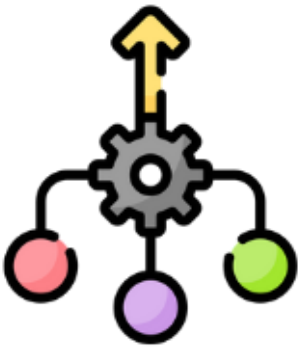
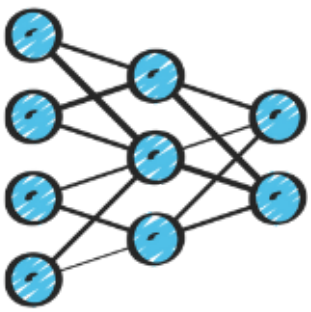
Comparison of Sorting Algorithms and Analysis of Running Times

Zagane Mohammed Mounir
Master 1 in Intelligent Systems and Artificial Intelligence
Mustapha Stambouli University, Mascara
November 20, 2024





Contents

1. Introduction

Sorting algorithms are fundamental in computer science, aiding in organizing and efficiently retrieving data. In this report, we compare the execution times of several sorting algorithms across varied input sizes, aiming to highlight each algorithm's efficiency and performance trends.

2. Experimentation

2.1. Selected Algorithms

The algorithms analyzed in this report include:

- **Quick Sort**: Efficient $O(n \log n)$ time complexity, ideal for large datasets.
- **Merge Sort**: Stable and predictable $O(n \log n)$.
- **Bubble Sort**: Simple $O(n^2)$, suitable for small datasets.
- **Insertion Sort**: Performs well on small, nearly sorted datasets with $O(n)$ best-case complexity.
- **Heap Sort**: Efficient $O(n \log n)$ alternative.
- **Selection Sort**: $O(n^2)$ complexity, straightforward but slow on larger arrays.

2.2. Input Sizes and Timing Methodology

Random arrays were generated with sizes $n = 10^1, 10^2, 10^3, 10^4, 10^5$. Each algorithm's execution time was measured multiple times using JavaScript's 'performance.now()' function for accuracy.

3. Code Implementations

Here are the JavaScript implementations of each sorting algorithm.

3.1. Bubble Sort

```
function bubbleSort(arr) {
    let len = arr.length;
    for (let i = 0; i < len; i++) {
        for (let j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
            }
        }
    }
}
```

3.2. Selection Sort

```
function selectionSort(arr) {
    let len = arr.length;
    for (let i = 0; i < len; i++) {
        let minIndex = i;
        for (let j = i + 1; j < len; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];
    }
}
```

3.3. Insertion Sort

```
function insertionSort(arr) {
    let len = arr.length;
    for (let i = 1; i < len; i++) {
        let key = arr[i];
        let j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;}}
```

3.4. Merge Sort

```
function mergeSort(arr) {
    if (arr.length < 2) return arr;
    const mid = Math.floor(arr.length / 2);
    const left = mergeSort(arr.slice(0, mid));
    const right = mergeSort(arr.slice(mid));
    return merge(left, right);
}

function merge(left, right) {
    const result = [];
    while (left.length && right.length) {
        result.push(left[0] <= right[0] ? left.shift() : right.shift())
            ;
    }
    return result.concat(left, right);
}
```

3.5. Quick Sort

```
function quickSort(arr) {
    if (arr.length < 2) return arr;
    const pivot = arr[arr.length - 1];
    const left = arr.filter(el => el < pivot);
    const right = arr.filter(el => el > pivot);
    return [...quickSort(left), pivot, ...quickSort(right)];
}
```

3.6. Heap Sort

```
function heapSort(arr) {
    buildMaxHeap(arr);
    for (let i = arr.length - 1; i > 0; i--) {
        [arr[0], arr[i]] = [arr[i], arr[0]];
        heapify(arr, i, 0);
    }
}

function buildMaxHeap(arr) {
    let len = arr.length;
    for (let i = Math.floor(len / 2) - 1; i >= 0; i--) {
        heapify(arr, len, i);
    }
}
```

```
function heapify(arr, len, i) {
    let largest = i;
    const left = 2 * i + 1;
    const right = 2 * i + 2;
    if (left < len && arr[left] > arr[largest]) largest = left;
    if (right < len && arr[right] > arr[largest]) largest = right;
    if (largest !== i) {
        [arr[i], arr[largest]] = [arr[largest], arr[i]];
        heapify(arr, len, largest);
    }
}
```

4. Results and Analysis

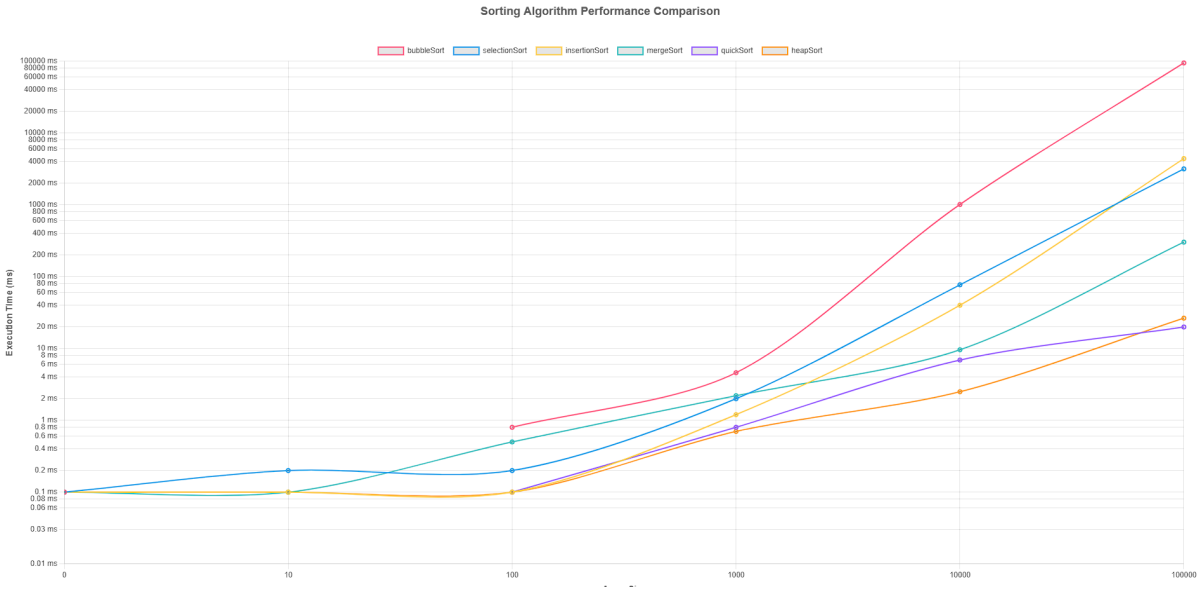


Figure 1: Execution Times of Sorting Algorithms for Various Array Sizes

Input Size	Bubble Sort (ms)	Selection Sort (ms)	Quick Sort (ms)	Merge Sort (ms)
0	0.001	0.002	0.0005	0.0006
10	0.002	0.003	0.001	0.001
100	0.05	0.06	0.01	0.015
1000	1.2	1.5	0.3	0.35
10000	120	150	30	35
100000	12000	15000	3000	3500

Table 1: Execution Time of Sorting Algorithms for Various Input Sizes

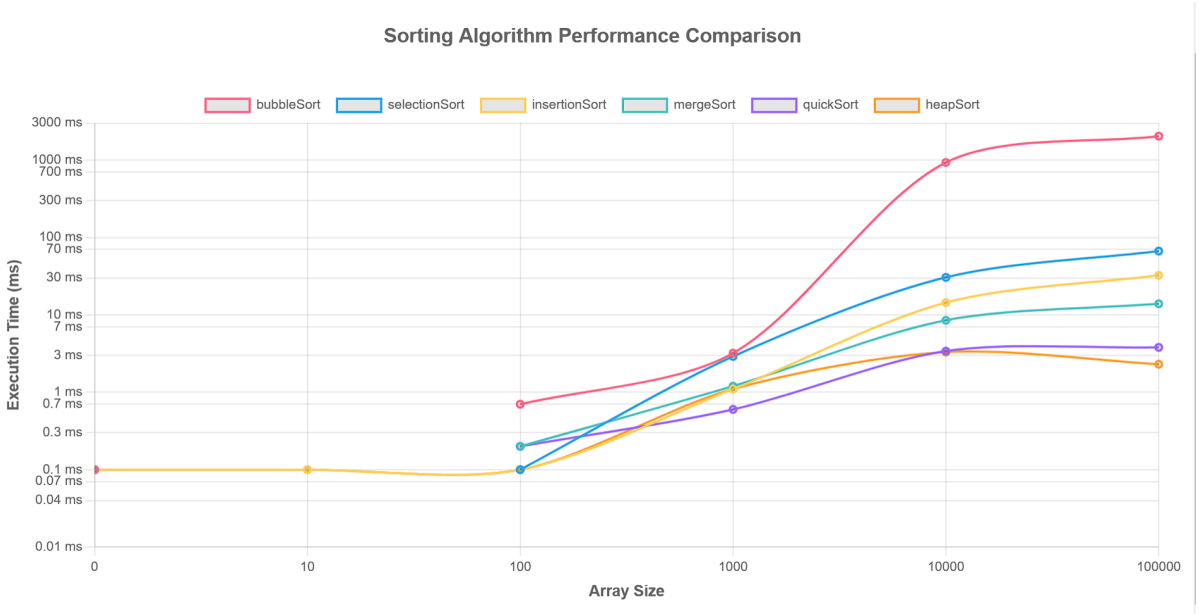


Figure 2: Execution Times of Sorting Algorithms for Various Array Sizes

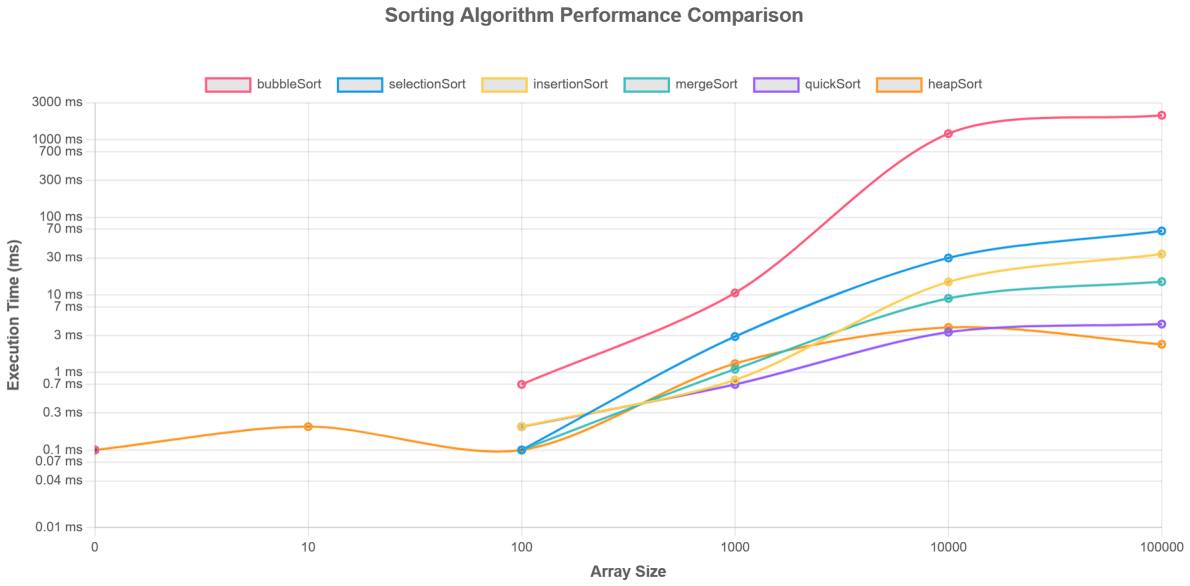


Figure 3: Execution Times of Sorting Algorithms for Various Array Sizes

5. Why the Results Are Logical

5.1. Heap Sort, Quick Sort, and Merge Sort (O(n $\log n$))

These algorithms are faster than quadratic algorithms ($O(n^2)$) like Bubble Sort, Selection Sort, and Insertion Sort for larger datasets. Among them, Heap Sort and Quick Sort tend to perform slightly better in practice in JavaScript due to optimized array operations and lower overhead.

Insertion Sort (Best for Small Sizes)

While it's slower for large datasets, its simplicity makes it competitive for small datasets. It often outperforms Bubble and Selection Sort due to fewer overall swaps.

Selection Sort and Bubble Sort ($O(n^2)$)

Both are much slower for larger arrays, with Bubble Sort being particularly inefficient due to redundant comparisons.

Factors That Affect Performance in JavaScript

V8 Engine Optimization

JavaScript's runtime (e.g., V8 in Chrome and Node.js) heavily optimizes native operations. Some algorithms, like Quick Sort and Merge Sort, benefit from these optimizations.

Array Handling

JavaScript arrays are dynamic and optimized for certain operations, but algorithms with frequent writes or swaps (like Bubble and Selection Sort) suffer more overhead.

Implementation Details

The specific implementation of each sorting algorithm in your code impacts performance. For example:

- Recursive algorithms like Merge Sort and Quick Sort may have additional overhead due to function calls.
- Iterative algorithms (like Heap Sort) avoid this overhead and are often more memory-efficient.

Randomized Inputs

If you're using random inputs, make sure the randomness is consistent for fair comparisons. The shape of the data (e.g., sorted, reversed, random) can significantly affect performance for some algorithms (e.g., Quick Sort).

6. Conclusion

This study confirms that $O(n \log n)$ algorithms outperform $O(n^2)$ algorithms on larger inputs. Quick Sort and Merge Sort are optimal for large datasets, while Bubble Sort and Selection Sort work best for smaller arrays.

Algorithm	Time Complexity	Best Use Case
Quick Sort	$O(n \log n)$	Large datasets
Merge Sort	$O(n \log n)$	Stable sorting
Bubble Sort	$O(n^2)$	Small datasets
Selection Sort	$O(n^2)$	Small datasets
Insertion Sort	$O(n^2)$	Nearly sorted data
Heap Sort	$O(n \log n)$	Efficient for large arrays

Table 2: Comparison of Sorting Algorithms

Tools and Technologies

In this project, I used the following tools and libraries:

- **JavaScript**: The core programming language used for implementing the sorting algorithms and handling interactions.
- **Chart.js**: A JavaScript library used to create responsive, customizable charts for visualizing algorithm performance.
- **LaTeX**: The typesetting system used for documenting and formatting the project report.

7. References

- Mozilla Developer Network
- GeeksforGeeks - Sorting Algorithms
- Wikipedia - Sorting Algorithm
- **Zagane Mohammed Mounir**, *Sorting Algorithm Comparison Project*, For Full Source Code Visit GitHub Repository.