A Design Analysis and Algorithm
on
“Sorting Visualizer”

By

Gaurav Jagtap [122B1B104]
Maheshwar Kadbane [122B1B116]
Harshal Ingale [122B1B097]

Under the Guidance of

Dr. Sarika Deokate

TY B.Tech (COMPUTER ENGINEERING)
2024-25 (Semester I)

DEPARTMENT OF COMPUTER ENGINEERING,
PCET’S PIMPRI CHINCHWAD COLLEGE OF ENGINEERING
Sector No. 26, Pradhikaran, Nigdi,
Pune - 411044

DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the project entitled **"Sorting Visualizer"** is successfully carried out as a mini project following students of PCET's Pimpri Chinchwad College of Engineering under the guidance of Dr. Sarika Deokate in the partial fulfillment of the requirements for the TY .Btech (Computer Engineering)

Gaurav Jagtap [122B1B104]
Maheshwar Kadbane [122B1B116]
Harshal Ingale [122B1B097]

Dr. Sarika Deokate
Project Guide

Acknowledgement

We would like to express our deepest gratitude to our guide, Dr. Sarika Deokate, for her constant encouragement, invaluable guidance, and unwavering support throughout the duration of this project. Her expertise and insights were key in shaping the course of our work, and her thoughtful suggestions, along with her detailed feedback at every stage, were crucial to our success. Dr. Sarika's guidance laid a solid foundation for our understanding, helping us navigate the complexities of sorting visualizers with clarity and precision.

We are also deeply thankful for her mentorship, which extended far beyond academic advice. Her passion for the subject in design and analysis, sparked our curiosity to delve into the real-world applications of these technologies. Her patience in answering our questions and her talent for breaking down intricate concepts greatly enhanced our learning and expanded our technical skills.

Her constructive feedback consistently pushed us to do better at every step, and her faith in our abilities encouraged us to overcome the challenges we encountered throughout this project. The success of this project is a direct reflection of her guidance, and we are sincerely grateful for her support and motivation.

Gaurav Jagtap [122B1B104]
Maheshwar Kadbane [122B1B116]
Harshal Ingale [122B1B097]

Title: Sorting Visualizer

Objectives:

1. Visualize and understand sorting algorithms.
2. Compare performance of multiple sorting techniques.
3. Demonstrate efficiency based on input size and order.
4. Understand time complexity and space complexity.
5. Bridge theoretical algorithm concepts with practical application.
6. Develop problem-solving and analytical skills.
7. Provide a platform for experimentation with sorting scenarios.

INDEX

INTRODUCTION

The "Sorting Visualizer" project is an interactive web-based tool designed to visually demonstrate various comparison-based sorting algorithms, such as Bubble Sort, Selection Sort, Insertion Sort, and Merge Sort. This application allows users to observe and understand the step-by-step functioning of these algorithms through a graphical representation. By providing a real-time visualization of how elements are sorted, this tool aids students in grasping complex sorting concepts more intuitively, especially beneficial for learners in computer science and engineering fields studying algorithm design and analysis.

Each algorithm's progress is highlighted dynamically, allowing users to compare their relative performance and characteristics, such as stability and time complexity. This interactive approach enhances the learning experience by bridging the gap between theoretical concepts and practical applications, making abstract algorithmic operations more tangible and engaging.

Overall, this project serves as a valuable educational tool that reinforces students' understanding of fundamental sorting algorithms and their behaviors under various conditions. It provides an insightful perspective on algorithm analysis by making complex computational processes visually accessible and engaging, catering to a deeper understanding of algorithmic efficiency and performance.

METHODOLOGY

1. **Project Setup and Environment**
    a. Used HTML, CSS, and JavaScript to create a static, web-based platform for visualization.
    b. Integrated Bootstrap for responsive design and styling, ensuring the interface is accessible across devices.

2. **Data Representation and Initialization**
    a. Created an array of randomly generated elements (bars) representing the dataset to be sorted.
    b. Provided sliders for users to control array size and sorting speed, offering flexibility in experimentation.
    c. Implemented a "Randomize" button to reset the array, allowing users to test algorithms on different data configurations.

3. **Algorithm Implementation**
    a. Developed sorting functions for each algorithm (Bubble Sort, Selection Sort, Insertion Sort, and Merge Sort) in separate JavaScript files.
    b. Each sorting function iterates through the array, performing swaps and updating the array state in real time.
    c. Added color-coding within each function to highlight the current and compared elements, enhancing the visual understanding of each step.

4. **Visualization and Animation**
    a. Used JavaScript to animate sorting steps, creating a pause between actions to allow users to follow along.
    b. Adjusted the animation speed based on user input from the speed slider, enhancing control over the pace of visualization.
    c. Displayed real-time sorting progress with transitions and color changes for better user engagement.

5. **User Interface and Controls**
    a. Implemented buttons to select sorting algorithms, with a "Sort" button to start the visualization.
    b. Added an alert system to notify users if no algorithm is selected before sorting begins.
    c. Incorporated a “Toggle Mode” button to allow users to switch between light and dark modes, enhancing usability in different lighting conditions.

6. **Testing and Optimization**
    a. Tested each sorting algorithm's visualizer for accuracy in sorting and fluidity of animation across various devices.
    b. Optimized code to reduce lag, especially for larger arrays and slower sorting algorithms.
    c. Conducted user testing to refine interface usability and improve educational effectiveness of the visualizations.

IMPLEMENTATION

//html code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
integrity="sha384-JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN
5t9UJ0Z" crossorigin="anonymous">
    <link
href="https://fonts.googleapis.com/css2?family=Catamaran:wght@100;200;300;400;500;600;700;
800;900&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="assets/style.css">
    <link rel="shortcut icon" href="assets/favicon.ico" type="image/x-icon">
    <title>Sorting Visualizer</title>
</head>
<body>
    <div class="container">
        <div id="array-container">
            <div id="array"></div>
            <h5 id="no-algo-warning" class="display-none">No Algorithm Selected!</h5>
        </div>
        <div id="algo-container">
            <button class="algo-btn">Bubble Sort</button>
            <button class="algo-btn">Selection Sort</button>
            <button class="algo-btn">Insertion Sort</button>
            <button class="algo-btn">Merge Sort</button>
            <button id="sort">Sort</button>
        </div>
        <div id="change-container">
            <button class="toggle-button" id="mode-toggle">Toggle Mode</button>
            <button id="randomize">Randomize</button>
            <div id="sliders">
                <div class="slider-container">
                    <label>Array Size</label>
```

```
            <input type="range" id="size-slider" class="slider" autocomplete="off">
          </div>
          <div class="slider-container">
            <label>Speed</label>
            <input type="range" class="slider" autocomplete="off">
          </div>
        </div>
      </div>
    </div>
    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJ
oZ+n" crossorigin="anonymous"></script>
    <script src="assets/script.js"></script>
    <script src="assets/bubble-sort.js"></script>
    <script src="assets/selection-sort.js"></script>
    <script src="assets/insertion-sort.js"></script>
    <script src="assets/merge_sort.js"></script>
</body>
</html>
```

//Bubble sort

```
async function bubbleSort() {
    var i, j;
    await sleep(delay);
    for(i = 0; i < size - 1; i++) {
        for(j = 0; j < size - i - 1; j++) {
            await sleep(delay);
            setColor(j, COMPARE);
            setColor(j + 1, COMPARE);
            await sleep(delay);
            if(arr[j] > arr[j + 1]) {
                swap(j, j + 1);
                await sleep(delay);
            }
            setColor(j, UNSORTED);
            setColor(j + 1, UNSORTED);
        }

        await sleep(delay);
```

```
        setColor(j, SORTED);
    }
    setColor(0, SORTED);
}
```

//Selection Sort

```
async function selectionSort() {
    var i, j, min_idx;
    for(i = 0; i < size - 1; i++) {
        await sleep(delay);
        min_idx = i;
        setColor(min_idx, SELECTED);
        for(j = i + 1; j < size; j++) {
            await sleep(delay);
            setColor(j, COMPARE);

            await sleep(delay);

            if(arr[j] < arr[min_idx]) {
                setColor(min_idx, UNSORTED);
                min_idx = j;
                setColor(min_idx, SELECTED);
                await sleep(delay);
            }
            else
                setColor(j, UNSORTED);
        }
        await sleep(delay);

        if(min_idx != i) {
            setColor(i, COMPARE);
            await sleep(delay);

            setColor(min_idx, COMPARE);
            setColor(i, SELECTED);
            swap(min_idx, i);
            await sleep(delay);
        }
```

```
        setColor(min_idx, UNSORTED);
        setColor(i, SORTED);
    }
    setColor(size - 1, SORTED);
}
```

//Insertion Sort

```
async function insertionSort() {
    var i, j, key;
    await sleep(delay);
    setColor(0, SELECTED);
    await sleep(delay);

    setColor(0, SORTED);

    for(i = 1; i < size; i++) {
        await sleep(delay);
        setColor(i, SELECTED);
        await sleep(delay);

        j = i - 1;
        key = arr[i];

        while(j >= 0 && arr[j] > key) {
            setColor(j, COMPARE);
            await sleep(delay);
            swap(j, j + 1);
            setColor(j, SELECTED);
            setColor(j + 1, COMPARE);
            await sleep(delay);

            setColor(j + 1, SORTED);
            await sleep(delay);

            j--;
        }
        setColor(j + 1, SORTED);
    }
}
```

//Merge Sort

```
async function mergeSort() {
    await mergeSortRecursive(0, size - 1);
    setSortedColors(0, size - 1);
}
async function mergeSortRecursive(low, high) {
    if (low < high) {
        var mid = Math.floor((low + high) / 2);

        await mergeSortRecursive(low, mid);
        await mergeSortRecursive(mid + 1, high);

        await merge(low, mid, high);
    }
}
async function merge(low, mid, high) {
    var n1 = mid - low + 1;
    var n2 = high - mid;

    var leftArray = new Array(n1);
    var rightArray = new Array(n2);

    for (var i = 0; i < n1; i++) {
        leftArray[i] = arr[low + i];
    }
    for (var j = 0; j < n2; j++) {
        rightArray[j] = arr[mid + 1 + j];
    }
    var i = 0, j = 0, k = low;

    setColor(low, COMPARE); // Set color for the first element
    await sleep(delay);

    while (i < n1 && j < n2) {
        setColor(mid + 1 + j, COMPARE);
        await sleep(delay);

        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            i++;
```

```
        } else {
            arr[k] = rightArray[j];
            j++;
        }
        setHeight(k, arr[k]);
        setColor(k, SORTED); // Set the color here to SORTED
        await sleep(delay);
        k++;
    }
    while (i < n1) {
        arr[k] = leftArray[i];
        setHeight(k, arr[k]);
        setColor(k, SORTED); // Set the color here to SORTED
        await sleep(delay);
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArray[j];
        setHeight(k, arr[k]);
        setColor(k, SORTED); // Set the color here to SORTED
        await sleep(delay);
        j++;
        k++;
    }
}
```

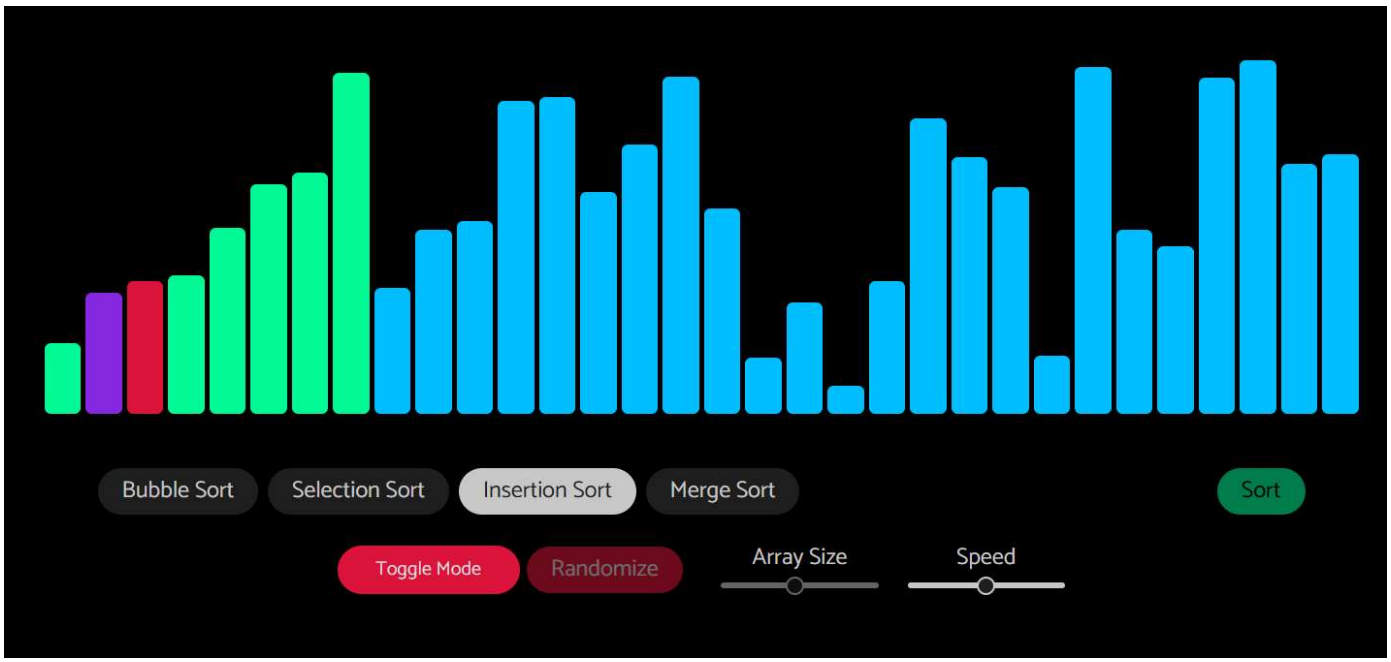
OUTPUT

Before Sorting:

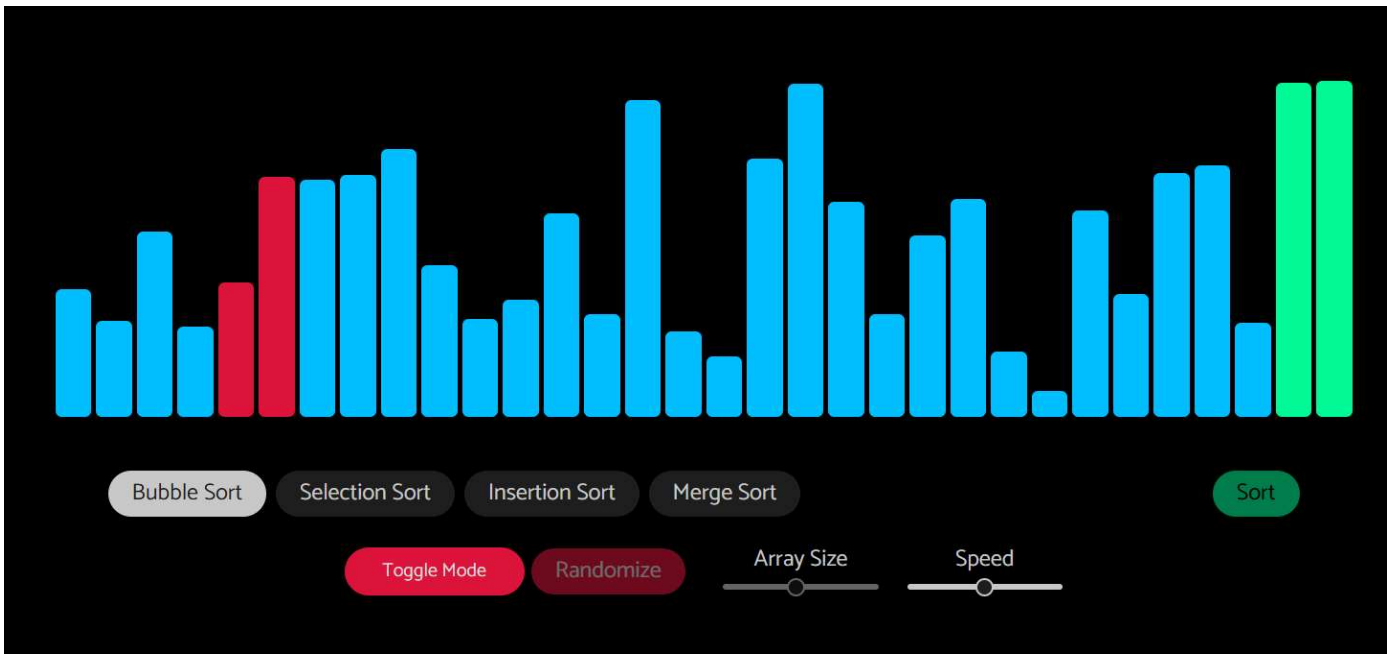
Merge Sort:

Selection Sort:

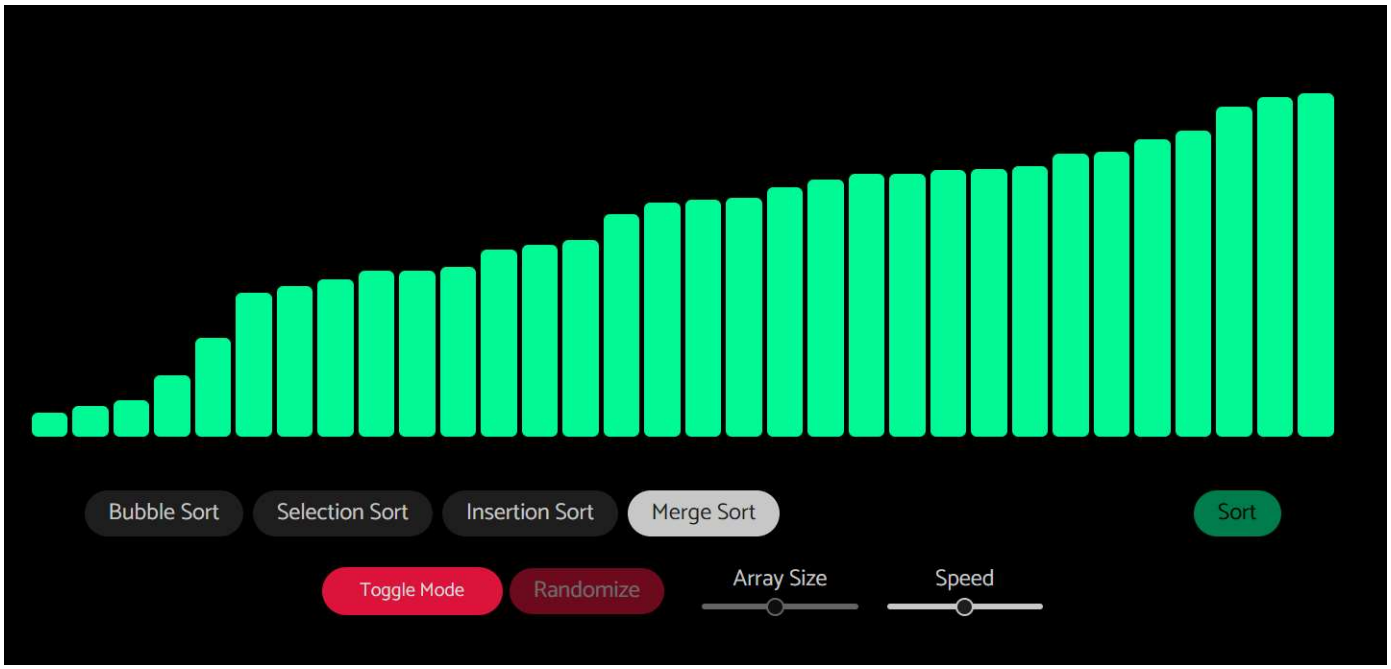
Insertion Sort:



Bubble Sort:



After Sorting :



CONCLUSION

1. Bubble Sort

- **Approach**: Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. It "bubbles" the largest element to the end of the array in each pass.
- **Time Complexity**:
  - Best case: O(n)O(n)O(n) (when the array is already sorted)
  - Average and Worst case: O(n2)O(n^2)O(n2)
- **Space Complexity**: O(1)O(1)O(1) (in-place sorting)
- **Stability**: Stable (preserves the order of equal elements)
- **Use Cases**: Simple to understand and implement but generally inefficient for large datasets. Suitable for educational purposes or very small arrays.

2. Selection Sort

- **Approach**: Selection Sort repeatedly selects the smallest (or largest) element from the unsorted portion of the array and places it at the start. It makes fewer swaps than Bubble Sort but still involves multiple comparisons.
- **Time Complexity**:
  - Best, Average, and Worst case: O(n2)O(n^2)O(n2)
- **Space Complexity**: O(1)O(1)O(1) (in-place sorting)
- **Stability**: Not stable (does not preserve the order of equal elements without modification)
- **Use Cases**: Useful when the number of swaps needs to be minimized, but not efficient for large datasets. Often used for teaching sorting concepts.

3. Insertion Sort

- **Approach**: Insertion Sort builds the sorted array one element at a time by inserting each element into its proper place in the already-sorted portion of the array.
- **Time Complexity**:
  - Best case: O(n)O(n)O(n) (for nearly sorted arrays)
  - Average and Worst case: O(n2)O(n^2)O(n2)
- **Space Complexity**: O(1)O(1)O(1) (in-place sorting)
- **Stability**: Stable (preserves the order of equal elements)
- **Use Cases**: Efficient for small arrays or nearly sorted datasets, making it suitable for applications where only a few elements are out of place, like maintaining a partially sorted list.

4. Merge Sort

- **Approach**: Merge Sort is a divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves.

- **Time Complexity**:
    - Best, Average, and Worst case: O(nlogn)O(n \log n)O(nlogn)
- **Space Complexity**: O(n)O(n)O(n) (requires additional space for merging)
- **Stability**: Stable (preserves the order of equal elements)
- **Use Cases**: Suitable for large datasets due to its O(nlogn)O(n \log n)O(nlogn) time complexity. Commonly used in applications where stability is important, such as sorting linked lists or large files.