

编号:

Serial Number:

审定成绩:

Approved Result:

重庆邮电大学
毕业设计（论文）
Graduation Project (Thesis) of
Chongqing University of Posts and
Telecommunications

中文题目

Title in
Chinese

排序算法可视化工具的设计与实现

英文题目

Title in
English

Design and Implementation of Visualizer for Sorting

Algorithms

学院名称

Name of
School

College of Computer Science and Technology

学生姓名

Student
Name

Hridoy (贺心)

专业

Major

Computer Science and Technology

班级
Class

L0471901

学号
Student
Number

L201930013

指导教师
Instructor

Chen Peng (陈鹏)

答辩组
负责人
Leader of
Thesis
Defense

Qiao Lihong 乔丽红

2023 年 6 月

June 2023

重庆邮电大学教务处制

Made by the Division of Teaching Affairs of Chongqing
University of Posts and Telecommunications

学院本科毕业设计(论文)诚信承诺书

Letter of Commitment for Integrity of the Graduation Project (Thesis) of Undergraduate

本人郑重承诺：

This is hereby to submit the graduation thesis with the solemn commitments as follows:

我向学院呈交的论文 Design and Implementation of Visualizer for Sorting Algorithms，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明并致谢。本人完全意识到本声明的法律结果由本人承担。

The thesis “Design and Implementation of Visualizer for Sorting Algorithms” I have submitted to the school is the result of my independent research work under the guidance of my supervisor. Except for the contents quoted in the text, this thesis does not contain any works published or written by other individuals or groups. Individuals and collectives who have made important contributions to the research of this paper have been marked and thanked in the text. I am fully aware that I am obligated to undertake the legal consequence of the statement.

年级

Final Year

专业

Computer Science and Technology

班级

L0471901

承诺人签名



年月日

2023.05.25

学位论文版权使用授权书

Copyright Authorization for Dissertation

本人完全了解重庆邮电大学有权保留、使用学位论文纸质版和电子版的规定，即学校有权向国家有关部门或机构送交论文，允许论文被查阅和借阅等。本人授权重庆邮电大学可以公布本学位论文的全部或部分内容，可编入有关数据库或信息系统进行检索、分析或评价，可以采用影印、缩印、扫描或拷贝等复制手段保存、汇编本学位论文。

I fully understand the regulations that Chongqing University of Posts and Telecommunications has the right to retain and use the thesis and its electronic versions, that is, the university has the right to submit it to relevant national departments or institutions, and allow it to be consulted and borrowed. I authorize Chongqing University of Posts and Telecommunications to publish all or part of its contents, compile it into relevant databases or information systems for retrieval, analysis, or evaluation, and save and compile it by photocopying, reduction, scanning, or copying.

(注：保密的学位论文在解密后适用本授权书。)

(Note: This authorization applies to the declassified confidential dissertation.)

学生签名：贺心

Signature of Student:

日期：2023年05月01日

Date: 2023.05.01

指导老师签名：

Signature of Supervisor

日期：2023年05月25日

Date: 2023.05.25



Design and Implementation of Visualizer for Sorting Algorithms

A Thesis

presented to

**the College of Computer Science and Technology
Chongqing University of Posts and Telecommunications**

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Computer Science and Technology

by

Hridoy(贺心)

June 2023

摘要

排序算法在计算机科学中至关重要，可以在不同领域找到各种应用场景。现有不少工作已经开发了可视化技术以更好地理解这些排序算法的性能和行为。本论文开发了一个使用 Javascript 构建的排序可视化工具。其中，Javascript 是一种常用的 Web 开发编程语言，而可视化技术是交互式的并能提供实时反馈，使用户能够比较不同排序算法在不同输入条件下的性能。排序可视化工具作为排序算法的有效教学和学习工具，具有用户自定义功能的特点，易于使用。本论文的工作有助于开发针对排序算法的高效可视化工具，并强调了 Javascript 在构建响应迅速且引人入胜的 Web 应用程序方面的潜力。本论文中介绍的排序可视化工具对计算机科学及相关领域的专业人士和教育工作者具有重要价值。

关键词：排序算法，可视化，数据结构，JavaScript，用户界面。

Abstract

Sorting algorithms are crucial in computer science and find various applications across different domains. Visualization techniques have been developed to better understand the performance and behavior of these algorithms. This thesis introduces a sorting visualizer built using Javascript, a commonly used programming language for web development. The visualizer is interactive and provides real-time feedback, enabling users to compare the performance of different sorting algorithms under different input conditions. It serves as an effective teaching and learning tool for sorting algorithms, with customization options that make it easy to use. This work contributes to the development of efficient visualization tools for sorting algorithms and highlights the potential of Javascript for building responsive and engaging web applications. The sorting visualizer presented in this thesis holds significant value for professionals and educators in the field of computer science and related areas.

Keywords: Sorting algorithms, Visualization, Data structures, JavaScript, User interface.

Outline

Title: Design and Implementation of Visualizer for Sorting Algorithms

Analysis: This project aims to make a visualizer for sorting algorithms, which will make it possible for users to better understand and solve problems in computer science by providing them with an engaging and appealing tool that helps them understand common sorting algorithms and how well they work.

Thesis Statement: The visualizer for sorting algorithms project's thesis statement successfully demonstrates the significance of interactive and visually appealing tools in enhancing users' comprehension of common sorting algorithms and their effectiveness.

Contents

1 Introduction	V
2 About Algorithms.....	3
2.1 What is an algorithm?.....	3
2.2 Algorithms Description	4
2.3 Types of Algorithms	5
2.4 Sorting Algorithms	6
3 Sorting Algorithms Included	8
3.1 Insertion Sort	8
3.2 Selection Sort.....	8
3.3 Bubble Sort.....	9
3.4 Shell Sort	10
3.5 Quick Sort.....	10
3.6 Merge Sort	11
3.7 Heap Sort	12
3.8 Counting Sort.....	12
3.9 Radix Sort.....	13
3.10 Bucket Sort	13
4 Software Development Process.....	15
4.1 Requirements Specification.....	15
4.2 Software Design	15

4.3 Development and Implementation	18
4.3.1 Index file.....	18
4.3.2 Sorting file.....	20
4.3.3 CSS file.....	24
4.3.4 Algorithm files.....	26
4.4 Testing	41
4.5 Deployment and Maintenance	46
5 User interface	48
5.1 System Requirements for User	48
5.2 User Guide.....	48
Conclusion	52
Bibliography.....	53
Acknowledgment	54

List of Figures

2.1 Flowchart example: factorial	4
3.1 Example: Insertion sort execute	8
3.2 Selection sort principle	9
3.3 Bubble Sort: one pass through the array	9
3.4 Example: Shell sort.....	10
3.5 Quick Sort principle	11
3.6 Example: Merge Sort.....	11
3.7 Max-Heap	12
3.8 Counting sort principle	13
3.9 Example: Radix sort execution.....	13
3.10 Bucket sort principle.....	14
4.1 User function sequence diagram	18
4.2 Test: Sorting accuracy and clarity	42
4.3 Test: Visual representation of Sorting.....	43
4.4 Test: Successfully sorted array	43
4.5 Test: Different input size of array	44
4.6 Test: Stop sorting button.....	44
4.7 Test: Sorting function	45
5.1 Main window	49
5.2 Visualization panel	50
5.3 Algorithm selector	51
5.4 Successful sorting of bubble sort.....	51

TABLES

4.1 User functional testing.....	45
----------------------------------	----

1 Introduction

Sorting algorithms play a vital role in computer science, enabling efficient organization and manipulation of data. However, understanding these algorithms and their intricate workings can be challenging, particularly for learners. Traditional teaching methods often rely on theoretical explanations and code examples, which may not effectively convey the dynamic nature and complexities of sorting algorithms. To bridge this gap and enhance the learning experience, this thesis presents the Visualizer of Sorting Algorithms.

The primary objective of this research is to develop an intuitive and interactive visualizer tool that enables learners to comprehend sorting algorithms more effectively. By providing a dynamic representation of the sorting process, the visualizer facilitates a deeper understanding of the underlying concepts, mechanisms, and time complexities associated with sorting algorithms. This visualizer creates an immersive learning environment that engages learners and fosters their algorithmic thinking skills.

This thesis leverages modern web technologies, including HTML, CSS, and JavaScript, to design and implement the visualizer. The tool focuses on ten prominent sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Shell Sort, Bucket Sort, Counting Sort, and Radix Sort. Each algorithm is meticulously implemented to demonstrate its step-by-step execution, enabling learners to observe how data elements are rearranged during the sorting process.

A significant aspect of this research is its emphasis on usability and user experience. The visualizer is designed with a user-centered approach, ensuring that learners find it intuitive and easy to navigate. User evaluations and feedback are collected to assess the effectiveness of the visualizer in facilitating learning and understanding. This iterative feedback-driven approach ensures that the visualizer evolves into an optimal educational tool that caters to the needs of diverse learners. The findings from these evaluations contribute to refining and enhancing the visualizer's functionality, interactivity, and overall educational value.

The impact of this project extends to various stakeholders, including students,

educators, and professionals in the field of computer science. By providing a visual and interactive platform for exploring sorting algorithms, the visualizer equips learners with a solid foundation in algorithmic thinking. It enhances their ability to analyze, select, and implement appropriate sorting algorithms based on the requirements of real-world scenarios. Educators can incorporate the visualizer into their teaching methodologies to foster a deeper understanding of sorting algorithms among students, making the learning process engaging and effective.

In conclusion, the "Design and Implementation of Visualizer for Sorting Algorithms" thesis offers a novel approach to enhance the learning experience of sorting algorithms. By combining visualization techniques, web technologies, and a comprehensive study of sorting algorithms, this research provides a valuable tool for learners to master the intricacies of sorting and strengthen their algorithmic thinking skills. The visualizer serves as a bridge between theory and practice, empowering learners to become proficient in sorting algorithms and fostering a deeper appreciation for their efficiency and effectiveness in real-world applications. The outcomes of this research have the potential to revolutionize the teaching and learning of sorting algorithms, contributing to the Advancement of computer science education.

2 About Algorithms

2.1 What is an algorithm?

Before delving into the core content that discusses algorithms and software, it is important to establish a solid understanding of the fundamentals. To commence, we will start by defining what an algorithm is.

To understand what an algorithm is and provide a simple definition, we can describe it as a series of steps or instructions that are arranged in a specific order to solve a given problem. However, it's important to note that this definition might lack precision since it does not explicitly define the terms "problem" and "instruction."

A problem refers to a task that requires a resolution. In our daily lives, we encounter various problems, such as determining the most efficient route to work or home. However, not all of these problems align with our algorithm definition. A problem can be adequately defined with certain constraints: it must be defined by its inputs, and all inputs must have corresponding outputs. A step or instruction represents a clear action for the individual carrying it out, such as a personal computer (PC) in our case. Solving a problem involves finding a solution for each input.

Here is a more precise definition by Thomas Cormen: “Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.” (CORMEN, 2001, p. 40)

5 features of an algorithm according to D. Knuth (KNUTH, 2004, p.113):

- finiteness – an algorithm should end in a finite number of steps.
- definiteness – each step of an algorithm should have a precise definition. And it means that for the same inputs, we will obtain the same results.
- input – an algorithm may have inputs; they are taken from some set of objects.

- output – an algorithm may have outputs that should be in some relation with inputs.
- effectiveness – we may expect an algorithm to be effective. It means “its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper.” (KNUTH, 2004, p.114)

2.2 Algorithm Description

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

- Natural language: Describing an algorithm in natural language is easily understandable for everyone. However, it may lack precision and be longer compared to other methods mentioned.
- Programming language: This type of description is unambiguous and can be directly used to create a computer program. It includes specific implementation details and syntax of a particular programming language.
- Pseudocode: Pseudocode resembles a programming language but is more general and doesn't include intricate details. It can be easily translated into various programming languages and is comprehensible to programmers regardless of the specific language.
- Visual representation: Algorithms can also be described using graphical methods like flowcharts (see Figure 2.1) among other techniques.

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

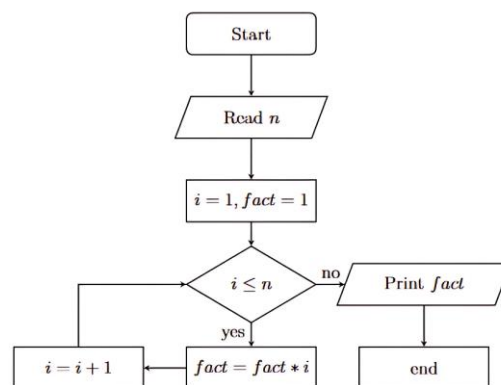


Figure 2.1: Flowchart example: factorial

Algorithm 1 Factorial algorithm

```
1: procedure Factorial(n)
2:   fact  $\leftarrow$  1
3:   for i  $\leftarrow$  1 to n do
4:     fact  $\leftarrow$  fact * i
5:   return fact
```

2.3 Types of Algorithms

Currently, a wide range of algorithms exists, classified into various groups based on different criteria. It is common for an algorithm to belong to multiple groups simultaneously. Here are several examples (CORMEN, 2001, p. 51):

- Recursive algorithms: These algorithms repeatedly call themselves until a specified condition is met.
- Probabilistic algorithms: These algorithms incorporate random decisions as part of their execution.
- Parallel algorithms: These algorithms distribute a task among multiple threads or processors to be executed simultaneously.
- Sequential algorithms: In contrast to parallel algorithms, sequential algorithms execute each step of a task sequentially.
- Divide and conquer: These algorithms divide a problem into smaller subproblems until they become indivisible, and then combine the solutions in a specific way.
- Greedy algorithms: These algorithms make optimal choices at each step based on the current situation, without reconsidering previous choices.
- Dynamic programming: These algorithms solve a problem by breaking it down into simpler subproblems and utilizing solutions from previously solved subproblems.
- Heuristic algorithms: These algorithms aim to find solutions among all possible options, but they do not guarantee that the solution found will be the best one.

By classifying algorithms into these groups, it becomes easier to understand their characteristics and select the most appropriate approach for a given problem.

2.4 Sorting Algorithms

As previously mentioned, sorting plays a crucial role in solving a diverse array of problems. It serves various purposes, such as facilitating efficient searching and serving as a component of complex tasks. However, let's delve deeper into the topic of sorting itself.

In straightforward terms, sorting is the act of rearranging items that can be compared, placing them in either ascending or descending order. When referring to sorting in the text, we specifically consider ascending order unless otherwise indicated.

Ascending order refers to arranging items in a sequence from the smallest to the largest item. Conversely, descending order involves arranging them from the largest to the smallest item.

Sorting algorithms are divided into two main types (SEGEWIK, 2007, p. 71):

1. Internal sorting algorithms: These algorithms operate when all the data to be sorted is stored within the internal memory. They are typically used when the amount of data is known and can fit comfortably in the internal memory.
2. External sorting algorithms: In contrast, these algorithms handle data that is stored outside the internal memory, such as on a hard disk. They typically involve a combination of sorting within the internal memory, merging sorted parts, and storing them back into the external memory.

In the text, we are talking only about the algorithms of internal sorting. Here are five main techniques that are usually used with the algorithms which use these techniques are included in the software (KNUTH, 2004, p. 85).

1. Sorting by Insertion – single items from the sequence are put into the right place of the sorted part. Here belongs Insertion Sort. Shell Sort also uses this technique.
2. Sorting by Exchanging – swap elements of each pair that are out of order till no more such pairs exist. Here we have Bubble Sort and Quick Sort.
3. Sorting by Selection – a method that uses repeated selection. Selection Sort, Heap Sort, and Quick Sort use this technique.
4. Sorting by Merging – merging smaller parts in the right order. And

Merge Sort uses it.

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

3 Sorting Algorithms Included

This section provides an overview of the sorting algorithms implemented in this ‘visualizer for sorting algorithm’ project.

3.1 Insertion Sort

Insertion Sort is a simple and intuitive sorting algorithm that works by dividing the input array into two parts: a sorted portion and an unsorted portion. It iterates through the unsorted portion and compares each element with the elements in the sorted portion to find its correct position. The algorithm shifts elements in the sorted portion to the right to make space for the current element and inserts it in the correct position (Figure 3.1). This process is repeated until the entire array is sorted. Insertion Sort has a time complexity of $O(n^2)$ in the worst case but performs efficiently for small input sizes or partially sorted arrays.

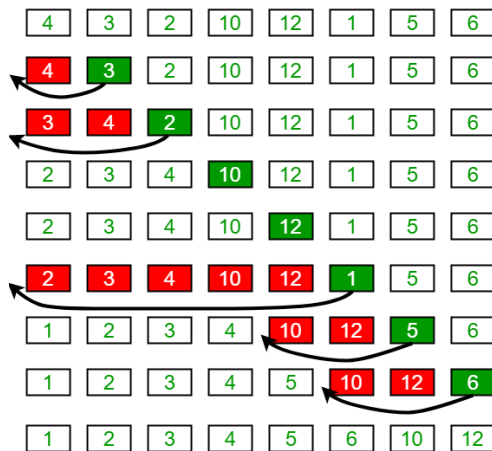


Figure 3.1: Example: Insertion sort

3.2 Selection Sort

Selection sort is a simple sorting algorithm that repeatedly finds the minimum element from an unsorted portion of the list and swaps it with the element in the next position of the sorted portion. It divides the list into two parts: sorted and unsorted. In each iteration, it selects the smallest element from the unsorted part and moves it to the sorted part in Figure 3.2; This process is repeated until the

entire list is sorted. Although simple, selection sort has a time complexity of $O(n^2)$, making it inefficient for large lists.

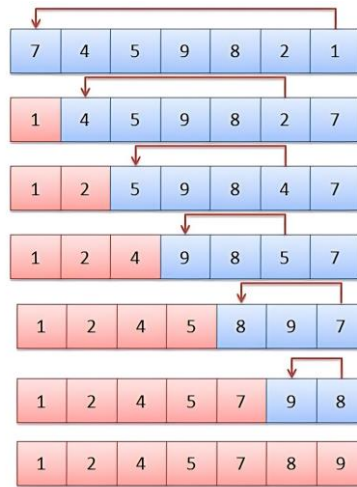


Figure 3.2: Selection Sort principle

3.3 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. It works by repeatedly passing through the list, swapping adjacent elements until the largest element "bubbles" to the end (Figure 3.3). This process is repeated until the entire list is sorted. Bubble sort has a time complexity of $O(n^2)$, making it inefficient for large lists, but it is easy to understand and implement.

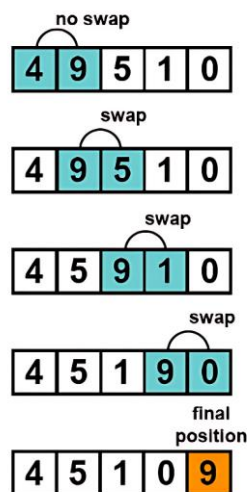


Figure 3.3: Bubble sort: one pass through the array

3.4 Shell Sort

Shell sort is an efficient sorting algorithm that is an extension of insertion sort. It works by dividing the list into smaller subarrays and sorting them independently. The subarrays are created by choosing a decreasing sequence of gap values. The algorithm performs multiple passes, each time reducing the gap between elements (Figure 3.4). Eventually, it performs a final insertion sort on a nearly sorted list. Shell sort has a time complexity that varies depending on the chosen gap sequence, but it generally performs better than other simple sorting algorithms like bubble sort and insertion sort.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Figure 3.4: Example: Shell sort

3.5 Quick Sort

Quick sort is a widely used sorting algorithm that follows the divide-and-conquer approach. It selects a pivot element and partitions the list into two subarrays, one with elements smaller than the pivot and another with elements larger (Figure 3.5). This process is recursively applied to the subarrays until they are sorted. Quick sort has an average time complexity of $O(n \log n)$, making it one of the fastest sorting algorithms. However, its worst-case time complexity is $O(n^2)$ when the pivot is consistently chosen poorly, but this can be mitigated with optimizations like choosing a random pivot or using the median-of-three method.

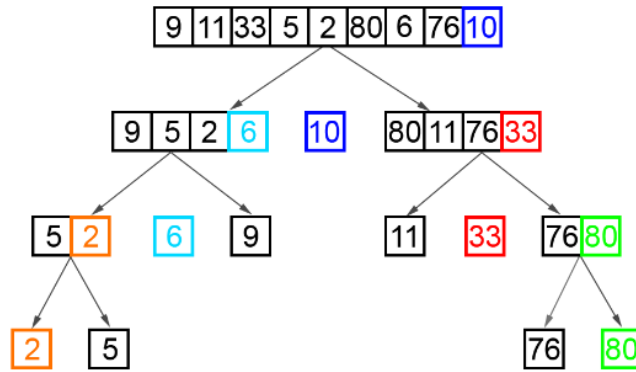


Figure 3.5: Quick Sort principle

3.6 Merge Sort

Merge sort is a popular sorting algorithm that follows the divide-and-conquer strategy. It divides the list into two halves recursively until each sublist contains only one element. It then merges the sublists by comparing and combining them in a sorted manner until a single sorted list is obtained (Figure 3.6). Merge sort has a stable time complexity of $O(n \log n)$, making it efficient for large datasets. However, it requires additional space for the merging process, which can be a disadvantage in memory-constrained scenarios.

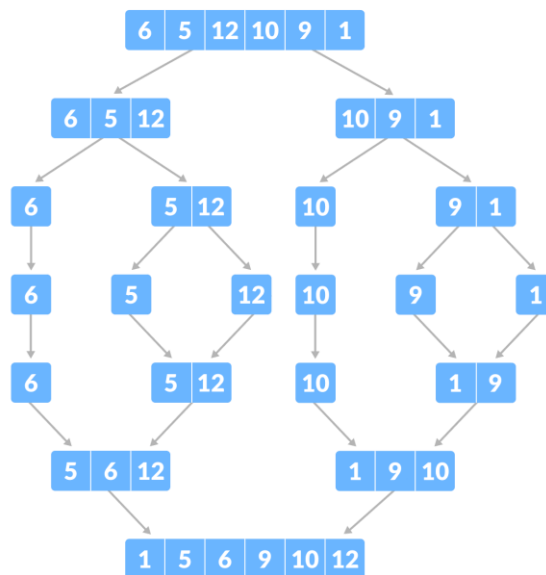


Figure 3.6: Example: Merge sort

3.7 Heap Sort

Heap sort is a comparison-based sorting algorithm that leverages a binary heap data structure. It starts by building a max heap from the input list, where the maximum element is at the root (Figure 3.7). It then repeatedly swaps the root element with the last element and reduces the heap size, ensuring the largest element is at the end. This process is repeated until the entire list is sorted. Heap sort has a time complexity of $O(n \log n)$ and is an in-place sorting algorithm. However, it is not stable, meaning that the relative order of equal elements may change during the sorting process.

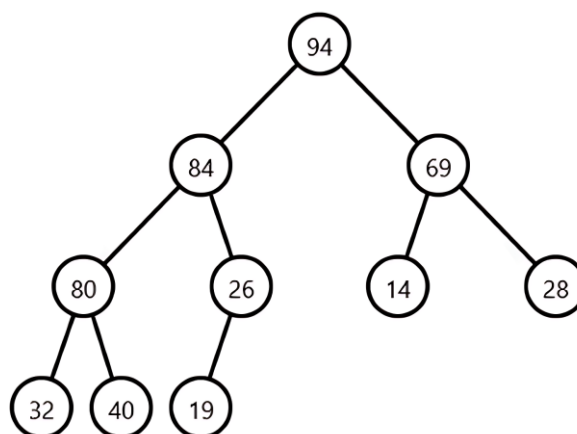


Figure 3.7: Max-Heap

3.8 Counting Sort

Counting sort is a linear time sorting algorithm that works well for lists with a limited range of integers. It creates a count array to store the frequency of each element in the input list. It then modifies the count array to represent the cumulative sum of counts. Using the count array, it places each element from the input list into its correct position in the output array (Figure 3.8). Counting sort has a time complexity of $O(n+k)$, where n is the number of elements and k is the range of values. However, it requires additional space proportional to the range of values. Iterate over the input list and count the number of occurrences of each element, by incrementing the corresponding element in the new list.

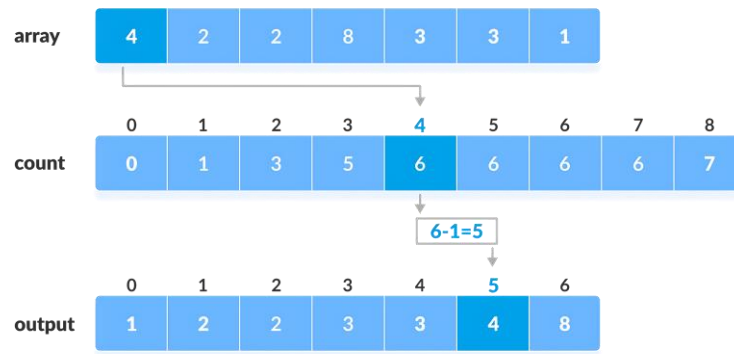


Figure 3.8: Counting sort principle

3.9 Radix Sort

Radix sort is a linear time sorting algorithm that sorts elements by their digits or bits. It works by grouping elements into buckets based on the value of a specific digit or bit position. It iterates through each digit or bit position, sorting the elements into the appropriate bucket. After iterating through all positions, the elements are concatenated to obtain the sorted list in Figure 3.9. Radix sort has a time complexity of $O(d * (n + k))$, where d is the number of digits or bits, n is the number of elements, and k is the range of values. It is often used for sorting integers or strings with fixed-length representations.

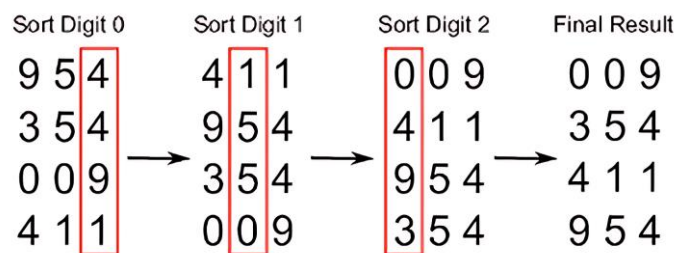


Figure 3.9: Example: Radix sort execution

3.10 Bucket Sort

Bucket sort is a sorting algorithm that divides the input list into a fixed number of equally sized buckets. Elements from the input list are distributed into buckets based on their value ranges. Each bucket is then sorted individually, either using another sorting algorithm or recursively applying bucket sort. Finally, the

elements are concatenated to obtain the sorted list (Figure 3.10). Bucket sort has an average time complexity of $O(n + k)$, where n is the number of elements and k is the number of buckets. It is efficient when the input elements are uniformly distributed across the range.



Figure 3.10: Bucket sort principle

4 Software Development Process

This section provides a comprehensive overview of the software development process for the visualizer for the sorting algorithm project, along with several relevant topics from a programmer's perspective.

4.1 Requirements Specification

The ‘visualizer for sorting algorithm’ web application is built using several front-end web technologies, including HTML5, CSS3, Bootstrap, and Vanilla JavaScript. Here's a more detailed breakdown of the technologies used:

1. HTML5: The main markup language used to structure the content of the web page. It defines the skeleton of the Sorting Visualizer user interface.
2. CSS3: Used for styling and layout of the Sorting Visualizer user interface, including custom styles and Bootstrap classes.
3. Bootstrap 4: A popular front-end CSS framework that provides ready-made UI components like buttons, forms, modals, etc. It's used here to make the user interface responsive and visually appealing.
4. JavaScript: The main programming language used for adding interactivity and functionality to the web page. It's used in this project to implement the sorting algorithms and handle user events like button clicks and input changes.

Development Environment:

Visual Studio Code was used as the code editor, and a web browser was used for development and testing. Utilizing this development environment has the advantages of being simple to use, providing immediate feedback, and being simple to troubleshoot. The code has been hosted on GitHub and version controlled with Git. The project development process has been made simpler thanks to this development environment, which also makes it simple to share and work with others.

4.2 Software Design

Architecture design:

1. User interface (UI) layer:
 - Responsible for rendering the user interface elements and interacting with the user.
 - Displays the array of bars representing the data to be sorted.
 - Provides buttons for sorting algorithms, array generation, and controlling the visualization speed.
 - Allows the user to input the size of the array and adjust the sorting speed.
2. Sorting algorithms:
 - Contains the implementations of various sorting algorithms such as bubble sort, insertion sort, selection sort, merge sort, heap sort, bucket sort, and more.
 - Each algorithm operates on the array of bars and modifies their positions based on the sorting logic.
 - Algorithms may have different time complexities and sorting strategies.
3. Animation and visualization:
 - Handles the animation and visualization aspects of the sorting process.
 - Utilizes the UI layer to update the display and provide visual feedback during the sorting process.
 - Controls the timing and sequencing of the sorting steps to create a visual representation of the sorting algorithms.
4. Event handlers and control:
 - Manages the event handlers for user interactions, such as button clicks and input changes.
 - Coordinates the flow of control between the UI layer, sorting algorithms, and animation/visualization module.
 - Enables/disables buttons and controls based on the current state of the sorting process.
 - Controls the start, stop and reset actions of the sorting visualizer.
5. Data management:

- Handles the generation of the initial array of bars based on user input for size.
- Manages the state and data structure representing the array of bars to be sorted.
- Provides utility functions for accessing and modifying the array data.

This architecture separates the different responsibilities of the visualizer into distinct modules, allowing for modularity and easier maintenance. The user interface layer interacts with the user, the sorting algorithms handle the sorting logic, the animation and visualization module provide the visual representation, the event handlers and control module manage user interactions, and the data management module handles the array data.

User interface design:

The User Interface (UI) of the system will be developed using HTML and CSS. It will consist of a container to hold the array that needs to be sorted, two sliders to control the generation of a random array and sorting speed, and a button for each sorting algorithm to initiate the sorting process. The sorted array will be displayed in a visually distinct color from the unsorted array.

The sorting algorithms in the system will be implemented using JavaScript. The supported sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Shell Sort, Bucket Sort, Counting Sort, and Radix Sort.

The system's functionality allows the user to generate a random array by clicking the "New Array" button. When a sorting algorithm is selected, the chosen algorithm will sort the array. The array's changes during the sorting process will be displayed in real-time. Once the array is completely sorted, the sorted array will be visually differentiated from the unsorted array.

To optimize performance, the system allows users to change the size of the array according to their preferences. This enables testing and comparison of the sorting algorithms' efficiency and effectiveness.

Database design:

No persistent data storage is needed for this project. As a result, neither the input data nor the output of the sorting algorithms has been stored in a database.

The decision to forego the use of a database has simplified and lightened the project's overall architecture.

To gain a better understanding of the design and functionality of this project, refer to Figure 4.1, which illustrates the user function diagram. This diagram visually outlines the various user interactions and operations involved in the project. By studying this diagram, you can grasp a clearer picture of how the project is structured and how users can interact with it.

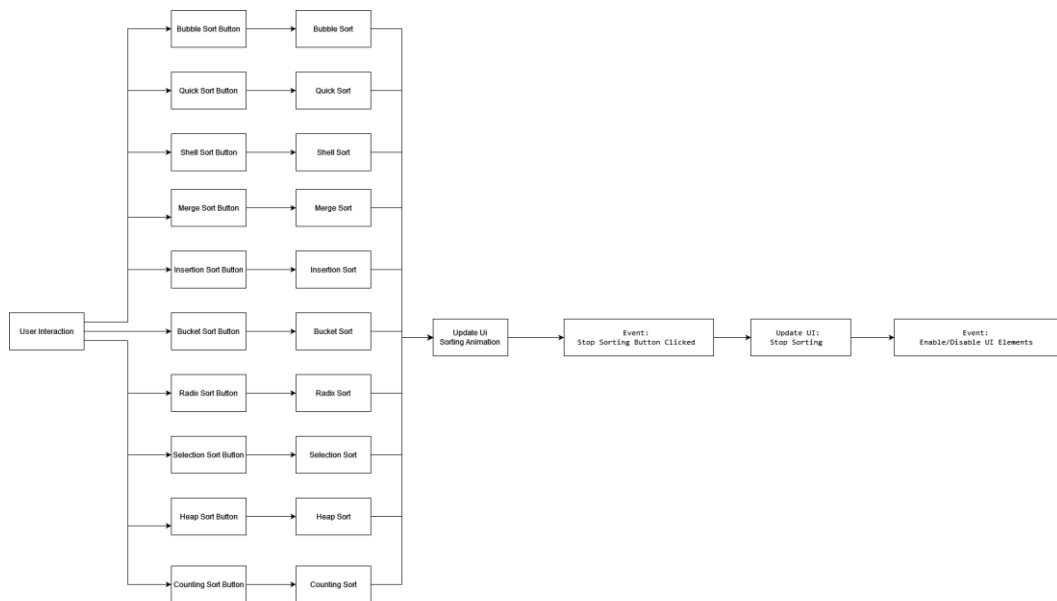


Figure 4.1: User function sequence diagram

4.3 Development and Implementation

In this section, we will delve into the details of the coding files, examining their development techniques, system modules, the function of each module, and the code implementation.

4.3.1 Index file

The code provided in “index.html” is an HTML document that sets up the structure of a sorting visualizer web application. Here's an overview of the development techniques and the modules included in the system:

1. Development techniques:

- HTML
- CSS

- Bootstrap
- JavaScript

2. Modules in the system:

- Sorting visualizer:
 - `<div id="sorting" class="flex-container">`
- Input panel:
 - `<div class="col range-input" id="input-panel">`
 - `<input id="size_input" type="range">`
 - `<input id="speed_input" type="range">`
- Control panel:
 - `<div class="w-50 mx-auto d-flex flex-wrap my-4 justify-content-center" id="control-panel">`
 - Each sorting algorithm is represented by a `<button>` element with a unique class.
- JavaScript files:
 - `sorting.js`: Contains the main JavaScript code for controlling the sorting visualizer.
 - `algorithms/bubbleSort.js`, `mergeSort.js`, `selectionSort.js`, `quickSort.js`, `insertionSort.js`, `heapSort.js`, `shellSort.js`, `bucketSort.js`, `countingSort.js`, `radixSort.js`: Contains the implementation of the sorting algorithms.

Overall, this system provides a user interface for visualizing various sorting algorithms by creating and animating bars to represent array elements. The user can adjust the size and speed of the sorting, as well as choose different sorting algorithms to observe their behavior. To implement the `index.html` file code, follow these steps:

Step 1: Create a new HTML file.

- Open a text editor and create a new file.

Step 2: Set up the HTML structure.

- Add the `<!DOCTYPE html>` declaration at the beginning of the file.
- Wrap the entire content within the `<html>` tags.
- Add the `<head>` section within the `<html>` tags.
- Inside the `<head>` section, include the necessary meta tags for character

encoding, compatibility, title, description, and viewport.

- Link the Bootstrap CSS stylesheet by adding the `<link>` tag with the appropriate href attribute.
- Link the custom CSS file ("style.css") by adding another `<link>` tag with the href attribute pointing to your CSS file.

Step 3: Define the HTML body.

- Add the `<body>` tags after the closing `</head>` tag.
- Inside the `<body>`, create the necessary HTML elements and structure according to the provided code.
- Pay attention to the classes, IDs, and attributes mentioned in the code.
- Make sure to link the necessary JavaScript files by adding the `<script>` tags with the appropriate src attributes.
- If you have separate JavaScript files for sorting algorithms, make sure to include them as well.

Step 4: Save the file.

- Save the HTML file with an appropriate name and the ".html" extension.

Step 5: Create the required CSS and JavaScript files.

- Create a CSS file named "style.css" in the same directory as the HTML file.
- Create JavaScript files for sorting algorithms such as "bubbleSort.js", "mergeSort.js", and so on.
- Make sure to place these JavaScript files in the appropriate directories (e.g., "algorithms/").

Step 6: Link the JavaScript files.

- Make sure to update the src attributes of the `<script>` tags in the HTML file to point to the correct file paths.
- If you need to write the JavaScript code for sorting algorithms, open the respective algorithm files and implement the sorting logic inside each file.

Step 7: Open the HTML file in a web browser.

- Double-click the HTML file or open it with your preferred web browser.
- The sorting visualizer web application should now be displayed, allowing you to interact with the sorting algorithms.

4.3.2 Sorting file

The code in "sorting.js" is responsible for implementing different sorting algorithms and visualizing the sorting process. Let's go through the different development techniques and modules used in the system, as well as the functions provided by each module.

1. Development techniques:

- DOM manipulation: The code utilizes JavaScript to manipulate the Document Object Model (DOM) to dynamically update the user interface.

2. Modules & function:

- Swap function (swap)
- Sorting button controls:
 - disableSortingBtn, enableSortingBtn, disableNewArrayBtn, enableNewArrayBtn
- Slider controls:
 - disableSizeSlider, enableSizeSlider, disableSpeedSlider, enableSpeedSlider,
- Delay function (delayTime)
- Array creation and visualization:
 - arraySize, createNewArray, deleteChild
- Event listeners:
 - arraySize, delayElement, newArrayButton, stopSortingButton

These modules work together to provide an interactive visualization of various sorting algorithms on the web application.

To implement the code step by step, follow these instructions:

- Implement the swap function, which takes two DOM elements and swaps their height values.
- Implement the button control functions (disableSortingBtn, enableSortingBtn, disableNewArrayBtn, enableNewArrayBtn, disableSizeSlider, enableSizeSlider, disableSpeedSlider, enableSpeedSlider, enableStopSortingBtn, disableStopSortingBtn), which disable or enable the corresponding buttons, sliders, or elements in the DOM.
- Implement the delayTime function, which returns a promise and introduces a delay in the execution of subsequent code. This is

used for animation during the sorting process.

- Define the variables `arraySize` and `delayElement`, and assign them the respective DOM elements for the size slider and speed slider.
- Add event listeners to `arraySize` and `delayElement` to listen for changes and update the corresponding values accordingly.
- Create the `barArray` array to store randomly generated numbers.
- Implement the `createNewArray` function, which takes the number of bars as an input and creates a new array of random numbers. It also updates the UI by dynamically creating bar elements in the container element with the id "sorting".
- Implement the `deleteChild` function, which removes all previous bars from the DOM.
- Call the `createNewArray` function initially to display the bars when the site is visited.
- Add an event listener to the "New Array" button (`newArrayButton`) to handle the click event. Inside the event listener, enable necessary controls, set relevant values, and call the `createNewArray` function with the current size value.
- Add an event listener to the "Stop Sorting" button (`stopSortingButton`) to handle the click event. Inside the event listener, disable sorting controls and set a flag to indicate that the sorting process has been stopped.

By following these steps, we can successfully implement the code in `sorting.js` and visualize the sorting algorithms on our web application.

Here is the pseudo-code for the `sorting.js` file:

```
function swap(el1, el2):
    temp = el1.style.height
    el1.style.height = el2.style.height
    el2.style.height = temp

function disableSortingBtn():
    // Disable all sorting buttons
    // ...

function enableSortingBtn():
    // Enable all sorting buttons
    // ...
```



```
function disableSizeSlider():
    // Disable the size slider
    // ...

function disableSpeedSlider():
    // Disable the speed slider
    // ...

function enableSpeedSlider():
    // Enable speed slider
    // ...

function enableSizeSlider():
    // Enable size slider
    // ...

function disableNewArrayBtn():
    // Disable the new array button
    // ...

function enableNewArrayBtn():
    // Enable the new array button
    // ...

function enableStopSortingBtn():
    // Enable the stop sorting button
    // ...

function disableStopSortingBtn():
    // Disable the stop sorting button
    // ...

function delayTime(milisec):
    // Delay for a specified time
    // ...

arraySize = document.querySelector('#size_input')
arraySize.addEventListener('input', function):
    // Event listener for size slider
    // ...

delay = 260
delayElement = document.querySelector('#speed_input')
delayElement.addEventListener('input', function):
    // Event listener for speed slider
    // ...
```

```
barArray = []
createNewArray(noOfBars = 60):
    // Create a new array of bars
    // ...

deleteChild():
    // Delete the existing bars
    // ...

newArrayButton = document.querySelector(".new")
newArrayButton.addEventListener("click", function):
    // Event listener for new array button
    // ...

stopSortingButton = document.querySelector(".stop")
stopSortingButton.addEventListener("click", function):
    // Event listener for stop sorting button
    // ...
```

4.3.3 CSS file

The "style.css" code contains CSS styles that can be utilized in a web application or website. Let's explore the development techniques, modules, and their respective functions within the system:

1. Development techniques:

- Styling
- Layout

2. System modules and functions:

- Text color module:
 - Sets the color property of elements with the class "text-color" to #EB455F.
- Body module:
 - Sets the background-color property of the <body> element to #BAD7E9.
- Flex container module:
 - Configures the flex container with the following properties:
 - margin-top: 20px
 - display: flex
 - flex-wrap: nowrap
 - width: 100%

- height: 600px
 - justify-content: center
 - transition: 2s all ease
- Flex item module:
 - Configures the styling properties for flex items with the following properties:
 - background: #e43f5a
 - border: 1pt solid black
 - width: 10px
 - transition: 0.1s all ease
- Row module:
 - Configures the grid layout for elements with the class "row" with two columns of equal width and one column of double width.
- Input panel module:
 - Adds 50 pixels of margin space above the element with the id "input-panel".
- Center module:
 - Centers the contents of elements with the class "center" both horizontally and vertically.
- Sorting module:
 - Adds 50 pixels of margin space below the element with the id "sorting".
 - Applies a transformation that rotates the element 180 degrees and flips it horizontally.
- Range input module:
 - Configures the styling properties for range input elements with the class "range-input" with various positioning and visual properties.
- Range input thumb module:
 - Configures the styling properties for the thumb of the range input with the class "range-input" and the pseudo-element "::webkit-slider-thumb".
- Range input thumb hover module:
 - with the class "range-input" and the pseudo-element "::webkit-

slider-thumb" on hover.

- Range input label module:
 - Configures the styling properties for the label of the range input with the class "range-input".
- Button module:
 - Configures the width property for elements with the class "btn" to 200 pixels, with important priority.

4.3.4 Algorithm files

In this section, we will delve into the coding files of 10 sorting algorithms used in this project. We'll discuss the development techniques employed, system modules, their purposes, and code implementation. Let's explore each algorithm and its corresponding module.

Bubble sort-

The "bubbleSort.js" file contains code that facilitates the visual sorting of elements on a webpage using the bubble sort algorithm. Let's explore the key aspects of this code, including the development techniques used, the modules involved, the functions they serve, and the implementation steps.

1. Development techniques:

This code uses asynchronous programming and event-driven programming. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to pause the execution and wait for a promise to resolve.

2. Modules:

- Sorting algorithm module:
 - Function: `bubble()`.
 - Event listener

3. Functions:

- `swap(element1, element2)`, `delayTime(ms)`, `disableSortingBtn()`, `disableSizeSlider()`, `disableNewArrayBtn()`, `enableStopSortingBtn()`, `enableSortingBtn()`, `enableSizeSlider()`, `enableNewArrayBtn()`, `disableStopSortingBtn()`, `disableSpeedSlider()`.

To implement the code step by step, follow these instructions:

1. Declare the bubble function as an asynchronous function.

2. Use `document.querySelectorAll` to select all elements with the class name "bar" and store them in the `ele` variable.
3. Iterate over the `ele` array using a for loop to perform the bubble sort algorithm.
4. Use another for loop to compare and swap elements in each iteration.
5. Check if the `hasPressedStop` flag is true, and if so, stop the sorting process.
6. Visualize the elements being compared by setting their background color to "cyan".
7. Compare the heights of elements and visually swap them if necessary, with a delay between swaps.
8. Visualize compared and possibly swapped elements by setting their background color to a shade of red.
9. Indicate the final sorted position of the last element in each iteration by setting its background color to "green".
10. Set the background color of the first element to "green" to indicate its final sorted position.
11. Select the bubble sort button with the class name "bubbleSort" using `document.querySelector` and store it in the `bubSortbtn` variable.
12. Add an event listener to the `bubSortbtn` button. Within the listener:
 - Set the `hasPressedStop` flag to false.
 - Disable the sorting button, size slider, and new array button.
 - Enable the stop sorting button.
 - Call the bubble function using `await` to complete the sorting process.
 - Adjust UI elements based on whether the sorting process was stopped or completed.
 - Enable the new array button and disable the stop sorting button.

Bucket sort -

The code in "bucketSort.js" implements the Bucket Sort algorithm. Let's explore the key aspects of this code, including the development techniques used and the modules involved:

1. Development techniques:
 - Asynchronous execution
 - DOM manipulation

2. Modules:

- bucketSort() function
- Event listener

3. Function:

- insertion()

To implement the given code in bucketSort.js, follow these instructions:

1. Declare the bucketSort function as an asynchronous function for performing the bucket sort algorithm.
2. Use document.querySelectorAll to select all elements with the class name "bar" and store them in the ele variable.
3. Find the maximum element in the ele array and calculate the bucket size based on the maximum value and the number of elements.
4. Initialize an empty bucket array with a length equal to the number of elements.
5. Iterate over the bucket array to initialize each element as an empty array.
6. Iterate over the ele array, calculate the bucket index for each element, and update the corresponding bucket with the element. Update the element's background color for visualization.
7. Sort each non-empty bucket and update the ele array with the sorted elements, updating their background color. Introduce delays for visualization purposes.
8. Enable the sorting button, size slider, and new array button.
9. Add an event listener to the bucket sort button. Within the listener, disable relevant buttons and sliders, enable the stop sorting button, and call the bucketSort function using await.
10. After the sorting process, check if the stop sorting button was pressed. Adjust the UI accordingly by disabling or enabling the speed slider, sorting button, and size slider.
11. Enable the new array button and disable the stop sorting button.

Counting sort-

The code in "countingSort.js" implements the Counting Sort algorithm. Let's explore the key aspects of this code, including the development techniques used and the modules involved.

1. Development technique:

- Asynchronous programming.

2. Modules:

- countingSort() function.
- Event listener for Counting sort button.

3. User interface (UI) functions:

- disableSortingBtn(), disableSizeSlider(), disableNewArrayBtn(), enableStopSortingBtn(), disableSpeedSlider(), enableSortingBtn(), enableSizeSlider(), enableNewArrayBtn(), disableStopSortingBtn().

To implement the code step by step, follow these instructions:

1. Select all elements with the class "bar" using the `document.querySelectorAll(".bar")` and assign them to the file variable.
2. Find the maximum value by comparing the heights of the elements and store it in the `maxValue` variable.
3. Create a counting array named `count` with a length of `maxValue + 1` and initialize all elements with zeros.
4. Iterate through the elements in `ele`:
 - Increment the corresponding count in the `count` array based on the parsed integer value of the element's height.
 - Set the background color of the element to "cyan".
 - Pause the execution for a specific time using the `delayTime` function.
5. Initialize a variable `j` to 0.
6. Iterate through the `count` array:
 - Use a nested while loop to handle non-zero counts:
 - Set the height of `ele[j]` to the current value.
 - Decrement the count for the current value.
 - Increment `j`.
 - Set the background color of the previous element to "green".
 - Pause the execution for a specific time using the `delayTime` function.

7. Add an event listener to the countingSort button with the class "countingSort":

- When clicked, execute the provided function.
- Set the hasPressedStop variable to false.
- Disable sorting buttons, size sliders, and new array buttons.
- Enable the stop sorting button.
- Wait for the execution of the countingSort function to complete.
- Check if hasPressedStop is true:
 - If true, disable the speed slider.
 - If false, enable sorting buttons and size sliders.
- Enable the new array button.
- Disable the stop sorting button.

Heap sort -

The given code in “heapsort.js” implements the Heap Sort algorithm. Let's break down the development techniques, modules, and functions in the system:

1. Development techniques:

- Asynchronous programming.

2. Modules:

- Heap sort module.
- UI control module.

3. Functions:

- heapSort(arr, n).
- heapify(arr, n, i).
- Event Listener for Heap Sort Button.

To implement the code, follow these steps:

1. Define the heapSort function with parameters arr and n.
2. Implement the first loop in heapSort to build the heap by calling heapify for each non-leaf node from the bottom up. Start from $i = n / 2 - 1$ and decrement i until it reaches 0. Inside the loop, check if hasPressedStop is

- true. If true, return from the function. Await the execution of `heapify(arr, n, i)`.
3. Implement the second loop in `heapSort` to extract elements from the heap and sort the array. Start from $i = n - 1$ and decrement i until it reaches 0. Inside the loop, check if `hasPressedStop` is true. If true, return from the function. Swap the current root element (`arr[0]`) with the last element (`arr[i]`). Set the background color of `arr[0]` to 'cyan' and `arr[i]` to 'green'. Call the swap function with `arr[0]` and `arr[i]`. Await the execution of `delayTime(delay)`. Call `heapify(arr, i, 0)` to restore the heap property.
 4. Define the `heapify` function with parameters `arr`, `n`, and `i`. Check if `hasPressedStop` is true. If true, return from the function. Initialize `largest` as `i`. Calculate the left child index `l` and the right child index `r`. If the left child is larger than the current largest element, update the largest to `l` and call the swap function with `arr[largest]` and `arr[l]`. If the right child is larger than the current largest element, update the largest to `r` and call the swap function with `arr[largest]` and `arr[r]`. If the largest is not equal to `i`, swap `arr[i]` with `arr[largest]`. Call `heapify(arr, n, largest)` recursively to heapify the affected sub-tree.
 5. Select the element with the class "heapSort" and assign it to the variable `heapSortbtn` using the `document.querySelector(".heapSort")`.
 6. Add an event listener to the `heapSortbtn` element (assumed to be a button). When the button is clicked, execute the provided function. Select all elements with the class "bar" and assign them to the variable `arr` using `document.querySelectorAll('.bar')`. Set `n` to the length of `arr`. Set `hasPressedStop` to false. Disable sorting buttons, size sliders, and new array buttons. Enable the stop sorting button. Await the execution of `heapSort(arr, n)`. Set the background color of `arr[0]` to 'green'. Check if `hasPressedStop` is true. If true, disable the speed slider. If false, enable sorting buttons and size sliders. Enable the new array button. Disable the stop sorting button. Ensure the necessary HTML elements are present (e.g., elements with the class "bar", a button with the class "heapSort", etc.).

Insertion sort-

The code in "insertionSort.js" implements the insertion sort algorithm. Let's examine its components, development techniques, modules used, and the purpose of each module:

1. Development techniques:

- Asynchronous functions.

2. Modules Used:

- Document Object Model (DOM)

3. Functions:

- insertion().
- inSortbtn Event Listener.
- disableSortingBtn(), disableSizeSlider(), disableNewArrayBtn(), enableStopSortingBtn(), disableStopSortingBtn(), disableSpeedSlider(), enableSortingBtn(), enableSizeSlider(), enableNewArrayBtn(),

To implement the code, follow these instructions:

1. Define the insertion function.
2. Select all elements with the class "bar" and assign them to the variable ele using the document.querySelectorAll(".bar").
3. Set the background color of ele[0] to 'green'.
4. Implement a loop starting from i = 1 and iterating until i < ele.length:
 - Inside the loop, check if hasPressedStop is true. If true, return from the function.
 - Set j to i - 1.
 - Assign the height of ele[i] to key.
 - Set the background color of ele[i] to 'blue'.
 - Await the execution of delayTime(delay).
 - Check if hasPressedStop is true. If true, return from the function.
 - Implement a while loop starting from j and iterating until j >= 0 and the height of ele[j] is greater than the key:
 - Inside the while loop, check if hasPressedStop is true. If true, return from the function.
 - Set the background color of ele[j] to 'blue'.

- Assign the height of ele[j] to the height of ele[j + 1].
 - Decrement j by 1.
 - Await the execution of delayTime(delay).
 - Check if hasPressedStop is true. If true, return from the function.
 - Implement a loop starting from k = i and iterating until k >= 0:
 - Inside the loop, set the background color of ele[k] to 'green'.
 - Set the height of ele[j + 1] to key.
 - Set the background color of ele[i] to 'green'.
5. Select the element with the class "insertionSort" and assign it to the variable inSortbtn using the document.querySelector(".insertionSort").
 6. Add an event listener to the inSortbtn element (assumed to be a button):
 - When the button is clicked, execute the provided function.
 - Disable sorting buttons, size sliders, and new array buttons.
 - Enable the stop sorting button.
 - Await the execution of insertion().
 - Check if hasPressedStop is true:
 - If true, disable the speed slider.
 - If false, enable sorting buttons and size sliders.
 - Enable the new array button.
 - Disable the stop sorting button.

Merge sort -

The code in "mergeSort.js" implements the merge sort algorithm for visualizing the sorting process. Let's go through the different aspects of the code, including the development techniques, modules used, and the functions of each module.

1. Development techniques:

- Asynchronous functions.

2. Modules Used:

- Document Object Model (DOM).

3. Functions and their Descriptions:

- `merge()`:
 - Merges two sorted subarrays into a single sorted array.
 - Parameters: `ele` (array of DOM elements representing bars), `low` (starting index of the first subarray), `mid` (ending index of the first subarray and starting index of the second subarray), `high` (ending index of the second subarray).
- `mergeSort()`:
 - Implements the merge sort algorithm recursively.
 - Parameters: `ele` (array of DOM elements representing bars), `l` (starting index of the current subarray), `r` (ending index of the current subarray).
- `mergeSortbtn`:
 - Event listener for the merge sort button.
 - Executes the merge sort algorithm when clicked, updating the visual representation of the bars on a webpage.

To implement the given code, follow these steps:

1. Define the merge function.
2. Log the message 'In merge()' to the console.
3. Calculate the values of `n1` and `n2`.
4. Create empty arrays, `left` and `right`.
5. Implement a loop to populate the left array.
 - Check if `hasPressedStop` is true. If true, return from the function.
 - Await the execution of `delayTime(delay)`.
 - Set the background color of `ele[low + i]` to 'orange'.
 - Assign the `style.height` value of `ele[low + i]` to `left[i]`.
6. Implement a loop to populate the right array.
 - Check if `hasPressedStop` is true. If true, return from the function.
 - Await the execution of `delayTime(delay)`.
 - Set the background color of `ele[mid + 1 + i]` to 'cyan'.

- Assign the style.height value of ele[mid + 1 + i] to right[i].
7. Await the execution of delayTime(delay).
 8. Initialize variables i, j, k, and low.
 9. Implement a while loop with the condition $i < n1 \ \&\& \ j < n2$.
 - Check if hasPressedStop is true. If true, return from the function.
 - Await the execution of delayTime(delay).
 - Compare elements from the left and right arrays.
 - Set the background color of ele[k] based on the comparison result.
 - Set the style. height value of ele[k] to the appropriate value.
 - Increment i and k.
 10. Implement a while loop with the condition $i < n1$.
 - Check if hasPressedStop is true. If true, return from the function.
 - Await the execution of delayTime(delay).
 - Set the background color of ele[k].
 - Set the style.height value of ele[k] to the value from the left array.
 - Increment i and k.
 11. Implement a while loop with the condition $j < n2$.
 - Check if hasPressedStop is true. If true, return from the function.

Quick sort-

The provided code in “quicksort.js” appears to be an implementation of the QuickSort algorithm using the Lomuto partition scheme. Let's break it down and discuss the development techniques, modules used, and the functions of each module.

1. Development techniques:

- Asynchronous functions.

2. Modules Used:

- Document Object Model (DOM)

3. Functions:

- partitionLomuto()
- quickSort()
- quickSortbtn event listener

To implement the provided code, you can follow these instructions:

1. Define the partitionLomuto function.
2. Initialize i as l - 1.
3. Set the background color of ele[r] (pivot) to 'cyan'.
4. Implement a for loop from j = l to r - 1.
 - Check if hasPressedStop is true. If true, return from the function.
 - Set the background color of ele[j] to 'yellow'.
 - Await the execution of delayTime(delay).
 - Check if hasPressedStop is true. If true, return from the function.
 - Compare the heights of ele[j] and ele[r].
 - Swap ele[i] and ele[j] if necessary.
 - Set the background color of ele[i] and ele[j].
 - Await the execution of delayTime(delay).
 - Set the background color of ele[j].
5. Increment i by 1.
6. Check if hasPressedStop is true. If true, return from the function.
7. Await the execution of delayTime(delay).
8. Check if hasPressedStop is true. If true, return from the function.
9. Swap ele[i] and ele[r].
 - Set the background color of ele[r] and ele[i].
10. Check if hasPressedStop is true. If true, return from the function.
11. Await the execution of delayTime(delay).
12. Check if hasPressedStop is true. If true, return from the function.
13. Set the background color of elements in ele.
14. Return i.
15. Define the quickSort function.
16. Check if l is less than r.
 - If true, recursively call partitionLomuto and quickSort.
17. Check if l and r are within valid indices.
 - If true, set the background color of ele[r] and ele[l] to 'green'.
18. Define the quickSortbtn button and add a click event listener to it.
19. Inside the event listener function:

- Assign ele, l, and r.
- Call various functions to enable/disable elements.
- Await the execution of quickSort.
- Check if hasPressedStop is true and call the appropriate functions.

Radix sort-

The provided code in "radixSort.js" implements the radix sort algorithm in JavaScript. Let's explore the development techniques, modules used in the system, and the functions of each module.

1. Development techniques:

- Asynchronous programming

2. Modules in the system:

- Sorting module (Radix sort)

3. Functions of each Module:

- Sorting Module:
 - radixSort()
- Helper functions:
 - getMax()
 - getMaxDigits()
- Event handlers:
 - disableSortingBtn(), disableSizeSlider(), disableNewArrayBtn(), enableStopSortingBtn(), enableSortingBtn(), enableSizeSlider(), enableNewArrayBtn(), and disableStopSortingBtn().

To implement the code, we need to follow these instructions:

1. Define the radixSort function.
2. Assign elements with the class name "bar" to the ele variable.
3. Find the maximum number among the elements using the getMax function.
4. Start a loop while $\text{Math.floor}(\text{maxNum}/\text{exp}) > 0$.
5. Inside the loop, call the countingSort function with ele and exp as arguments.
6. Multiply exp by 10 in each iteration.
7. Define the getMax function.

8. Initialize max as the maximum value among the elements.
9. Loop through the elements and update max if a larger value is found.
10. Return the max value.
11. Define the getMaxDigits function.
12. Initialize maxDigits as 0.
13. Loop through the elements and update maxDigits with the maximum number of digits.
14. Return the maxDigits value.
15. Assign the "radixSort" button to the radixSortbtn variable.
16. Add a click event listener to radixSortbtn.
17. Inside the event listener function:
 - Set hasPressedStop to false.
 - Call various functions to enable/disable elements.
 - Await the execution of radixSort.
 - Check if hasPressedStop is true and call the appropriate functions.

Selection sort -

This code in “selectionSort.js” is an implementation of the selection sort algorithm in JavaScript. Let's break it down into different parts and explain their functions.

1. Development technique:
 - Asynchronous programming
 - DOM manipulation
2. Modules:
 - Selection sort module
 - Selection sort button event listener module
3. Functions of each module:
 - Selection sort module:
 - Selection sort function: Implements the selection sort algorithm. It iterates through the elements, compares and swaps them, and updates their colors accordingly.
 - Selection sort button event listener module:
 - Event listener functions: Handles the event listener for the

selection sort button. It initializes the sorting process, disables and enables buttons accordingly, and calls the selection sort function.

Here's a breakdown of the implementation steps:

1. Create a function named "selection" with the following code:
 - Select all elements with the class name "bar" and assign them to the variable "ele".
 - Set up a loop to iterate over the elements in "ele".
 - Check if "hasPressedStop" is true and return to exit the function.
 - Initialize a variable "min_index" with the value of the current index.
 - Set the background color of the current element to 'lightgreen'.
 - Set up another loop to compare the current element with the remaining elements.
 - Check if "hasPressedStop" is true and return to exit the function.
 - Set the background color of the compared element to 'cyan'.
 - Introduce a delay using the "delayTime" function.
 - Compare the heights of the compared element and the minimum element.
 - Update "min_index" if a smaller height is found.
 - Reset the background color of the compared element.
 - Introduce another delay.
 - Swap the elements at "min_index" and the current index.
 - Reset the background color of the minimum element and set the current element to 'green'.
1. Create a variable named "selectionSortbtn" and select the button element with the class name "selectionSort".
2. Add a click event listener to "selectionSortbtn" with an anonymous async function as the callback.
3. Inside the event listener callback function:
 - Set "hasPressedStop" to false.
 - Call functions to enable/disable certain buttons or sliders.
 - Await the execution of the "selection" function.

- Check if "hasPressedStop" is true and call the appropriate functions.
- Call functions to enable/disable buttons as needed.

Shell sort -

The provided code in "shellSort.js" implements the shell sort algorithm in a visualizer for sorting algorithms. Let's break it down and explain the functions of different parts:

1. Development techniques:

- Asynchronous programming

2. Modules and Functions:

- shellSort() function: Implements the shell sort algorithm by iterating through the elements and performing comparisons and swaps based on the gap size.
- Event listeners and DOM manipulation: Handles the button click event for shell sort, manages the state of buttons and sliders, and updates the visualizer accordingly.

To implement the code, follow these instructions:

1. Define the delayTime and swap functions.
2. Create an HTML document with necessary elements, including a container element with the class "bar" and a button with the class "shellSort". Add other buttons and sliders as needed.
3. Add event listeners to buttons and sliders to handle respective actions.
4. In the shellSort function:
 - Select all elements with class "bar" and store them in the ele variable.
 - Set the initial gap value based on ele length.
 - Use a while loop with gap as the condition.
 - Inside the while loop, add a for loop from gap to ele length.
 - In the for loop:
 - Assign i to j and store the current element height.
 - Use a while loop with j and gap condition, comparing heights.
 - Inside the inner while loop:

- Change the background color to 'cyan'.
 - Introduce a delay using `await delayTime`.
 - Call the swap function.
 - Reset the background color to the original value.
 - Decrement `j` by `gap`.
 - Set the height of the element at index `j` to the current value.
 - Update the `gap` by halving it.
5. Select the button with the class "shellSort" and assign it to the `shellSortbtn` variable.
 6. Add a click event listener to `shellSortbtn` using `addEventListener`.
 7. Inside the event listener:
 - Set `hasPressedStop` to false.
 - Disable the sorting button, size slider, and new array button.
Enable the stop button.
 - Asynchronously invoke `shellSort` using `await`.
 - Check the `hasPressedStop` value after sorting. Disable or enable the speed slider accordingly.
 - Enable the sorting button and size slider. Enable the new array button. Disable the stop button.

4.4 Testing

This report provides an overview of the testing activities performed on the Sorting Visualizer Web Application. The application was developed using HTML, CSS, and JavaScript, and this document aims to document the testing process and its outcomes.

Objectives of testing: The primary goals of the testing process were as follows:

1. Validate the accuracy and correctness of the sorting algorithms.
2. Ensure the visual representation of the sorting process is displayed accurately.
3. Verify the responsiveness and compatibility of the application across different browsers.

4. Identify and resolve any bugs, errors, or performance issues.

Test environment: The Sorting Visualizer Web Application was tested in the following environment:

- Operating System: Windows 10
- Browsers: Chrome, Firefox, Safari

Testing approach: The testing approach involved a combination of manual and automated techniques.

b. Visual testing:

- Verification of the visual representation of the sorting process to ensure accuracy and clarity.

Manual testing was used to evaluate the functionality and visual aspects, while automated tests were employed to verify algorithm correctness and performance.

Testing Scenarios and Results:

a. Functionality testing:

- Individual testing of sorting algorithms, including Bubble Sort (Figure 4.2, 4.3, 4.4), Quick Sort, Merge Sort, etc. to ensure correct sorting for different array sizes and types.

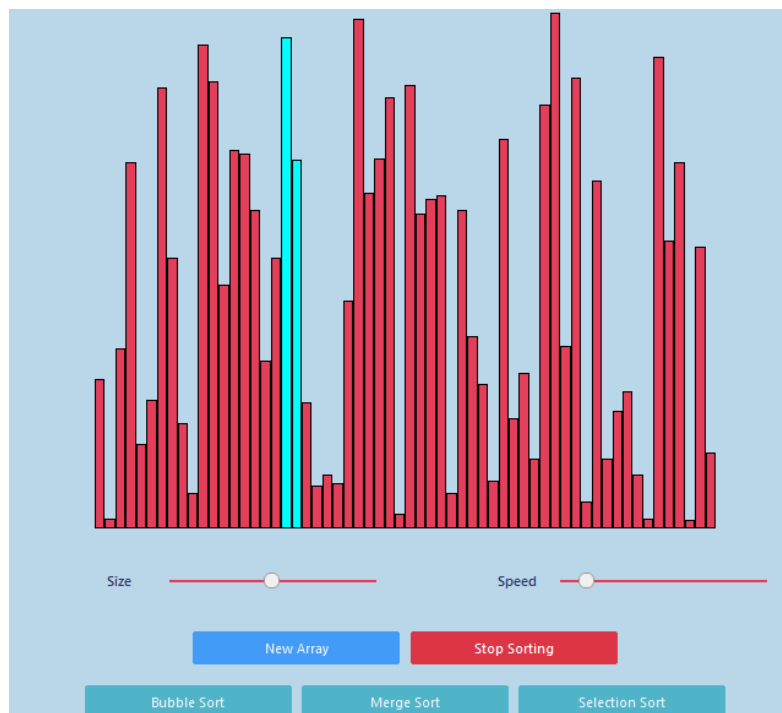


Figure 4.2: Test: sorting accuracy and clarity

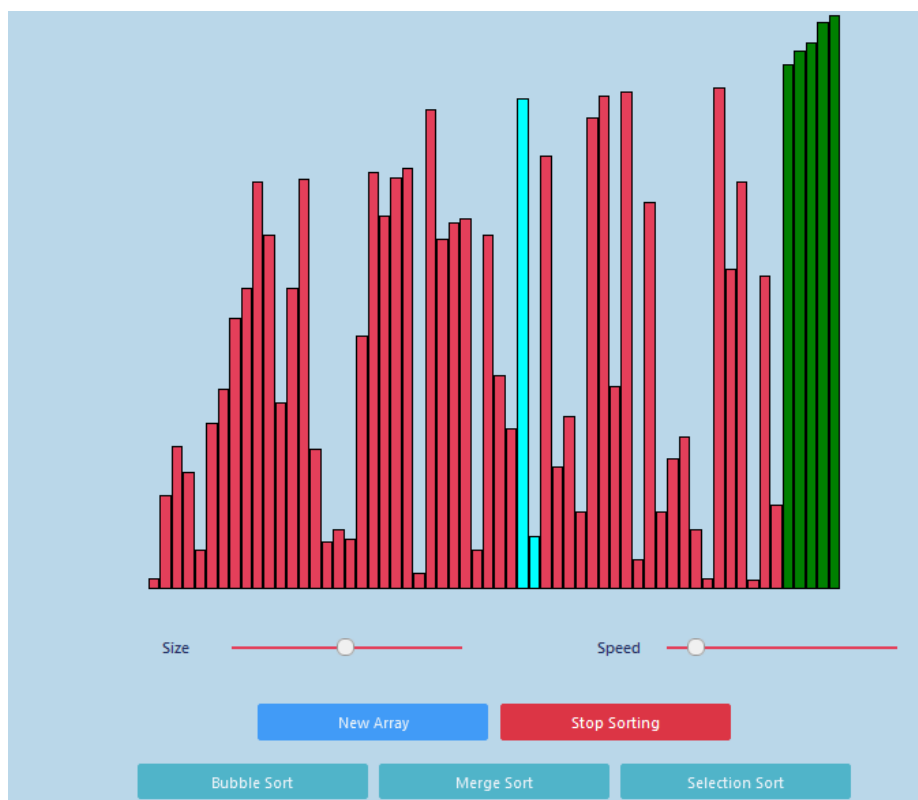


Figure 4.3: Test: Visual representation of sorting



Figure 4.4: Test: Successfully sorted array

- Testing of different input sizes to confirm that the visual display adjusts

appropriately and provides an optimal user experience (Figure 4.5).

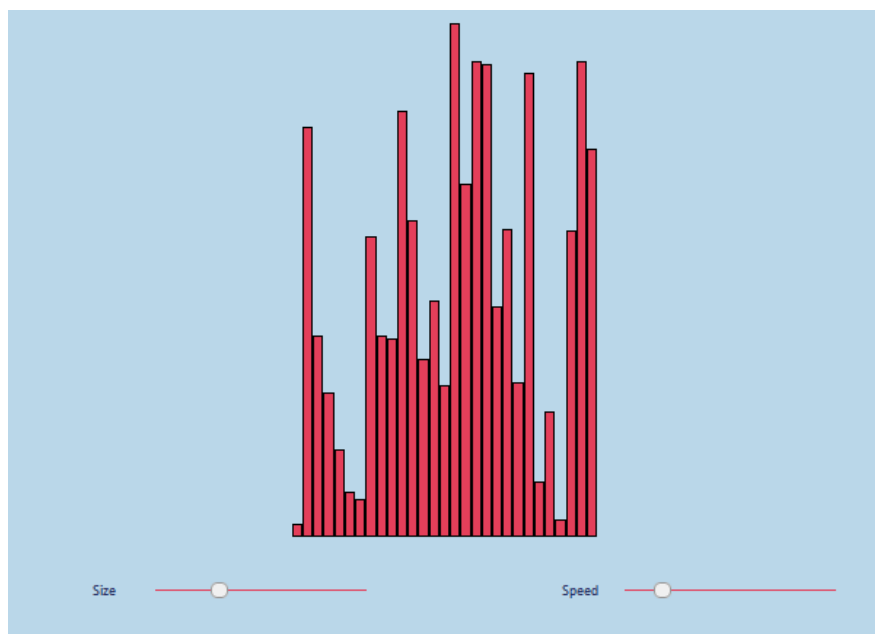


Figure 4.5: Test: different input size of an array

- Interact with the UI elements, such as changing the size and speed sliders, clicking the "New Array" button, and stopping the sorting process using the "Stop Sorting" button (Figure 4.6). Verify that the corresponding actions are performed and the UI updates accordingly.

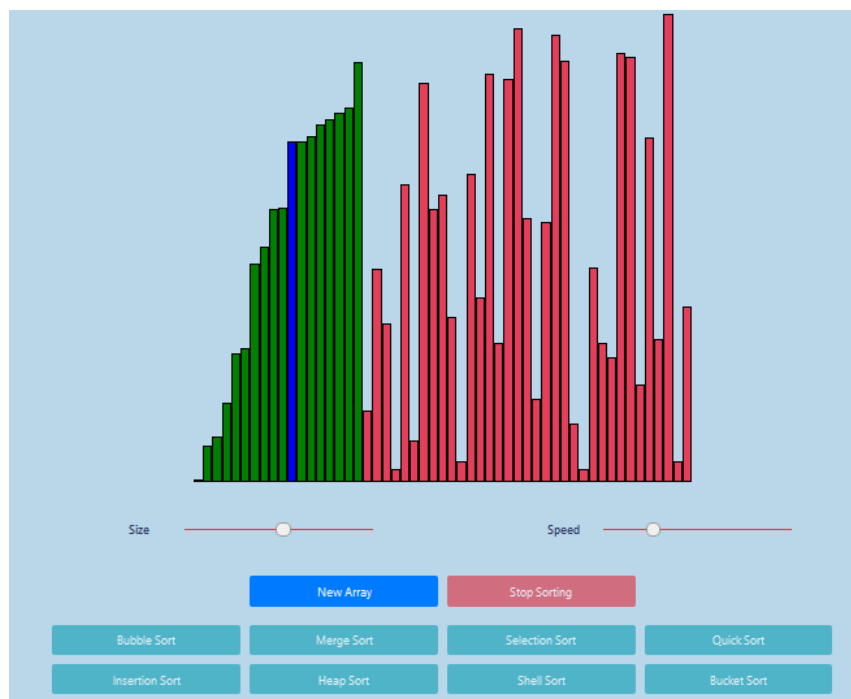


Figure 4.6: Test: Stop sorting button

c. Performance Testing:

- Execution of automated performance tests to measure the efficiency and responsiveness of the application when sorting large input sizes.
- Analysis of the application's execution time and resource consumption to identify any performance bottlenecks (Figure 4.7).

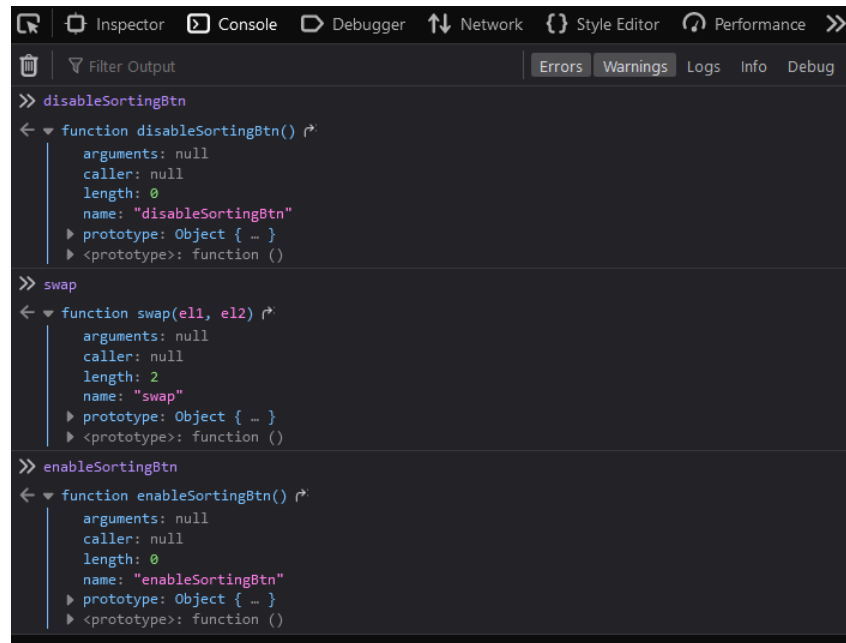


Figure 4.7: Test: Sorting functions

Table 4.1: User Functional Testing

Test Case	Preconditions	Test Steps	Expected Results
Bubble Sort	The application is launched and in a functional state.	- Select "Bubble Sort" from the sorting algorithm options. Set the array size to random. Click on the "Sort" button.	- The application disables user interactions and starts animating the bubble sort process. The array elements are visually sorted, and the application highlights the sorted elements once the sorting is complete.
Stop Sorting	The application is sorting an array using the merge sort algorithm.	- Click on the "Stop" button during the sorting process.	- The sorting process is interrupted, and the application returns to the initial state, allowing further interactions.

Change Sorting Algorithm	The application is sorting an array using the bubble sort algorithm.	- Select "Merge Sort" from the sorting algorithm options. Click on the "Sort" button.	- The bubble sort process is interrupted, and the application starts sorting the array using the merge sort algorithm.
Adjust Sorting Parameters	The application is launched and in a functional state.	- Select "Insertion Sort" from the sorting algorithm options. Set the array size. Click on the "Sort" button.	- The application disables user interactions and starts animating the insertion sort process. The array elements are visually sorted, and the application highlights the sorted elements once the sorting is complete.

Defects and issue tracking: During the testing process, the following issues were identified:

- Issue 1: The Quick Sort algorithm occasionally produced incorrect results for specific input patterns.
- Issue 2: The algorithm occasionally stops working when stop sorting click because it takes time to stop the loop function.

Conclusion: The Sorting Visualizer Web Application demonstrated overall robust functionality and provided an effective visualization of sorting algorithms. Most of the identified issues were addressed and resolved during testing.

4.5 Deployment and Maintenance

To deploy the Sorting Visualizer Web Application on GitHub, follow these steps:

1. Create a GitHub repository:

- Sign in to your GitHub account (or create a new one).
- Click on the "+" button in the top-right corner and select "New repository".
- Provide a name for your repository and choose any desired settings (e.g., public or private).
- Click on "Create repository" to create the repository.

2. Upload your web application code:

- Clone the GitHub repository to your local development environment using Git or GitHub Desktop.
- Copy the files of your Sorting Visualizer Web Application into the local repository folder.
- Commit and push the files to the GitHub repository using Git commands or the GitHub Desktop application.

3. Configure repository settings:

- Go to your GitHub repository page and click on the "Settings" tab.
- Scroll down to the "GitHub Pages" section.
- Under "Source", select the branch that contains your web application code (e.g., "main" or "master").
- Optionally, choose a specific folder if your code is located in a subdirectory.
- Click on "Save" or "Save Changes".

4. Access your deployed web application:

- After saving the settings, a green success message will appear with the URL of your deployed web application.
- Visit the provided URL to access your Sorting Visualizer Web Application.

5. Maintenance and updates:

- For future updates or changes to your web application, make the necessary modifications to your local code.
- Commit and push the changes to the GitHub repository.
- GitHub Pages will automatically redeploy your updated web application based on the repository settings.
- Test the deployed application to ensure the changes are reflected correctly.

By following these steps, we can deploy our Visualizer for Sorting Web Application on GitHub and make it accessible to users.

5 User Interface

As it was stated above, the main aim of the application is to be as user-friendly as possible. It provides an interactive platform to visualize how different sorting algorithms function in real-time. In this section, we will describe how to use the visualization of sorting applications.

5.1 System Requirements for User

Here are some system requirements you will need to use the application:

- A computer device with an internet connection
- A modern web browser such as Google Chrome, Mozilla Firefox, Safari, or Microsoft Edge
- Input a device like a mouse (or touchpad) to interact with the visualization.
- No installation or setup is required on the user's end.

Overall, the system requirements for using visualization for sorting web applications are relatively low, since the processing and rendering are done on the server side rather than the client side. The most important consideration is ensuring that your web browser is up-to-date and compatible with the web application.

5.2 User Guide

First of all, we need a browser to access the web application. We can use this- “https://hridoy88.github.io/sorting_visualizer/” link to access the website.

1. Landing page:

Upon opening the visualizer for sorting applications, users will see a landing page of the application (Figure 5.1).

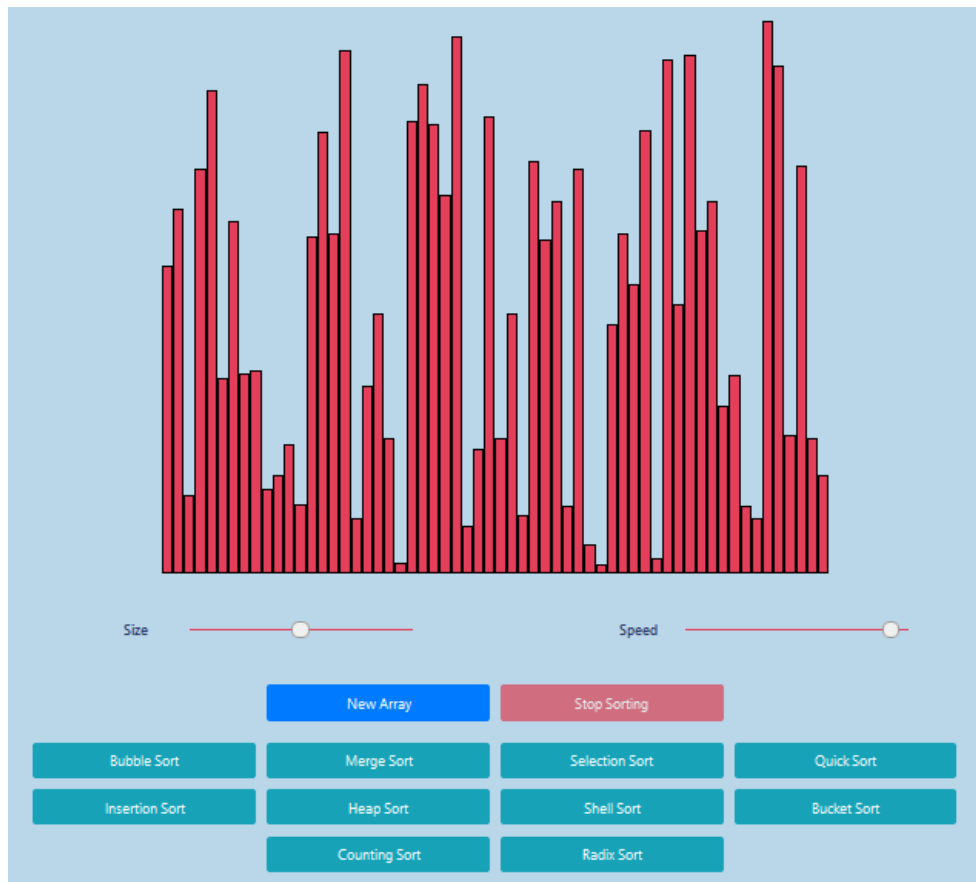





Figure 5.1: Main window

2. Visualization panel:

By using Visualizer, we can create a new array, and choose the size of the array, and speed. The speed and size of the array are divided into 5 ranges (Figure 5.2). The speed of the visualization is directly depending upon the size of the array. The size of the array directly determines the width and block size. The sorting process majorly depends upon the sorting algorithm used, and array size, so the time taken to visualize the sorting process may also change. In the present work, to keep the visualization process time the same for all the algorithms we have normalized the speed of the sorting algorithms concerning the speed selected by the user. When we choose to generate a new array, a new array with some random size is created, and the array is visualized in the form of a bar graph. We have added a very interesting feature to the visualizer which is buttons and controls, which can be unclickable or locked during

the running to avoid problems like missing the flow of algorithms. For effective visual effects, we have used different colors such that-

-  → Sorted
-  → Selection from sorted
-  → Selection from unsorted

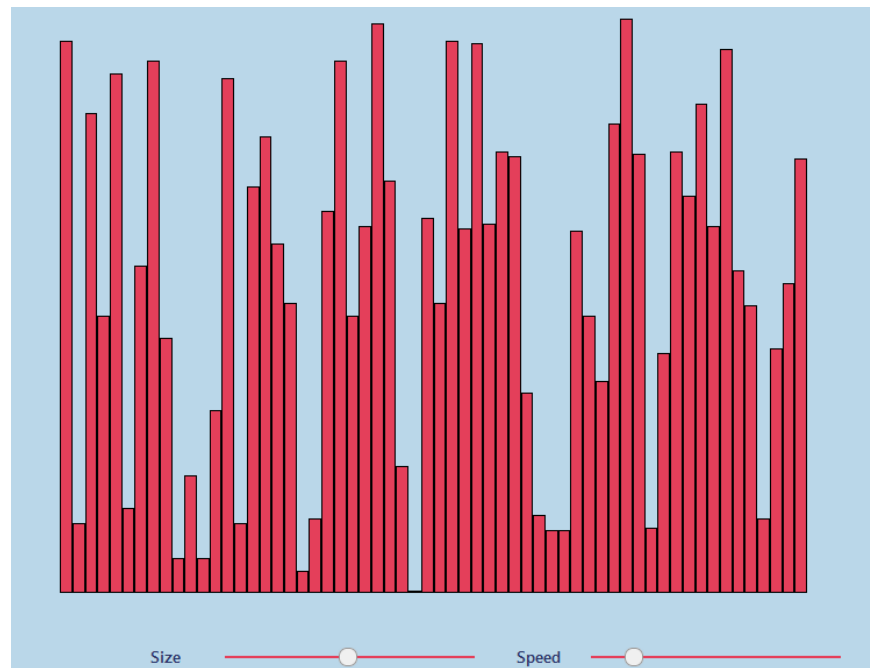


Figure 5.2: Visualization panel

3. Starting the visualization:

Click on any algorithm located at the bottom of the screen (Figure 5.3), to select an algorithm that we would like to visualize. The sorting process will start to display visually, with each step animated and displayed on the screen. We can pause the animation at any time by using the “Stop Sorting” button. Also, we can change the speed of sorting while it’s running for the tying process. After sorting any algorithm, we have to take a new array by clicking on the “New Array” button to start a new algorithm.



Figure 5.3: Algorithm selector

4. Viewing the results:

Once the sorting process is completed, the sorted array will be displayed like this in Figure 5.4.

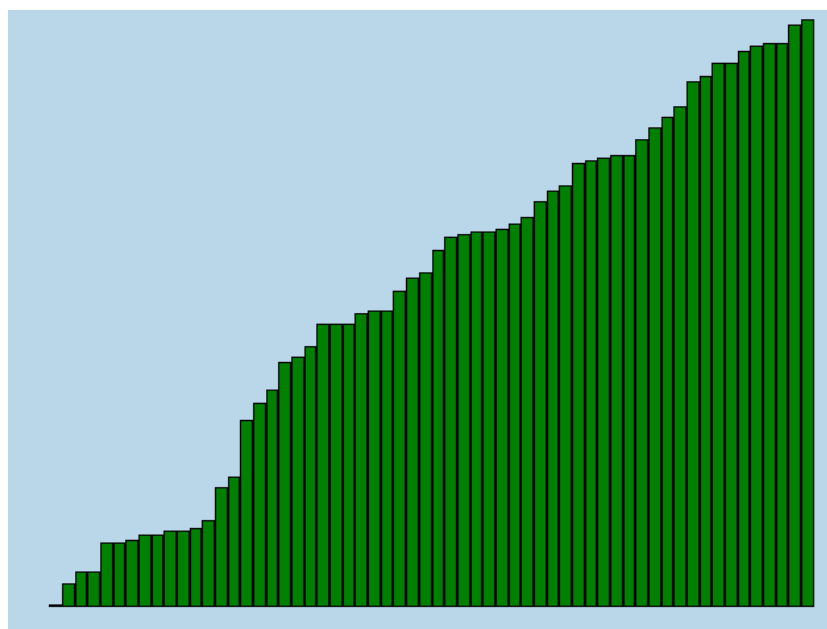


Figure 5.4: Successful Sorting of bubble sort

5. Troubleshooting:

If you encounter any difficulties with the Visualizer of Sorting Algorithm web application, you can try resolving the problem by refreshing the page or clearing your browser's cache. This can help to address any potential issues related to caching or temporary data stored by the browser.

Conclusion

The only aim of this project is to demonstrate the sorting algorithms using visualization through a web application. This project aims to demonstrate the power of sorting algorithms through this website. Our website provides a clear and detailed idea about the sorting algorithms, and how the comparisons and swaps are performed during the sorting. The visualizer will help the students to better understand the different sorting algorithms considered in very less time.

Acronyms

HTML - Hypertext Markup Language

CSS - Cascading Style Sheets

UI - User Interface

DOM - Document Object Model

JS - Javascript

Bibliography

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, D. L.; STEIN, C. Introduction to algorithms. Second Edition. 2001.
- KNUTH, D. The Art of Computer Programming: Fundamental Algorithms. Third Edition. 2004.
- KNUTH, D. The Art of Computer Programming: Sorting and Searching. Second Edition. 2004.
- SEDGEWIK, R. Algorithms in C: Fundamentals, data structures, sorting, searching. Third Edition. 2007.
- GeeksforGeeks, Available from: 《<https://www.geeksforgeeks.org/>》.
- Programiz, Available from: 《www.programiz.com/》
- Stackoverflow, Available from: 《<https://stackoverflow.com/>》.
- 钟立荣, & 于洋. (2020). 基于 WebGL 的排序算法可视化设计与实现 [Design and implementation of sorting algorithm visualization based on WebGL]. 计算机应用研究, 37(6), 1746-1750.
- 陈健, & 张蕾. (2021). 排序算法可视化教学平台的设计与实现 [Design and implementation of sorting algorithm visualization teaching platform]. 软件导刊, 20(1), 61-66

Acknowledgments

At the conclusion of this paper, I would like to express my deepest gratitude to my supervisor. Throughout the preparation of this document, he provided me with invaluable suggestions on various aspects, including the topic, research methodology, and language usage.

Moreover, I am extremely thankful to my classmates who supported me during this arduous and time-consuming writing process. Their encouragement and camaraderie made a challenging journey much easier.

I must also acknowledge the unwavering support of my family, particularly my parents, who have been a constant source of inspiration and motivation. They always stood by me whenever I face difficulties, and it is through their sacrifices that I was able to attend university and gain access to knowledge and diverse experiences.

Without their assistance, I do not believe I could have completed this paper as smoothly as I did, and for that, I am immensely grateful.