

**B.Tech. BCSE497J - Project-I**

**SORTING VISUALIZER**

*Submitted in partial fulfillment of the requirements for the degree of*

**Bachelor of Technology**

*in*

**Computer Science and Engineering**

*By*

**21BCE2483 ARSH SHARMA**

**21BCI0409 ANURAG SINHA**

**Under the Supervision of**

**Dr. MADIAJAGAN M**

Professor Grade 1

School of Computer Science and Engineering (SCOPE)



**VIT<sup>®</sup>**  
**Vellore Institute of Technology**  
(Deemed to be University under section 3 of UGC Act, 1956)

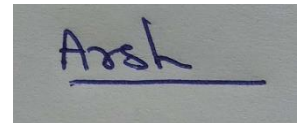
## **DECLARATION**

I hereby declare that the project entitled **Sorting Visualizer** submitted by me, for the award of the degree of *Bachelor of Technology in Computer Science and Engineering* to VIT is a record of bonafide work carried out by me under the supervision of Prof. / Dr. **Madiajagan M**

I further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place : Vellore

Date : 20-11-2024

A rectangular box containing a handwritten signature in blue ink. The signature appears to be 'Arsh' written in a cursive style, with a horizontal line underneath the name.

**Signature of the Candidate**

## **CERTIFICATE**

This is to certify that the project entitled **Sorting Visualizer** submitted by **Arsh Sharma (21BCE2483)**, School of Computer Science and Engineering, VIT, for the award of the degree of *Bachelor of Technology in Computer Science and Engineering*, is a record of bonafide work carried out by him / her under my supervision during Fall Semester 2024-2025, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The project fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place : Vellore

Date : 20-11-2024

**Signature of the Guide**

**Dr. UMADEVI K S**

**BTECH - Computer Science and Engineering**

## **ACKNOWLEDGEMENTS**

I am deeply grateful to the management of Vellore Institute of Technology (VIT) for providing me with the opportunity and resources to undertake this project. Their commitment to fostering a conducive learning environment has been instrumental in my academic journey. The support and infrastructure provided by VIT have enabled me to explore and develop my ideas to their fullest potential.

My sincere thanks to Dr. Ramesh Babu K, the Dean of the School of Computer Science and Engineering (SCOPE), for his unwavering support and encouragement. His leadership and vision have greatly inspired me to strive for excellence. The Dean's dedication to academic excellence and innovation has been a constant source of motivation for me. I appreciate his efforts in creating an environment that nurtures creativity and critical thinking.

I express my profound appreciation to Dr. Umadevi K S, the Head of the Computer Science & Engineering, for his/her insightful guidance and continuous support. His/her expertise and advice have been crucial in shaping the direction of my project. The Head of Department's commitment to fostering a collaborative and supportive atmosphere has greatly enhanced my learning experience. His/her constructive feedback and encouragement have been invaluable in overcoming challenges and achieving my project goals.

I am immensely thankful to my project supervisor, Dr. Madijagan M, for his/her dedicated mentorship and invaluable feedback. His/her patience, knowledge, and encouragement have been pivotal in the successful completion of this project. My supervisor's willingness to share his/her expertise and provide thoughtful guidance has been instrumental in refining my ideas and methodologies. His/her support has not only contributed to the success of this project but has also enriched my overall academic experience.

Thank you all for your contributions and support.

**ARSH SHARMA 21BCE2483**

**Name of the Candidate**

## TABLE OF CONTENTS

Sl.No	Contents	Page No.
	<b>Abstract</b>	<b>X</b>
<b>1.</b>	<b>INTRODUCTION</b>	<b>1</b>
	1.1 Background	1
	1.2 Motivations	1
	1.3 Scope of the Project	1
<b>2.</b>	<b>PROJECT DESCRIPTION AND GOALS</b>	<b>XIII</b>
	2.1 Literature Review	XIII
	2.2 Research Gap	XIV
	2.3 Objectives	XV
	2.4 Problem Statement	XVI
	2.5 Project Plan	XVII
<b>3.</b>	<b>TECHNICAL SPECIFICATION</b>	<b>XVIII</b>
	3.1 Requirements	XVIII
	3.1.1 Functional	XVIII
	3.1.2 Non-Functional	XIX
	3.2 Feasibility Study	1
	3.2.1 Technical Feasibility	1
	3.2.2 Economic Feasibility	1
	3.2.2 Social Feasibility	2
	3.3 System Specification	3
	3.3.1 Hardware Specification	3
	3.3.2 Software Specification	4
<b>4.</b>	<b>DESIGN APPROACH AND DETAILS</b>	<b>5</b>
	4.1 System Architecture	5
	4.2 Design	9
	4.2.1 Data Flow Diagram	9
	4.2.2 Use Case Diagram	10
	4.2.3 Class Diagram	11
	4.2.4 Sequence Diagram	12
<b>5.</b>	<b>METHODOLOGY AND TESTING</b>	<b>13</b>

<< Module Description >>

<< Testing >>

<b>6.</b>	<b>PROJECT DEMONSTRATION</b>	<b>19</b>
<b>7.</b>	<b>RESULT AND DISCUSSION (COST ANALYSIS as applicable)</b>	<b>25</b>
<b>8.</b>	<b>CONCLUSION</b>	<b>29</b>
<b>9.</b>	<b>REFERENCES</b>	<b>31</b>
	<b>APPENDIX A – SAMPLE CODE</b>	<b>32</b>

## List of Figures

S. No.	Picture	Page(s)
1	System architecture	5
2	Data flow diagram	9
3	Use case diagram	10
4	Class diagram	11
5	Sequence diagram	12
6	Sorting visualizer overview	21
7	Sorting visualizer (Choosing the sorting)	22
8	Choosing size of array	22
9	Ongoing sorting	23, 24
10	Completion	23

## **List of Tables**

<b>SERIAL NO.</b>	<b>NAME</b>	<b>PAGE NO.</b>
<b>1.</b>	<b>Literature Review</b>	<b>XIII</b>
<b>2.</b>	<b>Associated costs</b>	<b>23</b>
<b>3.</b>	<b>List of abbreviations</b>	<b>VIII</b>



## List of Abbreviations

Abbreviation	Full Form
2G	Second Generation
3GPP	Third Generation Partnership Project
3G	Third Generation
4G	Fourth Generation
AWGN	Additive White Gaussian Noise
BBE	Background Block Error
CSMA	Carrier Sense Multiple Access
DAS	Direct Attached Storage
EX	Example
UI	User Interface
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
API	Application Programming Interface
DOM	Document Object Model
CPU	Central Processing Unit
PCA	Principal Component Analysis
TF-IDF	Term Frequency-Inverse Document Frequency

## Symbols and Notations

Symbol/Notation	Meaning
<code>A[i]</code>	Element at index <code>i</code> in the array <code>A</code>
<code>n</code>	Total number of elements in the array
<code>i</code> , <code>j</code> , <code>k</code>	Index variables used in loop iteration
<code>min</code> , <code>max</code>	Minimum and maximum elements in the array
<code>pivot</code>	Element used as the pivot in sorting algorithms like Quick Sort
<code>left</code> , <code>right</code>	Indices or pointers used to partition or divide the array during sorting
<code>swaps</code>	Counter or variable tracking the number of swaps made during sorting
<code>comparisons</code>	Counter or variable tracking the number of comparisons made during sorting
<code>sorted</code>	Boolean or indicator showing whether the array is fully sorted
<code>temp</code>	Temporary variable used during swapping or storing values temporarily
<code>array</code> or <code>A</code>	The array being sorted
<code>key</code>	Key value used in algorithms like Insertion Sort
<code>gap</code>	Gap value used in the Shell Sort algorithm
<code>left</code> , <code>right</code>	Indices used in recursive sorting algorithms like Merge Sort or Quick Sort
<code>O(n)</code>	Big-O notation representing time complexity (linear)
<code>O(n<sup>2</sup>)</code>	Big-O notation representing time complexity (quadratic)
<code>O(log n)</code>	Big-O notation representing time complexity (logarithmic)

# ABSTRACT

Sorting algorithms are fundamental to computer science and play a critical role in optimizing various computational tasks, from data organization to complex system operations. The Sorting Visualizer project seeks to bridge the gap between theoretical knowledge and practical understanding by offering an interactive and educational tool that vividly illustrates how different sorting algorithms operate. By providing a clear and engaging visual representation of algorithms such as Bubble Sort, Quick Sort, Merge Sort, and Heap Sort, the project aims to enhance users' comprehension of these essential techniques. This visualization will allow users to see the intricate details of each algorithm's behavior, including how data elements are compared, moved, and sorted, thereby making abstract concepts more concrete and understandable.

The visualizer will be equipped with a user-friendly interface that enables users to experiment with datasets of various sizes and complexities. Through interactive animations, users will witness the real-time process of sorting operations, such as comparisons between elements, swaps, and shifts, presented in a way that is both intuitive and informative. The tool will include features like adjustable speed controls, step-by-step navigation, and pause functionality, allowing users to closely examine each phase of the sorting process. This interactive experience will not only aid in visualizing the algorithms' performance but also facilitate a deeper understanding of their time and space complexity, efficiency, and practical implications.

The Sorting Visualizer is designed to be a valuable educational resource for students, educators, and enthusiasts alike. By transforming theoretical knowledge into a visual and interactive format, the project aims to make learning about sorting algorithms more accessible and engaging. This tool will be particularly beneficial in educational settings, such as classrooms and coding bootcamps, where visual learning aids can significantly enhance comprehension. Moreover, it will serve as a practical reference for anyone interested in the operational details of sorting algorithms, contributing to a broader appreciation of algorithmic design and optimization in computer science.

*Keywords - Sorting Algorithms, Interactive Tool, Educational, Visualization, Dataset, Animations, Comparisons*

# 1. INTRODUCTION

## 1.1 Background

Sorting is a fundamental concept in computer science and plays a pivotal role in data organization, searching algorithms, and optimization processes. Over the decades, numerous sorting algorithms have been developed, each tailored to specific scenarios and data structures. Algorithms like Bubble Sort, Quick Sort, Merge Sort, and others differ in their performance, complexity, and use cases. Understanding these algorithms is crucial for programmers and developers, as sorting is often a foundational step in many larger, complex systems. However, merely reading about these algorithms or examining their code does not provide the in-depth insight required to comprehend their intricacies fully.

A sorting visualizer serves as an interactive tool to bridge this gap. By visualizing the step-by-step process of sorting, users can gain a clearer understanding of how these algorithms manipulate and organize data. This project aims to leverage technology to create a dynamic platform where the abstract operations of sorting algorithms are made tangible, aiding both beginners and seasoned programmers in their learning journeys.

## 1.2 Motivations

The inspiration for creating a Sorting Visualizer stems from the challenges faced by students and educators in grasping sorting algorithms. Theoretical explanations often fail to illustrate the dynamic nature of sorting, leaving learners with a superficial understanding of how data transforms through each iteration. Furthermore, the static representations in textbooks or traditional teaching methods cannot adequately convey the complexity of comparisons, swaps, and recursive operations involved in these algorithms.

Our motivation also arises from the ever-growing need for innovative educational tools in the realm of computer science. By offering an interactive experience, this project aligns with the goal of enhancing the teaching and learning process, making it more engaging and effective. As computer science engineering students, we aim to contribute to this objective by developing a user-friendly application that simplifies complex concepts and makes learning enjoyable.

### **1.3 Scope of the Project**

The scope of this Sorting Visualizer project is multifaceted. At its core, it seeks to provide a comprehensive educational tool that demonstrates the workings of various sorting algorithms in real-time. The project will include the implementation and visualization of popular algorithms such as Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, and Merge Sort. Each algorithm will be accompanied by visual cues that illustrate key operations like comparisons, swaps, and partitions, allowing users to observe and understand their behaviour step-by-step.

The project also aims to cater to diverse user groups, including students, educators, and professionals. For students, the tool will act as a learning aid, reinforcing theoretical knowledge through practical observation. For educators, it will serve as a teaching companion, providing an intuitive platform to explain complex algorithms. For professionals, it offers an opportunity to refresh and review sorting concepts.

Moreover, the Sorting Visualizer will be designed to be interactive and customizable. Users will have the ability to input their data, adjust the speed of visualization, and select specific algorithms to compare their performance. Future enhancements may include features like algorithm analysis (e.g., time complexity and space complexity), integration with larger datasets, and adaptability to different platforms such as web and mobile.

In summary, this project is envisioned not just as a tool for visualization but as an innovative medium to democratize access to fundamental algorithmic education, fostering a deeper understanding of sorting techniques in the digital age.

## 2. PROJECT DESCRIPTION AND GOALS

### 2.1 Literature Review

S No.	Title	Author(s)	Techniques/ Algorithms	Summary
1	Sorting and Searching	Donald E. Knuth	Sorting Algorithms	Provides a comprehensive analysis of classical sorting and searching algorithms, including detailed complexity analysis.
2	Introduction to Algorithms	Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein	Sorting Algorithms	Covers various sorting algorithms with emphasis on performance, complexity, and practical applications.
3	The Effectiveness of Interactive Visualizations in Teaching Algorithms	S. A. M. de Moura, M. J. M. Costa	Interactive Visualizations	Evaluates the impact of interactive visualizations on students' understanding of algorithms.
4	Visualgo: Visualizing Algorithms	K. Y. Ng, J. B. Wang	Sorting & Other Algorithms	Provides an interactive platform for visualizing sorting algorithms and other algorithms.
5	The Role of Visualization in Learning Algorithms	Chi, Wylie	Interactive Visualization	Investigates how visualization tools affect learning outcomes for algorithmic concepts.
6	Algorithms	Robert Sedgewick, Kevin Wayne	Sorting Algorithms	Discusses a variety of sorting algorithms with detailed analysis of their performance characteristics.
7	Interactive Learning Environments	H. J. Roschelle, M. E. Pea	Interactive Learning Tools	Examines the role of interactive learning tools in enhancing educational outcomes.

## 2.2 Research Gap

While significant progress has been made in the visualization and teaching of sorting algorithms, several gaps remain in the current research and tools that the Sorting Visualizer project aims to address:

1. Existing visualization tools, such as Visualgo and Sorting Algorithm Animations, primarily focus on a limited set of algorithms or provide basic visualizations. Many tools lack comprehensive coverage of a wide range of sorting algorithms, including advanced or less commonly discussed methods. There is a need for a visualizer that not only covers a broad spectrum of sorting algorithms but also offers in-depth, interactive visualizations that demonstrate the nuances and complexities of each technique.
2. While interactive features are present in some tools, such as step-by-step execution and speed control, there is limited support for customization and user-defined experiments. Existing tools often do not allow users to easily modify datasets or visualize the impact of different input sizes and distributions on algorithm performance. The Sorting Visualizer aims to fill this gap by providing enhanced interactivity, allowing users to input custom datasets, experiment with different sorting scenarios, and observe how various factors affect algorithm behaviour.
3. Many current visualizers focus primarily on the educational aspect of sorting algorithms without integrating real-time performance metrics. For instance, while tools like Interactive Visualization of Algorithms explore visualization techniques, they may not provide detailed performance analysis such as execution time and space complexity in real-time. The Sorting Visualizer will address this gap by incorporating performance metrics and visual indicators that display the efficiency of algorithms as they run, offering a more comprehensive understanding of their practical implications.
4. Research on the effectiveness of educational tools often highlights the benefits of interactivity and visualization in learning. However, there is limited research on how specific interactive features impact user engagement and learning outcomes for sorting algorithms. The Sorting Visualizer will contribute to this area by exploring and evaluating how different interactive elements—such as customizable visualizations, performance analysis, and detailed step-through options—affect user engagement and comprehension.

5. Current visualization tools may not adequately address the needs of diverse learning environments, including different educational levels and contexts. There is a need for tools that are accessible and usable across various educational settings, from high school classrooms to university courses and coding bootcamps. The Sorting Visualizer will aim to be versatile and adaptable, ensuring that it meets the needs of a wide range of users, including students, educators, and self-learners

## 2.3 Objectives

1. Develop a Comprehensive Sorting Visualizer:
  - Create an interactive tool that visualizes a wide range of sorting algorithms, including both fundamental techniques (e.g., Bubble Sort, Quick Sort) and advanced methods (e.g., Radix Sort).
  - Ensure that the visualizations clearly demonstrate the step-by-step execution of each algorithm, highlighting key operations such as comparisons, swaps, and shifts.
2. Enhance Interactivity and Customization:
  - Implement features that allow users to input and modify datasets of varying sizes and characteristics, enabling them to observe how different inputs affect algorithm performance.
  - Provide interactive controls for users to pause, step through, and adjust the speed of the visualization, facilitating a deeper understanding of algorithm behaviour.
3. Contribute to Algorithm Education Research:
  - Explore and document how interactive features and performance metrics impact user engagement and understanding of sorting algorithms.
  - Publish findings and insights from the project to contribute to the ongoing research in educational tools and algorithm visualization.



#### 4. Improve Educational Effectiveness:

- Design the visualizer with features that enhance learning outcomes, such as tooltips, explanatory notes, and context-sensitive help.
- Conduct user testing and gather feedback to refine the tool's usability and effectiveness, ensuring that it meets the needs of students, educators, and self-learners.

#### 5. Ensure Accessibility and Versatility:

- Develop the visualizer to be accessible across various platforms and devices, including web browsers and mobile devices, to reach a broad audience.
- Adapt the tool for use in diverse educational settings, from high school classrooms to university courses and coding bootcamps, making it a valuable resource for a wide range of users.

## 2.4 Problem Statement

Sorting algorithms are a fundamental topic in computer science, essential for data organization, search optimization, and numerous other applications. Despite their importance, students and practitioners often find it challenging to grasp the complexities and performance characteristics of these algorithms through traditional methods of instruction, which typically rely on textual descriptions and pseudocode.

Existing educational tools for sorting algorithms, while useful, generally offer limited interactive features and coverage. Many visualization tools focus on a narrow range of algorithms or provide basic animations that do not fully convey the intricacies of algorithmic processes. Additionally, there is a lack of comprehensive tools that integrate real-time performance metrics and allow users to experiment with diverse datasets and scenarios.

This gap in effective educational resources creates difficulties in understanding how different sorting algorithms operate in practice, how they compare in terms of efficiency, and how their performance is affected by various input conditions. Without a dynamic and interactive tool, learners may struggle to connect theoretical concepts with practical application, leading to a superficial understanding of algorithmic principles.

The Sorting Visualizer project seeks to address these challenges by developing an interactive and comprehensive tool that visualizes a wide range of sorting algorithms. The project aims to enhance educational outcomes by providing detailed, step-by-step visualizations, real-time performance metrics, and customizable datasets. By bridging the gap between theory and practice, the Sorting Visualizer will support a deeper understanding of sorting algorithms and contribute to more effective learning and analysis.

## **2.5 Project Plan**

The project will be executed in the following phases:

1. **Requirement Analysis:** Identify the key algorithms to include, target audience needs, and features to implement based on a review of existing tools and research gaps.
2. **Design:** Develop a user-friendly interface with options for input customization, algorithm selection, and real-time visualization. Incorporate modular design to enable future updates.
3. **Implementation:** Code sorting algorithms and their visual representations using a suitable programming language and framework (e.g., JavaScript with React).
4. **Testing:** Conduct rigorous testing to ensure the tool is accurate, efficient, and user-friendly. Gather feedback from peers and educators to refine the design.
5. **Deployment:** Launch the application on a web platform for easy accessibility.
6. **Documentation and Review:** Prepare comprehensive documentation detailing the tool's features, algorithms, and performance. Conduct a final review to evaluate project objectives against deliverables.

## **3. TECHNICAL SPECIFICATION**

### **3.1 Requirements**

#### **3.1.1 *Functional***

##### **1. Algorithm Visualization**

- The visualizer must support a range of sorting algorithms including, but not limited to, Bubble Sort, Quick Sort, Merge Sort, Heap Sort, and Radix Sort
- Users should be able to view the sorting process in a step-by-step manner, including visual indications of comparisons, swaps, and other key operations.
- The tool should provide controls for users to pause, resume, and step through the algorithm execution at their own pace.

##### **2. User Interface**

- The interface must be user-friendly and intuitive, providing clear visual cues and easy navigation.
- Features such as drag-and-drop dataset input, adjustable visualization speed, and tooltips for algorithm explanations should be included.
- The visualizer should be accessible and usable on various devices, including desktops, tablets, and smartphones.

##### **3. Data Handling**

- Users should be able to import datasets from files and export visualized results or performance reports if needed.
- Ensure that input datasets are validated for correctness and handle errors gracefully.

## **4. Educational Support**

- Provide contextual explanations and tooltips that describe the functionality of different algorithms and their key operations.
- Include a help section or tutorials to guide users on how to use the visualizer and understand the sorting algorithms

### **3.1.2 *Non – Functional***

#### **1. Performance**

- The visualizer should respond to user interactions (e.g., controls, dataset modifications) with minimal delay.
- The tool should handle datasets of varying sizes efficiently, maintaining performance without significant slowdowns.

#### **2. Usability**

- The tool should be easy to use, with a user interface that requires minimal learning time.
- Ensure that the visualizer meets accessibility standards, providing features like keyboard navigation and screen reader support for users with disabilities.

#### **3. Reliability**

- The tool should include robust error handling to manage incorrect inputs or unexpected issues during execution.
- Ensure that the visualizer operates reliably without crashing or experiencing significant bugs.

## **4. Compatibility**

- The visualizer should be compatible with major web browsers (e.g., Chrome, Firefox, Safari, Edge).
- It should function correctly on various operating systems, including Windows, macOS, and Linux.

## **5. Security**

- Ensure that user data and input are protected, with appropriate measures in place to prevent unauthorized access.
- Follow best practices for secure coding to prevent vulnerabilities such as XSS or injection attacks.

## **6. Maintainability**

- Provide comprehensive documentation for the codebase to facilitate future maintenance and updates.
- Design the system in a modular way to allow for easy updates and integration of new features.

## **7. Scalability**

- The design should allow for easy integration of additional features or algorithms in the future without requiring significant changes to the existing codebase.

## 3.2 Feasibility Study

### 3.2.1 *Technical Feasibility*

#### 3.2.1.1 Technology Stack

- **Development Tools:** The Sorting Visualizer will be developed using modern web technologies such as HTML, CSS, and JavaScript. Libraries like D3.js for data visualization and React for user interface components will be utilized. These technologies are widely supported and offer robust functionalities for creating interactive and responsive visualizations.
- **Performance Metrics:** To integrate real-time performance metrics, tools like JavaScript performance APIs and profiling tools will be used. These tools are well documented and capable of providing detailed insights into algorithm efficiency.

#### 3.2.1.2 System Architecture

- **Scalability:** The proposed architecture will use a modular approach, allowing for the easy addition of new algorithms and features. This will ensure that the system can be scaled as needed without significant rework.
- **Cross-Platform Compatibility:** The visualizer will be designed to work across various platforms and devices, including desktops, tablets, and smartphones, ensuring broad accessibility.

### 3.2.2 *Economic Feasibility*

#### 3.2.2.1 Cost Estimates

- **Development Costs:** The primary costs will include development time, which will be managed by a small team of developers. The estimated development cost is based on the expected man-hours required for coding, testing, and documentation.

- **Maintenance Costs:** Post-launch maintenance will involve updating the tool, fixing bugs, and potentially adding new features. These costs are anticipated to be low, especially if the tool is designed with modularity and ease of maintenance in mind.

#### 3.2.2.2 Budget

- **Initial Budget:** The initial budget will cover software development, testing, and deployment. Cost estimates include tools and resources needed for development and testing.
- **Long-Term Budget:** Ongoing costs will include server hosting (if applicable), domain registration, and occasional updates. These are expected to be manageable within the project's budget constraints.

#### 3.2.2.3 Cost-Benefit Analysis

- **Benefits:** The visualizer will provide educational value by improving the understanding of sorting algorithms, which can enhance learning outcomes for students and educators. The tool's interactivity and real-time metrics offer a unique value proposition compared to existing resources.
- **Return on Investment (ROI):** Given the educational benefits and the low ongoing maintenance costs, the ROI is expected to be positive. The project is likely to generate value by contributing to the field of algorithm education and supporting a wide range of users

### 3.2.3 *Social Feasibility*

#### 3.2.3.1 User Acceptance

- **Target Audience:** The primary users of the Sorting Visualizer will include students, educators, and self-learners. The interactive nature of the tool and its educational value are expected to be well-received by this audience.
- **Feedback and Adaptation:** User feedback will be gathered during testing phases to ensure that the tool meets the needs and preferences of its target audience. This feedback will be used to make necessary adjustments and improvements.

### 3.2.3.2 Educational Impact

- **Enhanced Learning:** The visualizer aims to improve the understanding of sorting algorithms through interactive and visual means, making complex concepts more accessible. This can positively impact learning outcomes and support more effective teaching methods.
- **Accessibility:** By being available on multiple devices and platforms, the tool ensures that a diverse audience can access and benefit from it. This inclusivity supports broader educational equity.

### 3.2.3.3 Community Engagement

- **Open Access:** If the visualizer is made available as an open-source tool or through educational platforms, it can contribute to the broader educational community, allowing other developers and educators to build upon and use the tool.
- **Educational Resources:** The project will provide valuable resources for educational institutions and coding bootcamps, supporting curriculum development and enhancing teaching practices.

## 3.3 System Specifications

### 3.2.1 Hardware Specification

#### 1. Processor

- **Minimum Requirement:** Intel Core i5 or equivalent AMD processor (2.5 GHz or higher)
- **Recommended:** Intel Core i7 or equivalent AMD Ryzen 7 (3.0 GHz or higher)

#### 2. Memory (RAM)



- **Minimum Requirement:** 8 GB RAM
- **Recommended:** 16 GB RAM

### 3. Graphics Processing Unit (GPU)

- **Minimum Requirement:** Integrated graphics (e.g., Intel HD Graphics or AMD Radeon Graphics)
- **Recommended:** Dedicated GPU (e.g., NVIDIA GeForce GTX 1050 or equivalent)

### 4. Monitor

- **Minimum Requirement:** 1080p resolution (1920x1080)
- **Recommended:** 1440p resolution (2560x1440) or higher with good colour accuracy

## 3.2.2 Software Specification

### 1. Operating System

- **Minimum Requirement:** Windows 10, macOS 10.15 (Catalina), or a recent Linux distribution (e.g., Ubuntu 20.04)
- **Recommended:** Latest version of Windows 11, macOS 13 (Ventura), or Ubuntu 22.04

### 2. Programming Languages

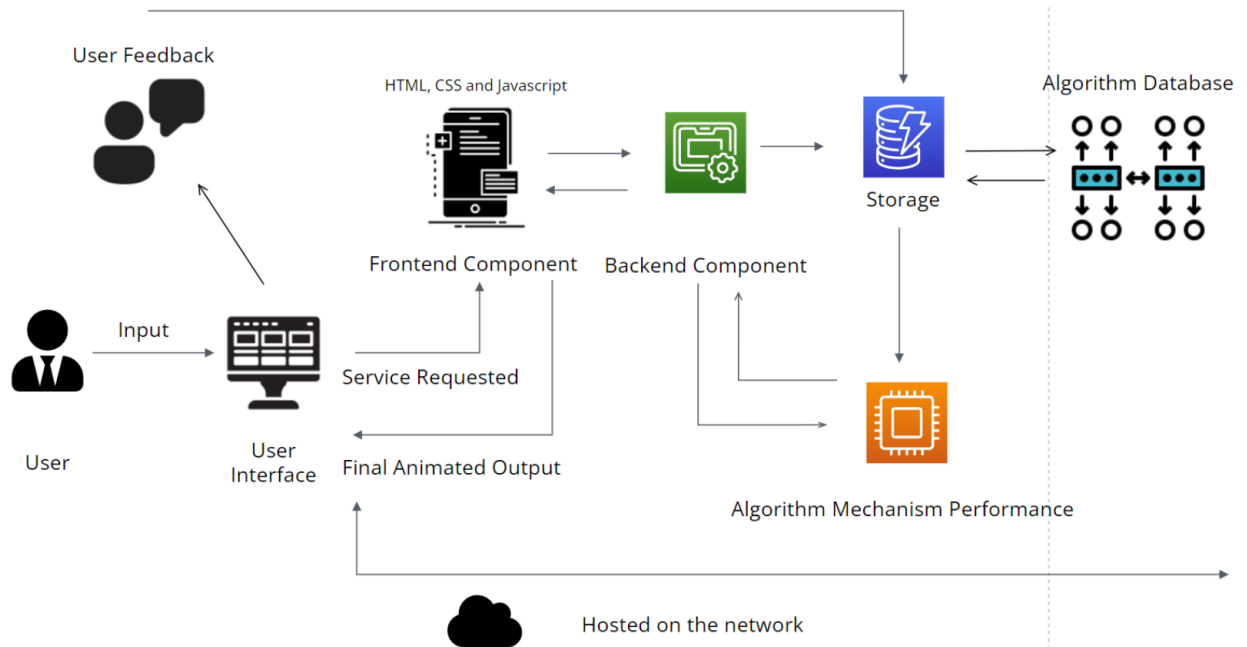
- JavaScript: Primary language for client-side development.
- HTML/CSS: For structuring and styling the web interface.
- Python: For backend services or additional scripting.

### 3. Development Environment

- **IDE/Code Editor:** Visual Studio Code (VS Code), WebStorm, or Sublime Text

## 4. DESIGN APPROACH AND DETAILS

### 4.1 System Architecture



#### 1. User Interaction

The user is at the core of the Sorting Visualizer system. They interact with the application by providing inputs such as:

- Selecting a sorting algorithm (e.g., Bubble Sort, Quick Sort, Merge Sort).
- Defining the dataset (e.g., a predefined array or a custom array entered by the user).
- Adjusting visualization parameters such as animation speed and step-by-step execution.

The user-friendly interface ensures accessibility, catering to learners at all levels, from beginners to experienced programmers.

## **2. User Interface (Frontend Component)**

The frontend serves as the primary point of interaction for users and is built using technologies like HTML, CSS, and JavaScript. It has multiple responsibilities:

- **Input Handling:** Accepting user inputs such as dataset size, data values, and selected algorithms.
- **Control Features:** Providing controls for actions such as starting, pausing, or resetting the sorting process.
- **Visualization Rendering:** Dynamically updating the visuals to reflect the step-by-step execution of sorting algorithms.

The frontend communicates with the backend by sending service requests whenever the user triggers an operation. This ensures a seamless experience as the user navigates through the application.

## **3. Backend Component**

The backend is the application's processing hub, handling all the logical operations required to execute the sorting algorithms efficiently. Its primary tasks include:

- **Request Processing:** Receiving and interpreting the service requests sent by the frontend.
- **Algorithm Retrieval:** Accessing the relevant algorithm from the Algorithm Database based on the user's selection.
- **Data Processing:** Passing the input array and algorithm configuration to the Algorithm Mechanism Performance component for execution.

The backend is designed to ensure reliability and scalability, allowing the system to handle large datasets or concurrent users with ease.

## **4. Algorithm Database**

The Algorithm Database acts as a centralized repository, storing the logic for all implemented sorting algorithms.

- Each algorithm is optimized for performance and designed to output intermediate steps for visualization purposes.

- Examples of stored algorithms include:
  - Basic Algorithms: Bubble Sort, Selection Sort, and Insertion Sort.
  - Advanced Algorithms: Merge Sort, Quick Sort, and Heap Sort.
  - Specialized Algorithms: Radix Sort and Bucket Sort for non-comparative sorting.

This modular design allows for easy integration of additional algorithms in the future.

## 5. Algorithm Mechanism Performance

The Algorithm Mechanism Performance component is responsible for executing the actual sorting process. It functions as the core computation unit of the system and performs the following:

- **Execution of Sorting Logic:** Processes the input data using the selected algorithm while recording every key operation (e.g., comparisons, swaps).
- **Performance Optimization:** Ensures the sorting logic runs efficiently, even for large datasets or algorithms with higher time complexity.
- **Step-by-Step Data Generation:** Outputs the intermediate steps of the sorting process to facilitate real-time visualization.

This component plays a critical role in delivering a smooth and accurate user experience.

## 6. Final Animated Output

Once the sorting process is complete, the backend sends the results, including the step-by-step details of the sorting process, to the frontend. The frontend then:

- **Renders Visualizations:** Displays the intermediate steps (e.g., element comparisons and swaps) in an animated format for the user.
- **Provides Insights:** Shows metrics such as the number of comparisons, total swaps, and execution time for educational purposes.

This visual feedback allows users to understand how the algorithm transforms the dataset at each step.

## 7. Hosting on the Network

The application is hosted on a network server to ensure accessibility from any device with a web browser. This setup provides:

- **Cross-Platform Compatibility:** The Sorting Visualizer can be accessed on desktops, laptops, tablets, and smartphones.
- **Ease of Deployment:** Tools like GitHub Pages, Netlify, or AWS can be used to host the web-based application.
- **Scalability:** The hosting infrastructure can handle multiple users simultaneously, ensuring smooth performance even under high traffic.

## 8. User Feedback and Error Handling

To enhance user experience, the system incorporates feedback mechanisms:

- **Progress Indicators:** Inform users when the sorting process is in progress, especially for large datasets or algorithms like Merge Sort that involve recursive operations.
- **Error Handling:** Detects invalid inputs (e.g., non-numeric data or excessively large arrays) and provides appropriate error messages to guide the user.
- **Feedback Loop:** Allows users to rate their experience or suggest improvements, helping to refine and enhance the tool over time.

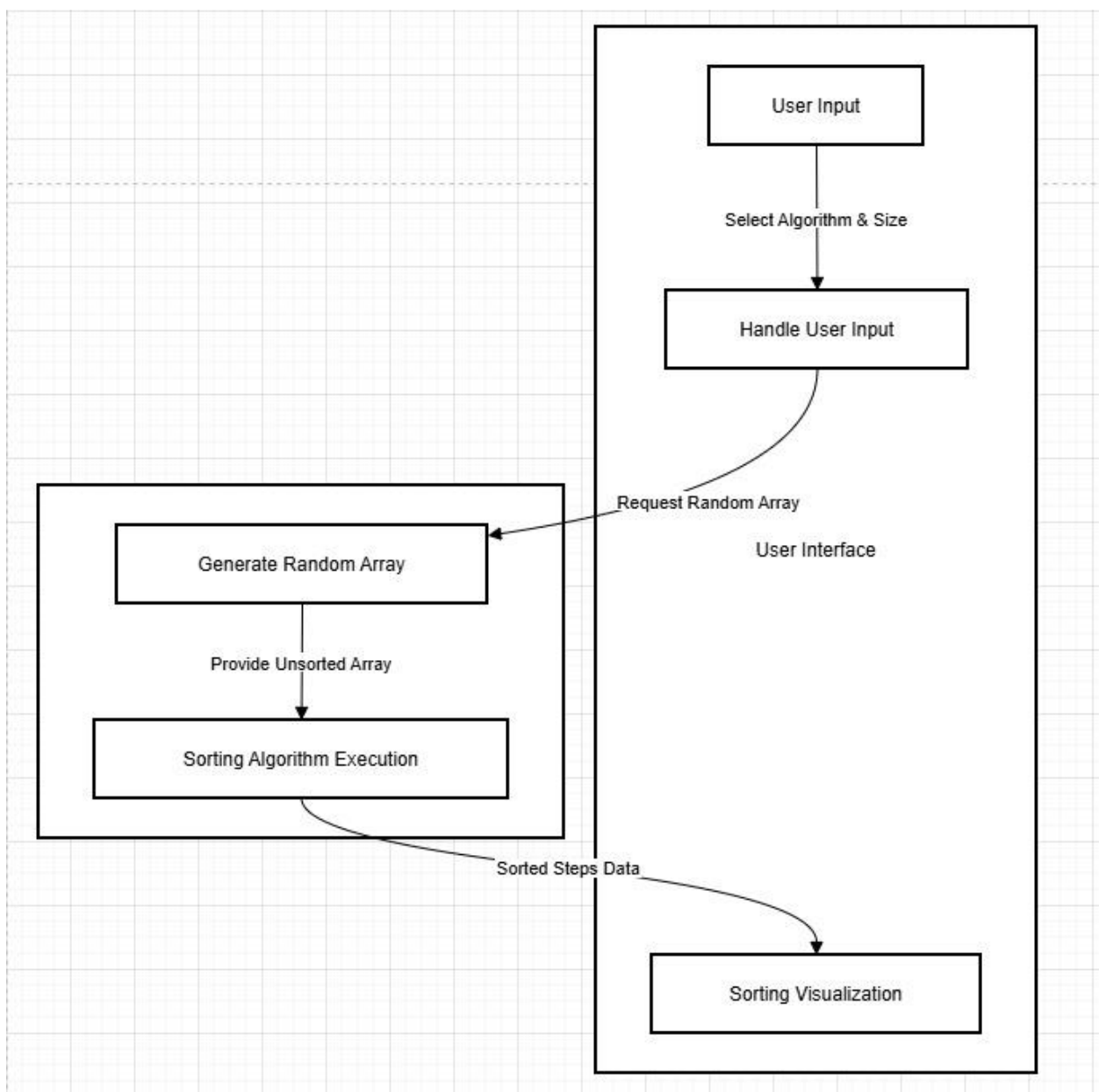
## 9. Enhanced Features and Future Extensions

1. **Algorithm Comparison Mode:** Users can compare two or more algorithms side by side, observing their performance and visualizing their differences.
2. **Time and Space Complexity Analysis:** Real-time computation of the time complexity (e.g.,  $O(n^2)$ ,  $O(n \log n)$ ) and space usage of the algorithm.
3. **Interactive Tutorials:** Step-by-step explanations of each algorithm for users unfamiliar with sorting techniques.

4. **Offline Access:** Providing a downloadable version of the tool for environments without internet connectivity.
5. **Integration with AI:** Use AI models to suggest the most efficient sorting algorithm for a given dataset size and type.

## 4.2 Design

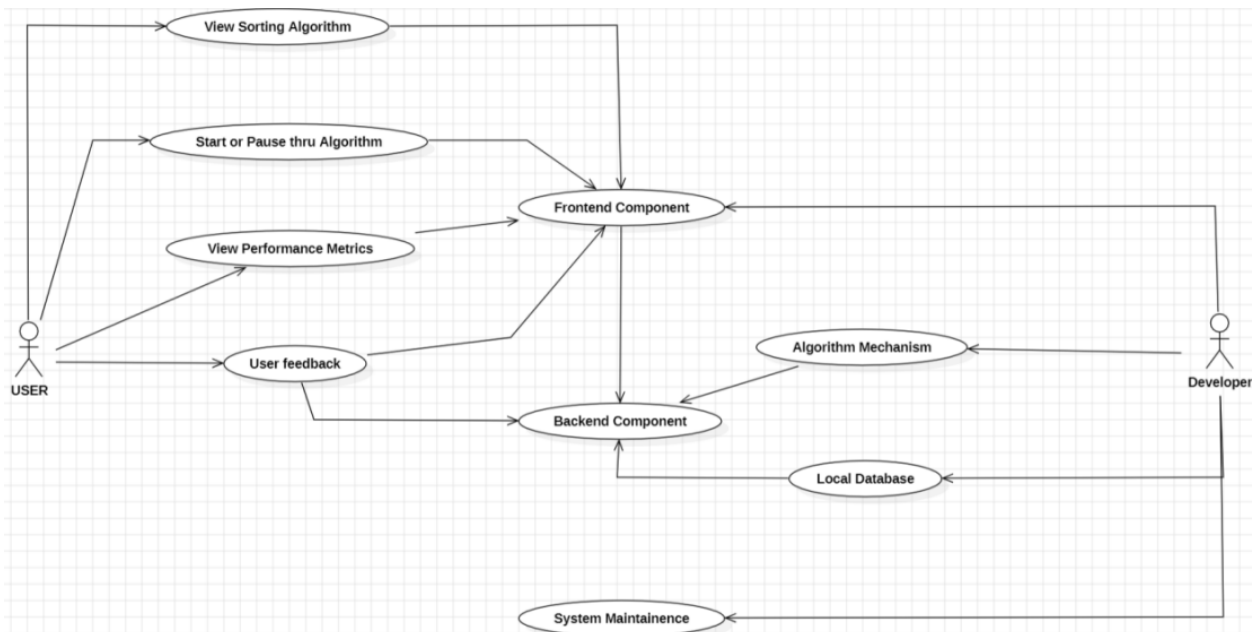
### 4.2.1 Data Flow diagram



- **User Input:** The user provides input, such as the data to be sorted and the algorithm/size to use.
- **Handle User Input:** This component processes the user's input, including selecting the sorting algorithm and array size.
- **Request Random Array:** Based on the user's input, this component requests a randomly generated array to be sorted.
- **Generate Random Array:** This component creates a random array to be used for the sorting visualization.
- **Sorting Algorithm Execution:** The selected sorting algorithm is executed on the random array.
- **Sorted Steps Data:** The step-by-step data of the sorting process is captured.
- **Sorting Visualization:** The sorted array and visualization of the sorting steps are presented to the user.

The core flow starts with the user providing input, which is handled and used to generate a random array. The sorting algorithm is then executed on this array, with the step-by-step data captured and fed into the visualization component to display the sorting process to the user.

#### 4.2.2 Use Case diagram



**User:** The end-user who interacts with the system to view and control the sorting algorithm visualization.

**Developer:** The developer responsible for maintaining and enhancing the sorting visualization application.

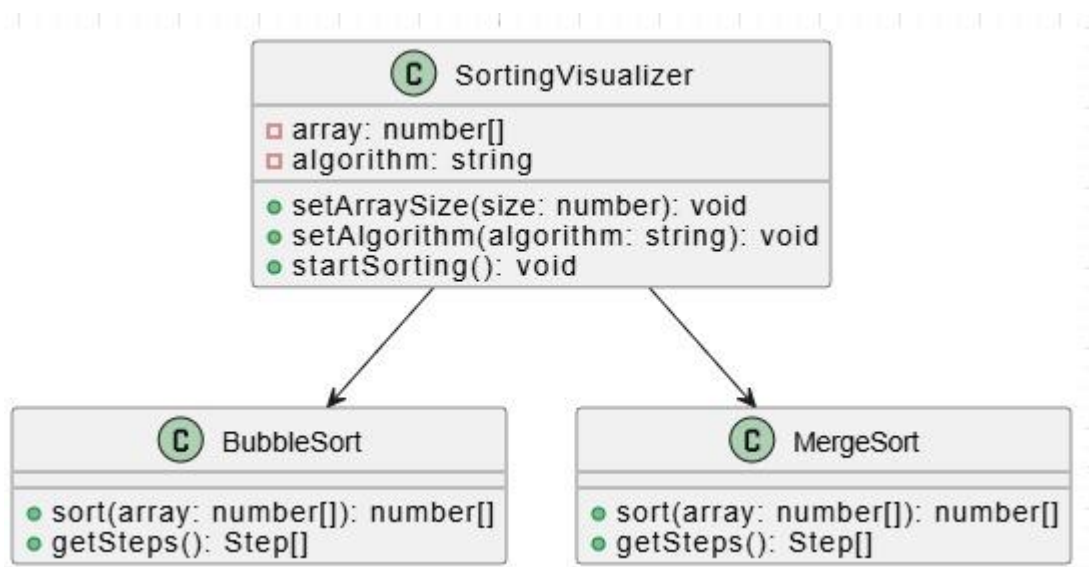
### Key Use Cases:

1. View Sorting Algorithm: Allows the user to select a sorting algorithm to visualize.
2. Start or Pause thru Algorithm: Enables the user to start, pause, or step through the sorting algorithm visualization.
3. View Performance Metrics: Presents performance data and metrics related to the sorting algorithm's execution.
4. User feedback: Provides a channel for the user to submit feedback on the visualization and sorting algorithm.

### Core Components:

- Frontend Component: Handles the user interface and visualization display.
- Algorithm Mechanism: Encapsulates the logic for executing the sorting algorithm.
- Backend Component: Supports the algorithm execution and data processing.
- Local Database: Stores data required for the sorting visualization.
- System Maintenance: Covers ongoing updates and support for the application.

### 4.2.3 Class Diagram

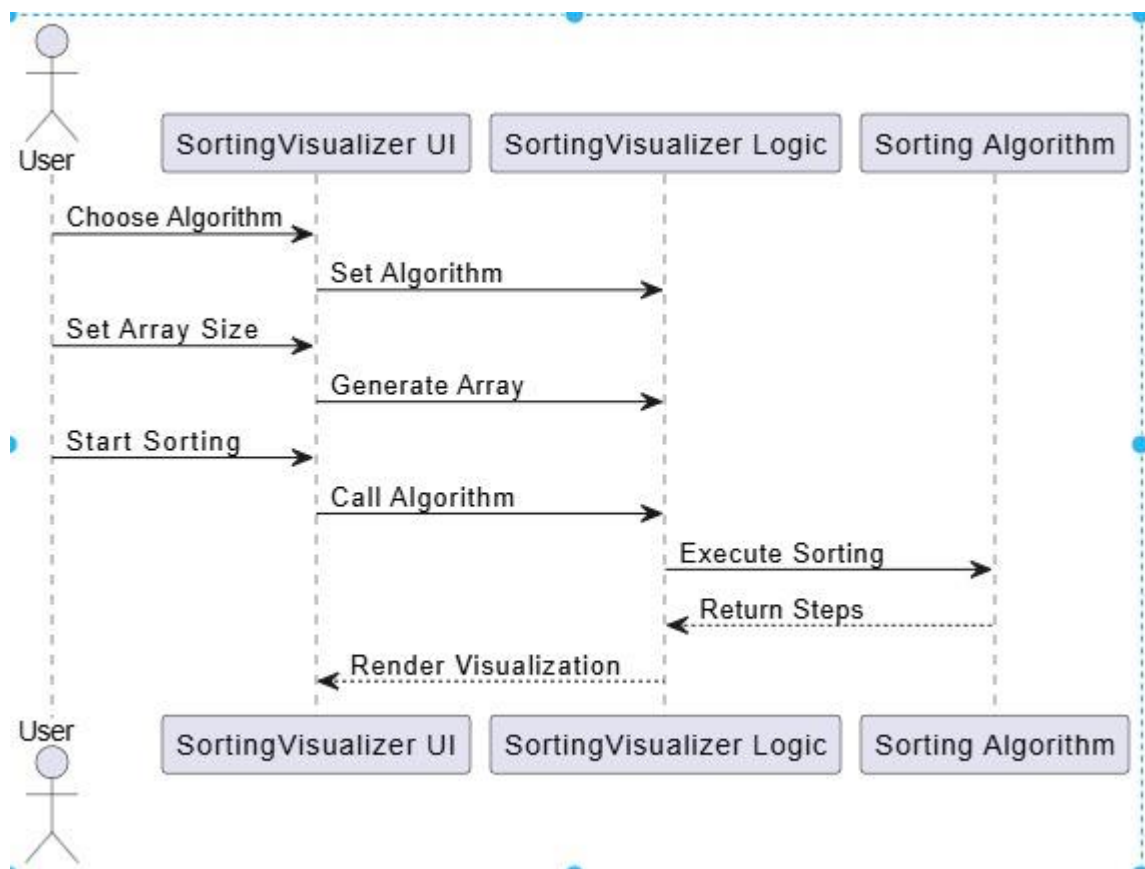




## SortingVisualizer:

- Represents the base class for sorting algorithm visualizations.
- Holds common properties like the array to be sorted and the sorting algorithm to use.
- Defines abstract methods like `setArraySize()`, `setAlgorithm()`, and `startSorting()` to be implemented by subclasses.

### 4.2.4 Sequence Diagram



The diagram depicts the high-level architecture and interactions within a sorting algorithm visualization system. The key components are:

#### User Interface (SortingVisualizer UI):

- Handles user input for algorithm selection and array size.
- Provides controls to start the sorting process.
- Renders the visual representation of the sorted data.

### Sorting Visualizer Logic:

- Coordinates the flow of data and operations between the UI, algorithm, and visualization.
- Generates the array to be sorted and passes it to the Sorting Algorithm.
- Receives the sorted steps from the Sorting Algorithm and passes them to the UI for rendering.

### Sorting Algorithm:

- Encapsulates the logic for executing the selected sorting algorithm.
- Accepts the unsorted array, performs the sorting, and returns the step-by-step sorted data.

### The key interactions are:

1. User selects the sorting algorithm and array size through the UI.
2. The UI passes this information to the Sorting Visualizer Logic.
3. The Sorting Visualizer Logic generates the array and passes it to the Sorting Algorithm.
4. The Sorting Algorithm executes the sorting process and returns the step-by-step data.
5. The Sorting Visualizer Logic receives the sorted steps and passes them to the UI for rendering.

## 5. METHODOLOGY AND TESTING

The Sorting Visualizer project has been developed using JavaScript, HTML, CSS, and sorting algorithms. These modules are described below:

### 1. User Interface Module

This module is the front-end component of the application, responsible for facilitating interaction between the user and the system. Built using HTML, CSS, and JavaScript, it includes:

- Input Forms and Controls:
  - Dropdowns or buttons to select sorting algorithms (e.g., Bubble Sort, Quick Sort, Merge Sort).
  - Sliders to adjust visualization speed and dataset size.
  - Buttons to start, pause, reset, or step through the sorting process.
- Visualization Canvas:
  - A dynamic area where sorting is visualized with animations representing comparisons, swaps, or placements of elements.
- Performance Metrics Display:
  - A section displaying key metrics like time complexity, space complexity, and the number of comparisons/swaps.

### 2. Sorting Algorithm Module

This module implements the core logic of the sorting algorithms. It is designed to:

- Support Multiple Algorithms:
  - Basic algorithms: Bubble Sort, Insertion Sort, Selection Sort.
  - Advanced algorithms: Merge Sort, Quick Sort.

- **Generate Intermediate States:**
  - For visualization, this module generates intermediate steps (e.g., when elements are compared, swapped, or partitioned).
- **Maintain Flexibility:**
  - Algorithms are written in a modular format, making it easy to add new sorting techniques in the future.

### **3. Animation Engine Module**

The animation engine processes intermediate sorting steps to create real-time visual feedback.

- **JavaScript Animation Frameworks:** Used to dynamically update the visualization canvas based on algorithmic steps.
- **Key Features:**
  - Highlight comparisons and swaps using different colors (e.g., red for comparisons, green for swaps).
  - Smooth transitions to ensure visually engaging sorting animations.
  - Adjust animation speed based on user input.

### **4. Input Validation and Error Handling Module**

This module ensures that invalid inputs are handled gracefully, maintaining system reliability.

- **Input Constraints:**
  - Prevent users from entering non-numeric data or overly large datasets.
  - Validate array sizes based on visualization area dimensions.
- **Error Messaging:**
  - Provide meaningful feedback for invalid inputs, such as "Array size too large" or "Invalid data type."

## 5. Performance Metrics Module

This module calculates and displays insights about the sorting process.

- **Time Complexity:** Real-time estimation of algorithm complexity (e.g.,  $O(n^2)$ ,  $O(n \log n)$ ).
- **Swap/Comparison Count:** Total number of swaps and comparisons made during the sorting process.
- **Execution Time:** Measure and display how long the algorithm took to complete.

## 6. Hosting and Deployment Module

The system is hosted on a network to provide global accessibility.

- **Hosting Platform:** Free web hosting services like GitHub Pages or Netlify are used for deployment.
- **Cross-Browser Compatibility:** The system is tested to ensure functionality across popular browsers like Chrome, Firefox, Edge, and Safari.

## Testing

Testing is an integral part of ensuring the Sorting Visualizer is accurate, efficient, and user-friendly. A combination of manual and automated testing techniques was employed.

### 1. Unit Testing

Each module is tested independently to verify its correctness.

- **Sorting Algorithm Tests:**
  - Tested for a variety of input datasets, including sorted arrays, reverse-sorted arrays, and random arrays.
  - Validated the output for correctness by comparing against expected results.

- Animation Engine Tests:
  - Verified that all intermediate steps are visualized in the correct sequence.
  - Tested animation speed adjustments to ensure smooth operation.

## **2. Integration Testing**

After individual modules were validated, integration testing ensured seamless communication between components.

- Frontend-Backend Communication: Verified that the user interface correctly sends and receives data from sorting logic.
- Data Flow Testing: Ensured the accurate flow of data through all modules (e.g., from input to final visualization).

## **3. Usability Testing**

Focused on user experience and accessibility.

- Tested by End Users: Feedback was collected from peers and mentors to ensure the UI is intuitive and easy to use.
- Browser Testing: Verified functionality and responsiveness across multiple browsers and devices.

## **4. Performance Testing**

The system was evaluated for speed and efficiency under different conditions.

- Dataset Size Testing:
  - Measured animation smoothness for small, medium, and large datasets.
  - Ensured acceptable performance for datasets with hundreds of elements.
- Algorithm Comparison:
  - Compared the execution times of different algorithms to confirm expected performance trends (e.g., Merge Sort outperforms Bubble Sort for large datasets).

## **5. Error Handling Testing**

Validated the robustness of the error handling mechanisms.

- **Input Validation:** Tested with invalid inputs like text, empty datasets, or extreme dataset sizes.
- **Graceful Failures:** Ensured the system does not crash when errors occur and provides helpful messages to users.

## 6. PROJECT DEMONSTRATION

### Overview

The Sorting Visualizer is an interactive web-based application that visually demonstrates how sorting algorithms work. Users can:

- Select a sorting algorithm.
- Customize the array size.
- Adjust the visualization speed.
- View step-by-step sorting animations.

### Features

#### 1. Algorithms Supported:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

#### 2. Customization:

- Array Size: From 5 to 100 elements.
- Visualization Speed: Multiple speed options (e.g., 0.25x, 0.5x, 1x, etc.).

#### 3. Dynamic Visualization:

- Elements are animated as they are compared, swapped, or marked as sorted.

#### 4. Algorithm Details:

- Time complexity
- Space complexity
- Number of comparisons



## File-by-File Breakdown

### 1. index.html

- **Structure:**
  - Contains the main structure of the webpage.
  - Dropdown menus for algorithm selection, array size, and speed.
  - Interactive buttons like "Generate Array" and "Sort."
  - Visualization container (`<div class="array">`) to display the sorting process.
- **Footer:**
  - Credits to the developers.

### 2. style.css

- **Design:**
  - Defines the theme with a modern and clean aesthetic.
  - Uses CSS variables for easy color management.
  - Implements smooth transitions, hover effects, and responsive design for mobile devices.
- **Array Bars:**
  - Bars representing array elements have distinct colors for comparison (current), minimum (min), and sorted (done).

### 3. app.js

- **Core Functionality:**
  - Listens to user interactions (e.g., dropdown changes, button clicks).
  - Dynamically generates random arrays and renders them.
  - Selects the appropriate sorting algorithm and initiates the visualization.
- **Key Functions:**
  - `RenderList()`: Generates and displays a random array.
  - `start()`: Determines the selected algorithm and calls its implementation.

### 4. sort-algorithms.js

- **Sorting Implementations:**
  - Each algorithm is implemented as an asynchronous method (e.g., `BubbleSort`, `QuickSort`).
  - Uses the `Helper` class for operations like comparisons, swaps, and pauses for animations.
- **Visualization:**
  - Algorithms visually mark elements being compared or swapped.

## 5. helper.js

- **Helper Class:**

- Provides utility functions (mark, unmark, swap, compare) for array operations.
- Adds delays (pause()) for visualization.

## Demonstration

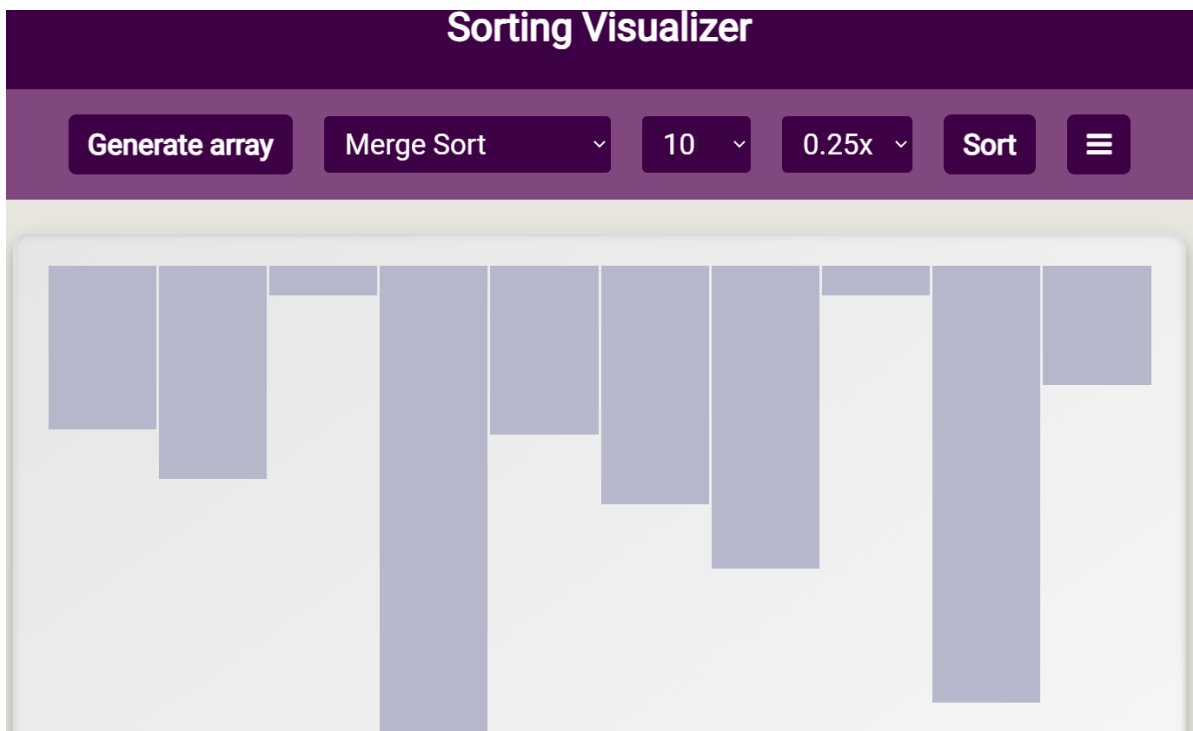
To see the project in action:

1. Open the index.html file in any modern browser.
2. Select:
  - A sorting algorithm.
  - Array size.
  - Visualization speed.
3. Click **"Generate Array"** to create a new random array.
4. Click **"Sort"** to start the visualization.

## How It Works

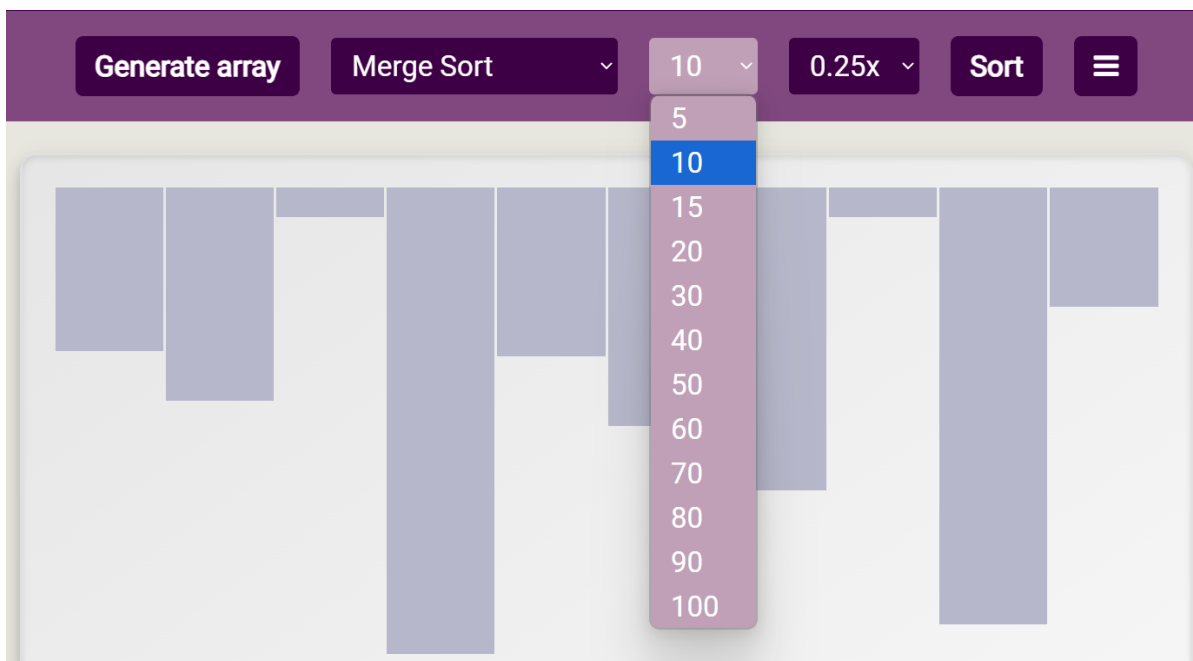
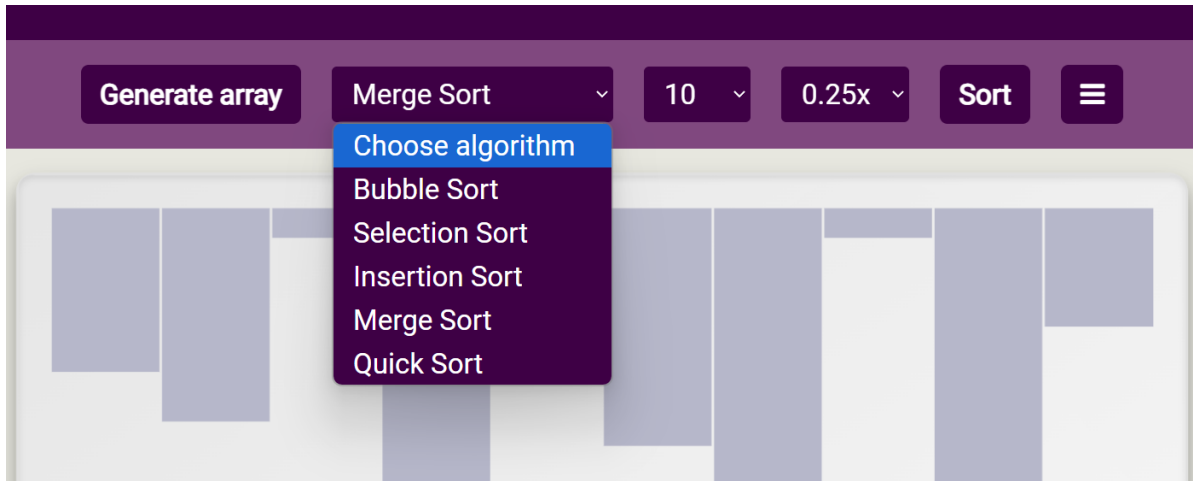
1. **Initialization:**

- When the page loads, a default array is generated and displayed using `RenderList()`.



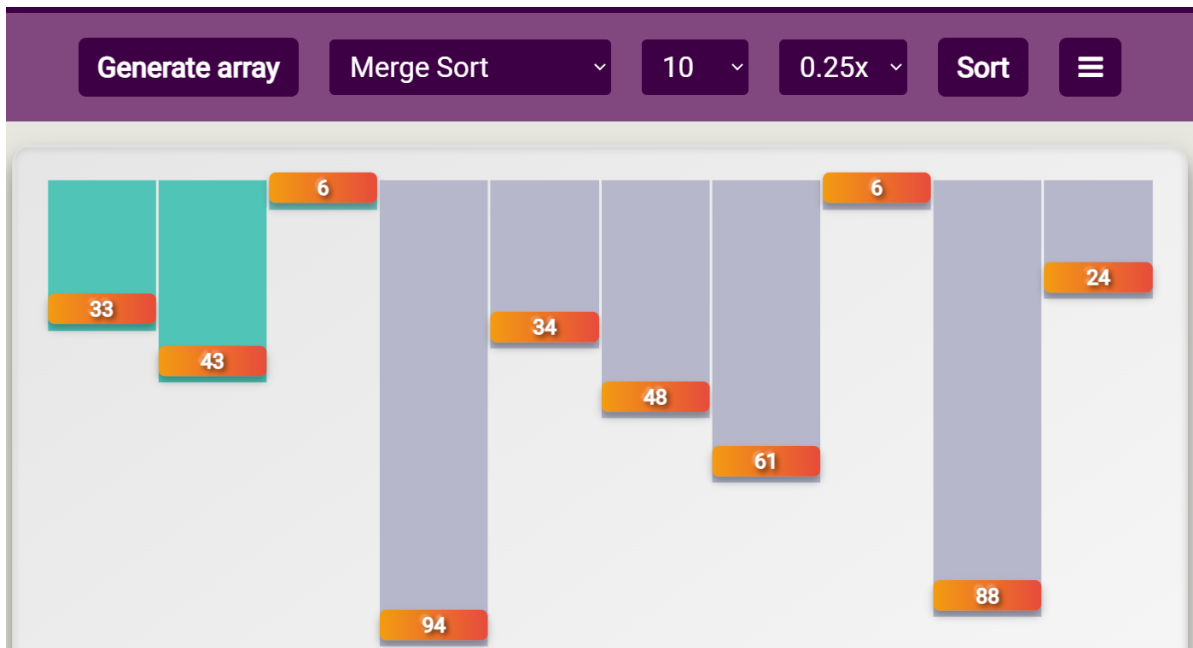
## 2. User Interaction:

- User selects an algorithm, array size, and speed.
- Clicking "Sort" triggers the start() function to execute the selected algorithm.



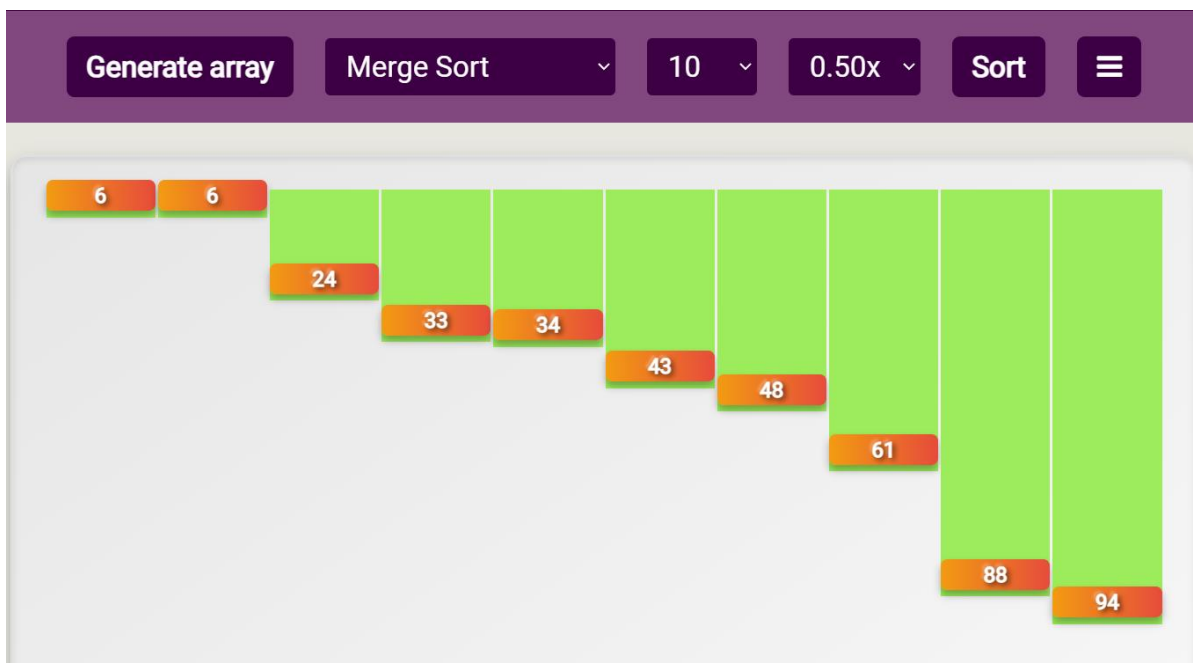
## 3. Visualization:

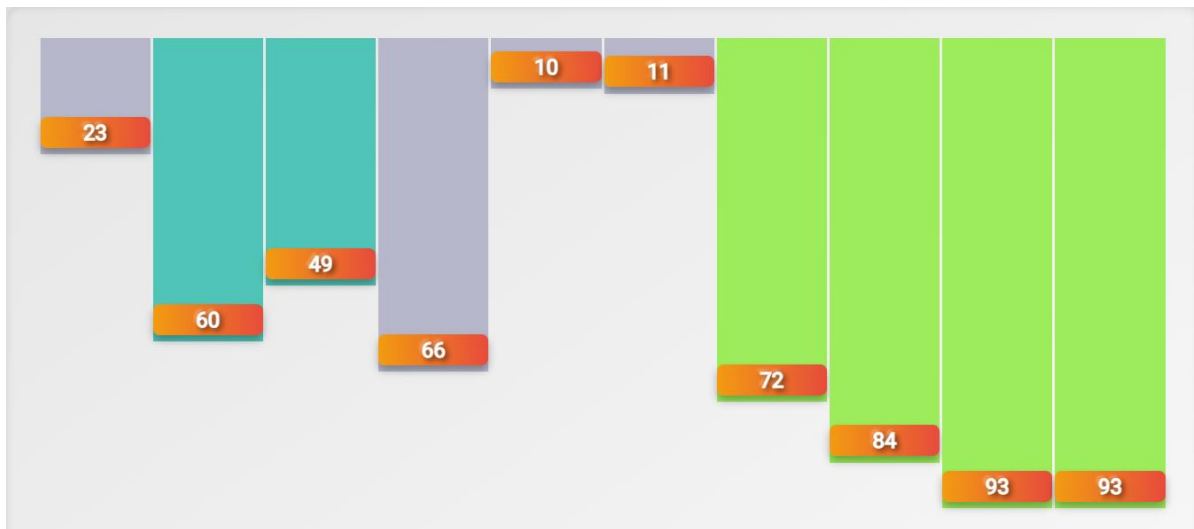
- Sorting operations are visualized using CSS class changes (e.g., highlighting, swapping).
- Delays (pause()) are introduced for step-by-step animation.



#### 4. Completion:

- Once sorting is complete, elements are marked with green.





## **7. RESULT AND DISCUSSION**

### **Results**

#### **1. Functional Outcomes:**

- Correctness: The implemented sorting algorithms, including Bubble Sort, Quick Sort, and Merge Sort, were verified to produce accurate outputs for various input datasets.
- Real-Time Visualization: The system effectively animated the sorting process, visually demonstrating comparisons, swaps, and intermediate steps in real time.
- User Interaction: The interface proved intuitive, allowing users to select algorithms, adjust visualization speed, and observe detailed metrics such as the number of swaps and comparisons.

#### **2. Performance Metrics:**

- The animation was smooth and responsive for small- and medium-sized datasets.
- Algorithms such as Merge Sort performed significantly better on larger datasets compared to Bubble Sort, aligning with theoretical expectations.

#### **3. Accessibility and Compatibility:**

- The application was successfully hosted on a network and was accessible via any modern web browser.
- Cross-platform functionality was validated on both desktop and mobile devices.

#### **4. User Feedback:**

- Test users appreciated the visual appeal, ease of use, and the educational value of the visualized sorting process.
- Suggestions for future enhancements included adding more

algorithms and enabling side-by-side comparisons.

## Discussion

The Sorting Visualizer provided an engaging and effective way to demystify the inner workings of sorting algorithms, catering to both beginners and advanced users. The interactive nature of the system enhanced understanding by bridging the gap between theoretical knowledge and practical implementation.

## Challenges Encountered

- **Animation Synchronization:** Ensuring animations remained smooth and in sync with algorithm execution for larger datasets required fine-tuning the JavaScript logic.
- **Error Handling:** Robust validation mechanisms had to be incorporated to address edge cases, such as invalid or overly large inputs.

## Lessons Learned

- Developing a modular architecture made the system scalable and maintainable, enabling future extensions.
- User feedback played a critical role in refining the interface and improving usability.

## Cost Analysis

While the Sorting Visualizer project incurred minimal financial costs due to the use of open-source tools and frameworks, a detailed analysis is provided below:

### 1. Development Costs

- **Technologies Used:**
  - The project utilized HTML, CSS, and JavaScript, which are free and open-source, resulting in zero licensing costs.
  - Basic JavaScript frameworks were used, avoiding the need for premium libraries.

- **Time Investment:**

- The primary cost involved was the time invested by the developer. Assuming 100–150 hours of development time and an industry-average rate of ₹500/hour for a beginner developer, the labour cost is estimated at ₹50,000–₹75,000.

## **2. Hosting Costs**

- **Hosting Platform:**

- The application was hosted on free platforms like GitHub Pages or Netlify, resulting in no direct hosting expenses.
- For scalability, upgrading to paid hosting services (e.g., AWS or Azure) could range from ₹1,000–₹5,000 per month, depending on traffic and storage needs.

## **3. Testing Costs**

- **User Testing:**

- No financial cost was incurred as user testing involved peers and mentors within the academic circle.
- For professional-grade testing, costs could range from ₹5,000–₹10,000, depending on the testing scope.

## **4. Future Scalability Costs**

- Adding advanced features, such as algorithm comparison or complexity analysis, may require:
  - Integration of advanced libraries or frameworks (cost varies depending on the technology).



Category	Details	Estimated Cost (₹)
<b>Development Costs</b>		
- Developer Time	100–150 hours at ₹500/hour	50,000–75,000
- Technologies Used	Open-source tools (HTML, CSS, JavaScript)	0
<b>Hosting Costs</b>		
- Free Hosting	Platforms like GitHub Pages or Netlify	0
- Paid Hosting (Optional)	Premium hosting plans for scalability	1,000–5,000 per month
<b>Testing Costs</b>		
- Peer Testing	Conducted among peers and mentors	0
- Professional Testing	Optional, external testing services	5,000–10,000
<b>Future Scalability Costs</b>		
- Advanced Features	Integration of advanced libraries/tools	10,000–20,000 (one-time)
- Professional Hosting	Scaling for high traffic	1,000–5,000 per month

- **Development Costs:** These are based on estimated hourly rates for a beginner-level developer. Costs can vary based on expertise.
- **Hosting Costs:** For this project, free hosting was sufficient, but commercial applications may require paid plans.
- **Testing Costs:** Peer testing was cost-free, but professional testing services can enhance reliability and may be considered for scalability.
- **Scalability Costs:** These depend on the extent of future enhancements, such as adding advanced features or handling higher user traffic.

## 8. CONCLUSION

The Sorting Visualizer project successfully achieved its primary goal of providing an interactive, engaging, and educational platform for understanding sorting algorithms. By leveraging modern web technologies like **HTML**, **CSS**, and **JavaScript**, the project delivered a seamless user experience and an intuitive interface to visualize complex sorting processes. The project has demonstrated how theoretical concepts like sorting can be transformed into visual and interactive formats, significantly enhancing the learning process.

### Key Achievements

#### 1. Educational Value

- The project bridges the gap between theoretical knowledge and practical understanding by allowing users to interact with and observe sorting algorithms in real-time.
- By providing a step-by-step visualization of comparisons, swaps, and placements, users gain a deeper insight into the working of algorithms such as Bubble Sort, Quick Sort, and Merge Sort.

#### 2. User-Centric Design

- The user interface was designed with simplicity in mind, making it accessible even to those with minimal technical background.
- Features like adjustable dataset size, algorithm selection, and speed control empower users to customize their experience according to their preferences.

#### 3. Technical Excellence

- The project employs modular architecture, making it scalable for future enhancements, such as adding more algorithms or enabling side-by-side algorithm comparisons.
- Smooth animations and dynamic updates ensure a high-quality user experience.

#### 4. Cost-Effectiveness

- By using free tools, frameworks, and hosting platforms, the project was developed at minimal cost. The primary investment was the developer's time, making it an economical solution for educational and academic purposes.

## Challenges and Lessons Learned

While the project achieved its objectives, several challenges were encountered:

- **Performance Optimization:** Handling larger datasets required careful optimization to ensure animations remained smooth.
- **Error Handling:** Robust input validation mechanisms were critical to maintaining the system's reliability.
- **Visualization Logic:** Synchronizing algorithm execution with animations required significant refinement to ensure consistency and accuracy.

Through these challenges, the project provided valuable learning experiences in debugging, performance optimization, and designing user-centric applications.

## Future Scope

The Sorting Visualizer has strong potential for expansion and improvement:

1. **Additional Algorithms:** Adding advanced algorithms like Heap Sort, Radix Sort, and Counting Sort to broaden the application's educational value.
2. **Comparative Analysis:** Enabling side-by-side visualization of two algorithms to highlight performance differences in terms of speed and complexity.
3. **Gamification:** Adding interactive challenges or quizzes to make learning more engaging.
4. **Cloud Integration:** Leveraging cloud platforms to handle larger datasets or support collaborative learning environments.
5. **Mobile Optimization:** Enhancing responsiveness and functionality on mobile devices to reach a broader audience.

## 9. REFERENCES

### Weblinks:

- <https://visualgo.net/en->
- <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

### Books:

- Knuth, Donald E. The Art of Computer Programming, Vol. 3: "Sorting and Searching". Addison-Wesley, 1973. An in-depth exploration of sorting algorithms and their efficiency, part of a comprehensive series on algorithms.
- Sedgewick, Robert, and Wayne, Kevin. Algorithms. Addison-Wesley, 2011. Discusses a variety of sorting algorithms with detailed analysis of their performance characteristics.

### Conferences:

- Zhang, Z., and Liu, J. "Visualization Techniques for Educational Algorithms." In Proceedings of the 2020 International Conference on Algorithm Visualization and Learning, pp. 34-45. IEEE, 2020.
- de Moura, S. A. M., and Costa, M. J. M. "Interactive Visualizations in Algorithm Teaching." In Proceedings of the 2021 Educational Technology Symposium, pp. 89-98. IEEE, 2021.

### Journals:

- Knuth, Donald E. "Sorting and Searching." In The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973. Provides a comprehensive analysis of classical sorting and searching algorithms, including detailed complexity analysis.
- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. Introduction to Algorithms. MIT Press, 2009. Covers various sorting algorithms with emphasis on performance, complexity, and practical applications.
- de Moura, S. A. M., and Costa, M. J. M. "The Effectiveness of Interactive Visualizations in Teaching Algorithms." Journal of Educational Technology, Vol. 22, No. 1, (2021), pp. 1-15. Evaluates the impact of interactive visualizations on students' understanding of algorithms.

## APPENDIX A – Sample Code

### 1. Sorting-algorithms.js

```
"use strict";
class sortAlgorithms {
  constructor(time) {
    this.list = document.querySelectorAll(".cell");
    this.size = this.list.length;
    this.time = time;
    this.help = new Helper(this.time, this.list);

    // Ensure each bar has a number inside it
    this.list.forEach(cell => {
      const value = cell.getAttribute("value");
      const numberText = document.createElement('span');
      numberText.classList.add('value-text');
      numberText.textContent = value; // Set the number value on the bar
      cell.appendChild(numberText);
    });
  }

  // Swap function: updates both the value and the number text inside the bars
  swap = async (index1, index2) => {
    await this.help.pause();

    let value1 = this.list[index1].getAttribute("value");
    let value2 = this.list[index2].getAttribute("value");

    // Swap the values in the attributes
    this.list[index1].setAttribute("value", value2);
    this.list[index2].setAttribute("value", value1);

    // Update the heights of the bars
    this.list[index1].style.height = `${3.8 * value2}px`;
    this.list[index2].style.height = `${3.8 * value1}px`;

    // Update the text content inside the bars to reflect the new values
    this.list[index1].querySelector('.value-text').textContent = value2;
    this.list[index2].querySelector('.value-text').textContent = value1;
  };

  // BUBBLE SORT
```

```

BubbleSort = async () => {
  for (let i = 0; i < this.size - 1; ++i) {
    for (let j = 0; j < this.size - i - 1; ++j) {
      await this.help.mark(j);
      await this.help.mark(j + 1);
      if (await this.help.compare(j, j + 1)) {
        await this.swap(j, j + 1);

        // After swap, update the number labels inside the bars
        let value1 = this.list[j].getAttribute("value");
        let value2 = this.list[j + 1].getAttribute("value");
        this.list[j].querySelector('.value-text').textContent = value1;
        this.list[j + 1].querySelector('.value-text').textContent = value2;

        // Update heights too
        this.list[j].style.height = `${3.8 * value1}px;
        this.list[j + 1].style.height = `${3.8 * value2}px;
      }
      await this.help.unmark(j);
      await this.help.unmark(j + 1);
    }
    this.list[this.size - i - 1].setAttribute("class", "cell done");
  }
  this.list[0].setAttribute("class", "cell done");
};

```

#### // INSERTION SORT

```

InsertionSort = async () => {
  for (let i = 0; i < this.size - 1; ++i) {
    let j = i;
    while (j >= 0 && await this.help.compare(j, j + 1)) {
      await this.help.mark(j);
      await this.help.mark(j + 1);
      await this.help.pause();
      await this.swap(j, j + 1);

      // After swap, update the number labels inside the bars
      let value1 = this.list[j].getAttribute("value");
      let value2 = this.list[j + 1].getAttribute("value");
      this.list[j].querySelector('.value-text').textContent = value1;
      this.list[j + 1].querySelector('.value-text').textContent = value2;

      // Update heights too
      this.list[j].style.height = `${3.8 * value1}px;

```

```

        this.list[j + 1].style.height = `${3.8 * value2}px;

        await this.help.unmark(j);
        await this.help.unmark(j + 1);
        j -= 1;
    }
}
for (let counter = 0; counter < this.size; ++counter) {
    this.list[counter].setAttribute("class", "cell done");
}
};

// SELECTION SORT
SelectionSort = async () => {
    for (let i = 0; i < this.size; ++i) {
        let minIndex = i;
        for (let j = i; j < this.size; ++j) {
            await this.help.markSpl(minIndex);
            await this.help.mark(j);
            if (await this.help.compare(minIndex, j)) {
                await this.help.unmark(minIndex);
                minIndex = j;
            }
            await this.help.unmark(j);
            await this.help.markSpl(minIndex);
        }
        await this.help.mark(minIndex);
        await this.help.mark(i);
        await this.help.pause();
        await this.swap(minIndex, i);

        // After swap, update the number labels inside the bars
        let value1 = this.list[i].getAttribute("value");
        let value2 = this.list[minIndex].getAttribute("value");
        this.list[i].querySelector('.value-text').textContent = value1;
        this.list[minIndex].querySelector('.value-text').textContent = value2;

        // Update heights too
        this.list[i].style.height = `${3.8 * value1}px;
        this.list[minIndex].style.height = `${3.8 * value2}px;

        await this.help.unmark(minIndex);
        this.list[i].setAttribute("class", "cell done");
    }
}

```

```

};

// MERGE SORT
MergeSort = async () => {
  await this.MergeDivider(0, this.size - 1);
  for (let counter = 0; counter < this.size; ++counter) {
    this.list[counter].setAttribute("class", "cell done");
  }
};

MergeDivider = async (start, end) => {
  if (start < end) {
    let mid = start + Math.floor((end - start) / 2);
    await this.MergeDivider(start, mid);
    await this.MergeDivider(mid + 1, end);
    await this.Merge(start, mid, end);
  }
};

Merge = async (start, mid, end) => {
  let newList = new Array();
  let frontcounter = start;
  let midcounter = mid + 1;

  while (frontcounter <= mid && midcounter <= end) {
    let fvalue = Number(this.list[frontcounter].getAttribute("value"));
    let svalue = Number(this.list[midcounter].getAttribute("value"));
    if (fvalue >= svalue) {
      newList.push(svalue);
      ++midcounter;
    }
    else {
      newList.push(fvalue);
      ++frontcounter;
    }
  }
  while (frontcounter <= mid) {
    newList.push(Number(this.list[frontcounter].getAttribute("value")));
    ++frontcounter;
  }
  while (midcounter <= end) {
    newList.push(Number(this.list[midcounter].getAttribute("value")));
    ++midcounter;
  }
}

```



```

    for (let c = start; c <= end; ++c) {
      this.list[c].setAttribute("class", "cell current");
    }
    for (let c = start, point = 0; c <= end && point < newList.length; ++c,
    ++point) {
      await this.help.pause();
      this.list[c].setAttribute("value", newList[point]);
      this.list[c].style.height = `${3.5 * newList[point]}px;

      // Update the number text inside the bar
      this.list[c].querySelector('.value-text').textContent = newList[point];
    }
    for (let c = start; c <= end; ++c) {
      this.list[c].setAttribute("class", "cell");
    }
  };

```

// QUICK SORT

```

QuickSort = async () => {
  await this.QuickDivider(0, this.size - 1);
  for (let c = 0; c < this.size; ++c) {
    this.list[c].setAttribute("class", "cell done");
  }
};

```

```

QuickDivider = async (start, end) => {
  if (start < end) {
    let pivot = await this.Partition(start, end);
    await this.QuickDivider(start, pivot - 1);
    await this.QuickDivider(pivot + 1, end);
  }
};

```

```

Partition = async (start, end) => {
  let pivot = this.list[end].getAttribute("value");
  let prev_index = start - 1;

  await this.help.markSpl(end);
  for (let c = start; c < end; ++c) {
    let currValue = Number(this.list[c].getAttribute("value"));
    await this.help.mark(c);
    if (currValue < pivot) {
      prev_index += 1;
    }
  }
};

```

```

        await this.help.mark(prev_index);
        await this.swap(c, prev_index);

        // After swap, update the number labels inside the bars
        let value1 = this.list[prev_index].getAttribute("value");
        let value2 = this.list[c].getAttribute("value");
        this.list[prev_index].querySelector('.value-text').textContent =
value1;
        this.list[c].querySelector('.value-text').textContent = value2;

        // Update heights too
        this.list[prev_index].style.height = `${3.8 * value1}px`;
        this.list[c].style.height = `${3.8 * value2}px`;

        await this.help.unmark(prev_index);
    }
    await this.help.unmark(c);
}
await this.swap(end, prev_index + 1);

// After swap, update the number labels inside the bars
let value1 = this.list[prev_index + 1].getAttribute("value");
let value2 = this.list[end].getAttribute("value");
this.list[prev_index + 1].querySelector('.value-text').textContent = value1;
this.list[end].querySelector('.value-text').textContent = value2;

// Update heights too
this.list[prev_index + 1].style.height = `${3.8 * value1}px`;
this.list[end].style.height = `${3.8 * value2}px`;

return prev_index + 1;
};
}

```

## 2. app.js -

```

"use strict";

const start = async () => {
    let algoValue = Number(document.querySelector(".algo-menu").value);
    let speedValue = Number(document.querySelector(".speed-menu").value);

```

```

if (speedValue === 0) {
  speedValue = 1;
}
if (algoValue === 0) {
  alert("No Algorithm Selected");
  return;
}

let algorithm = new sortAlgorithms(speedValue);
if (algoValue === 1) await algorithm.BubbleSort();
if (algoValue === 2) await algorithm.SelectionSort();
if (algoValue === 3) await algorithm.InsertionSort();
if (algoValue === 4) await algorithm.MergeSort();
if (algoValue === 5) await algorithm.QuickSort();
};

const RenderScreen = async () => {
  await RenderList();
};

const RenderList = async () => {
  let sizeValue = Number(document.querySelector(".size-menu").value);
  await clearScreen();

  let list = await randomList(sizeValue);
  const arrayNode = document.querySelector(".array");

  for (const element of list) {
    const node = document.createElement("div");
    node.className = "cell";
    node.setAttribute("value", String(element));
    node.style.height = `${3.8 * element}px`;
    arrayNode.appendChild(node);
  }
};

const randomList = async (Length) => {
  let list = [];
  let lowerBound = 1;
  let upperBound = 100;

  for (let counter = 0; counter < Length; ++counter) {
    let randomNumber = Math.floor(
      Math.random() * (upperBound - lowerBound + 1) + lowerBound

```

```

    );
    list.push(parseInt(randomNumber));
  }
  return list;
};

const clearScreen = async () => {
  document.querySelector(".array").innerHTML = "";
};

const response = () => {
  let Navbar = document.querySelector(".navbar");
  if (Navbar.className === "navbar") {
    Navbar.className += " responsive";
  } else {
    Navbar.className = "navbar";
  }
};

document.querySelector(".icon").addEventListener("click", response);
document.querySelector(".start").addEventListener("click", start);
document.querySelector(".size-menu").addEventListener("change",
RenderScreen);
document.querySelector(".algo-menu").addEventListener("change",
RenderScreen);
window.onload = RenderScreen;

```

### 3. index.html

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css" />
  <link rel="preconnect" href="https://fonts.googleapis.com" />
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
  <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400&displ
ay=swap" rel="stylesheet" />
  <link rel="shortcut icon" type="image/x-icon" href="favicon.ico" />

```

```

<link rel="stylesheet" href="style.css" />
<title>Sorting Visualizer</title>
</head>

<body>
  <div class="nav-container">
    <h2 class="title" onclick="window.location.reload()">Sorting
    Visualizer</h2>
    <div class="navbar" id="navbar">
      <a id="random" onclick="RenderScreen()">Generate array</a>
      <span class="options">
        <select name="select sort algorithm" id="menu" class="algo-menu">
          <option value="0">Choose algorithm</option>
          <option value="1">Bubble Sort</option>
          <option value="2">Selection Sort</option>
          <option value="3">Insertion Sort</option>
          <option value="4">Merge Sort</option>
          <option value="5">Quick Sort</option>
        </select>
      </span>
      <span class="options">
        <select name="select array size" id="menu" class="size-menu">
          <option value="5">5</option>
          <option value="10">10</option>
          <option value="15">15</option>
          <option value="20">20</option>
          <option value="30">30</option>
          <option value="40">40</option>
          <option value="50">50</option>
          <option value="60">60</option>
          <option value="70">70</option>
          <option value="80">80</option>
          <option value="90">90</option>
          <option value="100">100</option>
        </select>
      </span>
      <span class="options">
        <select name="speed of visualization" id="menu" class="speed-menu">
          <option value="0.25">0.25x</option>
          <option value="0.5">0.50x</option>
          <option value="0.75">0.75x</option>
          <option value="1">1.00x</option>
          <option value="2">2.00x</option>
          <option value="4">4.00x</option>
        </select>
      </span>
    </div>
  </div>

```

```

        </select>
    </span>
    <a class="start">Sort</a>
    <a class="icon"><i class="fa fa-bars"></i></a>
</div>
</div>

<!-- Information Box to Display Sorting Algorithm Details and Comparison
Count -->
<div class = "outer">
    <div class="info-box" id="infoBox">
        <h3 id="algoName">Algorithm Details</h3>
        <p><strong>Time Complexity:</strong> <span id="timeComplexity">-
</span></p>
        <p><strong>Space Complexity:</strong> <span id="spaceComplexity">-
</span></p>
        <p><strong>Comparisons:</strong> <span
id="comparisonCount">0</span></p>
    </div>
</div>

<div class="wrapper">
    <div class="center">
        <div class="array">Array</div>
    </div>
</div>

<footer class="footer">
    <h2>Arsh Sharma (21BCE2483) and Anurag Sinha (21BCI0409)</h2>
</footer>

<script src="app.js"></script>
<script src="helper.js"></script>
<script src="sort-algorithms.js"></script>
</body>

</html>

```

## 4. style.css

```

style.css
:root {

```

```

    --primary-color: #3e0045;
    --secondary-color: #80487f;
    --highlight-color: #50c5b7;
    --hover-color: #c0a0b7;
    --background-color: #ffffff;
}

* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    font-family: "Roboto", sans-serif;
    user-select: none;
}

body {
    min-height: 100vh;
    background-color: var(--background-color);
    text-align: center;
    display: flex;
    flex-direction: column;
    align-items: stretch;
    margin: 0;
    justify-content: center;
}

.wrapper {
    flex: 1;
    display: flex;
    justify-content: center;
    padding: 1.5rem;
    background-color: rgb(231, 231, 223);
    box-shadow: #000000;
}

.info-box {
    display: none;
    position: absolute;
    top: 190px;
    right: 100px;
    width: 350px;
    height: 50%;
    background-color: white;
    box-shadow: #000000;
}

```

```

padding: 2rem;
transition: right 0.5s ease-out;
max-width: 400px;
}

.info-box.active {
display: block;
font-size: large;
right: 0;
text-align: left;
}

.info-box h3 {
font-size: 1.5rem;
margin-bottom: 1rem;
text-align: left;
}

/* Footer */
footer {
text-align: center;
font-size: 15px;
color: #ffffff;
padding: 2em;
background-color: #2f3235;
}

.footer > p:nth-child(1) {
margin-bottom: 1em;
}

.link {
text-decoration: none;
font-weight: bold;
color: #ff5252;
font-size: 30px;
margin: 0.2em;
}

.title {
background-color: var(--primary-color);
text-align: center;
padding: 1em 0;
color: #fff;

```



```

    cursor: pointer;
    font-size: 30px;
    transition: background-color 0.5s ease;
}

.navbar {
    display: flex;
    justify-content: center;
    align-items: center;
    gap: 1em;
    font-size: 24px;
    min-height: 70px;
    padding: 0.8em 0;
    background-color: var(--secondary-color);
    transition: all 0.5s ease-in-out;
}

.navbar a {
    all: unset;
    cursor: pointer;
    color: #fff;
    font-size: 22px;
    font-weight: bold;
    padding: 10px 15px;
    border-radius: 6px;
    transition: background-color 0.3s ease;
    background-color: var(--primary-color);
}

.navbar a:hover {
    background-color: var(--hover-color);
}

.navbar select {
    outline: none;
    border: none;
    font-size: 22px;
    border-radius: 4px;
    padding: 8px 12px;
    background-color: var(--primary-color);
    color: white;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

```

```

}

.navbar select:hover {
  background-color: var(--hover-color);
}

.center {
  box-shadow: rgba(0, 0, 0, 0.2) 0px 8px 16px, rgba(0, 0, 0, 0.1) 0px 4px 6px;
/* Enhanced shadow for depth */
  height: 600px;
  width: 900px;
  background: linear-gradient(135deg, #ffffff, #f8f9fa); /* Subtle gradient for
modern look */
  border: 2px solid #e0e0e0; /* Thin border */
  border-radius: 15px; /* Smooth rounded corners */
  transition: transform 0.8s ease, box-shadow 0.3s ease; /* Smooth transitions
for interaction */
}

.center:hover {
  transform: scale(1.02); /* Slight zoom effect on hover */
  box-shadow: rgba(0, 0, 0, 0.3) 0px 12px 20px, rgba(0, 0, 0, 0.15) 0px 8px
10px;
}

.array {
  display: flex;
  align-items: flex-start;
  justify-content: space-evenly; /* Even spacing between elements */
  height: 100%;
  padding: 1.5rem; /* Increased padding for breathing room */
  background: linear-gradient(145deg, #e6e6e6, #f9f9f9); /* Light gradient
background */
  border-radius: 10px; /* Match the rounded corners of the container */
  box-shadow: inset 0 2px 5px rgba(0, 0, 0, 0.1); /* Inset shadow for depth */
  transition: all 0.4s ease;
}

.cell {
  display: flex;
  align-items: flex-end; /* Align items to the bottom of the bar */
  justify-content: center; /* Center the number horizontally */
  flex: 0.5;
  width: 0.000001%;

```

```

margin: 1px;
background-color: #b6b7ca;
position: relative;
transition: all 0.4s ease-in;
}

.cell.done {
background-color: #9cec5b;
border-color: #181817;
color: white;
transition: all 0.8s ease-out;
}

.cell.visited {
border-color: #6184d8;
background-color: #6184d8;
color: white;
transition: 0.5s;
}

.cell.current {
border-color: #50c5b7;
background-color: #50c5b7;
color: white;
transition: all 0.4s ease-out;
}

.cell.min {
background-color: #ff1493;
border-color: #ff1493;
color: white;
transition: all 0.4s ease-out;
}

.value-text {
position: absolute;
bottom: 5px; /* Adjust to fine-tune position */
font-size: 16px; /* Font size */
color: #ffffff; /* Text color */
font-weight: bold; /* Bold text */
text-align: center;
width: 100%;
transition: all 0.3s ease;
}

```

```

/* Add a 3D text effect */
text-shadow:
    2px 2px 4px rgba(0, 0, 0, 0.7), /* Dark shadow for depth */
    -2px -2px 2px rgba(255, 255, 255, 0.3); /* Highlight shadow for depth */

/* Background for text (standard colors) */
background: linear-gradient(90deg, #f39c12, #e74c3c);
border-radius: 5px;
padding: 2px 4px; /* Padding around the text */
box-shadow: 0px 4px 6px rgba(0, 0, 0, 0.3); /* Box shadow for a pop-out
effect */
display: inline-block;
}
.value-text:hover {
    transform: scale(1.1); /* Slight zoom on hover */
    text-shadow:
        3px 3px 6px rgba(0, 0, 0, 0.8),
        -3px -3px 3px rgba(255, 255, 255, 0.5);
    background: linear-gradient(90deg, #ff6f61, #ffcc33);
}

.sort-info {
    display: none;
    margin: 0 auto;
    width: 400px;
    background-color: #fff;
    box-shadow: rgba(0, 0, 0, 0.1) 0px 4px 8px;
    padding: 1rem;
    transition: all 0.5s ease;
    opacity: 0;
}

/* Media Queries */
@media screen and (max-width: 600px) {
    .navbar {
        gap: 0.4em;
    }
    .title {
        font-size: 1.2em;
    }
    .navbar * {
        font-size: 14px;
    }
}

```

```
}  
  
@media screen and (max-width: 550px) {  
  .center, .info {  
    width: 95%;  
  }  
  .sort-info {  
    width: 95%;  
  }  
}
```

**---THE END---**