A Visual Tool to Study Sorting Algorithms and Their Complexity

Tanyo Kostadinov, Ivon Nikolova, Radoslav Radev, Angel Terziev, and Lasko Laskov(✉)

Informatics Department, New Bulgarian University, Sofia, Bulgaria
ivonvesko@abv.bg, llaskov@nbu.bg

Abstract. Sorting algorithms are a well-known part of the curriculum in programming courses in the academia. They are taught not only because their numerous applications in practice, but also because they are a good and a comprehensive introduction to the topic of computer algorithms. However, the *asymptotic notation* used to describe algorithm complexity is not intuitive for beginners. A visual tool that demonstrates both the algorithm's steps and its time complexity makes the abstract notion *asymptotic notation* more intuitive, and can improve the learning curve of the students.

Keywords: Algorithms and data structures · Sorting algorithms · Informatics education

1 Introduction

Sorting algorithms have been a subject of extensive study, and an indivisible part of the academia courses of compute programming, and algorithms and data structures [13,19]. Actually, the history of studies of sorting algorithms can be traced back to the beginning of computing around the middle of the 20th century (see for example works [3,6–9]).

Of course, sorting algorithms are an important step in many other methods and algorithms, and different examples can be given on different level of computing and computer science. A basic example is the binary search algorithm that requires data to be sorted [15]. Sorting and ordering of data is a part of complex systems such as relational database management systems [18]. Sorting also is applied in image processing and computer vision methods, as in the case of the implementation of the median filtering [11].

Besides their practical importance, sorting algorithms are often part of a beginner's course in computer algorithms in the academia, since they give a good introduction to the complex topic of computer algorithms as a whole. The objective of sorting is easily defined: implement an algorithm that orders a given sequence in ascending or descending order using a predefined relation among its elements; however, different algorithms may have a significant complexity of

T. Zlateva and R. Goleva (Eds.): CSECS 2022, LNICST 450, pp. 183–195, 2022.
https://doi.org/10.1007/978-3-031-17292-2_15

their definition and may require different skill level in order to be understood. A problem that is easily formulated, and methods of different complexity that are applicable in its solution, open the possibility for creation of a *system of tasks* that can aid the notion formation during the course (for notion formation through a system of tasks see [2] and [14]).

Sorting algorithms are a good starting point to computer algorithms topic, but also they allow the introduction of the complicated notion *algorithm complexity* (see [4]) in a way that is relatively fluent and intelligible for the students in the first and second year of their studies. The understanding of the *asymptotic notation* that is used to denote algorithm complexity, frequently is an obstacle for students in the introductory course of computer algorithms, mainly because of their level of knowledge of calculus and analysis, where this notion comes from. For that reason, usually the notion algorithm complexity is introduced by measuring the execution time of the algorithm implementation running on a concrete computer system, as for example in [10], which helps the intuitive understanding, even though it cannot substitute the analytic complexity derivation.

The above motivates us to develop a software that is able to provide learners with two different visual representations of the sorting algorithms: (i) step-by-step visualization of the algorithm, and (ii) visual representation of the time complexity of the algorithm, compared with the same information for other algorithms in the same package. The graphical user interface of the software enables the experiment with a number of standard sorting algorithms, applied sequences of different data types which allows students to develop a visual concept of both algorithms themselves, and their computational complexity. This visual concept significantly improves the perception of the analytical explanation of the notion asymptotic notation.

This paper is organized as follows. In Sect. 2 we give a brief survey of the algorithms we have currently incorporated in our project grouped by their average-case complexity: square complexity algorithms (Sect. 2.1), and $n \log n$ complexity algorithms (Sect. 2.2). In Sect. 3 we present some details of visualization implementation. Finally, in Sect. 4 we present our conclusions and future work on the project.

2 Some Frequently Taught Sorting Algorithms

In this section we present some of the frequently taught sorting algorithms in an introduction course in computer algorithms, which we have implemented in our visualisation software. We separate the algorithms in two parts: methods that can be classified as square complexity algorithms, and those that can be classified as $n \log n$ complexity algorithms, according to their average-case execution complexity.

2.1 Average-Case Square Complexity Algorithms

Bubble Sort. Bubble sort is a classical square complexity algorithm, that used to be one of the first methods taught in academia programming courses [13,19]. It is

relatively straightforward both to understand and implement (see Algorithm 1), and yet many studies prove that is inefficient, even for a "slow" sorting algorithm (see [1]).

Bubble sort is invented around the middle of the previous century, and different names were used to refer to it (exchange sorting, or sorting by exchange) [3,6,8]. The first occurrence of the term *bubble sort* can be traced back to 1962 [12], and this is the name with which this algorithm gained popularity.

Algorithm 1. Basic version of the bubble sort, applied on a sequence a.

```
function BubbleSort(a)
    n ← Size(a)
    for i ← 0 to n − 2 do
        for j ← 0 to n − j − 2 do
            if a[j] > a[j + 1] then
                Swap(a[j], a[j + 1])
            end if
        end for
    end for
end function
```

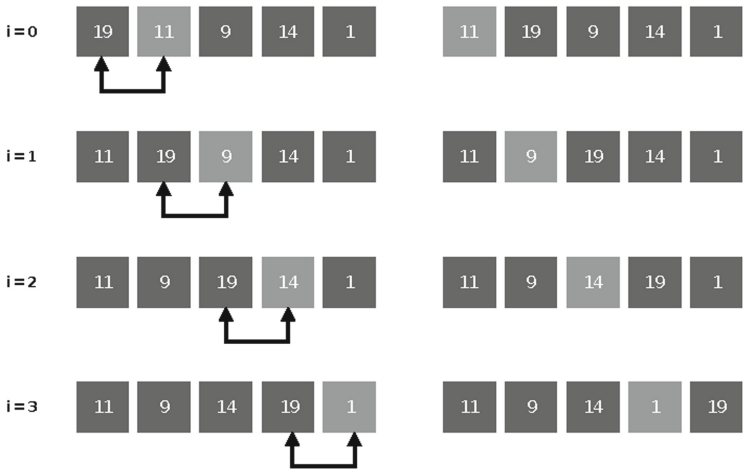

Fig. 1. Execution of the bubble sort algorithm for its first four iterations.

The algorithm works by repeatedly swapping adjacent positions that are not ordered (Algorithm 1). If the first element is greater than the second, it will swap these elements, it will then continue with second and third until the end of the sequence is reached. On each run, the method pushes the largest element to the

end of the sequence (see Fig. 1). Also, different variations of bubble sort exist, yet it is easy to be proven that the gained efficiency is not enough to match the performance of selection sort and insertion sort, described further.

For a sequence of n elements, each pass requires $n - i$ comparisons. Then the complexity of the algorithm can be expressed by

$$g(n) = \sum_{i=1}^{n}(n - i) = \frac{n(n-1)}{2}, \tag{1}$$

where $g(n)$ denotes the number of operations performed by the algorithm. Using big-O notation, the complexity of the bubble sort in the general case is $g(n) \in O(n^2)$, which is also its worst case performance in the case in which the input is sorted in reversed order. Best case may occur, if the sequence is already sorted, and the algorithm may stop after the first pass without a swap (with a slight modification of Algorithm 1).

Selection Sort. Another simple square complexity algorithm, that has been known from the same period as the bubble sort, is the *selection sort* algorithm [6]. Besides its simplicity, it outmatches the bubble sort efficiency, and in many academia courses now-a-days it is preferred as the first example for a sorting method [10].

Algorithm 2. Selection sort, applied on a sequence a. The function MinPos() finds the index of the minimum element in a, starting from position i.

```
function SelectionSort(a)
    n ← Size(a)
    for i ← 0 to n − 2 do
        j ← MinPos(a, i)
        if j ≠ i then
            Swap(a[i], a[j])
        end if
    end for
end function
```

On each step of the main loop of the algorithm (see Algorithm 2) the method selects the minimum element from the current unsorted sub-sequence. It swaps the discovered minimum element with the one on the first position of the sub-sequence (see Fig. 2). The main loop of the algorithm is repeated until the unsorted sequence is two elements big (the index i reaches the element $n - 1$).

On the first iteration, selection sort performs n visits to find the minimum element, and two visits to swap, totally $n + 2$ visits. On the second visit it performs $n + 2 + (n - 1) + 2$ visits, and since all swaps are $(n - 1)$, the total number of operations are (see also [10]):

$$g(n) = n + 2 + (n - 1) + 2 + \ldots + 2 + 2. \tag{2}$$

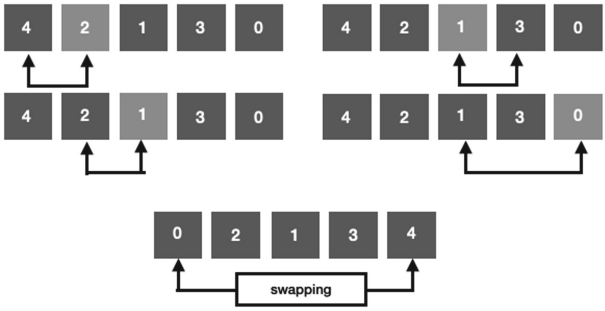

Fig. 2. The first swap operation performed by the selection sort algorithm

Since $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$:

$$g(n) = \frac{n(n+1)}{2} - 1 + (n-1) \cdot 2 = \frac{1}{2}n^2 + \frac{5}{2}n - 3, \tag{3}$$

and we get that $g(n) \in O(n^2)$.

Insertion Sort. The last square complexity algorithm that we will discuss here is the *insertion sort*. According to Knuth (see [13]), a version of this algorithm called *binary insertion* is mentioned in 1946 by John Mauchly in a legacy publication dedicated on computer sorting.

Algorithm 3. Insert an element x into sorted sequence a with length n.

```
function INSERT(a, n, x)
    i ← n − 1
    while i ≥ 0 and a[i] > x do
        a[i + 1] = a[i]
        i ← i − 1
    end while
end function
```

The algorithm is based on a routine (see Algorithm 3) that inserts an entry into a sorted sequence in its correct position. Insertion sort algorithm simply calls the routine INSERT() for all elements of the input sequence consecutively for a subsequence with length 1, 2 until $n-1$, where n is the length of a (see Algorithm 4) (Fig. 3).

In the general case we can assume that half of the elements of a are less than the current element $a[i]$ which implies $i/2$ comparisons on average. Then the total number of comparisons performed by the algorithm are (see also [13]):

$$g(n) = \frac{\sum_{i=1}^{n} i}{2} = \frac{n(n+1)}{4}. \tag{4}$$

Algorithm 4. Insertion sort based on the INSERT() routine.

```
function InsertSort(a)
    n ← Size(a)
    for i ← 1 to n − 1 do
        Insert(a, i, a[i])
    end for
end function
```

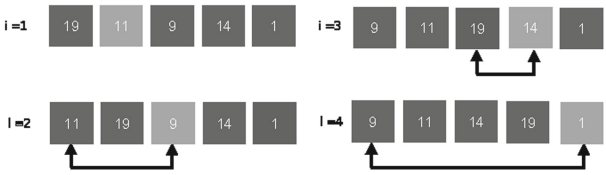

Fig. 3. Four iterations of the insertion sort algorithm.

Again, using asymptotic notation, the complexity is $O(n^2)$, however comparing equations (1), (3) and (4), we see that insertion sort performs better than selection sort, and much better than bubble sort in the general case. Also, different improvements of the algorithm can be applied, which can speed up its average performance (see [13] and [16]). For example, if binary search is used in the inserting routine (Algorithm 3), the complexity of the search of the insert location is reduced to $O(\log n)$. The latter method, called *binary insertion*, also does not solve the problem that the elements of the array must be moved to make space for the element to be inserted.

2.2 Average-Case $n \log n$ Complexity Algorithms

Merge Sort. Merge sort is a classical sorting algorithm that is based on a divide-and-conquer strategy. Maybe the first description of this algorithm can be found in the work of Goldstine and von Neumann [7], where merging procedure is referred to as "meshing". In this work a full explanation and analysis of *bottom-up* version of merge sort that is based on iteration, is given.

Algorithm 5. Recursive top-down version of the merge sort algorithm.

```
function MergeSort(a, f, t)
    if f = t then
        return
    end if
    m ← (f + t)/2
    MergeSort(a, f, m)
    MergeSort(a, m + 1, t)
    Merge(a, f, m, t)
end function
```

Here we will focus on the more popular contemporary *top-down* approach that uses recursion to be implemented. Often, recursive solutions provide clearer approach to complex problems (see for example [10]), and merge sort implementation is not an exception.

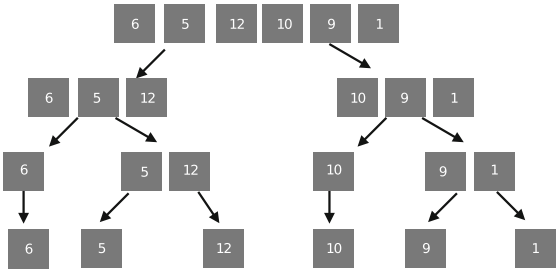

Fig. 4. Merge sort divides by half the sequence recursively until a sequence of length 0 is reached.

In the main function of the algorithm (Algorithm 5), the input sequence is divided by half, and each half is sorted recursively. The trivial case of the recursion is when the divided sequence is of length 0. The sequence that is composed by a single element can be considered sorted by definition. This is the *divide part* of the divide-and-conquer algorithm. An example of this process is given on Fig. 4.

After reaching the bottom of recursion, in the backwards function calls of the recursion, the algorithm performs its *conquer* part using the function MERGE() (see Algorithm 6) that merges two sorted sub-sequences into a resulting sorted sequence. An example of the merging process applied in the backward functions calls of the recursion is given on Fig. 5.

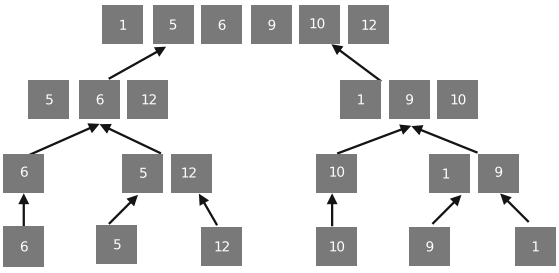

Fig. 5. Backward recursive calls of the merge sort algorithm consecutively merge the sub-sequences using the routine in Algorithm 6.

The complexity of the MERGE() procedure can be evaluated to three visits per single iteration, and hence $3n$ visits totally n iterations. To copy the sorted

Algorithm 6. Merge two sorted sub-sequences with ranges $[f, m]$ and $[m+1, t]$ stored in the same array a.

```
function MERGE(a, f, m, t)
    b is a temporary array
    i ← f, j ← m + 1, k ← 0
    while i ≤ m and j ≤ t do
        if a[i] < a[j] then
            b[k] ← a[i], i ← i + 1
        else
            b[k] ← a[j], j ← j + 1
        end if
        k ← k + 1
    end while
    while i ≤ m do
        b[k] ← a[i]
        i ← i + 1, k ← k + 1
    end while
    while j ≤ t do
        b[k] ← a[j]
        j ← j + 1, k ← k + 1
    end while
    COPY(a, b)
end function
```

sequence from the temporary array the algorithm spends $2n$ visits, and the whole procedure has complexity $5n$ visits.

The complexity of the recursive function MERGESORT() for $n = 2^m$ elements can be evaluated recursively as well (see also [10]):

$$g(n) = g\left(\frac{n}{2}\right) + g\left(\frac{n}{2}\right) + 5n = 2\,g\left(\frac{n}{2}\right) + 5n \tag{5}$$

and

$$g\left(\frac{n}{2}\right) = 2\,g\left(\frac{n}{4}\right) + 5\frac{n}{2} \tag{6}$$

then we get:

$$g(n) = 4\,g\left(\frac{n}{4}\right) + 10n. \tag{7}$$

Finally the above expression for $n = 2^m$:

$$g\,(n) = 2^m g\left(\frac{n}{2^m}\right) + 5nm = ng\,(1) + 5nm = n + 5n\log_2 n. \tag{8}$$

Of course, in asymptotic notation the complexity of the merge sort algorithm is written $O(n \log n)$.

Quicksort. Another algorithm that is based on the divide-and-conquer paradigm just like previously discussed merge sort, is the *quicksort* algorithm. It is developed by the British computer scientist Tony Hoare, and published in 1961 [9]. The algorithm originates from the need of fast sorting algorithm during his work

on machine translation project for the National Physical Laboratory while he was a visiting student at Moscow State University [17]. Since then the algorithm undergoes various modifications and improvements to become the most notable sorting algorithm, applicable on long input sequences.

Algorithm 7. Main function of the quicksort algorithm. The recursive function gets the input sequence a with the range in the closed interval $[l, r]$.

```
function QUICKSORT(a, l, r)
    if r ≤ l then
        return
    end if
    m ← PARTITION(a, l, r)
    QUICKSORT(a, l, m)
    QUICKSORT(a, m + 1, r)
end function
```

The divide part of the algorithm splits the input sequence recursively in the closed range $[l, r]$ (Algorithm 7). If the range contains one or less elements, the trivial case solution is reached, and the function returns. Otherwise, a function PARTITION() (Algorithm 8) finds the index to split the sequence in the given range. Also, the function PARTITION() ensures that there are no elements leftwards the pivot p at the returned index that are grater than it, and there are no elements rightwards p that are less than it. Then the two parts of the sequence (the one that is less than p, and the one that is greater than p) are partitioned recursively.

Algorithm 8. Basic version of the partitioning routine of the quicksort algorithm which selects as a pivot the first element in the given range.

```
function PARTITION(a, l, r)
    p ← a[l]                                   ▷ Pivot value.
    i ← l − 1, j ← r + 1
    while i < j do
        do
            i ← i + 1
        while a[i] < p
        do
            j ← j − 1
        while a[j] > p
        if i ¡ j then
            SWAP(a[i], a[j])
        end if
    end while
    return j
end function
```

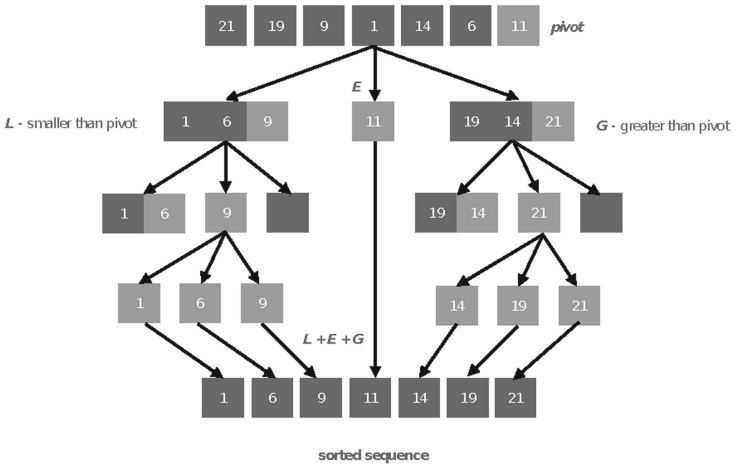

Fig. 6. Partitioning with the last element selected as the pivot.

There are different ways to select the pivot value p in the routine PARTITION(). In the basic version, the first or the last element is selected (on Fig. 6 we show an example with last element as the pivot, while Algorithm 8 is an example of the first element as the pivot). In the *worst case* this could be the maximum or the minimum element in the range, which will lead to square complexity if it is repeated on each recursive call. An improved version of the algorithm uses a random pivot which ensures that the partitioning of the sequence in the range will be roughly equal. In balanced partitioning on each recursive call, the *average-case complexity* of the algorithm can be evaluated in analogy to (5) using the recurrence relation (see [4]):

$$g(n) = O(n) + 2g(\frac{n}{2}), \tag{9}$$

where single call to the QUICKSORT() has complexity $O(n)$ plus the complexity of the two recursive calls. From the *Master Theorem* ([4], p. 94) we know this leads to algorithm average-case complexity $O(n \log n)$.

3 Visual Application Implementation

The software we present here is implemented using the programming language C++ and Qt platform (see [5]) which makes it fully portable on each operating system that is supported by Qt. It is composed by the following main sections:

1. Implementation of the sorting algorithms described in the previous section.
2. Modified versions of the algorithm functions to allow step-by-step algorithm visualization.
3. Timers and a graphical plot of the execution time of each algorithm (see Fig. 7).

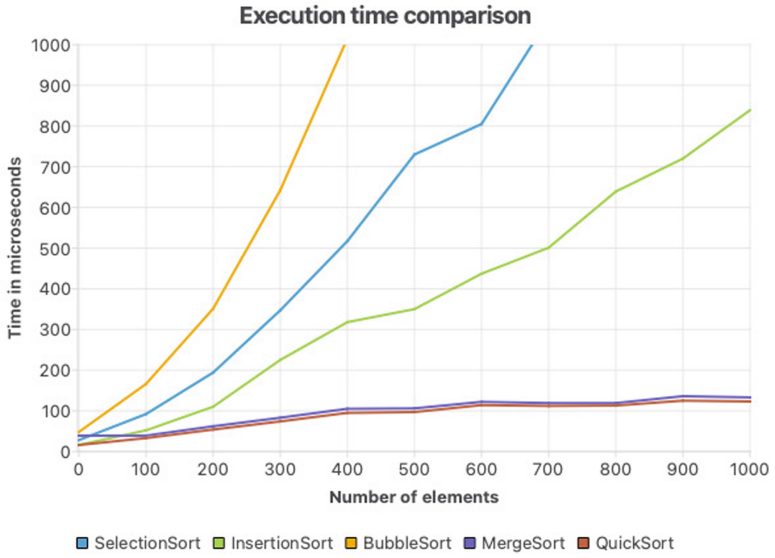

Fig. 7. Plot of the execution time of the examined algorithms produced by our software.

The application allows user to select whether algorithms to be tested on randomly generated sequence, or to load a predefined sequence from a file. Also, the user can select the data type of the input sequences. These two options allow to test sorting algorithms with a broad variety of input data, and also to load data sequences that are extracted from other problems. The latter can demonstrate the efficiency of a given algorithm in the context of a given practical situation.

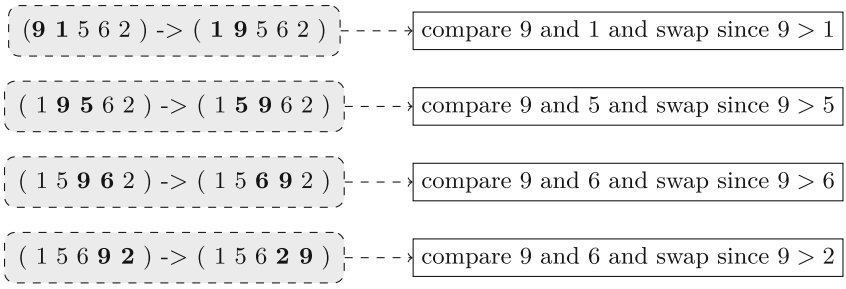

Fig. 8. A fragment of the visualization of the first pass of the bubble sort algorithm.

In order to visualize each step of the sorting algorithms, we use a function call at every key point inside the sorting algorithms and display the changes

that have been made so far to the unsorted sequence. The function provides the ability to set different pause times at the different key points. This allows an improved visualization and better understanding of the process of sorting, due to the difference in the way sorting algorithms work and the need of emphasis on some key points more than on others.

On Fig. 8 is given the visualization of the first pass of the bubble sort algorithm. The learner is able to run the algorithm and to monitor each stage of its execution on the selected input sequence.

The plot of the execution time of sorting algorithms give good visual representation of their complexity. For example, on the plot (Fig. 7) it is obvious that the complexity of the bubble sort algorithm can be modelled with a square function. Also, it is easily seen that among square complexity functions the worst performance is given by the bubble sort, selection sort performs significantly better than it, and insertion sort outmatches both of them. The demonstrated experiment clearly shows why bubble sort algorithm is usually ignored in contemporary programming courses.

4 Conclusion

In this paper we present our software tool to visualize sorting algorithms and their complexity. We provide a survey of some of the most frequently taught sorting algorithms in university course, tracing their origins, and providing analytical analyses of their complexity. The complexity of algorithms is an important notion usually presented in the course of computer programming, algorithms and data structures. However, it is not intuitively perceived by the students in the first years of their study because of the complexity of the asymptotic notation.

Our project can improve the intuitive understanding of the complex term complexity of algorithms by providing visual comparison of the execution time of different algorithms (see Fig. 7). Learners can perform experiments with different sequences, both random and predefined. Also, more advanced students can implement another sorting method as part of the same project, and to perform the same experiments with them.

The project is a part of our effort to organize the curriculum using a practical approach that is based on a system of task [14] that facilitates the development of complex notions during the learning process.

As future work we plan to perform both analytical and statistical analysis of algorithms with the same worst-case complexity that perform differently in practice. We also plan to compare the performance of such algorithms applied on different data structures: array based, such as vectors, and node/connection based, such as linked lists. Our software tool will be extended to visualize the execution of other types of algorithms, for example the standard graph algorithms that are taught in the courses of graph theory. Also, our source code is published as an open-source project[1], and can be extended and modified, according to the needs of a particular course.

[1] https://github.com/RitaPlusPlus/sorting.

References

1. Astrachan, O.: Bubble sort: an archaeological algorithmic analysis. SIGCSE Bull. **35**(1), 1–5 (2003)
2. Asenova, P., Marinov, M.: System of tasks in mathematics education. J. Educ. Res. Az Buki National Publishing House Educ. Sci. **62**(1), 52–70 (2019)
3. Bose, R.C., Nelson, R.J.: A sorting problem. J. ACM **9**(2), 282–296 (1962)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
5. Eng, L.Z.: Qt5 C++ GUI Programming Cookbook, 2nd edn. PACKT Publishing, Birmingham (2019)
6. Friend, E.H.: Sorting on electronic computer systems. J. ACM **3**(3), 134–168 (1956)
7. Goldstine, H.H., von Neumann, J.: Planning and Coding of Problems for an Electronic Computing Instrument, Part II, vol. 2, pp. 49–66. The Institute for Advanced Study Princeton, New Jersey (1947)
8. Gotlieb, C.C.: Sorting on computers. Commun. ACM **6**(5), 194–201 (1963)
9. Hoare, C.A.R.: Algorithm 64: quicksort. Commun. ACM **4**(7), 321 (1961)
10. Horstmann, C., Budd, T.: Big C++, 2nd edn. Addison-Wesley, Boston (2008)
11. Huang, S., Yang, J., Tang, Y.: A fast two-dimensional median filtering algorithm. IEEE Trans. Acoust. Speech Signal Process. **27**(1), 13–18 (1979)
12. Iverson, K.E.: A Programming Language. John Wiley, Hoboken (1962)
13. Knuth, D.: The Art Of Computer Programming, volume: 3 Sorting and Searching. Addison- Wesley, Boston (1973)
14. Laskov, L.: Introduction to computer programming through a system of tasks. Math. Inf. J. Educ. Res. Az Buki National Publishing House Educ. Sci. **64**(6), 634–649 (2021)
15. Nowak, R.: Generalized binary search. In: 2008 46th Annual Allerton Conference on Communication. Control, and Computing, pp. 568–574. IEEE, Monticello, IL, USA (2008)
16. Sedgewick, R.: Algorithms in C. Addison-Wesley Longman, Boston (2002)
17. Shustek, L.: Interview: an interview with CAR Hoare. Commun. ACM **52**(3), 38–41 (2009)
18. Sumathi, S., Esakkirajan, S.: Fundamentals of Relational Database Management Systems Studies in Computational Intelligence 47. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-48399-1
19. Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Hoboken (1976)