Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Laboratory No.2 Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, one of choice.

Verified:
Fistic Cristofor asist. univ.

Titerez Vladislav FAF-233

Moldova, February 2025

Contents

1 Algorithm Analysis

1.1 Objective

Learn sorting algorithms and their relevance. Evaluate the speed of sorting algorithms in the perspective of an increasing number of array elements.

1.2 Tasks

- Study sorting algorithms
- Implement the algorithms listed above in a programming language
- Establish the properties of the input data against which the analysis is performed
- Choose metrics for comparing algorithms
- Perform empirical analysis of the proposed algorithms
- Make a graphical presentation of the data obtained
- Make a conclusion on the work done.

1.3 Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer. In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.

3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate. After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Sorting algorithms play a crucial role in computer science as they serve as the foundation for many other algorithms and applications, such as searching, data compression, and computational geometry. The efficiency of a sorting algorithm significantly impacts overall system performance, especially when dealing with large datasets.

Time complexity analysis of sorting algorithms helps in choosing the most suitable algorithm for a given problem. For example, algorithms like QuickSort and MergeSort offer good average and worst-case performance, making them preferable for general-purpose sorting. On the other hand, simple algorithms like BubbleSort or InsertionSort might be suitable for small datasets due to their ease of implementation and low overhead.

Understanding the complexity of time (O (n log n), O (n 2), etc.) allows developers to predict performance under various conditions and make informed decisions when designing efficient applications. Empirical analysis of sorting algorithms by measuring execution time on different input sizes further helps in evaluating their real-world performance, considering factors such as hardware optimizations and caching mechanisms.

1.4 Introduction

Sorting algorithms play a fundamental role in computer science, serving as the backbone for various computational tasks such as searching, data compression, and computational geometry. The efficiency of a sorting algorithm significantly impacts overall system performance, especially when dealing with large datasets.

Sorting algorithms can be broadly categorized into two types: comparison-based sorting and non-comparison-based sorting. Comparison-based algorithms, such as QuickSort, MergeSort, and HeapSort, rely on pairwise element comparisons, typically achieving a time complexity of $O(n \log n)$ in the best and average cases. Conversely,

non-comparison-based algorithms, such as Counting Sort, Radix Sort, and Bucket Sort, leverage data properties to achieve linear time complexity under specific conditions.

Understanding the theoretical complexity of sorting algorithms (e.g., $O(n \log n)$, $O(n^2)$, $O(n)$) enables developers to predict performance under various conditions and make informed decisions when selecting an algorithm. However, theoretical analysis alone is not always sufficient. Empirical analysis plays a crucial role in evaluating real-world performance, considering factors such as hardware architecture, memory access patterns, and caching mechanisms.

This study explores different sorting algorithms through empirical analysis, measuring their execution times on various input sizes. By conducting experiments, we aim to compare the practical efficiency of sorting algorithms and determine their suitability for different types of datasets.

1.5 Comparison Metric

The primary comparison metric in this study is the execution time of each sorting algorithm, denoted as $T(n)$. This metric provides insights into the practical efficiency of algorithms under varying input sizes and conditions. While theoretical complexity provides an upper-bound estimate, execution time accounts for hardware optimizations, cache behavior, and implementation details that influence performance.

1.6 Input Format

Each sorting algorithm will be tested on multiple datasets of different sizes to evaluate its efficiency across various conditions. The datasets used for testing are:

- **Small Dataset:** Designed for analyzing the performance of algorithms on smaller inputs, this dataset consists of randomly generated arrays with sizes ranging from:

$$10, 50, 100, 200, 500$$

- **Large Dataset:** Used for testing scalability and efficiency on larger inputs, this dataset contains arrays of increasing sizes:

$$1000, 5000, 10000, 50000, 100000$$

This structured input format ensures a comprehensive evaluation of sorting algorithms, providing a clear comparison between their theoretical and empirical performance.

2 Implementation

All four algorithms will be implemented in their native form in JavaScript and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on the memory of the device used.
The error margin determined will constitute 2.5 seconds as per experimental measurement.

2.1 Quick Sort

2.1.1 Introduction

Quick Sort is a widely used and efficient sorting algorithm based on the divide-and-conquer paradigm. It was developed by Tony Hoare in 1960.

2.1.2 Working Principle

The algorithm selects a *pivot* element from the array and partitions the remaining elements into two subarrays: one containing elements smaller than the pivot and the other containing elements greater than the pivot. This partitioning step ensures that the pivot is placed in its correct sorted position. The process is then recursively applied to the two subarrays.

2.1.3 Advantages

Quick Sort is preferred due to its in-place sorting capability and high performance on large datasets.

2.1.4 JS Implementation

```
function quickSort(arr) {
    if (arr.length < 2) return arr;
    let min = 1;
    let max = arr.length - 1;
    let rand = Math.floor(min + Math.random() * (max + 1 - min));
    let pivot = arr[rand];
    const left = [];
    const right = [];
    arr.splice(arr.indexOf(pivot), 1);
    arr = [pivot].concat(arr);
    for (let i = 1; i < arr.length; i++) {
      if (pivot > arr[i]) {
        left.push(arr[i]);
      } else {
        right.push(arr[i]);
      }
    }
    return quickSort(left).concat(pivot, quickSort(right));
}
```

Listing 2.1: *Quick Sort*

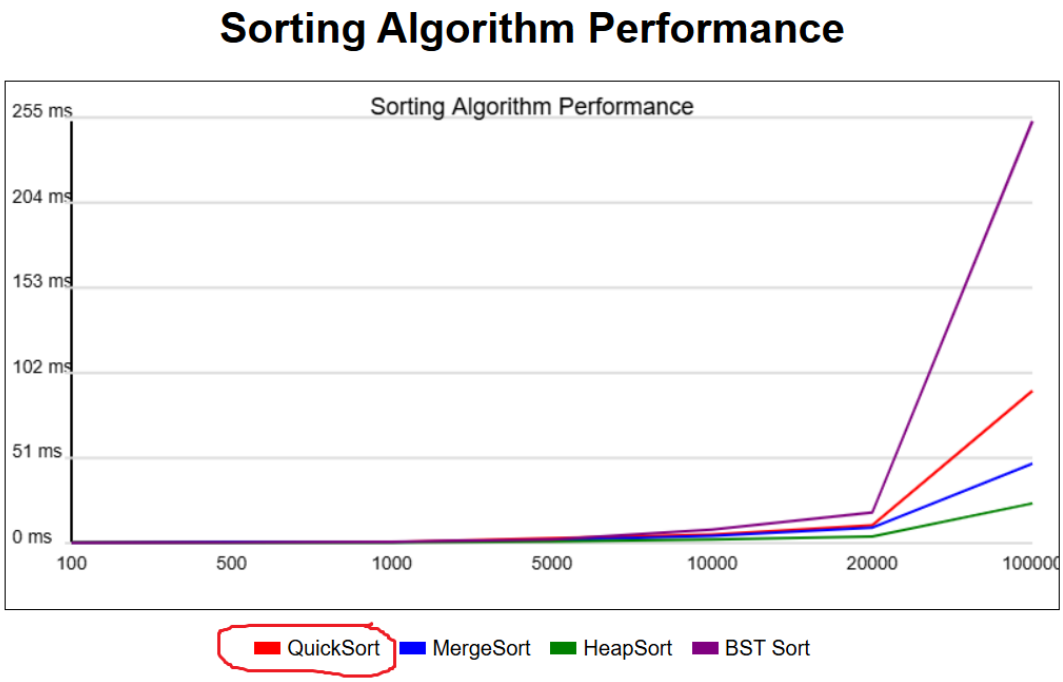

Figure 2.1: *Quick Sort graph*

2.1.5 Time Complexity

Time Complexity:

- Best and Average Case: $O(n \log n)$
- Worst Case: $O(n^2)$

2.1.6 Auxiliary Space

Auxiliary Space:

- Best and Average Case: $O(\log n)$
- Worst Case: $O(n)$

2.2 Merge Sort

2.2.1 Introduction

Merge Sort is a stable and efficient sorting algorithm that follows the divide-and-conquer approach. It was first introduced by John von Neumann in 1945.

2.2.2 Working Principle

The algorithm recursively divides the array into two halves until each subarray contains a single element. Then, it merges the subarrays in a sorted manner by comparing elements from both halves. The merging step ensures that the resulting array is fully sorted.

2.2.3 Advantages

Merge Sort is particularly useful for sorting linked lists and large datasets. It is stable and has a predictable time complexity, making it ideal for applications requiring reliable performance.

2.2.4 JS Implementation

```
function merge(left, right) {
    let resultArray = [],
        leftIndex = 0,
        rightIndex = 0;
    while (leftIndex < left.length && rightIndex < right.length) {
        if (left[leftIndex] < right[rightIndex]) {
            resultArray.push(left[leftIndex]);
            leftIndex++;
        } else {
            resultArray.push(right[rightIndex]);
            rightIndex++;
        }
    }
    return resultArray
        .concat(left.slice(leftIndex))
        .concat(right.slice(rightIndex));
}

function mergeSort(array) {
```

```
20      if (array.length === 1) {
21          return array;
22      }
23      const middle = Math.floor(array.length / 2);
24      const left = array.slice(0, middle);
25      const right = array.slice(middle);
26      return merge(
27          mergeSort(left),
28          mergeSort(right)
29      );
30  }
```

Listing 2.2: *Merge Sort*

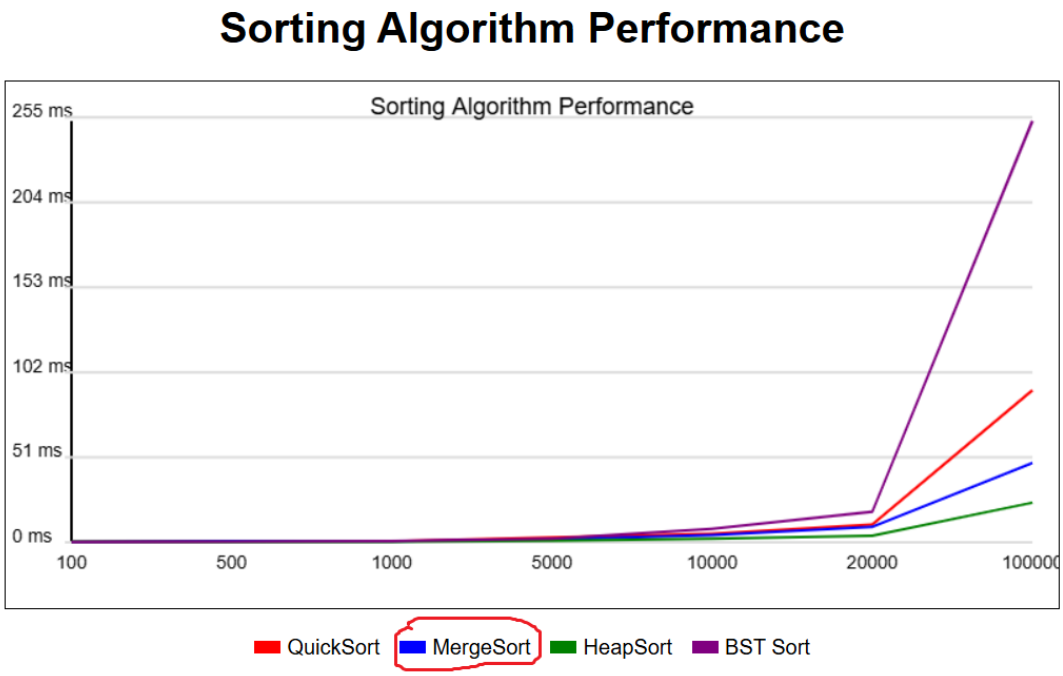

Figure 2.2: *Merge Sort graph*

2.2.5 Time Complexity

Time Complexity:

- Best, Average, and Worst Case: $O(n \log n)$

2.2.6 Auxiliary Space

Auxiliary Space:

- All Cases: $O(n)$

2.3 Heap Sort

2.3.1 Introduction

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap data structure. It was introduced by J. W. J. Williams in 1964. Unlike Merge Sort, Heap Sort is not stable but is efficient with a guaranteed $O(n \log n)$ time complexity and can be implemented in-place with constant auxiliary space, though the following implementation uses $O(n)$ space for clarity.

2.3.2 Working Principle

The algorithm first constructs a max heap from the input array. The largest element (at the root) is repeatedly swapped with the last element of the heap, reducing the heap size by one. The heap is restored by heapifying the root. This process continues until the entire array is sorted.

2.3.3 Advantages

Heap Sort is efficient for in-place sorting and performs consistently well with $O(n \log n)$ time complexity in all cases. It is particularly useful in systems with limited memory due to its minimal space usage when implemented in-place.

2.3.4 JS Implementation

```
function swap(array, index1, index2) {
    [array[index1], array[index2]] = [array[index2], array[index1]];
}

function heapify(array, index, length = array.length) {
    let largest = index,
        left = index * 2 + 1,
        right = index * 2 + 2;

    if (left < length && array[left] > array[largest]) {
        largest = left;
    }
    if (right < length && array[right] > array[largest]) {
        largest = right;
    }

    if (largest !== index) {
        swap(array, index, largest);
        heapify(array, largest, length);
    }

    return array;
}
```

```
24
25 function heapSort(array) {
26     let arr = [...array];
27
28     for (let i = Math.floor(arr.length / 2); i >= 0; i--) {
29         heapify(arr, i);
30     }
31
32     for (let i = arr.length - 1; i > 0; i--) {
33         swap(arr, 0, i);
34         heapify(arr, 0, i);
35     }
36
37     return arr;
38 }
```

Listing 2.3: *Heap Sort*

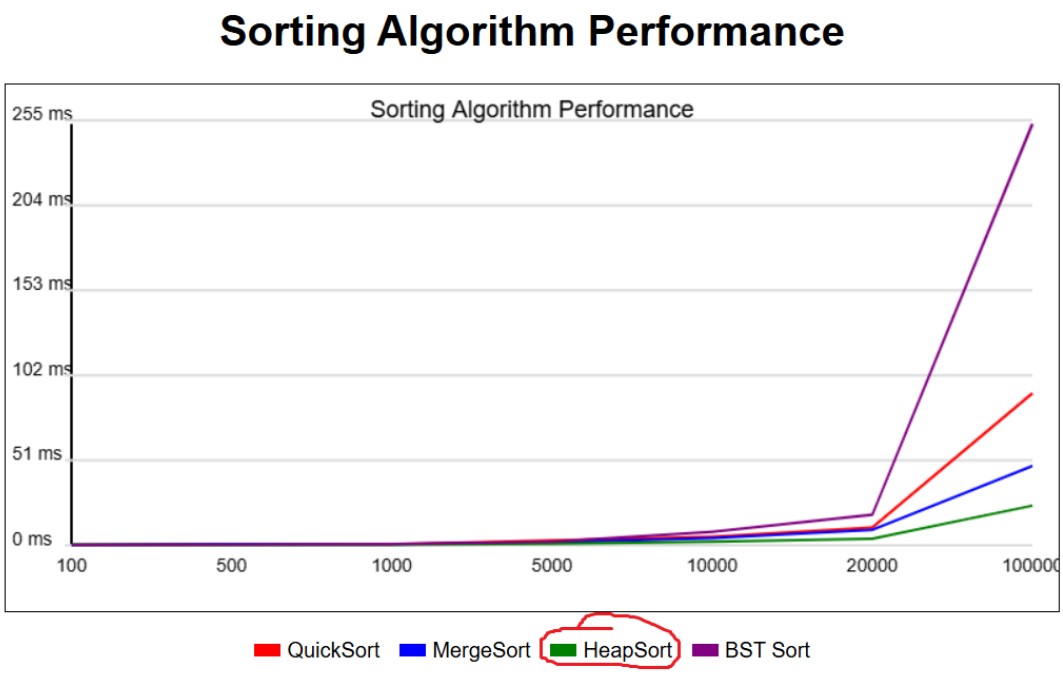

Figure 2.3: *Heap Sort graph*

2.3.5 Time Complexity

Time Complexity:

- Best, Average, and Worst Case: $O(n \log n)$

2.3.6 Auxiliary Space

Auxiliary Space:

- All Cases: $O(n)$

2.4 Binary Search Tree Sort

2.4.1 Introduction

Binary Search Tree (BST) Sort leverages the properties of a BST to sort elements. While not a traditional comparison-based sorting algorithm, it uses in-order traversal to produce a sorted sequence. This approach is conceptually distinct from divide-and-conquer algorithms like Merge Sort, and its efficiency depends on the balance of the tree.

2.4.2 Working Principle

The algorithm inserts all elements of an unsorted array into a BST. After construction, an in-order traversal of the tree retrieves elements in ascending order. Insertion follows BST rules: smaller values go to the left subtree, and larger/equal values go to the right. The traversal process guarantees sorted output due to the BST's inherent ordering.

2.4.3 Advantages

BST Sort provides intuitive sorting logic and works well for dynamically updated datasets. While the average-case time complexity matches efficient algorithms like Merge Sort, its performance degrades to $O(n^2)$ for skewed trees. Unlike Heap Sort, it preserves insertion order for equal elements (if stability is explicitly implemented).

2.4.4 JS Implementation

```
function sortWithBST(array) {
    class Node {
        constructor(data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    class BST {
        constructor() {
            this.root = null;
        }

        insert(data) {
            let newNode = new Node(data);
            if (this.root === null) {
                this.root = newNode;
            } else {
                this.insertNode(this.root, newNode);
            }
        }

```

```
24          insertNode(node, newNode) {
25              if (newNode.data < node.data) {
26                  if (node.left === null) {
27                      node.left = newNode;
28                  } else {
29                      this.insertNode(node.left, newNode);
30                  }
31              } else {
32                  if (node.right === null) {
33                      node.right = newNode;
34                  } else {
35                      this.insertNode(node.right, newNode);
36                  }
37              }
38          }
39
40          inOrder(node, result) {
41              if (node !== null) {
42                  this.inOrder(node.left, result);
43                  result.push(node.data);
44                  this.inOrder(node.right, result);
45              }
46          }
47
48          getSortedArray() {
49              let result = [];
50              this.inOrder(this.root, result);
51              return result;
52          }
53      }
54
55      let bst = new BST();
56      for (let i = 0; i < array.length; i++) {
57          bst.insert(array[i]);
58      }
59
60      return bst.getSortedArray();
61  }
```

Listing 2.4: *Binary Search Tree Sort*

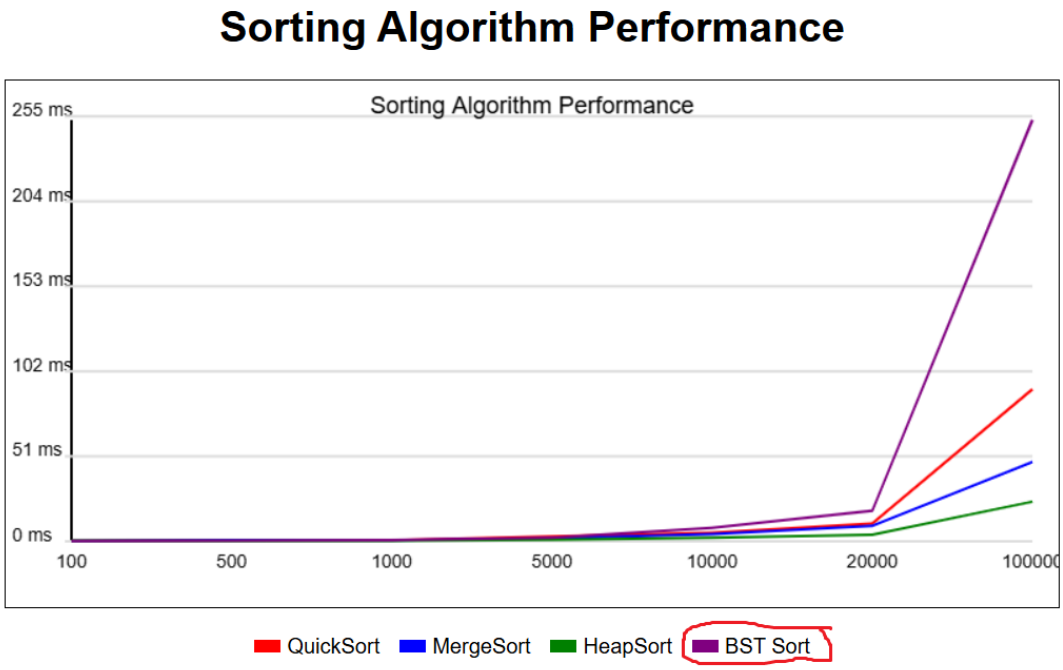

Figure 2.4: *Binary Search Tree Sort graph*

2.4.5 Time Complexity

Time Complexity:

- Best/Average Case: $O(n \log n)$ (balanced tree)
- Worst Case: $O(n^2)$ (completely skewed tree)

2.4.6 Auxiliary Space

Auxiliary Space:

- All Cases: $O(n)$ (tree storage + recursion stack)

2.5 Visualization

To better understand the inner workings and efficiency of the sorting algorithms discussed, I have included visual representations that illustrate the step-by-step processes involved. These visualizations not only provide a clearer picture of how each algorithm operates but also highlight the differences in their approaches to sorting data. Below, you will find a graphical representation of the Binary Search Tree (BST) Sort, which showcases how elements are organized and retrieved in a binary tree structure to achieve a sorted sequence.

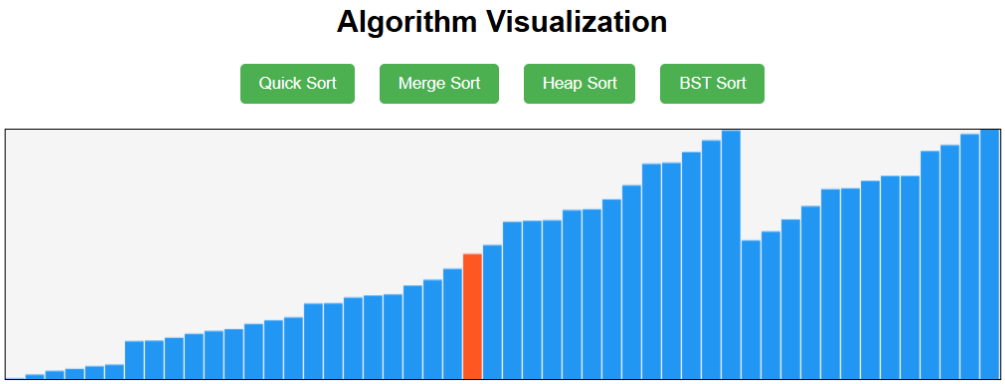

Figure 2.5: *Binary Search Tree Sort graph*

This figure demonstrates the elegant and efficient nature of BST Sort, where each node in the tree plays a crucial role in maintaining the order of elements. As you can see, the tree structure allows for quick insertion and retrieval, making it a powerful tool for sorting and searching operations.

3 Conclusion

Through empirical analysis, this paper has evaluated four sorting algorithms—Quick Sort, Merge Sort, Heap Sort, and Binary Search Tree (BST) Sort—for their efficiency, time complexity, and suitability in different scenarios. The goal was to identify the strengths and weaknesses of each algorithm and determine their optimal use cases.

Quick Sort

Quick Sort is a highly efficient divide-and-conquer algorithm with an average time complexity of $O(n \log n)$. However, its worst-case time complexity of $O(n^2)$ occurs when the pivot selection is unbalanced. Despite this, Quick Sort is widely used due to its in-place sorting capability and excellent performance on large datasets. It is ideal for applications where memory usage is a concern and average-case performance is acceptable.

Merge Sort

Merge Sort is a stable sorting algorithm with a consistent time complexity of $O(n \log n)$ across all cases. Its divide-and-conquer approach ensures reliable performance, making it suitable for sorting linked lists and large datasets. However, its $O(n)$ auxiliary space requirement can be a limitation in memory-constrained environments. Merge Sort is recommended for applications requiring stability and predictable performance.

Heap Sort

Heap Sort is an in-place sorting algorithm with a guaranteed time complexity of $O(n \log n)$ in all cases. It leverages the properties of a binary heap to sort elements efficiently. While not stable, Heap Sort is particularly useful in systems with limited memory due to its minimal space usage. It is an excellent choice for scenarios where both time and space efficiency are critical.

Binary Search Tree Sort

BST Sort uses the properties of a binary search tree to sort elements through in-order traversal. While its average-case time complexity is $O(n \log n)$, it degrades to $O(n^2)$ for skewed trees. BST Sort is intuitive and works well for dynamically updated datasets, but its performance is highly dependent on the balance of the tree. It is suitable for moderate-sized datasets where simplicity and dynamic updates are prioritized.

Summary

Each sorting algorithm has distinct advantages and trade-offs, making them suitable for different use cases:

- **Quick Sort**: Best for large datasets with average-case performance and in-place sorting.
- **Merge Sort**: Ideal for stable sorting and predictable performance, especially with linked lists.
- **Heap Sort**: Optimal for in-place sorting with guaranteed $O(n \log n)$ time complexity.
- **BST Sort**: Suitable for moderate-sized datasets and dynamic updates, but performance varies with tree balance.

Future work could focus on hybrid approaches that combine the strengths of these algorithms, such as using Quick Sort with improved pivot selection or integrating BST Sort with balancing techniques like AVL trees. Additionally, further optimization of auxiliary space usage in Merge Sort and Heap Sort could enhance their applicability in memory-constrained environments.
https://github.com/vvtttvv/AA/tree/main/lab2