this two elements are compared and possibly swapped, the array is considered as sorted. Bubble Sort complexity is $\Theta(n^2)$.

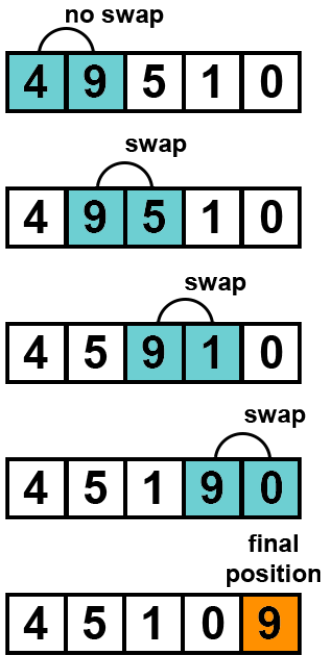


Figure 4: Bubble Sort: one pass through the array

3.4 Cocktail Sort

Cocktail Sort or also known as Bidirectional Sort. This algorithm similarly to the Bubble Sort uses the idea of exchanging unordered adjacent items of array.

Assume array that needs to be sorted in ascending order. Above we described Bubble Sort and this algorithm has a significant problem. It iterates through array only in one direction. This way, smaller items which are closer to the end of array reach its right positions slowly.

The solution is to make Bubble Sort iterate left-to-right and right-to-left. Cocktail Sort uses two cycles inside a big one. Here a and b are the positions of the leftmost and the rightmost elements of the array.

The way the algorithm works:

1. Iterate from a to b, compare adjacent elements and swap if they are not ordered.
2. Iterate from b to $a + 1$ same way as in the step 1
3. Repeat steps 1. and 2. but with a bit different range from $a = a + 1$ to $b = b - 1$

For Cocktail Sort, worst case complexity remains $\Theta(n^2)$.

3.5 Quick Sort

Quick Sort works on the principle "divide and conquer". It recursively applies itself on smaller parts of array until it is not sorted.

Algorithm takes one item at unsorted array or its part, usually it is the leftmost or the rightmost element of array. Then this item, also known as pivot, is moved to its final position in the array that is should occupy. While determining pivot's position, other elements of array are rearranged the way that no bigger elements are on the right and no smaller elements are on the left.

This way, it is enough to apply Quick Sort on each part of array not including pivot until array is not sorted.

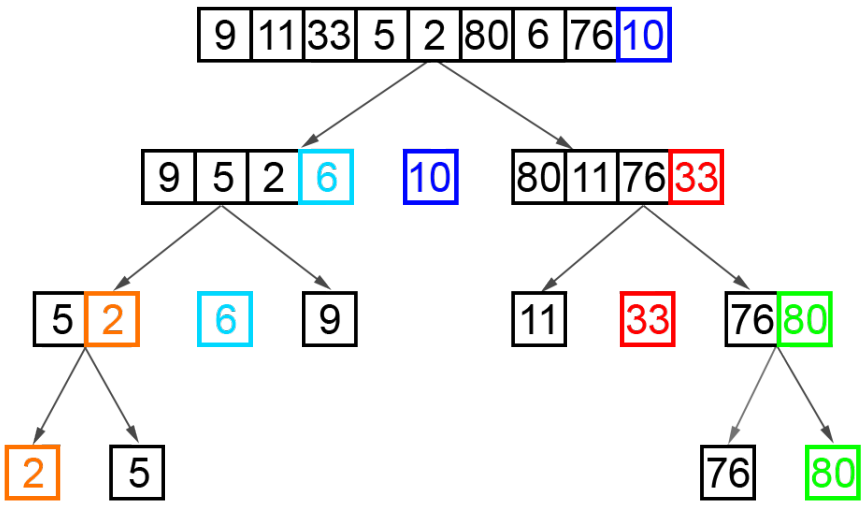


Figure 5: Quick Sort principle

There are several methods of partitioning of array into two parts, here I want to describe one that is demonstrated in the software part of this work.

Firstly, a pivot and index item are selected on the unsorted array or its part. Assume pivot is the rightmost item and index is the leftmost. Next, each item of the array except pivot is compared with the pivot. If a current item is less or equal to the pivot, it is swapped with the index item, next in order item becomes an index. Finally, index and pivot are swapped and this way pivot is on its final position.

Quick Sort is counted as an effective algorithm because its average complexity is $\Theta(n \log n)$. However, when array is maximally unbalanced it may show worst performance. Worst case complexity is $\Theta(n^2)$.

3.6 Merge Sort

Merge Sort as well as Quick Sort is an algorithm of type "divide and conquer". Its logic is simple: divide data into two parts, sort the left part, sort the right part, then "merge" the parts back.

The algorithm works by the recursive application itself on the unsorted parts. In the beginning, it selects the middle item, which becomes the rightmost element of the left part. Then, it recursively sorts both parts. Finally, the algorithm "merges" two sorted parts. Merging procedure itself takes items from each of two sorted parts one by one, compares them and moves the smallest to the output, repeats the previous step. Merge Sort complexity is $\Theta(n \log n)$

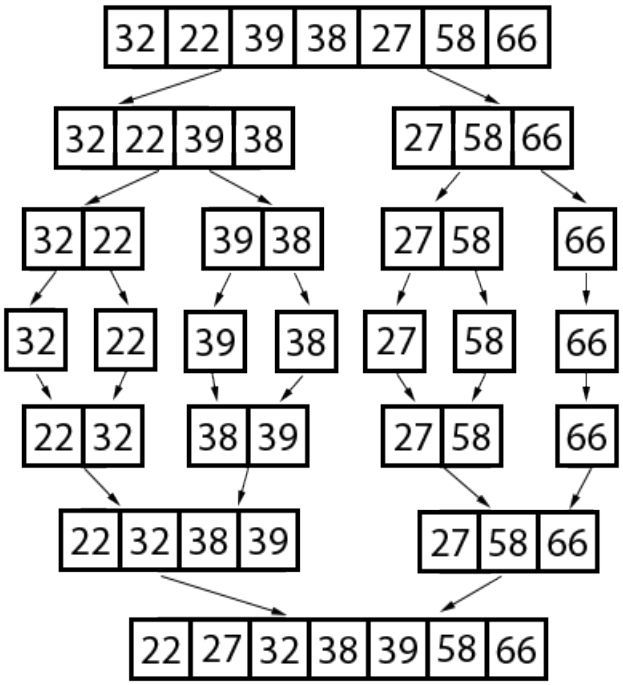


Figure 6: Merge Sort principle

3.7 Heap Sort

Heap Sort is a selection based algorithm and it offers another interesting approach to sorting. In comparison with the Selection Sort it has optimized selection by using binary heap data structure.

Binary heap is a complete binary tree; it means that all levels of tree, except the last one, must be completely filled with nodes. Also, this data structure satisfies the *heap condition*: each node key is greater than or equal to its child keys (this heap type is called *max-heap*).

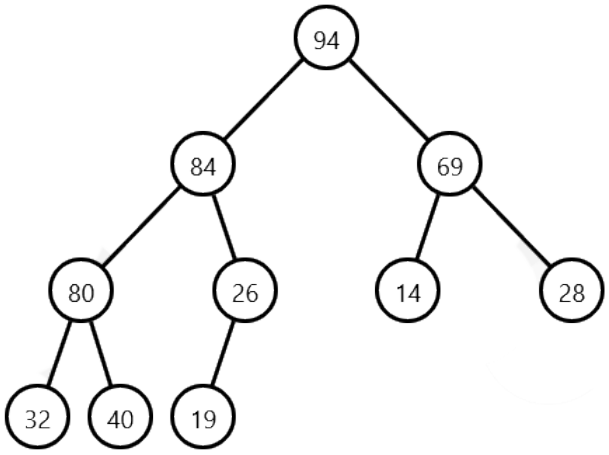


Figure 7: Max-Heap

Binary heap may be implemented by simple array. Item at position zero is a root node, items at position one and two are respectively left and right children of the root. From that representation it is easy to find children of each node (if they exist). Assume a node at position k then its left child is at $2k + 1$ and its right child is at $2k + 2$ (Figure 8).

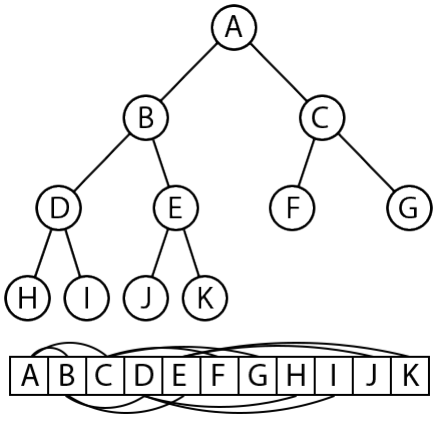


Figure 8: Children positions in heap

Heap Sort itself works as follows:

1. Build max-heap
2. Swap root and the last node, reduce size of heap by one
3. Build max-heap without the nodes on reduced positions
4. Repeat steps 2 and 3 until the range of array is one

To build *max-heap* from current node we need to assure that right and left children comprise max-heaps. This way, in the first step procedure for building max-heap is recursively applied for each node that has at least one child from bottom to top.

After each swap of the root node and the node at last considered position, the last node takes its final place. This way it joins the sorted part of an array. Worst and average case complexity of Heap Sort are both $\Theta(n \log(n))$.

3.8 Counting Sort

Counting Sort is usually used for sorting integer keys in the range from 0 to k. The algorithm is based on counting keys of distinct values. Final positions of keys are calculated from the previous computations. It means that the position of some key x depends on the count of keys that are less than x.

For better understanding let us have an example. Assume we have integers in the range from 0 to k and an empty output array. For this range, we create helping array that will keep counters for each number from the range. Then all counters from the helping array are set to 0. To obtain final positions for items to be sorted we need to make further computations.

Firstly, for each element from the initial array, we increase the respective counter which is determined from the element value. That means the value of the element is its counter position in the helping array. Secondly, we need to sum each counter with previous. Finally, for each item from the initial array, the respective counter is decreased by one, and the value of counter now is the final position, we move the item to the output array.

Worst case complexity for Counting Sort is $\Theta(k + n)$, where k is number of items in range and n is size of array to be sorted.

3.9 Radix Sort

The Radix sorting algorithm is based on the idea of using separate digits to determine final number positions. It works with the help of some stable sorting algorithm, for example Counting Sort, taking digits one by one from least significant to most significant.

For better understanding let us go to the details. Assume the array of integers. Let d be the number of digits of the largest item from the array. If there

are numbers from the array that have less count of digits than the d is, then insufficient digits are counted as digits with zero values.

This way, for each digit from least significant to most significant, we sort the input array according to the current digit (Figure 9). Here we use Counting Sort or another stable algorithm.

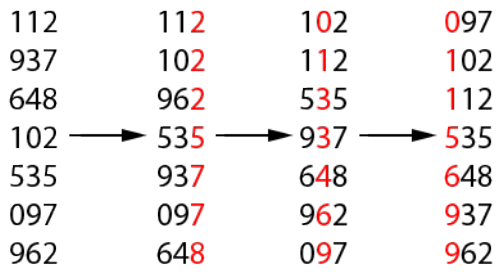


Figure 9: Radix Sort principle

The complexity of Radix Sort depends on three parameters. Assume we have the array of size n with d digits from 0 to k. If the complexity of the used stable algorithm is $\Theta(n + k)$, the complexity of Radix Sort is $\Theta(d(n + k))$.

3.10 Bucket Sort

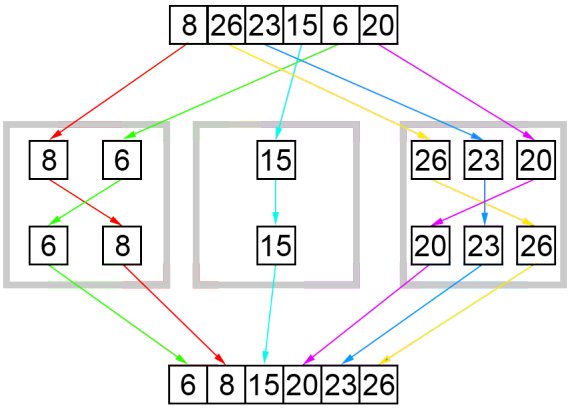


Figure 10: Bucket Sort principle

Bucket Sort, as well as Counting Sort, requires data in a known range with a uniform distribution over this range.

In the beginning, the algorithm creates n intervals of the same size. For each interval, it creates a dynamic structure which will hold items from the input. These dynamic structures are called buckets.

Firstly, Bucket Sort distributes data to buckets. Then, it sorts each non-empty bucket. For this task, we can use, for example, Insertion Sort. Finally, the algorithm moves items from each bucket one by one to the output array.

In case we selected Insertion Sort as the algorithm for sorting buckets, the worst case complexity for Bucket Sort is $\Theta(n^2)$. However, the average case complexity is $\Theta(n)$.

4 Documentation

This section describes the structure of the program and some useful topics from the programmer's side.

4.1 Used technologies

Nowadays there are a lot of programming languages, libraries, frameworks. On the one hand, a software developer has a wide choice. On the other hand, here comes up a new problem to make the right choice which should fit the best to the current task.

The main programming language of the thesis software is Java[3]. To be more precise, Java Platform, Standard Edition 8. Java is a cross-platform, object-oriented language. Its motto is "write once, run anywhere". It means that compiled java application runs on all platforms, that are supported by Java. Java applications are compiled to the kind of bytecode and may be running on the Java virtual machine (JVM)[4] regardless of the platform.

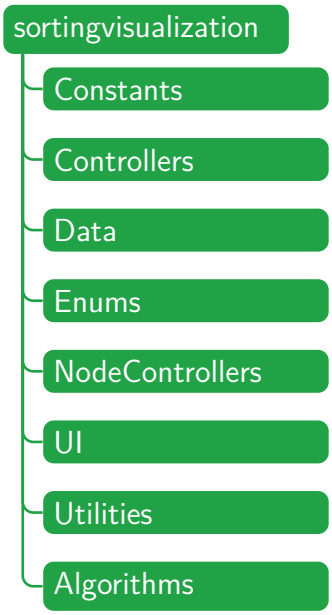
As for the visual side of the application, Java provides a good JavaFX[5] library which is enough for the purpose. Here JavaFX was used both for the user interface and algorithms visualization. Also, JavaFX provides the possibility of using Cascading Style Sheets (CSS). The visualization software uses CSS[6] for styling some elements of the user interface.

The project was created and developed in the NetBeans IDE, although you may use other IDEs that support importing the NetBeans projects. It was developed under the Windows operating system and was tested with different resolutions and on the macOS. A version control system Git[7] in connection with the GitHub was used. All the working steps of the project you may follow on the GitHub website.

[3] Java
[4] What is JVM
[5] JavaFX
[6] About CSS
[7] Learn and download Git

4.2 Structure overview

Program source codes are structured into packages:



Each package contains certain classes, which are grouped by the purpose of use. Next subsections contain some general descriptions of the classes from packages. More detailed descriptions of the class functions are located directly in the source codes.

4.2.1 A root package

The root package contains only one class. It is MainUI.java. This class serves as the main class which starts the application. Although JavaFX provides the possibility of using XML-based language for creating a user interface, here it is not used. The MainUI class defines main user interface elements and does the instantiating of controllers.

4.2.2 Constants, Data and Enum packages

All these three packages work with data and data structures, although a bit differently. Constants package has only an eponymous class. This class contains final static variables that are used as default values in the program. Data package contains classes which serve as definitions for data objects. *BindingData* class instance holds binding data for the buttons from the control panel. *Results* class defines object that is used for transferring results from the input dialog. Enum package contains one definition of the enum class. *Algorithm* enum defines list of algorithms that are visualized in the program. For example, a list with algorithm names in the main window is generated from this enum.

4.2.3 Controllers package

The package has two Java class files: *AnimationController.java* and *ViewController.java*. The first one is responsible for the control of animations run. And the second one controls the graphic part of the application: creation the right graphic representation of array nodes, control of animation creation, pseudo code creation control. Both are used in the main class.

4.2.4 NodeControllers package

NodeControllers package includes *BrickNode.java, ColorInfoManager.java, DynamicNodes.java, FixedNodes.java, Pseudocode.java, Tree.java and VariablesInfo.java*.

BrickNode class defines kind of representation of the key from the array. After instantiating it can be styled in many ways. This is done by the different node managers. We shall talk about them next. Before moving on, we describe node managers generally. These objects are responsive for creation a visual representation from the given data. Also, they may create some additional graphic items, e.g. buckets for the Bucket Sort. Node managers define animations for the certain types of nodes and define their own metric system for certain type of nodes.

The first is *DynamicNodes* class. *DynamicNodes* object manages visual nodes whose height depends on their value (Figure 11a). Then, there is *FixedNodes* class. This node manager class defines a manager that controls nodes of fixed size (Figure 11b). Finally, there is *Tree* node manager. *Tree* represents manager that creates and controls a visual binary heap (Figure 11c) and corresponding visual array.

(a) Dynamic-size nodes (b) Fixed-size nodes (c) Tree nodes

Figure 11: Node types

Pseudocode class, as it is clear from the title, defines a manager that controls the creation and animating of the pseudocode that appears on the information side panel. *VariablesInfo*, in turn, defines another dynamic part of the information side panel, that shows current details during the visualization process

and state of some variables. *ColorInfoManager* represents information about the meaning of single colors of nodes. Usually, a description is shown for the dynamic type of nodes which tend to change their colors.

4.2.5 UI package

UI or user interface package includes different additional classes that define additional parts of the graphic user interface. There are *InfoDialog* which represents dialog that shows short descriptions of individual algorithms. *InputDialog* instance accepts user input data. *AboutDialog* shows information about the application. *Toast* is a help message in Android OS[8] style.

4.2.6 Utilities package

The Utilities package contains different kinds of tools. *ArrayUtils* is a supporting tool while working with an array. *DescriptionReader* is a tool for reading algorithm description from a file. *Scaling* class defines a tool that counts the scaling factor to fit different screens.

4.2.7 Algorithms package

And the last, but not the least significant, package called Algorithms. It contains classes that actually do animating of the algorithms. Each class that creates animations is supposed to extend *Sorting* class and implement the *AbstractAlgorithm* interface. List of class definitions here corresponds to the list of algorithms in the *Algorithm* enum.

4.2.8 Resources

Apart from the source code packages, project has a resource folder inside. This folder contains images that are used in the program: icons, button images. Also, it includes CSS files that are used for the styling of the main window (style.css) and of the input dialog (dialog.css).

In the resource folder exists a subfolder that stores files with descriptions of the algorithms. File names here correspond to the long names from the *Algorithm* enum without spaces. Such name conventions help the reader tool to find the right description.

[8] Android web site

5 User Guide

5.1 System requirements

Since Java is cross-platform, you may use the application within the most popular PC operating systems where Java is supported. Here are given minimal system requirements for several operating systems.
Requirements:

- Operating system: minimum Windows XP SP3 or Mac OS X 10.4.10 or Ubuntu 8.04
- Java: Java SE 8 with minimum update 40 (8u40) or update 51 (8u51) in case of Windows 10 operating system
- Processor: Dual-Core processor, 1.8 GHz
- Memory: 512MB of RAM (1GB recommended)

5.2 Starting application

Since Java applications are not native applications to the most popular operating systems for PC[9], you need to have JDK[10] or JRE[11] installed and configured. In case it is already done, it is enough to run the file of the application with *.jar* extension as simple application.

Also, it is possible to use option "Open with". Here is necessary to select the right application. For example, in the Windows operating system it is usually "Java(TM) Platform SE binary" or *javaw.exe*.

Next opportunity is to run it through the command line. For Windows and macOS it is done the same way. Type the command from below and add the right path before the file name.

```
java -jar Sorting-Visualization.jar
```

Just after running the application, the main window (Figure 12) shows up and the application is ready to use.

5.3 User interface

As it was stated above, the main aim of the application is to be as much user-friendly as possible. This part describes the user interface of the application.

Main window (Figure 12) consists of the visualization panel (1), information panel (2), control panel (3) and upper panel (4).

[9] According to netmarketshare.com in May 2019
[10] Java Development Kit
[11] Java Runtime Environment

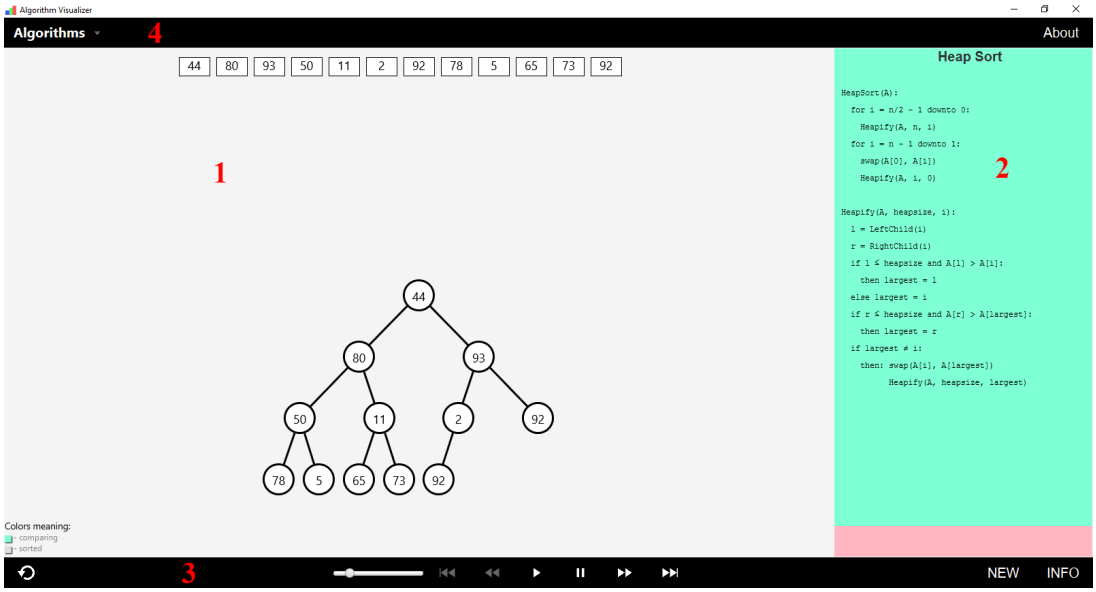


Figure 12: Main window

5.3.1 Visualization panel

Visualization panel (Figure 13) is the main space for the visualization of algorithms. It shows different types of nodes which represent array items. Animated movements of these nodes show individual steps of a sorting algorithm.

For some algorithms, there is a description of the node colors available. It is located in the bottom left corner.

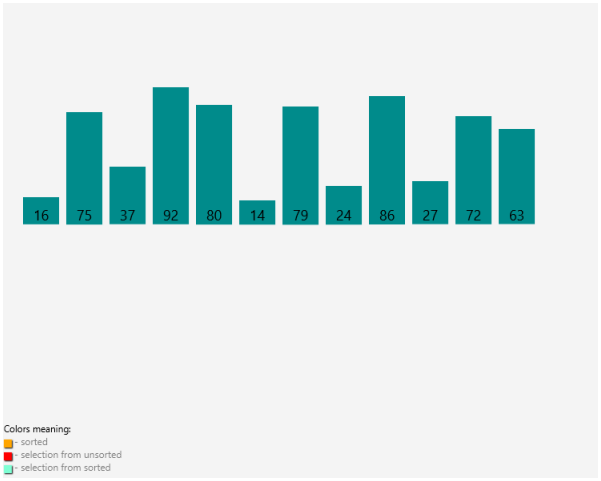


Figure 13: Visualization panel

5.3.2 Control panel

Control panel gives a user the possibility to have full control over the algorithm visualization process. It is possible to run visualization, to pause it, even to go through step by step forth and back. If you do not understand what the exact button means, hover the mouse cursor over some active button and you will see a tip.

Buttons and other interactive elements on the panel (Figure 14, from left to right) are "Reset", "Animation speed slider", "Go to start", "Step back", "Play", "Pause", "Step forth", "Go to end", "New" and "Info".

Figure 14: Control panel

"Reset" action creates a new sorting visualization of the current algorithm with a randomly generated array. "New" action allows a user possibility to enter custom data. "Info" opens a window with a description of the current algorithm.

5.3.3 Information panel

The information panel (Figure 15) is located on the right side of the window. It consists of the header with the name of the current algorithm, the pseudocode of the selected algorithm and description of the current state.

During the run of the visualization, this information is dynamically updated according to the current algorithm. Application synchronously highlights used lines of code and updates the current state line showing the current state of the algorithm and some values of used variables.

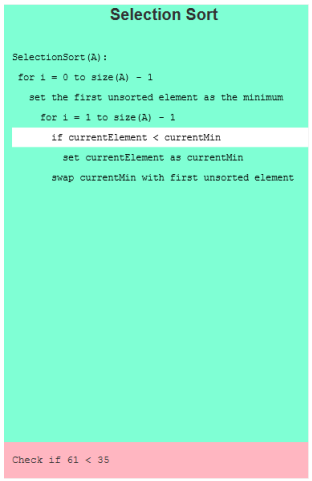


Figure 15: Information panel

When the window is resized and the information panel does not fit, it becomes scrolable. It is possible to hide this panel by clicking on it. In order to restore the panel, use the button with arrow which is located on the place of the side panel.

5.3.4 Upper panel

The upper panel includes selector of the algorithms. To see the list of available algorithms click on the "Algorithms" (Figure 16).

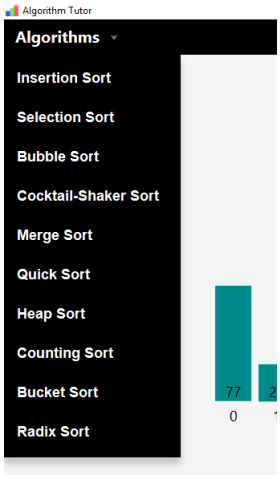


Figure 16: Algorithm selector

5.4 Interaction

This section describes possible interactions between user and the application.

5.4.1 Starting new visualization

To start a new visualization, there are some options. You may either start it by selecting an algorithm from the algorithm selector on the upper panel or click "Reset" to initialize new visualization of the current algorithm with random data. Finally, you can initialize new visualization with custom data.

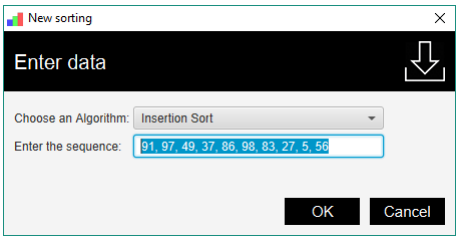


Figure 17: Window "New sorting"

To initialize new visualization with custom data, click the "NEW" button on the control panel. After it, there appears a window (Figure 17). To select an algorithm, choose one from the drop-down list. The input field, which is a line lower, allows a user to enter a sequence of integers. By default, it is filled with randomly generated numbers. For separation of numbers use space or comma. The input field accepts numbers from range 0 - 99, except a pair of algorithms. Counting Sort has range 0 - 9 and Radix Sort accepts numbers from range 0 - 9999.

5.4.2 Visualization control

After setting up a new visualization, it is easy to control sorting visualization. To run the visualization click on the "Play" button or press **R** key. "Play" button may not be available in case visualization is already running or is finished. Also, "Play" button resumes sorting process after it was paused. The process is paused by pressing "Pause" button.

Stepping through the sorting process is performed by two buttons. Step back action is done by clicking the "Step Back" button, which is to the left of the "Play" button. Step forth is made by clicking the "Step Forth" button, which is to the right of the "Pause" button. Both actions are not possible during sorting is running.

In addition, there are buttons that allow go to the beginning and to the end. "Go to start" returns sorting in the initial state. "Go to end" action is intended to fast-forward visualization to the "sorted" state.

Finally, the application allows changing the speed of visualization. By default, there is an optimal speed of animation. To slow down animations, move the knob to the left. To make the visualization run faster, slide the knob to the right.

5.4.3 Information about the application

To display a window with information about the application (Figure 18) click the "About" button on the upper panel or press **F1** key.

Figure 18: Window "About"

5.4.4 Opening description

There is a short description for each algorithm. To show information click "Info" button on the control panel or press **F2**. Then a window with the short description will show up (Figure 19). To close the window use standard close button or press **Esc**.

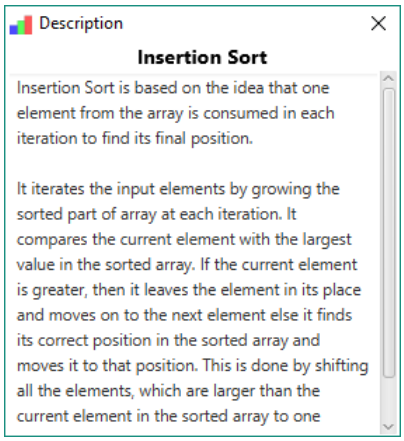


Figure 19: Window "Description"

5.4.5 Escaping application

Close application by clicking exit button. Its location depends on the operating system. Also, you may use the **Esc** button.

5.5 Keyboard shortcuts

Keyboard shortcuts for Windows and macOS are in the table 1.

Table 1: Shortcuts

Action	**Windows**	**macOS**
Run/pause visualization	R	R
Step forth	Ctrl + D	Cmd + D
Step back	Ctrl + A	Cmd + A
Show description	F2	F2
New random sorting	F5	F5
New custom sorting	F4	F4
About	F1	F1
Close application	Esc	Esc
Close "About" window	Esc	Esc
Close description	Esc	Esc

Závěr

Pro bakalářskou práci byl vytvořen software pro podporu výuky třídících algoritmů pomocí vizualizace průběhu třídění nejznámějšími algoritmy. Aplikace dovoluje krokování vypočtu dopředu a dozadu pro každý reprezentovaný algoritmus. Je možné spustit vizualizaci na zadaném či vygenerovaném vstupním poli. Při běhu vizualizace se zobrazují pseudokód použitého algoritmu a aktuální hodnoty některých proměnných.

Snažil jsem se vytvořit kvalitní software ze snadno ovladatelným uživatelským rozhraním, které by bylo použitelné pro přednášející, cvičící a studenty. Možným vylepšením aplikace by mohlo být rozšíření o další algoritmy.

První část textu bakalářské práce je teoretická. Zaměřuje se na obecný popis algoritmů, jejích vlastnosti a popis třídících algoritmů použitých v aplikaci. Druhá část popisuje samostatnou aplikaci.

Conclusions

As the main goal of this bachelor thesis, there was created the teaching support application which visualizes the most known sorting algorithms. The application allows stepping forward and backward through each represented algorithm. User may run sorting on a random or custom array. During the demonstration run, the application visualizes pseudocode and current information about some variables.

I tried to create high-quality software with a user-friendly and easy-to-use interface, which could be used by lecturers, tutors, and students. Possible next improvement of the applications is extension it by other algorithms.

The first part of the thesis text is more theoretical. It tells about algorithms in general and the algorithms represented in the application. The second part is focused on the application itself.

A Contents of enclosed CD/DVD

On enclosed CD/DVD you will find:

`bin/`
: Contains the entire program.

`doc/`
: This directory contains the text of the thesis in PDF created according to the KI RřF style for theses. Includes all the attachments and files needed for generating the text in PDF.

`src/`
: Complete source codes of the program "Sorting Visualizer".

Acronyms

CSS A style sheet language.

JVM A virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

References

[1] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, D. L.; STEIN, C. *Introduction to algorithms*. Second Edition. 2001. ISBN 0-262-03293-7.

[2] KNUTH, D. *The Art of Computer Programming: Fundamental Algorithms*. Third Edition. 2004. ISBN 0-201-89683-4.

[3] SIPSER, M. *Introduction to the Theory of Computation*. Boston, MA: PWS Publishing Company, 1997. ISBN 0-534-94728-X.

[4] BĚLOHLÁVEK, R. *Algoritmická matematika 1: část 2*. Available also from: ⟨http://belohlavek.inf.upol.cz/vyuka/algoritmicka-matematika-1-2.pdf⟩.

[5] KNUTH, D. *The Art of Computer Programming: Sorting and Searching*. Second Edition. 2004. ISBN 0-201-89685-0.

[6] SEDGEWIK, R. *Algorithms in C: Fundamentals, data structures, sorting, searching*. Third Edition. 2007. ISBN 0-201-31452-5.

[7] *GeeksforGeeks*. Available from: ⟨https://www.geeksforgeeks.org/⟩.

[8] BĚLOHLÁVEK, R. *Algoritmická matematika 1: část 1*. Available also from: ⟨http://belohlavek.inf.upol.cz/vyuka/algoritmicka-matematika-1-1.pdf⟩.

[9] *Stackoverflow*. Available from: ⟨https://stackoverflow.com/⟩.

[10] *Java documentation*. Available from: ⟨https://docs.oracle.com/javase/8/⟩.