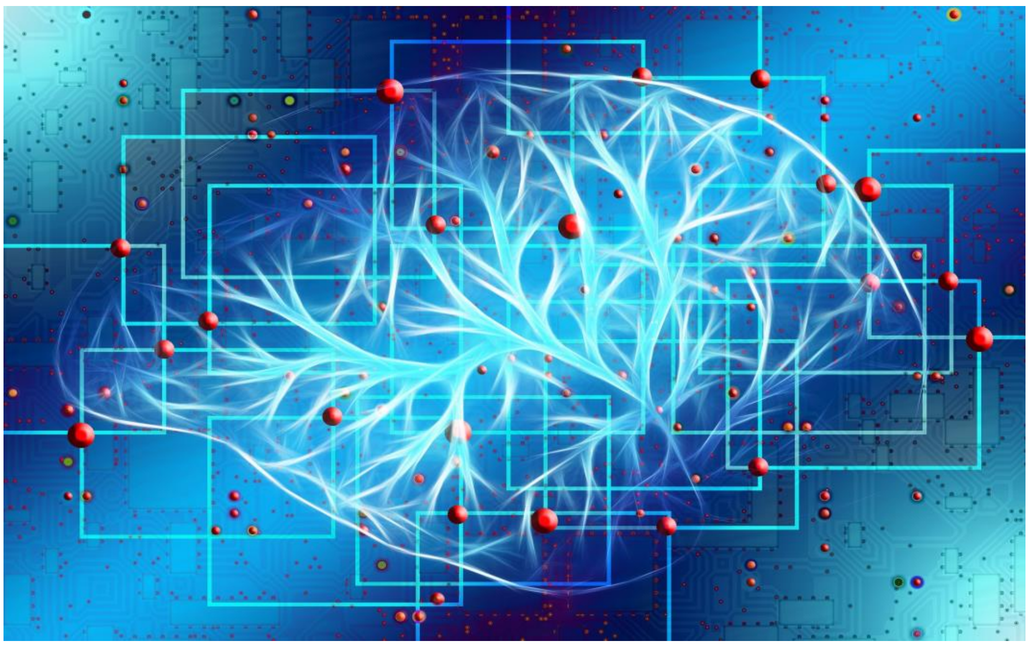
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ & ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

Διπλωματική Εργασία

ΑΛΓΟΡΙΘΜΟΙ ΚΑΙ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ: ΔΥΝΑΜΙΚΗ ΟΠΤΙΚΟΠΟΙΗΣΗ ΛΕΙΤΟΥΡΓΙΑΣ ΣΕ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΟ ΠΕΡΙΒΑΛΛΟΝ ΓΙΑ ΕΚΠΑΙΔΕΥΤΙΚΟΥΣ ΣΚΟΠΟΥΣ



Φοιτητής: ΜΠΑΝΑΣΙΟΣ ΓΕΩΡΓΙΟΣ, ΑΜ: 07088

Επιβλέπουσα Καθηγήτρια: ΡΑΓΚΟΥΣΗ ΜΑΡΙΑ

ΑΘΗΝΑ-ΑΙΓΑΛΕΩ, Φεβρουάριος 2023

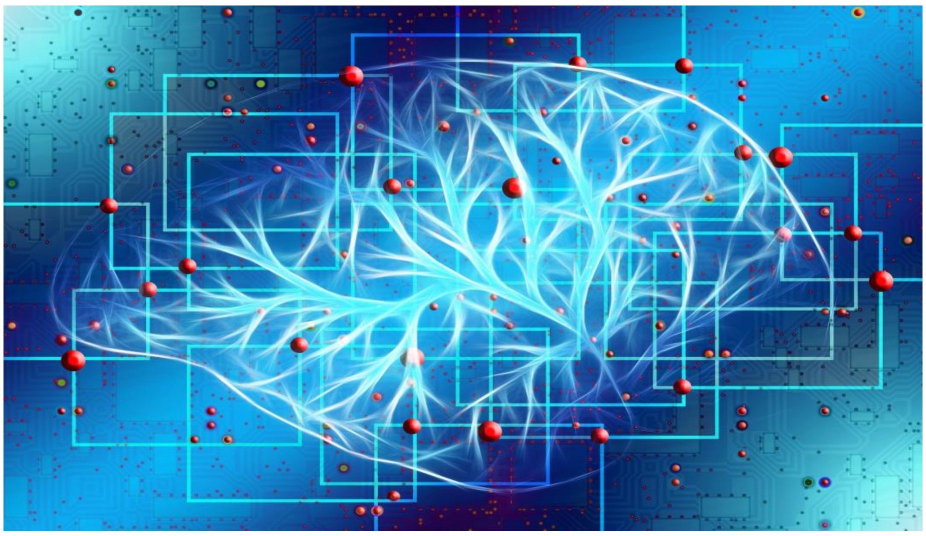
UNIVERSITY OF WEST ATTICA

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

Diploma Thesis

ALGORITHMS AND DATA STRUCTURES: DYNAMIC VISUALIZATION OF OPERATION IN A PROGRAMMING ENVIRONMENT FOR EDUCATIONAL PURPOSES



Student: GEORGE BANASIOS, Registration Number: 07088

Supervisor: Prof. Maria Rangoussi

ATHENS-EGALEO, February 2023

‘Algorithms and Data Structures: Dynamic visualization of operation in a programming environment for educational purposes’

Η Διπλωματική Εργασία έγινε αποδεκτή και βαθμολογήθηκε από την εξής τριμελή επιτροπή:

Μαρία Ραγκούση, Καθηγήτρια (επιβλέπουσα)	Αικατερίνη Ζαχαριάδου Καθηγήτρια	Δημήτριος Μετάφας Επ. Καθηγητής
Maria Ragkousi Digitally signed by Maria Ragkousi Date: 2023.03.16 21:21:20 +02'00' (Υπογραφή)	Aikaterini-Styliani Zachariadou Digitally signed by Aikaterini-Styliani Zachariadou Date: 2023.03.20 08:53:57 +02'00' (Υπογραφή)	Dimitrios Metafas Digitally signed by Dimitrios Metafas Date: 2023.03.22 18:27:16 +02'00' (Υπογραφή)

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος Γεώργιος Μπανάσιος του Κωνσταντίνου, με αριθμό μητρώου 07088 φοιτητής του Πανεπιστημίου Δυτικής Αττικής της Σχολής ΜΗΧΑΝΙΚΩΝ του Τμήματος ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ,

δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.

Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του διπλώματός μου.»

Ο Δηλών

[illegible]

Γεώργιος Μπανάσιος

Περίληψη

Οι δομές δεδομένων καθώς και οι αλγόριθμοι που χειρίζονται και μετασχηματίζουν αυτά τα δεδομένα για την επίλυση προβλημάτων αποτελούν θεμελιώδες γνωστικό αντικείμενο στην επιστήμη των υπολογιστών, με ευρύτατες εφαρμογές σε πολλά και διαφορετικά πεδία και κατηγορίες προβλημάτων. Παρά το θεμελιώδη ρόλο που διαδραματίζουν οι αλγόριθμοι στην επιστήμη των υπολογιστών, οι φοιτητές συχνά δυσκολεύονται να κατανοήσουν αυτό το αντικείμενο. Υπάρχουν διάφοροι λόγοι για τους οποίους οι φοιτητές δυσκολεύονται, όπως ο φόβος για τον προγραμματισμό (ειδικά για εκείνους που δεν έχουν προηγούμενη εμπειρία προγραμματισμού), η έλλειψη ενδιαφέροντος και η αφηρημένη φύση των εννοιών που εμπλέκονται, όπως η έννοια της πολυπλοκότητας ενός αλγορίθμου. Αυτές είναι οι κύριες αιτίες υψηλών ποσοστών εγκατάλειψης και αποτυχίας σε βασικά υποχρεωτικά μαθήματα του προγράμματος σπουδών, όπως μαθήματα προγραμματισμού, δομών δεδομένων και πολυπλοκότητας των αλγορίθμων. Ένα συναφές αντικείμενο που θεωρείται ταυτόχρονα δύσκολο αλλά και σημαντικό είναι η ανάλυση και ο σχεδιασμός αλγορίθμων. Για να αντιμετωπιστεί αυτό το πρόβλημα, έχουν γίνει πολλές ερευνητικές μελέτες για τον τρόπο βελτίωσης της διαδικασίας διδασκαλίας και μάθησης των αλγορίθμων.

Καθώς ο όγκος της πληροφορίας αυξάνεται με ταχείς ρυθμούς, αυξάνεται και η ανάγκη ταξινόμησης δεδομένων για καλύτερη ανάλυση και μελέτη, γεγονός που δίνει αυξανόμενη σπουδαιότητα στους αλγόριθμους ταξινόμησης. Αυτοί αποτελούν και το κύριο επίκεντρο της παρούσας διπλωματικής εργασίας. Η ταξινόμηση είναι μία ανάγκη που προκύπτει στο φυσικό κόσμο, για τη διευκόλυνση της αναζήτησης συγκεκριμένης οντότητας μέσα σε μεγάλο όγκο δεδομένων. Από τα αρχαία χρόνια οι άνθρωποι ανέπτυξαν ευρετήρια, λεξικά, καταλόγους και άλλες μορφές για να διατηρούν τη χρήσιμη πληροφορία ταξινομημένη, ειδικά αν το σύνολο δεδομένων είναι δυναμικό, δηλαδή νέα δεδομένα προκύπτουν διαρκώς ενώ παλιά πρέπει να αφαιρεθούν ή να διαγραφούν.

Πολλοί και αρκετά διαφορετικοί μεταξύ τους αλγόριθμοι ταξινόμησης έχουν αναπτυχθεί για τη βελτίωση της απόδοσης όσον αφορά την υπολογιστική πολυπλοκότητα και εν τέλει τους πόρους (χρόνο και χώρο) που θα απαιτηθούν, σε συνάρτηση με το μέγεθος των δεδομένων. Στη συγκριτική αξιολόγηση μεταξύ εναλλακτικών αλγορίθμων, υπάρχουν διάφοροι παράγοντες που πρέπει να ληφθούν υπόψη, όπως η πολυπλοκότητα ως προς το χρόνο, ο (επιπλέον) χώρος μνήμης που θα απαιτηθεί καθώς και η σταθερότητα του κάθε αλγορίθμου.

Επιστρέφοντας στο εκπαιδευτικό πρόβλημα της βέλτιστης διδασκαλίας και μάθησης του αντικειμένου για τους σημερινούς φοιτητές, με βάση έρευνες, η Οπτικοποίηση Αλγορίθμων έχει αποδειχθεί ότι έχει θετικό αντίκτυπο στην μάθηση των φοιτητών. Η παρούσα διπλωματική εργασία στοχεύει στη σχεδίαση και ανάπτυξη ενός γραφικού εργαλείου οπτικοποίησης αλγορίθμων ταξινόμησης που μπορούν να χρησιμοποιήσουν εύκολα οι φοιτητές. Η οπτικοποίηση δίνει έμφαση στην οπτική αναπαράσταση βημάτων και λειτουργιών ενός αλγορίθμου και γενικά θεωρείται πιο αποτελεσματική από τις προφορικές ή αριθμητικές παρουσιάσεις του αντικειμένου, όσον αφορά στην κατανόηση και στο γεγονός ότι είναι μία διαδικασία φιλική προς το χρήστη. Τα διαδραστικά εργαλεία οπτικοποίησης που διατίθενται σήμερα μπορούν να χρησιμοποιηθούν αποτελεσματικά για τη διδασκαλία και μάθηση σύνθετων εννοιών. Ειδικότερα, στο αντικείμενο της διπλωματικής, η γραφική απεικόνιση των βημάτων και των λειτουργιών των αλγορίθμων, δυναμικά ενώ εκτελούνται, μπορεί να βοηθήσει τους φοιτητές να κατανοήσουν και να συγκρίνουν διαφορετικούς αλγορίθμους ταξινόμησης πιο αποτελεσματικά.

Σκοπός της παρούσας διπλωματικής εργασίας είναι κατ’ αρχήν να γίνει μία σύντομη αναφορά στο αντικείμενο των αλγορίθμων και των διαφορετικών τύπων τους, προκειμένουν να κατανοηθεί η σημασία τους. Επίσης, πριν από το σχεδιασμό της εφαρμογής, θα γίνει μία σύντομη αναφορά στα βασικά θέματα της πολυπλοκότητας των αλγορίθμων. Η πολυπλοκότητα, χρησιμοποιείται ως εργαλείο για την μέτρηση της αποτελεσματικότητας ενός αλγορίθμου, καθώς είναι θεμελιώδες για την κατανόηση των διαφορετικών απαιτήσεων και επιδόσεων μεταξύ των αλγορίθμων ταξινόμησης. Με βάση αυτή τη γνώση, στη συνέχεια θα γίνει παρουσίαση των εξής αλγορίθμων ταξινόμησης:

- Bubble Sort,
- Selection Sort,
- Insertion Sort,
- Merge Sort, και
- Quick Sort.

Αυτό θα βοηθήσει τον αναγνώστη και τελικά το φοιτητή στην κατάκτηση ενός βασικού επιπέδου κατανόησης αυτών των μεθόδων, ως απαραίτηση βάση για την πλήρη κατανόησή τους με τη βοήθεια του εργαλείου οπτικοποίησης. Τέλος, θα σχεδιαστεί και θα υλοποιηθεί μία online, web-based εφαρμογή στην οποία ο χρήστης θα μπορεί, μεταξύ άλλων λειτουργιών, να εισάγει ή να επιλέξει μία ακολουθία θετικών ακεραίων αριθμών εισόδου και να επιλέξει έναν συγκεκριμένο από τους 5 διαθέσιμους αλγόριθμους ταξινόμησης.

Χρησιμοποιώντας ένα εργαλείο απεικόνισης που βασίζεται σε ραβδόγραμμα, ο φοιτητής θα αποκτήσει καλύτερη κατανόηση της λειτουργίας του συγκεκριμένου αλγορίθμου. Επιπλέον θα μπορέσει να τον εκτελέσει σε δικά του δεδομένα, ή να χρησιμοποιήσει τα ίδια «τυχαία» δεδομένα για να συγκρίνει μεταξύ τους δύο ή περισσότερους αλγορίθμους. Η σχεδίαση έχει δώσει έμφαση στην λειτουργικότητα από πλευράς εκπαιδευτικής/διδακτικής, καθώς ο πρωτεύων ρόλος της εφαρμογής είναι να αποτελέσει εργαλείο υποστήριξης του διδάσκοντα σε ένα πανεπιστημιακό τμήμα.

Λέξεις – κλειδιά

αλγόριθμος, δομές δεδομένων, υπολογιστική πολυπλοκότητα, ταξινόμηση, οπτικοποίηση αλγορίθμων, διαδραστική μάθηση, οπτικοποιητής, JavaScript, CSS, εφαρμογή παγκόσμιου ιστού

Abstract

Data structures and the algorithms that manipulate and transform these data to solve problems are a fundamental subject in computer science, with wide-ranging applications in many different fields and classes of problems. Despite the fundamental role that algorithms play in computer science, students often struggle to understand this subject. There are several reasons why students struggle, including fear of programming (especially for those with no prior programming experience), lack of interest, and the abstract nature of the concepts involved, such as the concept of the complexity of an algorithm. These are the main causes of high drop-out and failure rates in core compulsory courses of the curriculum, such as courses in programming, data structures and complex algorithms. A related subject that is considered both difficult and important is the analysis and design of algorithms. To address this problem, many research studies have been conducted on how to improve the teaching and learning process of algorithms.

As the volume of information increases rapidly, so does the need to sort data for better analysis and study, which gives increasing importance to sorting algorithms. These algorithms are also the main focus of this thesis. Sorting is a need that arises in the physical world, to facilitate the search for a specific entity within a large volume of data. Since ancient times, people have developed indexes, dictionaries, directories, and other forms to keep useful information organized, especially if the data set is dynamic, meaning if new data is constantly being generated while old data must be removed or deleted. Many and quite different sorting algorithms have been developed to improve performance in terms of computational complexity and ultimately the resources (time and space) required, depending on the size of the data. In comparative assessment between alternative algorithms, there are several factors to consider, such as the time complexity, the (extra) memory space that will be required, and the stability of each algorithm.

Returning to the educational problem of optimal teaching and learning of the subject for today's students, based on research, Algorithm Visualization has been shown to have a positive impact on student learning. This thesis aims to design and develop a graphical tool for visualizing sorting algorithms that can be easily used by students. Visualization emphasizes the visual representation of steps and operations of an algorithm and is generally considered more effective than verbal or numerical presentations of the subject in terms of understanding and being a user-friendly process. Interactive visualization tools available today can be used effectively to teach and learn complex concepts. In particular, in the subject matter, graphically illustrating the steps and operations of algorithms, dynamically while they are running, can help students understand and compare different classification algorithms more effectively.

The purpose of this thesis is, in the first place, to make a brief reference to the subject of algorithms and their different types, in order to understand their importance. Also, before designing the application, a brief reference will be made to the basic issues of the complexity of the algorithms. Complexity is used as a tool to measure the effectiveness of an algorithm, as it is fundamental to understanding the different requirements and performances between sorting algorithms. Based on this knowledge, the following sorting algorithms will then be presented:

- Bubble Sort,
- Selection Sort,
- Insertion Sort,
- Merge Sort, and
- Quick Sort.

This will help the reader and ultimately the student to gain a basic level of understanding of these methods, as a necessary basis for fully understanding them with the help of the visualization tool. Finally, an online, web-based application will be designed and implemented in which the user will be able, among other functions, to enter or select a sequence of positive integer input numbers and select a specific one of the 5 available sorting algorithms. By using a bar graph based visualization tool, the student will gain a better understanding of how the particular algorithm works. In addition, the student will be able to run it on its own data, or use the same "random" data to compare two or more algorithms. The design has emphasized functionality from an educational/teaching point of view, as the primary role of the application is to be a support tool for the teacher in a university department.

Keywords

algorithms, data structures, computational complexity, sorting, algorithm visualization, interactive learning, visualizer, JavaScript, CSS, web application

Table of Contents

List of Figures

INTRODUCTION

Understanding algorithms and data structures is essential for students seeking to pursue a career in computer science as they are key ideas in the subject. Particularly important to this topic are sorting algorithms, which are used to arrange data in a certain order for easier analysis and manipulation.

The subject of this thesis

The subject of this thesis is to enhance the learning experiences for students studying this topic by developing a web application for the visualization of sorting algorithms. The abstract nature of sorting algorithms makes it difficult for students to fully understand the concepts involved, which is one of the barriers of learning about them. This is especially true for people who do not have a lot of experience with programming or computer science. The web application tries to overcome this issue and make the topic more approachable and interesting for students by offering a more intuitive and interactive way to learn about these algorithms. Because it has been shown to improve learning outcomes, the use of visualization as an educational process is a hot topic in the field of computer science education. This makes the subject of this thesis seasonable and relevant, as it aims to utilize the advantages of visualization to improve the process of teaching and learning about algorithms.

Aim and objectives

The purpose of this thesis is to develop a web application for sorting algorithm visualization in order to enhance the learning process for students who are studying this topic. The specific goals of this thesis are to: examine the underlying ideas of algorithms and computational complexity, to comprehend the characteristics and performance of several sorting algorithms, analyze the potential educational benefits of visualization in the context of sorting algorithms and explore its application as an educational process, implement a web application for sorting algorithm visualization that takes into account the demands and educational objectives of students, and lastly to illustrate the operation of each of the sorting algorithms that is included in the application.

Methodology

In order to successfully create such an interactive educational tool, the students’ needs and learning objectives were taken into account. The application was created to be interactive as is it designed to give students a practical and interesting learning experience. One of the functionalities that achieve this purpose is the ability the student has, to provide their own input in order to test a specific algorithm. For displaying the outcome of each algorithm, a bar graph was used, giving the data a simple and understandable representation while also assisting students in better comprehending how the algorithm operates.

Innovation

This thesis presents an innovative and original approach to the teaching and learning of sorting algorithms through the development of a web application for their visualization. Although using visualization as an educational process is not new, the application created for this thesis elevates the idea by giving students a dynamic, interactive and interesting approach to learn about these methods. Students can test the algorithms using their own dataset among the built-in arrays in the application, they can also observe each iteration of an algorithm in a step-by-step procedure, which offers a clear and user-friendly graphical representation of the data to help students better grasp the algorithms’ key concepts and complexities.

Structure

This thesis is divided into five main chapters. The 1st chapter will give an overview of algorithms and computational complexity, while defining the key terms and concepts that are important for comprehending the subject. The details of sorting algorithms will be covered in more detail in the 2nd chapter, along with a discussion of how well they perform in terms of time and space complexity. The 3rd chapter will examine the use of visualization as a teaching tool, highlighting its advantages and how educators can utilize it to their advantage in the classroom. The 4th chapter will go into detail about the features and functionality that were included in the application as well as the decisions that were made at each stage of the development process. Finally, each sorting algorithm included in the application, will be explained in the 5th chapter, utilizing the visualization tool to demonstrate how they operate.

CHAPTER 1: Introduction to the concept of algorithms and complexity

1.1 Introduction

The first chapter gives a general introduction on the subject of algorithms and on the concept of computational complexity. Starting off, there will be a discussion about the definition of an algorithm, the various types of algorithms and their applications. An introduction will be provided on the concept of computational complexity, along with its uses in order to measure the effectiveness of an algorithm. The “big O” notation, which is typically used to describe the asymptotic complexity of an algorithm is covered in the last part. The comprehension of these basic concepts for computer science, as outlined in this chapter, is a necessary stepping stone for the development of the web application to visualize the selected sorting algorithms and their steps.

1.2 Algorithm: a definition

An algorithm is a set of instructions or procedures for solving computational problems or resolving a particular task. Algorithms are used to handle data, conduct complex calculations and automate different procedures. They offer a methodical approach to problem solving and are an imperative tool for creating and putting into practice quality software. Several problems can be addressed using algorithms, such as data searching and sorting, process optimization, and analysis and interpretation of big datasets. They can be utilized in a range of settings, including as scientific research, business and daily life, while they can be implemented in a plethora of programming languages.

Computer science relies heavily on algorithms which have a wide range of real-world applications. The effective manipulation and interpretation of large amounts of data is one of the major purposes of algorithms in computer science. For instance, algorithms are applied for data analysis and visualization, data compression and encryption, and searching and sorting of massive databases.

Countless real-world uses of algorithms can be found in areas like artificial intelligence, data mining, and machine learning. In these domains, algorithms are used to automate decision-making procedures, learn from data, and generate predictions.

1.3 Types of algorithms

Algorithms are divided into several different categories or classes, each category serving a specific purpose. Some of the most known types of algorithms are:

- Sorting algorithms: They are utilized to arrange data in a certain order, given a rule and a direction (ascending or descending arrangement). Arithmetic values (for numerical variables) or alphabetic (string) values (for categorical variables) can be sorted according to the arithmetic or lexicographic order. Bubble sort, insertion sort and quick sort are a few examples.
- Search algorithms: These algorithms are used in order to find particular items within a large collection of data. Some prominent examples are linear search and binary search. Search algorithms relay on a ‘key’ for searching.

According to their internal structure, algorithms may be characterized as

- Divide and conquer algorithms: These algorithms break a problem down into smaller subproblems, solve the subproblems and then combine these answers to solve the original problem. Merge sort and quick sort algorithms make use of this technique.
- Brute force algorithms: Brute force algorithms choose the optimal solution after trying out every possible one. The drawback of such algorithms is that they can be inefficient for problems of large dimensions even though they are typically simple to apply. Problems such as n-queens and the traveling salesman are a few examples that make use of brute force algorithms.
- Dynamic Programming (DP) algorithms: DP algorithms divide optimization issues into smaller subproblems and store the solution to these subproblems in order to reduce repeated work. The Fibonacci sequence and the Knapsack problem are two examples of DP algorithms.
- Greedy algorithms: these algorithms at each step choose the locally best option in the expectation that it will result in a globally optimal solution. Examples include the shortest path algorithm developed by Dijkstra and the Huffman coding algorithm.
- Graph algorithms: Networks like social networks and transportation networks can be evaluated and modified using graph algorithms. Shortest path algorithms and depth-first search algorithms and such examples.

1.4 What is computational complexity?

Computational complexity is a measure of the efficiency of an algorithm and is used to understand the resources required to solve a given problem. Complexity is important given the fact that, depending on the precise implementation and the input data, different algorithms may have different levels of efficiency. There are several factors that contribute to the computational complexity of an algorithm. A major factor is the time required for an algorithm to arrive to a solution, which is evaluated through the number of basic operations required by the algorithm in order to compute the solution. Algorithms can be also judged for their space complexity, i.e. the amount of memory or storage needed to execute a task. When memory limitations apply, space complexity becomes the crucial factor since it determines the feasibility of a specific algorithm for the problem. Time and space are the major factors used to assess algorithm efficiency and to compare alternative algorithms on the same problem.

1.5 Asymptotic notation

Asymptotic notation is a mathematical notation used to represent the behavior of a function as the size of the input data increases towards infinity. The terms “time complexity” and “space complexity” are extensively used in computer science to refer to the resources needed to solve a specific task.

There are several types of asymptotic notations, such as *big O*, *big Ω* and *big Θ.*

- Big *O* notation is used to express the upper bound on the time complexity of an algorithm, i.e., the maximum number of steps that may be needed to solve a problem.
- Big *Ω* notation is used to express the lower bound on the time complexity of an algorithm, meaning that it provides a lower limit of the steps necessary to solve a problem.
- When both the upper and lower bounds are known, the time complexity of an algorithm is expressed using the big *Θ* notation.

Asymptotic notation is helpful for comprehending the overall behavior an algorithm across any possible input data set, as opposed to how well it performs when given a particular input data set. It is typically used to comparatively evaluate how efficient different algorithms are at solving a given problem. By understanding the asymptotic complexity of an algorithm, it is possible to determine the feasibility and practicality of implementing and using a specific algorithm for a specific problem.

1.5.1 Asymptotic upper bound

Big O notation is a form of mathematical notation used to express the upper bound on the time complexity of an algorithm. It is commonly used to define an algorithm’s worst-case time complexity, meaning the maximum number of steps that can be applied to solve a task when the input data set is ‘adversarial’. The capital letter “O” is used to symbolize big O notation, which is then followed by a function that specifies the algorithm’s time or space complexity. In Chapter 2, for instance, a sorting algorithm is presented with a worst-case time complexity of $O(n^2)$, meaning that it will take at most n^2 steps to sort a collection of input data of size n. Big O notation is used to describe an algorithm’s behavior over time rather than how well it performs when given a particular input. It helps assess the feasibility of utilizing an algorithm for a particular problem by knowing how time-consuming it is.

1.5.2 Asymptotic lower bound

Big Ω notation is a form of mathematical notation used to express the lower bound on the time complexity of an algorithm. It is frequently used to define an algorithm’s best-case time complexity, which lowers the maximum number of steps that can be performed to solve a problem. The capital letter “Ω” is used to symbolize big Ω notation, which is then followed by a function that specifies the algorithm’s time or space complexity. For example, if an algorithm has a best-case time complexity of $\Omega(n)$ this indicates that it will need at least n steps to solve a problem of size n.

1.5.3 Asymptotic tight bound

When both the upper and lower bounds of an algorithm are known, the time complexity of an algorithm is expressed mathematically using the big Θ notation, symbolized by the greek capital letter “Θ”. It is used to describe the average-case time complexity of an algorithm, meaning that is serves as both an upper and lower bound on the number of steps necessary to solve a problem. To give a more complete understanding of the time complexity of an algorithm, big Θ notation is commonly used in conjunction with big O and big Ω notations. For instance, an algorithm with a worst-case time complexity of $O(2^n)$ and a best-case time complexity of $\Omega(n)$ may have average time complexity of $\Theta(nlogn)$. In this example, the average-case time complexity of the algorithm is described by the function *nlogn*. Figure 1.1 shows a graph of various Big O time complexities versus the input size of n data elements:

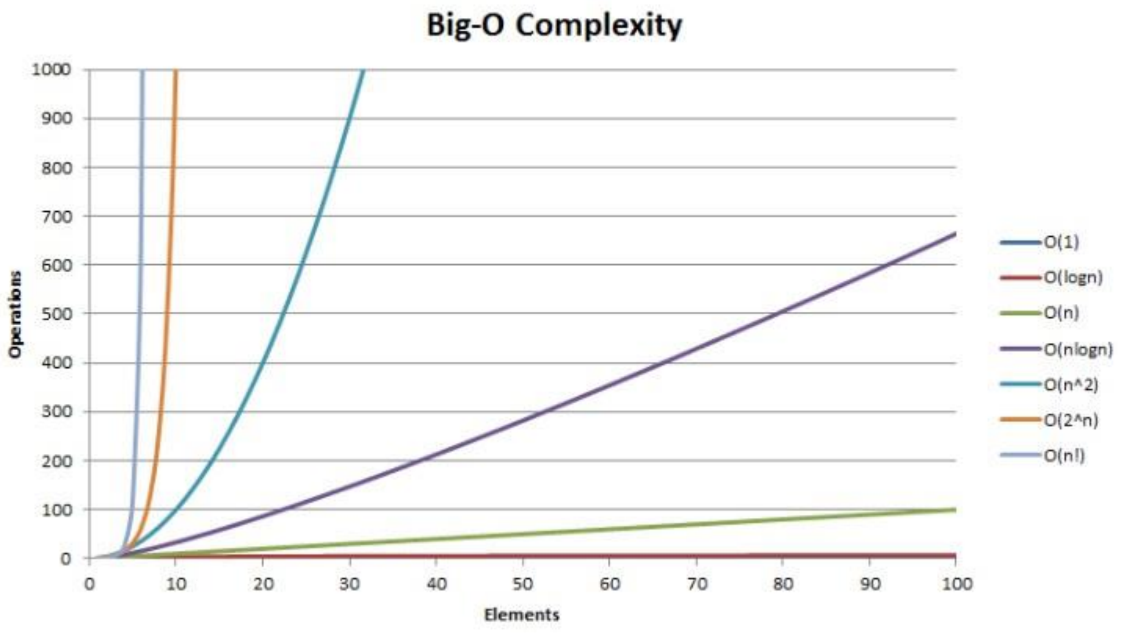


Figure 1.1: Big O complexity graph.

CHAPTER 2: Presentation and analysis of sorting algorithms

2.1 Introduction

After the brief explanation of what is an algorithm and what are the ways of measuring its efficiency, in this chapter we will proceed to the class of sorting algorithms. Different forms of data can be rearranged using sorting algorithms in an ascending or descending order. The following sorting algorithms will be covered:

1. bubble sort,
2. selection sort,
3. insertion sort,
4. merge sort and
5. quick sort.

For each one of them we will give a definition, a synopsis of how it works, an example and a breakdown of the algorithm time and space complexity using the big O notation. Understanding the various sorting algorithms and their features, is a crucial step before the design of the algorithm visualizer application.

2.2 The concept of sorting

Sorting is the process of rearranging a sequence of items into a specific order. Sorting can be done with numerous data types, such as numbers, words or strings of alphanumeric characters or more complex structures that combine domains of alphabetic, numeric or logical types. To accomplish a task like that, a plethora of sorting algorithms has been developed, each bearing certain advantages and disadvantages. The order of a given sorting task refers to the rule of ordering: numerical order for arithmetic values, alphabetic or lexicographical order for words or strings, or chronological order for dates, are some common examples of the ways in which data can be sorted. Another aspect is direction: ordering may be done in an ascending or a descending direction. Sorting algorithms are a first chapter in any course on Algorithms in a Computer Science or Engineering curriculum. In such academic contexts, sorting algorithms are used to introduce students to the fundamental concepts of algorithms, data structures and complexity analysis. Real-world applications of sorting algorithms expand to many fields such as Data Analysis which analyzes data to extract and understand patters within the data, or Database Management which assists in organizing and retrieving data efficiently based on queries.

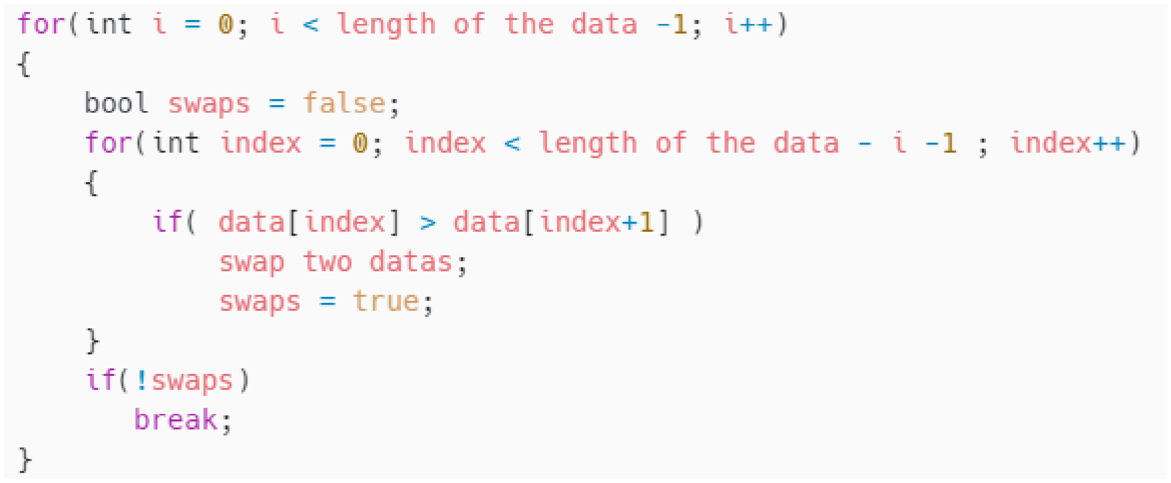
2.3 Bubble Sort

Bubble sort is a simple algorithm that sorts a sequence of input data (elements) by repeatedly iterating through the list, comparing adjacent elements and swapping them if they are in the wrong order. The sequence of input data is considered to be sorted when no further swaps are needed. It is called ‘bubble sort’ because the elements of lower values (the ‘lighter ones’) or will ‘bubble’ up to the top of the list as they are being sorted – assuming an ascending direction is required, from the lower (top) to the higher (bottom) value.

The worst-case time complexity of bubble sort is $O(n^2)$, the best-case time complexity is $O(n)$, and the average-case time complexity is $O(n^2)$. This indicates that as the size of the input collection of data increases, the time it takes for the algorithm to run grows exponentially. Bubble sort has a space complexity of $O(1)$, meaning that it only needs a constant amount of extra memory, since it is an *in-place* sorting algorithm.

Although bubble sort is straightforward to comprehend and implement, it is not particularly efficient, especially for large collections of data. Suppose we have the following sequence of integer numbers that we want to sort in ascending order:

Input Data: The list of integer numbers: [15, 16, 6, 8, 5]



```
for(int i = 0; i < length of the data -1; i++)
{
    bool swaps = false;
    for(int index = 0; index < length of the data - i -1 ; index++)
    {
        if( data[index] > data[index+1] )
            swap two datas;
            swaps = true;
    }
    if(!swaps)
       break;
}
```

Figure 2.1: Pseudocode for bubble sort.

First Iteration

[**15**, **16**, 6, 8, 5] → [**15**, **16**, 6, 8, 5], Since 15 < 16 there is no need to swap.

[15, **16**, **6**, 8, 5] → [15, **6**, **16**, 8, 5], Since 16 > 6 a swap is performed.

[15, 6, **16**, **8**, 5] → [15, 6, **8**, **16**, 5], Since 16 > 8 a swap is performed.

[15, 6, 8, **16**, **5**] → [15, 6, 8, **5**, **16**], Since 16 > 5 a swap is performed.

Second Iteration

[**15**, **6**, 8, 5, 16] → [**6**, **15**, 8, 5, 16], Since 15 > 6 a swap is performed.

[6, **15**, **8**, 5, 16] → [6, **8**, **15**, 5, 16], Since 15 > 8 a swap is performed.

[6, 8, **15**, **5**, 16] → [6, 8, **5**, **15**, 16], Since 15 > 5 a swap is performed.

[6, 8, 5, **15**, **16**] → [6, 8, 5, **15**, **16**], Since 15 < 16 there is no need to swap.

Third Iteration

[**6**, **8**, 5, 15, 16] → [**6**, **8**, 5, 15, 16], Since 6 < 8 there is no need to swap.

[6, **8**, **5**, 15, 16] → [6, **5**, **8**, 15, 16], Since 8 > 5 a swap is performed.

[6, 5, **8**, **15**, 16] → [6, 5, **8**, **15**, 16], Since 8 < 15 there is no need to swap.

[6, 5, 8, **15**, **16**] → [6, 5, 8, **15**, **16**], Since 15 < 16 there is no need to swap.

Fourth Iteration

[**6**, **5**, 8, 15, 16] → [**5**, **6**, 8, 15, 16], Since 6 > 5 a swap is performed.

[5, **6**, **8**, 15, 16] → [5, **6**, **8**, 15, 16], Since 6 < 8 there is no need to swap.

[5, 6, **8**, **15**, 16] → [5, 6, **8**, **15**, 16], Since 8 < 15 there is no need to swap.

[5, 6, 8, **15**, **16**] → [5, 6, 8, **15**, **16**], Since 15 < 16 there is no need to swap.

Output Data: The final sorted array is [5, 6, 8, 15, 16]

An important observation here is that for an array of size n, the algorithm requires n-1 iterations to sort the elements in ascending order.

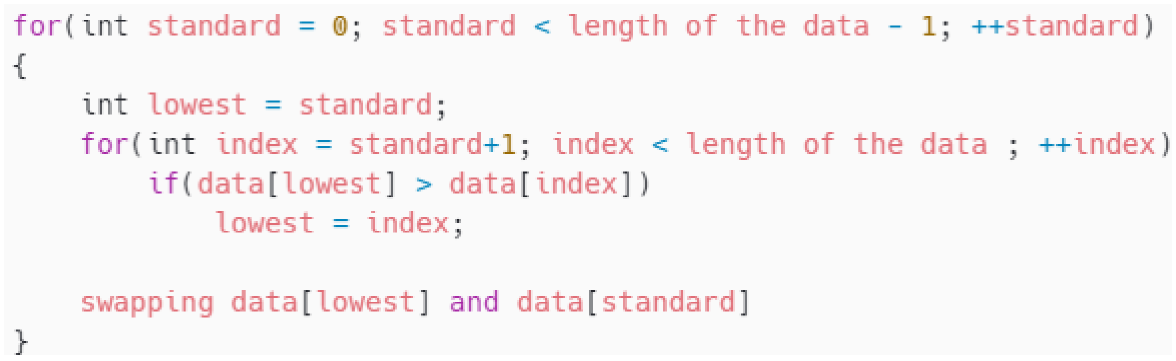
2.4 Selection Sort

Selection sort is a straightforward sorting algorithm that works by repeatedly finding the minimum element (when considering ascending order) from the unsorted portion of the array and placing it at the beginning. For any given array of input data, this algorithm maintains two subarrays: the subarray that is already sorted and the subarray that is unsorted and remains to be sorted. In every iteration of selection sort, the minimum element from the unsorted subarray is chosen and moved to the sorted subarray (taking into account ascending order).

The best-case time complexity of selection sort is O(n^2). Worst-case and average-case time complexities of selection sort are both equal to O(n^2) as well. This is because for each element inserted at the beginning, the algorithm must scan all n items. Just like bubble sort, selection sort is an *in-place* sorting algorithm, so there is no the need for additional space: the space complexity is O(1).

Suppose we have the following sequence of numbers that we want to sort in ascending order:

Input Data: list of integer numbers [7, 4, 10, 8, 3, 1]



```
for(int standard = 0; standard < length of the data - 1; ++standard)
{
    int lowest = standard;
    for(int index = standard+1; index < length of the data ; ++index)
        if(data[lowest] > data[index])
            lowest = index;

    swapping data[lowest] and data[standard]
}
```

Figure 2.2: Pseudocode for selection sort.

Sorted subarray: []

Unsorted subarray: [7, 4, 10, 8, 3, 1]

First Iteration

The minimum element in the unsorted portion of the subarray is 1. Swap 1 with the element at index 0 in the sorted portion of the array.

Sorted array: [1, 4, 10, 8, 3, 7]

Unsorted subarray: [4, 10, 8, 3, 7]

Second Iteration

The minimum element in the unsorted portion of the subarray is 3. Swap 3 with the element at index 1 in the sorted portion of the array.

Sorted array: [1, 3, 10, 8, 4, 7]

Unsorted array: [10, 8, 4, 7]

Third Iteration

The minimum element in the unsorted portion of the subarray is 4. Swap 4 with the element at index 2 in the sorted portion of the array.

Sorted array: [1, 3, 4, 8, 10, 7]

Unsorted array: [8, 10, 7]

Fourth Iteration

The minimum element in the unsorted portion of the subarray is 7. Swap 7 with the element at index 3 in the sorted portion of the array.

Sorted array: [1, 3, 4, 7, 10, 8]

Unsorted array: [10, 8]

Fifth Iteration

The minimum element in the unsorted portion of the subarray is 8. Swap 8 with the element at index 4 in the sorted portion of the array.

Sorted array: [1, 3, 4, 7, 8, 10]

Unsorted array: [10]

Output Data: The final sorted array is [1, 3, 4, 7, 8, 10].

Again here, just like bubble sort algorithm, for an array of n elements, selection sort needs n-1 iterations to sort the elements.

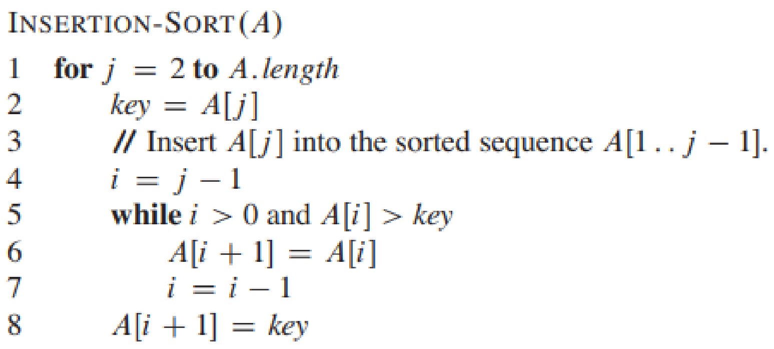
2.5 Insertion Sort

Insertion sort is a straightforward, intuitive sorting algorithm that works efficiently on short collections of data. This algorithm sorts a sequence of input data by continuously inserting each item into the suitable position. Just like selection sort, this algorithm maintains two sub-lists of the original list: an unsorted one and a sorted one. It starts by considering the first element in the input data as the only sorted element while all the rest are placed in the unsorted list. Then progressively the algorithm selects the next unsorted element and inserts it into the sorted sub-list, at the appropriate position (whence its name). To insert a single element into the sorted sub-list, the algorithm compares it with every element in the sorted sub-list until it finds the correct position for this element, based on the sorting order rule and direction (ascending or descending). The element is then inserted into the correct position, after a necessary number of elements have been shifted to make room for the new insertion.

When the list is sorted, the best-case time complexity for insertion sort is O(n). Both the worst-case and average-case time complexities are O(n^2). Because it sorts the list *in-place*, without using additional space, insertion sort has space complexity of O(1).

Suppose we have the following sequence of numbers that we want to sort in ascending order:

Input Data: list of integer numbers [5, 4, 10, 1, 6, 2]



```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Figure 2.3: Pseudocode for insertion sort.

The first element, 5, is regarded as the only sorted element.

Sorted portion of the list: [5]

Unsorted portion of the list: [4, 10, 1, 6, 2]

First Iteration

The next unsorted element, 4, is compared to all the elements in the sorted portion of the list (here, [5]).

Since 4 < 5, it is inserted at the start of the sorted portion.

Original list: [4, 5, 10, 1, 6, 2]

Sorted portion of the list: [4, 5]

Unsorted portion of the list: [10, 1, 6, 2]

Second Iteration

The next unsorted element, 10, is compared to all the elements in the sorted portion of the list (here, [4, 5])

Since 10 > 5, it is inserted at the end of the sorted portion.

Original list: [4, 5, 10, 1, 6, 2]

Sorted portion of the list: [4, 5, 10]

Unsorted portion of the list: [1, 6, 2]

Third Iteration

The next unsorted element, 1, is compared to all the elements in the sorted portion of the list (here [4, 5, 10]).

Since 1 < 4, it is inserted at the start of the sorted portion.

Original list: [1, 4, 5, 10, 6, 2]

Sorted portion of the list: [1, 4, 5, 10]

Unsorted portion of the list: [6, 2]

Fourth Iteration

The next unsorted element, 6, is compared to all the elements in the sorted portion of the list (here, [1, 4, 5, 10]).

Since 6 > 5 but 6 < 10, it is inserted after the element 5.

Original list: [1, 4, 5, 6, 10, 2]

Sorted portion of the list: [1, 4, 5, 6, 10]

Unsorted portion of the list: [2]

Fifth Iteration

The next unsorted element, 2, is compared to all the elements in the sorted portion of the list (here [1, 4, 5, 6, 10]).

Since 2 > 1 but 2 < 4, it is inserted after the element 1.

Original list: [1, 2, 4, 5, 6, 10]

Sorted portion of the list: [1, 2, 4, 5, 6, 10]

Unsorted portion of the list: [ ]

Output Data: the final sorted array is [1, 2, 4, 5, 6, 10].

2.6 Merge Sort

Merge sort is one of the most efficient sorting algorithms. It is a recursive algorithm, i.e., it uses the ‘divide and conquer’ strategy to repeatedly divide the array into shorter arrays, until each array is short enough to be easily sorted at the cost of one swap at most. These sorted subarrays are then merged back together to form a completely sorted array. In more detail, the first step is to divide the array into two halves, which this normally done by calculating the array midpoint and splitting it in half along this middle. The merge sort algorithm will then recursively call itself on each of these two subarrays. This implies that the algorithm will continue to split each half into smaller subarrays until its sub-list consists of only one element. The algorithm will now consider each element as having been "sorted". In the final phase, the algorithm will start the process of merging the sorted subarrays of the original list back together after the two halves of the array have been sorted. This is accomplished by comparing the first element of each half and putting back into the original array the smaller one. Once all of the elements from both halves have been merged back into the original list, the process is repeated.

Merge sort has a best-case time complexity of O(nlog(n)), and both worst-case and average-case complexities O(nlog(n)) as well. The space complexity of merge sort is O(n), however, because of the additional space needed to store the two half subarrays of the original list.

Suppose we have to sort the list of integer numbers: [70, 50, 30, 10, 20, 40, 60].

```
mergeSort(Arr,start,end)
if ( start < end )
       mid = ( start+end ) / 2
       mergeSort(Arr,start,mid)
       mergeSort(Arr,mid+1,end)
       Merge(Arr,start,mid,end)

Merge(Arr,start,mid,end)
n1 = mid - start + 1
n2  = end - mid
Let P[1,2,.,n1+1] and Q[1,2,..,n2+1] are two new arrays
For i = 1 to n1
      P[i] = A[start + i - 1]
for j=1 to n2
      Q[ j ] = A[ mid + j ]
P[ n1 + 1 ] = ∞
Q[ n2 + 1 ] = ∞
i = 1
j = 1
for k = start to end
      if P[i] <= Q[i]
             A[k]=P[i]
             i = i + 1
      else A[k] = R[j]
             j = j + 1
```

Figure 2.4: Pseudocode for merge sort.

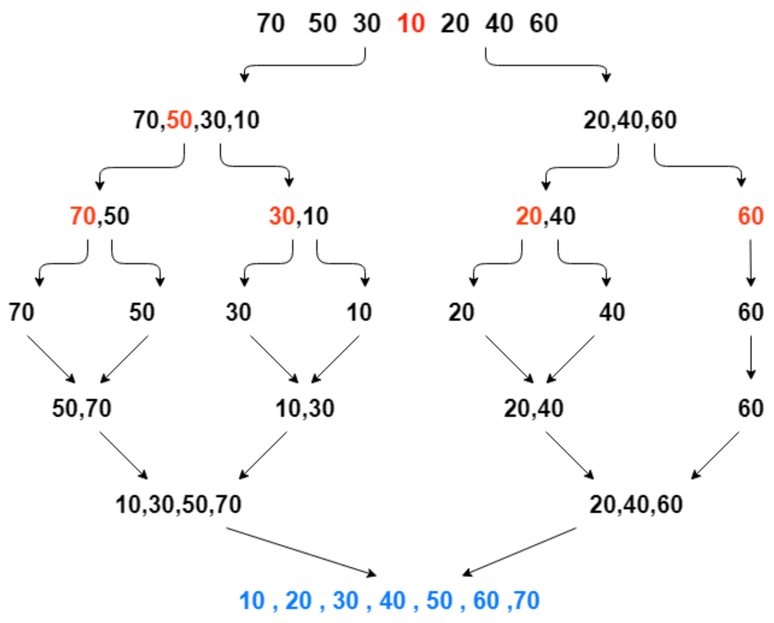


Figure 2.5: Merge sort example.

Splitting phase down to sub-arrays of 1 element each:

We first divide the array in half to obtain: [70, 50, 30, 10] and [20, 40, 60]. After that, we recursively divide each of these arrays in half again, until we get subarrays consisting of only one element: [70], [50], [30], [10], [20], [40], [60].

Merging phase:

We merge together these single elements as follows:

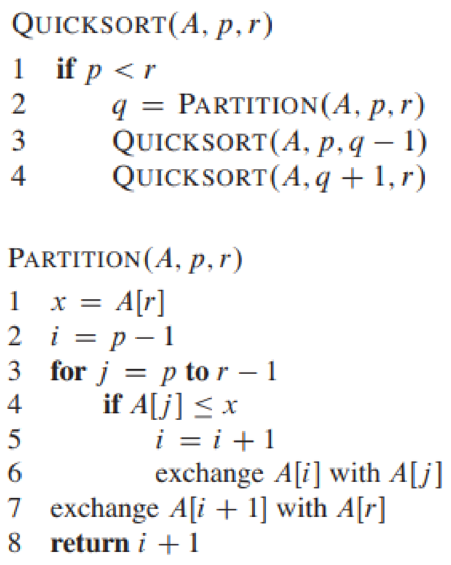
1. We compare 70 and 50 and we sort them as follows: [50, 70]
2. We compare 30 and 10, and we sort them as follows: [10, 30]
3. We compare 20 and 40, and we sort them as follow: [20, 40]
4. We merge [50, 70] with [10, 30] and the result is the following: [10, 30, 50, 70]
5. We merge [20, 40] with 60 and the result is the following: [20, 40, 60]
6. We finally merge the subarrays that were created from the two halves of the original array to get the final sorted array: [10, 20, 30, 40, 50, 60, 70]

2.7 Quick Sort

Quick sort is another recursive sorting algorithm which, like merge sort, uses the ‘divide and conquer’ technique. The algorithm selects a *pivot* element from the sequence and by using partitioning, it recursively rearranges the list such that all elements that are less than the pivot element are positioned to the left of the pivot element, and all the elements that are greater than the pivot element are placed to the right of it.

Quick sort has best-case time complexity of O(nlogn), meaning the pivot element would be selected so that the two sublists would roughly have the same number of elements. In the worst-case, the pivot element is selected in such a way that one of the sublists has just one element and the other n-1 elements, resulting to a time complexity of O(n^2). Quick sort has an average-case time complexity of O(nlogn). Quick sort has a space complexity of O(logn). There are some optimization techniques that ensure that quick sort is done in-place, meaning that it does not require extra space makes it efficient for sorting large lists.

Suppose we have to sort the list of integer numbers [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]



```
QUICKSORT($A, p, r$)
if $p < r$
    $q$ = PARTITION($A, p, r$)
    QUICKSORT($A, p, q - 1$)
    QUICKSORT($A, q + 1, r$)

PARTITION($A, p, r$)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \le x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$
```

Figure 2.6: Quick sort pseudocode.

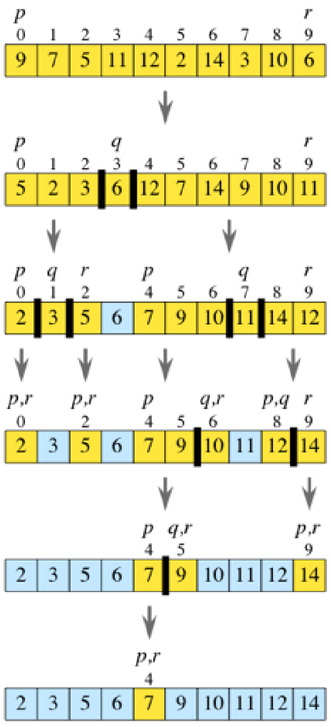


Figure 2.7: Quick sort example.

We assume that the pivot element is the last element in the array, which is 6. The initial partition would be the following: [5, 2, 3 | 6 | 12, 7, 14, 9, 10, 11]. The sublist on the left consists of all the elements that are less than the pivot element, 6, and the sublist to the left of 6, consists of all the elements that are greater than the pivot element.

The left subsequence is recursively partitioned again using the last element, 3, as the pivot element, and so is the right subsequence using the element 11 as the pivot:

[2 | 3 | 5, **6**, 7, 9, 10 | 11 | 14, 12]

The sublists [2] and [5] are already sorted. The sublist [7, 9, 10] is partitioned with the pivot element being 10, and the sublist [14, 12] is also being partitioned with the pivot element being 12:

[2, **3**, 5, **6**, 7, 9 | 10 | **11** | 12 | 14]

The sublist [7, 9] is partitioned with the pivot element being 9:

[**2**, **3**, **5**, **6**, 7 | 9 | **10** | **11** | **12**, 14].

Since the single elements are already sorted, the final sorted array is:

[2, 3, 5, 6, 7, 9, 10, 11, 12, 14].

CHAPTER 3: Visualization as an educational process

3.1 Introduction

Visualization has been proven to be a powerful tool for enhancing learning and understanding of complicated concepts in a variety of educational contexts. Students can better understand and retain information by utilizing visual representations like diagrams, charts, images and graphs. This is especially true for subjects such as sciences, including math and algorithms, where visualization can help students make sense of abstract and complex concepts. This chapter includes a review on the advantages of visualization in education and a discussion on the various methods visualization may be employed to improve instruction and learning.

3.2 The concept of visualization

Visualization is the process of conveying information and ideas through the use of visual aids, tools and techniques. Charts, diagrams, maps and other graphic representations of data and concepts are only a few examples of the many tools and methods that can be used to this end. Visualization makes it possible for the learner to observe the connections and relationships among different parts or pieces of information, e.g., among different facts, variables, concepts, procedures, etc. For this reason, it is frequently used to facilitate and enhance the comprehension and interpretation of complex fields of knowledge.

Visualization as an educational process refers specifically to the use of visualization as a tool for teaching and learning. Visualization is utilized in this scenario to make learning more interactive and engaging while assisting students to grasp the meaning and to better remember hard concepts, ideas, procedures, etc.

3.3 Visualization techniques

Some of the most commonly used visualization tools and techniques in an academic environment include:

- **Charts and graphs**: Charts and graphs are helpful for displaying numerical data and highlighting the connections among various variables. They are extensively used in fields such as

- ▪ economics, to represent data on topics such as GDP, unemployment, inflation and other economic indicators,
- ▪ social sciences, to demonstrate results on topics such as demographics, attitudes, psychology, behaviors, and other social phenomena, and
- ▪ data science, to depict data from a wide range of sources, including databases, sensor networks, social media etc. Data scientists utilize these charts to describe their findings and to detect trends, patterns and outliers/anomalies in massive datasets.

- **Diagrams**: Diagram is a form of visual representation that displays the connections and relationships between several concepts or ideas. They are often used in fields like
    - ▪ physics or engineering, where they demonstrate how systems or processes work,
    - ▪ computer science, where they illustrate how computer systems, networks and algorithms are built and function,
    - ▪ medicine, where they describe the structure and function of living organisms and their systems.
- **3D models**: 3D models are visual representations of three-dimensional systems or entities. They are most used in disciplines such as
    - ▪ architecture and civil engineering, where they allow architects to visually explain in three dimensions to clients and other interested parties how a building will be like when constructed,
    - ▪ engineering and manufacturing, where they aid in the system and/or product design and testing phases,
    - ▪ computer science, where they aid in the development of graphics characters for digital media, e.g., animated agents (characters) and/or other digital constructs within a virtual environment that can be animated in real-time,
    - ▪ sciences like chemistry or biology, where they aid represent in 3D constructs that cannot be directly observed because of volatility or scale,

- infographics: Infographics are graphic representations of data that are intended to be quickly absorbed by the spectator. They are frequently used in business and marketing to visualize data on sales, market trends or customer demographics,
- journalism, where they display data on noteworthy subjects like politics, economy, and health, and
- other fields, thanks to their ability to clearly and concisely convey complicated ideas.

3.4 Benefits of visualization

Over the past ten years, the development and quick adoption of new e-learning or distance learning methods has gained momentum. The traditional, face-to-face model of instruction is often supplemented and, in some cases, completely substituted by the e-learning / distance learning paradigm. Moreover, the traditional ‘one-to-many’ lecturing type of instruction has been gradually replaced by pedagogically advantageous methods such as collaborative learning, discovery learning, problem-based learning, etc. Visualization is a valuable tool along the paths of modern ways of instruction and learning.

One of the main benefits of the use of visualization and, in particular, of algorithm visualization tools in education, is that the introduction and use of such tools in the classroom is now easier than before, thanks due to the recent technological advancements. There are now more algorithm visualization tools available than before due to the easier accessibility through the Internet. The accessibility to teachers and learners of learning tools and content over the Internet, both from inside and outside of the classroom, makes the usage of algorithm visualization tools feasible. For instance, while it has long been “possible” to project Internet content from a computer on a screen in a classroom, such access has just recently become commonplace and Internet connections have become stable enough in all classrooms. This progress has a significant impact on how experienced and how confident instructors are in their ability to use such technologies.

Another benefit of visualization as an educational process is the ability to engage students and maintain their interest in the material especially when it involves explaining dynamic behavior or state changes of systems. The use of dynamic visualization has the advantage of the immediate feedback, that enhances the comprehension of both theoretical and practical issues. Indeed, it can be more interesting and entertaining for students to interact with visual aids when they can see them, as opposed to just reading about them or listening about them in a lecture. This is particularly true for students who have different learning preferences and may not respond well to conventional teaching strategies such as lectures or reading assignments. Regarding the subject of data structures and algorithms, dynamic and interactive visualization tools are a considerable help thanks to the fact that implementation details no longer obscure the students’ view, while visual depiction of the data structure enhances conceptual understanding.

Visualization also enables group collaboration and a more participatory discussion of ideas and concepts. As it enables students to see the connections and interactions between different parts, visualization can also assist students to think critically and creatively on the particular subject. Students can take part in more engaged and dynamic discussions that can help develop a deeper understanding and cultivate their problem-solving abilities by working together to analyze visual aids. In addition to the fact that it helps students to understand how their peers are conceptualizing and representing the material, visualization can also be leveraged to facilitate peer criticism and review because, apart from working together, such tools can inspire students to learn from and with each other as this promotes a climate of collaboration and support between them.

In addition to its advantages for students, visualization can be a useful tool for educators in terms of evaluation and assessment. It empowers teachers to design more interactive and interesting assessment activities that let students show their understanding in a more creative and exciting way. Professional educators can develop exam tests that are more interactive and interesting for students by integrating visual aids and approaches like diagrams, charts, and other graphic representations of information. This is very helpful in areas like art or design where students may feel more comfortable expressing themselves visually as opposed to in written or oral form.

Visualization can also support educators to assess and analyze student learning more accurately. Teachers can design evaluation activities that are better suited to the course specific learning objectives and more closely related to the piece of knowledge that has been taught by utilizing visual aids and strategies. As a result, it might be more straightforward for students to demonstrate their comprehension of the material in a way that is more meaningful and relevant, and it could be simpler to judge students’ learning more accurately.

It is possible that not all students will profit from visualization as a teaching strategy. Some students could have difficulty understanding or interpret visual information, or they might suffer from visual impairments that prevent them from utilizing visual tools effectively. In these circumstances, it would be required to use different teaching strategies or to offer more guidance.

Despite this possible drawback, visualization has the potential to be a prominent and useful tool for promoting greater comprehension and improving learning in a range of contexts. Visualization can assist educators in involving students and supporting their learning in significant and efficient ways, whether it is used in traditional classroom environments or online learning scenarios. By utilizing the advantages of visualization as an educational process, educators can support the development of a more dynamic and interactive learning process for their students, increase their active participation and promote critical thinking and problem-solving abilities.

CHAPTER 4: Design and Development of the Web Application

4.1 Introduction

This chapter presents the design and development of an educational visualizer in the form of a web application. More specifically, the application is a sorting algorithm visualizer which allows students to view the inner workings of the selected set of sorting algorithms in real time. A combination of HTML, CSS and JavaScript was used for the creation of the application. One of the key development considerations that was made during the design of the application was to include interaction between the application and the student. This enables the student to not only observe how the algorithms behave but also to experiment with different sets of input values and compare algorithm responses and efficiency.

Regarding the input, a justification is necessary for the choice that the student is allowed to enter input data only of the numerical type - positive integers, in particular. This choice was made to increase application simplicity and usability. Additionally, input data is given through the standard input device, i.e., the keyboard, rather than an external device like a hard disk. This allows students to easily and quickly enter values to test an algorithm. The application output is displayed in the form of a bar graph, with horizontal bars representing each element of the array. In order to offer the user control over the various functionalities of the application, a control panel with a variety of buttons was integrated in the GUI. Finally, considerable thought was given to the design of the screen layout, ensuring that all important buttons and controls are self-explanatory and easy to access and use.

4.2 Software tools employed for the development of the web application

The visualization application is essentially a web application which is accessible through a web browser. The application has a graphics user interface (front-end material) while it relies on standard web infrastructure regarding software and tools. Specifically, a combination of languages is employed for the implementation of this project, namely, HTML, CSS and JavaScript. These are supported by all major and popular web browsers as they are essential for web access over the Internet. The main programming language employed is JavaScript. The main reason behind the choice of JavaScript is because it is a programming language that runs directly on a web browser – this is the reason why most developers use this language, [2].

4.2.1 HTML

HyperText Markup Language (HTML) is a descriptive language for building web pages. This markup language tells the browser how to display a web page different forms of multimedia on such as text and images. It also handles the formatting of the digital content served within a web page. Although direct use of HTML for the development of a web page is not the way to go today, many of the modern sophisticated and user-friendly web page development environments automatically produce HTML code as the common intermediate step.

4.2.2 CSS

Cascading Style Sheets (CSS) is a W3C (World Wide Web Consortium) standard to describe the appearance of HTML elements. Through CSS, web designers and developers are able to define formatting properties for the web page digital content appearance such as fonts, colors, sizes, borders, images, background colors, etc.

4.2.3 JavaScript

JavaScript is a programming language that is used to create and control *dynamic* content on the Web. JavaScript along with HTML and CSS form the core technologies (software tools) used in modern web content development. The main purpose of JavaScript is to make web pages interactive. Dynamic content and interactivity constitute significant progress steps in the field of web site / web page development and usage, as compared to the previous technology that allowed the display of only static content and limited interactivity to just navigation. In contrast, JavaScript allows the web designer to add many interactive functionalities, to add dynamic content display, to embed animations and in general to offer the digital content in various multimedia forms that improve the overall user experience.

4.3 Major decisions on the functionalities of the application

4.3.1 Interactivity

As mentioned earlier, this application is meant for educational purposes. Application design is therefore strongly directed by the prominent factor of student or learner interactivity. It is widely known that technology has rendered modern education more engaging. Interactive learning is the process of making learning more active and engaging. Moreover, interactivity supports modern education scenarios that do not just lecture or instruct students on theoretical subjects but also provide hands-on experience (laboratories, workshops, collaborative projects, etc.) on real-world cases or setups, to effectively embed acquired knowledge. The aim of such scenarios and tools is to make sure that students really understand new knowledge and not just memorize and reproduce it.

In the case of the present application, interactive content allows students to not just read or listen as they would do in a traditional setup, but rather to participate and furthermore to explore and experiment with each algorithm step-by-step. Interactivity and multimedia forms allow learners to use multiple parts of their brains and fully grasp the new material they study. They are thus expected to develop the mental structure necessary in order to understand the concepts involved in the relevant data structures and algorithms. As demanding such a goal may be, yet, it is critical for the success of academic study programs.

Another advantage of practical interest is that interactive learning may be utilized by the class teacher or instructor in multiple ways. A first aim of such tools is to render class sessions attractive in the sense that they become more engaging and entertaining. Students are more likely to continue learning in their own time when they are immersed in an interactive stream of teaching.

4.3.2 Input provided by the user

Regarding the user input, we decided to allow the student to sort numbers and more specifically positive integers, instead of other forms of data such as strings, dates, names or even more complex data structures like objects and that is for a few reasons.

One of the main advantages of using positive numbers when sorting an array, is that they are typically easier to work with mathematically. Using simple mathematical processes, it is really simple to compare numbers to each other and it makes the order of the numbers obvious to perceive. As a result, students will find it simpler to comprehend the sorting algorithms and how they operate when they do not have to worry about an array consisting of more complex data structures.

On the same note, when sorting positive integers, there are no exceptional circumstances to take into consideration because they have a distinct, well-defined order. This makes it easier for students, instead of being caught down in marginal or exceptional cases, to be able to concentrate on the fundamental ideas of a sorting algorithms.

Finally, the choice of sorting positive integers rather than other types of numbers was made because positive integers are more intuitive for most people when the task is ‘sorting’. For instance, it is more intuitive to consider prices as positive numbers rather than negative numbers when sorting a list of prices.

In general, sorting positive integers rather than other types of data was used to render the application simpler and more efficient for the students to study and grasp the mechanism of each sorting algorithm.

Another important point on the input is that among all the popular user input methods, such as input through keyboard, input via a file stored in memory or input via data input line from an external peripheral device, the student is given the option to insert values only through the keyboard.

One reason behind this decision, is that typing in data through a keyboard is usually quicker and more convenient than opening a file and reading the data from it, especially when the number of data points is limited and accuracy is not crucial. The educational nature of this application limits the length of data input arrays. Real-time data entry is possible using keyboard input, which is useful in this situation as the aim of the application is interactivity between the student and the sorting visualizer.

Another reason is that input via keyboard is frequently considered as superior because it offers more freedom in terms of the format of the data, meaning that the data must be in a certain format that the application can understand when reading it from a file. In contrast, data can be entered using the keyboard in any format as long as the program knows how to interpret it.

Third, typing data in through a keyboard can be more secure than input from a file. There is a possibility that a file may have been altered or corrupted in some way when data is inserted from a file, which could result in errors or security flaws. Taking input from a keyboard, on the other hand, leads directly into the program and is not exposed to these risks.

Apart from the input via the keyboard, an alternative functionality is provided, namely, the automatic generation of a random array of a user-defined length, with values within a certain value range. Furthermore, the user may ask the application to replicate the last input data, for algorithm comparison purposes.

- First, utilizing a random array allows the student to observe how the sorting algorithm behaves on a wide range of various inputs, which can help in better understanding the specific algorithm with more clarity. The explanation for that is because a random array will always have a distinct set of elements, allowing the student to examine how the algorithm responds to different inputs and how its performance varies.
- Second, the use of a random array renders the sorting visualizer more interactive and engaging because the student can generate several different arrays and observe how

the algorithms sort them in real time. This can improve the application educational and entertaining value while also help the student understand the underlying principles of sorting algorithms.

- Finally, since the data that needs to be sorted is usually unknown in advance, implementing a random array can make the sorting visualizer more realistic and representative of real-world events. Instead of using a predetermined array, the student can experiment on how the algorithms would operate on real data by using a random array in the application.

As mentioned earlier, replication of the same data of a random array as many times as the user wants has a simple reason behind it. A student can first experiment with a random set of data observing a specific set of elements, and then for better comprehension just by pressing the “Same Data” button, the previous displayed array is being displayed again which gives the student the functionality for a deeper observation using now the step-by-step capabilities of the application.

At this point, it is worth mentioning the alternative user input methods that were rejected and the justification of these decisions. A prevalent method of input is via reading a file stored on the user’s computer. It may take longer and be a lot slower to read from a file which has been stored on a computer than it getting input from the keyboard. This is because the program must first locate the file on the computer, open it, then read its contents. Reading from a file can also increase the chance of an error since the file might not be correctly formatted or it might be missing which could have incorrect results or even a software crash. Moreover, validating the data inserted from a file might be more difficult. For instance, the program might not be able to make sure that the data in the file are numerical or within the specified range that is expected. Furthermore, taking input from keyboard gives the program the ability to process it right away, while when reading data from a file it could require the user to have particular permission rights which can be a barrier utilizing the application.

The choice of data input from an external peripheral device was also rejected. As already stated, the user can simply type the input values for the array directly into the input field and the software can instantly process it makes it a more fast and convenient method. In the case of an external peripheral device, the student must first connect the device to the computer before entering the data, which might be less intuitive and more time consuming. This makes the process of getting data from an external device much slower and more complicated.

In general, keyboard input is perceived as more effective, practical and secure than taking input from other sources.

4.3.3 Output produced by the application

The sorting algorithms application uses a graphical representation, more specifically a bar graph, along with animations showing the steps of the sorting algorithm as it processes (sorts) input data. The width of each bar represents the value of each element (integer) of the input array which allows the student to visualize relative sizes of all array elements. Animation gives the student the ability to observe the changes that are happening with every swap in the array during sorting, as the bar graph is updated in real-time. Overall, the combination of the bar chart and the animations on every iteration enables the student to easily comprehend the algorithm and see how the elements are sorted. This is the goal of the application, as it allows the student to easily interpret the results.

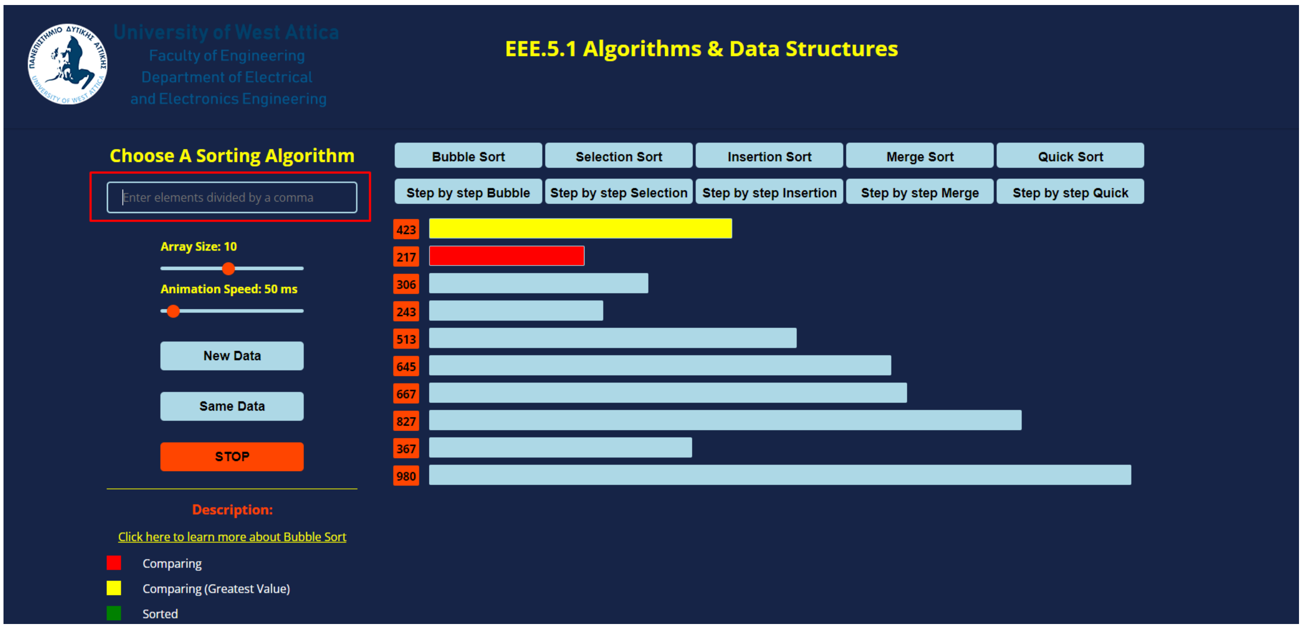
4.3.4 Control Panel

As already stated, the main utilization of this application is for educational purposes, hence, emphasis is given on the interactivity between the student and the application. Action takes place in the central screen of the application and the user interacts with it through the selection and pressing of graphics buttons.

The major decision on the design of the central application screen is the placement of the bars (i.e., elements of the input array); everything else is then placed according to that first decision. Horizontal bars placed vertically the one on top of the other are a design decision made to emphasize the idea of sorting as drawn for bubble sort: in bubble sort, the smaller bubble floats upwards to the top or ‘liquid’ surface, followed by the second-smaller, etc.

The next step was to decide on the overall functionalities of the application, meaning what the user should / should not be able to do. A major decision here is to allow the user control not only the speed of execution but also the mode: one-off or step-by-step. The step-by-step option is considered crucial as it allows the user run an algorithm at his/her own pace and have time to manually verify results on his/her list of input data. Moreover, the idea of watching each input number move to its correct position under the constraints of an algorithm was much easier to follow than tracing the code by hand. For that reason, students have the ability to control the application through the following buttons and utilities:

- **User input field:** A user input field is provided where the student can test and analyze a specific algorithm with their desired input. The values are separated by comma and the allowed input range is 0 to 999.

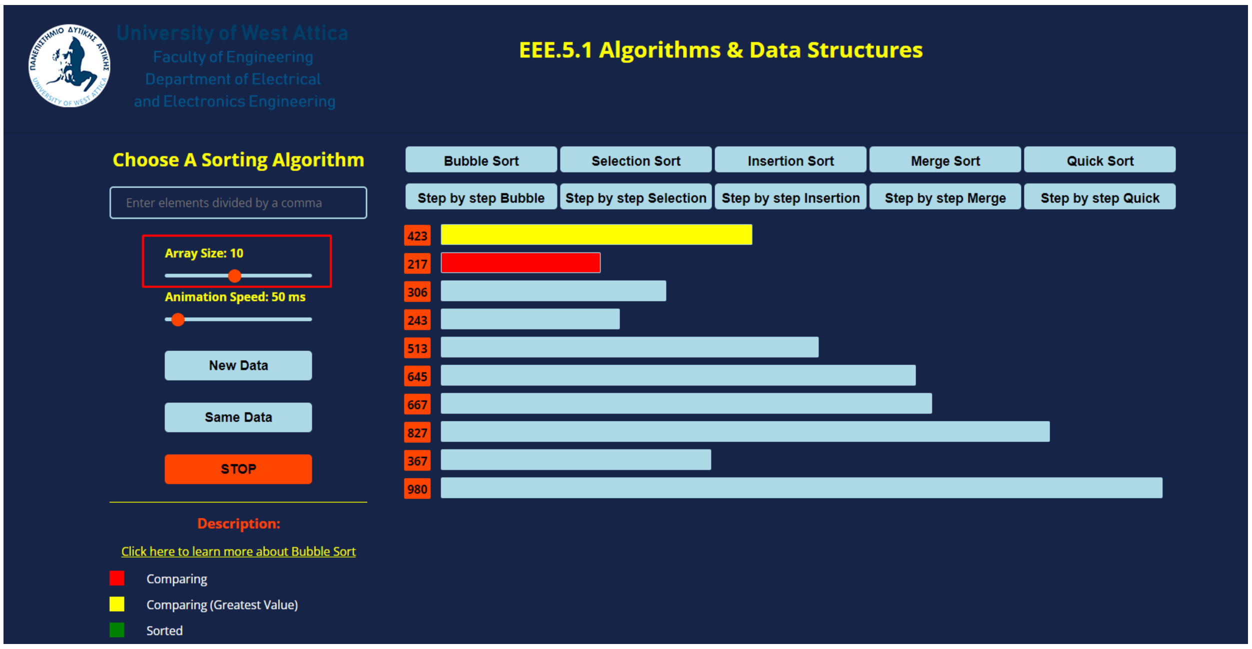


Array Size: 10
Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 4.1: The user input field.

- **Array size slider:** An HTML range element controls the size of the array that is currently displayed from the application. The student has the ability to create an array from just one element up to 20 elements.



Array Size: 10
Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 4.2: The array size slider.

- **Animation speed slider:** An HTML range element controls the animation speed of the algorithm. Higher values mean that the animation speed is slower which gives the student a better understanding of how the algorithm operates, where lower values mean faster animation speed that give a quick glance at the algorithm. The values are displayed in milliseconds (ms).

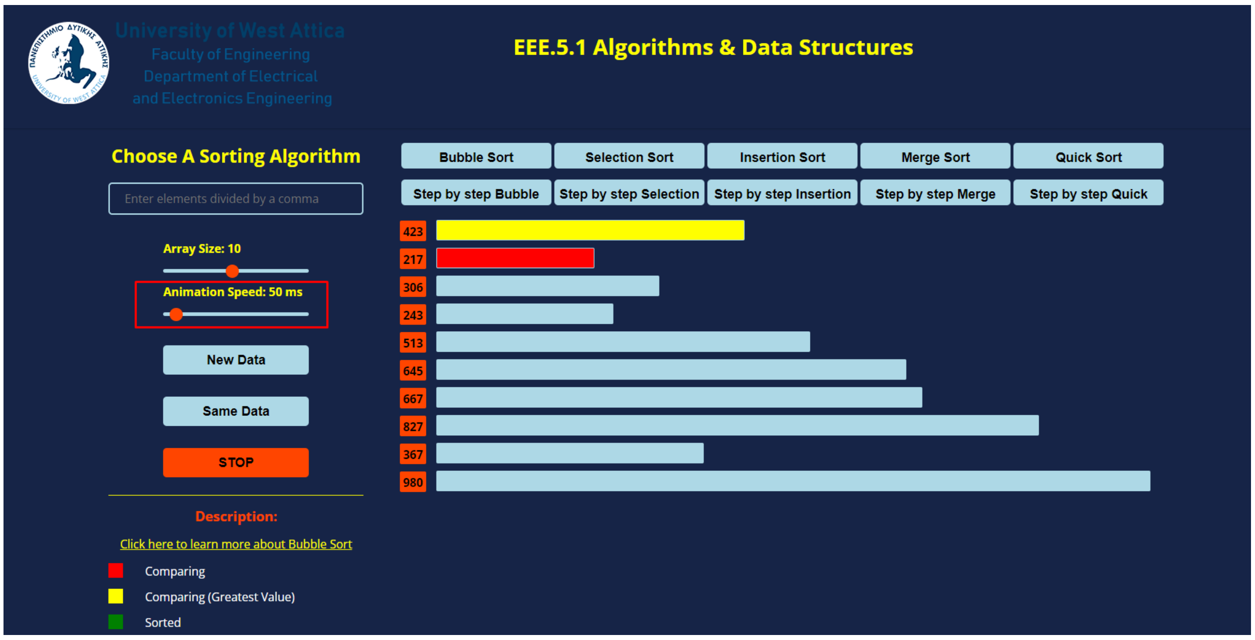


Figure 4.3: The animation speed slider.

- **Algorithm choice buttons:** Each algorithm has its corresponding button that enables it to rum. By clicking on the button with the specified algorithm name on it, the student commands the application to run the specific algorithm on the input data already defined by the value of array size slider and with the specified animation speed defined by the animation speed slider. If the users choose to enter their own input array via the user input field, they can still specify the animation speed but the array size slider remains disabled. By clicking the corresponding algorithm button in this case, the algorithm is tested using the input array that was given.

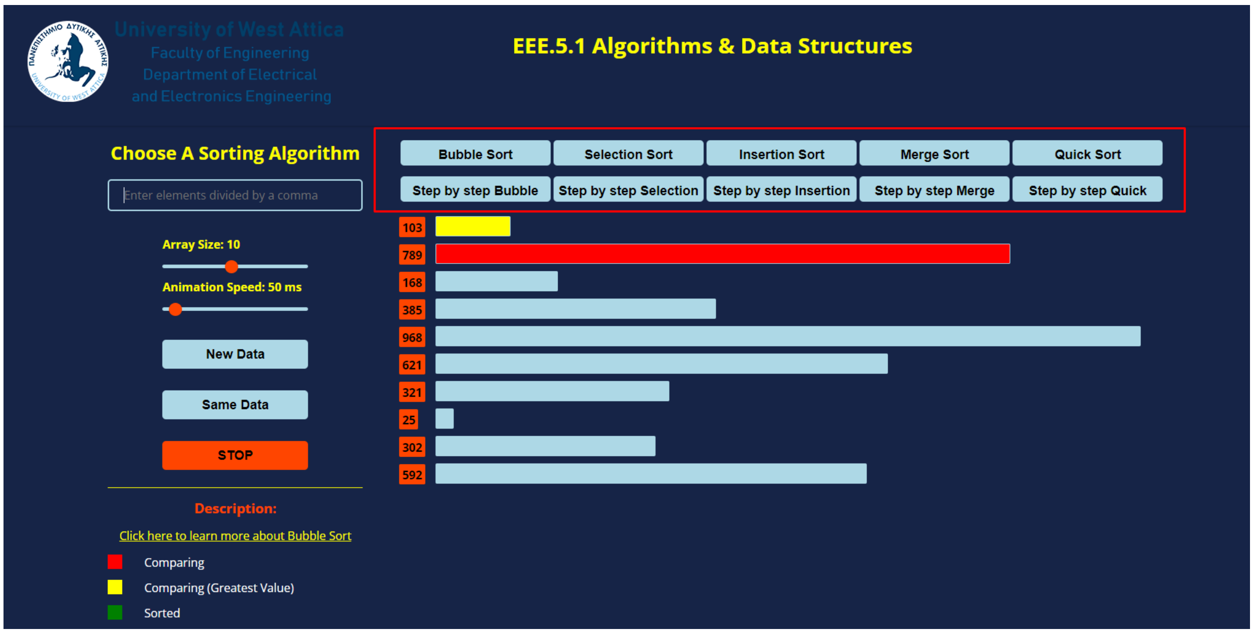


Figure 4.4: The algorithm buttons.

- **Step-by-step buttons**: For better understanding the student has the ability to test each algorithm step-by-step. By clicking the step-by-step button, the algorithm proceeds one iteration at a time. This option gives the student the capability to run the algorithm at his/her own pace and stop the application at each step to manually verify the changes made or not made at this step and compare his/her own results with those of the application, for better understanding of the inner mechanism of the algorithm.
- **New data button:** This button generates new data of the specified input value from the array size slider. New data are automatically generated when reloading the application. The generated values are in the range of 1 to 999.

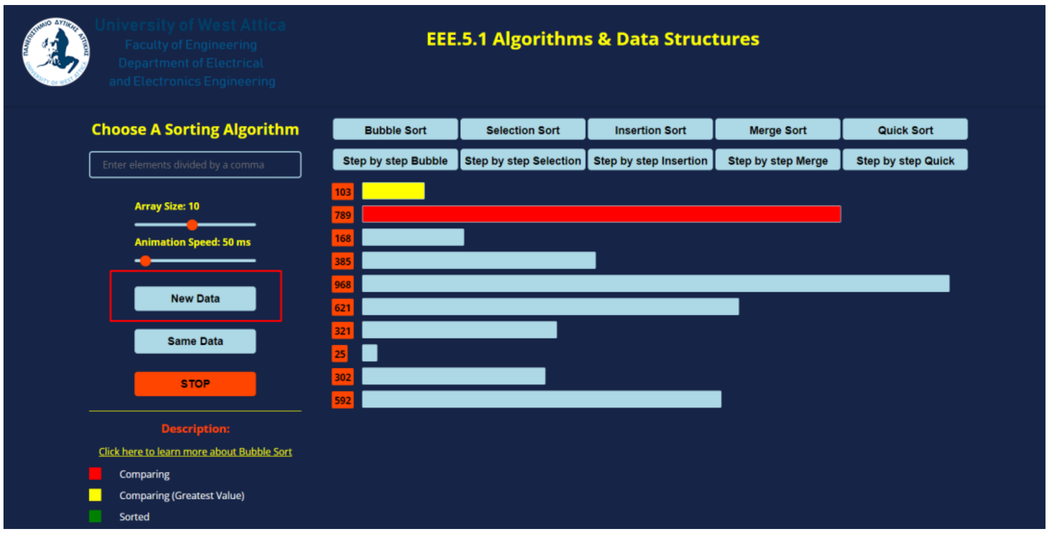


Figure 4.5: The new data button.

- **Same data button:** This button generates and displays again a copy of the original array that was already tested. It gives the student the opportunity to work the algorithm alongside the application. The idea is to first have a quick glance at the algorithm that they want to test, see the output, and then generate the same array again and with the help of the corresponding step-by-step button trace the algorithm accordingly and get the expected result.

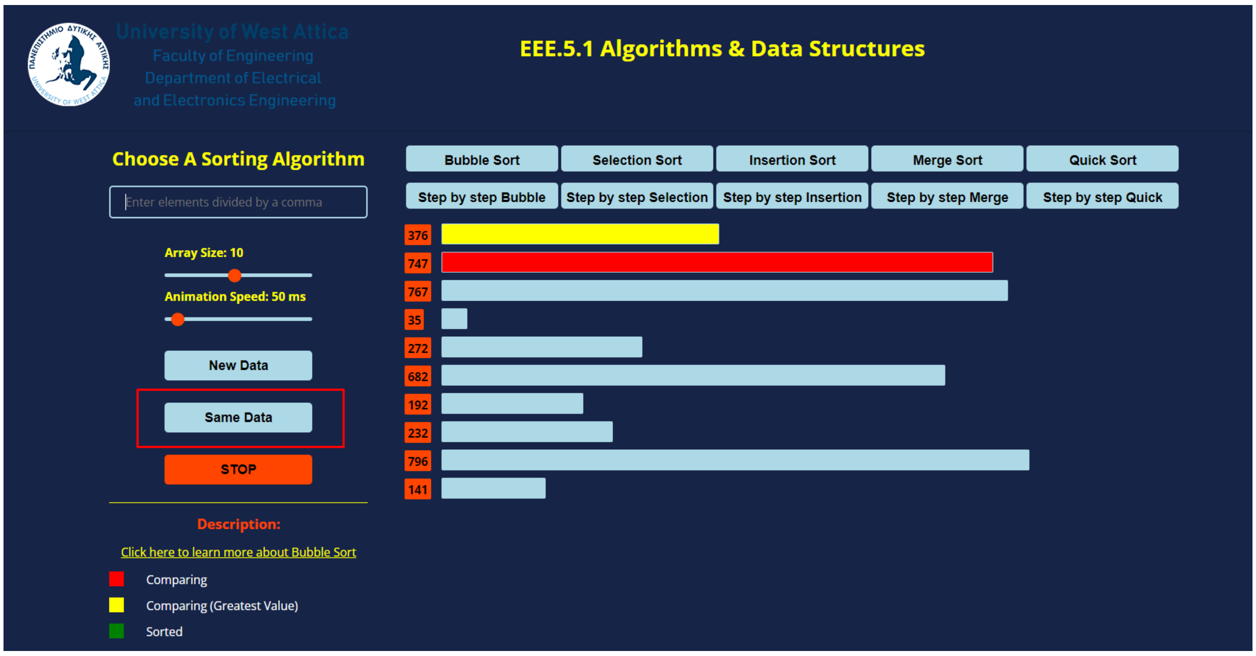


Figure 4.6: The same data button.

- **Stop button:** It allows the student to terminate the current run of an algorithm.

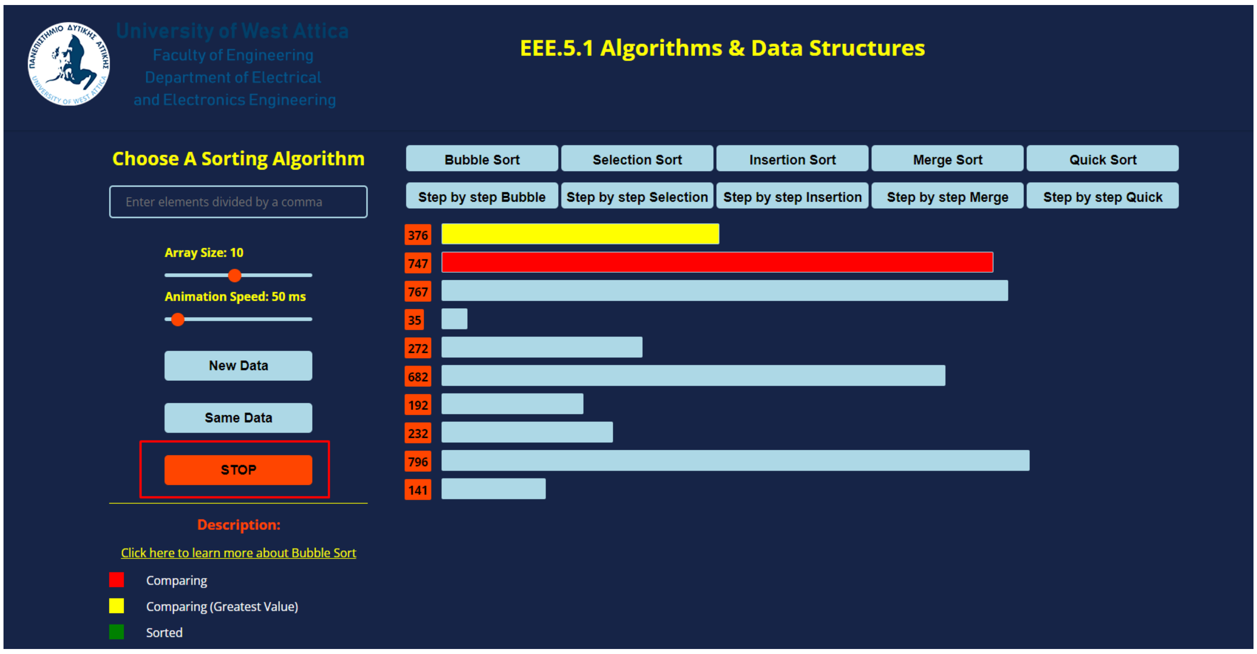


Figure 4.7: The Stop button.

- **Algorithm Description:** While the algorithm is running, the student can click on the link ‘Description’ that is provided on the lower left side of the screen and a helping modal will appear. A short description is provided as well as the time and space complexity of the specified algorithm. By clicking the “Back to Visualizer” button the student returns to the algorithm visualizer application. In conjunction to the algorithm description, a brief description is given as far as the corresponding colors and their meaning that each algorithm uses.

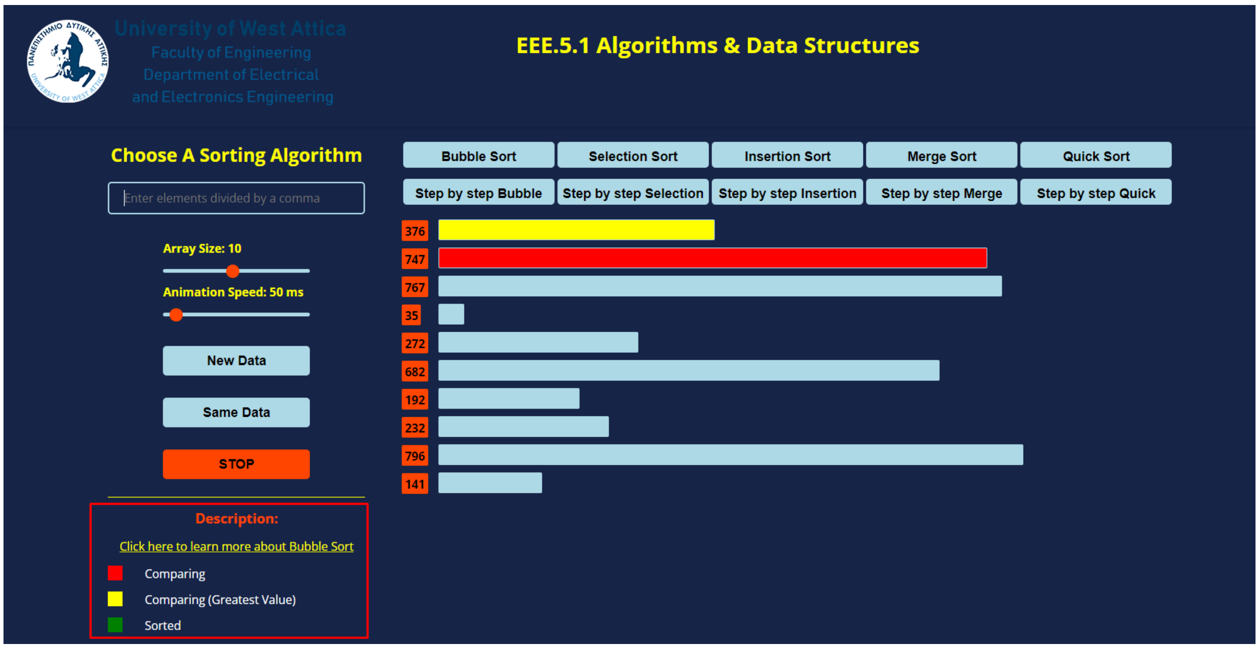


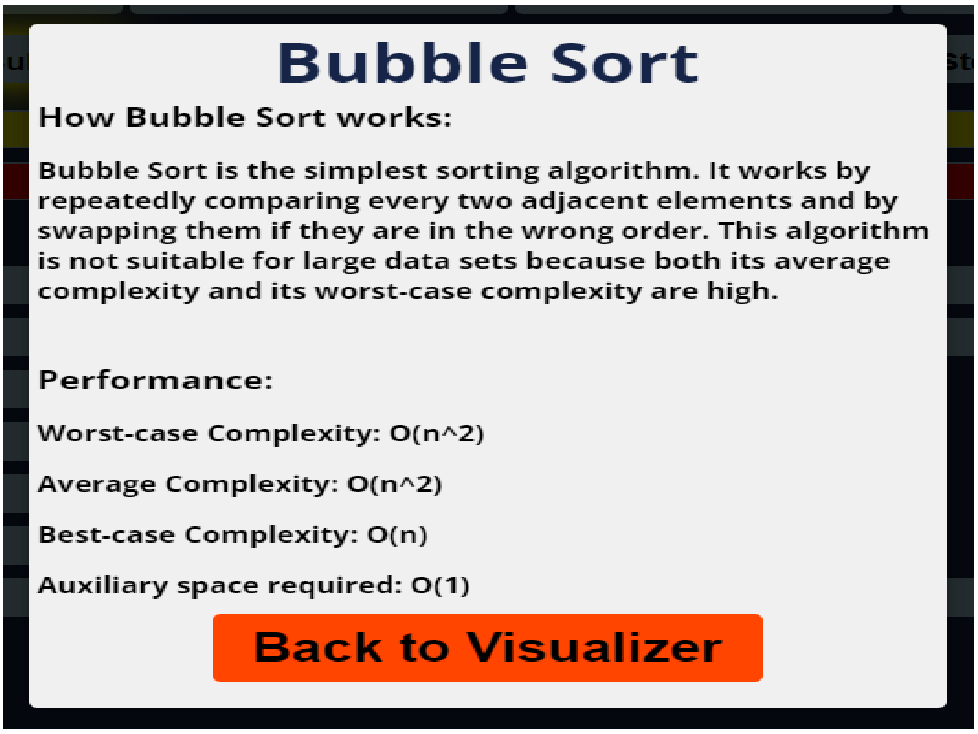
Figure 4.8: The algorithm description area.

The decision to include simplified summaries of each algorithm’s time and space complexity along with brief descriptions of how they operate, was made with the understanding that the visualizing tool is intended to server as a practice and review application for students who have already been taught the subject of algorithms and data structures elsewhere or otherwise.

Given this context, the application’s main objective is neither to introduce students to sorting algorithms or to provide a thorough examination of their features. Instead, it aims to increase students’ comprehension of these algorithms by giving them an interactive, hands-on experience that lets them examine each algorithm in action.

Before using the tool for practice, students can quickly review the key concepts and characteristics of each algorithm with the help of the short summaries of each algorithm’s time and space complexity and concise descriptions of how the algorithm works. This approach assumes that students are already familiar with the material on a basic level and are merely using the tool to supplement their knowledge.

The visualizing application can assist students in concentrating on the practical aspects of working with algorithms and obtaining a better understanding of how they operate in specific circumstances by giving them a clear and easily understood overview of each algorithm’s characteristics. As a result, students will then have a strong basis for further study and research of the topic as well as the ability to work with sorting algorithms with confidence.



Bubble Sort

How Bubble Sort works:

Bubble Sort is the simplest sorting algorithm. It works by repeatedly comparing every two adjacent elements and by swapping them if they are in the wrong order. This algorithm is not suitable for large data sets because both its average complexity and its worst-case complexity are high.

Performance:

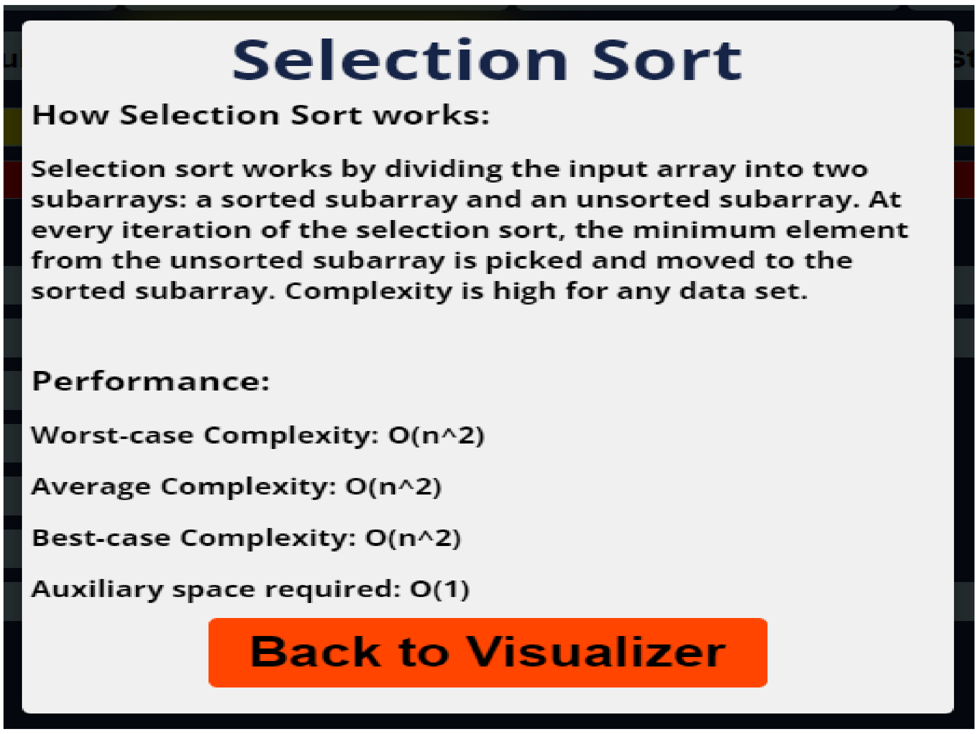
Worst-case Complexity: O(n^2)

Average Complexity: O(n^2)

Best-case Complexity: O(n)

Auxiliary space required: O(1)

Figure 4.9: Bubble sort description summary.



Selection Sort

How Selection Sort works:

Selection sort works by dividing the input array into two subarrays: a sorted subarray and an unsorted subarray. At every iteration of the selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray. Complexity is high for any data set.

Performance:

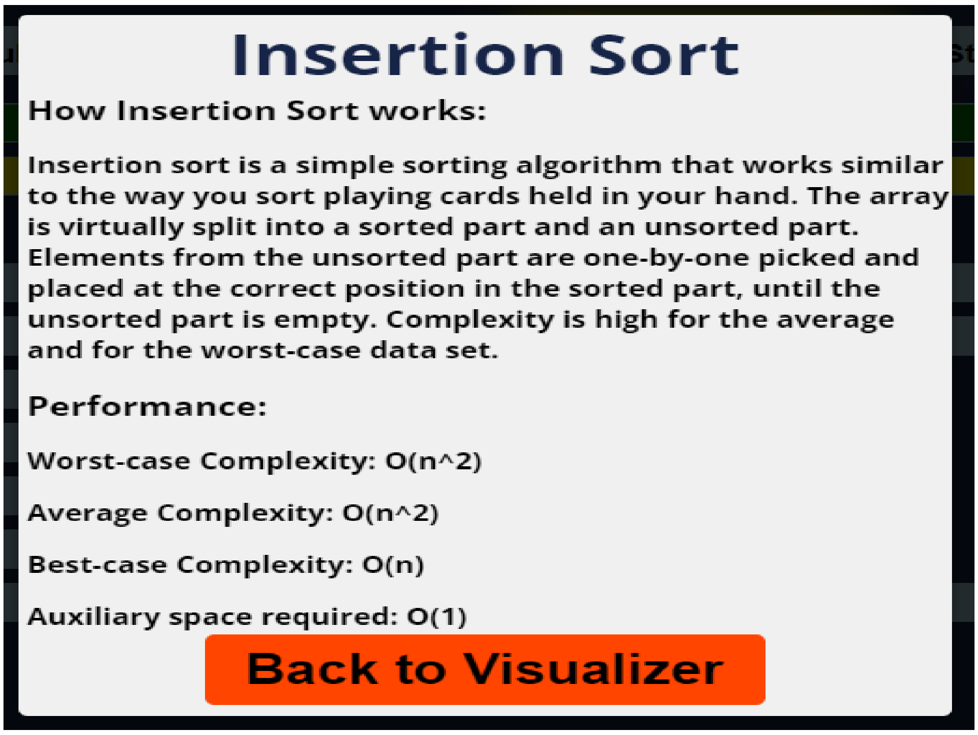
Worst-case Complexity: O(n^2)

Average Complexity: O(n^2)

Best-case Complexity: O(n^2)

Auxiliary space required: O(1)

Figure 4.10: Selection sort description summary.



Insertion Sort

How Insertion Sort works:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards held in your hand. The array is virtually split into a sorted part and an unsorted part. Elements from the unsorted part are one-by-one picked and placed at the correct position in the sorted part, until the unsorted part is empty. Complexity is high for the average and for the worst-case data set.

Performance:

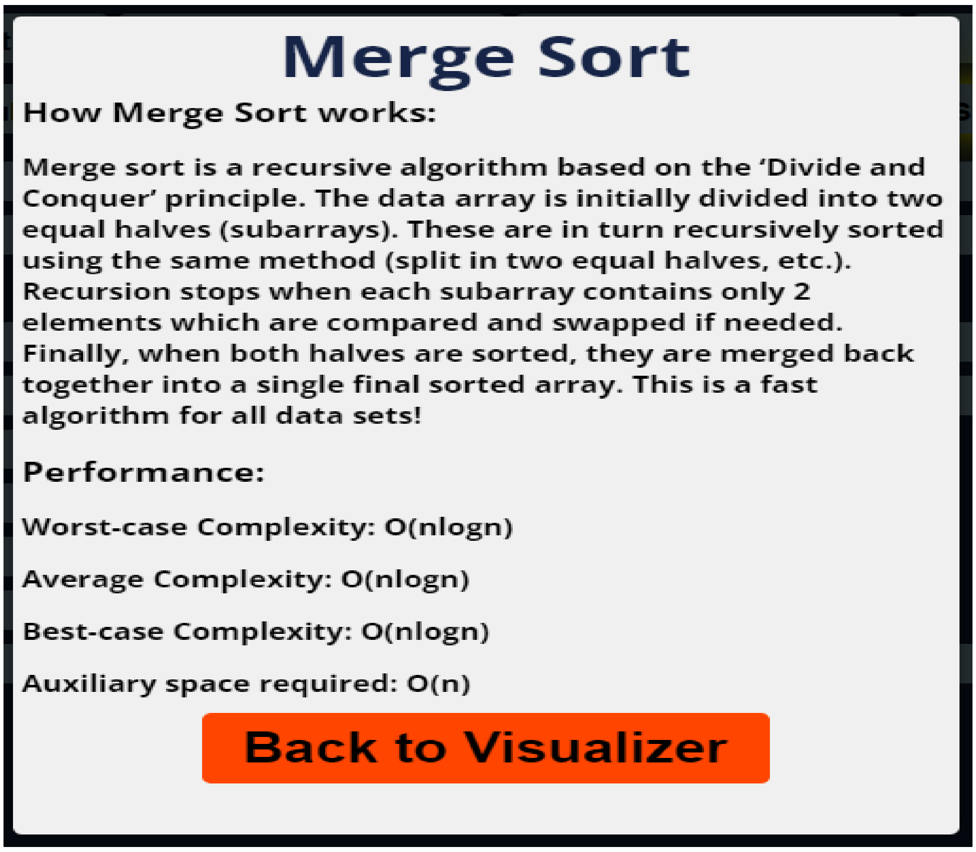
Worst-case Complexity: O(n^2)

Average Complexity: O(n^2)

Best-case Complexity: O(n)

Auxiliary space required: O(1)

Figure 4.11: Insertion sort description summary.



Merge Sort

How Merge Sort works:

Merge sort is a recursive algorithm based on the ‘Divide and Conquer’ principle. The data array is initially divided into two equal halves (subarrays). These are in turn recursively sorted using the same method (split in two equal halves, etc.). Recursion stops when each subarray contains only 2 elements which are compared and swapped if needed. Finally, when both halves are sorted, they are merged back together into a single final sorted array. This is a fast algorithm for all data sets!

Performance:

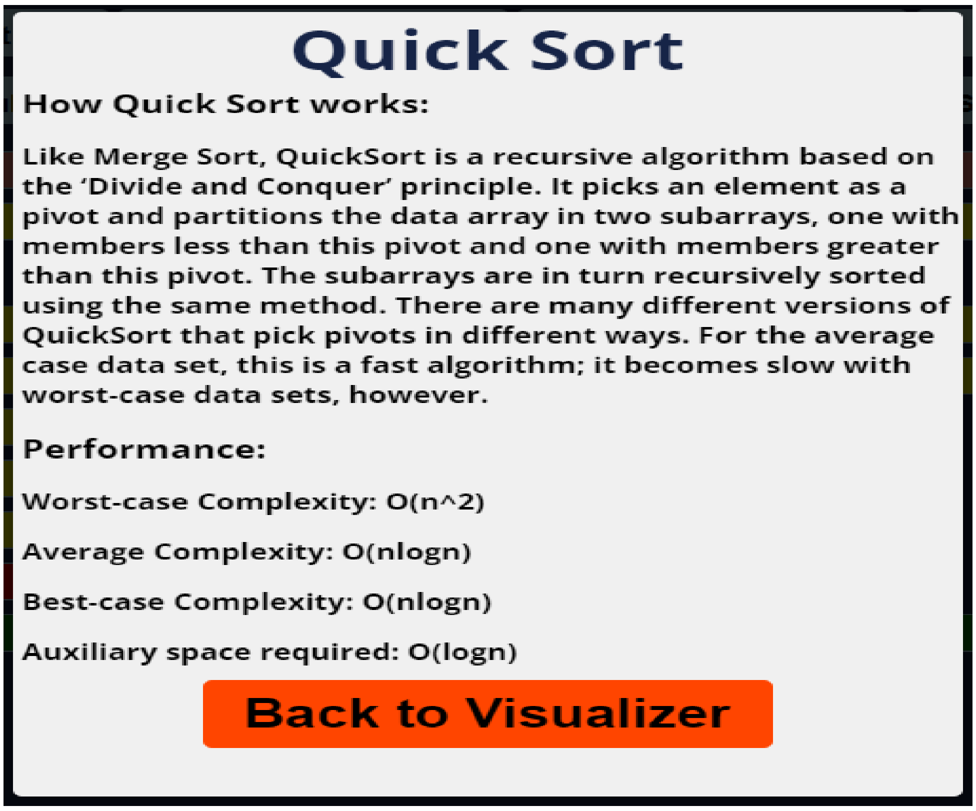
Worst-case Complexity: O(nlogn)

Average Complexity: O(nlogn)

Best-case Complexity: O(nlogn)

Auxiliary space required: O(n)

Figure 4.12: Merge sort description summary.



Quick Sort

How Quick Sort works:

Like Merge Sort, QuickSort is a recursive algorithm based on the ‘Divide and Conquer’ principle. It picks an element as a pivot and partitions the data array in two subarrays, one with members less than this pivot and one with members greater than this pivot. The subarrays are in turn recursively sorted using the same method. There are many different versions of QuickSort that pick pivots in different ways. For the average case data set, this is a fast algorithm; it becomes slow with worst-case data sets, however.

Performance:

Worst-case Complexity: O(n^2)

Average Complexity: O(nlogn)

Best-case Complexity: O(nlogn)

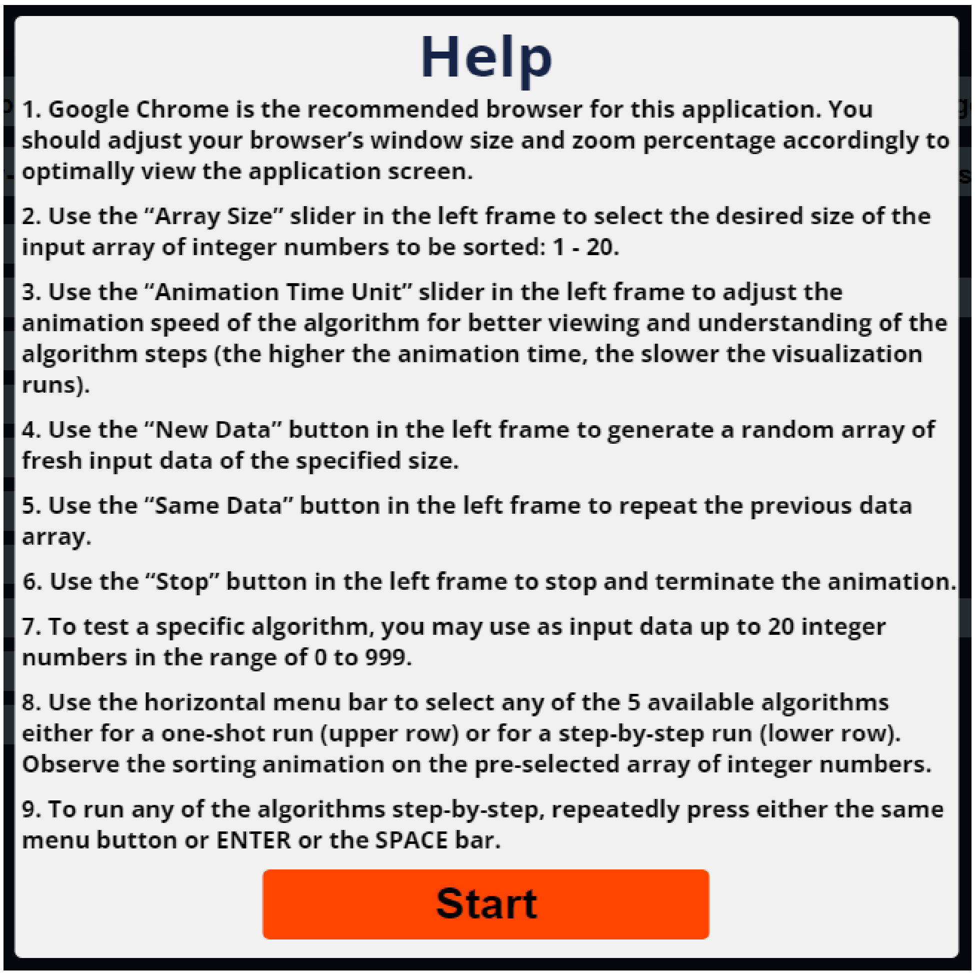
Auxiliary space required: O(logn)

Figure 4.13: Quick sort description summary.

4.3.5 Help modal

Providing a help modal into the sorting algorithms visualizer application is great approach to direct students and make sure they understand exactly how to use the tool. The help modal, which gives users a brief overview of the features contained in the application, is intended to automatically show when the application loads for the first time and every time the page is updated. The help modal provides instructions for using various application buttons as well as recommendations for best practices and appropriate usage. Also, the modal makes sure users receive the information they need before utilizing the tool, which can mitigate misunderstanding and confusion.

The modal is accessible from any screen inside the application with the proper adjustments for better screen coverage and is created to be user-friendly and simple to read. By pressing the start button or elsewhere outside the modal, users can launch the application



Help

1. Google Chrome is the recommended browser for this application. You should adjust your browser’s window size and zoom percentage accordingly to optimally view the application screen.

2. Use the “Array Size” slider in the left frame to select the desired size of the input array of integer numbers to be sorted: 1 - 20.

3. Use the “Animation Time Unit” slider in the left frame to adjust the animation speed of the algorithm for better viewing and understanding of the algorithm steps (the higher the animation time, the slower the visualization runs).

4. Use the “New Data” button in the left frame to generate a random array of fresh input data of the specified size.

5. Use the “Same Data” button in the left frame to repeat the previous data array.

6. Use the “Stop” button in the left frame to stop and terminate the animation.

7. To test a specific algorithm, you may use as input data up to 20 integer numbers in the range of 0 to 999.

8. Use the horizontal menu bar to select any of the 5 available algorithms either for a one-shot run (upper row) or for a step-by-step run (lower row). Observe the sorting animation on the pre-selected array of integer numbers.

9. To run any of the algorithms step-by-step, repeatedly press either the same menu button or ENTER or the SPACE bar.

Figure 4.14: Help modal feature.

4.4 Major Decisions on the Layout and Look-and-Feel of the application

4.4.1 Basic data form

As mentioned earlier, the data used for sorting in this application can be considered as an array of positive integers. The bar chart (or bar graph) is the way of choice here in order to represent such an array. The idea behind this decision is simple: students are familiar with bar graphs. More specifically, bar charts are frequently used in similar courses taken earlier in their studies, such as probability and statistics, calculus etc. and in general bar charts are easily understood by most people. A bar chart is a diagram that represents numerical data in visual form. Each numerical value is represented by a rectangular bar whose length or height is proportional to this value, depending on whether the orientation of the bars on the screen is vertical or horizontal. A glance at a bar chart allows the user to quickly grasp key concepts, such as the variation among the values of the elements of the array. For example, relative value of an array element as compared to any other array element becomes immediately visible through the length of the corresponding bars. Apart from the visual advantages, a bar graph is ideal for an application like that, because of its simplicity and versatility. Indeed, it is very easy to draw and the user may easily compare more than one data sets and make quick and accurate estimations.

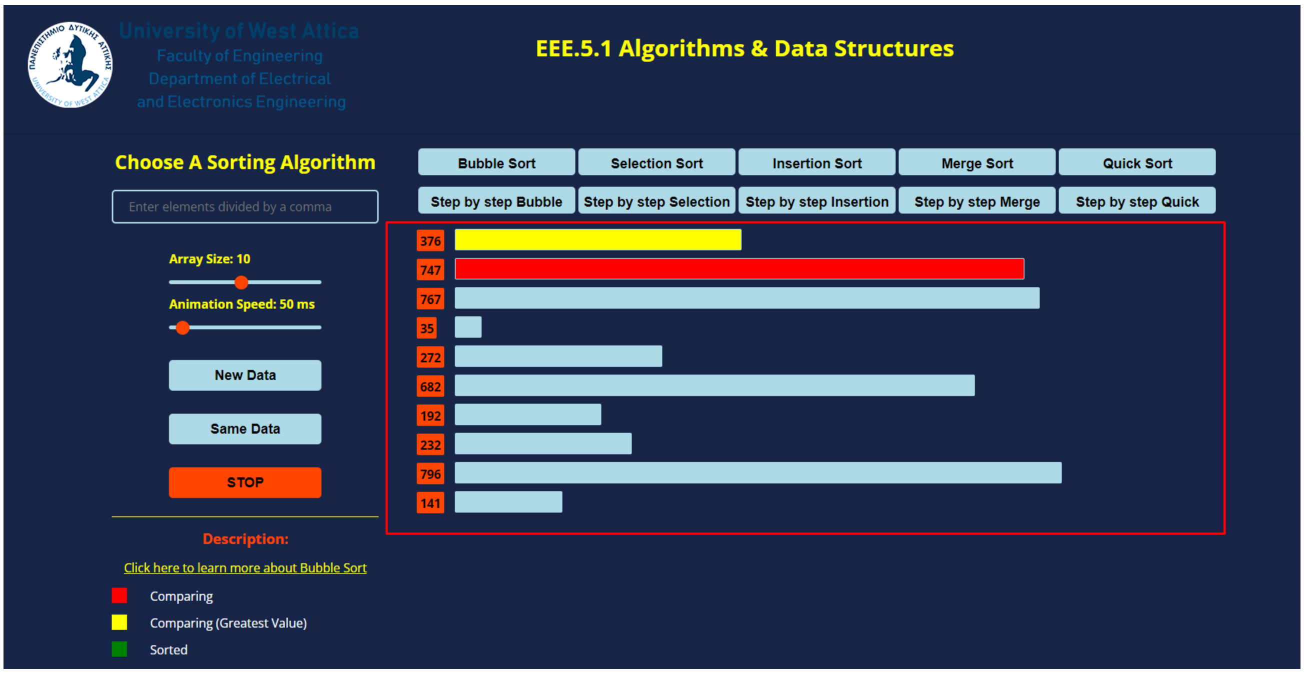


Figure 4.15: The data displayed in a bar chart with horizontal bars, vertically stack.

A drawback of a bar chart is that it requires additional written or oral explanation. The data in a bar chart can be difficult to interpret on its own, so in order for the user to draw the correct conclusions from the diagram, an additional explanation should be provided by the class instructor. As the present interactive application is meant for educational purposes, it is also expected for students to work on the application asynchronously, for better understanding. A narrative modal is provided during the loading phase of the application is employed for this purpose: it plays the role of the oral instructions the students would otherwise receive from the class instructor.

4.4.2 Choice of colors

As far as the styling of the application goes, the choice of colors soon came to question. Something enticing was definitely needed in order to catch students’ attention and also for them to find the time spent on the application enjoyable but so was the engagement part of it. That’s where website color palettes come into play. The main reasons of choosing the following colors were: avoidance of eye fatigue, project the appropriate effect for learning, maximize information retention and stimulate participation. In general, bright colors were chosen:

- Blue: It has a calming effect on the heart rate and respiratory system of students. It encourages a sense of well-being, making it ideal for learning situations that are intensely challenging and cognitively taxing. It is highly suggested that people with intellectual work that requires a high cognitive load are most productive in blue environments.
- Yellow: The main characteristic of yellow color is that it grabs the attention. It generates positive energy and encourages creativity.
- Orange: Orange encourages critical thinking and memory. It is shown that it also has an especially high effect on circulation and the nervous system and increases the oxygen supply to the brain, stimulating mental activity while simultaneously loosening inhibitions.
- Green: It promotes calmness and a sense of relaxation and is great for encouraging long-term concentration. It is the most restful color for the eye and creates a feeling of ease when used in a classroom.

4.4.3 Screen Layout Design

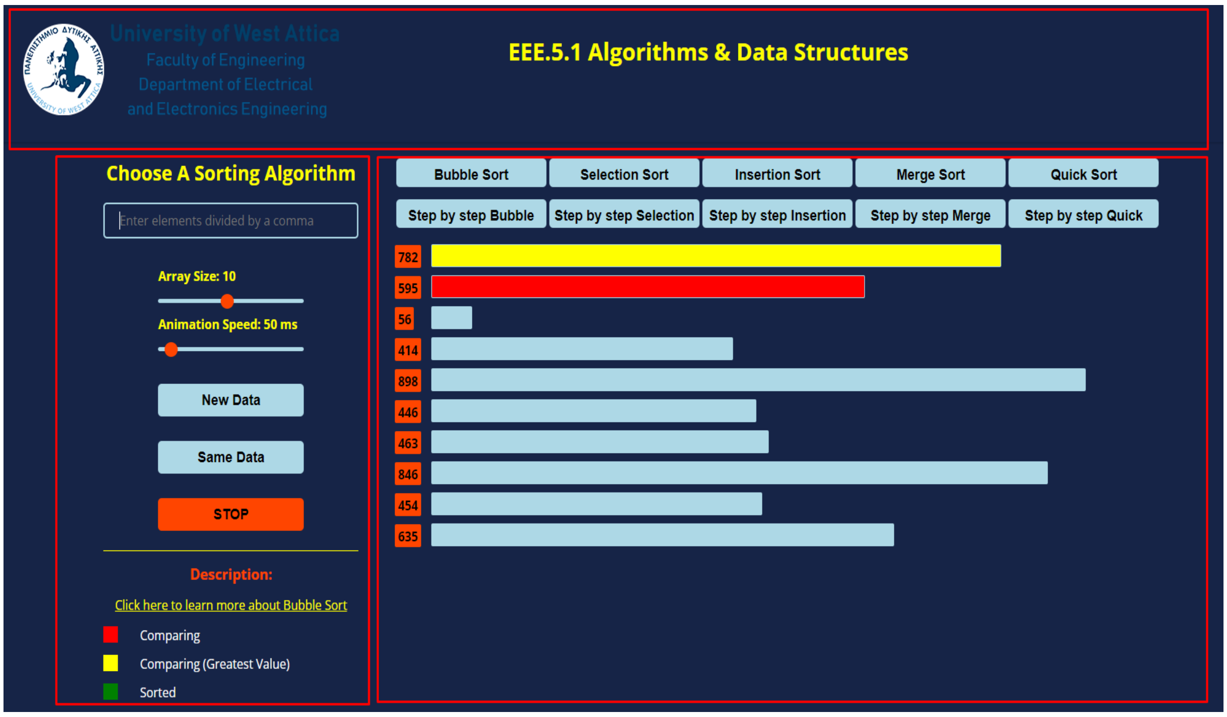
The entry screen design of the application is divided up into three on-screen frames. The UNIWA logo is located on the upper left corner of the screen, which also acts as a visual cue. The course title and code, “*EEE.5.1 Algorithms & Data Structures*” is located in the upper center.

The student has the ability to interact with the application using the control panel that is located below the header and on the left frame of the main part of the screen. An input field for entering data, a slider element to change the array size being sorted, a slider element for adjusting the animation speed of the sorting process, buttons to generate new data of the specified array size or reuse previously displayed data, a button to stop the sorting process and a description of the particular sorting algorithm that takes place at any moment are all found in this control panel. The control panel was positioned on the left region of the screen to keep it apart from the center frame’s depiction of the data being sorted. Because of this separation, the student is able to quickly interact with the control panel without being sidetracked by the data dynamic visual display. To be immediately accessible to the student, the buttons for generating new data, displaying the same data again and stopping process were put below the sliders.

On the center frame, a list of the implemented sorting algorithms is displayed right below the header, above the data display portion of the screen. Each algorithm has two corresponding buttons that provide with student with two choices. Either execute the algorithm continuously until it is finished (sorted), or to be executed for each iteration step-by-step with the student’s control. The reason behind the specific location of the buttons is to maintain a barrier between the control panel and the data visualization. This layout makes it simple for the student to select the desired algorithm without being sidetracked by other screen components. In order to make the visual sorting representation of the data the main emphasis of the screen, it was positioned in the center part of the main frame, in the form of numbers in labels (on the left) and horizontal bars, where the width of each bar has the numerical value of the corresponding element of the array, on the right. This allows the student to immediately observe the changes of each algorithm while it runs thanks to this positioning.

In general, the design of the different regions of the screen was intended to make it simple for the student to interact with the control panel, choose the desired algorithm and view the visual display of the data without being overwhelmed by the screen’s layout

Overall, the screen layout is shown in Figure 4.10.



Array Size: 10

Animation Speed: 50 ms

Description:

Click here to learn more about Bubble Sort

🟥 Comparing

🟨 Comparing (Greatest Value)

🟩 Sorted

Figure 4.16: The overall screen layout.

CHAPTER 5: The sorting algorithms within the visualization application

5.1 Introduction

This particular chapter will present every sorting algorithm of the application. Each element of the array is depicted as a horizontal bar, and the width of each bar defines the value of that element. As it will be discussed for each algorithm specifically, the visualization process it uses different colors to denote the status of the elements. A description is provided on each algorithm to clarify the meaning of each color. The animated swaps between these bars, represent the individual steps of the sorting procedure of each algorithm.

5.2 Bubble Sort

By utilizing different colors, the visualizer helps the student in comprehending the sorting algorithm’s steps as well as how the array’s elements are compared and rearranged. The fact that the bars are gradually turning from blue to green as the array is being sorted simplifies the process for the student to follow the algorithm’s progress as it sorts the array.

The following colors are used in bubble sort:

- Light blue: The initial color of the bars used to represent the elements in the array. This makes it easier for the student in distinguishing between elements that have already been sorted and those that are still through processing.
- Red: This color is utilized to draw attention to the element that the algorithm is currently processing. This makes it easier for the student to monitor the algorithm’s progress and comprehend how it operates.
- Yellow: This color is used to draw attention to the element that is being compared to the current element of the algorithm. This makes it easier for the student to comprehend how the algorithm compares elements and selects which ones to swap.
- Green: This is the last color used to indicate the array has finished sorting.

We will examine the algorithm within the application with an array of five elements. The elements are placed in descending order for a specific reason. Due to the small length of the array and that is for demonstration purposes, unless the array is in descending order, after two or three iterations the array will be sorted already. The algorithm works as follows:

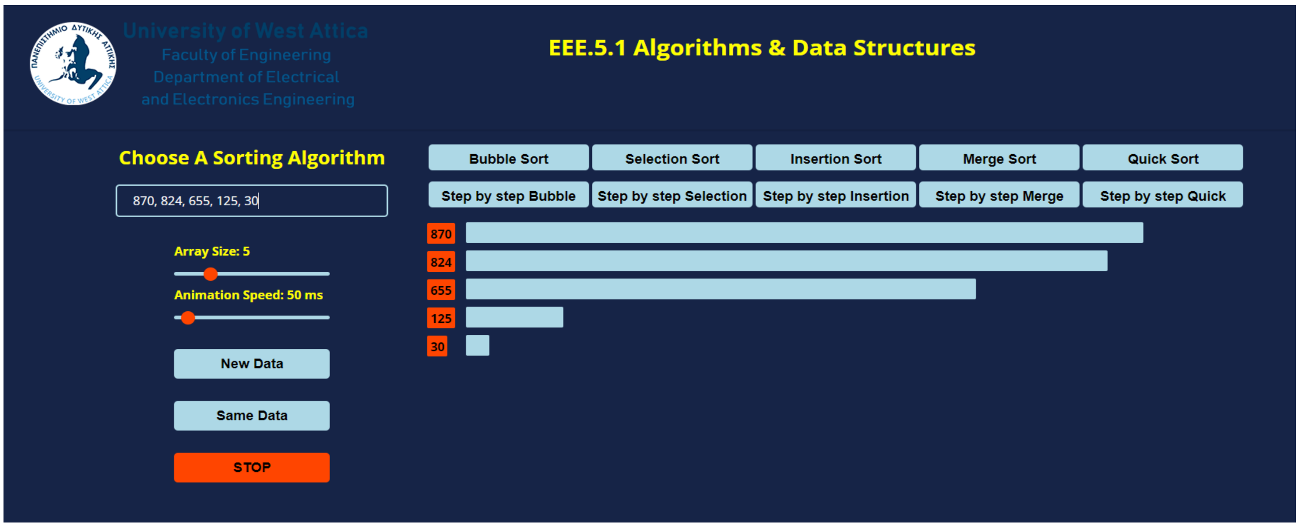


Figure 5.1: BUBBLE SORT: The data array consists of elements: 870, 824, 655, 125, 30.

Starting at the beginning and comparing the first two elements, 870 and 824, would represent the first iteration through the array. Since 870 is greater than 824, they would be swapped. Resulting in the sequence seen below:

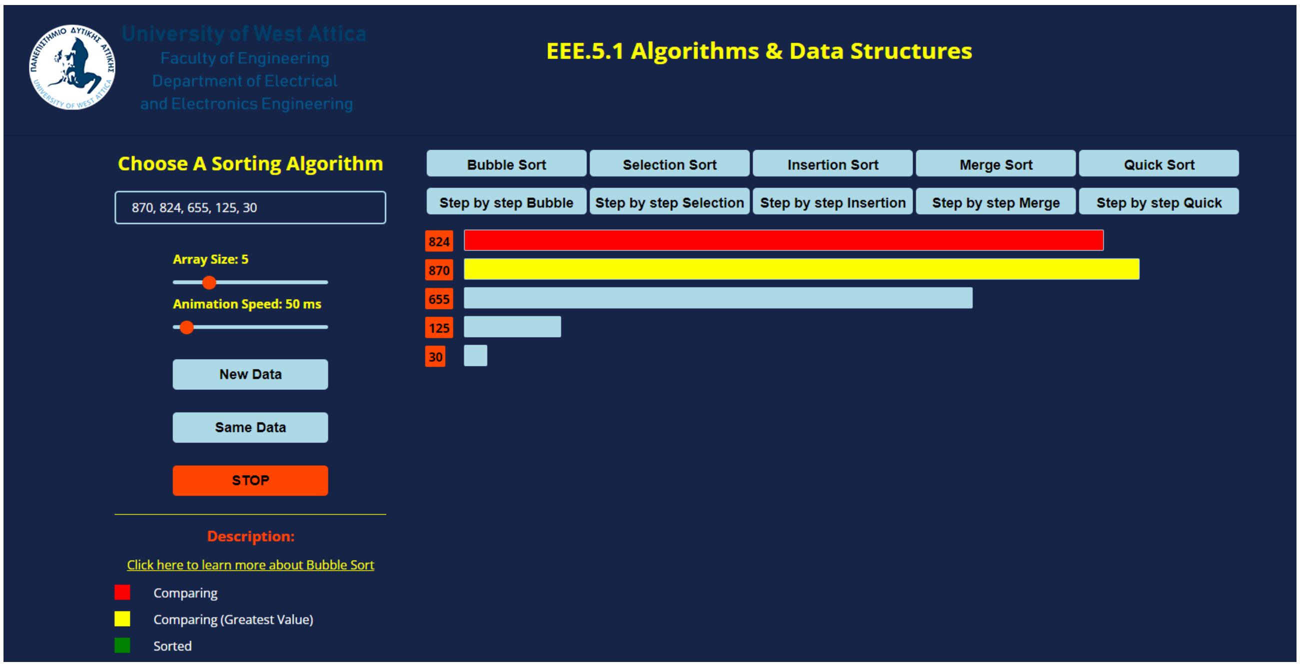
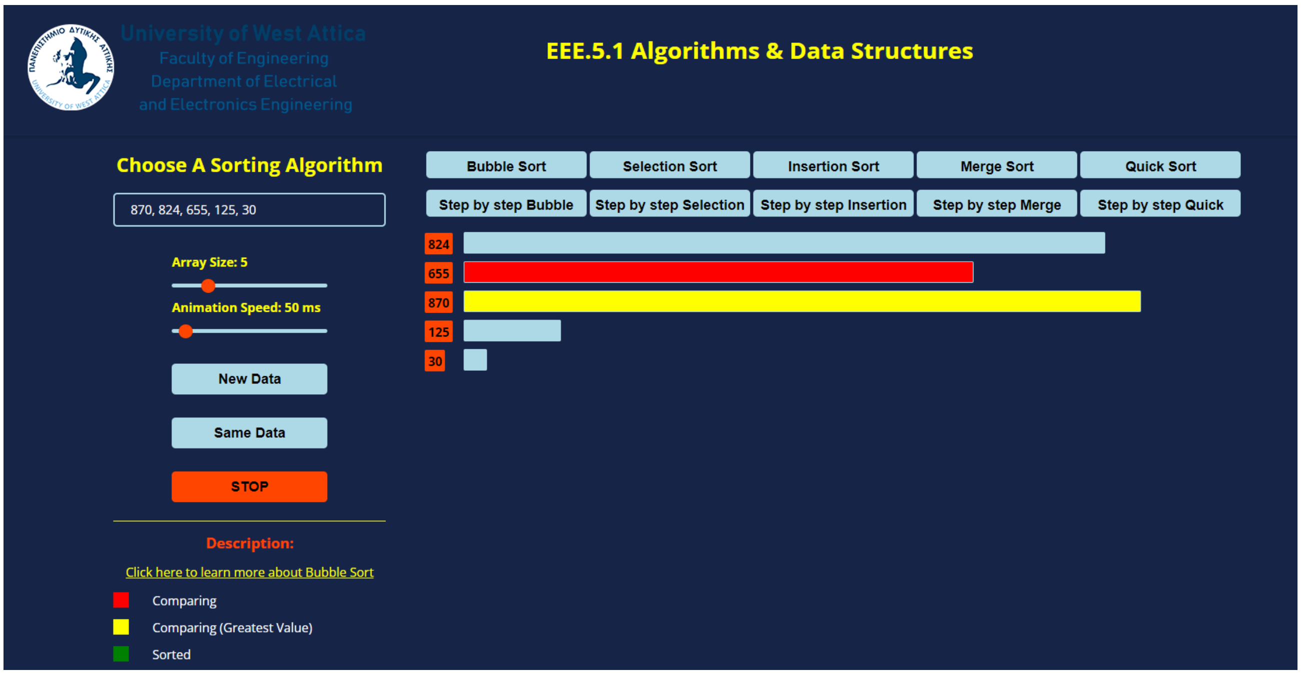


Figure 5.2: BUBBLE SORT: First pass, comparing 870 and 824.

The next step would be to compare 870 and 655. Since 870 is greater than 655, they would be swapped, giving us the following sequence:



870, 824, 655, 125, 30

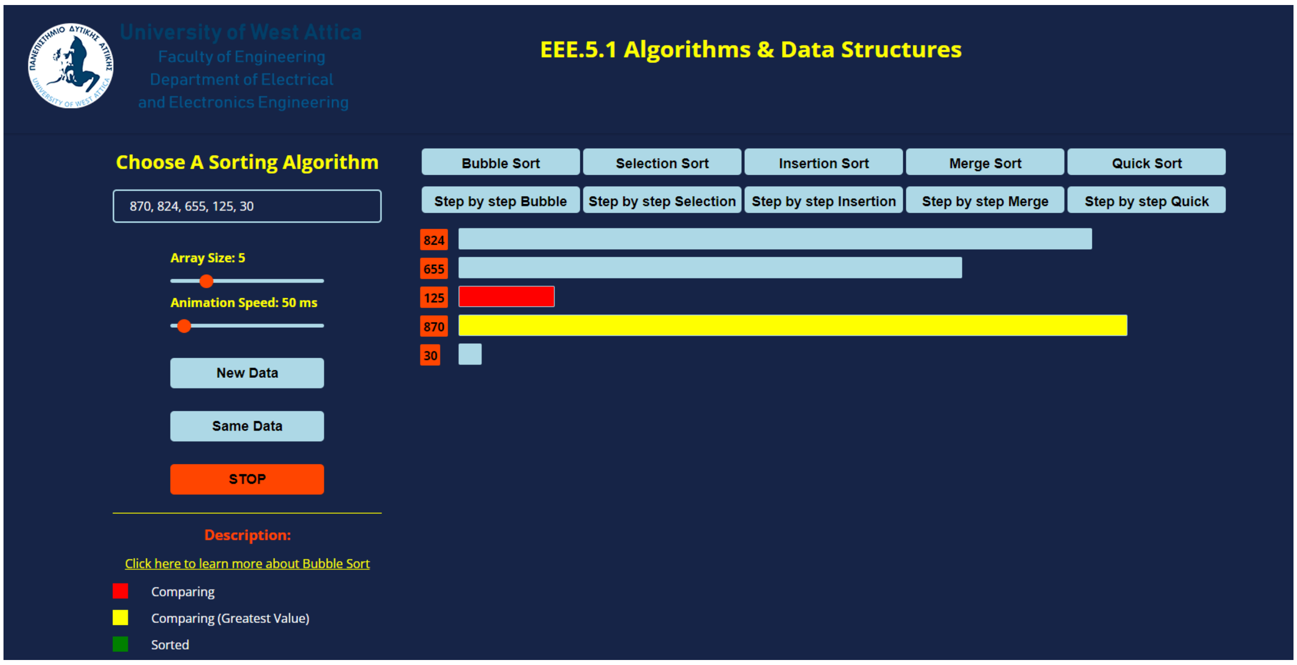
Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.3: BUBBLE SORT: First pass, comparing 870 and 655.

The next step would be to compare 870 and 125. Since 870 is greater than 125, they would be swapped, resulting in the sequence seen below:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.4: BUBBLE SORT: First pass, comparing 870 and 125.

Finally, the last two elements, 870 and 30, would be compared. Since 870 is greater than 30, they would be swapped, giving us the sequence seen below:

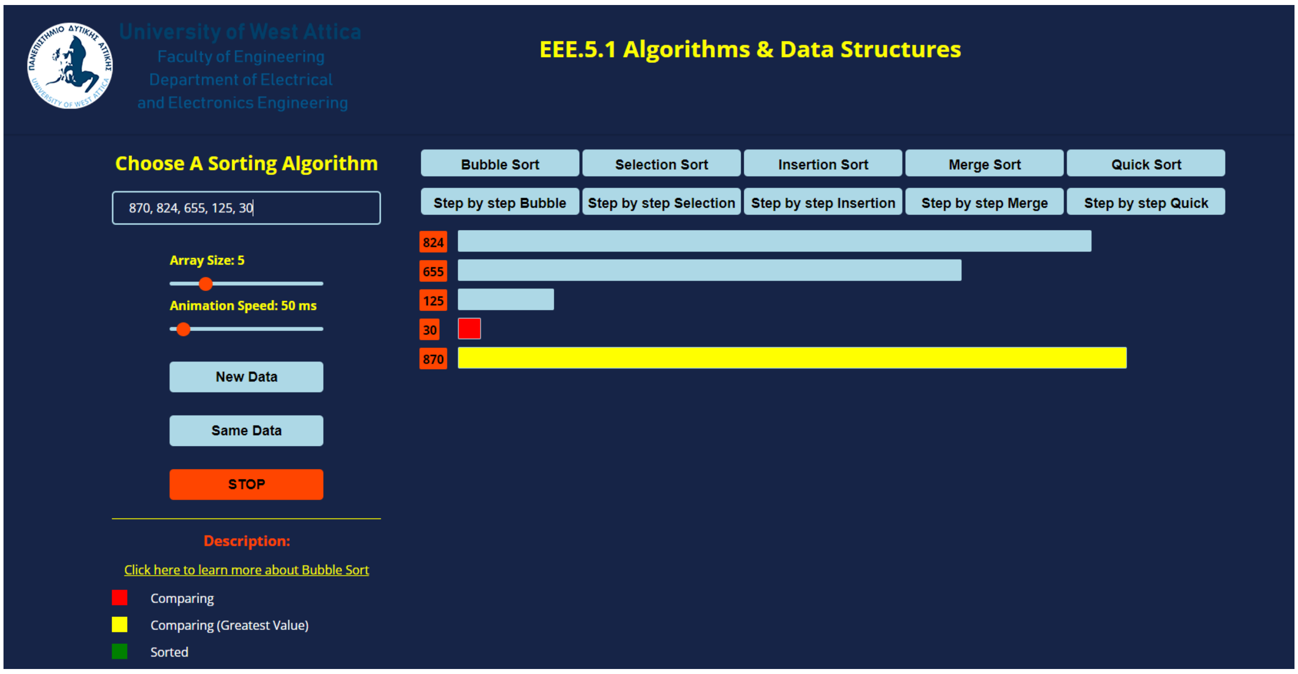


Figure 5.5: BUBBLE SORT: First pass, comparing 870 and 30.

After the first pass, the largest element, 870, is at the end of the array.

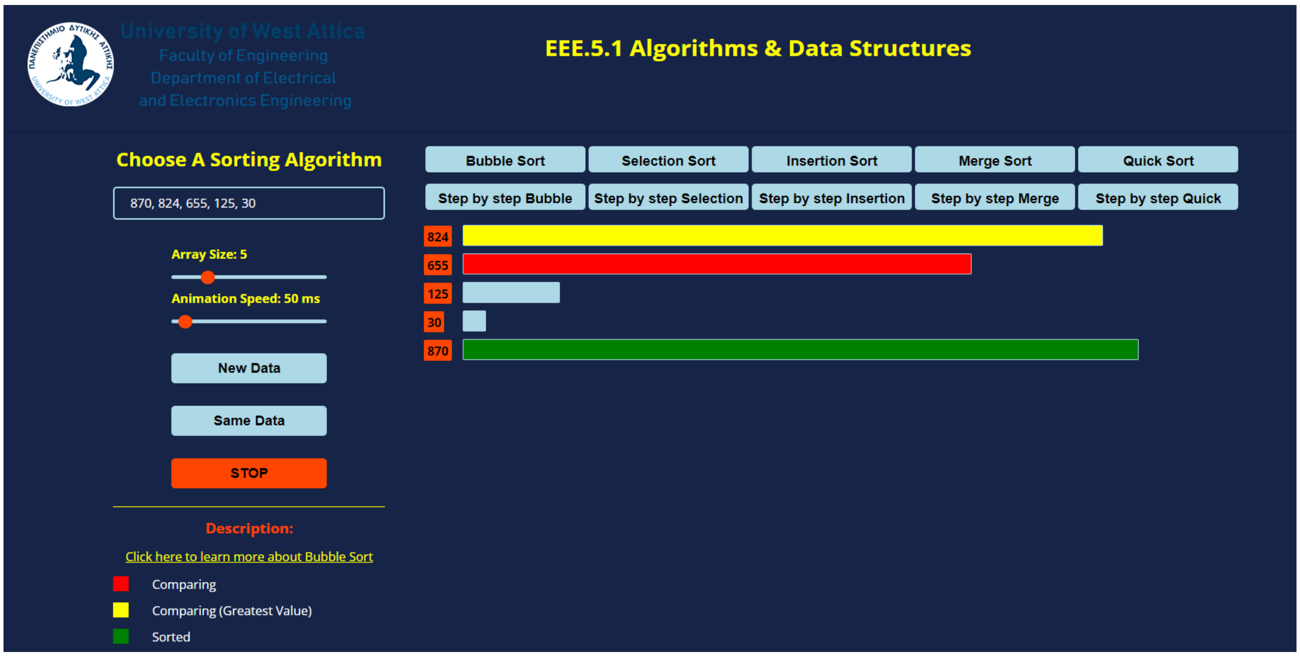
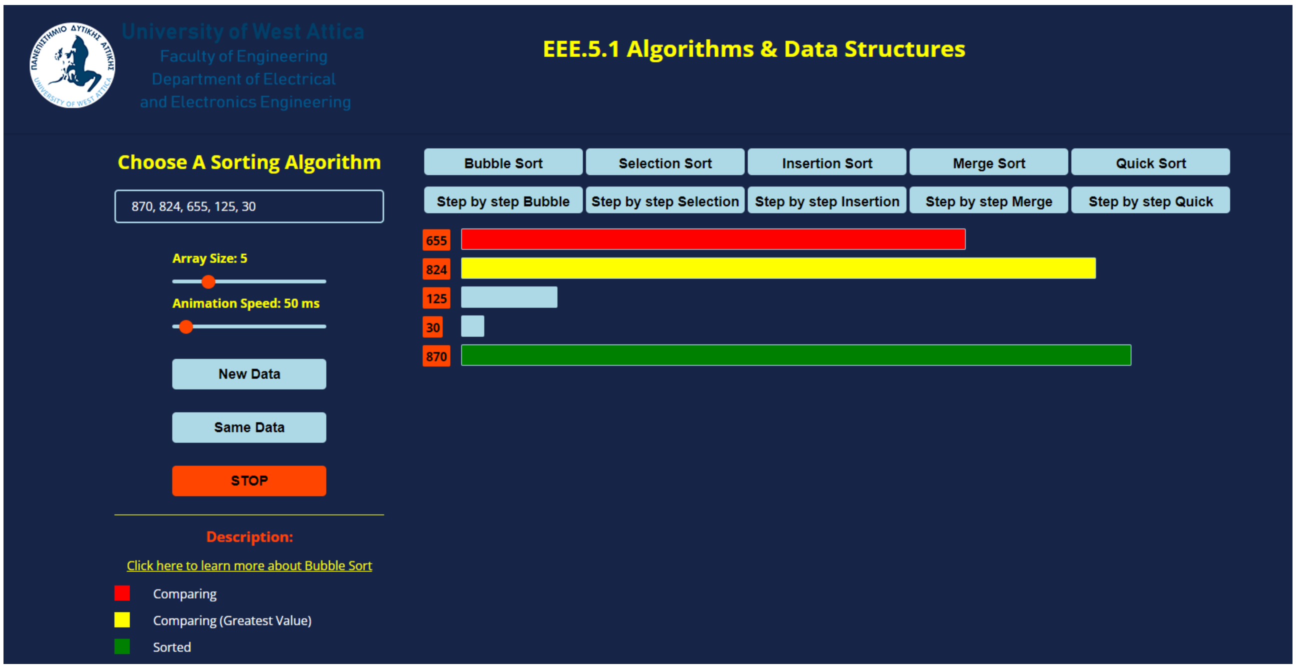


Figure 5.6: BUBBLE SORT: First iteration is over and the green color indicates the sorted portion of the array.

Starting again at the beginning and comparing the first two elements, 824 and 655, would represent the second iteration through the array. Since 824 is greater than 655, they would be swapped. Resulting in the sequence seen below:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

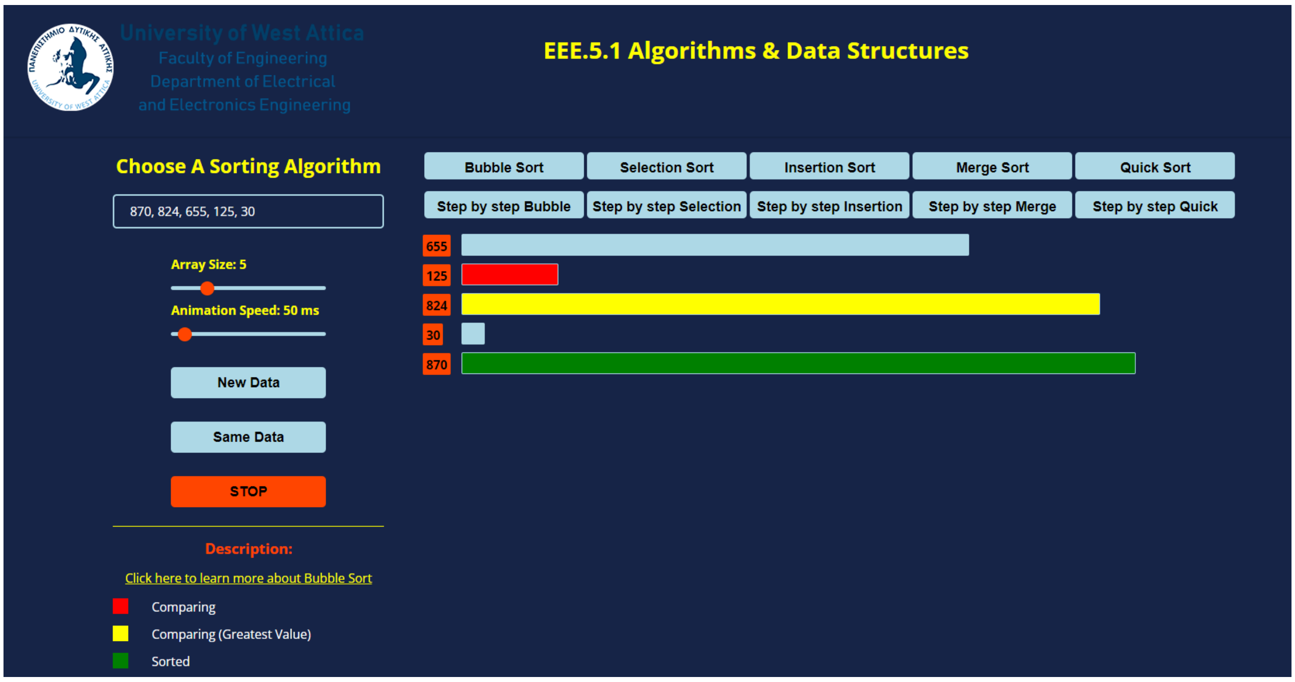
🟥 Comparing

🟨 Comparing (Greatest Value)

🟩 Sorted

Figure 5.7: BUBBLE SORT: Second pass, comparing 824 and 655.

The next step would be to compare 824 and 125. Since 824 is greater than 125, they would be swapped, giving us the following sequence:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

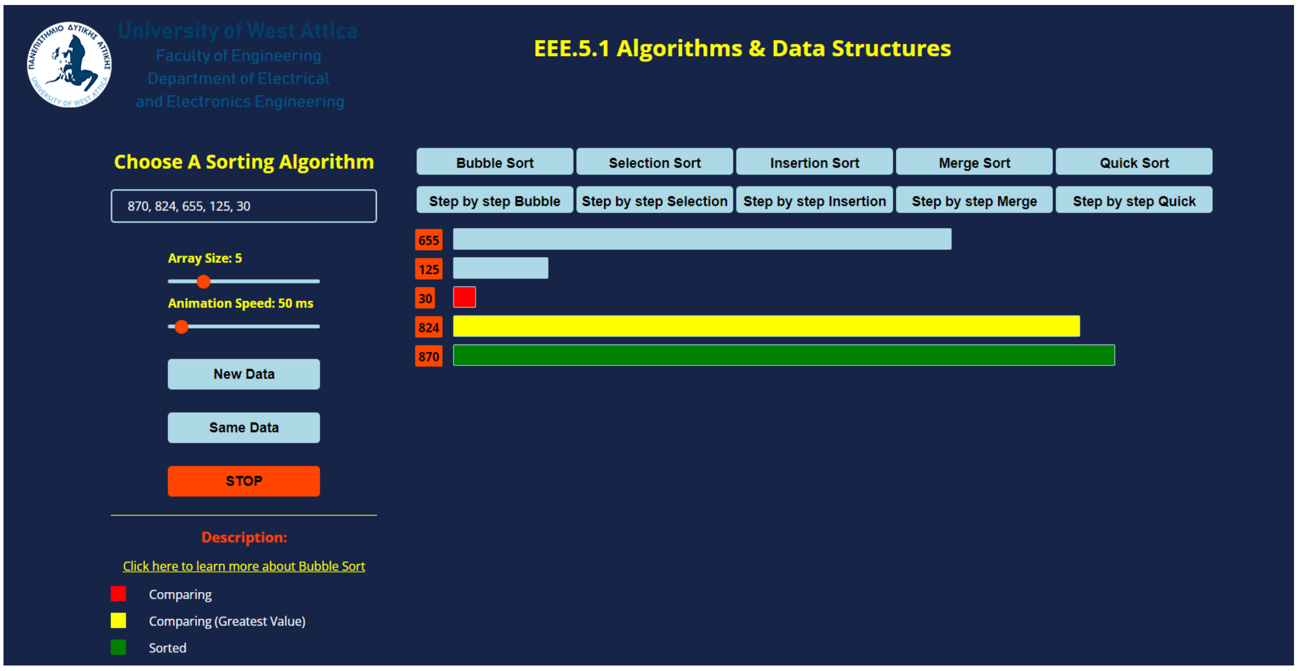
🟥 Comparing

🟨 Comparing (Greatest Value)

🟩 Sorted

Figure 5.8: BUBBLE SORT: Second pass, comparing 824 and 125.

The next step would be to compare 824 and 30. Since 824 is greater than 30, they would be swapped, resulting in the sequence seen below:



870, 824, 655, 125, 30

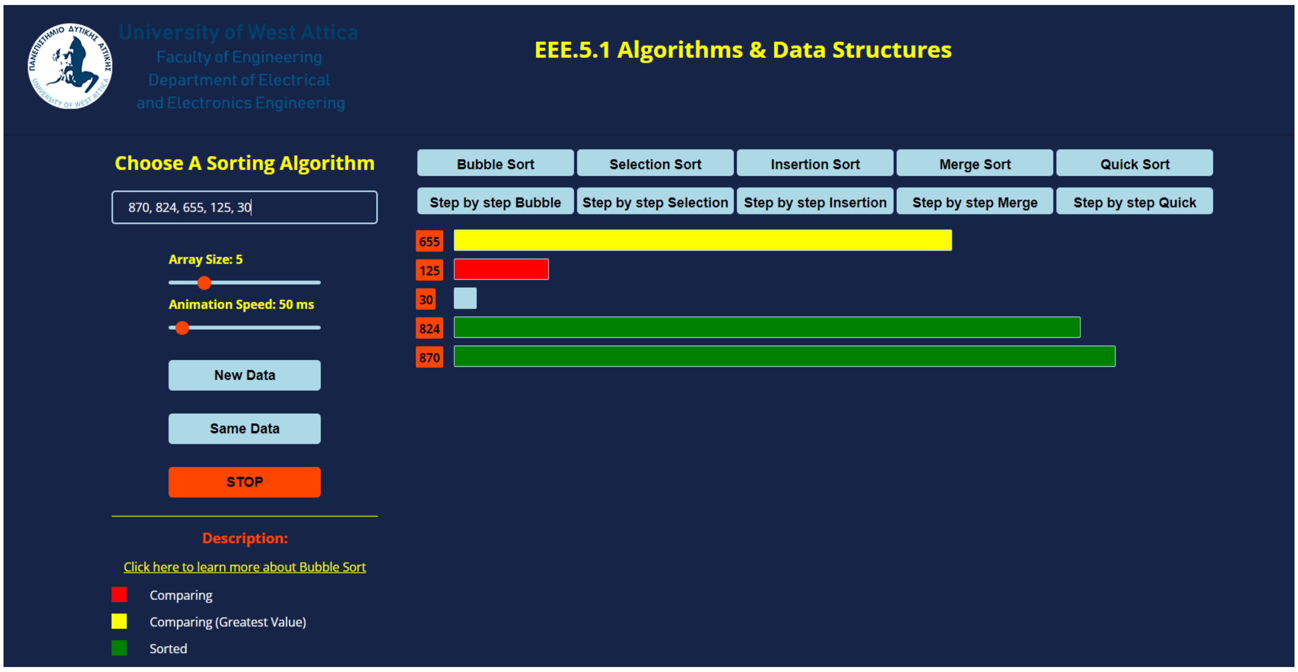
Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.9: BUBBLE SORT: Second pass, comparing 824 and 30.

After the second pass, the second largest element, 824, is at the end of the array.



870, 824, 655, 125, 30

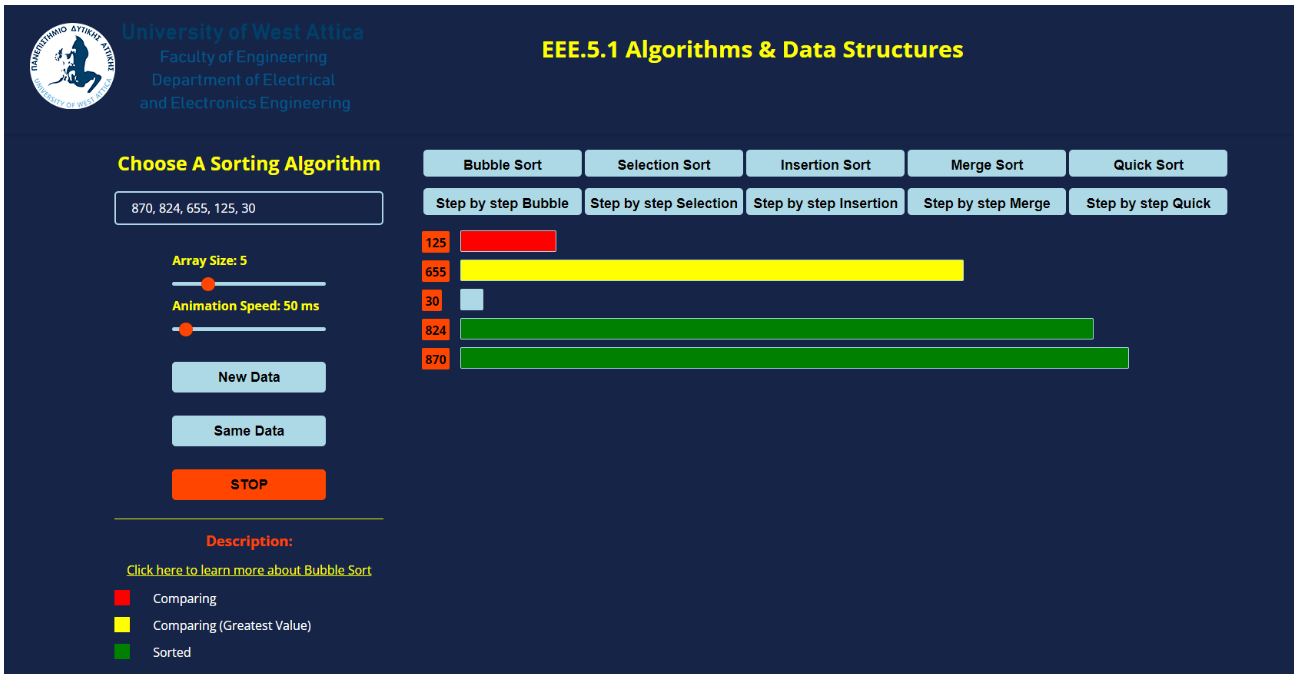
Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.10: BUBBLE SORT: Second iteration is over and the green color indicates the sorted portion of the array.

Starting again at the beginning and comparing the first two elements, 655 and 125, would represent the third iteration through the array. Since 655 is greater than 125, they would be swapped, giving us the following sequence:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

Description:

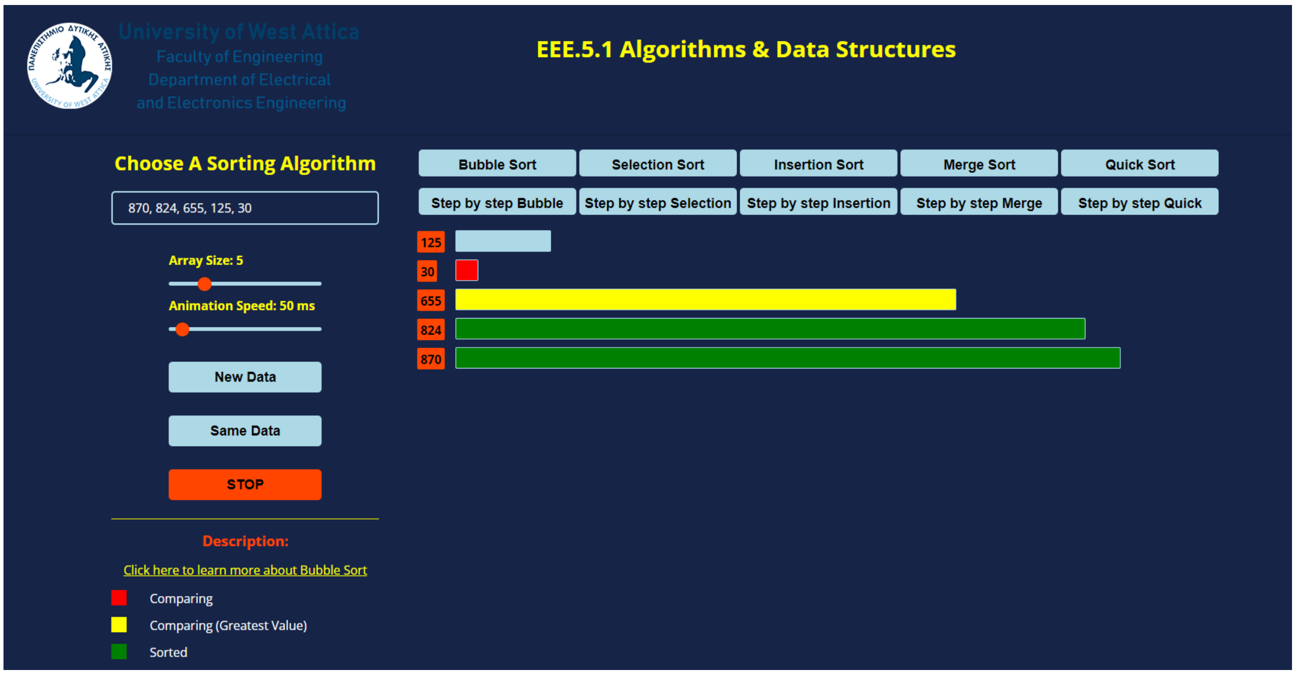
🟥 Comparing

🟨 Comparing (Greatest Value)

🟩 Sorted

Figure 5.11: BUBBLE SORT: Third pass, comparing 655 and 125.

The next step would be to compare 655 and 30. Since 824 is greater than 30, they would be swapped, resulting in the sequence seen below:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

Description:

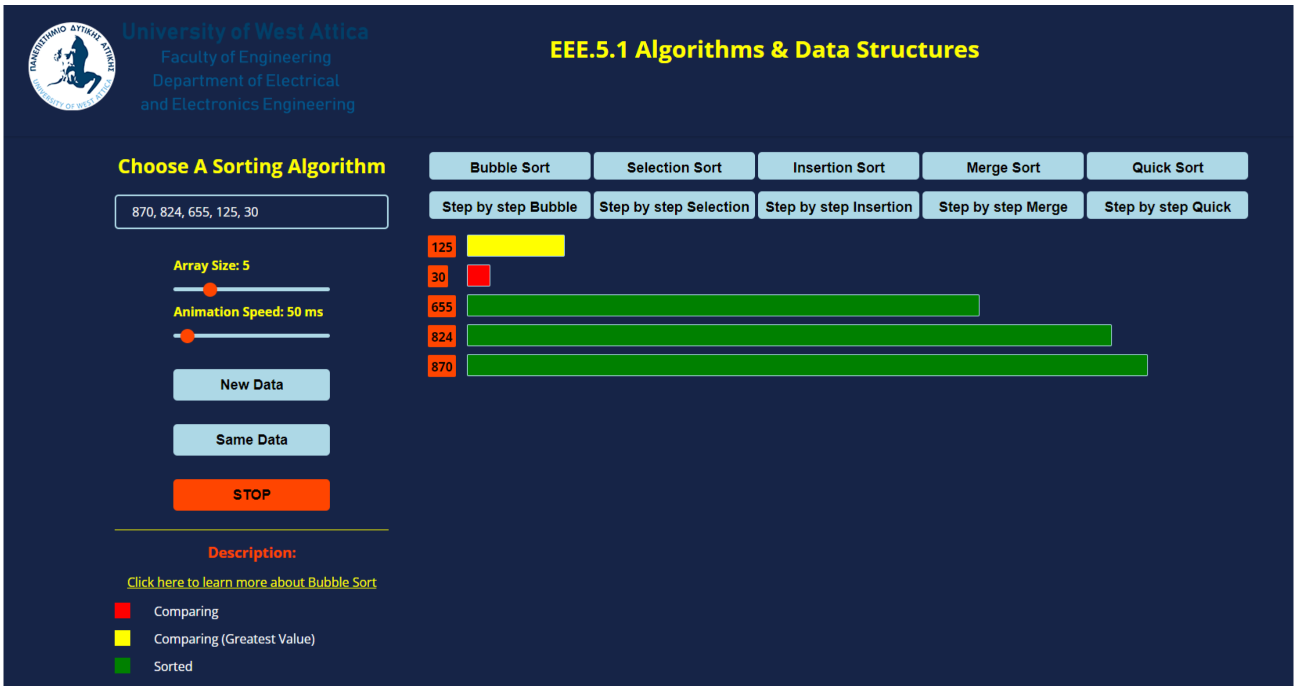
🟥 Comparing

🟨 Comparing (Greatest Value)

🟩 Sorted

Figure 5.12: BUBBLE SORT: Third pass, comparing 655 and 30.

After the third pass, the third largest element, 655, is at the end of the array.



870, 824, 655, 125, 30

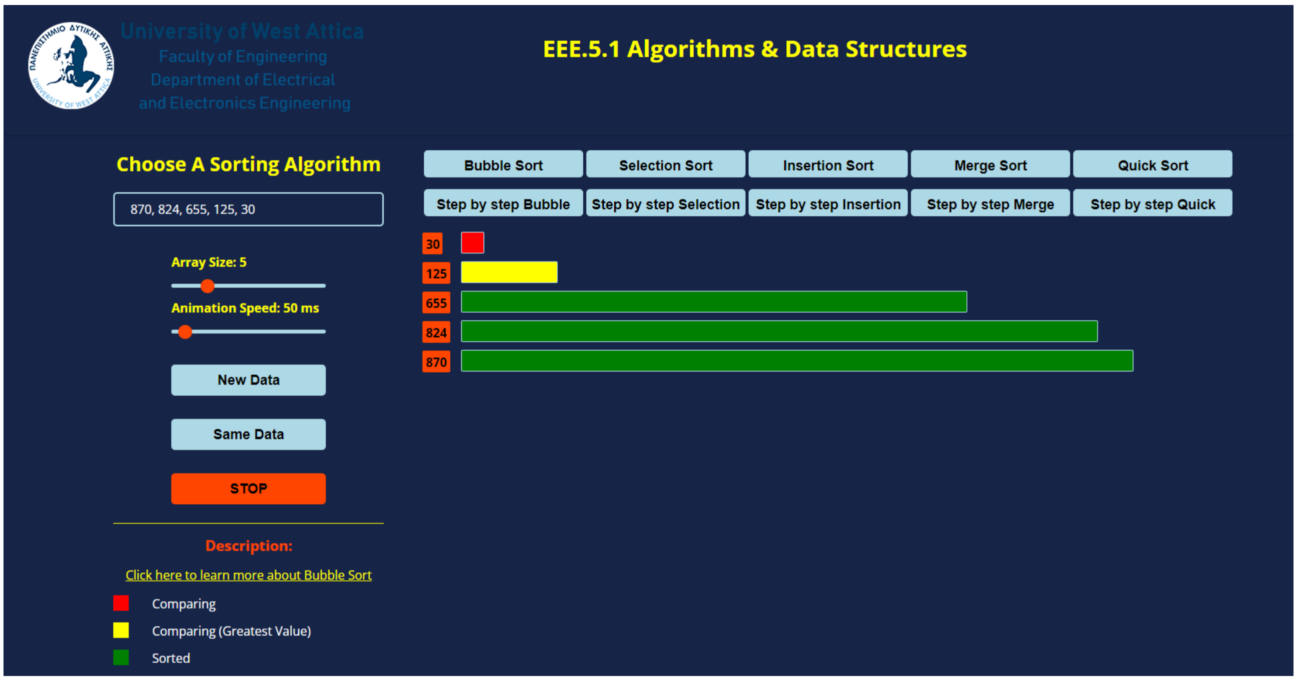
Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.13: BUBBLE SORT: Third iteration is over and the green color indicates the sorted portion of the array.

Starting again at the beginning and comparing the first two elements, 125 and 30, would represent the fourth and final iteration through the array. Since 125 is greater than 30, they would be swapped, giving us the following sequence:



870, 824, 655, 125, 30

Array Size: 5

Animation Speed: 50 ms

🟥 Comparing
🟨 Comparing (Greatest Value)
🟩 Sorted

Figure 5.14: BUBBLE SORT: Fourth pass, comparing 125 and 30.

At this point, no more swaps are needed, so the algorithm would terminate and the array would be regarded as having been sorted in ascending order as shown in Figure 5.15.

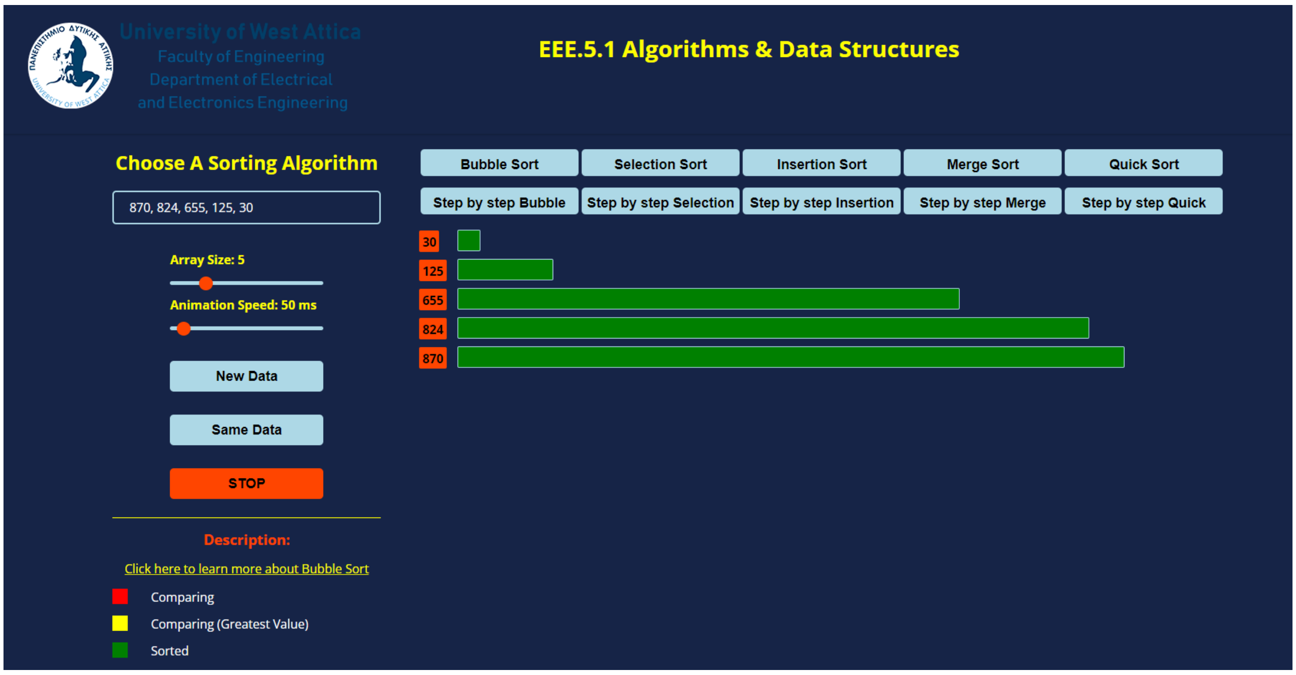


Figure 5.15: BUBBLE SORT: The final sorted array.

5.3 Selection Sort

The following colors are used in selection sort:

- Light blue: The initial color of the bars used to represent the elements in the array. This makes it easier for the student in distinguishing between elements that have already been sorted and those that are still through processing.
- Red: The element with the lowest value in the current iteration of the inner loop is denoted by the color red. This makes it easier for the student to monitor which element is being compared to the array’s sorted portion and thus will be positioned correctly.
- Yellow: The smallest element’s placement is indicated by the color yellow.
- Green: The green color is used to indicate that an element is sorted. When an element is position correctly, it is said to be sorted and is no longer a part of the array’s unsorted portion.

We will analyze the application’s algorithm using a ten-element array. This is how the algorithm operates:

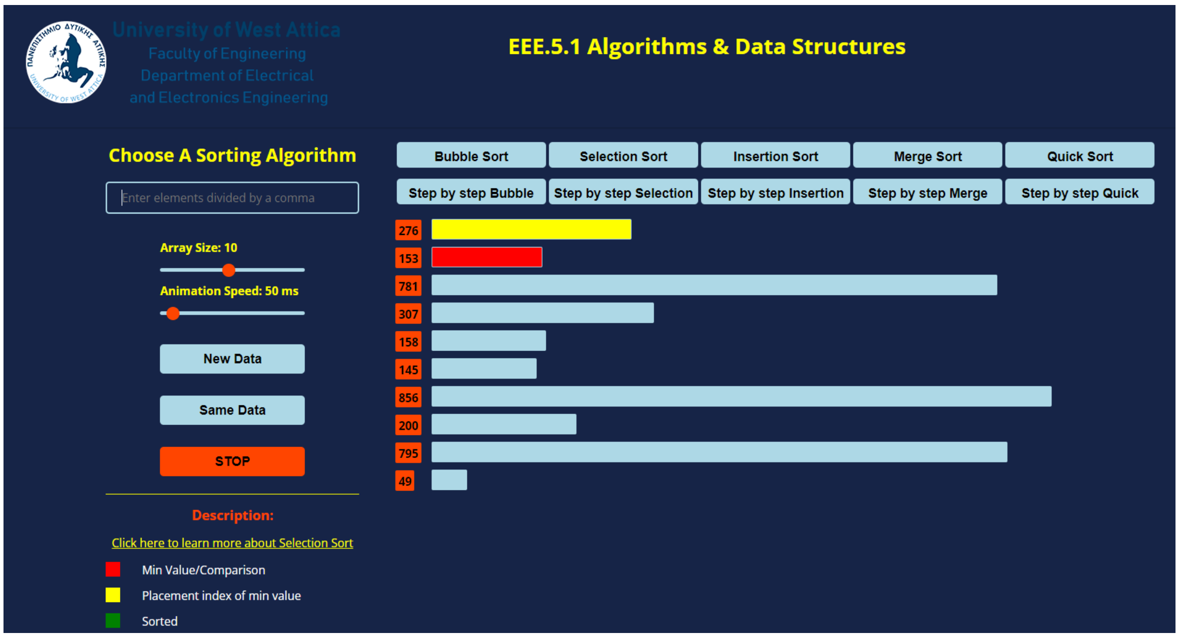


Figure 5.16: SELECTION SORT: Array consists of elements: 276, 153, 781, 307, 158, 145, 856, 200, 795, 49.

Starting with the first iteration, a comparison is being made between the first element, 276, and the rest of the elements in the array in order to find that 49 is the smallest element. Since 49 is less than 276, we swap the first element with the smallest element, 49. Figure 5.17 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

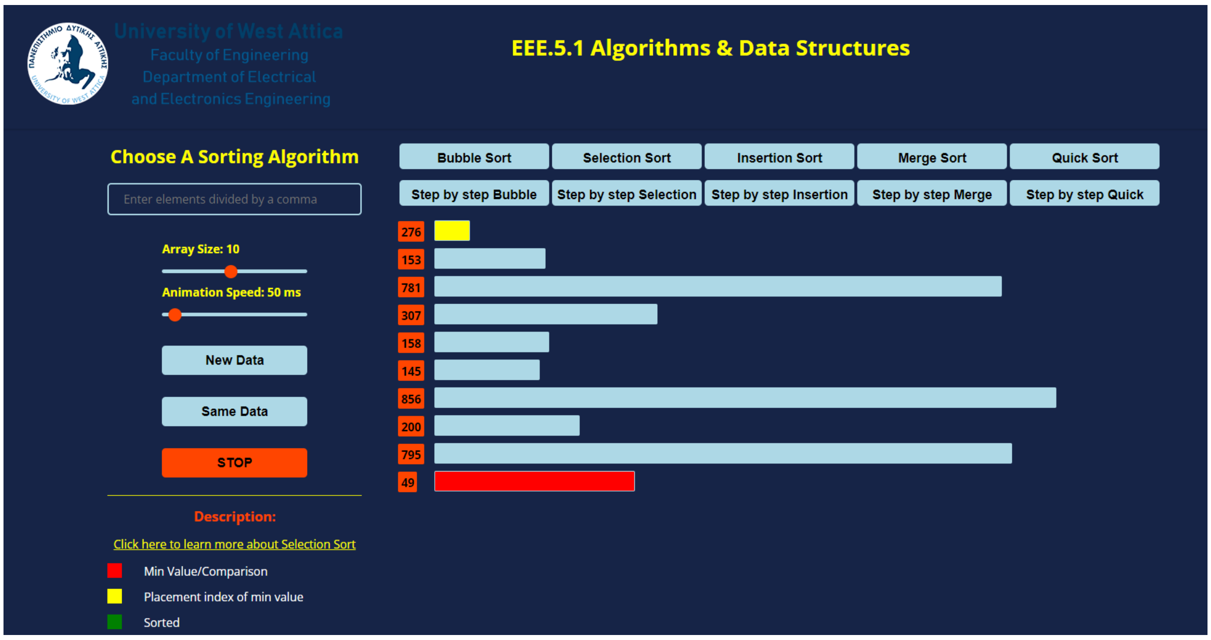


Figure 5.17: SELECTION SORT: First iteration indicates the correct position for the smallest element.

First iteration results in the sequence seen below:

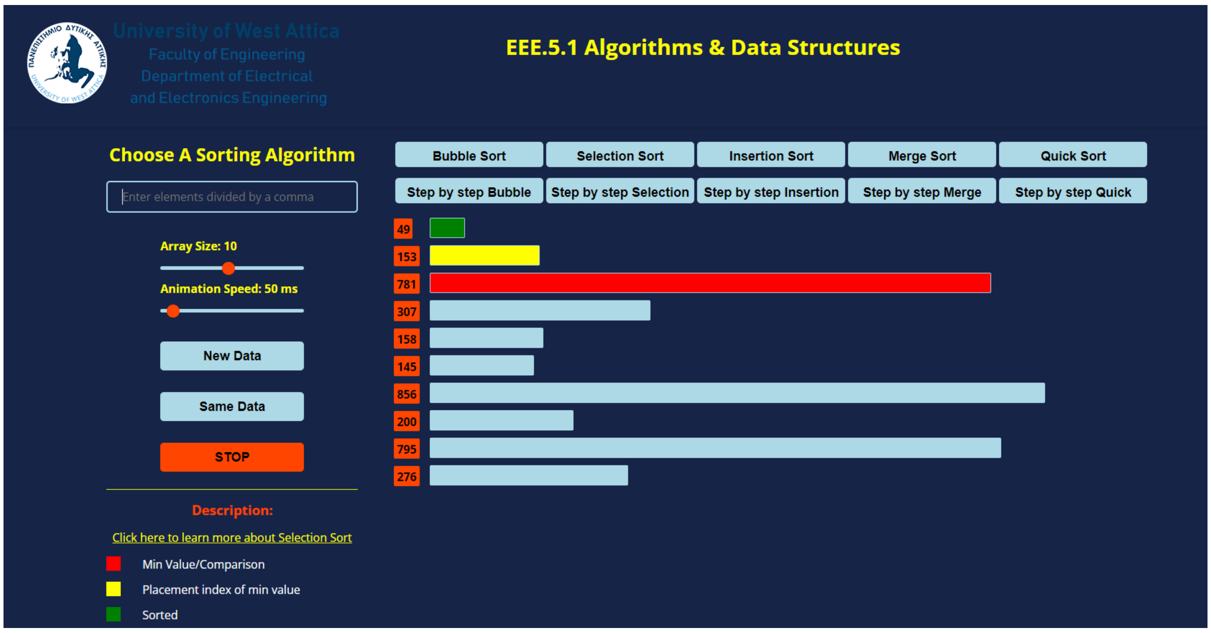


Figure 5.18: SELECTION SORT: First iteration is over and the green color indicates the sorted portion of the array.

Continuing with the second iteration, a comparison is being made between the second element, 153, and the rest of the elements in the unsorted portion of the array in order to find that 145 is the second smallest element. Since 145 is less than 153, we swap the second element with the second smallest element, 145. Figure 5.19 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

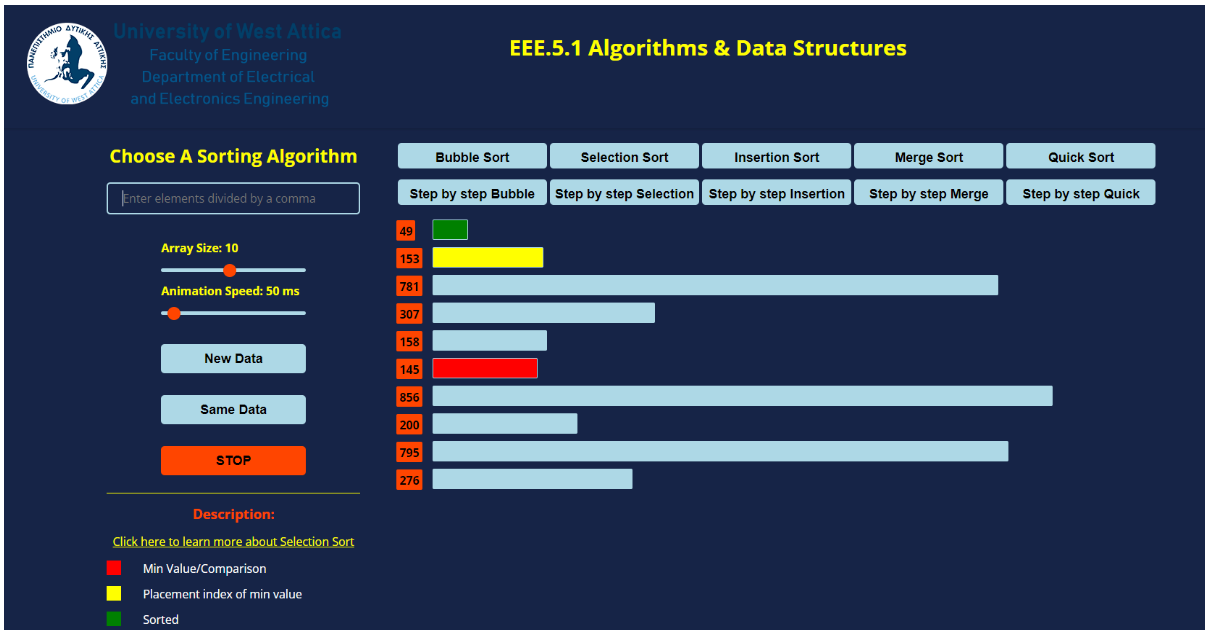


Figure 5.19: SELECTION SORT: Second iteration indicates the correct position for the second smallest element.

The aforementioned swap is giving us the following sequence:

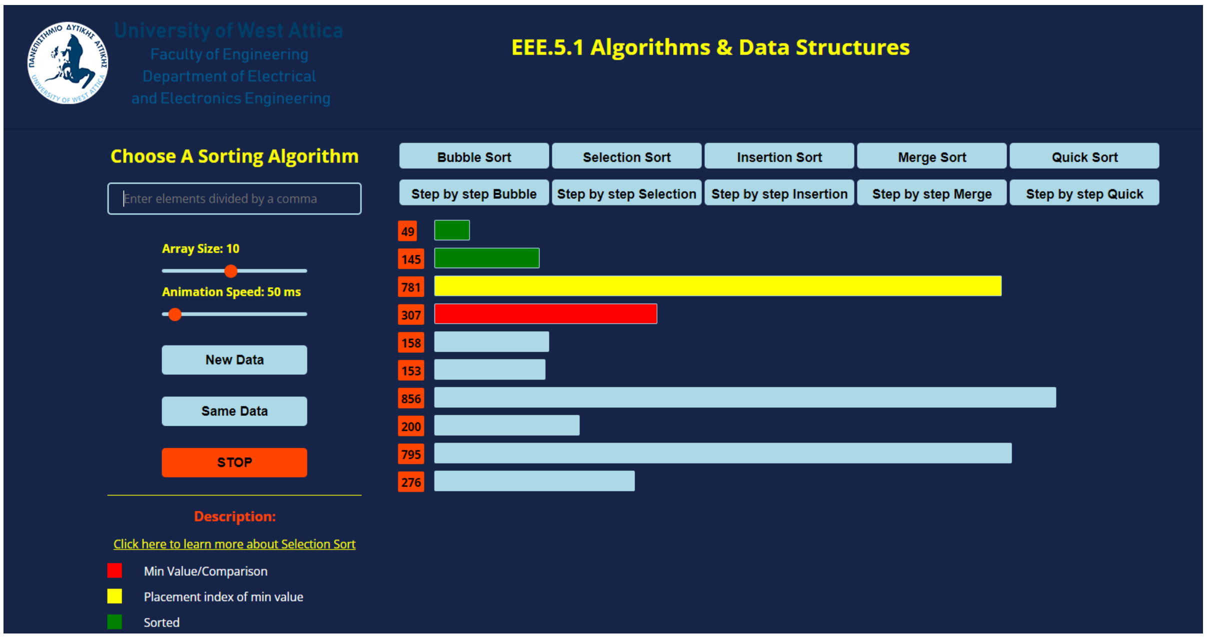


Figure 5.20: SELECTION SORT: Second iteration is over and the green color indicates the sorted portion of the array.

On the third iteration, a comparison is being made between the third element, 781, and the rest of the elements in the unsorted portion of the array in order to find that 153 is the third smallest element. Since 153 is less than 781, we swap the third element with the third smallest element, 153. Figure 5.21 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

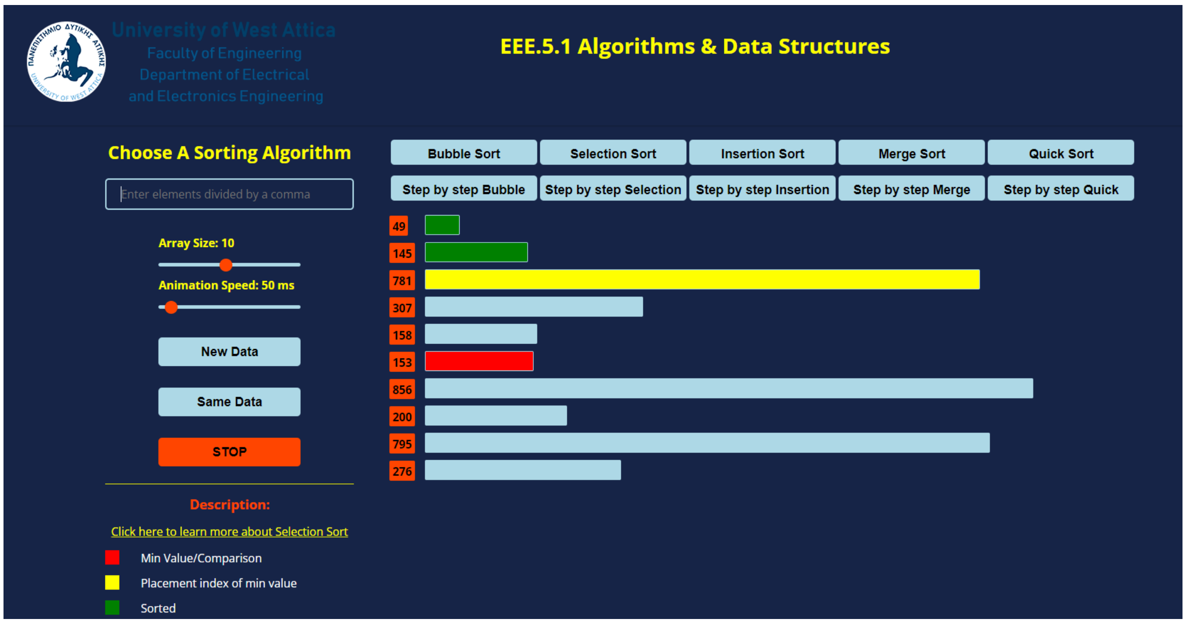


Figure 5.21: SELECTION SORT: Third iteration indicates the correct position for the third smallest element.

Third iteration results in the sequence seen below:

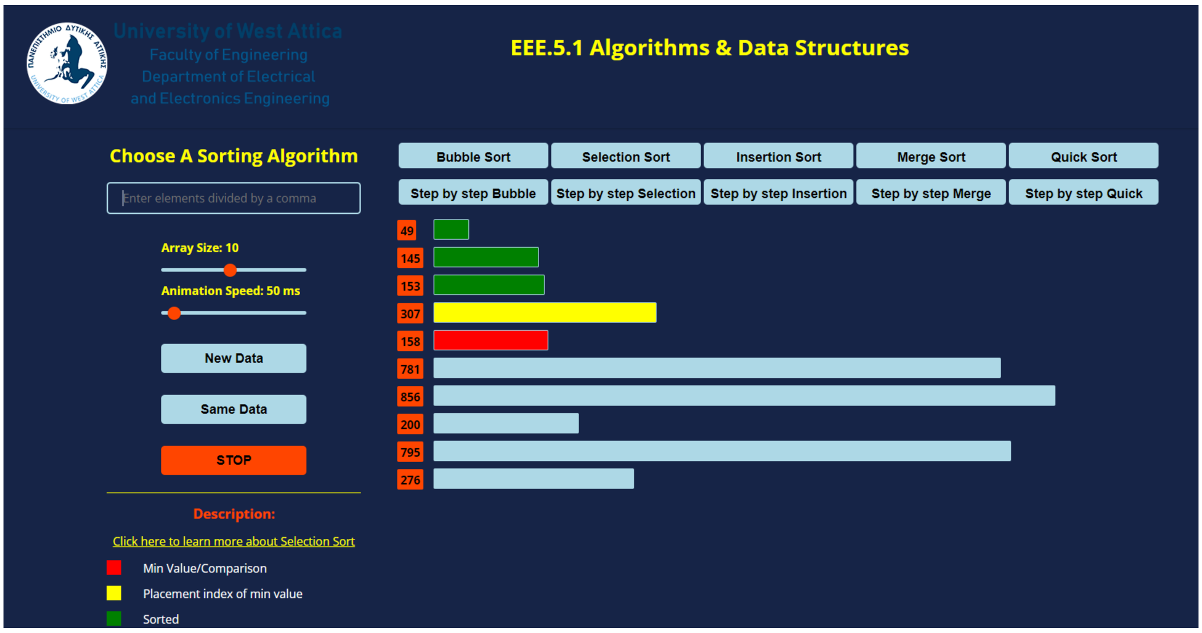


Figure 5.22: SELECTION SORT: Third iteration is over and the green color indicates the sorted portion of the array.

Starting with the fourth iteration, a comparison is being made between the fourth element, 307, and the rest of the elements in the unsorted portion of the array in order to find that 158 is the fourth smallest element. Since 158 is less than 307, we swap the fourth element with the fourth smallest element, 158. Figure 5.23 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

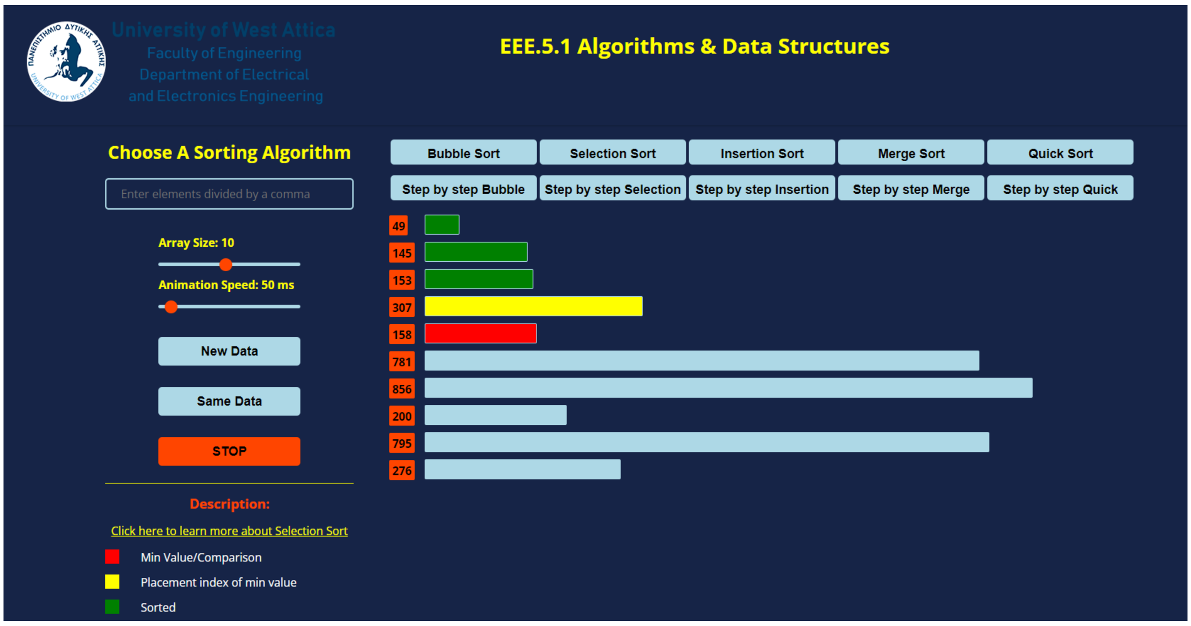


Figure 5.23: SELECTION SORT: Fourth iteration indicates the correct position for the fourth smallest element.

The aforementioned swap is giving us the following sequence:

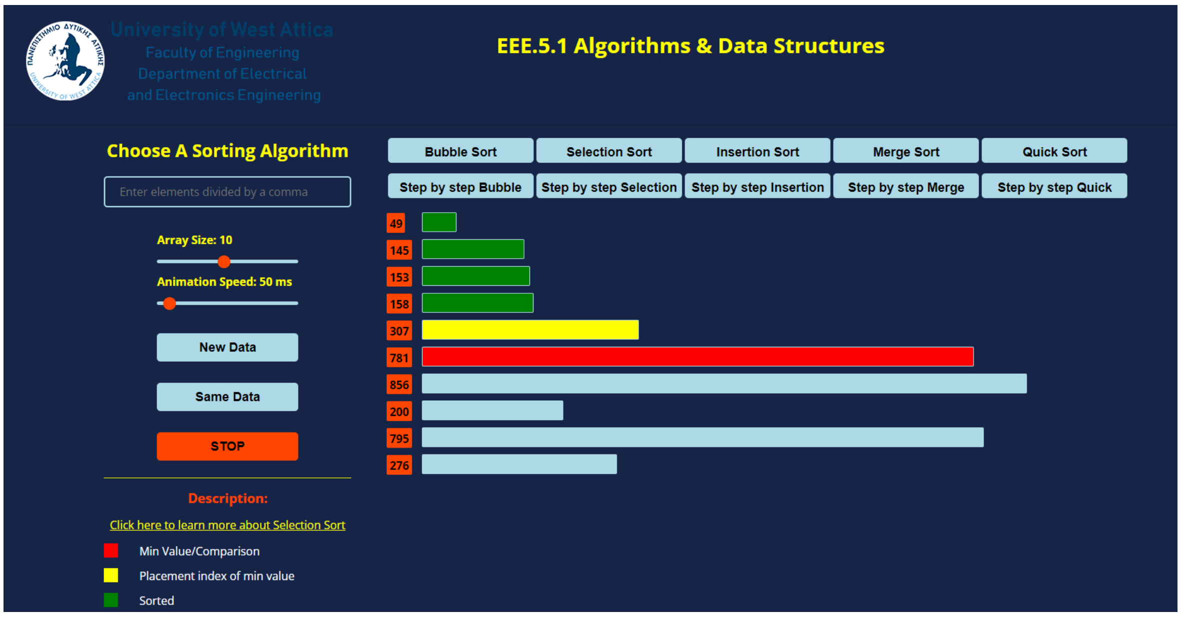


Figure 5.24: SELECTION SORT: Fourth iteration is over and the green color indicates the sorted portion of the array.

Continuing with the fifth iteration, a comparison is being made between the fifth element, 307, and the rest of the elements in the unsorted portion of the array in order to find that 200 is the fifth smallest element. Since 200 is less than 307, we swap the fifth element with the fifth smallest element, 200. Figure 5.25 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

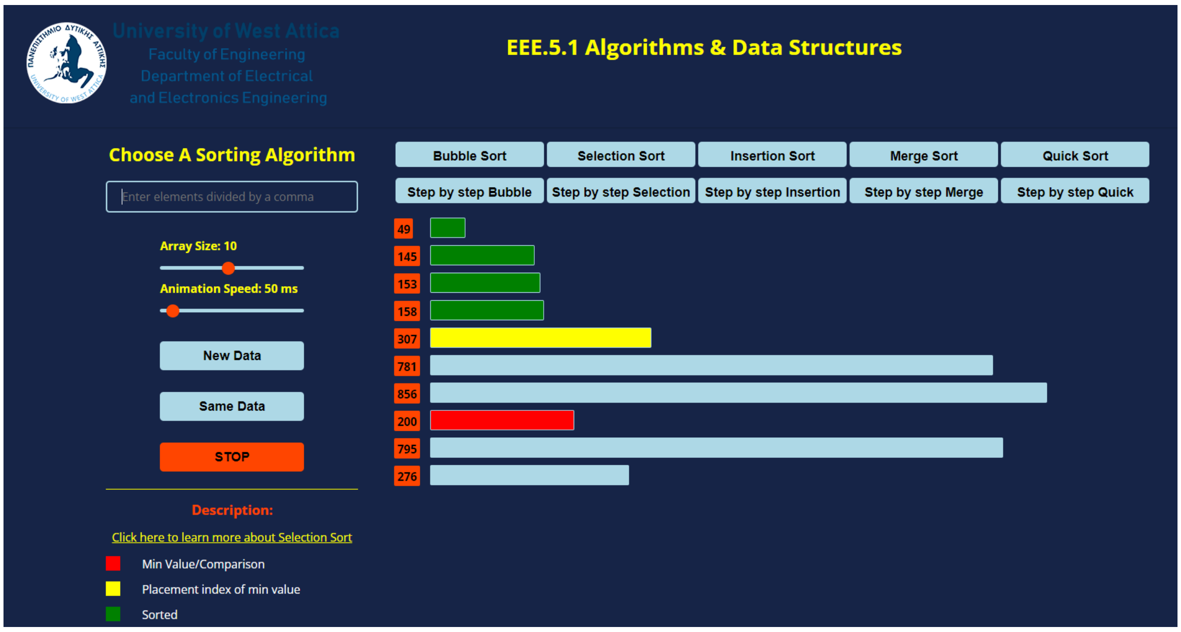


Figure 5.25: SELECTION SORT: Fifth iteration indicates the correct position for the fifth smallest element.

Fifth iteration results in the sequence seen below:

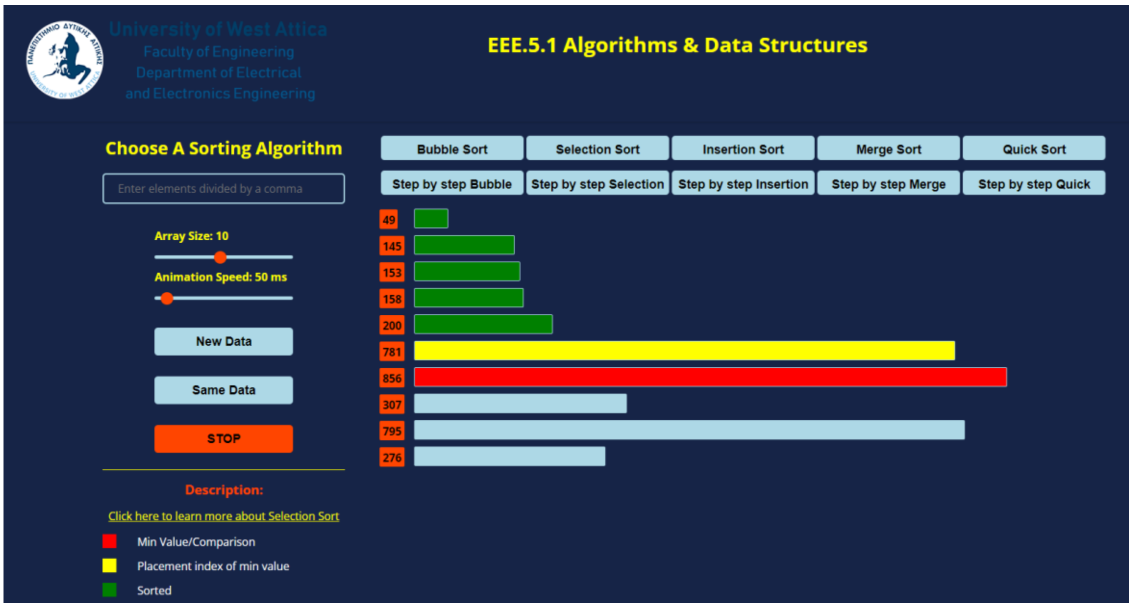


Figure 5.26: SELECTION SORT: Fifth iteration is over and the green color indicates the sorted portion of the array.

On the sixth iteration, a comparison is being made between the sixth element, 781, and the rest of the elements in the unsorted portion of the array in order to find that 276 is the sixth smallest element. Since 276 is less than 781, we swap the sixth element with the sixth smallest element, 276. Figure 5.27 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

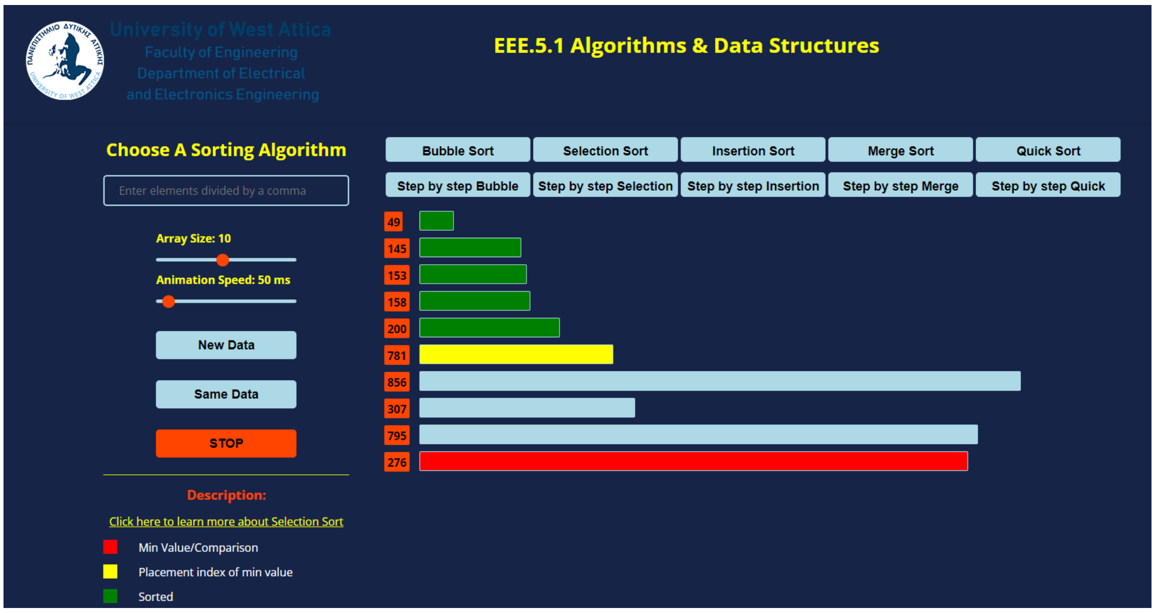


Figure 5.27: SELECTION SORT: Sixth iteration indicates the correct position for the sixth smallest element.

The aforementioned swap is giving us the following sequence:

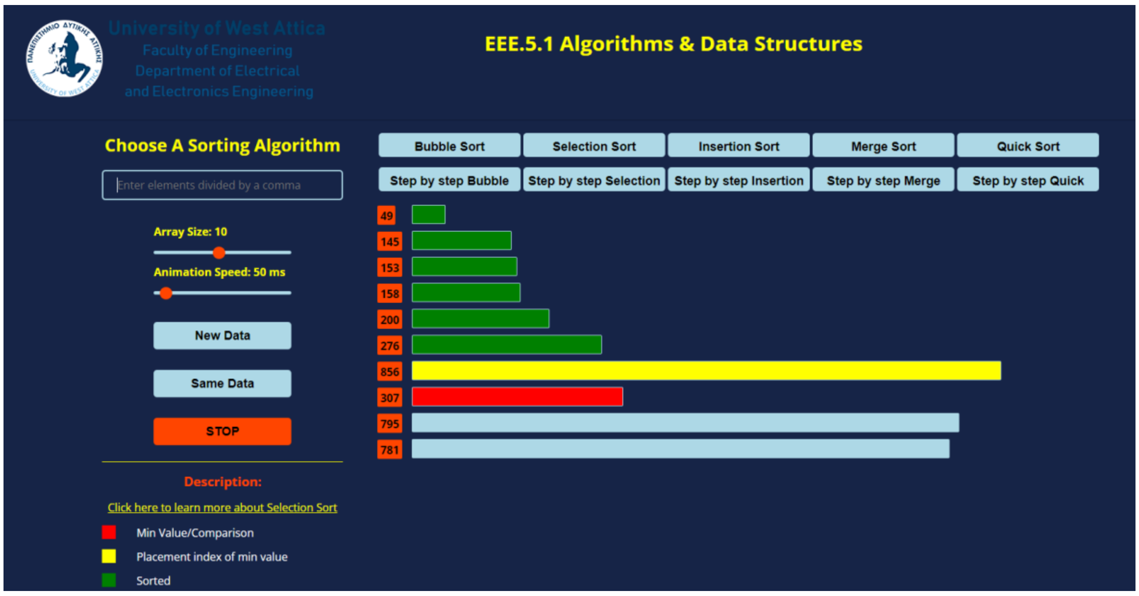


Figure 5.28: SELECTION SORT: Sixth iteration is over and the green color indicates the sorted portion of the array.

Continuing with the seventh iteration, a comparison is being made between the seventh element, 856, and the rest of the elements in the unsorted portion of the array in order to find that 307 is the seventh smallest element. Since 307 is less than 856, we swap the seventh element with the seventh smallest element, 307. Figure 5.29 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

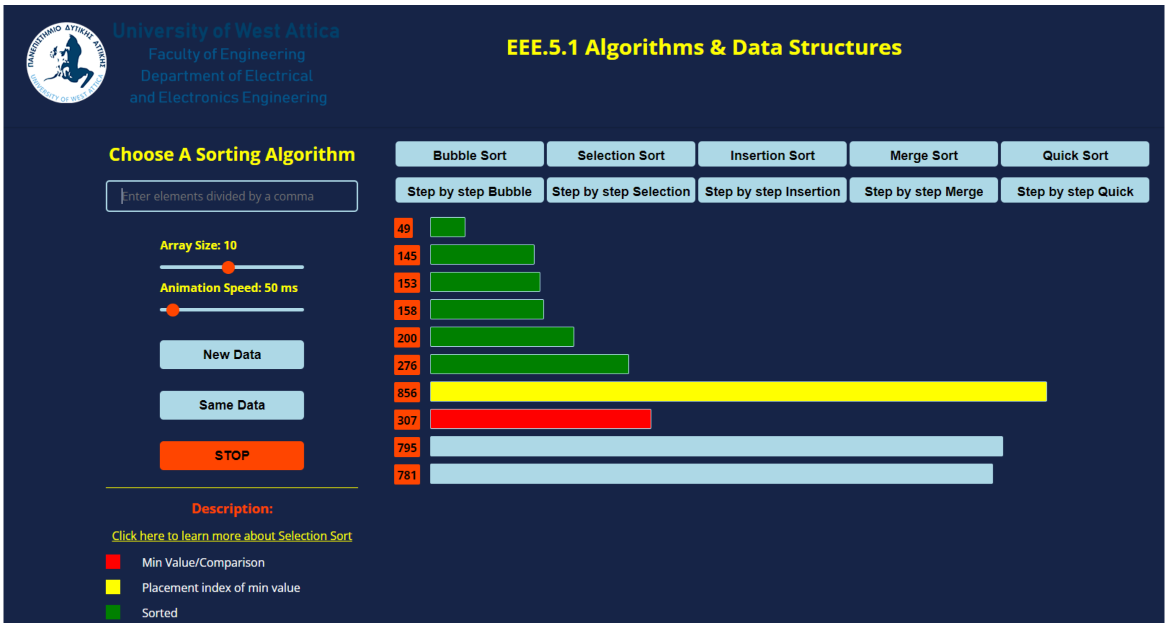
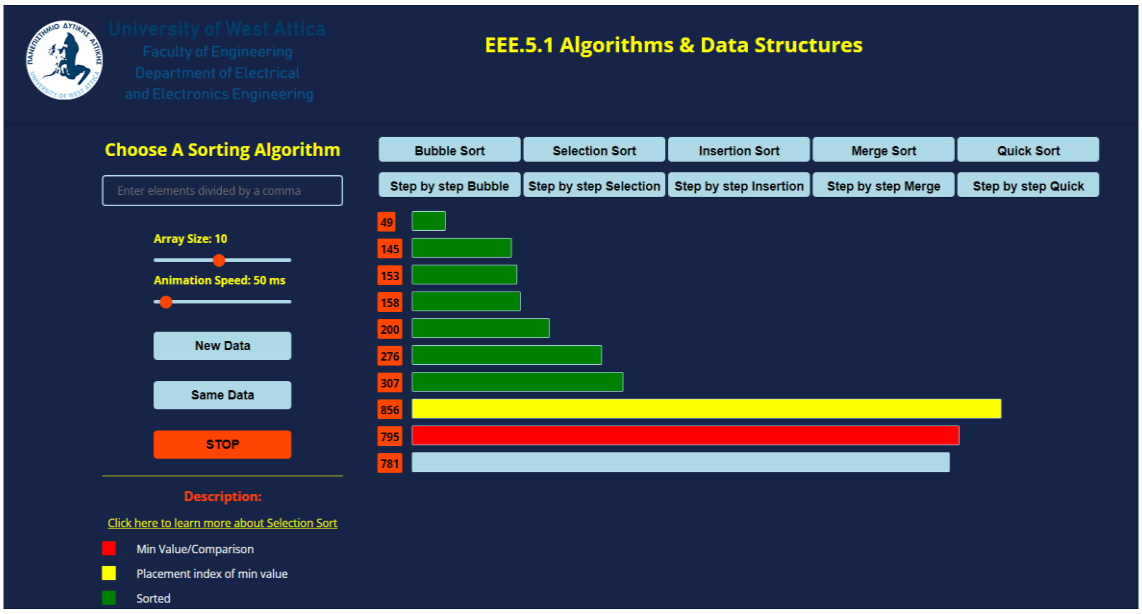


Figure 5.29: SELECTION SORT: Seventh iteration indicates the correct position for the seventh smallest element.

Seventh iteration results in the sequence seen below:



Array Size: 10

Animation Speed: 50 ms

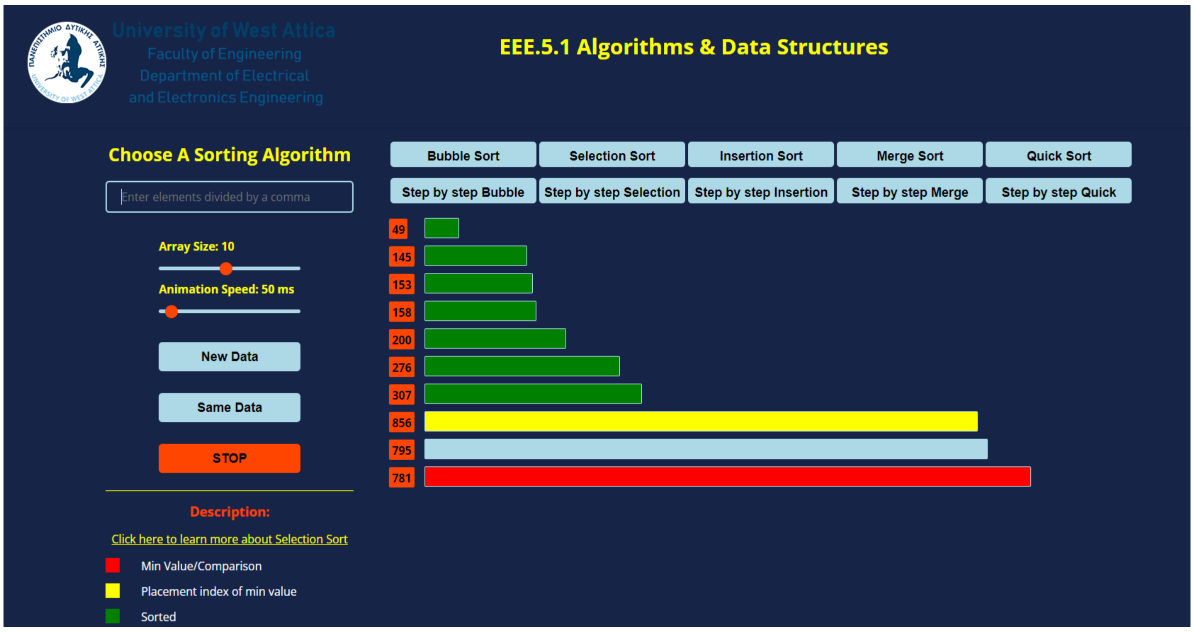
🟥 Min Value/Comparison

🟨 Placement index of min value

🟩 Sorted

Figure 5.30: SELECTION SORT: Seventh iteration is over and the green color indicates the sorted portion of the array.

On the eighth iteration, a comparison is being made between the eighth element, 856, and the rest of the elements in the unsorted portion of the array in order to find that 781 is the eighth smallest element. Since 781 is less than 856, we swap the eighth element with the eighth smallest element, 781. Figure 5.31 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.



Array Size: 10

Animation Speed: 50 ms

🟥 Min Value/Comparison

🟨 Placement index of min value

🟩 Sorted

Figure 5.31: SELECTION SORT: Eighth iteration indicates the correct position for the eighth smallest element.

The aforementioned swap is giving us the following sequence:

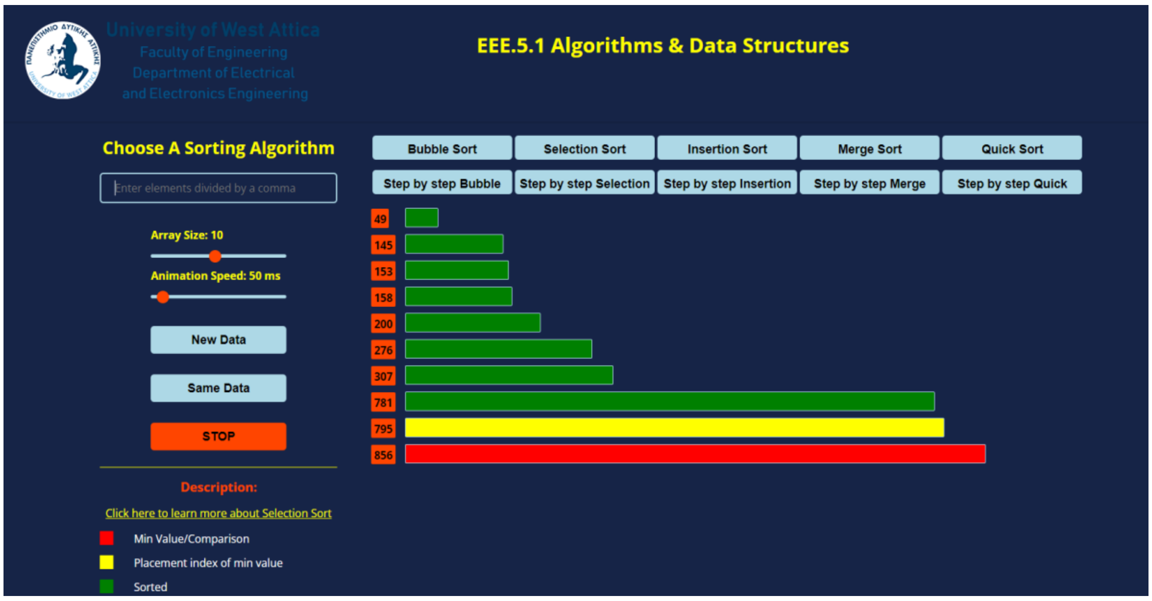


Figure 5.33: SELECTION SORT: Eighth iteration is over and the green color indicates the sorted portion of the array.

Continuing with the ninth and last iteration, a comparison is being made between the ninth element, 795, and the rest of the elements in the unsorted portion of the array in order to find that 856 is the seventh smallest element. At this point, since there is only one element, there is not need to perform a swap because 856 is already in the correct position. Figure 5.33 shows with color red the minimum element in the unsorted array and with color yellow the placement index of that value.

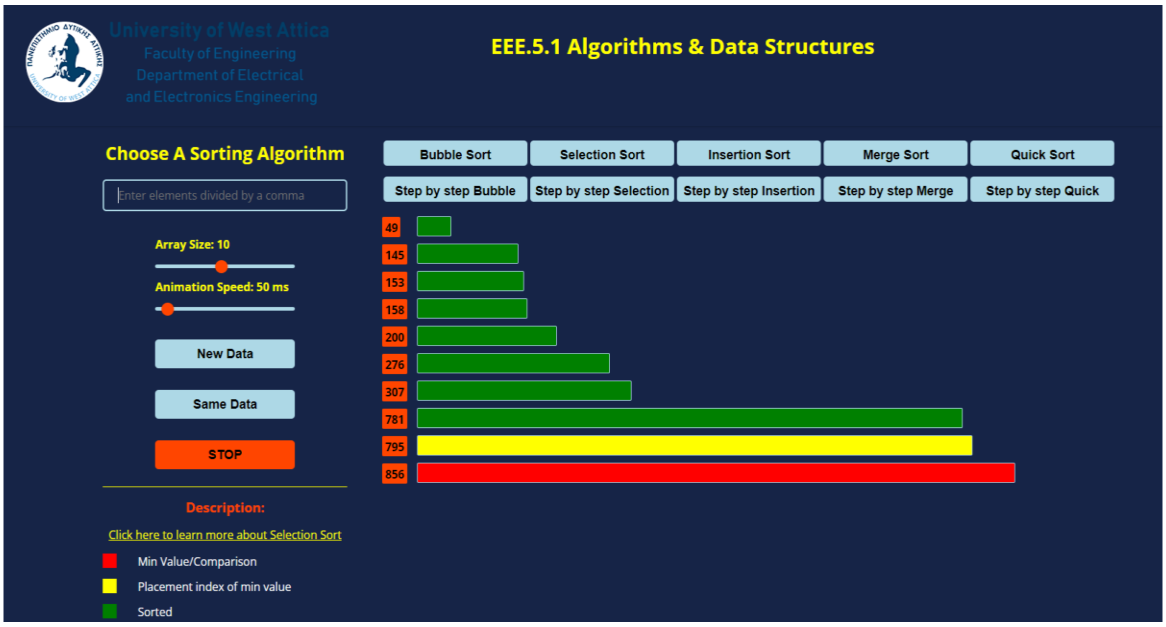


Figure 5.33: SELECTION SORT: Ninth iteration indicates the correct position for the ninth smallest element.

Ninth iteration results in the sequence seen below as the final sorted array:

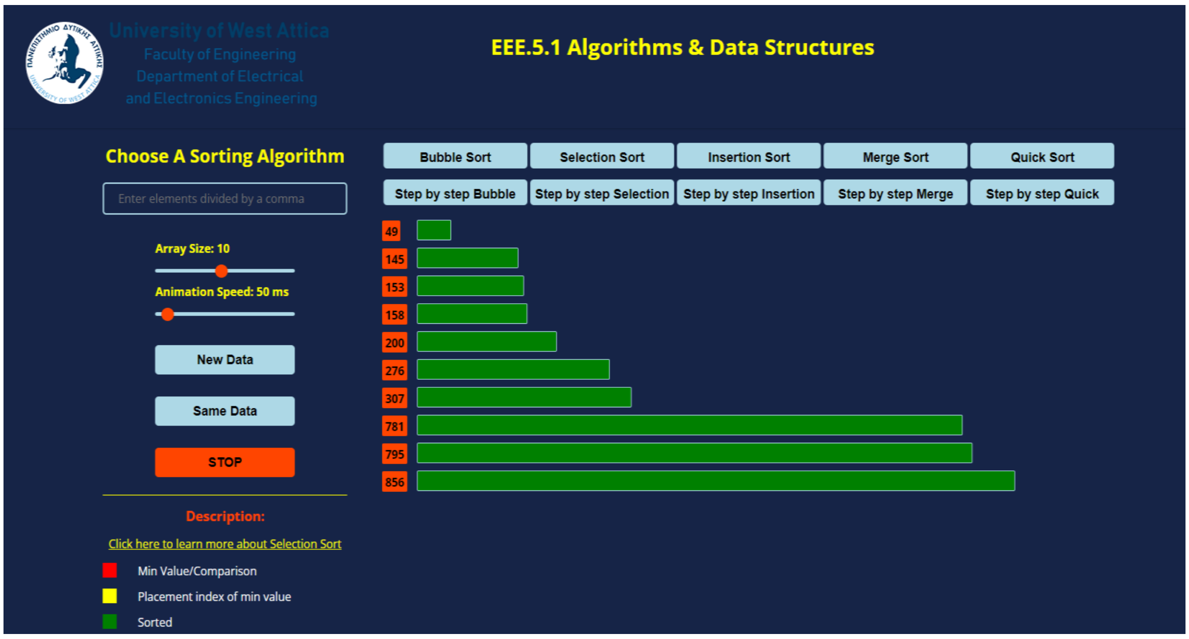


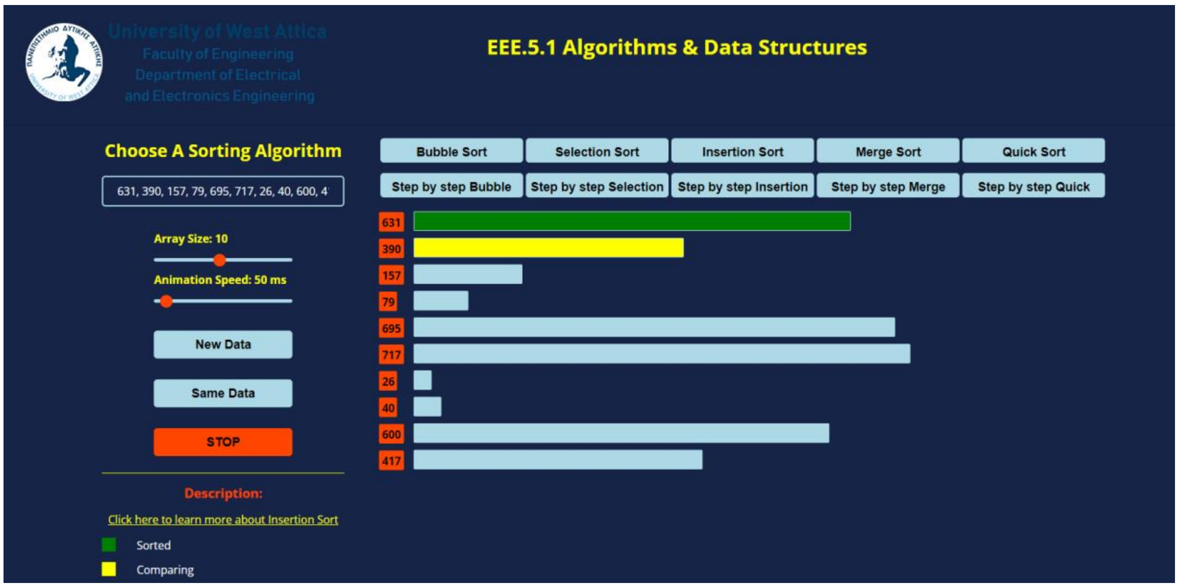
Figure 5.34: SELECTION SORT: Seventh iteration is over and the green color indicates the sorted portion of the array.

5.4 Insertion Sort

The following colors are used in insertion sort:

- Light blue: The initial color of the bars used to represent the elements in the array. This makes it easier for the student in distinguishing between elements that have already been sorted and those that are still through processing.
- Yellow: The element that is being compared to the elements in the sorted portion of the array is shown on each iteration of the insertion sort algorithm by the color yellow. Each element is compared to the elements on its left and inserted into the correct position in the sorted portion of the array, as the algorithm progresses. The yellow color helps the student comprehend how the algorithm is working and how the elements are being sorted
- Green: The green color is used to indicate that an element is sorted. When an element is position correctly, it is said to be sorted and is no longer a part of the array’s unsorted portion.

We will analyze the application’s algorithm using a ten-element array. This is how the algorithm operates:



631, 390, 157, 79, 695, 717, 26, 40, 600, 4

Array Size: 10

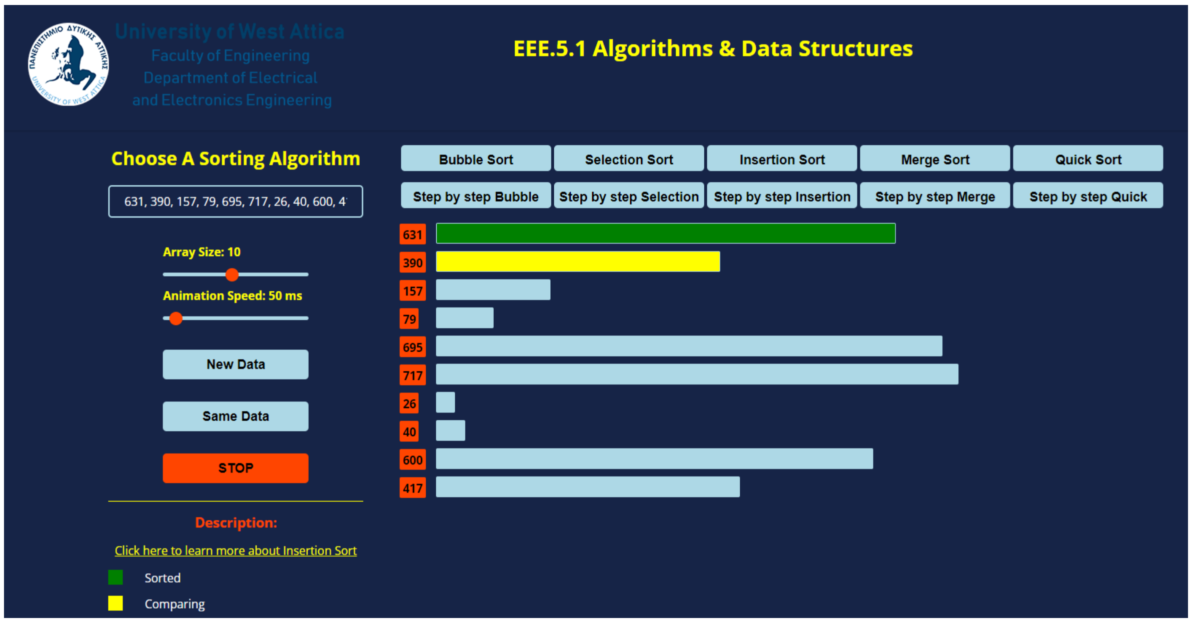
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.35: INSERTION SORT: Array consists of elements: 631, 390, 157, 79, 695, 717, 26, 40, 600, 417.

Starting with the first iteration, we begin with the first element of the array, which is 631. This element is considered “sorted”. Figure 5.36 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



631, 390, 157, 79, 695, 717, 26, 40, 600, 4

Array Size: 10

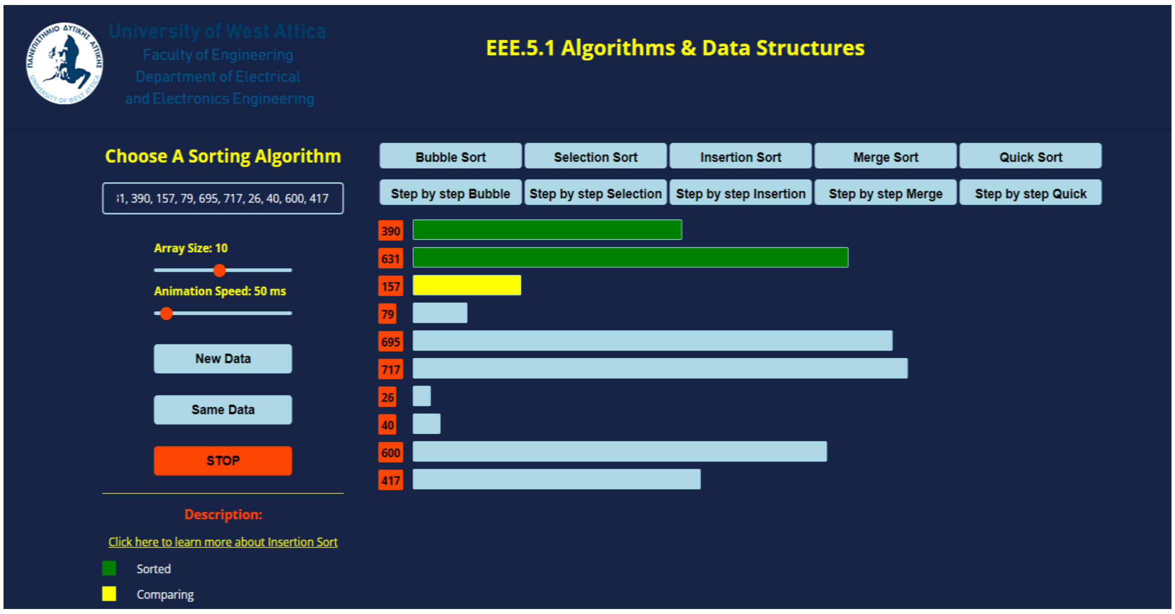
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.36: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the second element, 390, and the elements to its left, meaning the sorted portion of the array. Since 390 is smaller than 631, 631 is shifted to the right and 390 is inserted to the left:



1, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

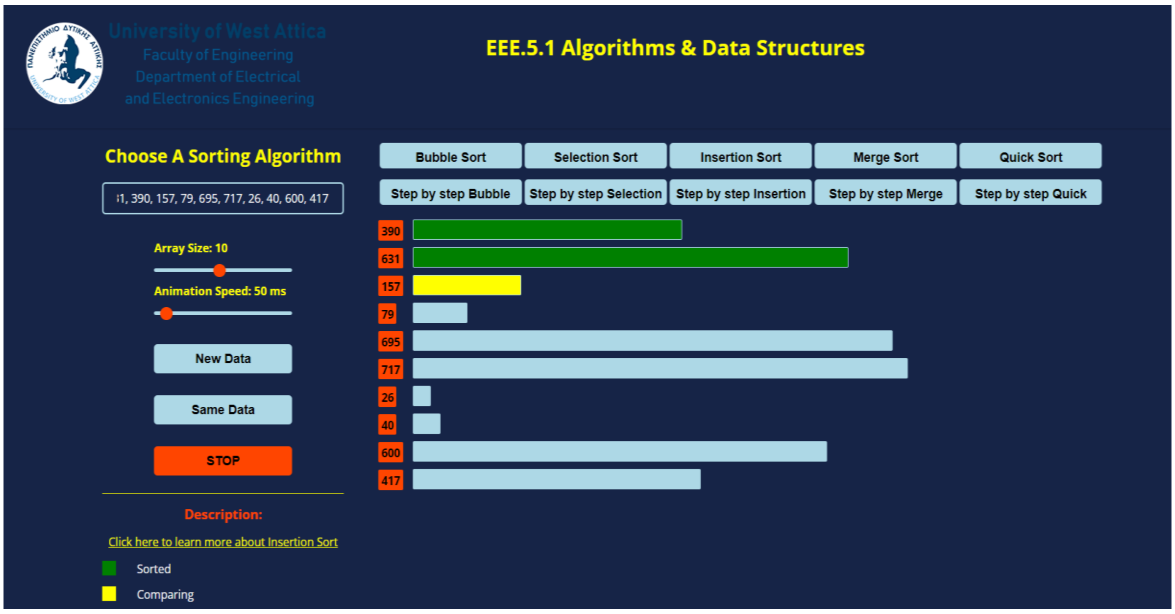
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.37: INSERTION SORT: The second element is now considered part of the sorted portion of the array.

On the second iteration, we continue the process with the third element of the array, which is 157. This element is considered “sorted”. Figure 5.38 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



1, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

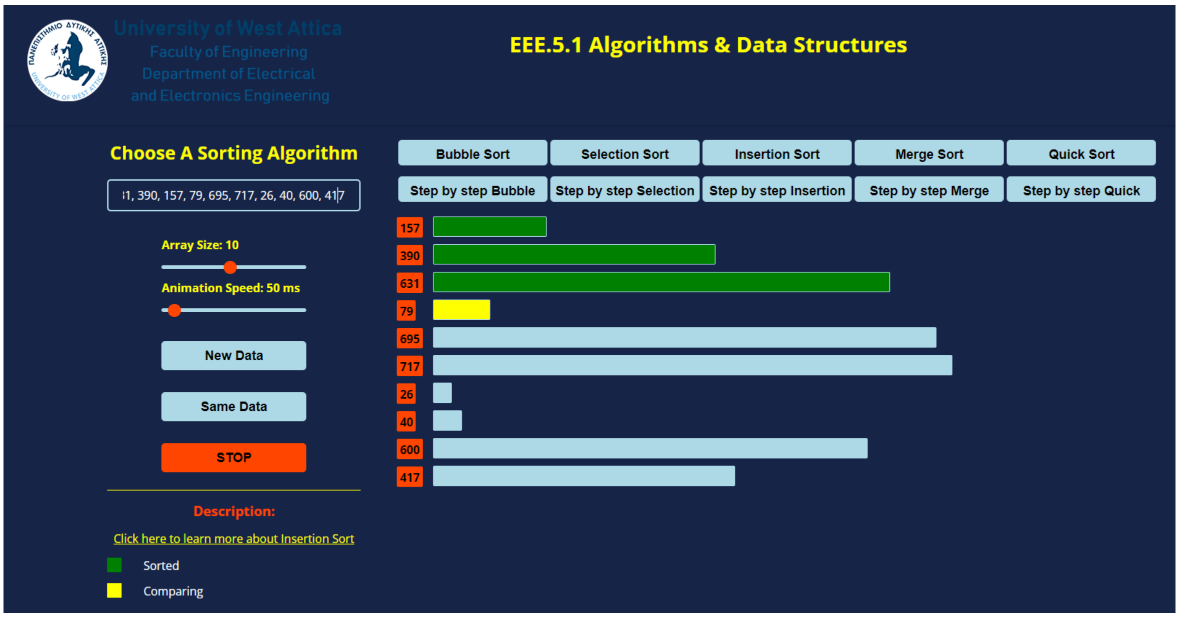
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.38: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the third element, 157, and the elements to its left, meaning the sorted portion of the array. Since 157 is smaller than 390, 390 is shifted to the right and 157 is inserted to the left:



1, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

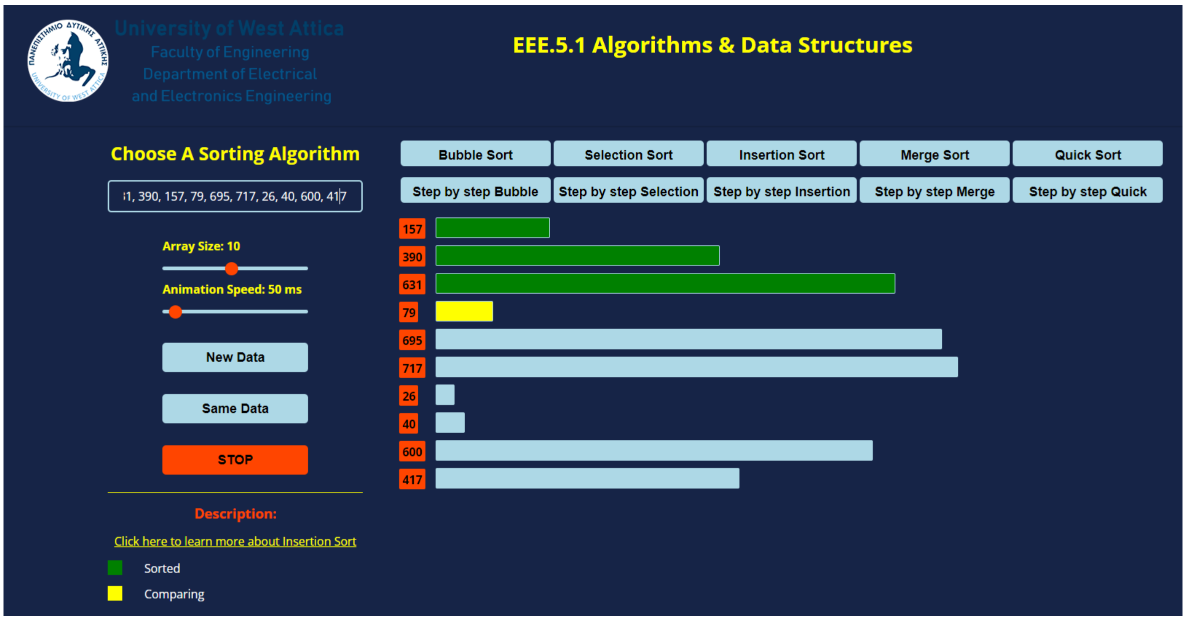
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.39: INSERTION SORT: The third element is now considered part of the sorted portion of the array.

On the third iteration, we continue the process with the fourth element of the array, which is 79. This element is considered “sorted”. Figure 5.40 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



1, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

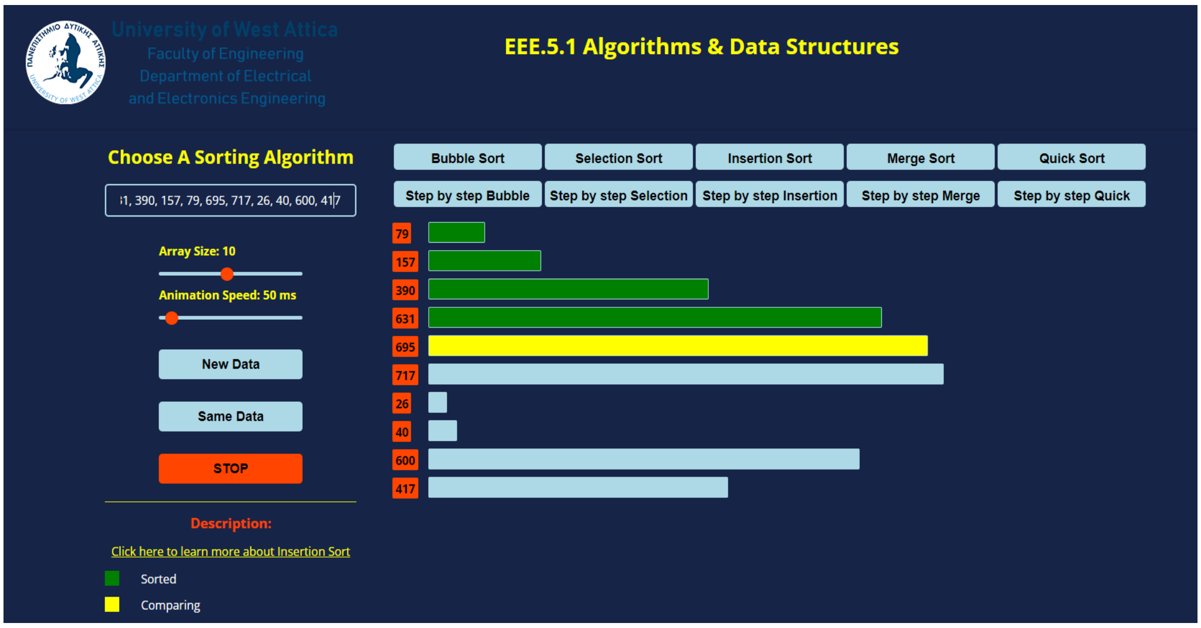
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.40: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the fourth element, 79, and the elements to its left, meaning the sorted portion of the array. Since 79 is smaller than 157, 157 is shifted to the right and 79 is inserted to the left:



31, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

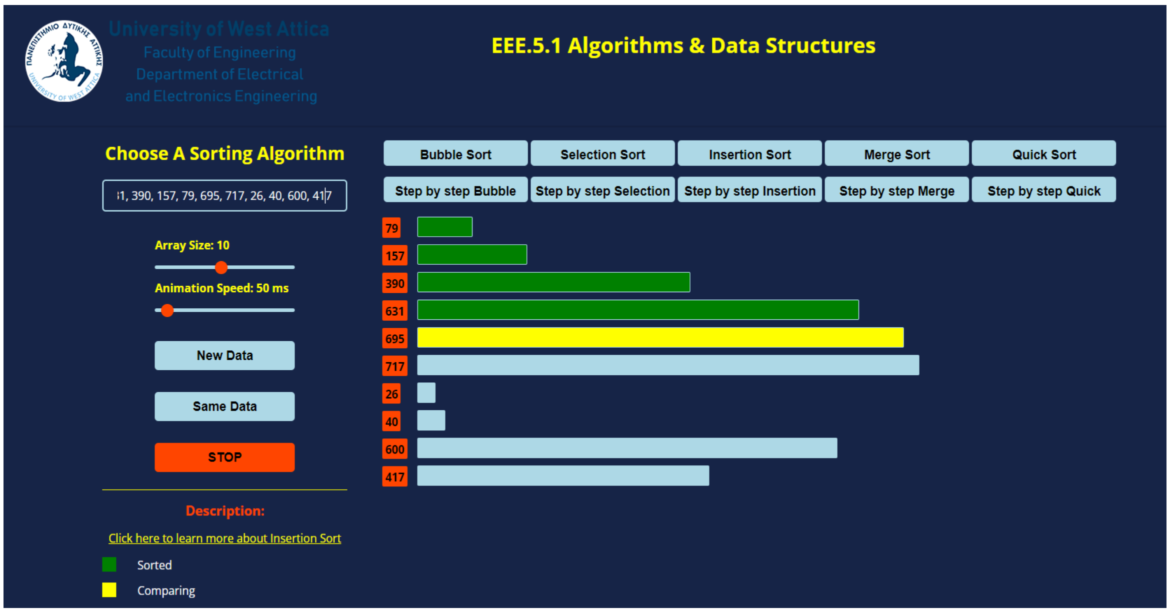
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.41: INSERTION SORT: The fourth element is now considered part of the sorted portion of the array.

On the fourth iteration, we continue the process with the fifth element of the array, which is 695. This element is considered “sorted”. Figure 5.42 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



31, 390, 157, 79, 695, 717, 26, 40, 600, 417

Array Size: 10

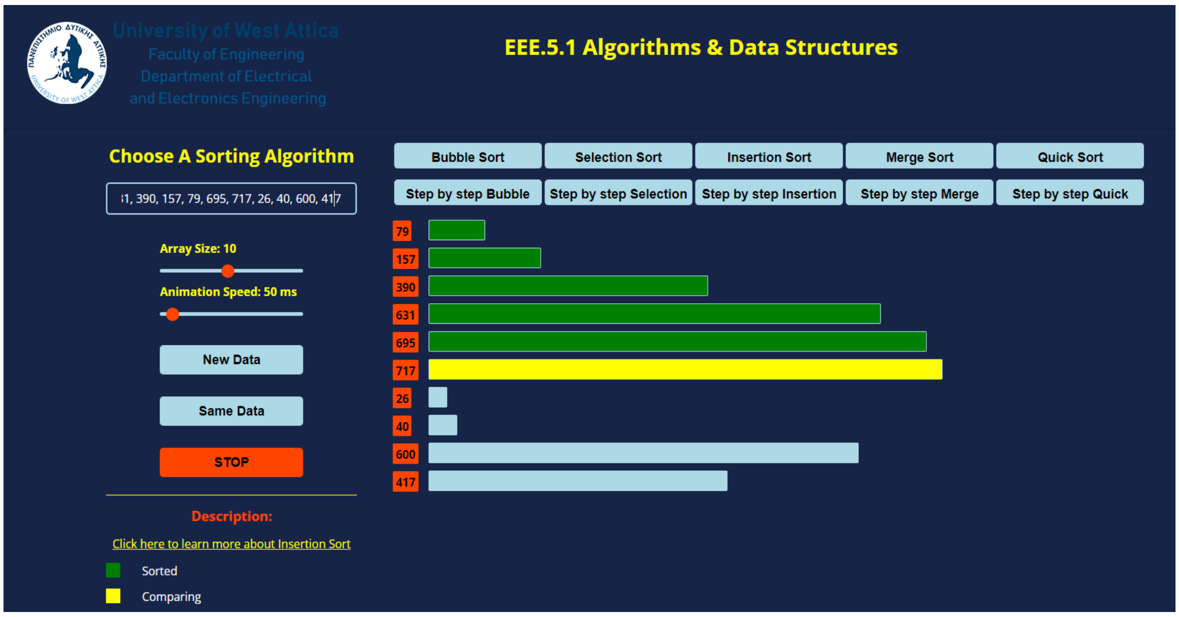
Animation Speed: 50 ms

🟩 Sorted

🟨 Comparing

Figure 5.42: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the fifth element, 695, and the elements to its left, meaning the sorted portion of the array. Since 695 is larger than all of the elements in the sorted portion, it is inserted to the right:



ı1, 390, 157, 79, 695, 717, 26, 40, 600, 41|7

Array Size: 10

Animation Speed: 50 ms

79, 157, 390, 631, 695, 717, 26, 40, 600, 417

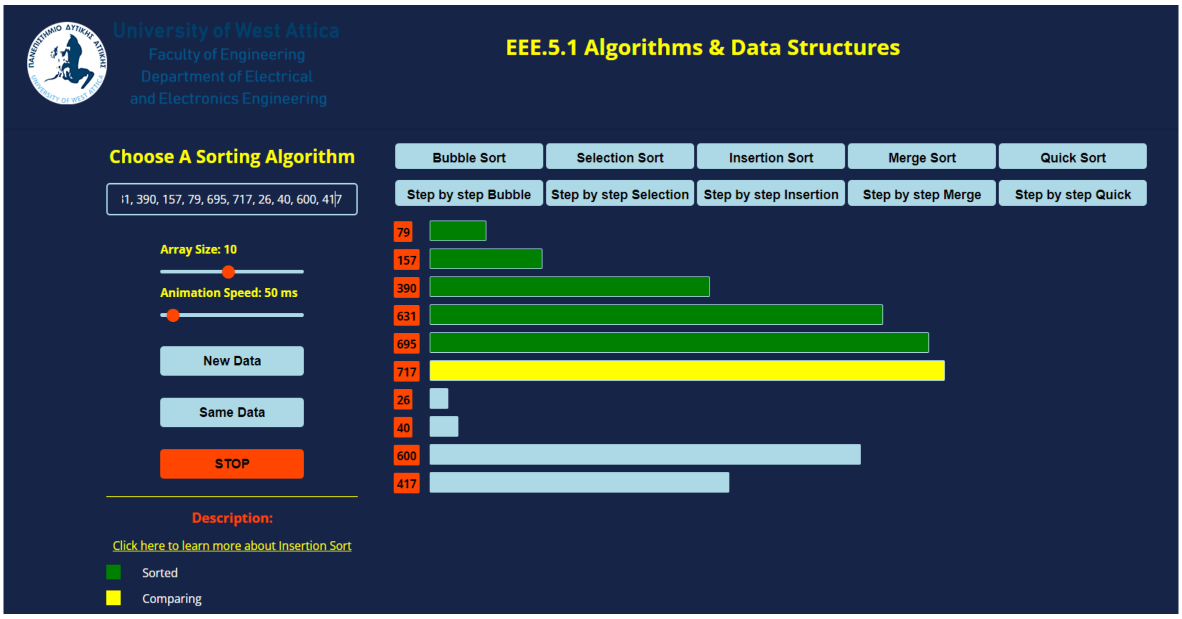
Description:

🟩 Sorted

🟨 Comparing

Figure 5.43: INSERTION SORT: The fifth element is now considered part of the sorted portion of the array.

On the fifth iteration, we continue the process with the sixth element of the array, which is 717. This element is considered “sorted”. Figure 5.44 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



ı1, 390, 157, 79, 695, 717, 26, 40, 600, 41|7

Array Size: 10

Animation Speed: 50 ms

79, 157, 390, 631, 695, 717, 26, 40, 600, 417

Description:

🟩 Sorted

🟨 Comparing

Figure 5.44: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the sixth element, 717, and the elements to its left, meaning the sorted portion of the array. Since 695 is larger than all of the elements to its left, it is inserted to the right:

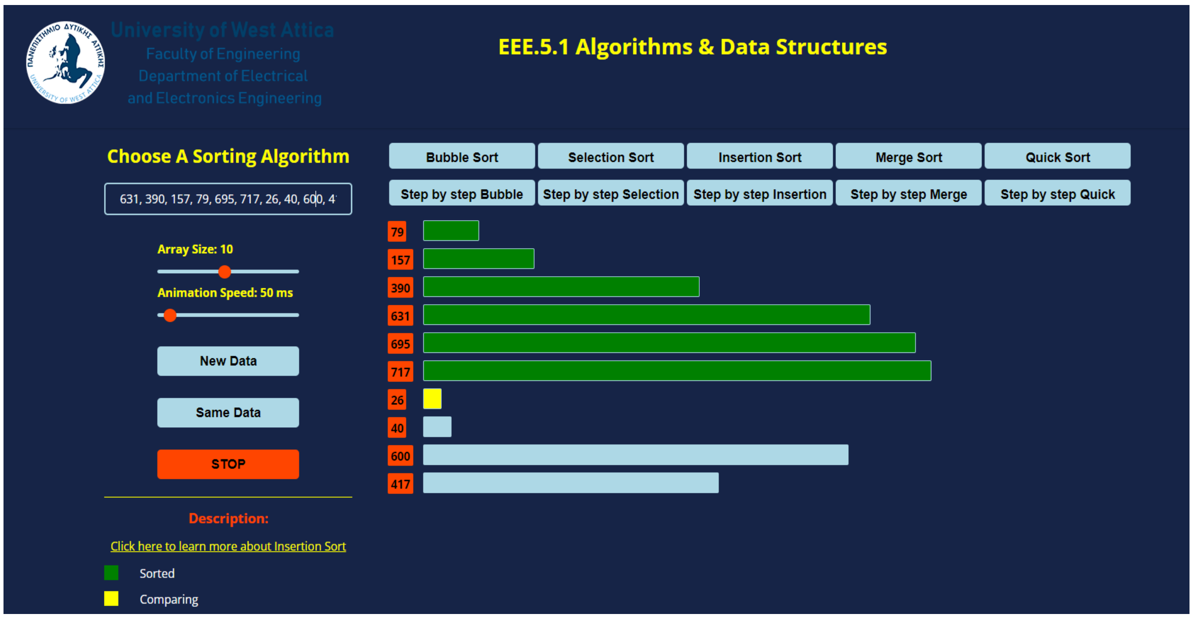


Figure 5.45: INSERTION SORT: The sixth element is now considered part of the sorted portion of the array.

On the sixth iteration, we continue the process with the seventh element of the array, which is 26. This element is considered “sorted”. Figure 5.46 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.

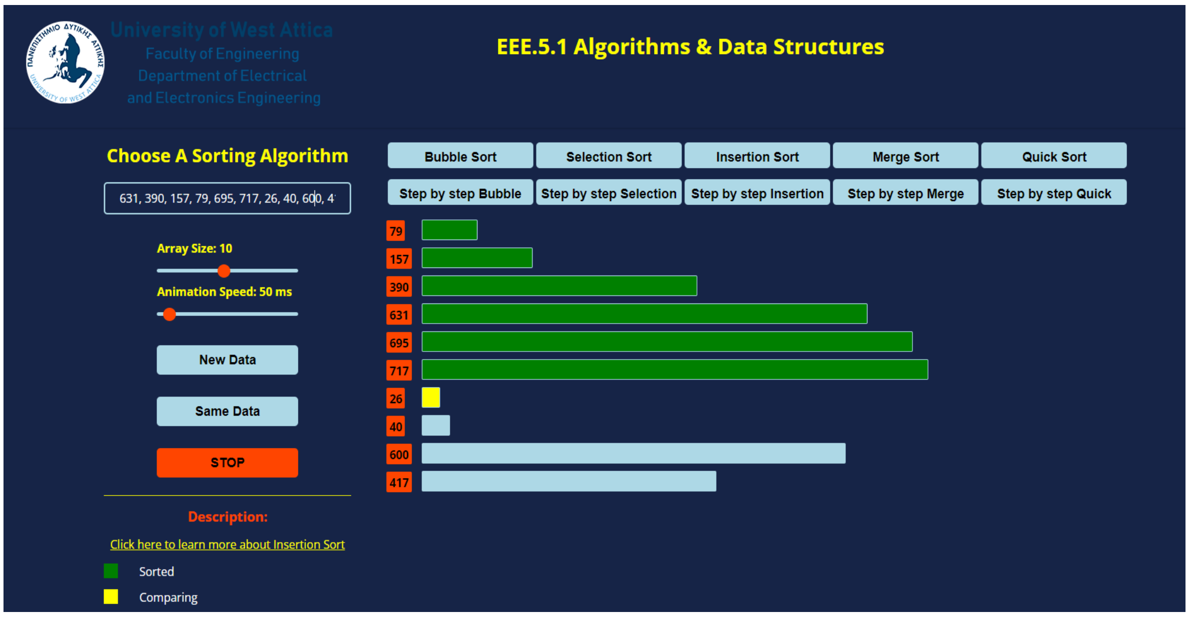


Figure 5.46: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the seventh element, 26, and the elements to its left, meaning the sorted portion of the array. Since 26 is smaller than 79, 79 is shifted to the right and 26 is inserted to the left:

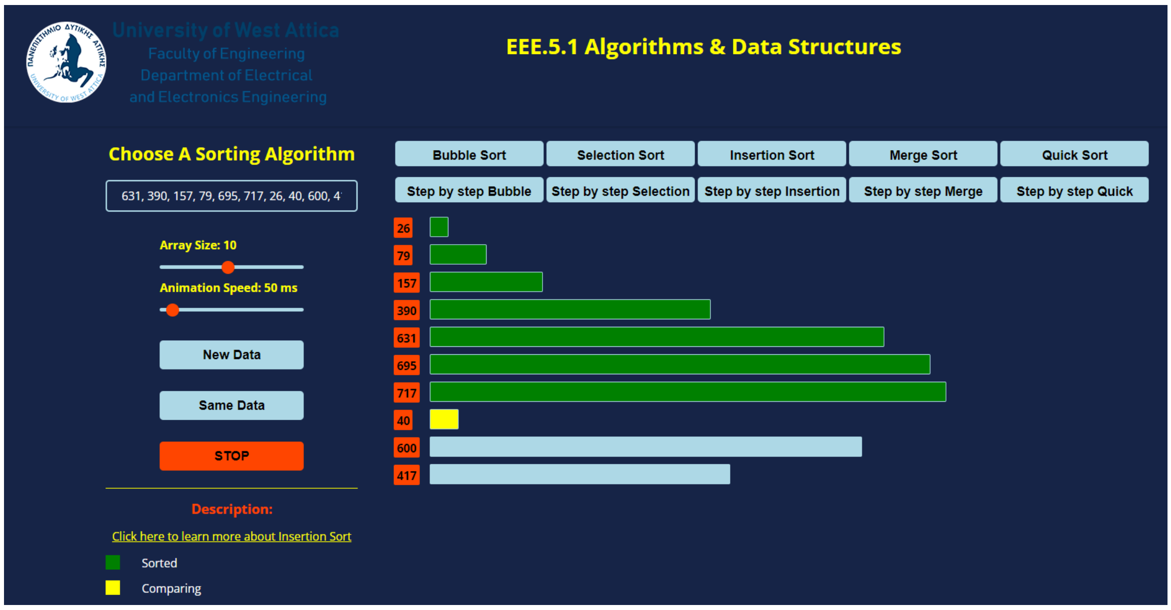


Figure 5.47: INSERTION SORT: The seventh element is now considered part of the sorted portion of the array.

On the seventh iteration, we continue the process with the eighth element of the array, which is 40. This element is considered “sorted”. Figure 5.48 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.

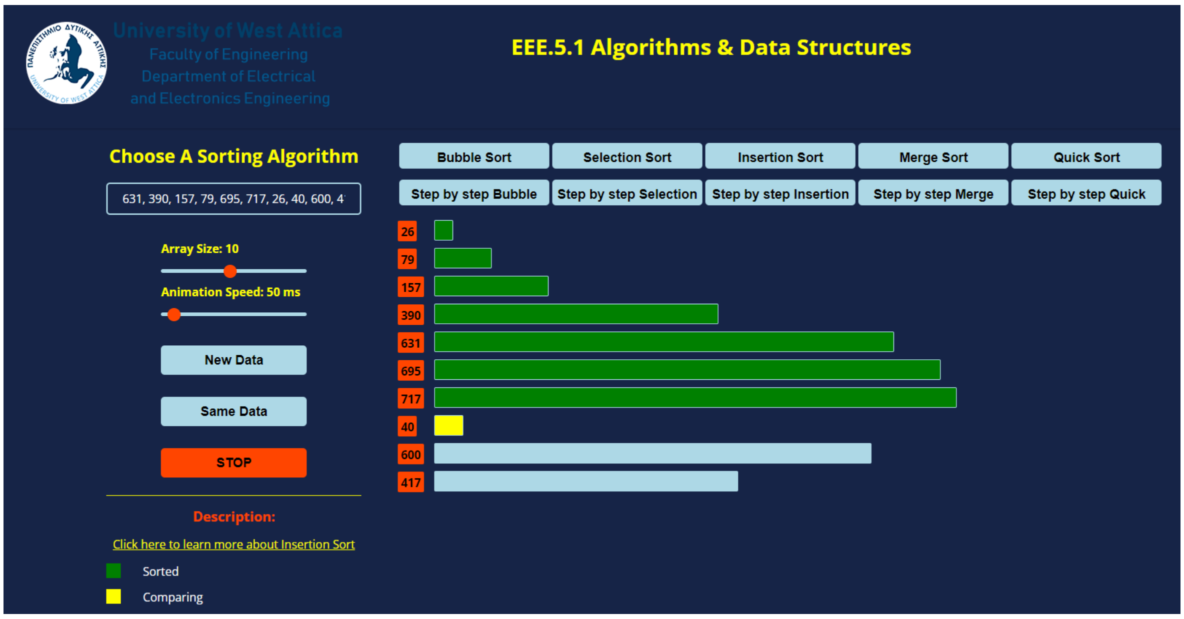


Figure 5.48: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the eighth element, 40, and the elements to its left, meaning the sorted portion of the array. Since 40 is larger than 26, but smaller than 79, 79 is shifted to the right and 50 is being inserted to the left:

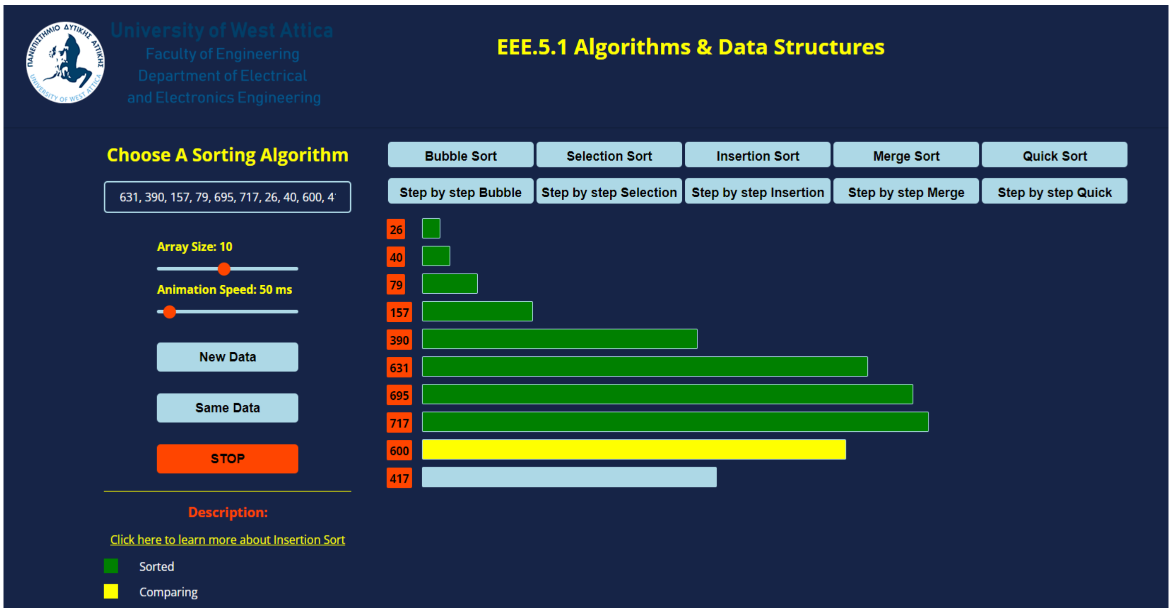


Figure 5.49: *INSERTION SORT: The eighth element is now considered part of the sorted portion of the array.*

On the eighth iteration, we continue the process with the ninth element of the array, which is 600. This element is considered “sorted”. Figure 5.50 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.

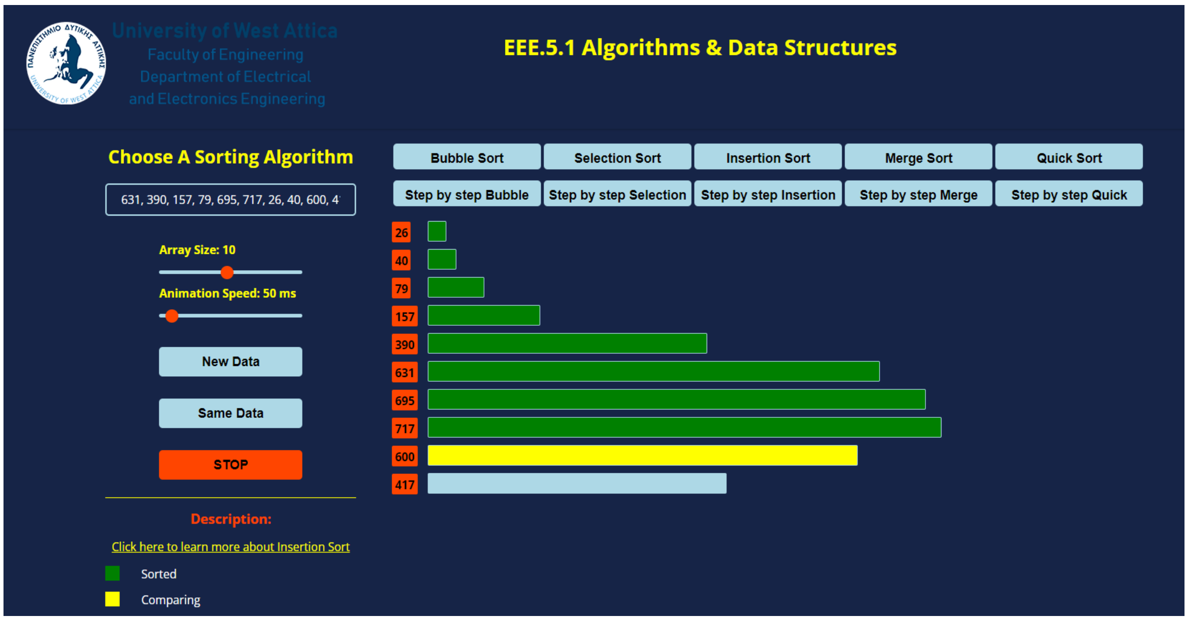
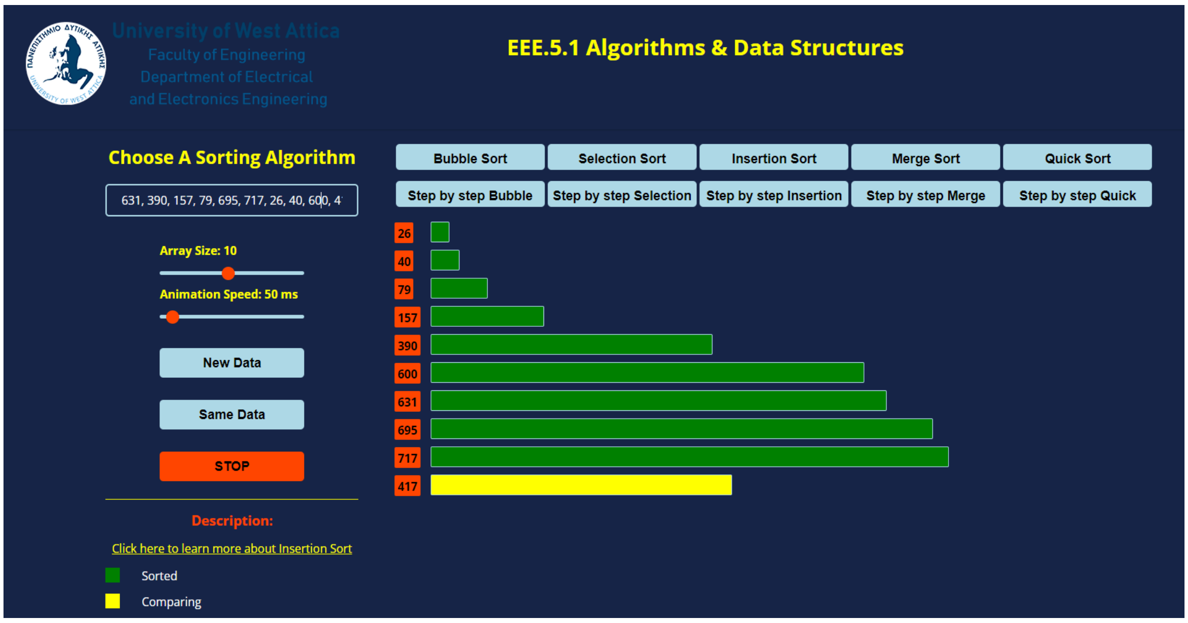


Figure 5.50: *INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.*

Next, a comparison is being made between the ninth element, 600, and the elements to its left, meaning the sorted portion of the array. Since 600 is larger than 40, 79, 157, and 390, it is inserted to the right of the sorted portion of the array:



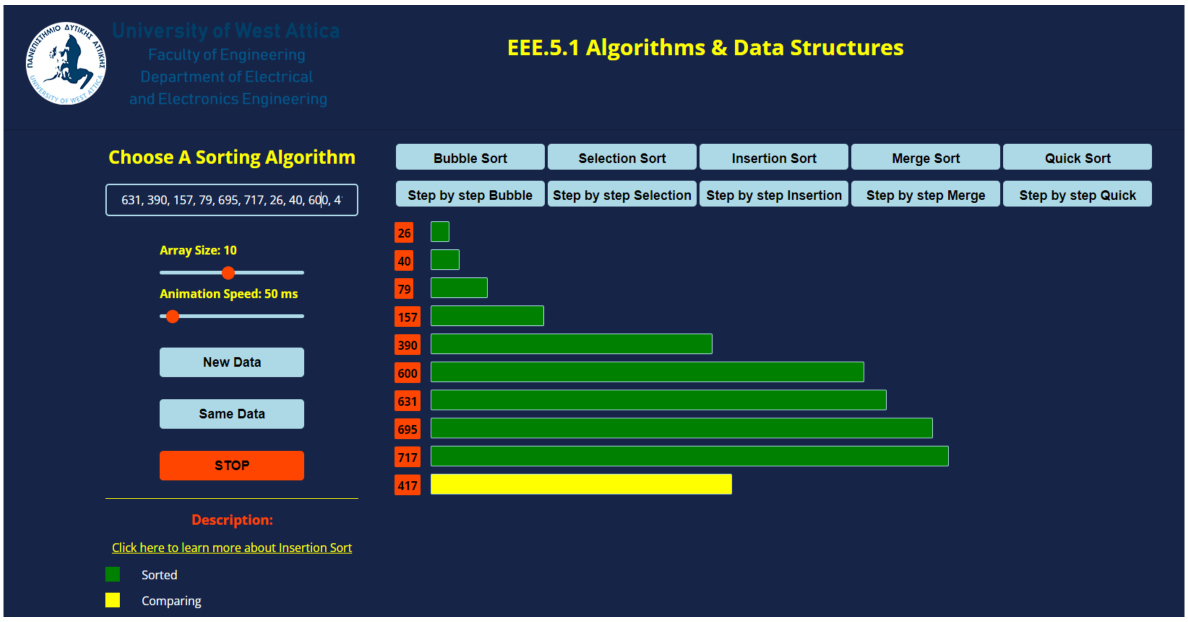
631, 390, 157, 79, 695, 717, 26, 40, 600, 4

Array Size: 10

Animation Speed: 50 ms

Figure 5.51: INSERTION SORT: The ninth element is now considered part of the sorted portion of the array.

Finally on the ninth and last iteration, we end the process with the tenth element of the array, which is 417. This element is considered “sorted”. Figure 5.52 shows with color yellow the element that will be compared with the sorted portion of the array, which is denoted by the color green.



631, 390, 157, 79, 695, 717, 26, 40, 600, 4

Array Size: 10

Animation Speed: 50 ms

Figure 5.52: INSERTION SORT: Color yellow demonstrates the element that will be compared with the sorted portion of the array.

Next, a comparison is being made between the tenth element, 417, and the elements to its left, meaning the sorted portion of the array. Since 40 is larger than 390, but smaller than 600, 79 is shifted to the right of the sorted array:

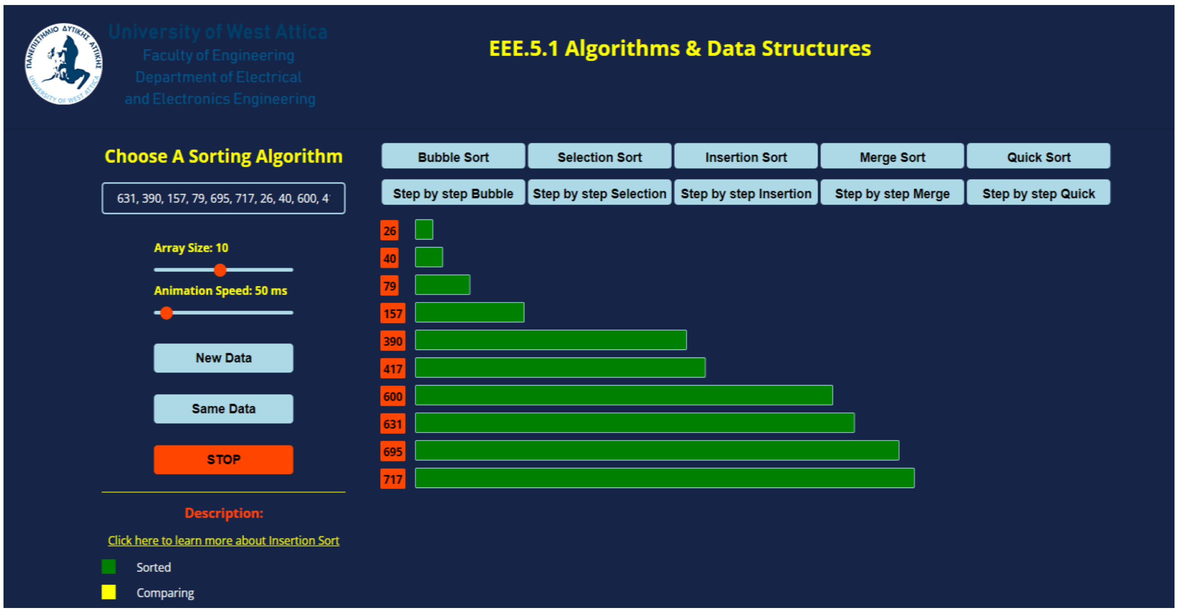


Figure 5.53: INSERTION SORT: The final sorted array.

5.5 Merge Sort

The following colors are used in merge sort:

- Light blue: The initial color of the bars used to represent the elements in the array. This makes it easier for the student in distinguishing between elements that have already been sorted and those that are still through processing.
- Red: The right subarray of the merge sort is indicated by the color red. This is one of the two smaller arrays that are produced as a result of the merge sort algorithm’s division of the initial array in half.
- Yellow: The yellow color is used to denote the left subarray that is created when the original array is divided in half during the first step of the merge sort algorithm.
- Light green: The sorted final subarray is indicated by the light green color. This is the result of the two smaller arrays being merged back together and sorted in ascending order.
- Green: The green color is used to indicate that an element is sorted. When an element is position correctly, it is said to be sorted and is no longer a part of the array’s unsorted portion.

We will analyze the application’s algorithm using a ten-element array. This is how the algorithm operates:

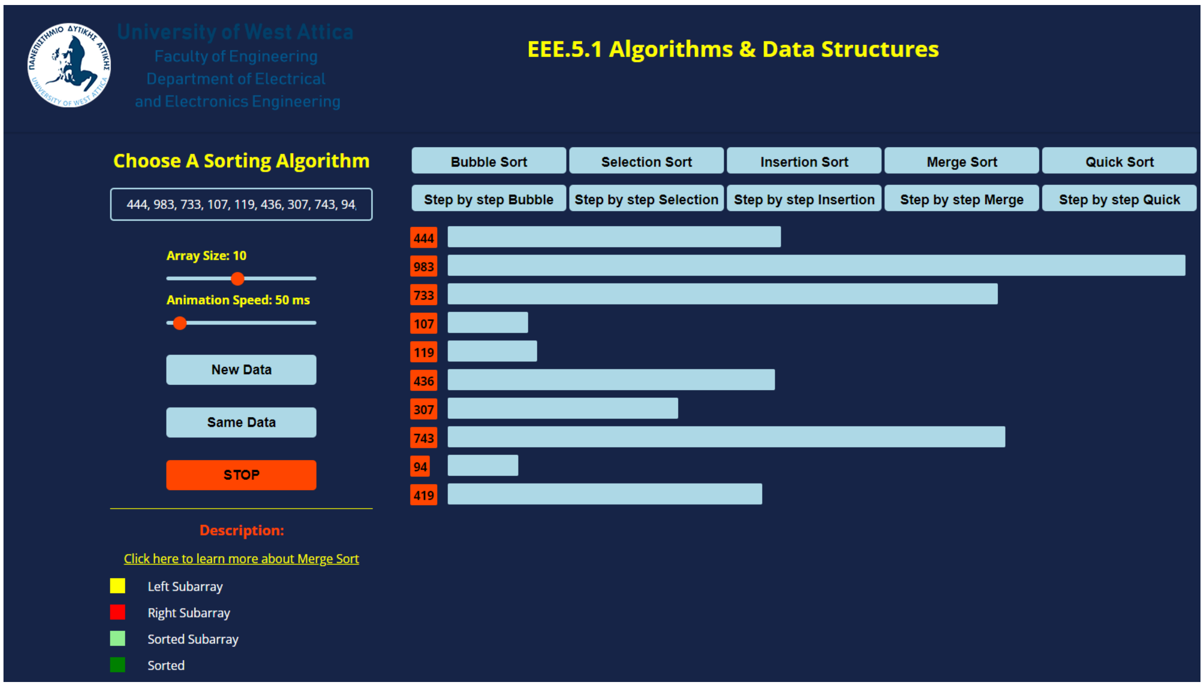
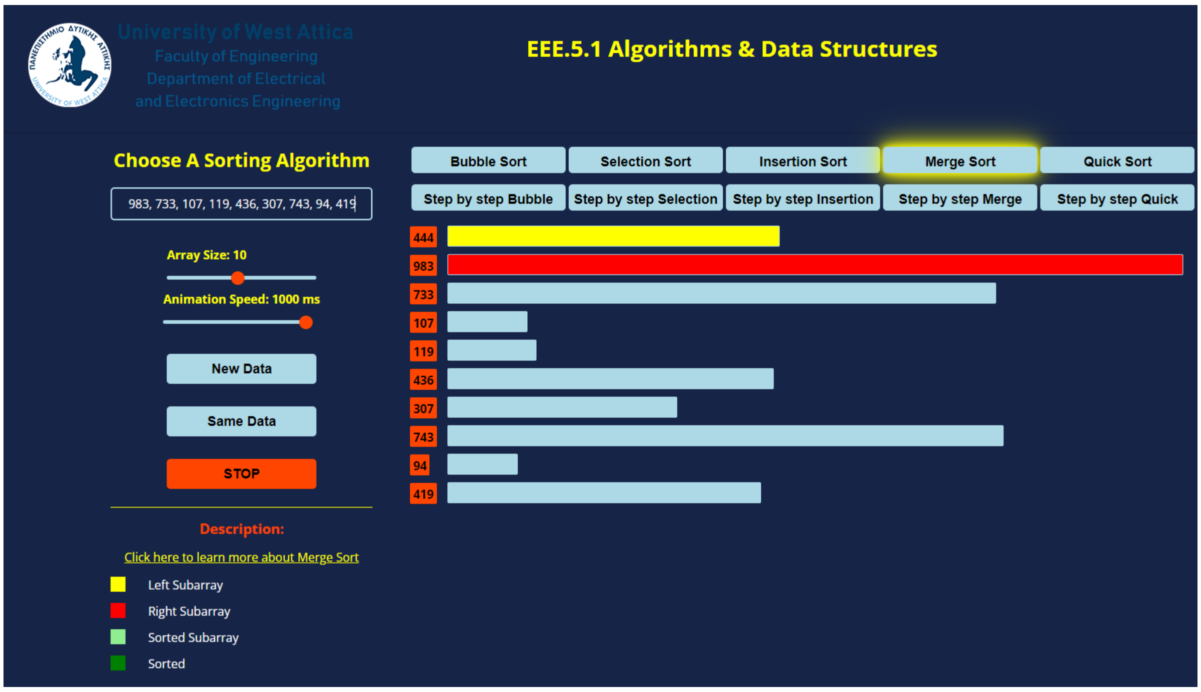


Figure 5.54: MERGE SORT: Array consists of elements: 444, 983, 733, 107, 119, 436, 307, 743, 94, 419.

As was already described, the recursive algorithm first divides the array in half to create two smaller arrays: [444, 983, 733, 107, 119] and [436, 307, 743, 94, 419]. The two smaller arrays are then further divided until there is only one element in each array: [444], [983], [733], [107], [119], [436], [307], [743], [94], [419].

Next, the algorithm merges the arrays back together, comparing the elements at the beginning of each array and taking the smaller of the two to put back into the main array.

The first step on merging the arrays back together is to merge the first two elements, 444 and 983.



983, 733, 107, 119, 436, 307, 743, 94, 419

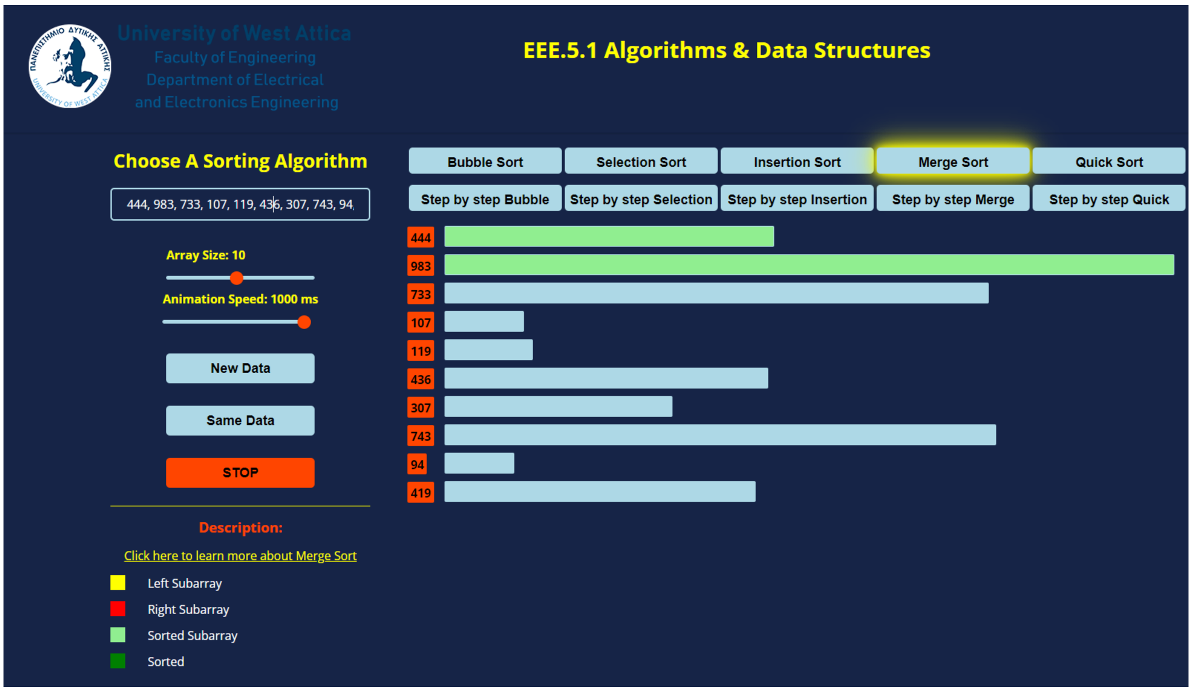
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.55: MERGE SORT: Element 444 is the left subarray and element 983 is the right subarray.

The next step would be to sort those single elements into one subarray.



444, 983, 733, 107, 119, 436, 307, 743, 94,

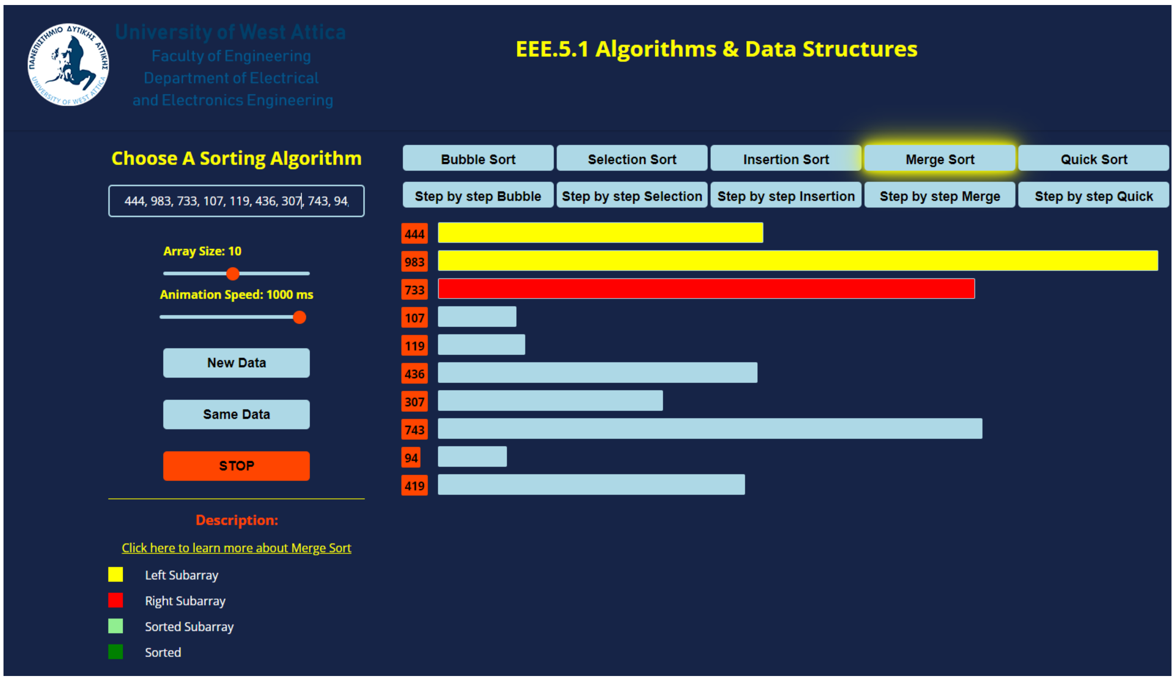
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.56: MERGE SORT: Sorted subarray consisting of elements 444 and 983.

The next single element is the 733 will be considered as the right subarray, which will be merge with the aforementioned subarray, which now will be the left sublist.



444, 983, 733, 107, 119, 436, 307, 743, 94,

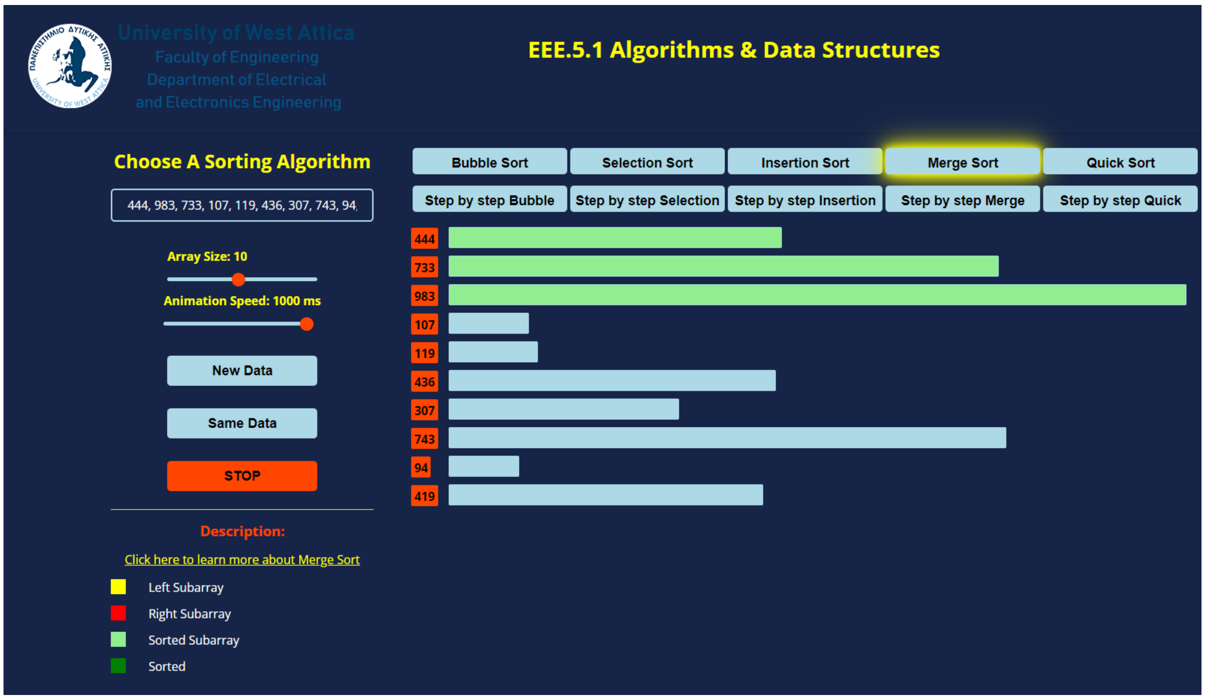
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.57: MERGE SORT: The left sorted subarray and the right subarray.

The next step would be to sort those elements into one sublist.



444, 983, 733, 107, 119, 436, 307, 743, 94,

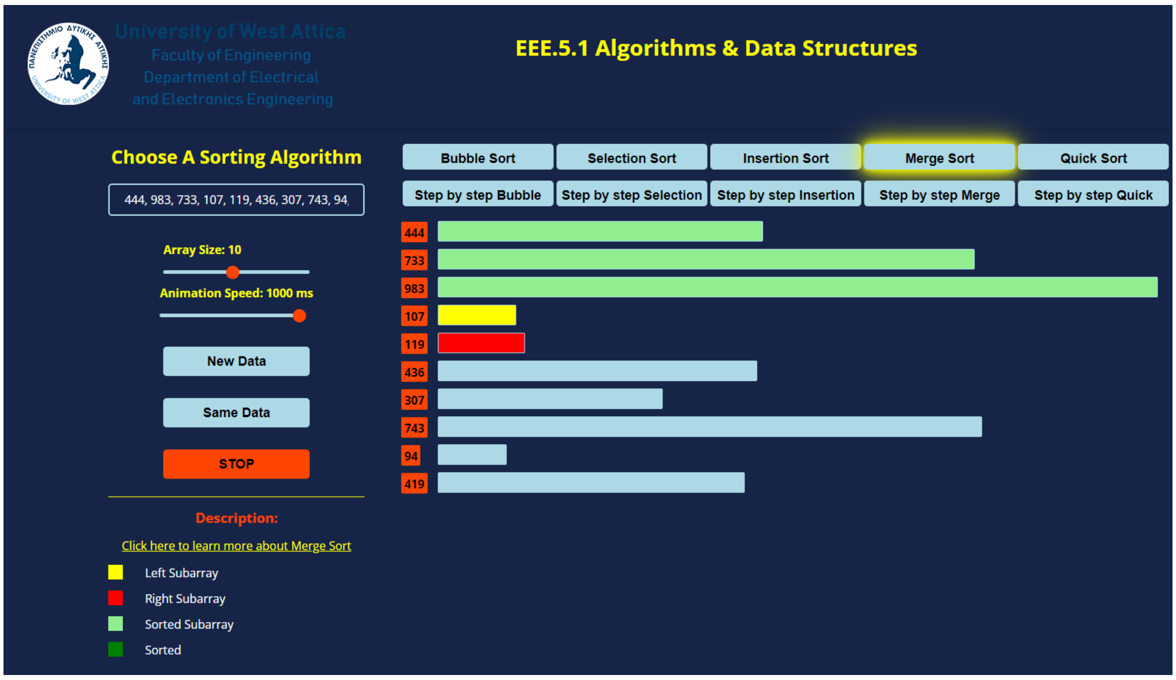
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.58: MERGE SORT: Sorted subarray consisting of elements 444 and 733 and 983.

Continuing with the process, the next two single elements that are the next to be merged are 107 and 119.

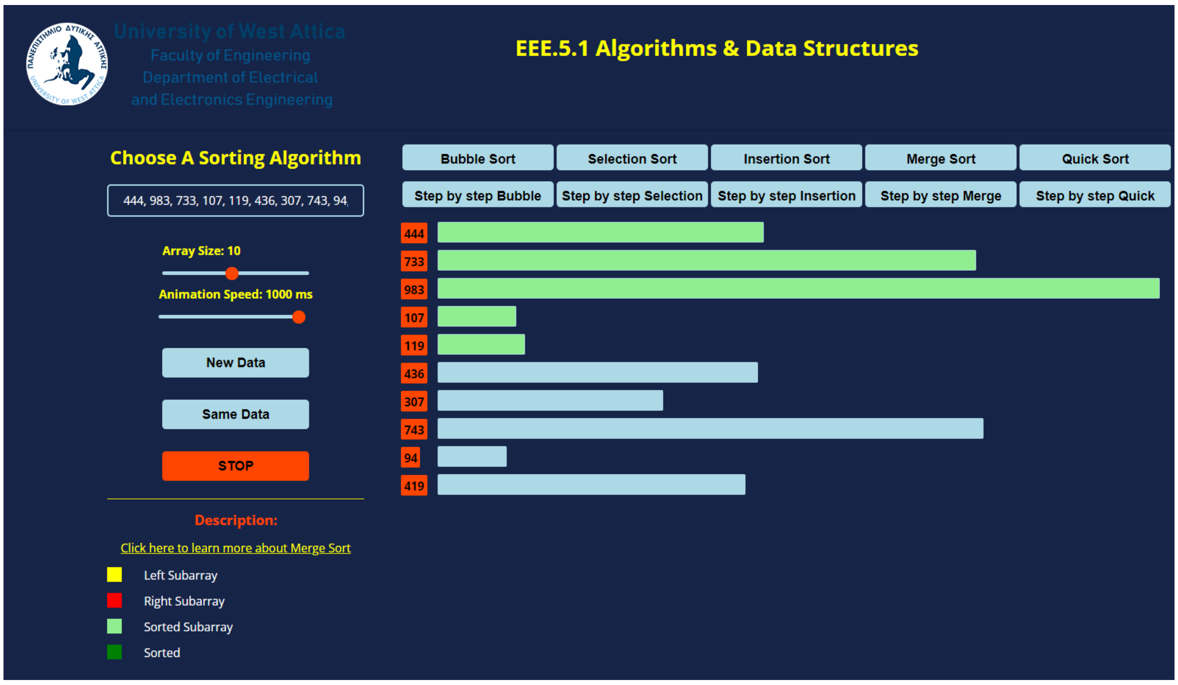


444, 983, 733, 107, 119, 436, 307, 743, 94

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.59: MERGE SORT: Elements 107 and 117 are being merged into a subarray.

At this point, as we can see, the algorithm has covered the elements of the first half of the main array.



444, 983, 733, 107, 119, 436, 307, 743, 94

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.60: MERGE SORT: Elements 107 and 119 are in a sorted subarray.

To fully cover the first half of the original array, the algorithm’s next step is to merge those two subarrays, [444, 7333, 983] with [107, 119] as seen in Figure 5.61.

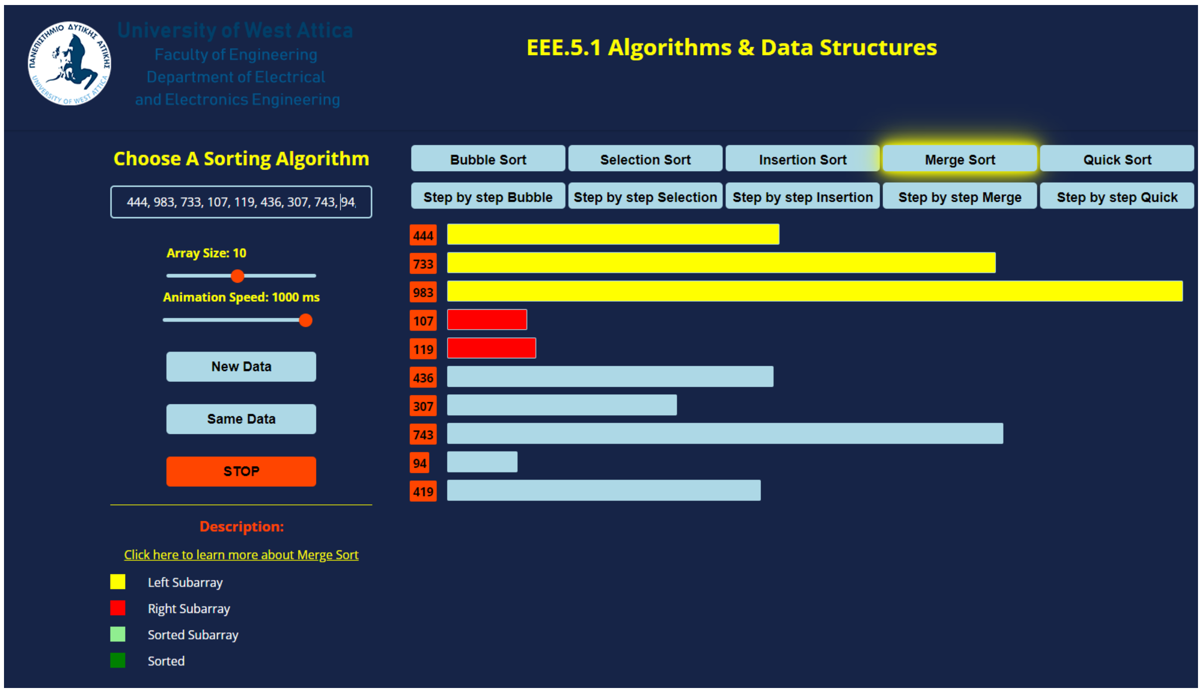


Figure 5.61: MERGE SORT: The left and right subarray of the first half of the main array.

The first sublist consisting of the first half of the main array is depicted in Figure 5.62:

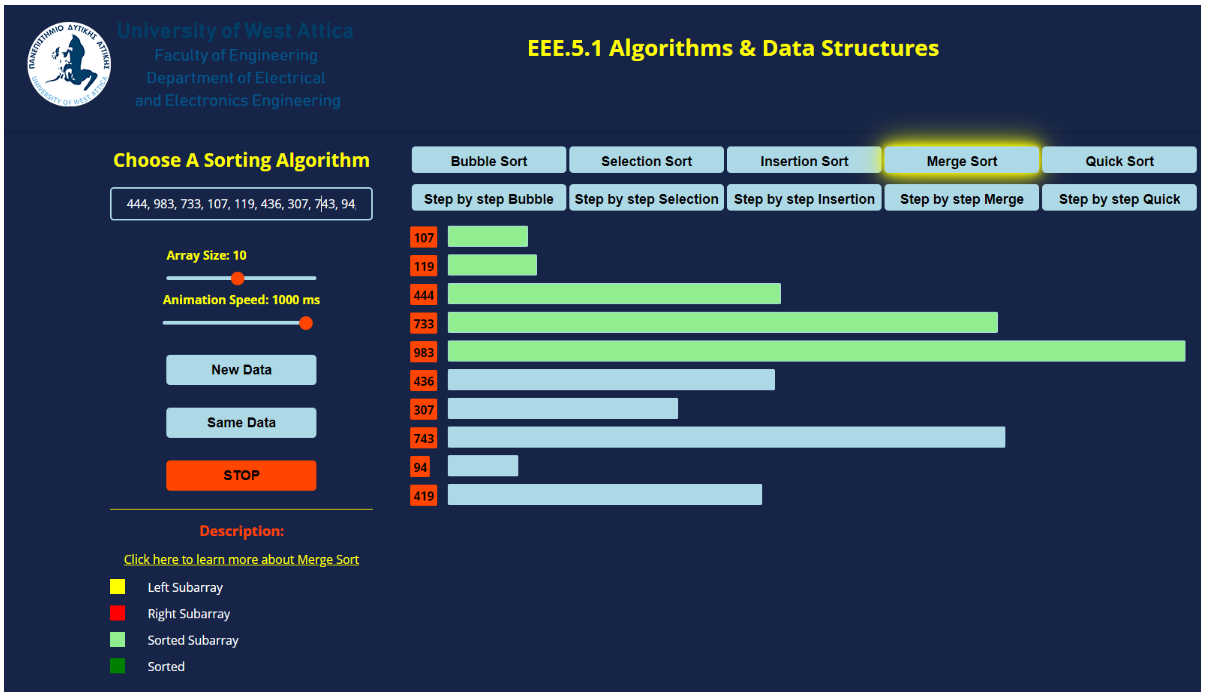
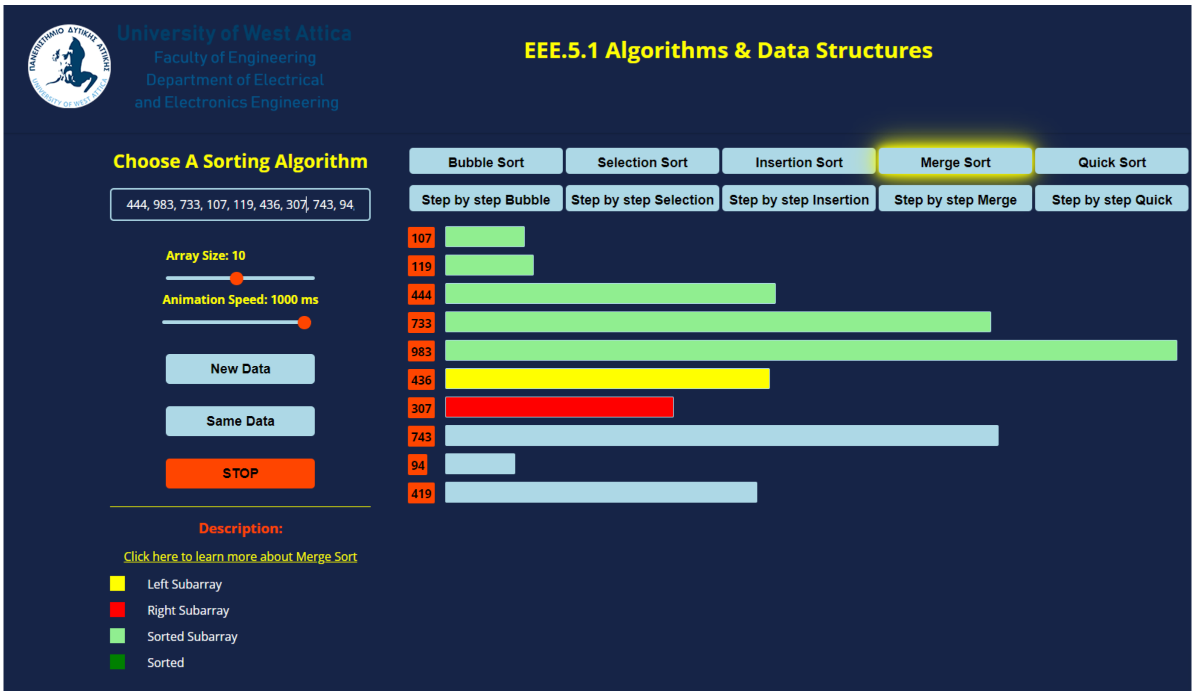


Figure 5.62: MERGE SORT: The first subarray.

The next step, as was already stated, is now for the second out of the two subarrays to be sorted. First the elements 436 and 307 will be formed into one sublist.



444, 983, 733, 107, 119, 436, 307, 743, 94

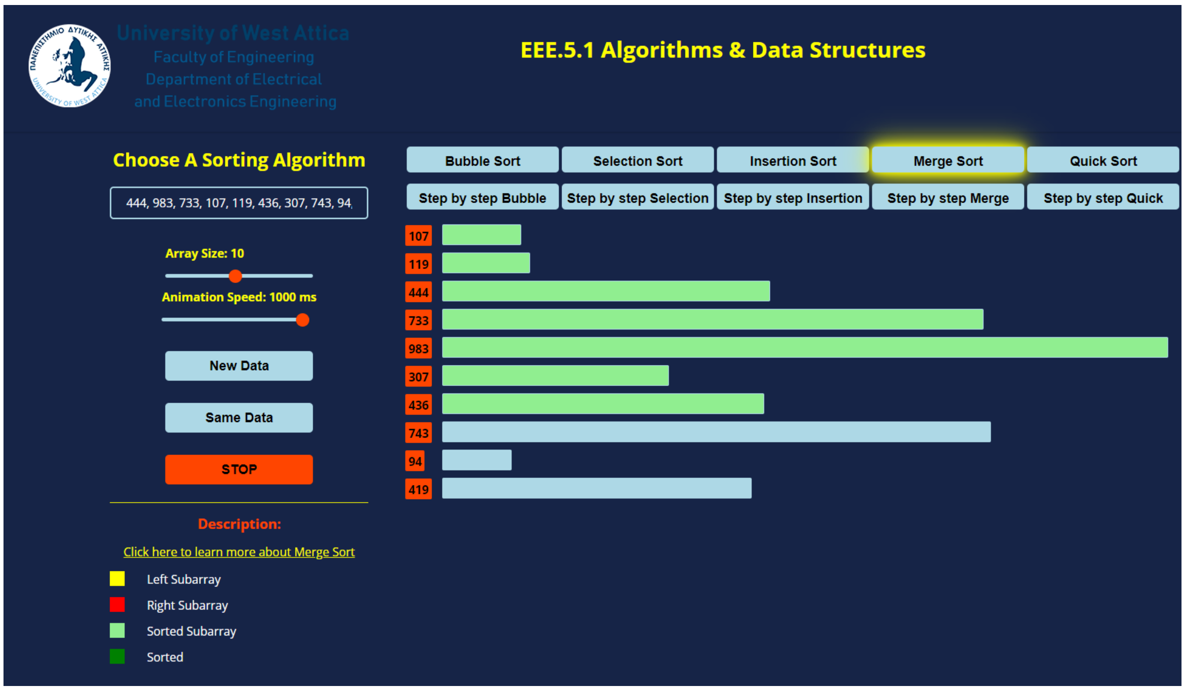
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.63: MERGE SORT: Merge of the elements 436 and 307.

Elements 436 and 307 form a subarray as seen in Figure 5.64:



444, 983, 733, 107, 119, 436, 307, 743, 94

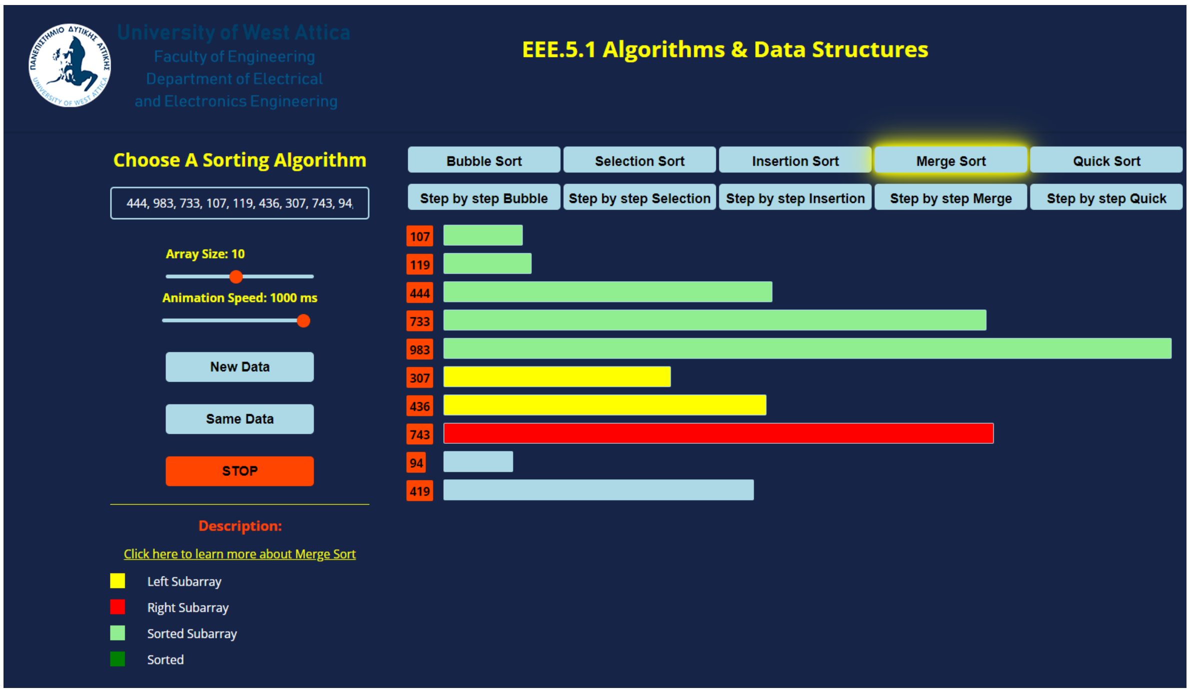
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.64: MERGE SORT: Elements 436 and 307 are in a sorted subarray.

Next step of the algorithm is to merge the element 743 which is considered the right subarray, with the aforementioned array.



444, 983, 733, 107, 119, 436, 307, 743, 94,

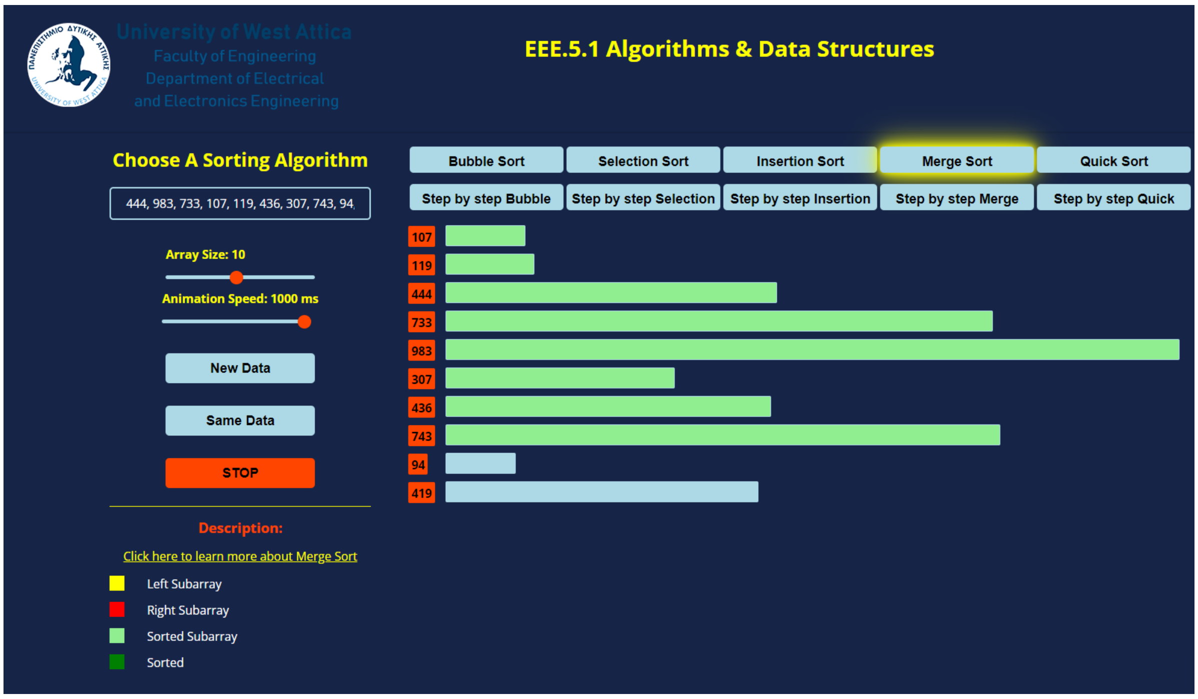
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.65: MERGE SORT: Left subarray consisting of elements 307 and 436 is merged with element 743.

The value 743 merged with the left subarray of 307 and 436 is giving us the following result:



444, 983, 733, 107, 119, 436, 307, 743, 94,

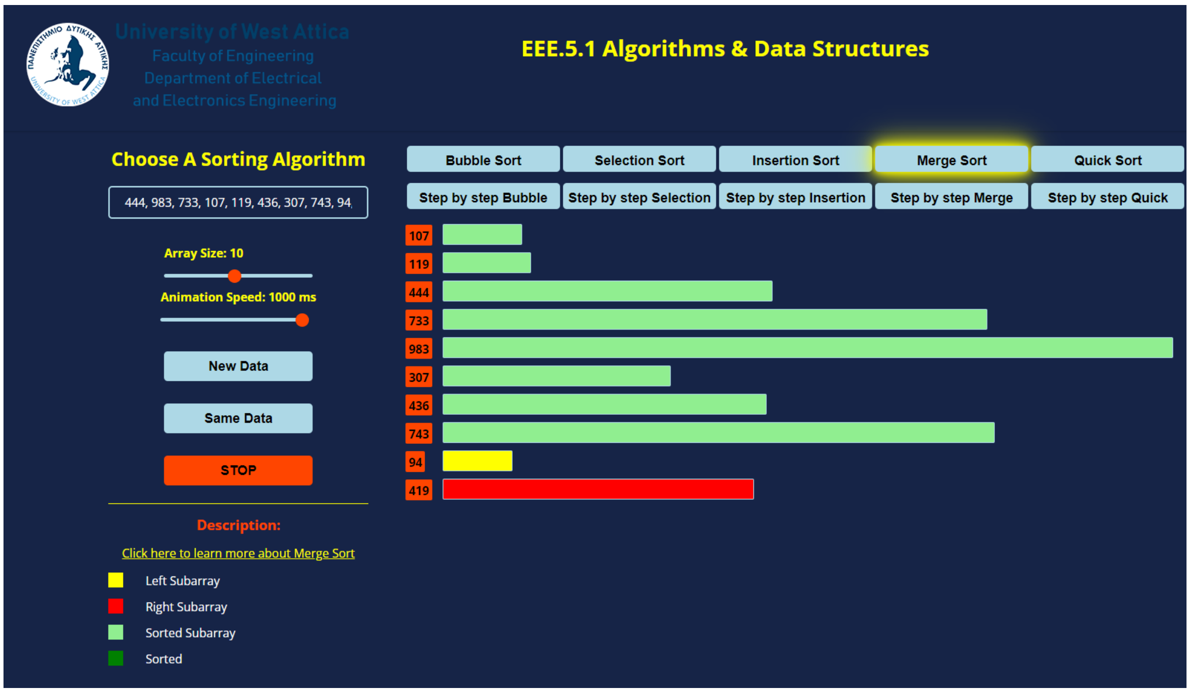
Array Size: 10

Animation Speed: 1000 ms

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.66: MERGE SORT: Element 743 is merged with the subarray of 307 and 436.

The last two elements of the array, 94 and 419 are next to be merged in a subarray.



444, 983, 733, 107, 119, 436, 307, 743, 94,

Array Size: 10

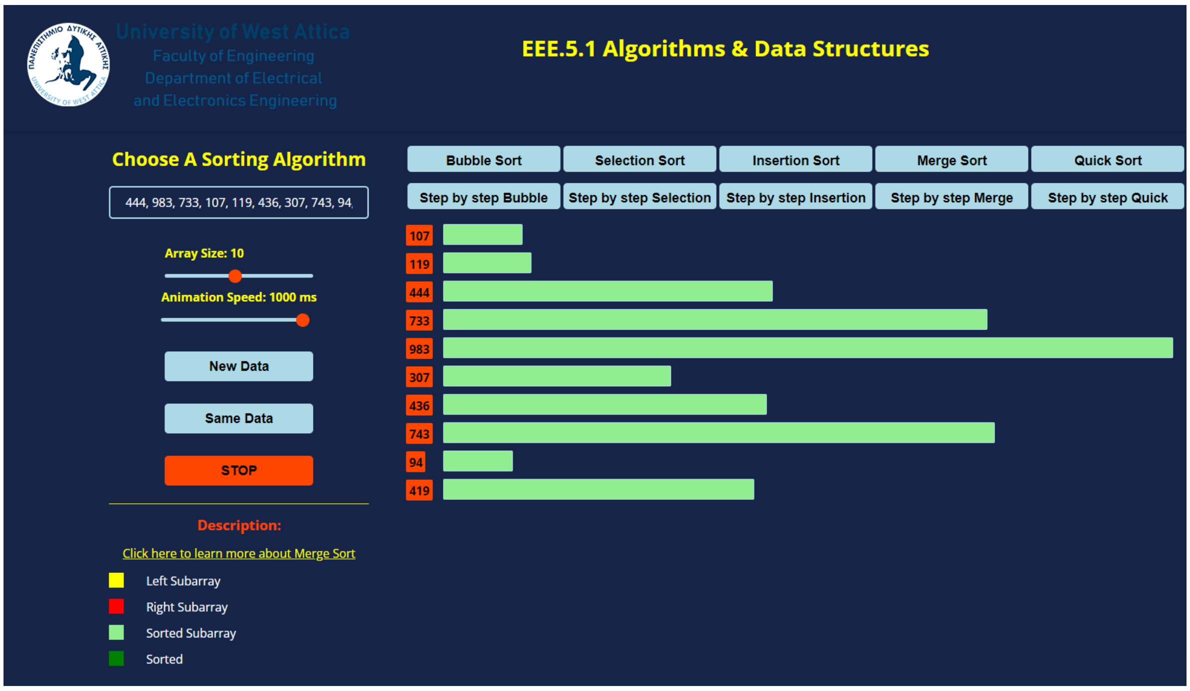
Animation Speed: 1000 ms

107
119
444
733
983
307
436
743
94
419

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.67: MERGE SORT: Element 94 is considered the left subarray and element 419 the right subarray.

The merge of the single elements 94 and 419 result in the following subarray:



444, 983, 733, 107, 119, 436, 307, 743, 94,

Array Size: 10

Animation Speed: 1000 ms

107
119
444
733
983
307
436
743
94
419

🟨 Left Subarray
🟥 Right Subarray
🟩 Sorted Subarray
🟩 Sorted

Figure 5.68: MERGE SORT: Elements 94 and 419 are merged into a subarray as denoted by the color light green.

The next step would be to merge the left subarray of 307, 436 and 743 with the right subarray of 94 and 419 into one, which would be the right sublist of the main array.

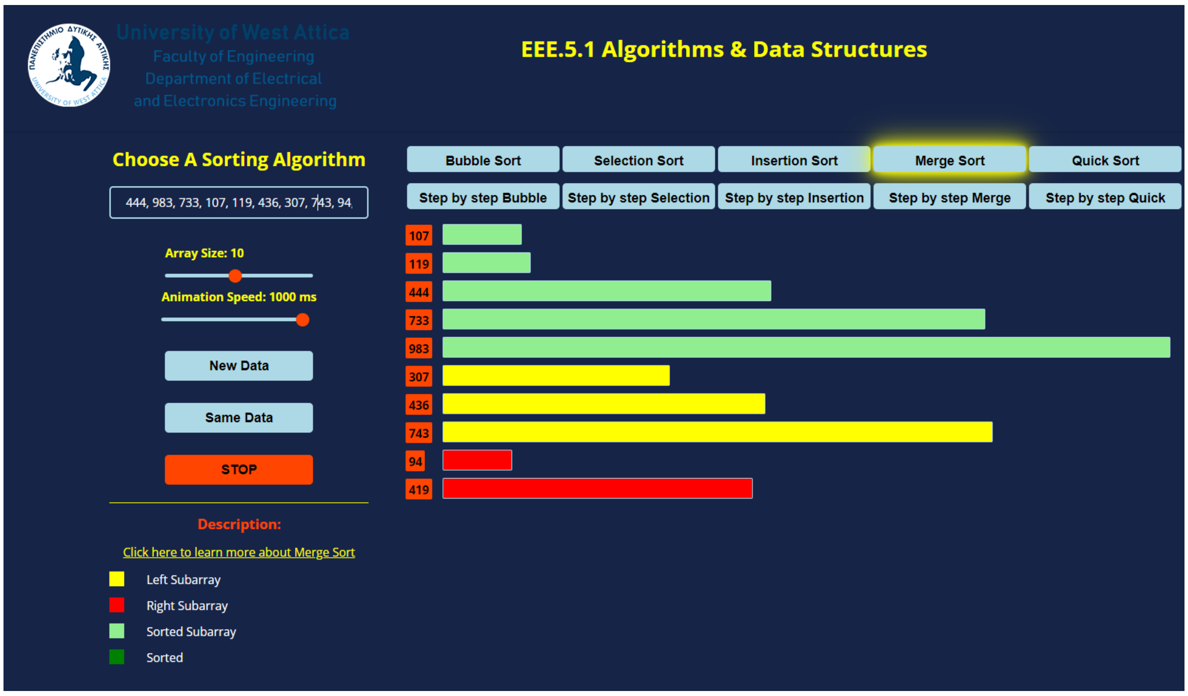


Figure 5.69: MERGE SORT: Merging the two subarrays to form the right subarray of the original array.

The two subarrays of the main array, before being merged into one final sorted array, are shown in Figure 5.70:

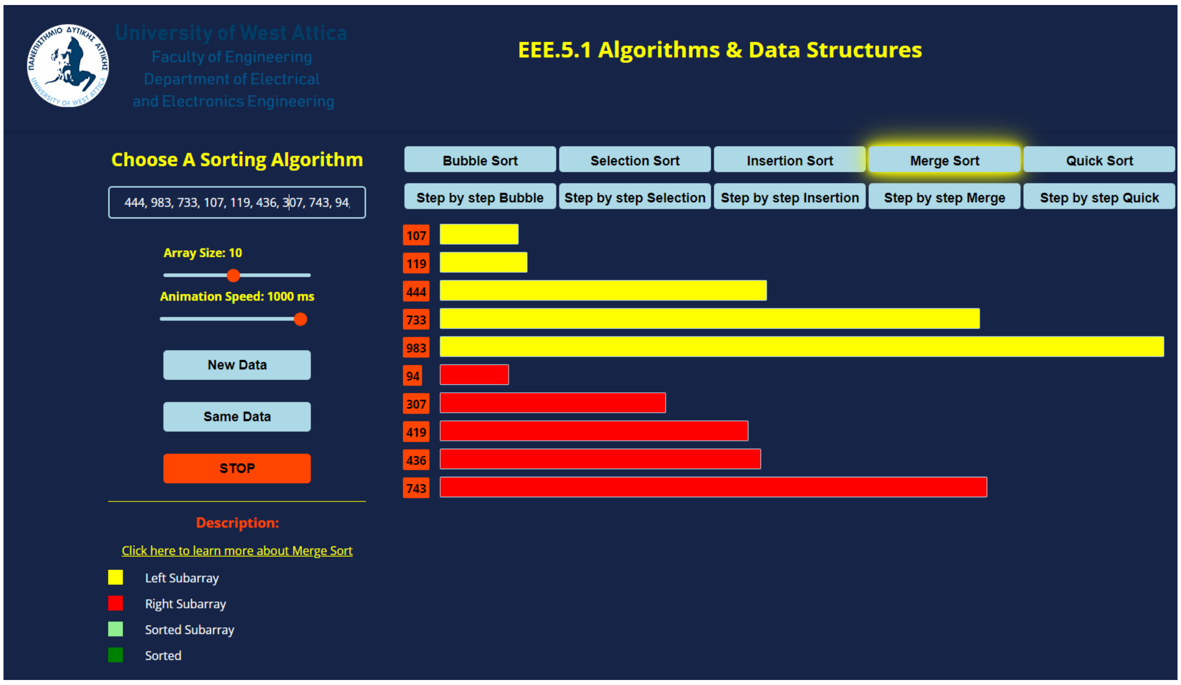


Figure 5.70: MERGE SORT: The left and right sorted subarrays.

The final step of the algorithm is to merge the two sorted subarrays into one final array, which is the sorted version of the original array. The merging at this point is done by comparing element by element the two sorted subarrays. The final sorted array is shown in Figure 5.71:

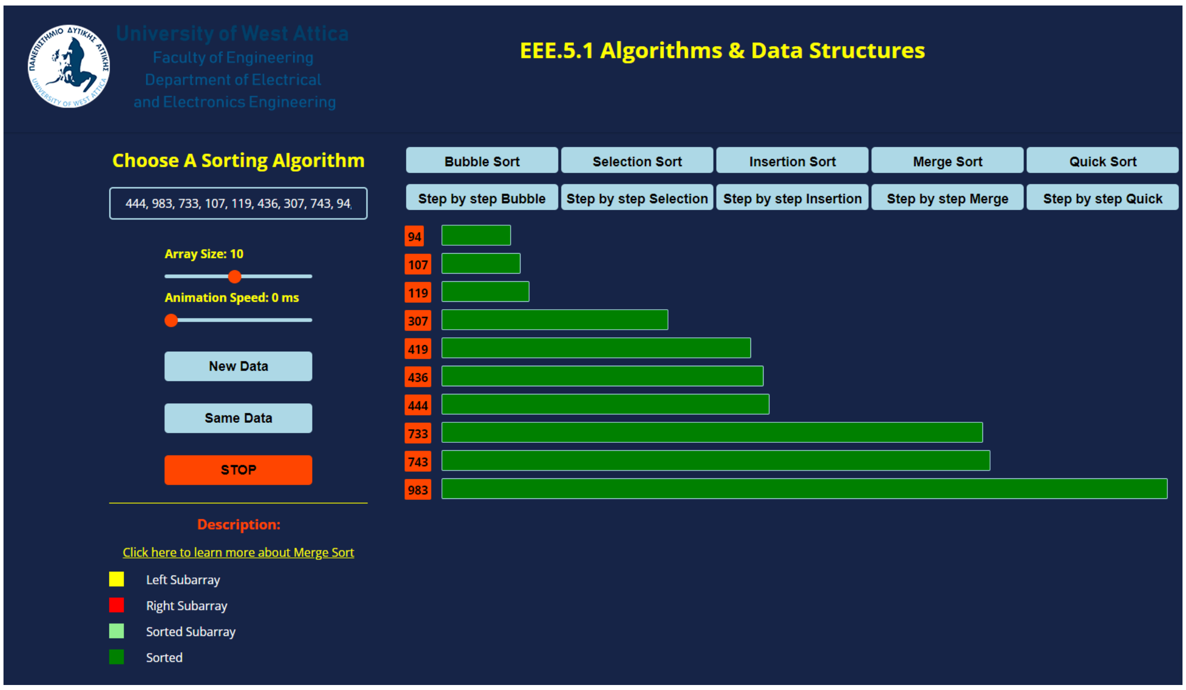


Figure 5.71: MERGE SORT: The final sorted array.

5.6 Quick Sort

The following colors are used in quick sort:

- Light blue: The initial color of the bars used to represent the elements in the array. This makes it easier for the student in distinguishing between elements that have already been sorted and those that are still through processing.
- Red: The pivot element is identified by the color red. This is the element around which the array is partitioned and sorted.
- Yellow: Elements that are being compared to the pivot element and are found to be less than the pivot element are indicated by the color yellow. During the partitioning step of the quick sort algorithm, these elements are placed into the left subarray.
- Salmon: The salmon color is used to denote elements that are being compared to the pivot element and are found to be greater than the pivot element. These elements are placed in the right subarray during the partitioning step of the quick sort algorithm.
- Green: The green color is used to indicate that an element is sorted. When an element is position correctly, it is said to be sorted and is no longer a part of the array’s unsorted portion.

We will examine the algorithm within the application with an array of ten elements. This is how the algorithm operates:

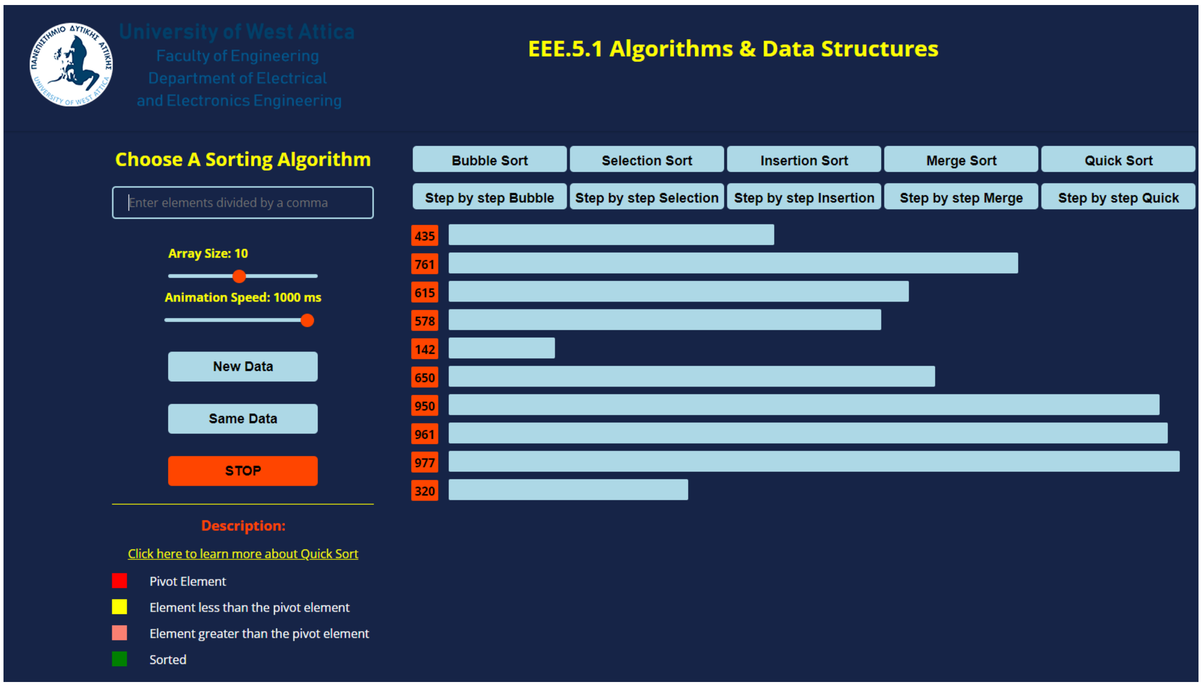


Figure 5.72: QUICK SORT: Array consists of elements: 435, 761, 615, 578, 142, 650, 950, 961, 977, 320.

I = 0:

Starting off the algorithm, the pivot element is selected as the last element on the array, 320. PivotIndex = 0

Element at index 0 = 435 is greater than pivot (320), so pivotIndex is not incremented.

I = 1:

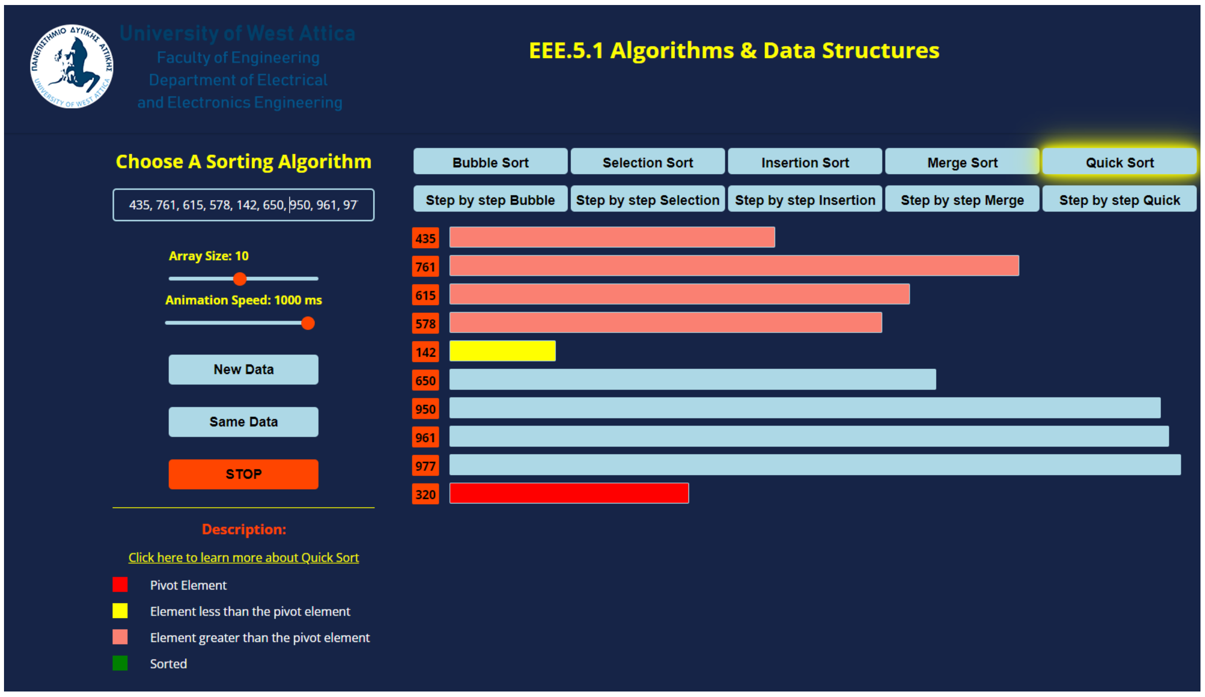
Element at index 1 = 761 is greater than pivot (320), so pivotIndex is not incremented.

I = 2:

Element at index 2 = 615 is greater than pivot (320), so pivotIndex is not incremented.

I = 3:

Element at index 3 = 578 is greater than pivot (320), so pivotIndex is not incremented.



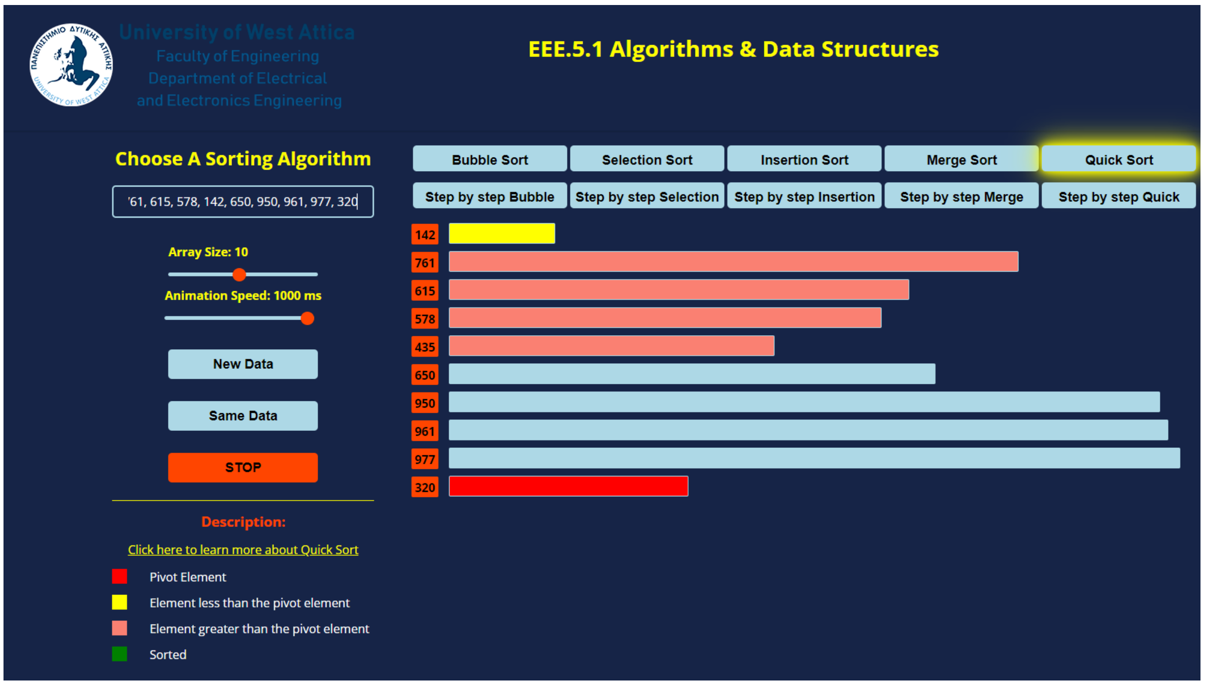
🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.73: QUICK SORT: Pivot element is 320.

I = 4:

Element at index 4 = 142 is less than pivot (320), so it is swapped with the element at PivotIndex = 435.

PivotIndex is incremented. PivotIndex = 1



🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.74: QUICK SORT: 142 is swapped with 435.

I = 5:

Element at index 5 = 650 is greater than pivot (320), so pivotIndex is not incremented.

I = 6:

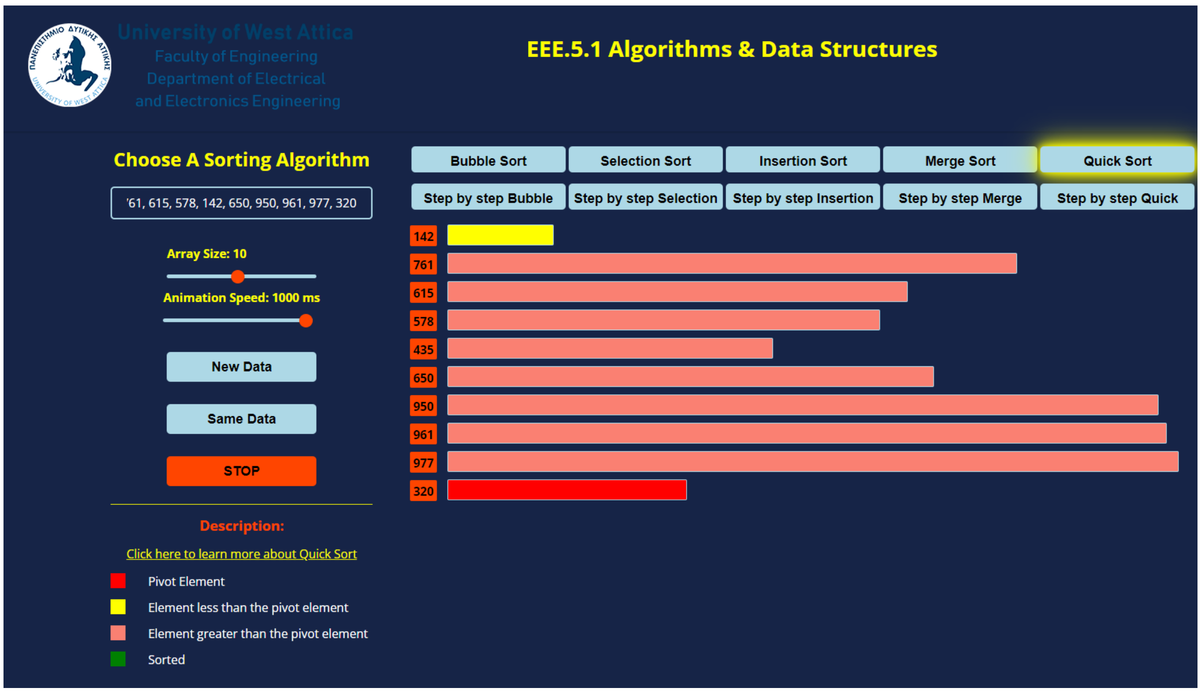
Element at index 6 = 950 is greater than pivot (320), so pivotIndex is not incremented.

I = 7:

Element at index 3 = 961 is greater than pivot (320), so pivotIndex is not incremented.

I = 8:

Element at index 3 = 977 is greater than pivot (320), so pivotIndex is not incremented.

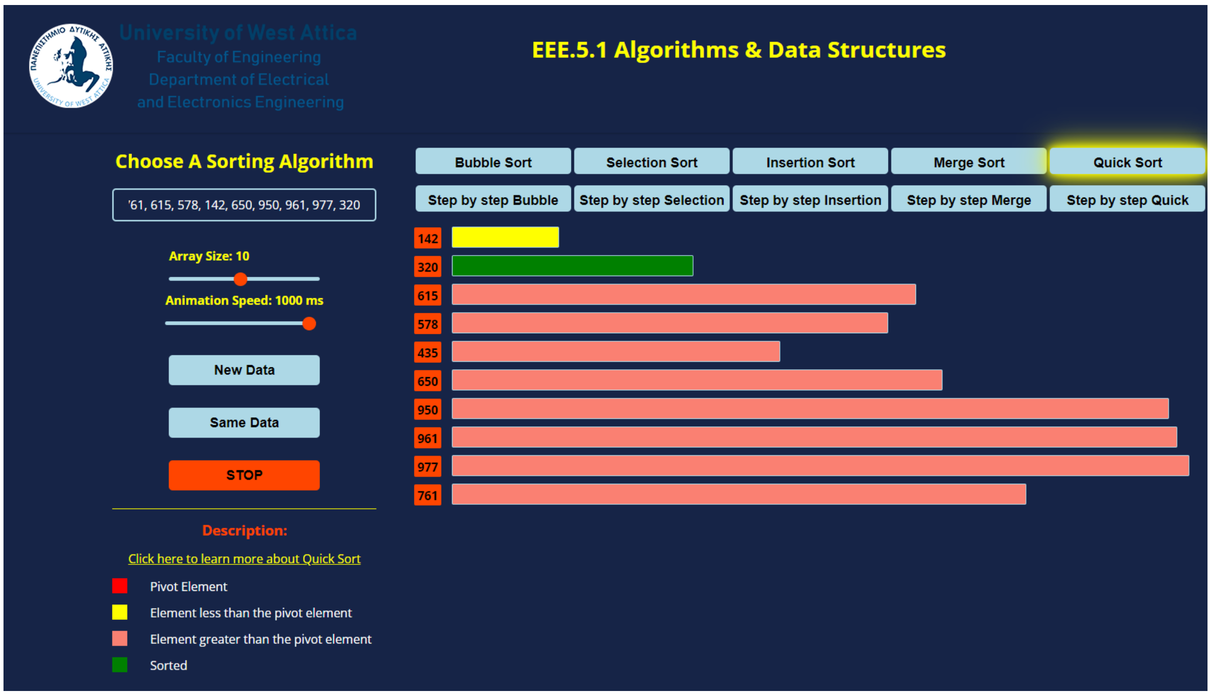


🟥 Pivot Element
🟨 Element less than the pivot element
▨ Element greater than the pivot element
🟩 Sorted

Figure 5.75: QUICK SORT: The for loop has reached the pivot element.

I = 9:

The for loops terminates. At this point the pivot element (320) is swapped with the element at PivotIndex = 761.



'61, 615, 578, 142, 650, 950, 961, 977, 320

Array Size: 10

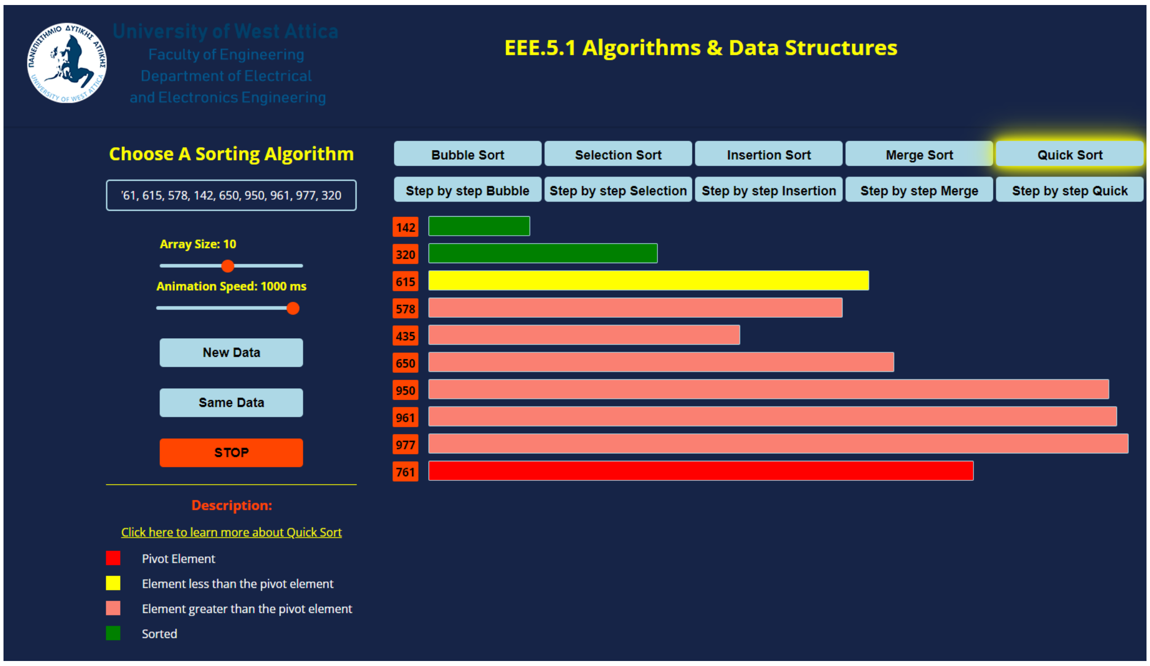
Animation Speed: 1000 ms

🟥 Pivot Element
🟨 Element less than the pivot element
▨ Element greater than the pivot element
🟩 Sorted

Figure 5.76: QUICK SORT: Pivot element is in the correct position.

At this point, the array is partitioned into two sublists: [142] which is less than the pivot element and [615, 578, 435, 650, 950, 961, 977, 761] which are all the elements that are greater than the pivot element. The sublists on either side of pivot are then recursively sorted using quick sort.

For the sublist [142]: Since the sublist consists of only a single element, is already sorted.



'61, 615, 578, 142, 650, 950, 961, 977, 320

Array Size: 10

Animation Speed: 1000 ms

🟥 Pivot Element
🟨 Element less than the pivot element
▨ Element greater than the pivot element
🟩 Sorted

Figure 5.77: QUICK SORT: Element 142 is in the correct position.

For the sublist [615, 578, 435, 650, 950, 961, 977, 761]:

Pivot element is again the last element on the array, 761. PivotIndex = 2

- I = 2:

  Element at index 2 = 615 is less than pivot (761), so it is swapped with the element at PivotIndex = 615.

  PivotIndex is incremented.

  PivotIndex = 3

- I = 3:

  Element at index 3 = 578 is less than pivot (761), so it is swapped with the element at PivotIndex = 578.

  PivotIndex is incremented.

  PivotIndex = 4

- I = 4:

  Element at index 4 = 435 is less than pivot (761), so it is swapped with the element at PivotIndex = 435.

  PivotIndex is incremented.

  PivotIndex = 5

- I = 5:

  Element at index 5 = 650 is less than pivot (761), so it is swapped with the element at PivotIndex = 650.

  PivotIndex is incremented.

  PivotIndex = 6

- I = 6:

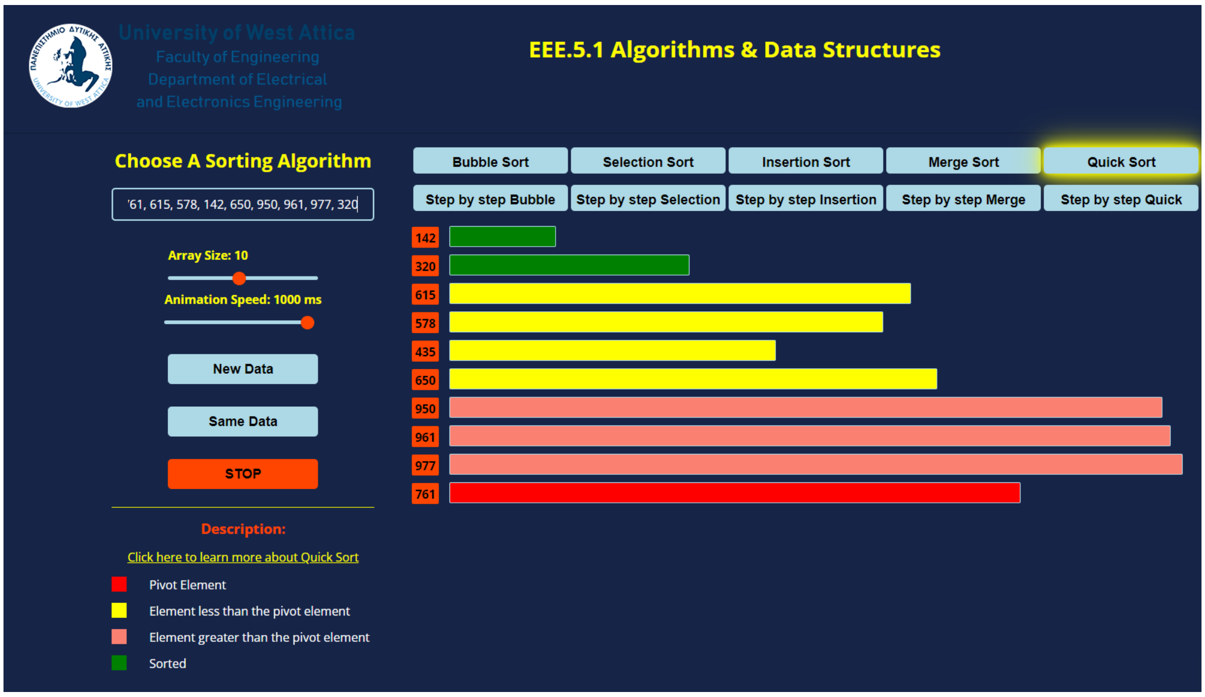
  Element at index 6 = 950 is greater than pivot (761), so pivotIndex is not incremented.

- I = 7:

  Element at index 7 = 961 is greater than pivot (761), so pivotIndex is not incremented.

- I = 8:

  Element at index 8 = 977 is greater than pivot (761), so pivotIndex is not incremented.



'61, 615, 578, 142, 650, 950, 961, 977, 320

Array Size: 10

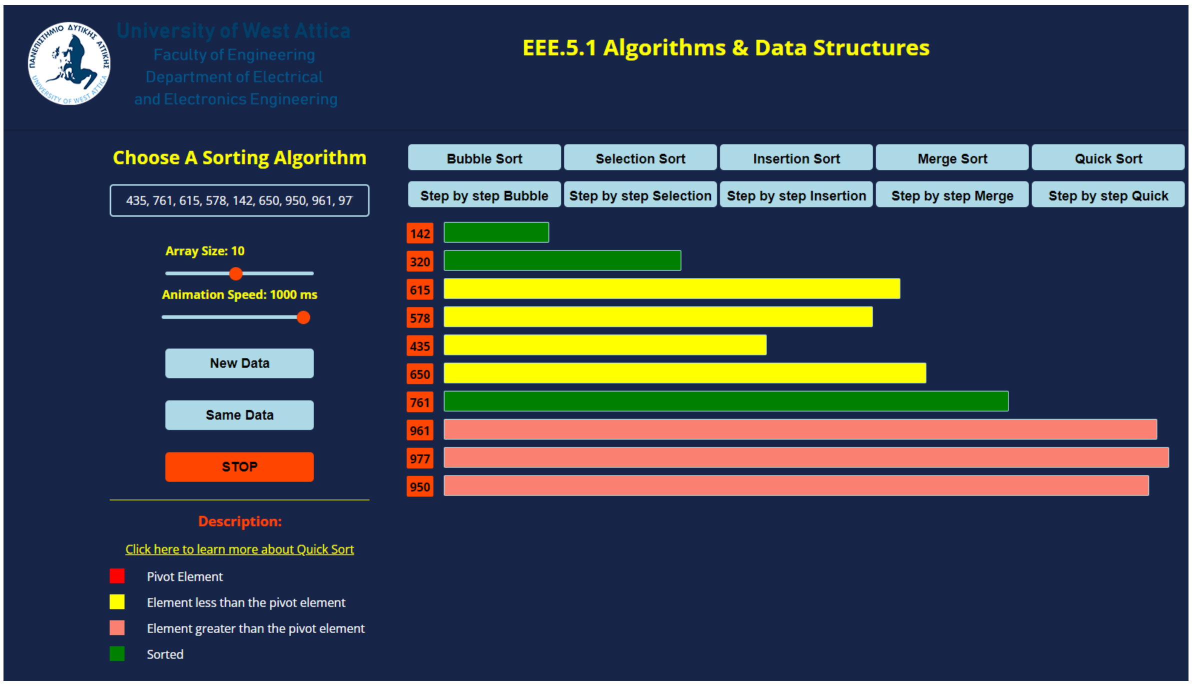
Animation Speed: 1000 ms

- 🟥 Pivot Element
- 🟨 Element less than the pivot element
- 🟧 Element greater than the pivot element
- 🟩 Sorted

Figure 5.78: QUICK SORT: Elements that are less and greater than pivot element, 761.

- I = 9:

The for loop terminates. At this point the pivot element (761) is swapped with the element at PivotIndex = 650.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

- 🟥 Pivot Element
- 🟨 Element less than the pivot element
- 🟧 Element greater than the pivot element
- 🟩 Sorted

Figure 5.78: QUICK SORT: Pivot element, 761 is in the correct position.

Continuing, again we have two unsorted sublists: [615, 578, 435, 650] which is less than the pivot element and [961, 977, 950] which are all the elements that are greater than the pivot element. The sublists on either side of pivot are then recursively sorted using quick sort.

For the sublist [615, 578, 435, 650]:

Pivot element is again the last element on the array, 650. PivotIndex = 2.

- I = 2:

  Element at index 2 = 615 is less than pivot (650), so it is swapped with the element at PivotIndex = 615.

  PivotIndex is incremented.

  PivotIndex = 3

- I = 3:

  Element at index 3 = 578 is less than pivot (650), so it is swapped with the element at PivotIndex = 578.

  PivotIndex is incremented.

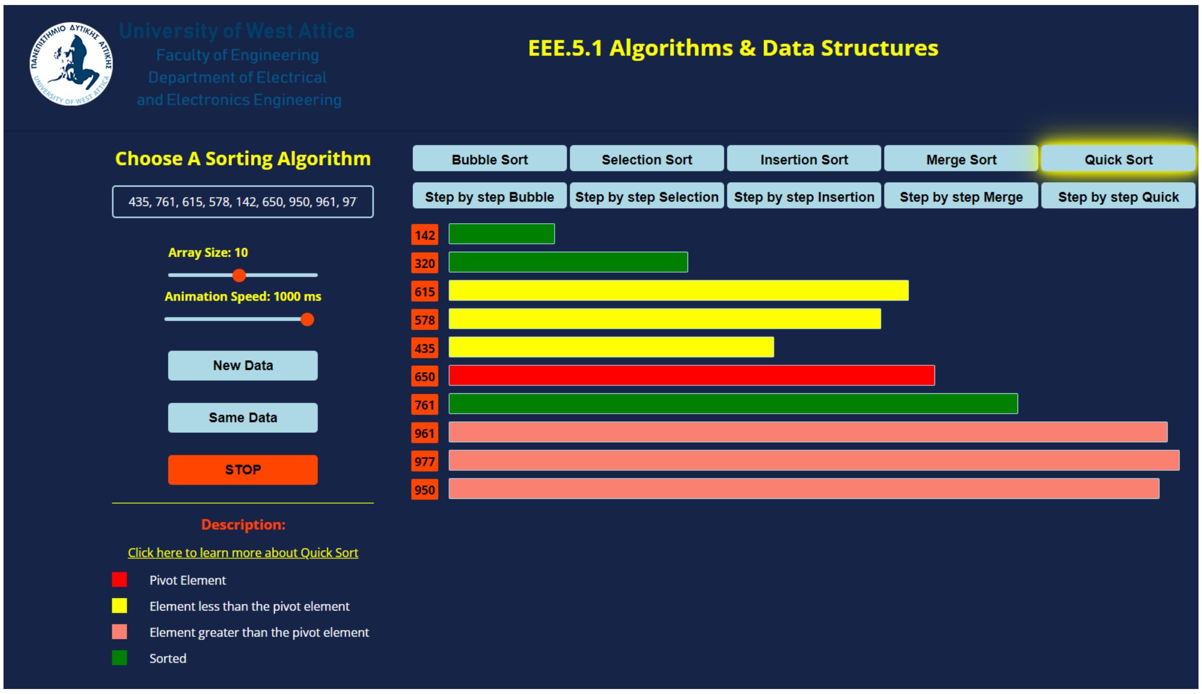
  PivotIndex = 4

- I = 4:

  Element at index 4 = 435 is less than pivot (650), so it is swapped with the element at PivotIndex = 435.

  PivotIndex is incremented.

  PivotIndex = 5



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

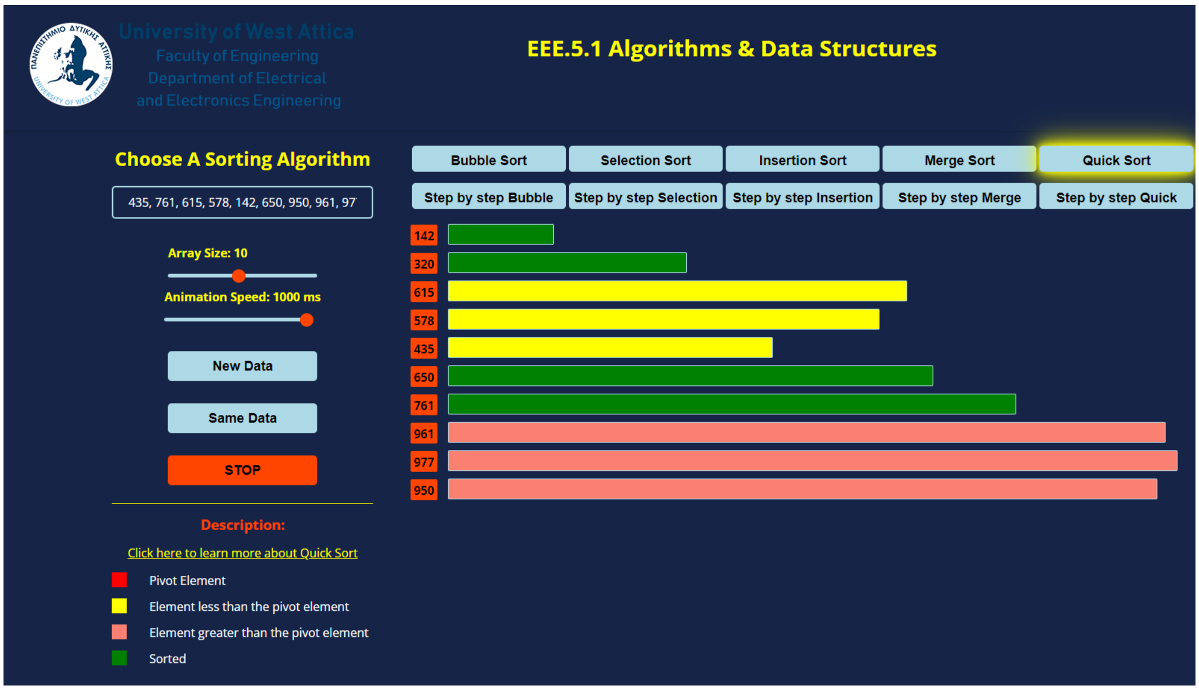
Description:

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.79: QUICK SORT: All the elements left than pivot, are less than 650.

- I = 5:

The for loop terminates. At this point the pivot element (650) is swapped with the element at PivotIndex = 650.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

Description:

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.80: QUICK SORT: Pivot element 650 is in the correct position.

For the sublist [615, 578, 435]:

Pivot element is again the last element on the array, 435. PivotIndex = 2.

- I = 2:

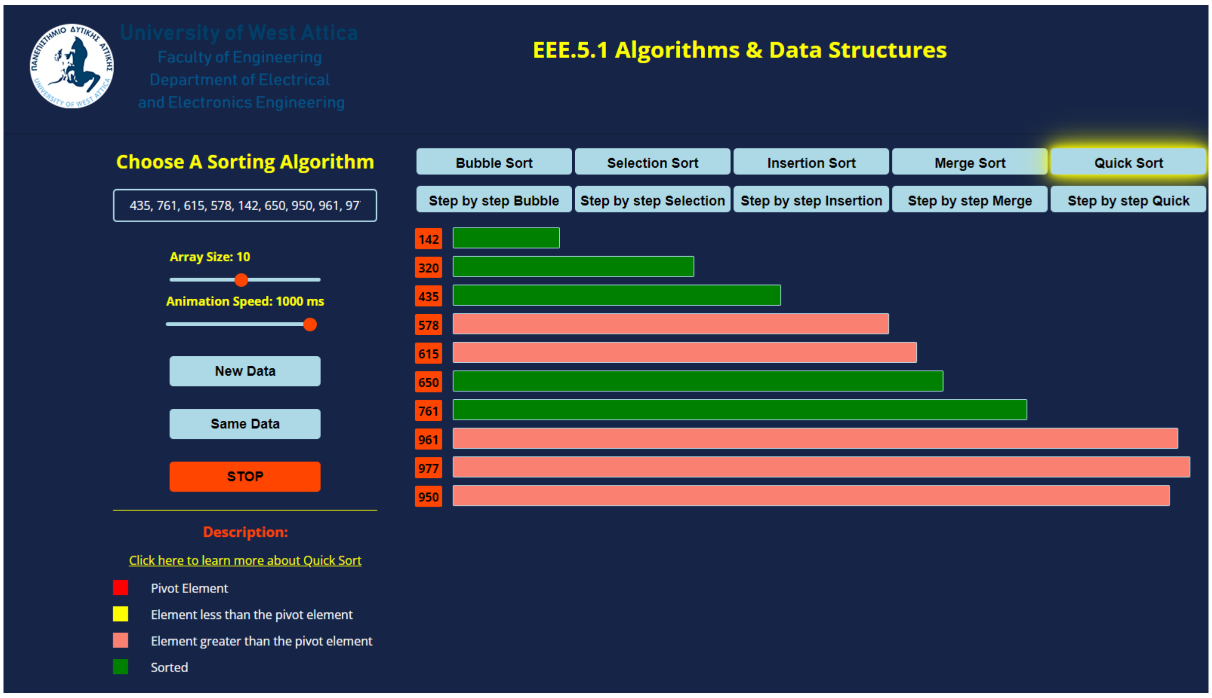
  Element at index 2 = 615 is greater than pivot (435), so pivotIndex is not incremented.

- I = 3:

  Element at index 3 = 578 is greater than pivot (435), so pivotIndex is not incremented.

- I = 4:

  The for loops terminates. At this point the pivot element (435) is swapped with the element at PivotIndex = 615.



EEE.5.1 Algorithms & Data Structures

Choose A Sorting Algorithm

435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

Description:

🟥 Pivot Element

🟨 Element less than the pivot element

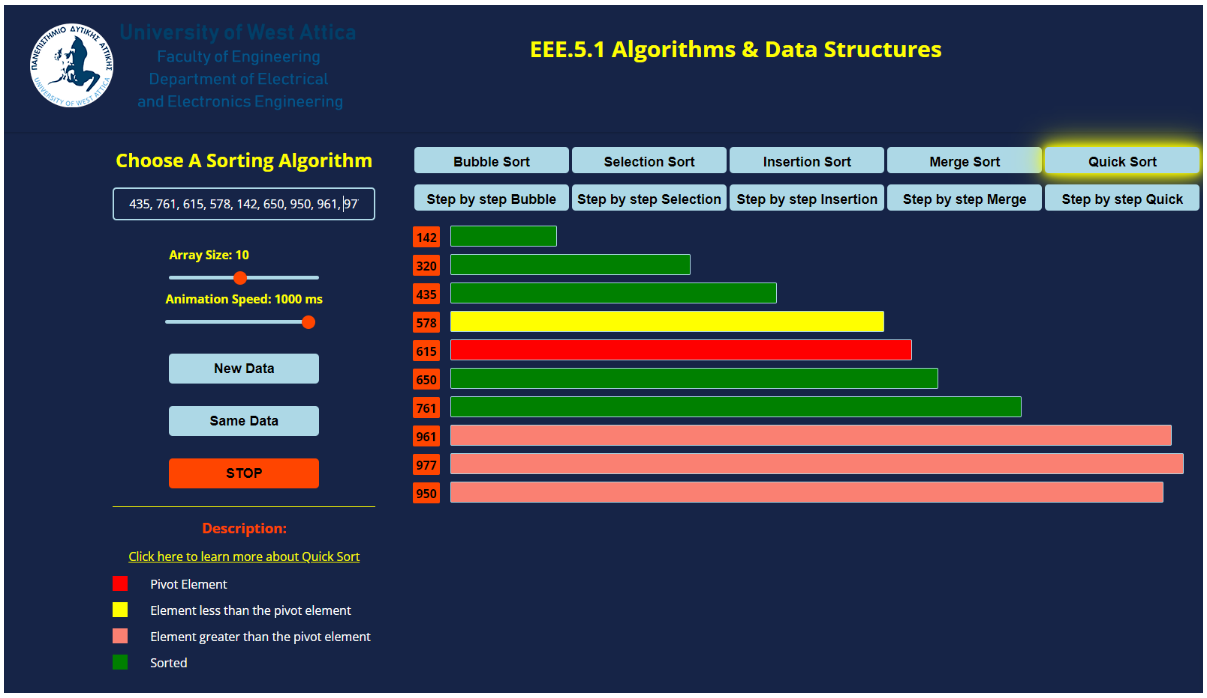
🟧 Element greater than the pivot element

🟩 Sorted

Figure 5.81: QUICK SORT: Pivot element 435 is in the correct position.

For the sublist [578, 615]:

Pivot element is again the last element on the array, 615. PivotIndex = 3.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

142
320
435
578
615
650
761
961
977
950

Description:

🟥 Pivot Element

🟨 Element less than the pivot element

🟥 Element greater than the pivot element

🟩 Sorted

Figure 5.82: QUICK SORT: Pivot element is 615.

- I = 3:

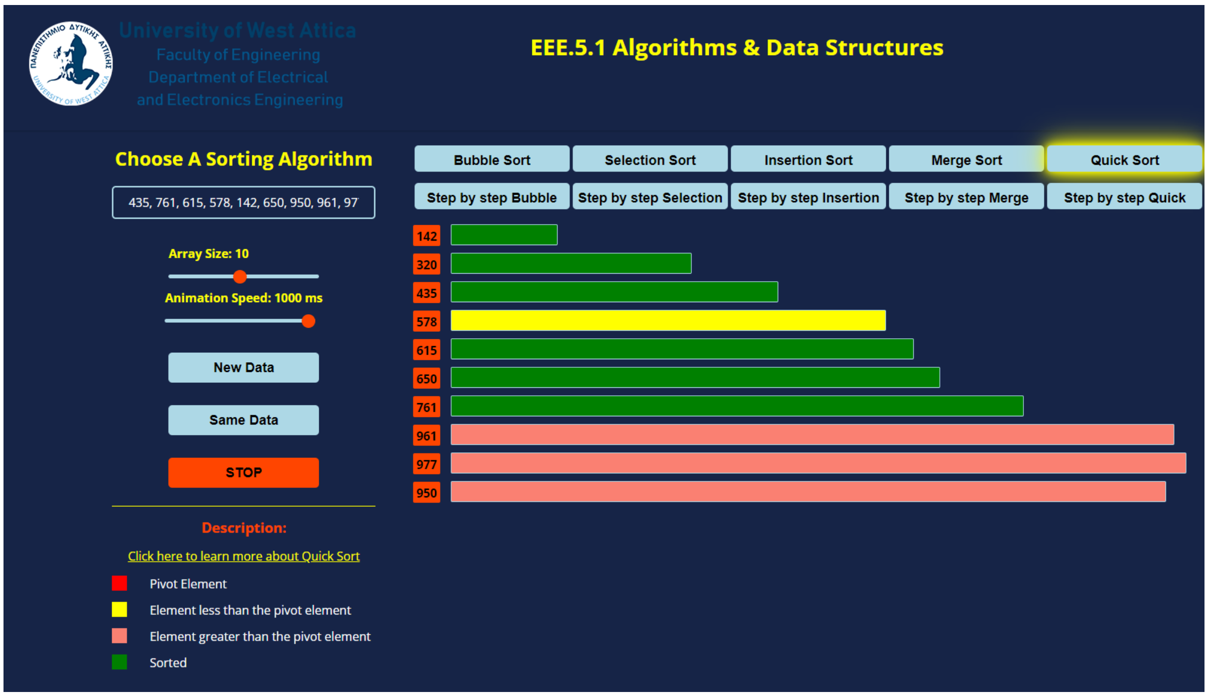
  Element at index 3 = 578 is less than pivot (615), so it is swapped with the element at PivotIndex = 578.

  PivotIndex is incremented.

  PivotIndex = 4

- I = 4:

The for loops terminates. At this point the pivot element (615) is swapped with the element at PivotIndex = 615.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

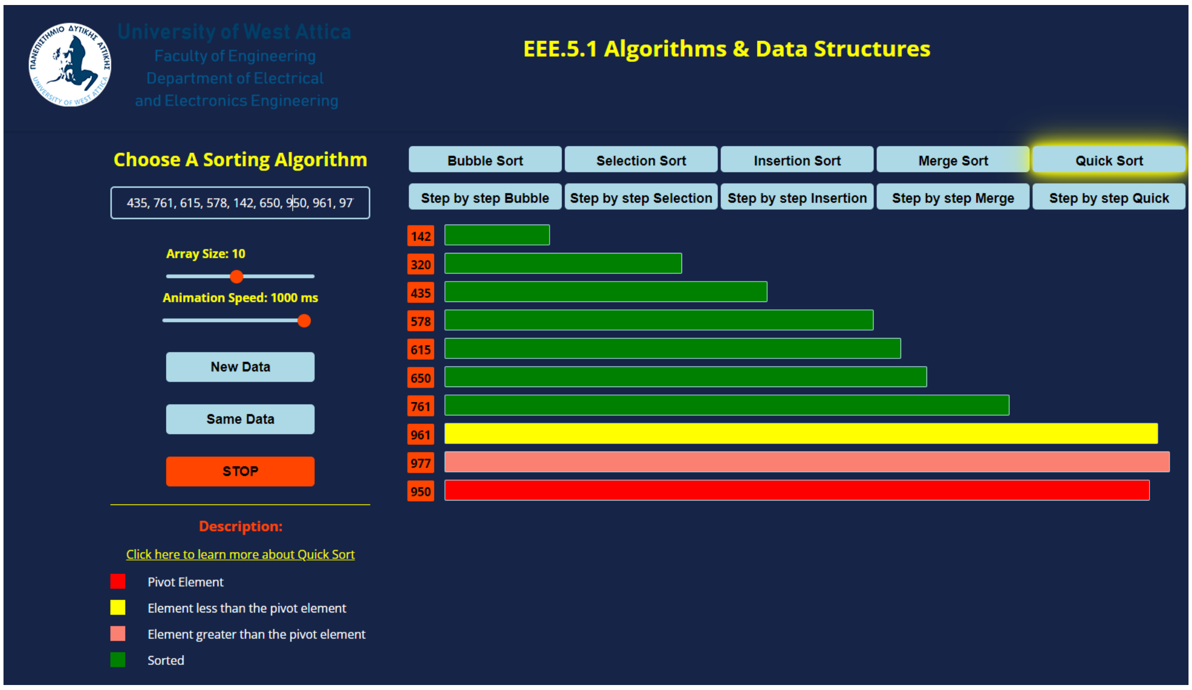
Description:

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.83: QUICK SORT: Pivot element 615 is in the correct position.

For the sublist [578]:

Since the sublist consists of only a single element, is already sorted.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

Description:

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.84: QUICK SORT: Element 578 is in the correct position.

For the sublist [961, 977, 950]:

Pivot element is again the last element on the array, 950. PivotIndex = 7.

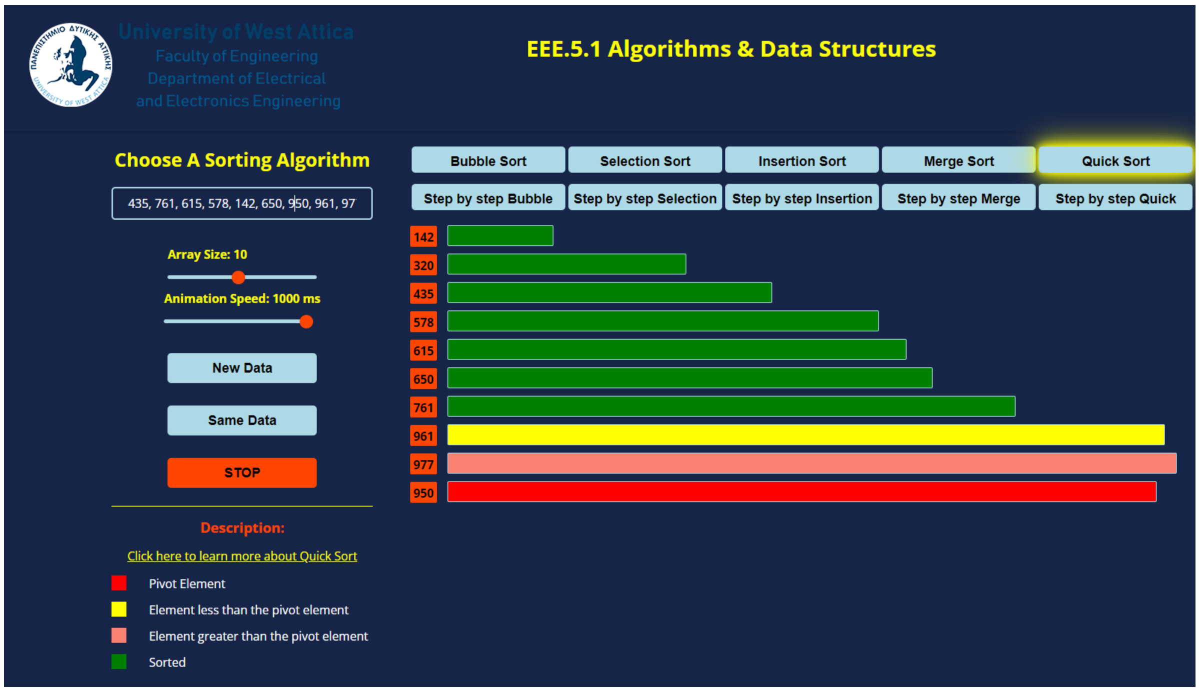


Figure 5.85: QUICK SORT: Pivot element is 950.

- I = 7:

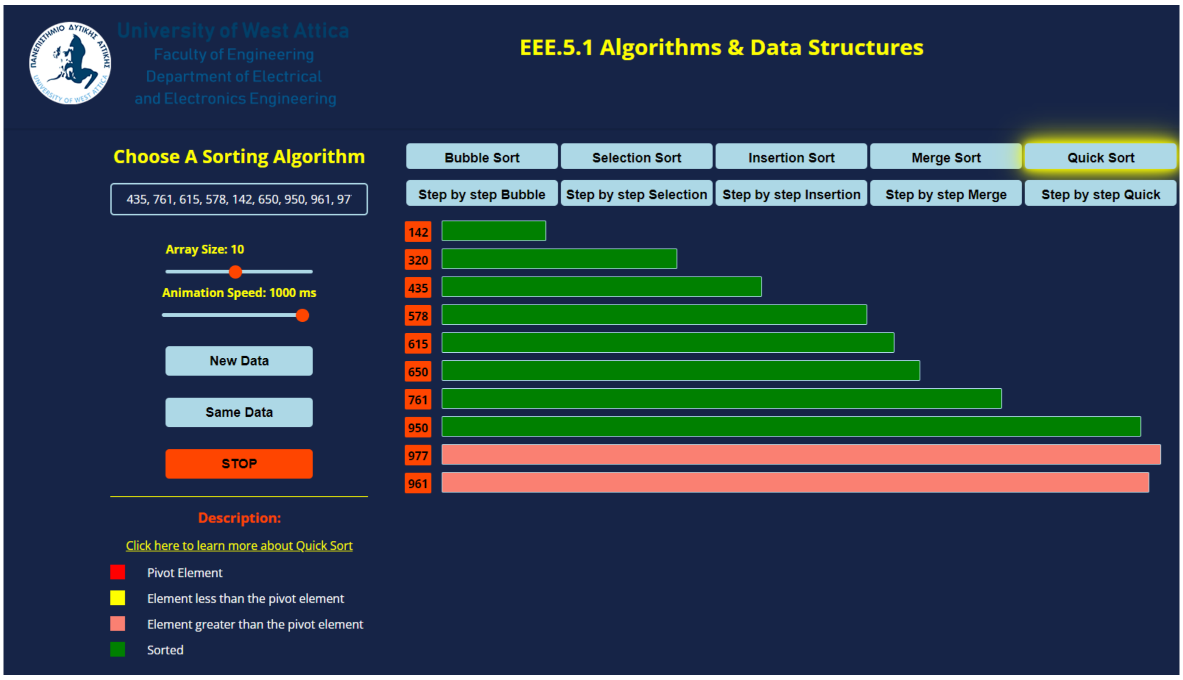
Element at index 7 = 961 is greater than pivot (950), so PivotIndex is not incremented.

- I = 8:

Element at index 8 = 977 is greater than pivot (950), so PivotIndex is not incremented.

- I = 9:

The for loops terminates. At this point the pivot element (950) is swapped with the element at PivotIndex = 961.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

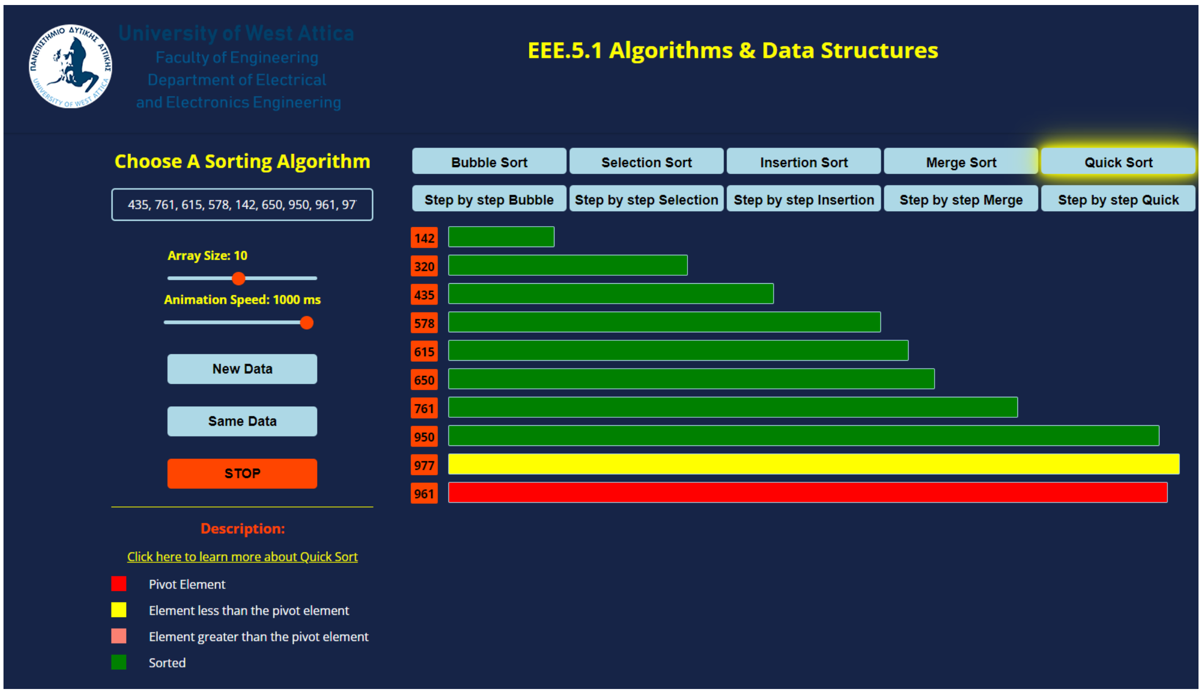
Description:

Click here to learn more about Quick Sort

🟥 Pivot Element
🟨 Element less than the pivot element
🟥 Element greater than the pivot element
🟩 Sorted

Figure 5.86: QUICK SORT: Element 950 is in the correct position.

<u>For the sublist [977, 961]:</u> Pivot element is again the last element on the array, 961. PivotIndex = 8.



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

Description:

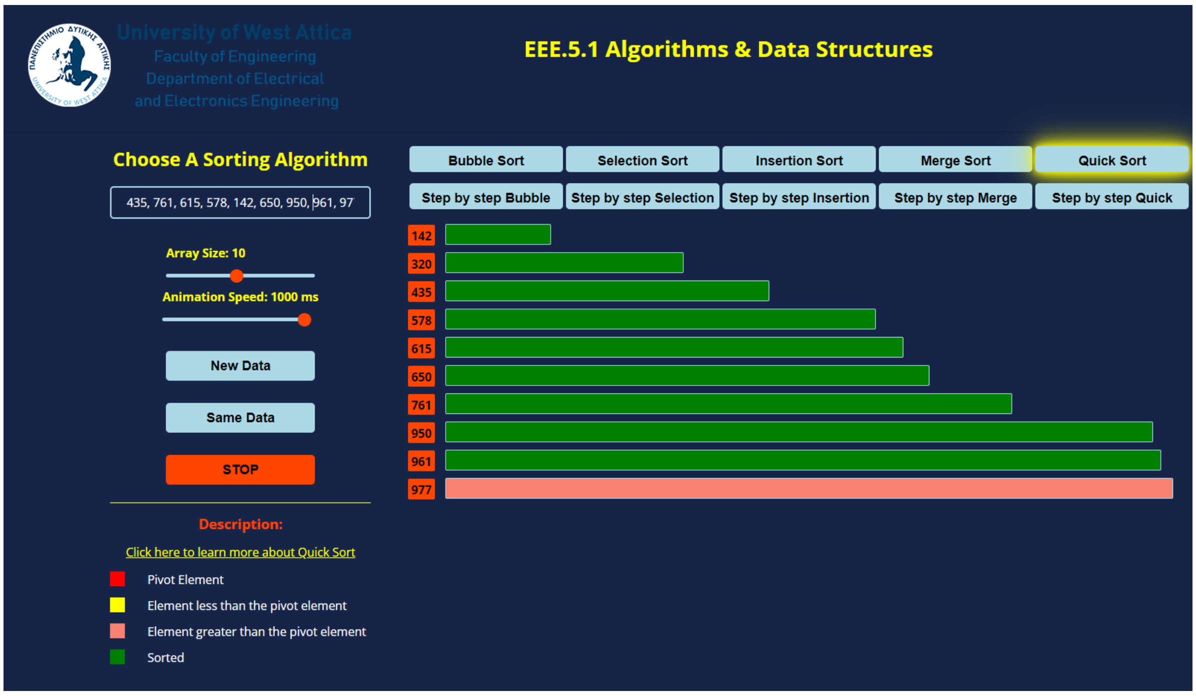
Click here to learn more about Quick Sort

🟥 Pivot Element
🟨 Element less than the pivot element
🟥 Element greater than the pivot element
🟩 Sorted

Figure 5.87: QUICK SORT: Pivot element is 961.

- I = 8: Element at index 8 = 977 is greater than pivot (961), so PivotIndex is not incremented.

- I = 9: The for loops terminates. At this point the pivot element (961) is swapped with the element at PivotIndex = 977.



435, 761, 615, 578, 142, 650, 950, 961, 97

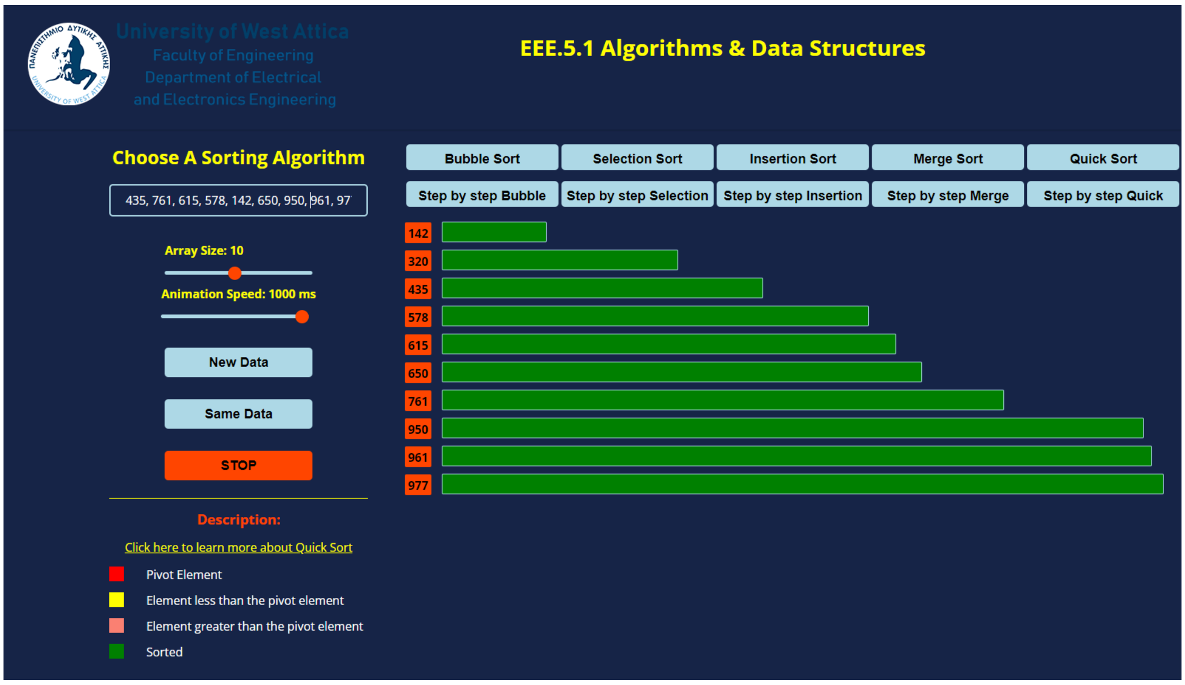
Array Size: 10

Animation Speed: 1000 ms

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.88: Pivot element, 961 is in the correct position.

Since the final sublist [977] is a single element, is already sorted, giving us the final sorted array:



435, 761, 615, 578, 142, 650, 950, 961, 97

Array Size: 10

Animation Speed: 1000 ms

🟥 Pivot Element
🟨 Element less than the pivot element
🟧 Element greater than the pivot element
🟩 Sorted

Figure 5.89: QUICK SORT: The final sorted array.

CHAPTER 6: Conclusions and future work

In order to enhance the learning process for students studying algorithms and data structures, this thesis has presented a web application for sorting algorithm visualization. The application offers a dynamic and useful aid for students to test these algorithms and their complexities through the use of visualization and interactivity. The application is a complete resource for students who want to understand in depth sorting algorithms and practice on the subject; for the same reason, the thesis also contains an in-depth analysis and description of each sorting algorithm that may be exploited for a future digital complete course. In general, the results of this thesis demonstrate the value of visualization as a teaching tool and indicate that using interactive and graphical assets might be useful in improving student understanding of difficult ideas. The current application could be improved and expanded upon in a number of ways in the future.

Expanding the scope of this application to include visualization of more data structures in addition to sorting algorithms is an obvious direction for future work. In addition to the several different data structures that could be used in the application, sorting algorithms are merely one subset of the larger area of algorithms and data structures, as we have already mentioned. For example, linear data structures such as stacks, queues, and linked lists can be incorporated in the application to visualize their functionalities using the same interactive method that the application already implements for the sorting algorithms. To provide students a more detailed understanding of the comparative merits of the various families of existing data structures, non-linear data structures such as trees and graphs could also be included – the later, at the cost of major modifications of the overall design.

Another potential improvement for the application is the incorporation of more detailed information on the operations executed by each algorithm at each step. The application in its current state offers merely a visual representation of the data. The number of comparisons and swaps performed at each step of the algorithm, is not counted or shown on screen. This could be included in a future version of the application to enhance understanding. A feature like that would provide students a more detailed understanding of how the algorithm operates and help them comprehend the notions of time and space complexity.

The incorporation of deep learning or machine learning methods is another option. The application could be improved, for instance, by utilizing machine learning algorithms to assess how well sorting algorithms perform on various input datasets and to make recommendations for the most effective algorithm to apply in a particular scenario. Students that are working on real-world projects and need to select the best algorithm for their requirements may find this to be a valuable addition.

In conclusion, this thesis represents progress in the area of computer science, and it is believed that the use of the web application would be valuable to both teachers and students. The program serves as a starting point for additional development and adjustments that can be made to improve students’ academic performance. It is anticipated that the application will improve and become a more potent educational tool for algorithms and data structures with further testing and development.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

[2] https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages

[3] Fouh, E., Akbar, M., & Shaffer, C. A. (2012). The role of visualization in computer science education. *Computers in the Schools*, *29*(1-2), 95-117.

[4] Lis, R. (2014). Role of visualization in engineering education. *Advances in Science and Technology. Research Journal*, *8*(24).

[5] Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., ... & Velázquez-Iturbide, J. A. (2002). Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education* (pp. 131-152).

[6] Abu-Naser, S. S. (2008). Developing visualization tool for teaching AI searching algorithms.

[7] Lin, S., Fortuna, J., Kulkarni, C., Stone, M., & Heer, J. (2013, June). Selecting semantically-resonant colors for data visualization. In *Computer Graphics Forum* (Vol. 32, No. 3pt4, pp. 401-410). Oxford, UK: Blackwell Publishing Ltd.

[8] Cetin, I., & Andrews-Larson, C. (2016). Learning sorting algorithms through visualization construction. *Computer Science Education*, *26*(1), 27-43.

[9] Unwin, A. (2020). Why is data visualization important? What is important in data visualization? *Harvard Data Science Review*, *2*(1), 1.

[10] Haque, M. (2001). Web based visualization techniques for structural design education. In *2001 Annual Conference* (pp. 6-1148).

[11] Zhang, G., Zhu, Z., Zhu, S., Liang, R., & Sun, G. (2022). Towards a better understanding of the role of visualization in online learning: A review. *Visual Informatics*.

[12] Danziger, P. (2010). Big o notation. Source internet: http://www. scs. ryerson. ca/~ mth110/Handouts/PD/bigO. pdf, Retrieve: April.

[13] Rutanen, K., Gómez-Herrero, G., Eriksson, S. L., & Egiazarian, K. O. (2014). A general definition of the big oh notation for algorithm analysis.

[14] Krone, J., Ogden, W. F., & Sitaraman, M. (2003). *Oo big o: A sensitive notation for software engineering*. Technical Report RSRG-03-06, Department of Computer Science, Clemson University, Clemson, SC 29634-0974.

[15] https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/

[16] Astrachan, O. (2003). Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, *35*(1), 1-5.

[17] https://mg729.github.io/algorithm/2020/03/01/Algorithm_%281%29_BubbleSort/

[18] https://mg729.github.io/algorithm/2020/03/08/Algorithm_(2)_SelectionSort/

[19] https://www.cyberithub.com/what-is-merge-sort-algorithm-explained-with-examples/

[20] https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python

[21] https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort