CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Nuclear Sciences and Physical Engineering

Educational web application for Basics of Algorithmization

Výuková webová aplikace k předmětu Základy algoritmizace

Bachelor's Degree Project

Author: **Karolína Žatečková**

Supervisor: **Doc. Ing. Miroslav Virius, CSc.**

Language advisor: **Mgr. Hana Čápová**

Academic year: 2023/2024

České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská

Katedra: matematiky Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Karolína Žatečková
Studijní program:	Aplikovaná informatika
Název práce (česky):	Výuková webová aplikace k předmětu Základy algoritmizace
Název práce (anglicky):	Educational web application for Basics of Algorithmization

Pokyny pro vypracování:

1) Seznamte se s problematikou webových aplikací.
2) Sestavte požadavky na webovou aplikaci demonstrující základní metody třídění (řazení). Aplikace by měla umožňovat vizualizaci řadicích algoritmů a porovnání jejich rychlosti pro různě velké datové sady.
3) Požadavky analyzujte a na základě výsledků analýzy navrhněte požadovanou webovou aplikaci.
4) Navrženou aplikaci implementujte a otestujte.
5) K navržené aplikaci sestavte webovou uživatelskou příručku.

Doporučená literatura:

1) M. Virius. Základy algoritmizace. Praha: ČVUT 2008. ISBN 978-80-01-04003-4.

2) Tomáš Valla. Průvodce labyrintem algoritmů. Praha: CZ.NIC 2022. ISBN 978-80-88168-63-8.

3) M. Virius: Programování v C# od základů k profesionálnímu použití. Praha: Grada Publishing 2020. ISBN 978-80-271-1216-6 (tisk), 978-80-271-4004-6 (elektronická publikace), ISBN 978-80-271-4003-9 (pdf).

Jméno a pracoviště vedoucího bakalářské práce:

Doc. Ing. Miroslav Virius, CSc.
Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze, Trojanova 339/13, 120 00 Praha 2,

Jméno a pracoviště konzultanta:

Datum zadání bakalářské práce: 31.10.2022

Datum odevzdání bakalářské práce: 2.8.2023

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 31.10.2022

[illegible]

....
garant oboru

[illegible]

....
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-3-

[illegible]

....
děkan

Acknowledgment:
I would like to thank Doc. Ing. Miroslav Virius, CSc. for his expert guidance and express my gratitude to Mgr. Hana Čápová for her language assistance.

Author's declaration:
I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, January 8, 2024 Karolína Žatečková

Název práce:

Výuková webová aplikace k předmětu Základy algoritmizace

Autor: Karolína Žatečková

Obor: Aplikovaná informatika

Druh práce: Bakalářská práce

Vedoucí práce: Doc. Ing. Miroslav Virius, CSc., Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze, Trojanova 339/13, 120 00 Praha 2

Abstrakt: Řadicí algoritmy jsou důležitou a detailně probádanou součástí informatiky a proto tvoří značnou část výukových materiálů předmětů informatiky. Tato bakalářská práce popisuje návrh a sestavení výukové webové aplikace umožňující vizualizaci a časové porovnání vybraných řadicích algoritmů. Cílem práce tak bylo zpřístupnit informace a usnadnit jejich výuku prostřednictvím uživatelsky přívětivé webové aplikace. Implementaci webové aplikace předcházela detailní analýzu již existujících vizualizačních aplikací, která vedla k návrhu sjednocující klíčové aspekty a funkce pro efektivní výuku. Výsledná webová aplikace je specificky přizpůsobena obsahu předmětu *Základy algoritmizace*, kterou je možno použít jako studijní nástroj v oblasti informatiky.

Klíčová slova: D3.js JavaScript knihovna, JavaScript, řadicí algoritmy, vizualizace řadicích algoritmů, vzdělávací aplikace, webová aplikace

Title:

Educational web application for Basics of Algorithmization

Author: Karolína Žatečková

Abstract: Sorting algorithms are an essential and well-studied part of computer science and, therefore, form a significant part of the teaching materials of information technologies courses. This bachelor's degree project describes the design and assembly of an educative web application that enables the visualisation and time comparison of selected sorting algorithms. The aim of the work was to make the information available and facilitate its teaching through a user-friendly web application. The implementation of the web application was preceded by a detailed analysis of already existing visualisation applications, which led to the design of an application that unifies key aspects and functions for effective teaching. The resulting web application is specifically adapted to the content of the *Basics of Algorithmization* course, which can be used as a study tool in the field of computer science.

Key words: D3.js JavaScript library, JavaScript, sorting algorithms, visualisation of sorting algorithms, educational application, web application

Contents

Introduction

Sorting algorithms are a significant and well-studied part of computer science, which is why they are included in the curriculum of most study programs focused on information technology. Therefore, it is crucial that students can easily understand them. Visualisation applications can, therefore, serve such a purpose. Despite the fact that there are several web applications on the market dealing with this issue, e.g. [5, 25], not all of them have sufficient educational potential. These web applications are often not a sufficient source of information for students. For example, they tend to miss pseudocode or step descriptions, which can be a very useful tool for understanding sorting algorithms.

The aim of this Bachelor's Degree Project is to create a web application whose content will be adjusted to the *Basics of Algorithmization* course. It should visualise the basic sorting algorithms discussed in this course and allow their speed comparison. It will eliminate above mentioned problems, will be user-friendly and interactive, and should be a sufficient source of information for understanding the selected sorting algorithms. The application should provide the students with the opportunity to experiment with these algorithms for better understanding of their efficiency and logic.

This Bachelor's Degree Project consists of six chapters. Chapters 1 and 2 deal with the theory of algorithms in general and the theory of selected sorting algorithms. Their time and memory requirements are also mentioned here, as this information is stated in the application and is also reflected when comparing the algorithm speed. Chapter 3 is devoted to a comprehensive analysis of existing visualisation web applications and the comparison of their strengths and weaknesses. Based on this analysis, a clear list of requirements is subsequently made. Chapter 4 describes the technologies used. Chapter 5 then goes into details of the design and implementation of the application, as well as the description of the logic that runs the application. Chapter 6 describes and summarizes user testing, including subsequent changes made.

Chapter 1

Theory of algorithms

This chapter is dedicated to refreshing the basic concepts related to the theory of algorithms. Specifically, it covers the definition of the algorithm and the time and space complexity with respect to the sorting algorithms, as they are the main subjects of this work. This chapter is based on [11, 13].

1.1 Definition of algorithm

Throughout the history, the definition of an algorithm has changed its meaning several times. At present, an algorithm can be understood as a procedure or a recipe for solving certain problems. Within the sequence of steps, the algorithm transforms an input into an output. However, this procedure has to satisfy several criteria to be called an algorithm:

1. *Finiteness* – The number of steps of an algorithm is always finite.
2. *Definiteness* – An algorithm itself is firmly defined, and so are its steps.
3. *Input* – An algorithm is externally supplied by zero or more quantities.
4. *Output* – An algorithm produces at least one quantity.
5. *Effectiveness* – Each step of an algorithm is expected to be effective. Steps are elementary and can be performed precisely and in a finite time.

1.2 Time complexity

Time complexity is the amount of time required to complete an algorithm. It is necessary to take it into account while working with large data sets, especially with regard to sorting algorithms. There are three types of time complexity:

1. *Average case* – Average running time required to complete an algorithm.
2. *Worst case* – Maximum amount of time required to complete an algorithm.
3. *Best case* – Minimum amount of time required to complete an algorithm.

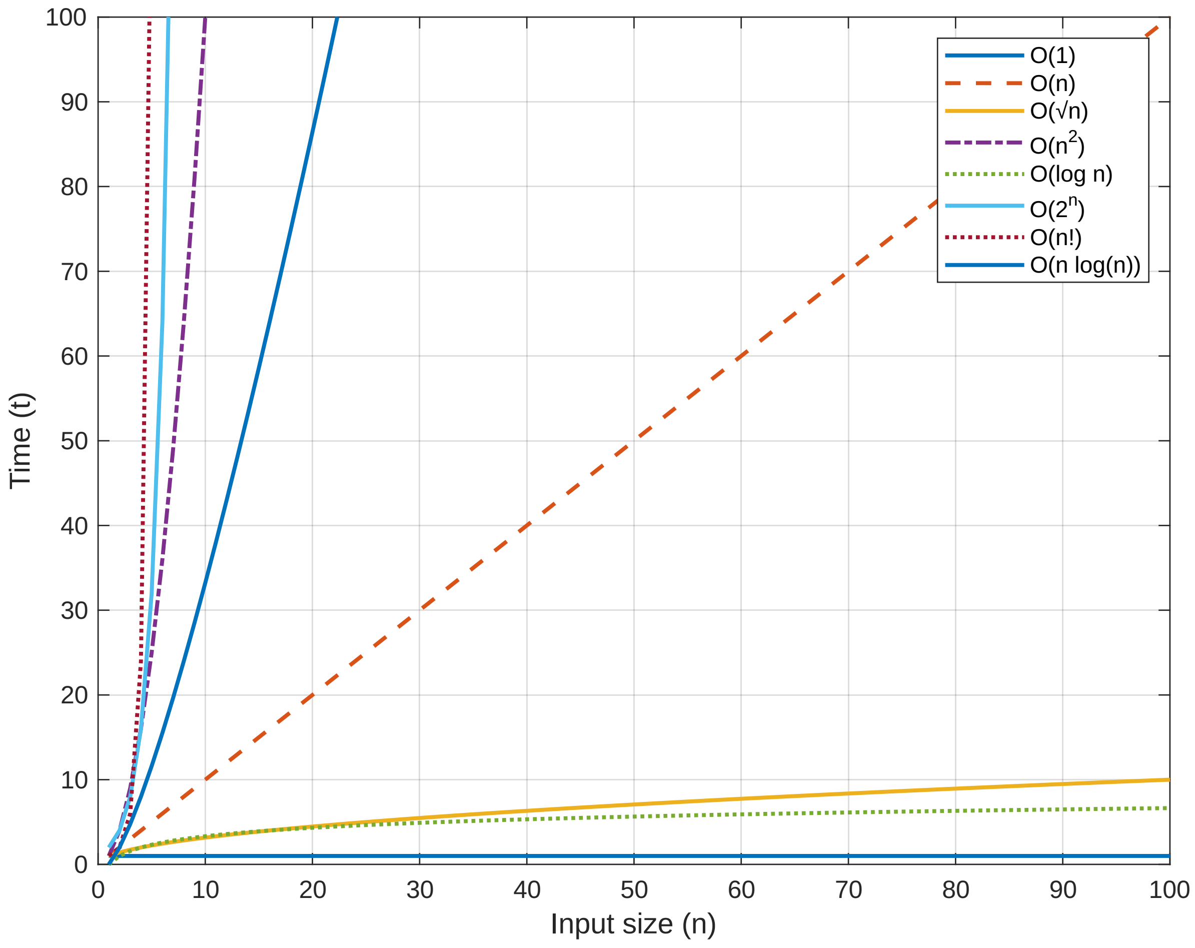


Figure 1.1: Common time complexities of sorting algorithms

Time complexity is written using the notation $O(g(n))$, where $g(n)$ is the asymptotic time complexity function of an algorithm. Figure 1.1 graphically represents the most common time complexities of sorting algorithms.

1.3 Space complexity

Space complexity is the amount of memory required by an algorithm to complete. A memory used for the input itself is not a part of the space complexity. Thus, the object of interest is the maximum number of elementary memory cells which are used at the moment. Like the time complexity, the space complexity is written using the notation $O(g(n))$, where $g(n)$ is the function expressing the number of memory cells used by an algorithm and n is the size of an input.

Algorithms are divided into:

1. *In-place* – Algorithms whose memory usage does not depend on the input size.
2. *Out-of-place* – Algorithms which are not considered in-place.

Chapter 2

Sorting algorithms

Sorting algorithms are a large and very well-studied part of computing science. The way how the data are sorted may highly influence the speed and difficulty of working with them. Therefore, sorting algorithms are widely used.

Their task is to sort the data in a particular order. These are most often numbers, which are sorted by a numerical value in ascending or descending order. However, the input can be elements of any type, but always of the same one. This chapter uses strictly numerical value examples of an ascending sorting order. Additionally, all the sorting mentioned algorithms are iterative as they are used in this form in the application.

2.1 Insertion sort

Insertion sort [4] is usually described as a method used by card players. The left hand represents the *sorted* part of the cards, and the deck of cards on the table is the *unsorted* part. This analogy describes how the algorithm works.

In general, insertion sort is very efficient while working with a small number of elements or while the elements are nearly sorted because, in these cases, the sorting can be done in linear time. However, the worst case of this algorithm is quadratic complexity. Therefore, this algorithm is often used as a part of other more complex algorithms.

Algorithm 1 Insertion sort

```
1: procedure INSERTIONSORT(A)
2:     for $i \leftarrow 1$ to $A.length - 1$ do
3:         $temp \leftarrow A[i]$
4:         $j \leftarrow i - 1$
5:         while $j >= 0 \wedge A[j] > temp$ do
6:             $A[j+1] \leftarrow A[j]$
7:             $j \leftarrow j - 1$
8:         $A[j+1] \leftarrow temp$
```

The name of the algorithm already suggests how the algorithm works. The principle lies in inserting elements into the final list. As mentioned above in the example with cards, elements are conceptually divided into a sorted and unsorted part. The first element is trivially sorted, and it is considered to be on the left side. Therefore, the insertion starts with the second element from the left. The algorithm gradually goes through all the elements in the unsorted part and compares the current element with already sorted ones, usually from the right to the left, to find its correct position where the element is later placed. If the current element is not smaller than the elements in the sorted part, it stays in place (see Figure 2.1(d)). Since the values of individual sorted elements are changed during the insertion of the current element, the value of this element is saved in the temporal variable.

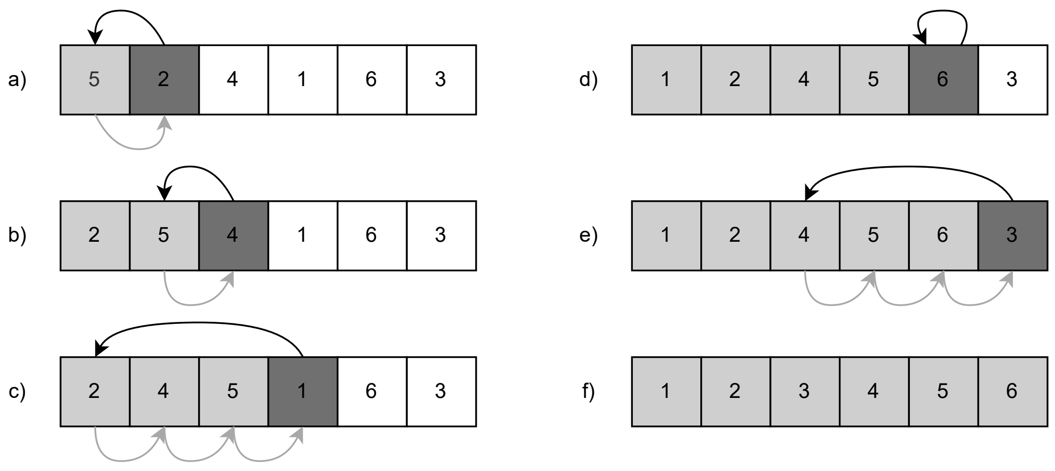


Figure 2.1: Example of a process of the insertion algorithm where **(a)-(e)** represent iterations of the algorithm and **(f)** is the completely sorted array. The light grey colour shows the sorted part, and the dark grey colour shows the current element.

2.2 Binary insertion sort

Binary insertion sort [28, 29] is basically insertion sort improved by binary search used for finding the right place for the current element. Basic insertion sort uses the linear search to find the location, which results in the comparison complexity of $O(n)$. However, when the binary search is used instead, the comparison complexity is reduced to $O(\log(n))$. Despite this reduction, the algorithm still needs to move all elements in the array when the correct position is found. This results in the time complexity of $O(n^2)$, which is the same as basic insertion sort.

Although pseudocode Algorithm 2 may look significantly different, the only difference is that the algorithm does not go through all the elements in the sorted part. Instead, it finds the correct position of the current element with a binary search represented by the while loop.

Algorithm 2 Binary Insertion sort

```
procedure BinaryInsertionsort(A)
    for $i \leftarrow 1$ to $A.length - 1$ do
        $temp \leftarrow A[i]$
        $left \leftarrow 0$
        $right \leftarrow i - 1$
        while $left <= right$ do                                ▷ Binary search
            $mid \leftarrow \lfloor (left + right) \text{ div } 2 \rfloor$
            if $temp < A[m]$ then
                $right \leftarrow mid - 1$
            else
                $left \leftarrow mid + 1$
        for $j \leftarrow i - 1$ downto $left$ do
            $A[j + 1] \leftarrow A[j]$
        $A[left] \leftarrow temp$
```

2.3 Selection sort

Selection sort [27] can be called the opposite of insertion sort. Instead of picking a particular element and finding the place, it picks a particular place and finds the smallest or the largest element. Furthermore, looking at Figure 2.2, it is clear that selection sort requires a smaller number of moves. However, the number of comparisons is much higher. This means that insertion sort and selection sort have exactly opposite advantages. However, these algorithms still have many similarities. They require minimum extra memory, are good with small data sets and are trivial.

Algorithm 3 Selection sort

```
procedure Selectionsort(A)
    for $i \leftarrow 0$ to $A.length - 2$ do
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $A.length - 1$ do
            if $A[j] < A[min]$ then
                $min \leftarrow j$
        $temp \leftarrow A[min]$
        $A[min] \leftarrow A[i]$
        $A[i] \leftarrow temp$
```

The algorithm conceptually divides sequences into the sorted and the unsorted part. The sorted part is empty at the beginning and is considered to be on the left. The algorithm then goes through all the elements in the unsorted part and finds the minimum, which is then swapped with the first element of the unsorted part. After inserting the minimum into the correct position, it becomes an element of the sorted part, and the unsorted part is decremented. This process continues until the unsorted part is empty.

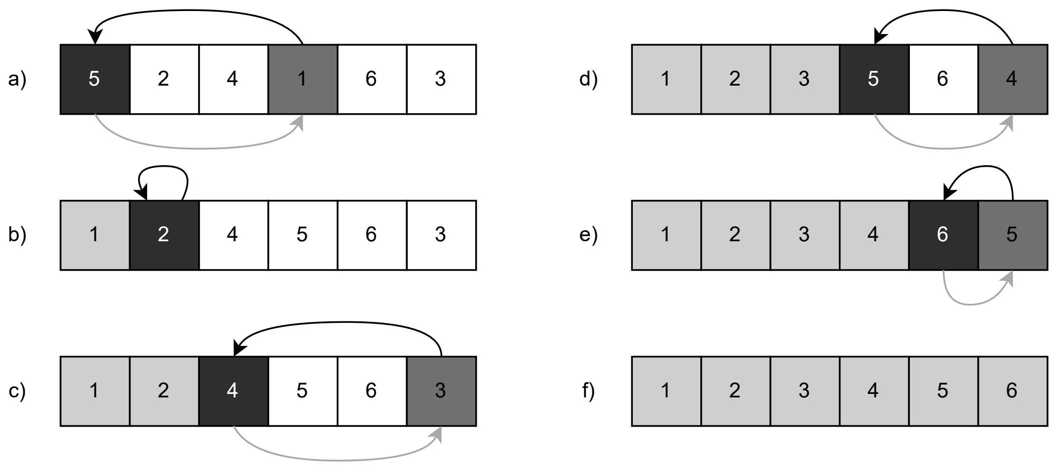


Figure 2.2: Example of a process of selection sort algorithm where **(a)-(e)** represent iterations of the algorithm and **(f)** is a completely sorted array. Light grey shows the sorted part, darker grey shows the smallest element, and the darkest colour marks the place where the smallest element will be moved.

2.4 Bubble sort

This part is based on [27, 28]. The name of this algorithm was derived from the idea of rising bubbles in soda drinks, where large bubbles gradually rise to the top. This means that the algorithm repeatedly goes through the whole list, moving the largest elements one by one to the end of the list. It is obvious that bubble sort is not very efficient as its complexity is quadratic. However, it is considered to be one of the easiest sorting algorithms and is often used as the first example.

Algorithm 4 Bubble sort

```
1: procedure BUBBLESORT(A)
2:     for $i \leftarrow 0$ downto $A.lenght - 2$ do
3:         for $j \leftarrow 0$ to $A.lenght - i - 2$ do
4:             if $A[j] > A[j+1]$ then
5:                 $temp \leftarrow A[j+1]$
6:                 $A[j+1] \leftarrow A[j]$
7:                 $A[j] \leftarrow temp$
```

As mentioned above, the algorithm goes through the whole list repeatedly, exactly $n-1$ times, where n is a number of elements in the list. In every iteration, the algorithm selects an element and compares it with the adjacent element, and if the left element is bigger, it swaps them. As it always compares two elements, it cannot select a single element at the end or beginning of the list. At the end of the iteration, the last element standing is marked as sorted.

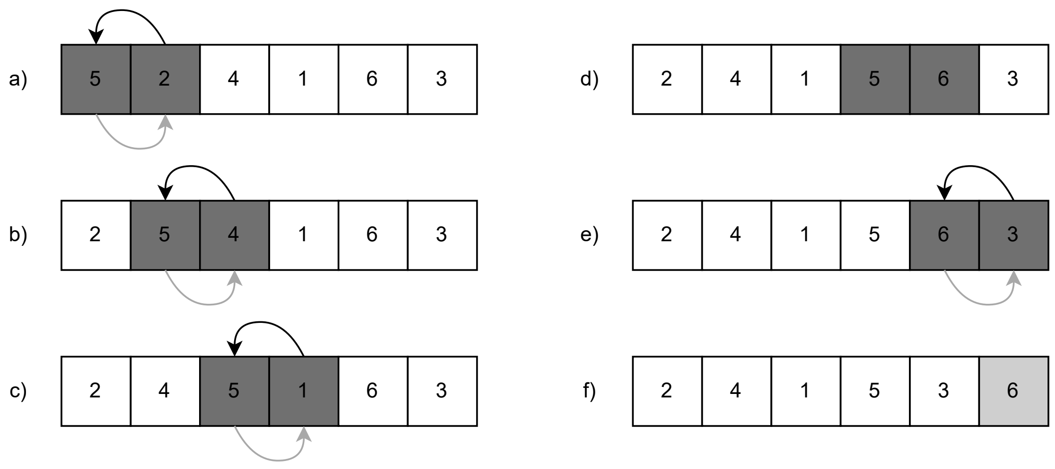


Figure 2.3: Example of the first round of bubble sort algorithm, where **(a)-(e)** represent iterations of the inner for loop of the algorithm and **(f)** is a completed round. Light grey shows the sorted element, and the dark grey colour shows the current elements.

2.5 Shaker sort

Shaker sort [27, 28] (sometimes called the cocktail-shaker sort [11, p. 110]) is an upgraded version of bubble sort. The algorithm goes both ways, unlike bubble sort. When the algorithm goes forward, it moves large elements to the end, and when it goes backwards, it moves small elements to the start. Despite this upgrade, the complexity of the algorithm is still quadratic.

Algorithm 5 Shaker sort

```
procedure SHAKERSORT(A)
    $start \leftarrow 0$; $end \leftarrow A.length - 1$; $swapped \leftarrow true$
    while $swapped$ do
        $swapped \leftarrow false$
        for $i \leftarrow start$ to $end - 1$ do
            if $A[i] > A[i+1]$ then
                $temp \leftarrow A[i+1]$
                $A[i+1] \leftarrow A[i]$
                $A[i] \leftarrow temp$
                $swapped \leftarrow true$
        if $swapped = false$ then break
        $swapped \leftarrow false$
        $end \leftarrow end - 1$
        for $j \leftarrow end - 1$ downto $start$ do
            if $A[j] > A[j+1]$ then
                $temp \leftarrow A[j+1]$
                $A[j+1] \leftarrow A[j]$
                $A[j] \leftarrow temp$
                $swapped \leftarrow true$
        $start \leftarrow start + 1$
```

The algorithm uses a *while loop* and two inner *for loops*. One is for the way forward, and the other for the way backwards. There is also an improvement in the form of *swapped* variable. Its function is to stop the algorithm if no element is swapped. Unlike bubble sort, shaker sort needs to decrease the number of elements it passes through on both sides. This requires separate variables, *start* and *end*.

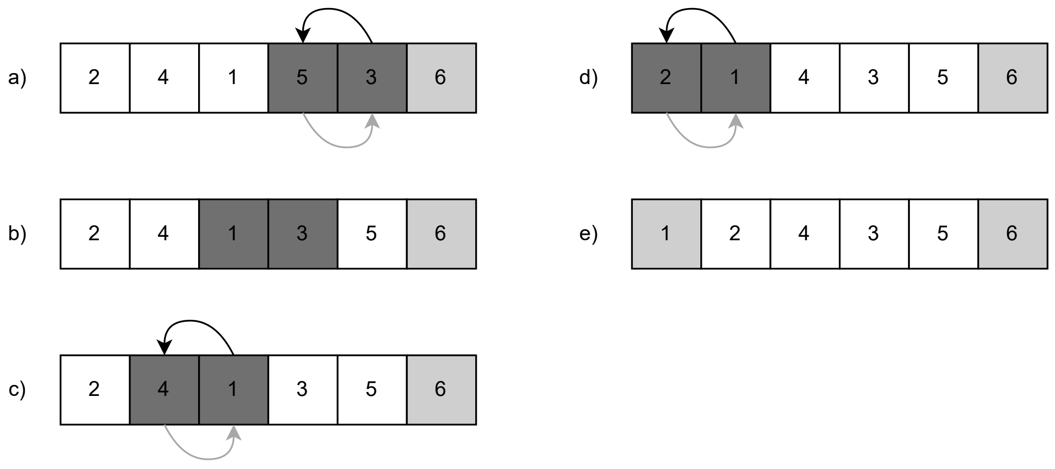


Figure 2.4: Example of the second round of shaker sort algorithm, where **(a)-(d)** represent iterations of the second for loop of the algorithm and **(e)** is a completed round. Light grey shows the sorted elements, and the dark grey colour shows the current elements.

2.6 Shell sort

Shell sort [27, 11, p. 83-95] algorithm was developed by Donald L. Shell in 1959. It can be considered an extension to insertion sort algorithm as it works on the same principle with the difference that shell sort does not compare only the adjacent elements. Thus, the advantage is that smaller and larger elements are quickly moved to the place where they belong.

Algorithm 6 Shell sort

```
1: procedure ShellSort(A)
2:     $gap \leftarrow \lfloor \frac{A.length}{2} \rfloor$
3:     while $gap > 0$ do
4:         for $i \leftarrow gap$ to $A.length - 1$ do
5:             $temp \leftarrow A[i]$
6:             $j \leftarrow i$
7:             while $j >= gap \land A[j - gap] > temp$ do
8:                 $A[j] \leftarrow A[j - gap]$
9:                 $j \leftarrow j - gap$
10:            $A[j] \leftarrow temp$
11:        $gap \leftarrow \lfloor \frac{gap}{2} \rfloor$
```

The algorithm uses the *gap* calculated at the start as half the length of the list and later as the half of the *gap* itself, which can be seen in the eleventh row of pseudocode Algorithm 6. However, there are different ways of setting the gap, which may lead to slightly better results. When the gap reaches a value of 1, the algorithm is degraded to insertion sort as the algorithm is sorting adjacent elements. However, the number of elements requiring a move in the last iteration is minimal.

The algorithm contains three loops. The first one covers iterations until the gap is greater than zero. The for loop goes through the groups of elements connected because of the gap, and the inner while loop swaps the elements in the ground as long as needed.

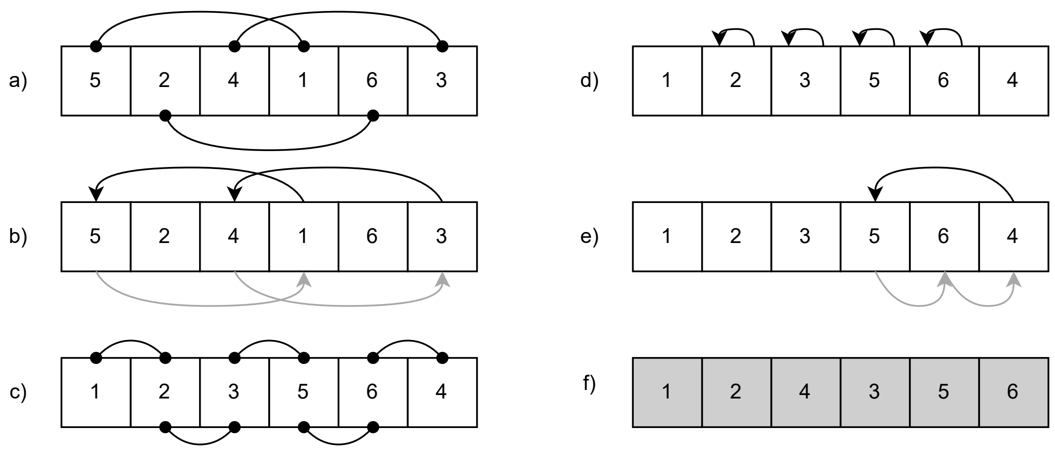


Figure 2.5: Example of shell sort algorithm. **(a)** and **(c)** show the connection between elements. **(b)** represents one iteration of outer while loop with the $gap = 3$. **(d)** shows the second iteration with $gap = 1$ to the point where $i = 4$. **(e)** shows the last iteration of for loop with $i = 5$. Lastly, **(f)** is a sorted list.

2.7 Heap sort

Heap sort [29, 11] was proposed and named by J. W. J. Williams in 1964 [7] and later it was improved by R. W. Floyd [11, p. 145]. This sorting algorithm is considered to be very effective, as its average complexity is $O(n \ln n)$ and it does not use any extra memory. The algorithm uses a very effective special tree-based data structure called heap. The max-heap is a binary tree, whose root is the largest element of the array, and all parents are greater or equal to their children. What is more, there is also a min-heap whose parents are smaller than or equal to their children. Generally, however, max-heap is the more used variant.

Heap sort can occur in many different interpretations in both recursive and iterative variants. Pseudocode Algorithm 7 shows the iterative variant. The basis of the algorithm is to create a max-heap. The example uses the Maxheap function for this task. After that, it just uses the information obtained by creating the max-heap. Repeatedly, the first element is removed from the max-heap. These removed elements are then gradually moved to the back of the array and are considered sorted. After each removal of an element, it is necessary to reorder the elements so that the root is the largest unsorted element.

Algorithm 7 Iterative Heap sort

```
1: procedure HEAPSORT(A)
2:     $MaxHeap(A)$
3:     for $i \leftarrow A.length - 1$ downto 1 do
4:         $temp \leftarrow A[0]$
5:         $A[0] \leftarrow A[i]$
6:         $A[i] \leftarrow temp$
7:         $j \leftarrow 0$
8:         do
9:             $index \leftarrow 2 * j + 1$
10:            if $index < (i - 1) \wedge A[index] < A[index + 1]$ then
11:                $index \leftarrow index + 1$
12:            if $index < i \wedge A[j] < A[index]$ then
13:                $temp \leftarrow A[j]$
14:                $A[j] \leftarrow A[index]$
15:                $A[index] \leftarrow temp$
16:            $j \leftarrow index$
17:        while $index < i$
1: procedure MAXHEAP(A)
2:     for $k \leftarrow 1$ to $A.length - 1$ do
3:         if $A[k] > A[\lfloor \frac{k-1}{2} \rfloor$ then
4:             $l \leftarrow k$
5:             while $A[l] > A[\lfloor \frac{l-1}{2} \rfloor]$ do
6:                 $temp \leftarrow A[l]$
7:                 $A[l] \leftarrow A[\lfloor \frac{l-1}{2} \rfloor]$
8:                 $A[\lfloor \frac{l-1}{2} \rfloor] \leftarrow temp$
9:                 $l \leftarrow \lfloor \frac{l-1}{2} \rfloor$
```

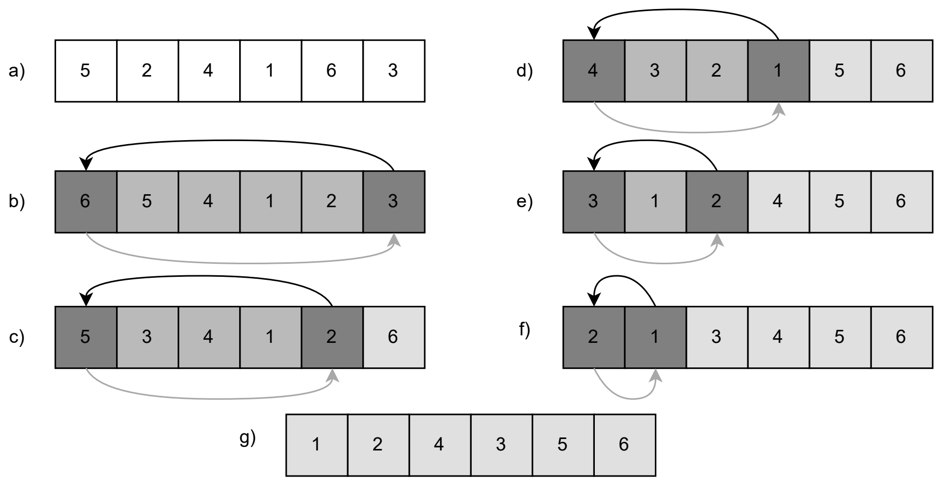


Figure 2.6: Example of a process of the heap sort algorithm where **(a)** is unsorted array, **(b)** shows the state of the array after the maxheap procedure, **(c)**-**(f)** represent iterations of the *for loop* in the heapsort procedure, and **(g)** is the completely sorted array. The light grey colour shows the sorted elements, the darker grey colour shows the max heap part of the array and the darkest grey shows the elements that are swapped at the beginning of each *for loop* of the heapsort procedure.

2.8 Quick sort

Quick sort [28, 27] was probably the most used sorting algorithm[1], although its complexity in the worst case is still $O(n^2)$. Despite the worst-case scenario, the average complexity is $O(n \ln n)$. The principle of this algorithm lies in swapping elements over a long distance. If the elements are in reverse order and the pivot is chosen correctly, the data can be sorted with only n/2 swaps. However, this case is rare.

Algorithm 8 Iterative Quick sort

```
1: procedure QUICKSORT(A)
2:     $top \leftarrow -1$
3:     $stack[++top] \leftarrow 0$
4:     $stack[++top] \leftarrow A.length - 1$
5:     while $top >= 0$ do
6:         $h \leftarrow stack[top--]$
7:         $l \leftarrow stack[top--]$
8:         $p \leftarrow partition(A, l, h)$
9:         if $p - 1 > l$ then
10:            $stack[++top] \leftarrow l$
11:            $stack[++top] \leftarrow p - 1$
12:        if $p + 1 < h$ then
13:            $stack[++top] \leftarrow p + 1$
14:            $stack[++top] \leftarrow h$
```

```
1: procedure PARTITION(A, L, H)
2:     $pivot \leftarrow A[h]$
3:     $i \leftarrow l - 1$
4:     for $j \leftarrow l$ to $h - 1$ do
5:         if $A[j] <= pivot$ then
6:             $i \leftarrow i + 1$
7:             $temp \leftarrow A[i]$
8:             $A[i] \leftarrow A[j]$
9:             $A[j] \leftarrow temp$
10:    $temp \leftarrow A[h]$
11:    $A[h] \leftarrow A[i + 1]$
12:    $A[i + 1] \leftarrow A[h]$
13:    return $i + 1$
```

The quick sort uses the *divide and conquer* method for sorting. The array is divided by the *pivot*, which is a chosen element from the array. There are many ways to choose the pivot: first, last, middle or random element. The next step is to rearrange the array so the smaller elements are before the pivot and larger elements behind it. The pivot will end up in the right place and is marked as sorted. Then, this process is repeated for the created sub-arrays on the sides of the pivot.

[1]In these days, intro sort is preferable as it combines quick sort and heap sort. The application implements both of these sorting algorithms and for this reason, it was considered unnecessary to implement intro sort itself.

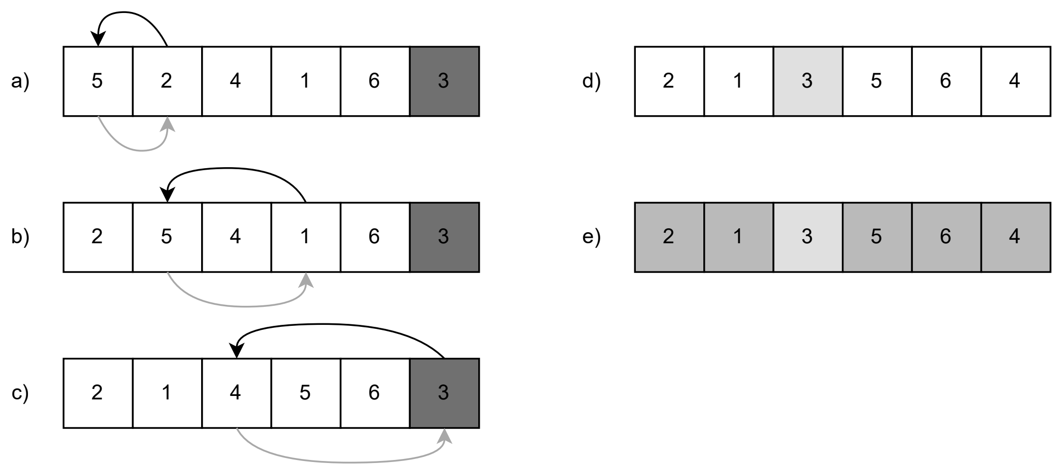


Figure 2.7: Example of one iteration of the iterative quick sort algorithm where **(a)-(d)** represent the steps of the partition procedure and **(e)** shows two sub-arrays created on sides of the pivot. The dark grey colour shows the pivot, lighter grey shows two sub-arrays and the lightest grey colour shows the sorted pivot.

2.9 Merge sort

Merge sort [27] is another algorithm that uses the *divide and conquer* method. The principle lies in merging (putting together) two or more elements. The algorithm was invented by J. von Neumann in 1945 [4, p. 42]. Unlike quick sort, this algorithm has the complexity of $O(n \ln n)$ in the average case.

Merge sort shown in pseudocode Algorithm 9 is its iterative version and, therefore, uses a bottom-up approach. The algorithm starts by comparing a group of two elements. The group is always divided into left and right sides and is subsequently stored in two auxiliary arrays L and R. The elements of arrays L and R are then gradually compared and put in the correct order. The size of the group is then doubled with each iteration of the outer *for loop*.

Algorithm 9 Iterative Merge sort

```
1: procedure MERGESORT(A)
2:     for $size \leftarrow 1$ to $A.length - 1$ step $size * 2$ do
3:         for $left \leftarrow 0$ to $A.length - 2$ step $size * 2$ do
4:             $mid \leftarrow min(left + size - 1, A.length - 1)$
5:             $right \leftarrow min(left + 2 * size - 1, A.length - 1)$
6:             $Merge(A, left, mid, right)$
1: procedure MERGE(A, L, M, R)
2:     $leftN \leftarrow m - l + 1$
3:     $rightN \leftarrow r - m$
4:     for $i \leftarrow 0$ to $leftN - 1$ do
5:         $L[i] \leftarrow A[l + i]$
6:     for $j \leftarrow 0$ to $rightN - 1$ do
7:         $R[j] \leftarrow A[m + 1 + j]$
8:     $i \leftarrow 0, j \leftarrow 0, k \leftarrow l$
9:     while $i < leftN \wedge j < rightN$ do
10:        if $L[i] <= R[j]$ then
11:            $A[k] \leftarrow L[i], i \leftarrow i + 1$
12:        else
13:            $A[k] \leftarrow R[j], j \leftarrow j + 1$
14:        $k \leftarrow k + 1$
15:    while $i < leftN$ do
16:        $A[k] \leftarrow L[i], i \leftarrow i + 1, k \leftarrow k + 1$
17:    while $j < rightN$ do
18:        $A[k] \leftarrow R[j], j \leftarrow j + 1, k \leftarrow k + 1$
```

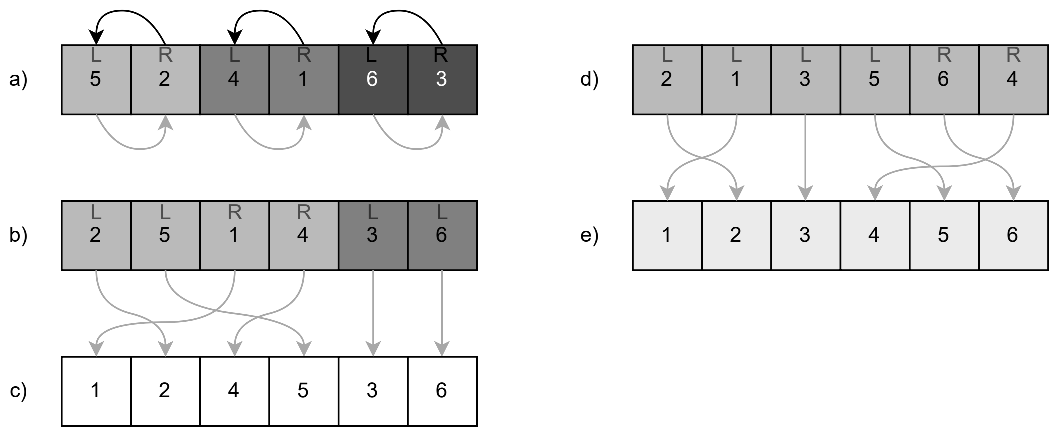


Figure 2.8: An example of iterative merge sort algorithm, where **(a)** shows the first iteration with a group of size 2, **(b)**-**(d)** show the next iterations with the larger group size, and **(e)** is the sorted array. The three darker shades of grey represent the individual groups into which the array is divided. The lightest grey then represents the sorted array. In addition, the individual elements of the group are marked with the letters *L* and *R*, depending on which auxiliary array they are stored in.

Chapter 3

Analysis

In this chapter, the analysis of competing applications is comprehensively described. Due to this analysis, a list of requirements could be determined, which are further detailed below.

3.1 Competitive analysis

This section is devoted to the analysis of already existing web applications enabling the visualisation and time comparison of sorting algorithms. As a number of applications have already been created for this purpose, the aim is to evaluate their strengths and weaknesses. The information obtained through this analysis will then be further used for the compilation of requirements.

3.1.1 xSortLab

xSortLab is primarily a program for learning algorithms. However, this program was implemented into a web application [5]. It includes visualisation and time comparison of sorting algorithms. Visualisation is done by an animated bar chart. The algorithm can be changed by the dropdown menu on the right side of the chart.

The visual sort can be done in the run or the step mode with a description of each step displayed below the chart. The application also measures the number of comparisons and copies on the right. The bar chart contains 16 bars, and the number of bars cannot be changed.

The time sort offers an option to write in the number of arrays up to 250 million and the number of items per array up to 500 million. It measures the time elapsed, the number of comparisons, copies, and already sorted arrays. However, there is no description of what programming language is used for sorting and how the algorithm is written. Below the time sort, there is a basic log system which will archive all time sorts done by the user until the page is refreshed.

Overall, although this web application has an old-looking design, the visual sort has excellent and clear animations, and the description of steps is well done. The most significant disadvantage is the lack of sorting algorithms, as the web application only includes the bubble, insertion, selection, merge and quick sort. Furthermore, there is no description of any of these algorithms. However, the idea of logging previous time sorts is inspiring.

3.1.2 Sort Visualizer

Sort Visualizer [25] is a web application created by a team of three people: Daniel Scanu, Tommaso Tovoli and Odd Elof Larsson. This web application includes a fairly large number of algorithms, and the visualisation is done by the animation of the bar chart. It also includes a description, a complexity table and a few examples of implementations.

The visualisation part of the web application uses not only the bar chart, but also other functioning features: a button for a random order of the bars, a play/stop button and a slider for setting the number of elements. The bars do not come with the number, which can lead to confusion in the case of setting a larger number of bars, as it is hard to see which particular bar is higher. As for the buttons, the button for random order works properly. However, the play/stop button behaves in a different way than a user might suspect, as the button does not stop the process of the animation. Instead it skips the whole animation to the already sorted bar chart. This misunderstanding is caused by the wrong button icon, which uses an icon known as a stop symbol instead of a right-facing arrow pointing to a vertical line. Furthermore, the buttons have a zooming hover effect, which can be seen as a design mistake. The slider bar behaves properly and has a range of 10 to 1000 elements.

The process of animation is not described in any way, excluding the change in colour of the bars, which is also not explained. The animation is unpleasantly fast, with a greater number of elements, which makes it even harder to understand how the algorithm works. The animation of randomising the bars is necessary as it can take more than 6 seconds to randomise the bars.

The description of the algorithm is placed on the left under the bar chart, and it is short and simple. The complexity table shows the average, best and worst time complexity together with the space complexity as last. The last section is the implementation. This includes examples of the code of the particular sorting algorithm in five different programming languages. The languages included are C, C++, Java, JavaScript and Python.

In summary, the Sort Visualizer is a web application including a large number of sorting algorithms, from the more well-known algorithms to the almost unknown ones by the majority of people. The idea of including the implementations in the code and the complexity table is compelling. However, this website is not suitable for educational purposes, as the description of the steps is missing, and the animation is fast and cannot be stopped.

3.1.3 visualgo.net

Visualgo.net [3] is a web application created for the educational purpose of the students of Dr. Steven Halim. The web application includes the visualisation part in the form of a bar chart, with the exception of the radix sort. On the left there are the buttons for manipulation with the array and the chart and the slider for the speed of animation. In the middle of the bottom bar there is a panel for controlling the animation, and on the right there is the description of the steps and the code of the particular algorithm.

The web application offers two settings for the chart bar, which can be changed with the button on the left. The default setting, which has the numerical values attached to the individual bars. The compact setting, which is more suitable for mobile devices as there are no numerical values and the bars are thinner. The next button provides various settings of the array with the option to enter ones own sequence with a limit of 50 elements. The last button starts the sorting algorithm. The animation starts instantly, and it needs to be stopped manually to perform the steps separately. This can lead to several steps being overlooked, and the process of the algorithm has to be manually undone with the “step backward” or “go to beginning” button. The last outstanding element is the steps bar, which basically tracks the process and can be used similarly to the video timeline.

The description of the steps is clear and concise. However, the code example of the algorithm might create confusion, as it does not have a standard structure of pseudocode. The current steps of the process are highlighted in the code. The web application also change the colour of the bars based on the steps, which can lead to confusion as merge sort uses the whole spectrum of colours, and it is easy to lose track of the process.

Ultimately this web application has a significant animation, which could be expected, taking into account the number of people who worked on this web application. It also offers many settings for the chart and array itself, which can, however, be unnecessary for educational purposes, as they are rather bonus features. The code representation of the algorithm has a more informal form, as it is not a standard structure of pseudocode, which is one of the most significant disadvantages of this website, together with the auto-play animation.

3.1.4 Comparison sorting algorithms

Comparison sorting algorithms [8] is presented as an educational web application developed by David Galles. The web application includes six sorting algorithms, but in fact it contains four more algorithms. These algorithms have their own page and they can be found in the web guide [9].

This web application includes only the visualisation part. There is no description of the algorithm and the steps at all. The upper row of the buttons is used for randomising the array and choosing the algorithm. In the middle, there is a bar chart used for visualisation of the array. Each bar has its own number displayed directly under the bar. The bar uses different colours in the sorting process. However, this is the only useful tool offered by this web application. As for the other four sorting algorithms with their own pages, the middle part uses different visualisation depending on the particular algorithm. For example, the heap sort uses the tree. The last part is the button bar under the bar chart. It includes buttons for controlling the animation and the process of sorting, a slider for the animation speed, and an option for changing the canvas size. However, the change of the canvas seems to be bugged, as it is impossible to enter numbers with the numerical keyboard.

Overall, this web application has very non-intuitive controls, as half of the sorting algorithms is separated on their own pages, which are accessible only through the web guide. Moreover, the link to the web guide is not highlighted in any way, and it looks like the plain text. The animation looks relatively good, but without a description, it is hard to understand how some of the algorithms work. Thus, this web application is not suitable for educational purposes.

3.1.5 algoritmy.net

Algoritmy.net [19] is a Czech website containing various content specialised in algorithms and other topics dealing with informatics. The section on sorting algorithms includes the theory and number of sorting algorithms, from the well-known ones to the almost unknown ones. Although the web application is written in Czech, the sorting algorithms have their names written in English. The page is divided into several parts: description, principle, example (not for all algorithms), visualisation, and code. However, some algorithms have their own specialised sections, e.g. algorithm performance on the quick sort page.

The content size of the description varies from algorithm to algorithm. In general, it contains only a few sentences. However, the principle tends to be denser in content, as more than a few sentences are needed to explain the individual algorithms. The more complex the algorithms, the more explanation is needed. The example section, if present, includes the manually written input array with the description of steps until the array is sorted. Unfortunately, the example section comprehends only a few algorithms.

This web application uses three types of visualisation: picture, GIF and animated visualiser. However, the visualisation part is present for only some sorting algorithms, and it seems that if it is a more complex algorithm to implement, it does not include animated visualisation.

The animation is done on a bar chart with twenty bars. The number of bars cannot be changed, and there are no numbers tied to the bars. When looking at two bars which are not next to each other, it might be unclear which bar is higher. Under the bar chart there are located three buttons: start, stop and reset, and they work as expected. The animation is relatively fast and uses different colours during the sorting, but the steps are not described.

The last section is an example of the code. This section is part of all the sorting algorithms available on this web application. However, the examples written in different programming languages differ from algorithm to algorithm, and some of them include only pseudocode.

In general, the problem with this web application lies in inconsistency. Different algorithms contain different sections, and even sections themselves are inconsistent. Although it contains a fairly large number of sorting algorithms, its educational capabilities are very low. What could also be unacceptable is a casino advertisement in the lower part of the page.

3.1.6 toptal.net

Toptal.net is mainly a company providing a platform for freelancers. However, their web also includes some educational or interactive tools, e.g. sorting algorithms animations [2]. The main subject of interest is the table of bar charts. This table includes eight algorithms and offers the animation on several different initial conditions of the array, including: random, nearly sorted, reversed and few unique. Animation can be played in three ways. The first one is to choose the algorithm, which plays an animation of all four initial conditions of the algorithm at the same time. The second one is to choose an initial condition, which starts the animation of all the algorithms at the same time. The last option is to play all the algorithms with all the initial conditions at the same time.

Under the table of charts, there is an option to open a particular algorithm or initial condition on a separate page and change the number of bars in the bar chart. There is also a *key* section briefly describing the colours and helpful tools of the table.

In summary, this page is an efficient visualisation tool rather than an educational application, as there is no description of the included sorting algorithms. The animation is too fast, and it is difficult to understand how the algorithm works.

3.1.7 Sorting Visualizer

Sorting Visualizer [26] is a web application created in 2021 by a student of the University of Engineering and Management in Kolkata, Sneha M. [1]. The assignment was to create a web application using HTML, CSS and JavaScript to create a visualisation for various sorting algorithms.

This web application has a very clear and simple interface. There are buttons for choosing an algorithm on the right. In the middle there is a bar chart and two sliders. The first one is used for changing the number of bars in a range of 5 to 100 bars. However, the slider does not display the value, so the user has to count the bars manually. The second slider is for the animation speed, which also does not display any value. The last button is on the left side and is used for randomising the array.

The main subject of the page, the bar chart, is placed upside down, which may seem confusing, but it does not cause any complications in the sorting and visualisation itself. The colour of the bars is changed throughout the sorting, indicating what is going on. However, there is no description of the algorithm or the steps. Once the animation is started, it cannot be stopped, and the animation speed is the only part of the interface that remains active. Thus, if the user wants to change the sorting algorithm, they have to wait for the end of the animation or reload the page.

Overall, this web application is a great visualisation tool, but it is not suitable for educational purposes because it lacks a description of how the algorithms work. The most significant disadvantages are the missing stop button and the values of the sliders, as it complicates working with the application.

3.1.8 Summary

Overall, all web applications have in common that, to some extent, they contain the visualisation of sorting algorithms. The vast majority of the sorting algorithms were visualised using the bar chart, as it can be considered the easiest way to demonstrate how these algorithms work. However, several applications also include other features that are more or less useful for educational purposes. The aim of this analysis was, therefore, to map the diverse functions offered by already existing applications and to further consider the usefulness of these individual functions.

Web application	**Control**	**Clarity**	**Description**	**Code**	**Data**
xSortLab	4	5	5	-	1
Sort Visualizer	2	1	-	-	2
visualgo.net	5	5	5	4	5
Comparison sorting algorithms	5	5	-	-	1
algoritmy.net	2	3	-	-	1
toptal.net	1	2	-	-	3
Sorting Visualizer	1	2	-	-	2

Table 3.1: Competitive web application ranking table with regard to visualisation.

Table 3.1 shows the rating of individual web applications based on their features with regard to visualisation. The scale used to evaluate applications ranges from 1 to 5, the highest value being 5 and the lowest being 1. Applications that did not contain the evaluated function have a minus sign written in the column.

The features evaluated are as follows:

1. *Control* – Possibility of controling the visualisation process. Including the possibility of playing, stopping and stepping animation, but also other features such as animation speed and the look of the bar chart itself.

2. *Clarity* – Clarity of the visual process and to what extent the user is able to understand the sorting algorithm from it.

3. *Description* – Description of the individual steps of the sorting algorithm explaining why the algorithm performed a particular step.

4. *Code* – Preview of the code showing which steps are being executed.

5. *Data* – Options for choosing the data. They include the possibility of both determining the number of data and setting initial conditions, e.g. descending or random data.

3.2 Requirements

The requirements for the web application are compiled based on the instructions of the bachelor's thesis itself, as well as on the analysis of already existing applications, which was carried out in the previous section.

3.2.1 R1 Visualisation method

Based on the analysis of other already existing applications, the visualisation of the sorting algorithms will be carried out using a bar chart. For better clarity, each bar will have its own value displayed at the top.

3.2.2 R2 Control of the visualisation process

The web application should allow the user to play, stop and step the animation.

3.2.3 R3 Visualisation process clarity

The process will have to be straightforward so that the user will be able to understand the individual functions of the sorting algorithm. Both colours and other functions of the web application will be used for this purpose, see 3.2.5, 3.2.4 and 3.2.7.

3.2.4 R4 Step description

The application should include a text description explaining each step of the algorithm. The description should explain what kind of action took place and, possibly, explain why it happened.

3.2.5 R5 Sorting algorithm pseudocode

The application should contain pseudocode of the individual sorting algorithms, which will clearly show which parts of the code were executed in each step. Individual lines will be highlighted depending on the step taken. If the executed code is part of, for example, a while loop, it will be marked in the code that this loop has not yet been exited.

3.2.6 R6 Data generation

The user should be allowed to set the number of elements (to a limited extent) and generate their random and descending order.

3.2.7 R7 Live projection of variables

The application should include a table containing the variables of the sorting algorithm. These variables will be updated live, and will be consistent with pseudocode that will also be part of the application.

3.2.8 R8 Information about the sorting algorithm

Since the application will be used for educational purposes, it should include a short and concise description of sorting algorithms as well as a complexity table. The table should contain the minimum, average and maximum time complexity and also the memory complexity.

3.2.9 R9 Sorting algorithms speed comparison

The application will contain a comparison of the speed of all included sorting algorithms. The user should be allowed to choose from several data sets of different initial conditions and sizes. The results of the comparison will be stored in the page log system and will be available until the page is refreshed.

Chapter 4

Technologies

The web application was developed using HTML, CSS, PHP and JavaScript. In addition, the *D3.js* JavaScript library was used to generate the bar chart and provide all the animations. The development environment used for this project was Visual Studio Code. The technologies were chosen according to practicality and overall popularity among the wider public in the developer sphere.

4.1 HTML

HTML (HyperText Markup Language) is probably the most used markup language for creating web pages. The language works on the basis of so-called tags. These tags carry the name of their purpose, and when using individual tags, this name is written in angle brackets. Tags are divided into paired and unpaired tags. Paired tags are, as the name suggests, written in pairs. The second part of the pair contains a slash inside the angle brackets. What is more, tags can contain attributes that provide the tag with additional information and also allow, for example, referring to a specific tag and changing its content. Among the frequently used attributes are *class*, *id*, *href*, *src* or *style*.

Every HTML often contains an initial header <!DOCTYPE html> which defines the document itself. Although it contains angle brackets, it is not an HTML tag and it only serves as information for the web browser about the type of document, as the name itself suggests. The header is followed by a paired <html> tag, which is the root element of the page. It is a kind of wrapper for other HTML elements. The <html> tag often contains the language attribute of the web page.

The <html> tag is further divided into sections created using the <head> and <body> tags. All mentioned tags are paired. The <head> section usually contains information that can then be displayed in search engines, and all stylesheets (see 4.2) and scripts are also linked here. The <body> contains all the content of the page and can be further divided using the <main> and, possibly, <footer> tags.

4.2 CSS

CSS[1] (Cascading Style Sheets) is a language used for styling all the elements contained inside an HTML file. Individual styles can thus be applied both to general HTML elements and to individual specific tags if they are marked with the ID or class attribute. CSS then allows the change of colours, fonts, dimensions, and the entire layout of the web page. This styling can then be divided into individual

[1]More information about the CSS can be found at `https://web.dev/learn/css/` and `https://developer.mozilla.org/en-US/docs/Web/CSS`.

files with the .css extension. Each file can control, for example, only a section of the page. One CSS file can take care of styling the header, while the second one styles the body of the document.

The syntax of the CSS language is very simple. The file contains individual sections that always start with the name of an element that will be styled. Styling an element using an ID attribute requires adding a hashtag sign before the name of the attribute. The same applies to the class attribute with the difference that a dot is used instead of a hashtag. After the name of an element or attribute of the element is followed, curly brackets in which the stylisation itself is located. It is written in the form, e.g. color: white;.

4.3 PHP

PHP[2] is a widely used scripting language. The first version of PHP was created by Rasmus Lerdorf and was published in 1995. The simplicity of this language is probably responsible for its success. It is primarily used together with HTML, as PHP was designed for generating its content. PHP is installed on the server and is usually used for communication with a database, e.g. MySQL [12].

PHP therefore performs actions on the server side and only then sends the result to the browser. PHP can often be inserted as part of the HTML file; in this case, the code is placed in angle brackets and question marks.

4.4 JavaScript

JavaScript is one of the most widely used object-oriented scripting languages for creating web pages. JavaScript is supported by the majority of web browsers, and together with the fairly easy syntax, which is similar to the C language, it is an obvious choice for most programmers. It is possible to use the language outside the web environment, but it is not its main purpose. The language was first released by Netscape in 1995 and has become a standard not long after [24].

JavaScript is therefore often used together with HTML. It is possible to write the script directly into the HTML file using the pair tag <script> </script>, where the script itself will be located. The second option is to create a separate file with a .js extension that is referenced in the HTML file. In the second case, it is also necessary to use a pair tag, but the code itself is located outside the HTML file.

When creating a website, often not only pure JavaScript is used, but also JavaScript libraries and frameworks. Since the existence of JavaScript, a large number of them have been created. These can often make website creation much easier. Popular libraries and frameworks include React, jQuery, D3.js, Vue.js, and Lodash. However, there are a lot more, which are specialized in particular tasks.

4.5 D3.js library

D3.js library is an open-source JavaScript library created by Mike Bostock in 2011. It is mainly used for creating interactive visualisation of data using the standard Scalable Vector Graphics (SVG) format. This library is very extensive; it contains both tools for visualisation and a vast number of options for animation and interaction with data [23].

This library does not necessarily make the programmer's work easier in the sense of shortening the code to a few lines, but rather offers options. Therefore, this library is unsuitable for a simple display of data in a chart, which is not interacted with in any way or does not need to be animated. For such cases, there are many other libraries that are easier to use.

[2]Additional information about PHP can be found at `https://www.php.net/manual/en/`.

Dynamic visualisations are therefore created through the connection of data and DOM(Document Object Model) elements, where this connection enables various manipulations with these elements, such as creating, removing or modifying elements. In this case, elements can directly react to data changes and use smooth and dynamic animations.

As this library is mainly used for visualisation of data, there are already a lot of pre-prepared methods for creating bar charts, pie charts and other graphic representations. The same principle applies to the animations; there are multiple animation presets with various animation progressions. All these preset options can be adjusted so that every aspect of the visualisation matches according to preferences.

Chapter 5

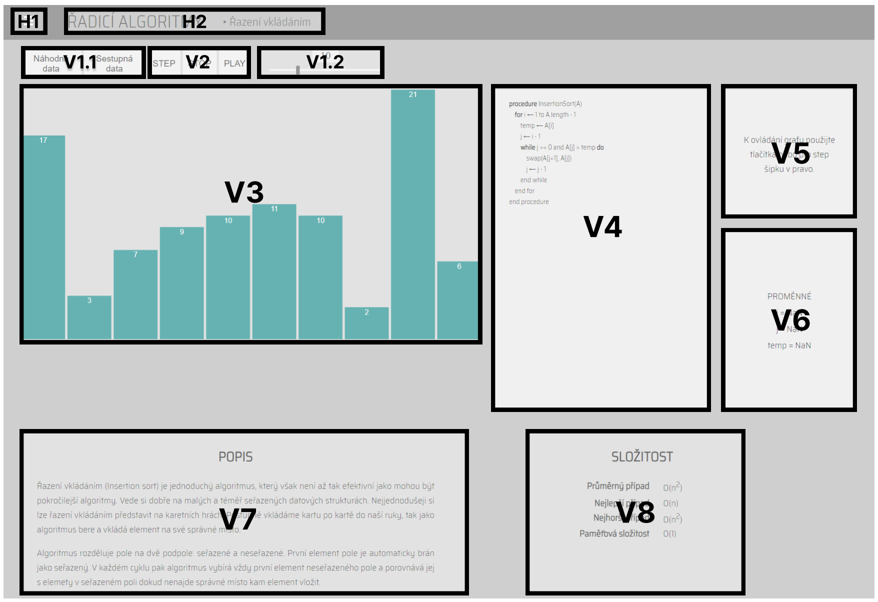
Implementation

This chapter will explain in detail the application design created on the basis of the analysis and requirements for the application, the diagram of activities within the interactive elements of individual parts of the application and the functions operating the logic behind the entire application.

Some of the code samples described in this chapter can be found in the appendix. The source code is available for download from GitHub[1].

5.1 Application layout

The following Figures show a simplified layout of the most important elements of the visualisation and of the time comparison part of the application. All these elements will be properly described and tabulated to show clearly the coverage of all application requirements. It should be remembered that all these elements have been created to meet the requirements described in the requirements Section 3.2.



```
procedure InsertionSort(A)
  for i ← 1 to A.length - 1
    temp ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > temp do
      swap(A[j+1], A[j])
      j ← j - 1
    end while
  end for
end procedure
```

Řazení vkládáním (Insertion sort) je jednoduchý algoritmus, který však není až tak efektivní jako mohou být pokročilejší algoritmy. Vede si dobře na malých a téměř seřazených datových strukturách. Nejjednodušeji si lze řazení vkládáním představit na karetních hrách. Postupně vkládáme kartu po kartě do naší ruky, tak jako algoritmus bere a vkládá element na své správné místo.

Algoritmus rozděluje pole na dvě podpole: seřazené a neseřazené. První element pole je automaticky brán jako seřazený. V každém cyklu pak algoritmus vybírá vždy první element neseřazeného pole a porovnává jej s elemety v seřazeném poli dokud nenajde správné místo kam element vložit.

Průměrný případ	$O(n^2)$
Nejlepší případ	$O(n)$
Nejhorší případ	$O(n^2)$
Paměťová složitost	$O(1)$

Figure 5.1: Basic layout of elements of the visualisation part of the application.

[1]https://github.com/Gatorixx/Bachelors-project

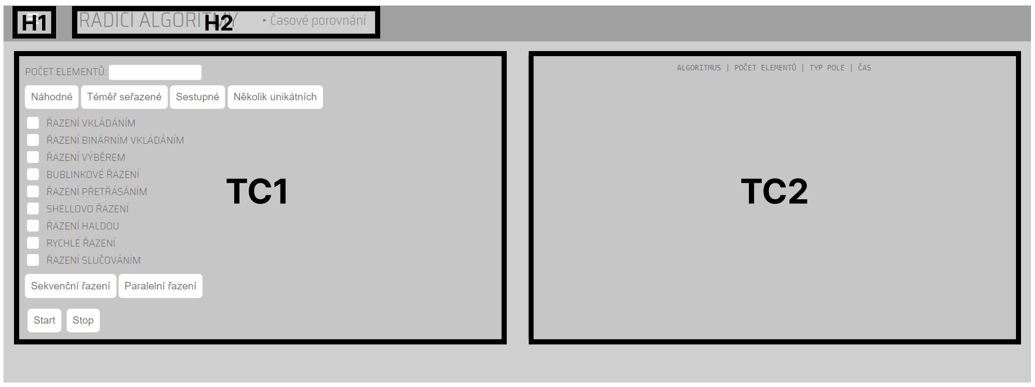


Figure 5.2: Basic layout of elements of the time comparison part of the application.

Figures 5.1 and 5.2 contain the following elements:

H1: pop-up menu for selecting subpages

H2: heading informing the user which of the subpages they are currently on

V1: two buttons and a slider for changing data

V2: step, play and stop buttons allowing the user to control the visualisation process

V3: bar chart

V4: pseudocode

V5: text description

V6: variables

V7: description of the algorithm

V8: complexity of the algorithm

TC1: form for setting initial conditions and choosing algorithms for time comparison

TC2: log system

The following table shows how the above elements cover the application requirements. At least one element covers each of the requirements.

	V1	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**	**TC1**	**TC2**
R1			x							
R2		x								
R3			x	x	x	x				
R4					x					
R5				x						
R6	x									
R7						x				
R8							x	x		
R9									x	x

Table 5.1: Requirements coverage mentioned in Section 3.2

5.2 Content and styling

All the content on the pages is written in HTML, but since the header and footer are added here using PHP, it is necessary that all these files end with the suffix .php. This structure makes it possible to have the header and footer in separate files and does not have to be repeated.

The styling of individual pages is done using CSS, which is located in their own files. As such, the application is targeted for use on a desktop device. However, the added styling allows it to be used on mobile devices as well.

5.3 Activity diagrams

The following activity diagrams simply describe the activities of the visualisation and time comparison part of the application. Due to this diagram, the basic logic of the application can be understood even without knowing the specific functions. However, the individual functions that are triggered when working with the application will be described in the following chapter.

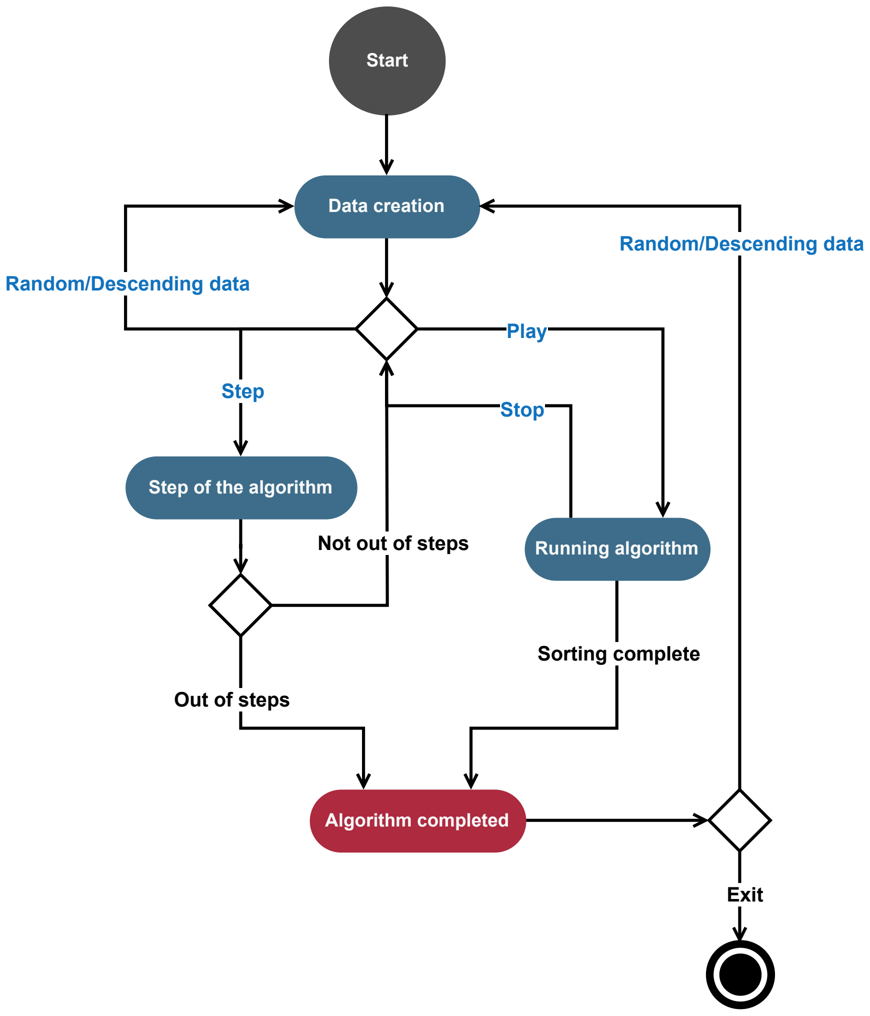


Figure 5.3: Activity diagram of the visualisation part of the application

Figure 5.3 describes the activities in the visualisation part of the application. The blue text indicates the flow of control triggered by interactions with individual user interface (UI) elements. The black text then indicates a flow of control that does not require any user action.

At the beginning, when the application is loaded, the data are automatically created. These data can then be repeatedly changed using the random and descending data button or slider. The step and play button will run the given sorting algorithm. However, using the step button, the sorting algorithm stops at predetermined places and, if not completed, it waits for the user's instruction. If the running of the algorithm was started using the play button, the algorithm runs until it reaches the end or is interrupted by the stop button. So the user can start, stop and step the algorithm at will because he always gets either to the end of the algorithm or to the initial decision. However, it is necessary to consider that the re-creation of the data resets the entire algorithm process. After completing the algorithm, it is possible to create new data and repeat the whole process of activities again.

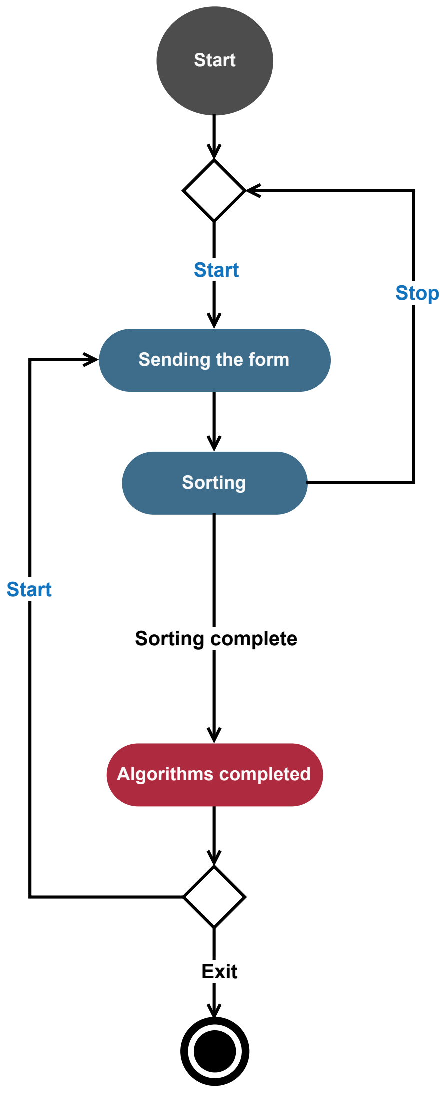


Figure 5.4: Activity diagram of the time comparison part of the application

Figure 5.4 describes the activities taking place in the time comparison part of the application. As with the visualisation part, the blue and black colours of the text here represent the same thing.

At the start, the user fills out the form containing the initial settings for the arrays, which the selected sorting algorithms will then sort. This form is submitted with the start button, and after submitting the form, the sorting process of the selected algorithms is automatically started. All selected algorithms can be run parallelly or sequentially using workers, which will be explained in a separate chapter. Then, the sorting activity can either be interrupted using the stop button (whereby it would return to the initial state) or the activity finishes. From this state, the user can return to sending the form using the start button.

5.4 Application functionality

Since the application has many smaller and less essential functions, only the essential ones will be mentioned in this chapter. Individual methods of the D3.js library and JavaScript workers will also be explained here.

5.4.1 Data and bar chart creation

When the user loads the algorithm visualisation page, it is necessary to create data and elements forming the bar chart. Furthermore, the chart must be updated if the data change.

Data generation

The application offers the user two types of initial arrays: Random and descending. The random array is generated using `Math.random()` [14], which returns a pseudo-random number between zero and one. This number is then multiplied, so the resulting array contains numbers between 0 and 25. However, this generation method is not completely random, as it depends on the seed and, therefore, it is predictable. However, for the purposes of this application, this method of number generation is sufficient. A descending array has the value of the first element derived from the size of the array; each subsequent element has a value reduced by one.

Bar chart creation

The D3.js JavaScript library is used here to create elements in the SVG HTML element. At the beginning, it is necessary to determine the size of the chart itself and then create the SVG element using <svg> tag, as the bar chart will then use these values to generate individual chart bars. SVG element is simply added to an HTML document using the `.append()` method. Then, the array values must be mapped to the available area of the SVG element. For this purpose, `d3.scaleBand()` is used for the *X* axis and `d3.scaleLinear()` for the *Y* axis.

`d3.scaleBand()` is mainly used when it is necessary to divide the elements evenly in the assigned range. Since each bar has the same width, this function is the most suitable. Thus, the input domain is each data point in the array, and the range is the size of the area intended for the chart.

`d3.scaleLinear()` is mainly used for continuous numerical data, where the input domain is the range of these numerical values. In this case, the domain is the scale from zero to the maximum value in the array. The range is determined the same way as the previous. The only difference is the *Y* axis and the start point, which starts at 20 instead of zero because even a bar with a value of zero must have some height.

The `.attr()` method is then used to set the individual attributes of the elements. This method accepts two parameters. The first is the name of the attribute itself, and the second is the value to be set for this attribute.

The bars are inserted into the HTML using the `.append()` method and bound to the data in the array using the `.enter()` method. This method is used when creating and then binding new data points that have not yet had an associated domain object model (DOM) element. Using these D3.js JavaScript library methods described above, bars can easily be assigned the correct size and position on the *X* and *Y* axes. The same process then takes place for the text value of the bar.

Updating the bar chart

The chart is updated when the data themselves are changed. In this case, changing the range is no longer necessary, but it is sufficient to change the input domain only. This occurs because the highest value in the array changes, and probably the length of the array changes as well. Most of the methods are the same as when creating the bars, with the difference in using the `.join()` method instead of `.enter()`.

The `.enter()` method is mainly used in cases when new data points have not yet had their assigned DOM element.

In contrast, the `.join()` method combines create, update and exit. This means that if the data points do not have an assigned DOM element, these elements are created. If the data points have an associated DOM element, they are updated based on the new data. Finally, if there are elements that no longer have data points assigned to them, they are removed. Thus this method simply combines all actions that may be needed when changing data. In addition, there is no need to use the `.append()` method.

Animation methods provided by the D3.js are also used here. These methods are not modified fundamentally, and their function itself does not need to be explained.

5.4.2 Implementation of sorting algorithms

Sorting algorithms are used in their basic form, except for minor changes needed for their visualization. As for sorting algorithms that use recursion, their iterative versions have been used for better clarity.

The following functions have been added to each sorting algorithm to provide all the added features of the application:

1. `waitForUserInput()`: pauses the sorting algorithm function and waits for a user input
2. `swap_bars()`: swaps selected chart bars
3. `pseudoColours()`: colours individual lines of the pseudocode
4. `writeVar()`: updates the displayed variables
5. `setColour()`: colours the chart bars
6. `writeText()`: updates the description of the steps

Sorting algorithms are paused at predetermined places in the code using the `waitForUserInput()` function. These places were chosen so that the individual steps made the most possible sense. For added functions, several conditions that were not in the original code of the given sorting algorithms had to be added. These added functions will be explained in detail in Sections 5.4.3 and 5.4.4.

There is a special exception for binary insertion sort. This is due to the need to display arrows to better understand the binary search. Thus, separate functions are created for the arrows, which change their position depending on the bars.

5.4.3 User instructions

This function `waitForUserInput()` always ensures that the user's instructions are waited for. The exception, however, is using the play button. This button sets the *play_flag* variable to true, in which case the algorithm continues until it ends or the user presses the stop button. This button thus switches the *play_flag* variable to false, and the `waitForUserInput()` function again pauses the algorithm.

This function is implemented as an asynchronous function that returns a `promise`. `Promise` is the object used for these asynchronous operations and can be in three states:

1. *Pending* – The initial state of the operation.
2. *Fulfilled* – The operation was successful (`resolve()` function was called).
3. *Rejected* – The operation was not successful (`reject()` function was called).

A `promise` thus waits in the pending state until it is fulfilled or rejected. The `new Promise()` constructor takes a function, referred to as the executor function, as an argument. In the executor function, `event listeners` are then added to the individual interactive elements, which, in the event of interaction, causes (among other things) calling the `resolve()` method. It should be noted that the `waitForUserInput()` function must be called with the `await` keyword, so it always waits for user interaction.

5.4.4 Chart handling functions

Bar swap and animation

The `swap_bars()` function handles swapping the places of two bars. These bars not only need to override their *X* and *Y* values but they also need to change their *IDs*. The *IDs* are exchanged because they must always be in the right order as they are used for selecting the bars. The same applies to their text assigned. The exception in this regard is merge sort, where it is necessary first to change the position of all active bars, and only then are the *IDs* of all these bars changed at once.

First, both the bars and their texts are stored in temporary variables, including their *X* and *ID* parameters. Furthermore, the following special methods of the D3.js library are used for the animation of bars and their texts:

1. `transition()` – it smoothly animates the transition from the current state to the target state of selected DOM elements. This applies not only to changing the position but also, for example, to changing the colour [21].
2. `duration()` – its parameter specifies the animation duration of each selected element in milliseconds [22].
3. `ease()` – its parameter specifies the speed curve of the animation. However, the parameter has to be an easing function that returns the values in the range [0,1] [20].

Individual attributes of the bars and texts, such as *IDs* or *X* values, are changed using the `.attr()` method. The `swap_bars()` is also an asynchronous function called with the `await` keyword. This keyword can also be found inside the function itself and, together with the `.end()` method, it ensures that the animation of the bars and texts is not interrupted until it is finished. The `.end()` method is also part of the D3.js library. This method returns a `promise` and will be resolved when `transition()` ends.

Pseudocode colouring

The `pseudoColours()` function takes care of colouring individual lines of the pseudocode. The parameters of the function must be given in order, line number and colour. The function then always decolours all rows to their base colour and then recolours the specified rows with the specified colour.

Displaying variables

After loading the page or resetting the data, the variables are always shown as undefined. Their values are changed only when the `writeVar()` function is called with their exact parameter. The function then changes the content of the text element. Each of the sorting algorithms has this function adapted to its variables.

Colouring the chart bars

When going through the individual sorting steps, the individual bars need to be coloured to help the user better understand the entire process. For this purpose the `setColour()` function is used. It accepts two numbers and a string as parameters. The two numbers then represent the columns to be coloured, and the string represents that particular colour. If only one bar needs to be coloured, the function is called with the second parameter specified as *undefined*. If no colour is specified, the colour is set to red by default. The function itself then just selects a bar based on its *ID* and changes its colour using the `.attr()` method.

Displaying step description

The function for writing out the description of the steps `writeText()` takes the text string as the parameter and changes the content of the corresponding text element.

5.4.5 Time comparison data creation

Random and descending data are generated similarly to the visualisation part. However, the difference occurs for a nearly-sorted array and a few-unique arrays.

An almost sorted array is an array of length n where each element is at most k from its proper position. Such an array is basically generated again using the `Math.random()` function, this time multiplied by ten. Together with the `Math.floor()` function, which rounds the number down, numbers from 0 to 9 are generated. The sequence of numbers then ensures the addition of the i variable, which is part of the for loop, in which the given number is generated and simultaneously pushed into the array. Variable i goes from zero to the size of the array itself. The range of k is then ensured by subtracting the number 5.

A few-unique array is an array that contains a limited number of unique values. In this case, the limit is set to a third of the size of the entire array. First, these unique values are generated, again using the `Math.random()` function. These values are stored in a separate array. Subsequently, additional numbers are randomly generated within the size of the array of unique values. Each iteration of the second for

loop uses a random number to select a unique value from the array, which is then pushed into the final array.

5.4.6 Workers

The following paragraphs are based on information obtained from [6, 10]. By default, JavaScript is a single-threaded language. However, there is an option to create separate threads using Web workers. The worker created then runs separately from the JavaScript code that created it. Moreover, workers can interact with the JavaScript code that created them.

The worker object is created using the `new Worker()` keyword, which takes as a parameter a URL referring to the given worker's JavaScript file, e.g. `new Worker("js/AlgorithmWorkers.js")`.

The `.postMessage()` and the `.onmessage`. are used for two-way communication between the worker and the thread which created the worker. The `.postMessage()` method is used to send messages. The first parameter of this method is the data to be sent. The first parameter is mandatory, but the others are optional (options and transfer) [18]. The data sent can be any JavaScript object that allows the creation of a copy [16]. Receiving messages is ensured by the `.onmessage` event handler, which, when a message event occurs, executes the function assigned to it [17]. So if one party uses `.postMessage()` to send the message, the other party receives the message using `.onmessage`.

Parallel and sequential sorting

The application allows sorting algorithms to run parallelly and sequentially. After submitting the form, a function is called, which, based on the selection of the sorting type, calls the function to create the worker.

For parallel sorting, workers are created for all selected algorithms, and they work in parallel. Creating and then running all these workers at the same time is ensured using the `Promise.all()` method. This method takes an iterable, e.g. an array, as a parameter and returns a promise that is released only when all promises in the iterable are released [15]. If the stop button is used, all the still-running algorithms are stopped by terminating their workers.

In the case sequential sorting is selected, the function responsible for creating a worker is called repeatedly after the previous worker is terminated. If the user stops sorting using the stop button, the running algorithm and all algorithms that have not started yet are automatically interrupted.

Workers creation

The creation of workers is handled by the `algorithmWorkers()` function, which accepts the algorithm name, the array to be sorted, the array type and the sort type as parameters. This function creates workers inside a promise, and when the algorithm is finished, or the user stops sorting, the promise is resolved. The creation of the worker is done with the `new` keyword, and immediately after creation, the algorithm and array to be sorted are passed to the worker.

In addition to creating the worker, this function continues to handle the creation of HTML elements for the log section, the stop button function, and the messages sent by the worker.

All elements are held in an *elements* array and are created using the `Object.assign()` method. This method accepts a target object and one or more source objects as parameters. The method then returns the target object with the changed properties based on the source objects. These elements are then added to HTML using the classic method `.appendChild()`.

The `stopSorting()` function is assigned to the stop button. This function changes the log texts of any stopped algorithm and terminates the worker immediately using the `.terminate()` method. This method terminates the worker regardless of whether the worker itself has completed its code or not. Furthermore, the promise is resolved.

The function handling worker messages updates the recorded time text in the logging system. It only modifies the content of the <p> element using the data sent by the worker. There is also a *finished* variable among the data sent by the worker. If this variable is true, the event listener on the stop button is removed, the text of a particular algorithm in the logging system is marked with a colour indicating its completion, the worker itself is terminated, and the promise is resolved.

Workers code

The functions contained in the separate AlgorithmWorkers.js file are only individual algorithms and functions that handle messages. The function that handles receiving the message contains a switch that, based on the name of the algorithm, starts the given algorithm and passes it the array as a parameter. The function sending the message back only uses `.postMessage()`, which sends back the finished variable and the elapsed time. This function is called after the end of each algorithm or separately in the `messageToSend()` function. This function only ensures that messages are not unnecessarily sent too often, and thus it ensures that a message is sent up to every half a second. Individual algorithms have the same form as in the visualisation part of the application. The only difference is the addition of timing and messaging. In this case, the `performance.now()` method is used to measure the time of execution. The method returns a timestamp in milliseconds.

Chapter 6

User testing

This web application was created for educational purposes. Therefore, it was designed to be user-friendly even for those who have no experience with sorting algorithms. Three people were asked to test this application. All of these people have different experiences in the field of computer science, therefore, their expectations for the application itself were also different.

6.1 Testers background

Tester A

Tester A is a person without any education in the field of information technology. Nevertheless, he pursues this field as a hobby. His knowledge is the result of self-study, mostly using the internet, and therefore, his comments can be very important. Furthermore, this tester might be most likely to encounter other web applications similar to this one.

Tester B

Tester B is a person without any background in information technology. She will thus serve as an example of a user who has never encountered sorting algorithms and has no knowledge about them. Therefore, her knowledge can be used for future application improvements. These improvements would then allow easier understanding of the sorting algorithms for other users without any experience.

Tester C

Tester C is a person who has already completed her bachelor's degree in the field of information technology and thus completed the subject for which this web application was created. Her experience should be very beneficial, as she knows first-hand which sorting algorithms can be difficult for students to understand.

6.2 Testing process

All testers were provided with the web application, including the user manual that is part of the application itself. Testers were also given enough time to study all parts of the application. The application and the testing process were conducted in the Czech language, as the application is aimed at Czech students.

6.3 Briefing

Before starting the actual testing, the testers were asked several questions and given a few instructions:

1. Have you ever come across the term sorting algorithms? If so, how do you rate your knowledge in this field? Have you ever used a web application that would allow them to be visualised?
2. Use only this web application as a source during testing.
3. Write down any comments on areas of the application that you find unclear or insufficiently explained.
4. Spend time on each part of the application so that you can tell how useful this app is.

6.4 Debriefing

After the testers studied the application in detail, they were asked a few additional questions:

1. If you had no knowledge of sorting algorithms, was this application able to present information about the field in a simple enough form so that you could easily understand it?
2. If you already had some knowledge, did the application bring you any new knowledge? Alternatively, do you think it would have been easier for you in the past to learn about sorting algorithms from this application or from another source?
3. If you have used other visualisation applications, how would you rate them compared to this application?
4. Was the application sufficiently understandable and intuitive for you?
5. Are there any flaws that made working with the application unpleasant for you?
6. Do you have any other comments on the application?

6.5 Results

Tester A insights

Tester A mainly uses videos and not web applications to learn about sorting algorithms. However, from the already existing web applications mentioned in the competitive analysis chapter, he knew the visualgo.net application.

The application itself brought him several new insights, especially regarding algorithms that he himself had not explored sufficiently. On the basis of his comments, several modifications have been made for better navigation in the application, such as a notification of the pop-up menu. As the first tester, he discovered several minor bugs that could be fixed immediately.

He considered the language to be the biggest drawback of the application. Since he is used to looking for information primarily by himself, he searches for it in most cases in English. Therefore, the names of the individual algorithms were an obstacle for him. The application is intended for students learning in the Czech language, therefore, English versions of all algorithms have been added to their description. This has been done in case students want to look up more information about a specific algorithm.

Overall, apart from a few objections, the application was clear for him. He considers the execution of the animations and the actual stepping of the algorithm to be done very well. He considered the missing step time bar, which is available in the visualgo.net application, to be a disadvantage of this application. Unfortunately, this feature would be too demanding to implement at this point in the project, considering the remaining time to finish the project. On the other hand, the tester greatly appreciates the time comparison of individual algorithms.

Tester B insights

Tester B roughly knows what an algorithm is, but she has never directly encountered sorting algorithms. Therefore, she does not know of any other applications similar to this one. The application seemed simple to her, but she came across several concepts that she did not know before, e.g., recursion or how *for loops* work. However, these terms are commonly used in the field of information technology, and students should encounter them during their studies.

Since she has no programming experience, pseudocode part was almost useless to her. However, due to the variables and the description of the step, she was able to derive the individual steps. She also mentioned that for most algorithms, a simple bar chart animation was sufficient to understand how the algorithm works.

As this tester has not come across any other applications that allow the visualisation of sorting algorithms, it is difficult for her to find the advantages and disadvantages of this application. However, she indicated that the application was user-friendly, and she could imagine that it would be useful for students. During the tests, she did not come across any bugs, but, as an expert in the Czech language, she was able to point out several shortcomings in the algorithm descriptions and the user manual.

Tester C insights

Tester C has already encountered a few visualisation applications. She thinks that this application explains the sorting algorithms clearly and has everything needed for better understanding. All technical terms were clear to her, which is to be expected from a person with education in the field of information technology.

As such, the application did not bring her too much new knowledge, since she passed the *Basics of Algorithmization* course and had to acquire all the knowledge about sorting algorithms. However, she acknowledged that the information available in this application could be a beneficial resource for future students, as they will have everything in one place.

She considers the time comparison of algorithms as a big advantage of this application, which can more realistically present students with the time requirements of individual algorithms on arrays of different sizes and initial conditions. She then mentioned the speed of the animation and text changes in play mode and the strange behaviour of the slider as a disadvantage. Since the play mode is mainly intended for watching the animation only, its speed has been preserved. As for the slider, its behaviour has been slightly modified to make it more intuitive for the user.

Overall, tester C was very satisfied with the application and thinks that, compared to other visualisation web applications, this one is very nicely done and is more suitable for the very course in the *Basics of Algorithmization*.

6.5.1 Summary

After the testing was completed, several changes were made. The small animated notification was added on the main page and on the user manual page to alert the pop-up menu in the upper left corner.

English names of all the sorting algorithms have been added to the description. Furthermore, for example, the behaviour of the slider when setting the number of bars in the visualisation part was modified. However, some changes, such as the step time bar, were too demanding and would have to make too much intervention in the application.

In general, it can be said that the testing was successful, and all the testers were satisfied with the application. It can also be considered a great success that although some testers did not have education in the field of computer science, they were still able to fully understand almost everything. Due to their observations, most of the shortcomings were corrected, and the application could then become more intuitive. It is necessary to emphasize that all the testers indicated that they believe that this application will help future students easily understand sorting algorithms.

6.6 User manual

The user manual is available as part of the web application itself. As mentioned in the previous chapter, the manual is written in the Czech language, as the primary users of this application are Czech students of the Faculty of Nuclear Sciences and Physical Engineering (FNSPE) of the Czech Technical University in Prague (CTU), or other Czech universities or secondary schools.

Conclusion

The project was aimed at the creation of an educational tool for the *Basics of Algorithmization* course. The application is focused on sorting algorithms, especially their visualisation and time comparison. Its source code can be downloaded from GitHub[1]. The web application can also be visited on website[2].

As part of the project, an analysis of existing web applications was carried out. Comparing the strengths and weaknesses of individual applications leads to the inclusion of all the important features that an application should have. This analysis led to the accurate determination of the requirements for the newly created application. It is important to note that the resulting web application fulfils the initial task and, in some respects, even surpasses it. An example can be that, as part of the visualisation, the application clearly shows the user which parts of the code were executed in each step.

The technologies were chosen with the aim of reducing the overly complex portion of the project. For example, the D3.js JavaScript library was a suitable choice for creating and handling the bar chart in the visualisation part of the application as it enabled the simplicity of the entire work process. In addition, all important functionalities are described in detail in this work.

Due to user testing, most of the shortcomings of the application, which can be easily overlooked by the developer, have been fine-tuned. The application is stable, and no problems with breaking functionality have been detected. This allows the application to be included in the educational process as the useful material.

This project covers the entire development process and was a useful experience in web application development. The resulting application is a sufficient tool that can be used in the educational process and help the next generation of programmers in their understanding of sorting algorithms.

In the future, user data may be analyzed to reveal additional requests and feature enhancements that could improve the quality of the application. An improvement could be the addition of the ability to go backward in the visual process or the inclusion of additional languages to extend usability.

[1] `https://github.com/Gatorixx/Bachelors-project`

[2] `https://baka.darzy.art/index`

Bibliography

[1] *Sneha M* [online]. [cit. 2023-06-18]. Available at: `https://devfolio.co/@sneha24102000`.

[2] *Sorting Algorithms Animations | Toptal®* [online]. [cit. 2023-06-18]. Available at: `https://www.toptal.com/developers/sorting-algorithms`.

[3] *Sorting (Bubble, Selection, Insertion, Merge, Quick, Counting, Radix) - VisuAlgo* [online]. [cit. 2023-06-14]. Available at: `https://visualgo.net/en/sorting`.

[4] Cormen, T. H. *Introduction to algorithms*. 3rd ed. London: The MIT Press, 2009. ISBN 978-0-262-53305-8.

[5] Eck, D. *Exploring Sorting Algorithms* [online]. revised 2017-04-01 [cit. 2023-04-12]. Available at: `https://math.hws.edu/eck/js/sorting/xSortLab.html`.

[6] Flanagan, D. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*. 7th ed. Sebastopol, California: O'Reilly Media, 2020. ISBN 978-1491952023.

[7] Forsythe, G. E. Algorithms. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jun 1964, vol. 7, no. 6, p. 347–349. DOI: 10.1145/512274.512284. ISSN 0001-0782. Available at: `https://doi.org/10.1145/512274.512284`.

[8] Galles, D. *Comparison Sorting Visualization* [online]. [cit. 2023-06-15]. Available at: `https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html`.

[9] Galles, D. *Data Structure Visualization* [online]. [cit. 2023-06-15]. Available at: `https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`.

[10] Hunter, T. II. and English, B. *Multithreaded Javascript: Concurrency Beyond the Event Loop*. 1st ed. Sebastopol, California: O'Reilly Media, 2021. ISBN 978-1098104436.

[11] Knuth, D. E. *The art of computer programming: Fundamental algorithms*. 3rd ed. Upper Saddle River: Addison-Wesley, 1997. ISBN 0-201-89683-4.

[12] Lerdorf, R., Tatroe, K. and MacIntyre, P. *Programming Php*. 2nd ed. " O'Reilly Media, Inc.", 2006. ISBN 9780596006815.

[13] Mareš, M. and Valla, T. *Průvodce labyrintem algoritmů*. 2nd ed. Praha: CZ.NIC, z.s.p.o, 2022. ISBN 978-80-88168-66-9.

[14] Mozilla Foundation. *Math.random() - JavaScript | MDN* [online]. Mozilla. revised 2023-08-25 [cit. 2023-11-18]. Available at: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random`.

[15] Mozilla Foundation. *Promise.all() - JavaScript | MDN* [online]. Mozilla. 2023-06-25 [cit. 2023-11-22]. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all.

[16] Mozilla Foundation. *The structured clone algorithm - Web APIs | MDN* [online]. Mozilla. revised 2023-11-07 [cit. 2023-11-22]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[17] Mozilla Foundation. *Worker: message event - Web APIs | MDN* [online]. Mozilla. revised 2023-04-08 [cit. 2023-11-22]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Worker/message_event.

[18] Mozilla Foundation. *Worker: postMessage() method - Web APIs | MDN* [online]. Mozilla. revised 2023-09-21 [cit. 2023-11-22]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Worker/postMessage.

[19] Neckář, J. *Algoritmus* [online]. 2016 [cit. 2023-06-17]. Available at: https://www.algoritmy.net/.

[20] Observable. *D3-ease | D3 by Observable* [online]. [cit. 2023-11-28]. Available at: https://d3js.org/d3-ease.

[21] Observable. *D3-transition | D3 by Observable* [online]. [cit. 2023-11-28]. Available at: https://d3js.org/d3-transition.

[22] Observable. *Timing | D3 by Observable* [online]. [cit. 2023-11-28]. Available at: https://d3js.org/d3-transition/timing.

[23] Observable. *What is D3? | D3 by Observable* [online]. [cit. 2023-09-20]. Available at: https://d3js.org/what-is-d3.

[24] Palsberg, J. and Su, Z. Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009, Proceedings. In:. Springer Berlin Heidelberg, 2009, p. 237. Lecture Notes in Computer Science. ISBN 9783642032370.

[25] Scanu, D., Tovoli, T. and Larsson, O. E. *Sort Visualizer* [online]. [cit. 2023-06-14]. Available at: https://www.sortvisualizer.com/.

[26] Sneha, M. *Sorting Visualizer* [online]. [cit. 2023-06-18]. Available at: https://xenodochial-darwin-bd74c7.netlify.app/.

[27] Tvrdlík, P. *Algoritmy řazení* [online]. 2014 [cit. 2019-05-06]. Available at: https://courses.fit.cvut.cz/BI-EFA/media/lectures/bi-efa2014prednaska3-sort-anim.pdf.

[28] Virius, M. *Základy algoritmizace*. 2nd ed. Praha: České vysoké učení technické v Praze, 2008. ISBN 9788001040034;8001040038.

[29] Wirth, N. *Algoritmy a štruktúry údajov*. 2nd ed. Bratislava: Alfa, 1989. ISBN 8005001533, 9788005001536.

Appendix

A Code examples

Bar chart creation

```
function createSVG()
{
  let width = document.getElementById("chart").offsetWidth;
  let height = document.getElementById("chart").offsetHeight;

  // ...   (Special conditions for merge sort and binary insertion sort)
  //ordinal scale
  X = d3.scaleBand()
          .domain(d3.range(data.length))  //input domain
          .range([0,width])               //
          .paddingInner(0.05);            //5% padding mezi sloupci

  Y = d3.scaleLinear()
    .domain([0, d3.max(data)])
    .range([20, height]);

  svg = d3.select("#chart")               //přidání svg
  .append("svg")                          //grafický objekt svg
  .attr("height", height)
  .attr("width", width)


  svg.selectAll("rect")                 //přidání barů
      .data(data)
      .enter()
      .append("rect")
      .attr("x", function(d,i){
        return X(i);
      })
      .attr("y", function(d){
        return height - Y(d);
      })
      .attr("width", X.bandwidth())
      .attr("height", function(d) {
        return Y(d);
      })
      .attr("fill" ,"teal")
      .attr("id",function(d,i){
        return "bar" + i;
      });

  let text = svg.selectAll("text")        //přidání textů barů
  text
    .data(data)
    .enter()
    .append("text")
    .text(function(d) {
      return d;
    })
    .attr("x", function(d,i){
      return X(i) + X.bandwidth() / 2;
    })
    .attr("y", function(d){
      return height - Y(d) + 16;
    })
    .attr("id",function(d,i){
      return "text" + i;
    });
}
```

Handling user input

```
async function waitForUserInput()
{
    return new Promise(resolve =>
    {
        let stepButton = document.getElementById("step"); //Proměnné pro tlačítka
        let playButton = document.getElementById("play");
        let slider = document.getElementById("slider");
        let randomButton = document.getElementById("random");
        let descendingButton = document.getElementById("descending");

        stepButton.addEventListener("click", stepHandle);
        playButton.addEventListener("click", playHandle);
        slider.addEventListener("input",  sliderHandle);
        randomButton.addEventListener("click", randomButtonHandle);
        descendingButton.addEventListener("click", descendingButtonHandle);

        if(!play_flag)
        {
            buttons();
            slider.disabled = false;
        }

        const rightArrow = event =>
        {
            if (event.key === "ArrowRight")
            {
                stepButton.click();
            }
        };
```

```
if(play_flag)
{
    cleanHandlers();
    resolve();
}
else
{
    document.addEventListener("keydown", rightArrow);
}

function stepHandle()
{

    buttons("step");
    slider.disabled = true;
    cleanHandlers();
    resolve();
}

function playHandle()
{
    play();
    cleanHandlers();
    resolve();
}

function sliderHandle()
{
    dataChange_flag = true;
    play_flag = false;
    cleanHandlers();
    resolve();
}

function randomButtonHandle()
{
    dataChange_flag = true;
    play_flag = false;
    cleanHandlers();
    resolve();
}

function descendingButtonHandle()
{
    dataChange_flag = true;
    play_flag = false;
    cleanHandlers();
    resolve();
}

function cleanHandlers()
```

```
        {
            document.removeEventListener("keydown", rightArrow);

            stepButton.removeEventListener("click", stepHandle);
            playButton.removeEventListener("click", playHandle);

            slider.removeEventListener("input",  sliderHandle);
            randomButton.removeEventListener("click", randomButtonHandle);
            descendingButton.removeEventListener("click", descendingButtonHandle);

        }
    });
}
```

A demonstration of bubble sort and its embedded functions

```
async function bubblesort()
{
    dataChange_flag = false; //Data zatím nejsou změněna
    sort_flag = true;  //Začalo řazení

    for(i = 0; i < data.length - 1; i++)
    {
        if(i == 0) {pseudoColours("0","LR","1","LR","2","LR","3","LR");}
        else {pseudoColours("0","DR","1","LR","2","LR","3","LR");}
        for (j = 0; j < data.length - i - 1; j++)
        {
            writeText("Vybereme dva sloupce k porovnání.");
            writeVar();
            if(!(j == 0))pseudoColours("0","DR","1","DR","2","LR","3","LR");
            await setColour(j, j + 1);
            await waitForUserInput(); if(dataChange_flag) return;
            if (data[j] > data[j + 1])
            {
                writeText("Hodnota ... Proto sloupce jsme vyměnili.");
                writeVar();
                pseudoColours("0","DR","1","DR","2","DR","3","DR","4","LR")

                await swap_bars(j, j + 1);
                [data[j], data[j + 1]] = [data[j + 1], data[j]];

                await waitForUserInput(); if(dataChange_flag) return;
            }
            else
            {
                pseudoColours("0","DR","1","DR","2","DR","3","LR")
                writeText("Hodnota ... Sloupce zůstávají na svém místě.");
                writeVar();
                await waitForUserInput(); if(dataChange_flag) return;
            }
        }
        if(i <= data.length)
        {
            writeText("Sloupec ... označíme jako seřazený.");
            writeVar();
            pseudoColours("0","DR","1","DR","5","LR");
            svg.selectAll("rect[fill='red']").attr("fill", "teal");
            svg.select("#bar" + String(data.length - i - 1)).attr("fill","orange");
            if(play_flag) {await new Promise(r => setTimeout(r, 800));}
            await waitForUserInput(); if(dataChange_flag) return;
        }
    }
    writeText("Graf je nyní seřazen.");
    writeVar();
    pseudoColours("6","LR","7","LR");
    play_flag = false;
    buttons("finished");
    svg.selectAll("rect").attr("fill", "orange");

    let slider = document.getElementById("slider");
    slider.disabled = false;
}
```

Workers creation

```
function algorithmWorkers(algorithm, array, array_selection, sort_type)
{
    return new Promise((resolve, reject) =>
    {
        const worker = new Worker("javaScript/AlgorithmWorkers.js"); //Vytvoření workera
        worker.postMessage({algorithm, array}); //pošlu jméno algoritmu a pole workeru

        document.getElementById("stopButton").addEventListener("click", stopSorting);

        // ... (Creation of elements for the log system)

        function stopSorting() //Funkce pro zrušení
        {
            // ... (Changing elements and their colour, and removing listener)

            stopFlag = true; //Flag pro sekvenční spouštění
            worker.terminate(); //Ukončení workera
            resolve(); //Uvolnění promise
        };

        function workerMessage(msg) //Zpráva od workera
        {
            var data = msg.data;
            elements[4].innerHTML =
                data[1].toFixed(5).toString().padStart(5, " ") + " sekund"; //Výpis času

            if(data[0])
            {
                // ... (Changing the colour of elements and removing listener)
                worker.terminate(); //Ukončení workera
                resolve();
            }
        }


        worker.onmessage = workerMessage;  //Zpráva workera => funkce workerMessage

        worker.onerror = function(error) //v případě chyby
        {
            worker.terminate();
            reject(error);
        };
    });
}
```

Handling messages in worker

```
function sendMessage() //poslání zprávy zpět se stavem sortu a časem
{
    postMessage({finished, elapsedTime});
}

function messageToSend()
{
    endTime = performance.now();
    let lastMessageTime = (endTime - sendTime);

    if(lastMessageTime > 500) //posílání zprávy jen každou půl sekundu
    {
        sendTime = performance.now();
        elapsedTime = (endTime - startTime) / 1000;
        sendMessage();
    }
}
```