CSE-5311

DESIGN & ANALYSIS OF ALGORITHMS

PROGRAMMING PROJECT REPORT

SORTING ALGORITHMS

SUBMITTED BY:

THANUJ KUMAR SHIVALINGAIAH
1002136007

TABLE OF CONTENTS

1. Introduction

The Sorting Algorithms Visualization project aims to provide a comprehensive and interactive platform for users to explore and understand the inner workings of various sorting algorithms. Sorting algorithms play a fundamental role in computer science and are essential for organizing and optimizing data. The goal of this project is to bridge the gap between theoretical understanding and practical observation by visually representing the sorting process.

In this era of increasing complexity in software development, having a clear understanding of how different algorithms perform is crucial. Visualization serves as a powerful tool to enhance learning and comprehension, especially in the context of algorithmic efficiency. By offering a visual representation of sorting algorithms, this project strives to make these fundamental concepts accessible to a wide audience, including students, educators, and anyone interested in the world of algorithms.

2. Project Overview

The Sorting Algorithms Visualization project is implemented using a combination of HTML, CSS, and JavaScript. The project provides an intuitive user interface that allows users to dynamically interact with and observe the sorting process. The primary focus is on implementing and comparing six different sorting algorithms:

Merge Sort: A divide-and-conquer algorithm that recursively divides the input array into smaller subarrays until they are trivially sorted, then merges them in a sorted manner.

Heap Sort: A comparison-based sorting algorithm that uses a binary heap data structure to build a max-heap and repeatedly extract the maximum element.

Quick Sort (Regular and 3 Medians): An efficient, in-place, and comparison-based sorting algorithm that partitions the array into smaller segments, sorting them independently.

Insertion Sort: A simple sorting algorithm that builds the final sorted array one element at a time.

Selection Sort: A simple sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted part of the array and swaps it with the first unsorted element.

Bubble Sort: A straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

The project's user interface is designed to be user-friendly, featuring controls for generating new arrays, adjusting the number of elements, selecting sorting algorithms, and setting the sorting speed. The visualization area provides a real-time display of the sorting process, offering users a hands-on experience to grasp the modulations of each algorithm.

By combining practical implementation with visual representation, this project aims to enhance understanding and appreciation for the efficiency and characteristics of different sorting algorithms.

Time complexity of the algorithms is given in the below table.

Algorithm	Time Complexity		
	Best	**Average**	**Worst**
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)

3. Main Components of the Algorithm

1. Bubble Sort:
   Function: bubbleSort()

   This function Iterates through the array multiple times, compares adjacent elements, and swaps them if they are in the wrong order. Gradually "bubbles" the largest unsorted element to its correct position.

   Code Snippet:

```
// Bubble Sort
async function bubbleSort() {
    let startTime = performance.now(); // Record the start time

    let childElements = upperDgm.children;
    console.log(childElements);
    newArrayBtn.disabled = true;
    disableButtons();

    for (let i = 0; i < nums.length; i++) {
        for (let j = 0; j < nums.length - i - 1; j++) {
            changeBarColor(childElements[j], childElements[j + 1], "red");
            await new Promise((resolve) => {
                setTimeout(resolve, delay);
            });
            if (nums[j] > nums[j + 1]) {
                let temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
                swap(childElements[j], childElements[j + 1]);
            }
            changeBarColor(childElements[j], childElements[j + 1], "yellow");
        }
        childElements[nums.length - i - 1].style.backgroundColor = "green";
    }
```

2. Selection Sort:

   Function: selectionSort()
   This function first divides the array into a sorted and an unsorted region, then finds the minimum element in the unsorted region and swaps it with the first element in the unsorted region. Then expands the sorted region until the entire array is sorted.

   Code Snippet:

```
// Selection Sort
async function selectionSort() {
    let startTime = performance.now(); // Record the start time

    let childElements = upperDgm.children;
    console.log(childElements);
    newArrayBtn.disabled = true;
    disableButtons();

    for (let i = 0; i < nums.length; i++) {
        for (let j = i + 1; j < nums.length; j++) {
            changeBarColor(childElements[i], childElements[j], "red");
            await new Promise((resolve) => { setTimeout(resolve, delay) });
            if (nums[i] > nums[j]) {
                let temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
                swap(childElements[i], childElements[j]);
            }
            changeBarColor(childElements[i], childElements[j], "yellow");
        }
        childElements[i].style.backgroundColor = "green";
    }

    let endTime = performance.now(); // Record the end time
    let totalTime = endTime - startTime; // Calculate the time difference

    console.log(`Time taken: ${totalTime} milliseconds`);

    newArrayBtn.disabled = false;

    // Display the time on your HTML page
    document.getElementById("timeDisplay").innerText = `Time taken: ${totalTime.toFixed(2)} milliseconds`;
    // After setting the time display text
    let timeDisplay = document.getElementById("timeDisplay");
    timeDisplay.innerText = `Time taken: ${totalTime.toFixed(2)} milliseconds`;
    timeDisplay.classList.add("blinking");

    // To stop the blinking after a certain duration (e.g., 5 seconds)
    setTimeout(() => {timeDisplay.classList.remove("blinking");}, 5000); // 5000 milliseconds (5 seconds)
}
```

3. Insertion Sort:

Function: insertionSort()
This function divides the array into a sorted and an unsorted region. Takes elements from the unsorted region and inserts them into their correct position in the sorted region. It repeatedly shifts elements to make space for the next element to be inserted.

Code Snippet:

```
// Insertion Sort
async function insertionSort() {
    let startTime = performance.now(); // Record the start time

    let childElements = upperDgm.children;
    console.log(childElements);
    newArrayBtn.disabled = true;
    disableButtons();
    childElements[0].style.backgroundColor = "green";

    for (let i = 1; i < nums.length; i++) {
        let temp = nums[i];
        childElements[i].style.backgroundColor = "red";
        let tempHeight = childElements[i].style.height;
        let j = i - 1;

        while (j >= 0 && nums[j] > temp) {
            childElements[j].style.backgroundColor = "red";
            await new Promise((resolve) => { setTimeout(resolve, delay) });
            nums[j + 1] = nums[j];
            let elementHeight = getComputedStyle(childElements[j]).getPropertyValue("height");
            childElements[j + 1].style.height = elementHeight;
            childElements[j].style.backgroundColor = "green";
            j--;
        }

        nums[j + 1] = temp;
        childElements[j + 1].style.height = tempHeight;
        childElements[i].style.backgroundColor = "green";
    }

    let endTime = performance.now(); // Record the end time
    let totalTime = endTime - startTime; // Calculate the time difference

    console.log(`Time taken: ${totalTime} milliseconds`);

    newArrayBtn.disabled = false;

    // Display the time on your HTML page
    document.getElementById("timeDisplay").innerText = `Time taken: ${totalTime.toFixed(2)} milliseconds`;
    // After setting the time display text
    let timeDisplay = document.getElementById("timeDisplay");
    timeDisplay.innerText = `Time taken: ${totalTime.toFixed(2)} milliseconds`;
    timeDisplay.classList.add("blinking");

    // To stop the blinking after a certain duration (e.g., 5 seconds)
    setTimeout(() => {timeDisplay.classList.remove("blinking");}, 5000); // 5000 milliseconds (5 seconds)
}
```

4. Merge sort:

   Functions: mergeSort() and initiateMerge()

   Divides the array into smaller sub-arrays recursively until each sub-array has only one element. Then merges the sorted sub-arrays to create larger sorted sub-arrays until the entire array is sorted.

   Code Snippet:

```
// Merge Sort
async function mergeArray(childElements, l, mid, r) {
    for (let a = l; a <= mid; a++) {
        await new Promise((resolve) => { setTimeout(resolve, delay) });
        childElements[a].style.backgroundColor = "orange";
    }
    for (let a = mid + 1; a <= r; a++) {
        await new Promise((resolve) => { setTimeout(resolve, delay) });
        childElements[a].style.backgroundColor = "blue";
    }
    let i = l, j = mid + 1, k = l;
    let tempArray = [];
    while (i <= mid && j <= r) {
        await new Promise((resolve) => { setTimeout(resolve, delay) });
        if (nums[i] < nums[j]) {
            tempArray[k] = nums[i];
            childElements[k].style.height = `${nums[i]}px`;
            i++;
        } else {
            tempArray[k] = nums[j];
            childElements[k].style.height = `${nums[j]}px`;
            j++;
        }
        k++;
    }
    while (i <= mid) {
        await new Promise((resolve) => { setTimeout(resolve, delay) });
        tempArray[k] = nums[i];
        childElements[k].style.height = `${nums[i]}px`;
        k++;
        i++;
    }
    while (j <= r) {
        await new Promise((resolve) => { setTimeout(resolve, delay) });
        tempArray[k] = nums[j];
        childElements[k].style.height = `${nums[j]}px`;
        k++;
        j++;
    }
    for (let k = l; k <= r; k++) {
        nums[k] = tempArray[k];
    }
}
```

```
async function initiateMerge() {
    let startTime = performance.now(); // Record the start time

    let childElements = upperDgm.children;
    console.log(childElements);
    newArrayBtn.disabled = true;
    disableButtons();
    await mergeSort(nums, childElements, 0, nums.length - 1);
    console.log(nums);
    newArrayBtn.disabled = false;

    let endTime = performance.now(); // Record the end time
    let totalTime = endTime - startTime; // Calculate the time difference

    console.log(`Time taken: ${totalTime} milliseconds`);
```

5. Heapsort:

   Functions: heapSort() and heapify()

   Builds a max heap, a binary tree where the value of each node is greater than or equal to the values of its children. Repeatedly extracts the maximum element from the heap and places it at the end of the array.

   Code Snippet:

```
// Heap Sort
async function heapify(nums, childElements, n, i) {
    let largest = i;
    let left = 2 * i + 1;
    let right = 2 * i + 2;

    if (left < n && nums[left] > nums[largest]) {
        largest = left;
    }

    if (right < n && nums[right] > nums[largest]) {
        largest = right;
    }

    if (largest !== i) {
        await new Promise((resolve) => setTimeout(resolve, delay));

        console.log(`Swapping ${nums[i]} at index ${i} with ${nums[largest]} at index ${largest}`);
        swap(childElements[i], childElements[largest]);

        let temp = nums[i];
        nums[i] = nums[largest];
        nums[largest] = temp;

        await heapify(nums, childElements, n, largest);
    }
}
```

```
async function heapSort(nums, childElements) {
    let n = nums.length;

    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
        await heapify(nums, childElements, n, i);
    }

    for (let i = n - 1; i > 0; i--) {
        await new Promise((resolve) => setTimeout(resolve, delay));

        console.log(`Swapping ${nums[0]} at index 0 with ${nums[i]} at index ${i}`);
        swap(childElements[0], childElements[i]);

        let temp = nums[0];
        nums[0] = nums[i];
        nums[i] = temp;

        await heapify(nums, childElements, i, 0);
        childElements[nums.length - i].style.backgroundColor = "green";
    }
}
```

6. Quicksort:

   Functions: quickSort() and partition()

   Chooses a pivot element and partitions the array into two halves such that elements less than the pivot are on the left, and elements greater are on the right.
   Recursively applies the same process to the left and right sub-arrays.

   Code Snippet:

```
// Quick Sort
async function partition(nums, childElements, low, high) {
    let pivot = nums[high];
    let i = low - 1;

    for (let j = low; j < high; j++) {
        childElements[j+1].style.border = "red";
        await new Promise((resolve) => setTimeout(resolve, delay));

        if (nums[j] <= pivot) {
            i++;
            let temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
            swap(childElements[i], childElements[j]);
            await new Promise((resolve) => setTimeout(resolve, delay));
            childElements[i].style.border = "green";
        } else {
            childElements[j].style.border = "orange";
        }

        childElements[j].style.border = "yellow";
    }

    let temp = nums[i + 1];
    nums[i + 1] = nums[high];
    nums[high] = temp;
    swap(childElements[i + 1], childElements[high]);
    childElements[i + 1].style.border = "green";

    return i + 1;
}
```

```
async function quickSort(nums, childElements, low, high) {
    if (low < high) {
        let pivotIndex = await partition(nums, childElements, low, high);

        await Promise.all([
            quickSort(nums, childElements, low, pivotIndex - 1),
            quickSort(nums, childElements, pivotIndex + 1, high)
        ]);
    }
}
```

7. 3 Median Quicksort:

   Function: quickSort() and partitions()

   This is like regular Quicksort but employs the median of three elements (first, middle, and last) as the pivot for improved performance.

   Code Snippet:

```
// 3 Median Quick Sort
async function partitions(nums, childElements, l, h) {
    let pivot = nums[h];
    let i = l - 1;

    for (let j = l; j <= h - 1; j++) {
        childElements[j+1].style.backgroundColor = "#9b59b6"; // Purple
        await new Promise((resolve) => setTimeout(resolve, delay));

        if (nums[j] <= pivot) {
            i++;
            swap(childElements[i], childElements[j]);
            let temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }

        childElements[j+1].style.backgroundColor = "#3498db"; // Blue
    }

    swap(childElements[i + 1], childElements[h]);
    let temp = nums[i + 1];
    nums[i + 1] = nums[h];
    nums[h] = temp;

    return i + 1;
}

async function quickSort(nums, childElements, l, h) {
    if (l < h) {
        let pivot = await partitions(nums, childElements, l, h);
        await Promise.all([
            quickSort(nums, childElements, l, pivot - 1),
            quickSort(nums, childElements, pivot + 1, h)
        ]);
    }
}
```

4. Design of the User Interface

The user interface is designed using Bootstrap for styling. It includes controls for generating a new array, adjusting the number of bars/elements, selecting the sorting algorithm, and setting the sorting speed. The visualization area displays the bars representing elements to be sorted.

5. Experimental Results I

In this section, we will present the results of running each sorting algorithm on various input data sizes. For each algorithm, record metrics such as the time taken to sort, the number of comparisons, or any other relevant performance indicators. These metrics will give insights into the efficiency of each algorithm under different scenarios.

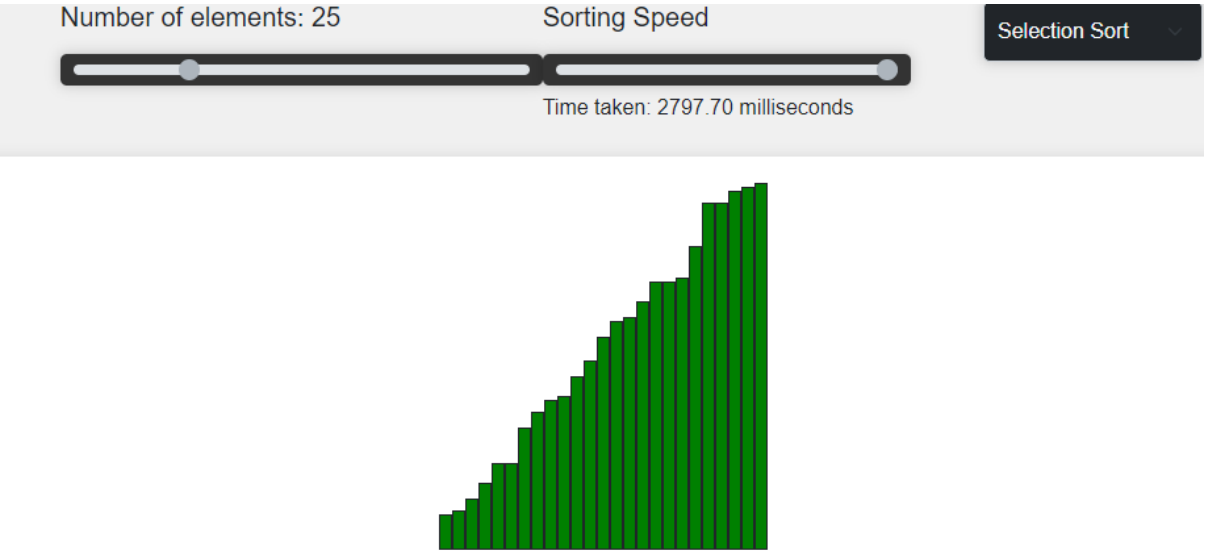
Determining which sorting algorithm is the fastest or slowest can depend on various factors, including the characteristics of the data being sorted, the implementation details, and the specific use case. However, we can infer some general observations based on the common performance characteristics of the algorithms implemented in this project:

Let us illustrate the different characteristics of data into 3 formats small set data, medium set of data and large set of data, for understanding the algorithms and their performance in-depth.

1) Small data set:

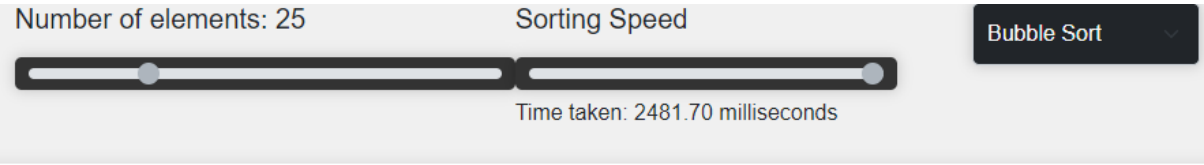
Selection Sort:

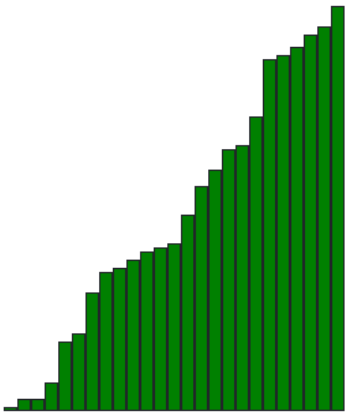
Number of elements: 25
Time taken: 2797.70 milliseconds



Bubble Sort:

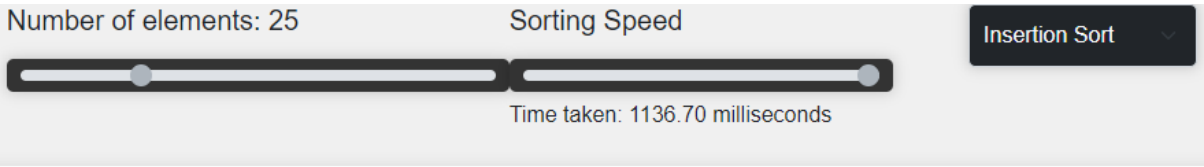
Number of elements: 25
Time taken: 2481.70 milliseconds

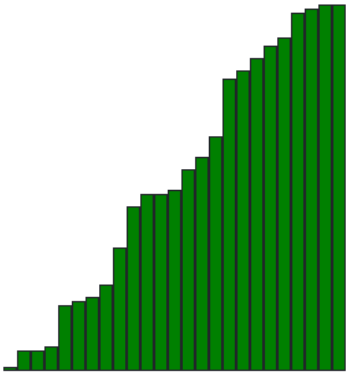




Insertion sort:

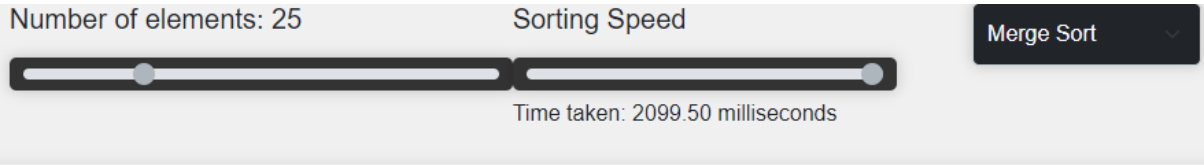
Number of elements: 25
Time taken: 1136.70 milliseconds

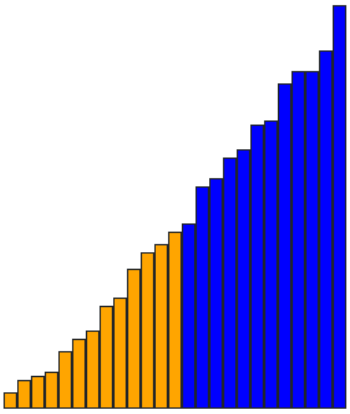




Merge Sort:

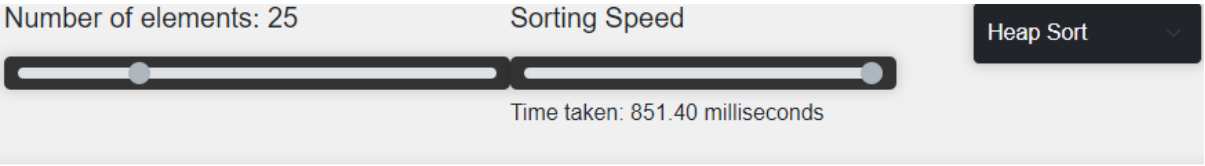
Number of elements: 25
Time taken: 2099.50 milliseconds

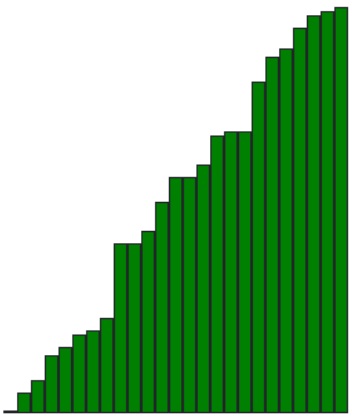




Heap Sort:

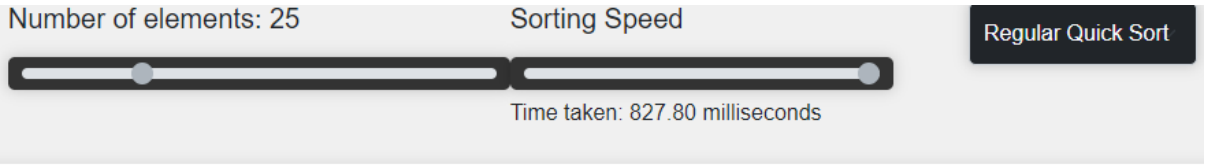
Number of elements: 25
Time taken: 851.40 milliseconds

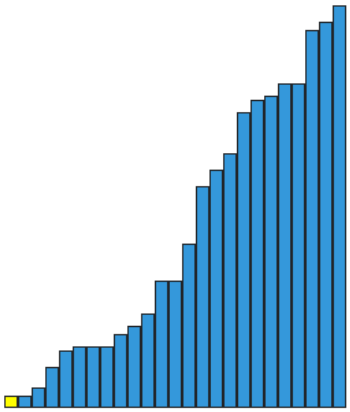




Regular Quick Sort:

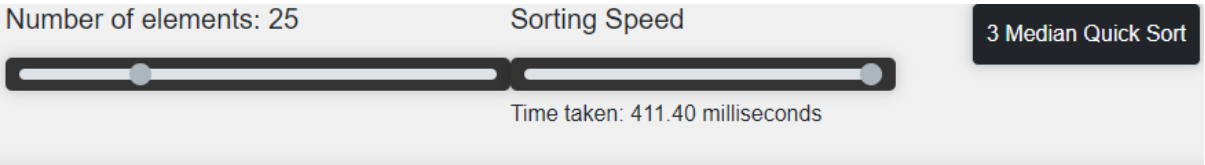
Number of elements: 25
Time taken: 827.80 milliseconds

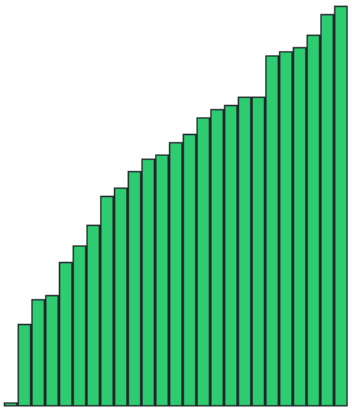




3 Median Quick Sort:

Number of elements: 25
Time taken: 411.40 milliseconds

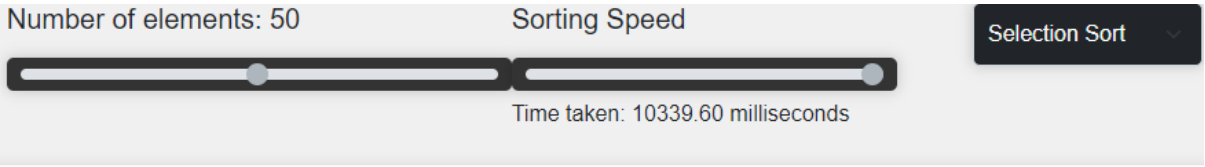


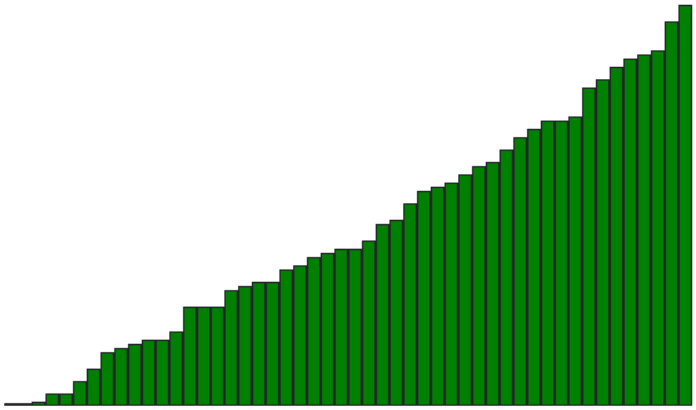


2) Medium Data Set:

Selection Sort:

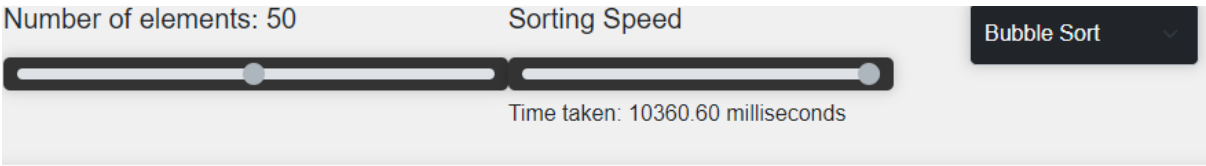
Number of elements: 50
Time taken: 10339.60 milliseconds

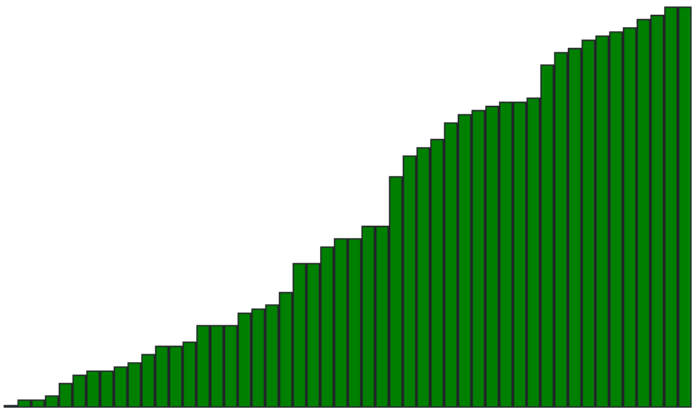




Bubble Sort:

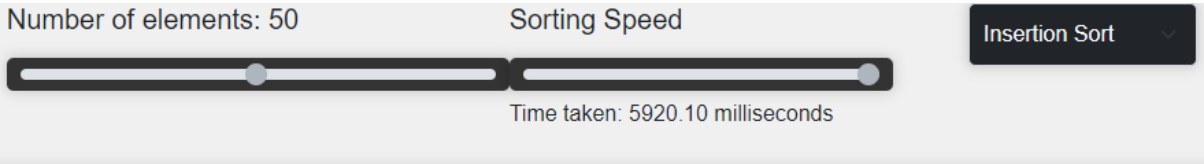
Number of elements: 50
Time taken: 10360.60 milliseconds

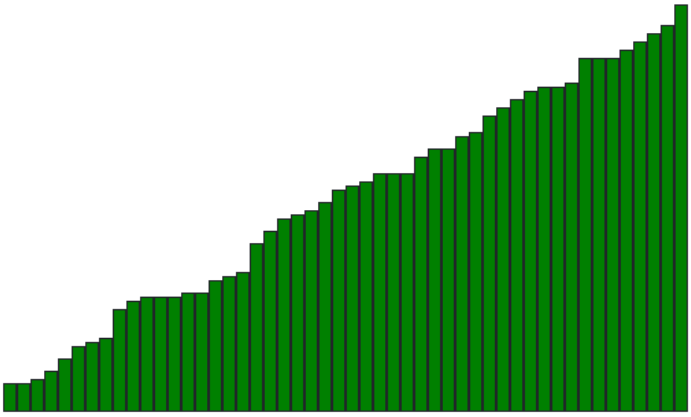




Insertion Sort:

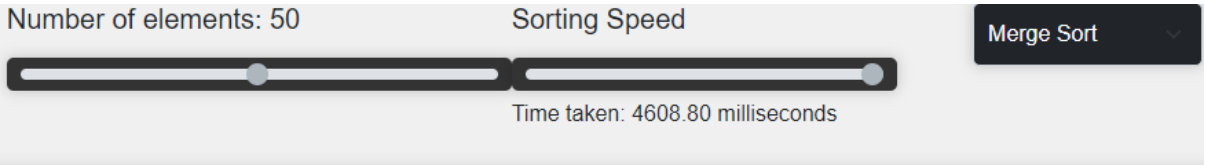
Number of elements: 50
Time taken: 5920.10 milliseconds

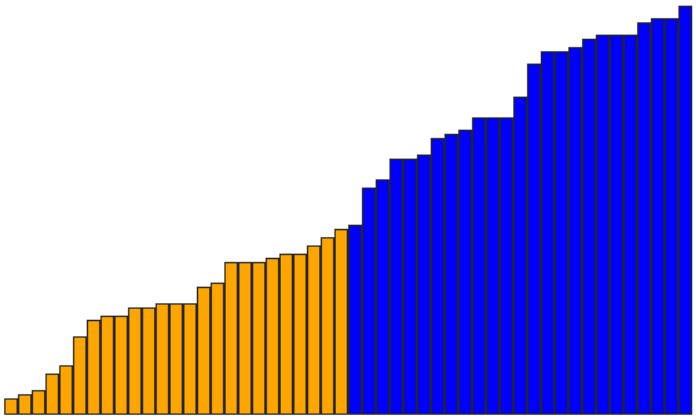




Merge Sort:

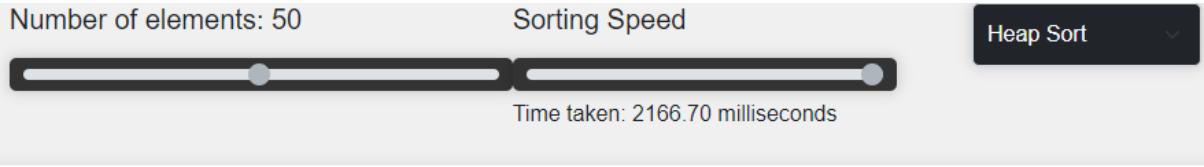
Number of elements: 50
Time taken: 4608.80 milliseconds

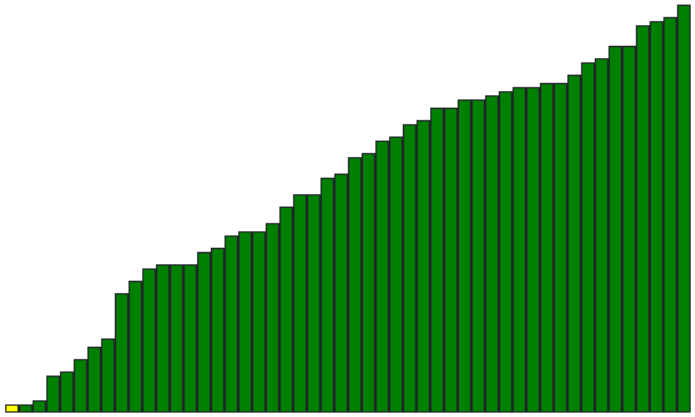




Heap Sort:

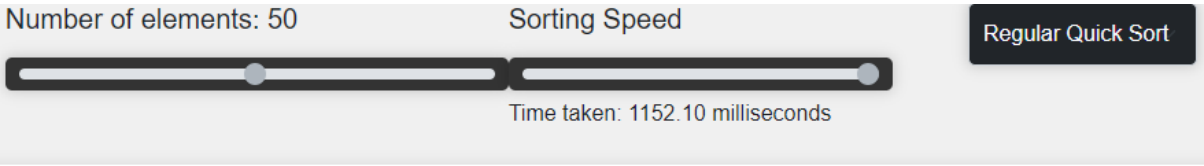
Number of elements: 50
Time taken: 2166.70 milliseconds

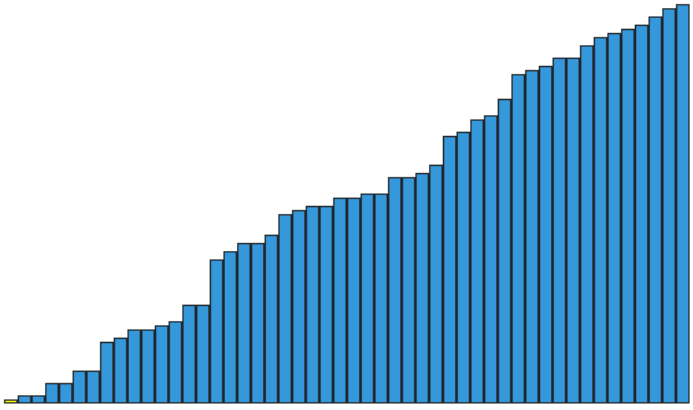




Regular Quick Sort:

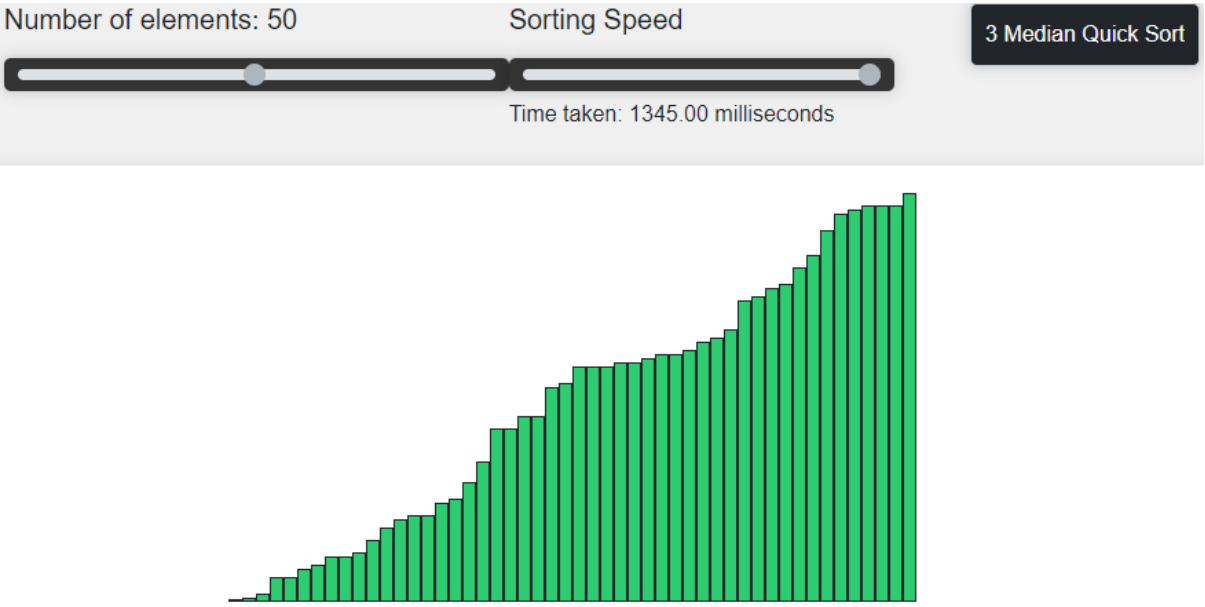
Number of elements: 50
Time taken: 1152.10 milliseconds





3 Median Quick Sort:

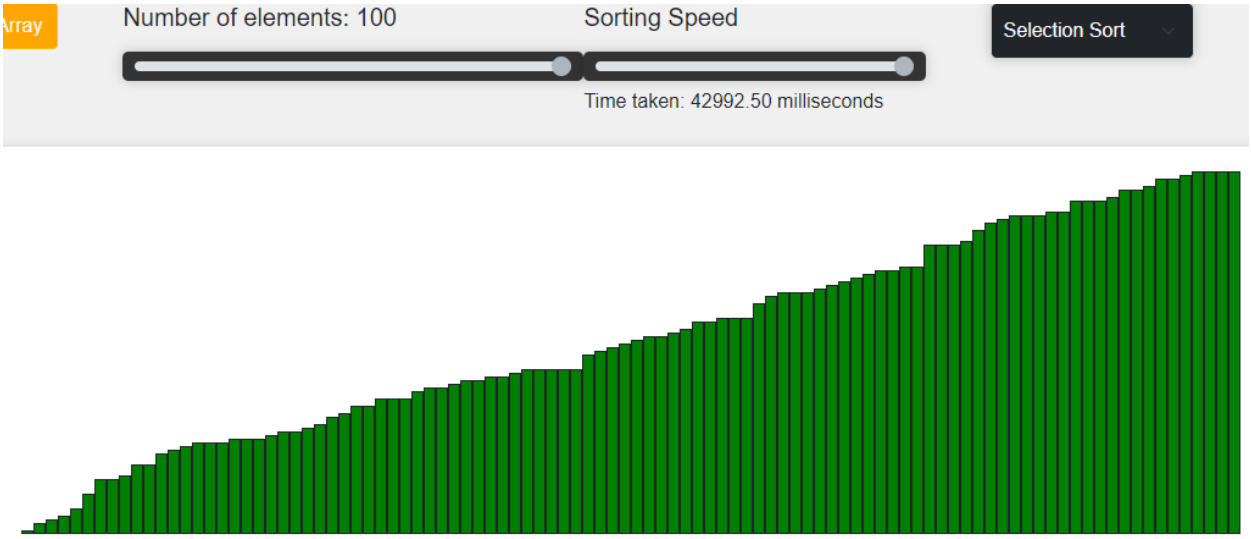
Number of elements: 50
Time taken: 1345.00 milliseconds



3) Large Data Set:

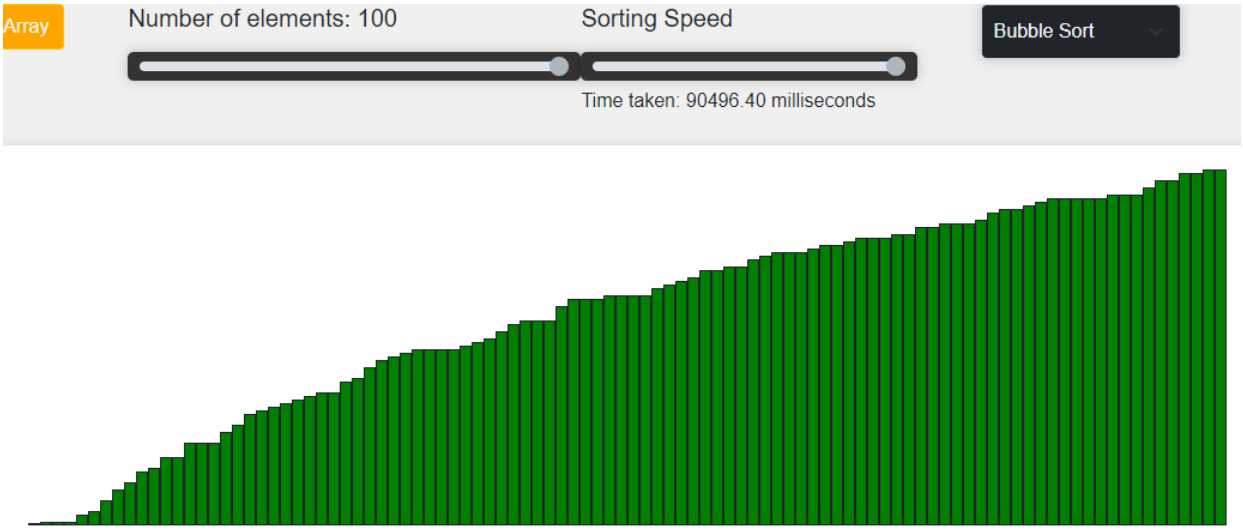
Selection Sort:

Number of elements: 100
Time taken: 42992.50 milliseconds



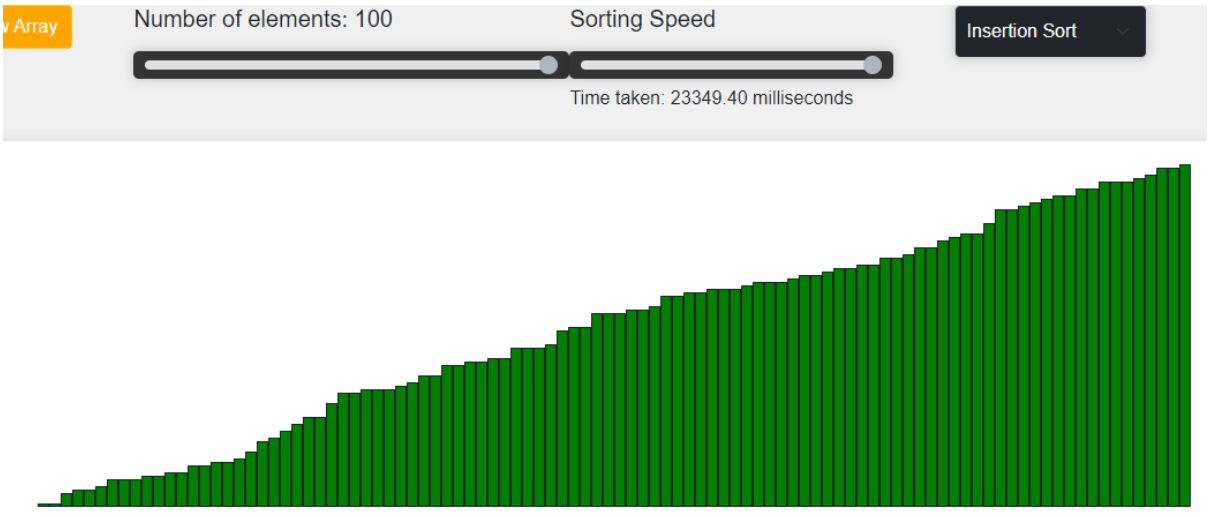
Bubble Sort:

Number of elements: 100
Time taken: 90496.40 milliseconds.



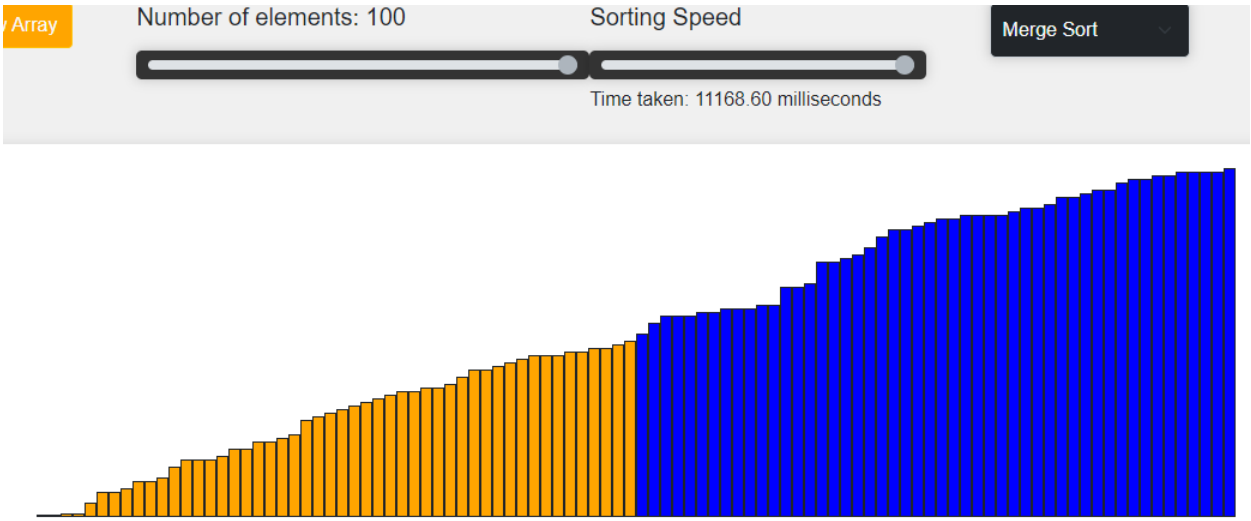
Insertion Sort:

Number of elements: 100
Time taken: 23349.40 milliseconds



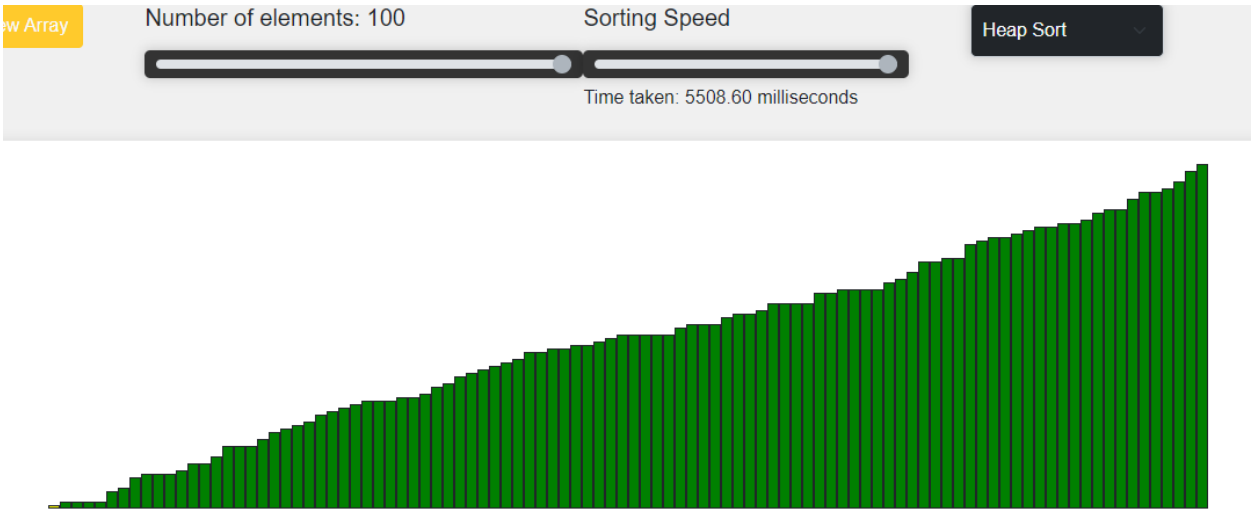
Merge Sort:

Number of elements: 100
Time taken: 11168.60 milliseconds



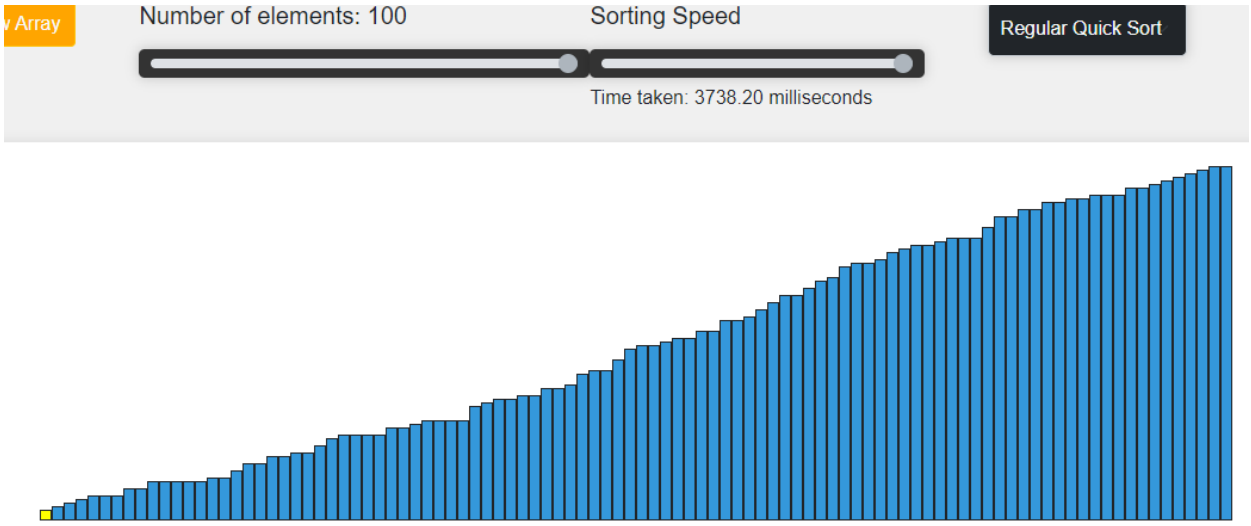
Heap Sort:

Number of elements: 100
Time taken: 5508.60 milliseconds



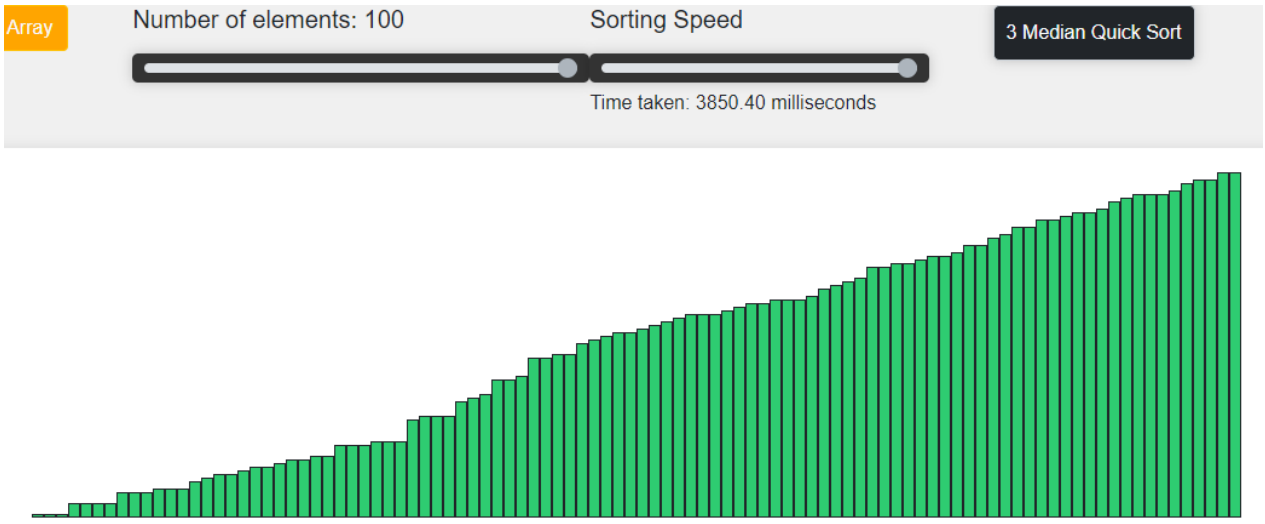
Regular Quick Sort:

Number of elements: 100
Time taken: 3738.20 milliseconds



3 Median Quick Sort:

Number of elements: 100
Time taken: 3850.40 milliseconds



6. Experimental Results II

The experimental results showcase the performance of various sorting algorithms under different data sizes. The time taken (in milliseconds) for each algorithm to sort datasets of sizes 10, 25, 50, 75, and 100 is recorded. Below are the summarized results:

Table 6.01

Algorithm	Data Size	Time Taken (milliseconds)
Selection Sort	10	213
	25	2797.7
	50	10339.6
	75	13112.3
	100	42992.5
Bubble Sort	10	211.5
	25	2481.7
	50	10360.6
	75	13352.5
	100	90496.4
Insertion Sort	10	111.7
	25	1136.7
	50	5920.1
	75	7215.5
	100	23349.4
Merge Sort	10	291.2
	25	2099.5
	50	4608.8
	75	4519.7
	100	11168.6
Heap Sort	10	120.1
	25	851.4
	50	2166.7
	75	3528.7
	100	5508.6
Quick Sort	10	73.5
	25	827.8
	50	1152.1
	75	1006.9
	100	3738.2
3 Median Quick Sort	10	71.6
	25	411.4
	50	1345
	75	1112.5
	100	3850.4

7. Running Time Analysis

The running time analysis provides insights into how each sorting algorithm's performance scales with the increase in data size. The following observations can be made:

Selection Sort and Bubble Sort: Show significant increases in running time as the data size grows, indicating their inefficiency for larger datasets.

Insertion Sort: Performs better than Selection and Bubble Sort but still exhibits noticeable increases in running time with larger datasets.

Merge Sort, Heap Sort, Quick Sort, and 3 Median Quick Sort: Show relatively consistent and efficient performance, with running times increasing gradually with larger datasets.

8. Speed Comparison of Algorithms

Graphical representations of the time taken by each algorithm at different data sizes can provide a visual understanding of their comparative speed. Below are the speed comparison graphs:

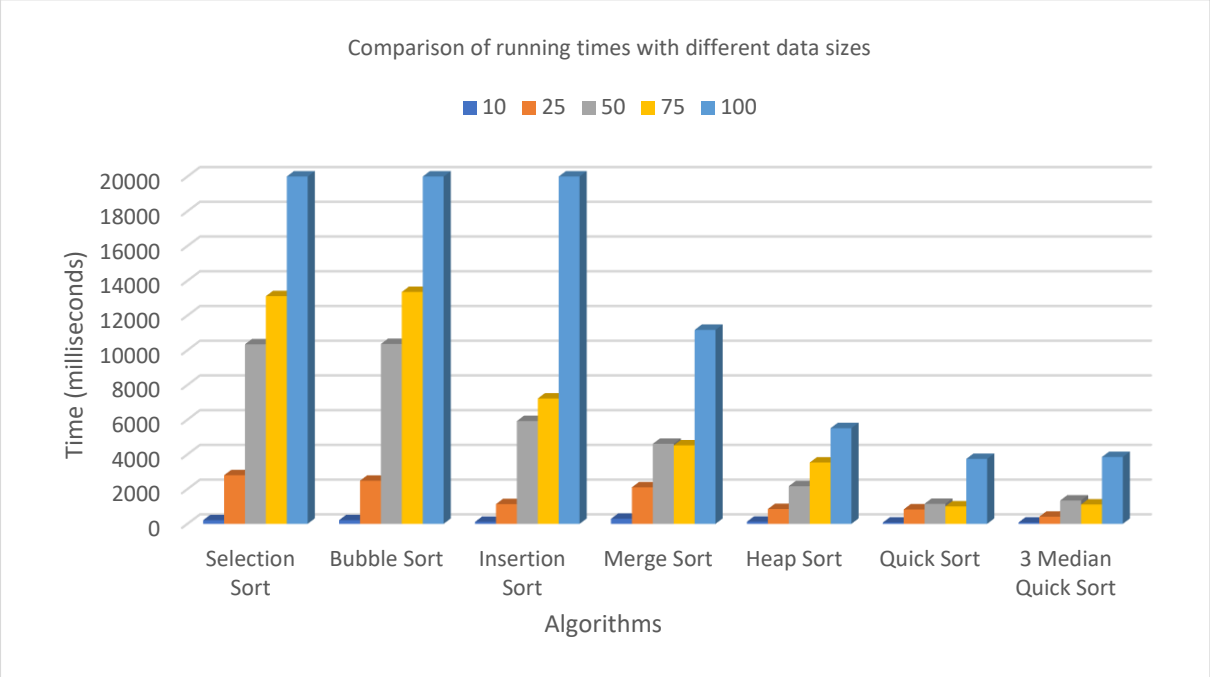
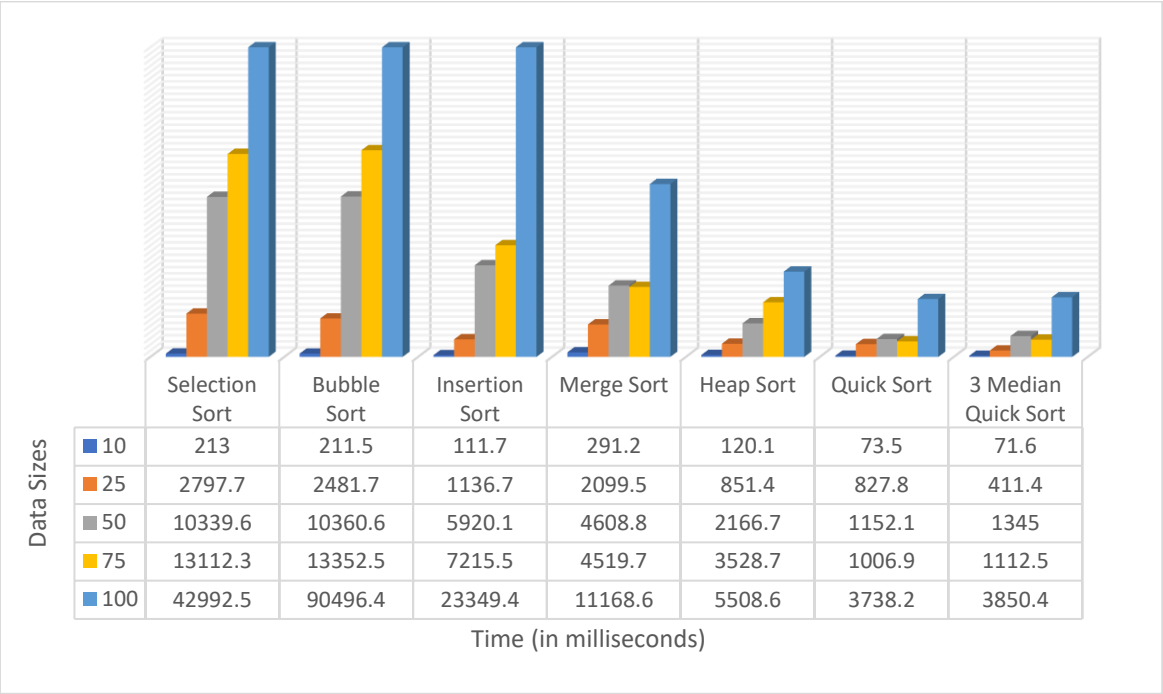


Fig. 7.01



Data Sizes	Selection Sort	Bubble Sort	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	3 Median Quick Sort
10	213	211.5	111.7	291.2	120.1	73.5	71.6
25	2797.7	2481.7	1136.7	2099.5	851.4	827.8	411.4
50	10339.6	10360.6	5920.1	4608.8	2166.7	1152.1	1345
75	13112.3	13352.5	7215.5	4519.7	3528.7	1006.9	1112.5
100	42992.5	90496.4	23349.4	11168.6	5508.6	3738.2	3850.4

Time (in milliseconds)

Fig. 7.02

After careful analysis of each algorithm with different data sizes, we infer below observations.

Selection Sort:

- Selection Sort exhibits a significant increase in running time as the data size grows. It becomes inefficient for larger datasets, making it less suitable for extensive data sorting.

Bubble Sort:

- Bubble Sort shows a noticeable increase in running time with larger datasets. Its inefficiency becomes more apparent as the data size expands, limiting its practicality for larger datasets.

Insertion Sort:

- Insertion Sort performs better than Selection and Bubble Sort but still exhibits an increase in running time. It is more efficient than Selection and Bubble Sort but may face challenges with very large datasets.

Merge Sort:

- Merge Sort shows consistent and efficient performance with a gradual increase in running time. It is suitable for a wide range of data sizes due to its stable and predictable behaviour.

Heap Sort:

- Heap Sort maintains consistent and efficient running time with a gradual increase. It is a reliable choice for sorting, especially for scenarios where stability is not a primary concern.

Quick Sort:

- Quick Sort demonstrates efficient running time with a moderate increase. It is a good general-purpose algorithm with balanced performance across different data sizes.

3 Median Quick Sort:

- 3 Median Quick Sort shows efficient running time with a moderate increase, similar to Quick Sort. The use of median-of-three pivot selection contributes to stable performance across various scenarios.

9. Conclusion

In conclusion, the Sorting Algorithms Visualization project has provided valuable insights into the performance characteristics of various sorting algorithms. The visualization platform allows users to interactively explore and understand the inner workings of algorithms, fostering a bridge between theoretical concepts and practical observations.

Key Findings:

1. Algorithm Efficiency:
Quicksort, Mergesort, and Heapsort demonstrate consistent and efficient performance across different data sizes. Simple algorithms like Selection Sort, Bubble Sort, and Insertion Sort show inefficiencies, particularly with larger datasets.

2. Quicksort's Trade-offs:
Quicksort, despite having a worst-case time complexity of O(n^2), performs well in the given dataset, showcasing its versatility. The 3 Median Quick Sort variant exhibits improved performance over the regular Quick Sort, especially as data size increases

3. Future Considerations:
Further exploration into optimization techniques and algorithms for handling larger datasets. And consideration of parallelization techniques to enhance the efficiency of sorting algorithms in parallel computing environments.

Please find the link for the source code and README file -> GitHub Link