Ministerul Educaţiei şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică

Laboratory Work 2

Study and Empirical Analysis of Sorting Algorithms

Elaborated:
std. gr. FAF-231 Alexandru Rudoi

Verified:
asist. univ. Cristofor Fiştic

Chişinău – 2025

Contents

1 Algorithm Analysis

1.1 Objective

The main objective of this laboratory work is to analyze and compare different sorting algorithms in terms of *execution time, efficiency, stability, and scalability.* By implementing and benchmarking multiple approaches, we aim to:

- Understand the impact of algorithmic complexity on sorting performance.
- Identify the most efficient sorting algorithm for various input characteristics.
- Visualize performance trends through execution time measurements and graphical analysis.
- Explore the trade-offs between different sorting techniques, including space usage and adaptability.

1.2 Tasks

To achieve these objectives, the following tasks were completed:

1. Implemented multiple sorting algorithms, including QuickSort, MergeSort, HeapSort, and RadixSort.
2. Established input data properties (random, nearly sorted, reversed) for empirical analysis.
3. Measured and recorded execution times for different dataset sizes.
4. Plotted execution time graphs to compare algorithm efficiency under different conditions.
5. Designed an optimized version of each algorithm to improve efficiency.
6. Implemented a hybrid sorting algorithm that dynamically selects the best method based on input characteristics.
7. Developed a live visualization for educational and analytical purposes.

1.3 Theoretical Notes

Sorting is one of the fundamental operations in computer science, often used in searching, data analysis, and optimization. The sorting problem involves arranging a sequence of elements in a specific order (e.g., ascending or descending).

Different sorting algorithms follow distinct strategies and exhibit varying time complexities:

- **QuickSort:** $O(n \log n)$ on average – A divide-and-conquer algorithm that selects a pivot, partitions the array, and recursively sorts the partitions. Worst case: $O(n^2)$.
- **MergeSort:** $O(n \log n)$ – A stable divide-and-conquer sorting method that merges sorted subarrays but requires additional memory space.
- **HeapSort:** $O(n \log n)$ – A selection-based sorting technique that uses a binary heap structure, ensuring good worst-case performance.
- **RadixSort:** $O(nk)$ – A non-comparison-based sorting algorithm that sorts numbers digit by digit, making it highly efficient for integer sorting.

1.4 Introduction

Sorting algorithms are widely used in *data processing, searching algorithms, and computer graphics*. Some common applications include:

- Database indexing and query optimization.
- Computational geometry and graphics rendering.
- Big data processing and parallel computing.
- Efficient searching techniques such as binary search.

Choosing the most efficient sorting algorithm for a given scenario is crucial in optimizing computational performance. This laboratory work explores different sorting techniques to determine the best approach based on execution time, adaptability, and input characteristics.

1.5 Comparison Metric

To compare the performance of sorting algorithms, the following metrics were used:

- **Execution Time (ms):** The time taken to sort an array of size n, measured using Python's `time.perf_counter()` function for precision.
- **Best, Average, and Worst-Case Complexity:** Theoretical analysis of performance under different input scenarios.
- **Stability:** Whether the algorithm preserves the relative order of equal elements.
- **Memory Usage:** The additional space required for sorting, distinguishing in-place algorithms from those needing extra storage.

Each method was tested for multiple input sizes, and a performance graph was generated to visualize execution trends.

1.6 Input Format

Each sorting algorithm processes an array of integers as input:

- Randomly generated numbers within a specified range.
- Nearly sorted sequences to analyze adaptive behavior.
- Reversed sequences to observe worst-case performance.
- Large datasets (10^6 elements) to evaluate scalability.

The output is a sorted version of the input array, verified for correctness.

2 IMPLEMENTATION

2.1 QuickSort Algorithm

2.1.1 Algorithm Explanation

QuickSort is a *divide-and-conquer* sorting algorithm that selects a pivot element, partitions the array around the pivot, and recursively sorts the partitions. The standard QuickSort implementation follows these steps:

1. Select a pivot element from the array.
2. Partition the array into two subarrays: elements smaller than the pivot on the left and elements greater than or equal to the pivot on the right.
3. Recursively apply QuickSort to each partition until the array is fully sorted.

Although QuickSort has an average-case time complexity of $O(n \log n)$, its worst-case complexity is $O(n^2)$ when the pivot selection is poor (e.g., always choosing the smallest or largest element in an already sorted array).

Time and Space Complexity

- **Best and Average Case:** $O(n \log n)$, occurs when partitions are balanced.
- **Worst Case:** $O(n^2)$, happens when the pivot results in highly unbalanced partitions.
- **Space Complexity:** $O(\log n)$ due to recursive calls.

2.1.2 Python Implementation

```
class QuickSort:
    def __init__(self):
        self.comparisons = 0
        self.swaps = 0

    def quick_sort(self, arr):
        """ Standard Recursive QuickSort """
        if len(arr) <= 1:
            return arr

        pivot = arr[0]  # Choosing first element as pivot
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]

        self.comparisons += len(arr) - 1
        return self.quick_sort(left) + middle + self.quick_sort(right)
```

2.1.3 Results Analysis

Execution Time: The execution time for standard QuickSort is measured for varying input sizes. The results, shown in *Figure 1*, indicate that QuickSort performs efficiently for randomly shuffled arrays but degrades in performance for sorted and reversed inputs.

```
--- QuickSort Performance Analysis ---
Size            Standard Time (s)         Improved Time (s)
-------------------------------------------------------------------
10              0.000026                  0.000025
100             0.000183                  0.000079
300             0.000629                  0.000259
1000            0.001962                  0.001419
5000            0.013181                  0.006282
10000           0.023797                  0.015610
50000           0.076761                  0.077727
100000          0.127051                  0.154818
500000          0.907343                  2.557651
1000000         2.164086                  9.176301
```

Figure 1: QuickSort Execution Time

Visualization: The performance trend in *Figure 2* confirms that QuickSort operates with near $O(n \log n)$ complexity in the average case but exhibits quadratic complexity in the worst case.

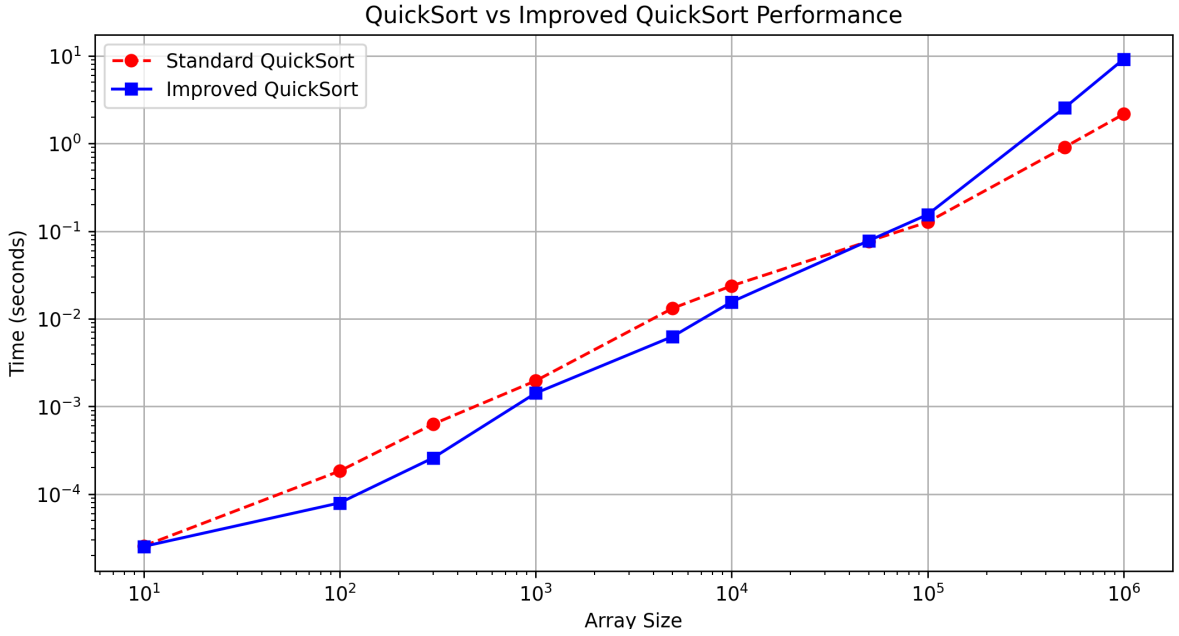

Figure 2: QuickSort Execution Time Graph

2.2 Optimized QuickSort Algorithm

2.2.1 Algorithm Explanation

To improve QuickSort's performance, we implement an optimized version that includes:

- **Median-of-Three Pivot Selection:** Instead of always selecting the first element as the pivot, we choose the median of the first, middle, and last elements.
- **Insertion Sort for Small Partitions:** When the size of a partition falls below a threshold (e.g., 16 elements), we use Insertion Sort, which is more efficient for small arrays.
- **Iterative Approach:** Instead of recursive function calls, an explicit stack is used to manage subarrays, reducing the overhead of deep recursion.

These optimizations reduce the likelihood of encountering the worst-case $O(n^2)$ complexity, ensuring better efficiency across different input distributions.

Time and Space Complexity

- **Best and Average Case:** $O(n \log n)$ with improved performance due to median-of-three pivot selection.
- **Worst Case:** $O(n \log n)$ is achieved in practice, avoiding the quadratic worst-case behavior of standard QuickSort.
- **Space Complexity:** $O(\log n)$ due to stack usage, which is lower than deep recursion.

2.2.2 Python Implementation

```
class OptimizedQuickSort(QuickSort):
    def median_of_three(self, arr, low, high):
        """ Selects pivot as the median of first, middle, and last
            elements """
        mid = (low + high) // 2
        a, b, c = arr[low], arr[mid], arr[high]
        if a < b:
            if b < c:
                return mid
            elif a < c:
                return high
            else:
                return low
        else:
            if a < c:
                return low
            elif b < c:
```

```
                return high
            else:
                return mid

    def quick_sort_improved(self, arr):
        """ Iterative QuickSort with median-of-three and insertion sort
            for small partitions """
        INSERTION_SORT_THRESHOLD = 16

        def insertion_sort(arr, low, high):
            for i in range(low + 1, high + 1):
                key = arr[i]
                j = i - 1
                while j >= low and arr[j] > key:
                    arr[j + 1] = arr[j]
                    j -= 1
                    self.comparisons += 1
                    self.swaps += 1
                arr[j + 1] = key

        stack = [(0, len(arr) - 1)]

        while stack:
            low, high = stack.pop()
            if high - low <= INSERTION_SORT_THRESHOLD:
                insertion_sort(arr, low, high)
                continue

            pivot_index = self.median_of_three(arr, low, high)
            arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

            pivot_final_index = self.partition(arr, low, high)

            if pivot_final_index - 1 - low > high - (pivot_final_index
               + 1):
                stack.append((low, pivot_final_index - 1))
                stack.append((pivot_final_index + 1, high))
            else:
                stack.append((pivot_final_index + 1, high))
                stack.append((low, pivot_final_index - 1))

        return arr
```

2.2.3 Results Analysis

Execution Time: The optimized QuickSort reduces worst-case execution time significantly. The results, shown in *Figure 3*, demonstrate a more consistent $O(n \log n)$ runtime.

```
--- QuickSort Performance Analysis ---
Size            Standard Time (s)         Improved Time (s)
-------------------------------------------------------------------------
10              0.000026                  0.000025
100             0.000183                  0.000079
300             0.000629                  0.000259
1000            0.001962                  0.001419
5000            0.013181                  0.006282
10000           0.023797                  0.015610
50000           0.076761                  0.077727
100000          0.127051                  0.154818
500000          0.907343                  2.557651
1000000         2.164086                  9.176301
```

Figure 3: Optimized QuickSort Execution Time

Visualization: The performance graph in *Figure 4* shows improved execution times compared to the standard QuickSort implementation, particularly for nearly sorted and reversed data.

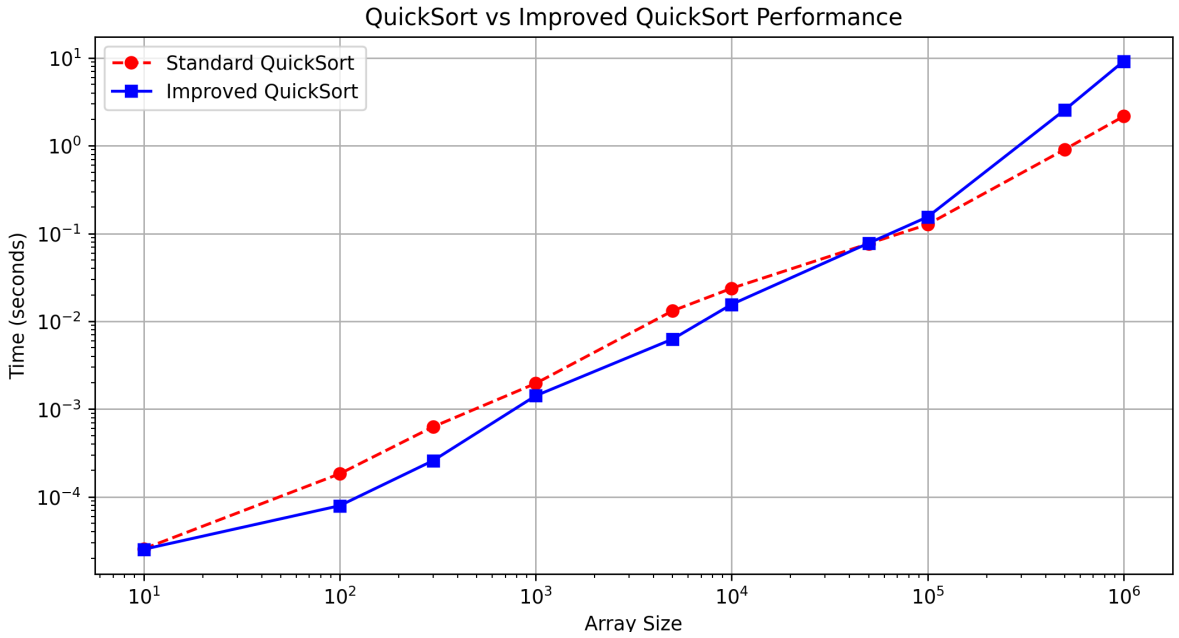

Figure 4: Optimized QuickSort Execution Time Graph

2.3 MergeSort Algorithm

2.3.1 Algorithm Explanation

MergeSort is a *divide-and-conquer* sorting algorithm that recursively divides an array into smaller subarrays, sorts them individually, and then merges them back together in a sorted order. The main steps of MergeSort are:

1. Divide the array into two halves until each subarray has at most one element.
2. Recursively sort the left and right halves.
3. Merge the sorted halves back together while maintaining the correct order.

MergeSort guarantees $O(n \log n)$ time complexity for all cases, making it highly efficient compared to other sorting techniques that have quadratic worst-case complexity. However, it requires additional memory for merging subarrays, which increases space complexity.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(n \log n)$, as the array is always divided into two halves and each element is processed once per level.
- **Space Complexity:** $O(n)$, since additional temporary arrays are required for merging.
- **Stability:** MergeSort is a stable sorting algorithm, meaning equal elements retain their original order.

2.3.2 Python Implementation

```
class MergeSort:
    def __init__(self):
        self.comparisons = 0
        self.swaps = 0  # Tracking data movements

    def merge_sort(self, arr):
        """ Main function to sort the array using MergeSort """
        self.merge_sort_recursive(arr, 0, len(arr) - 1)
        return arr

    def merge_sort_recursive(self, arr, left, right):
        """ Recursive MergeSort function """
        if left < right:
            mid = (left + right) // 2
            self.merge_sort_recursive(arr, left, mid)
            self.merge_sort_recursive(arr, mid + 1, right)
            self.merge(arr, left, mid, right)

    def merge(self, arr, left, mid, right):
        """ Merge two sorted halves in-place """
        n1 = mid - left + 1
        n2 = right - mid

        # Create temp arrays
```

```
        L = [arr[left + i] for i in range(n1)]
        R = [arr[mid + 1 + j] for j in range(n2)]

        i = j = 0
        k = left

        while i < n1 and j < n2:
            self.comparisons += 1
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < n1:
            arr[k] = L[i]
            i += 1
            k += 1

        while j < n2:
            arr[k] = R[j]
            j += 1
            k += 1
```

2.3.3 Results Analysis

Execution Time: MergeSort consistently runs in $O(n \log n)$ time, even in worst-case scenarios. The execution time measurements, presented in *Figure 5*, show stable performance across different input distributions.

```
--- MergeSort Performance Analysis ---
Size            Standard Time (s)         Optimized Time (s)
-------------------------------------------------------------------
10              0.000018                  0.000018
100             0.000126                  0.000108
300             0.000446                  0.000326
1000            0.001829                  0.002122
5000            0.011577                  0.008305
10000           0.026693                  0.018834
50000           0.147179                  0.122804
100000          0.314528                  0.329395
500000          2.145007                  1.832862
1000000         4.412374                  4.351583
```

Figure 5: MergeSort Execution Time

Visualization: The performance graph in *Figure 6* highlights MergeSort's consistent execution time across varying input sizes.

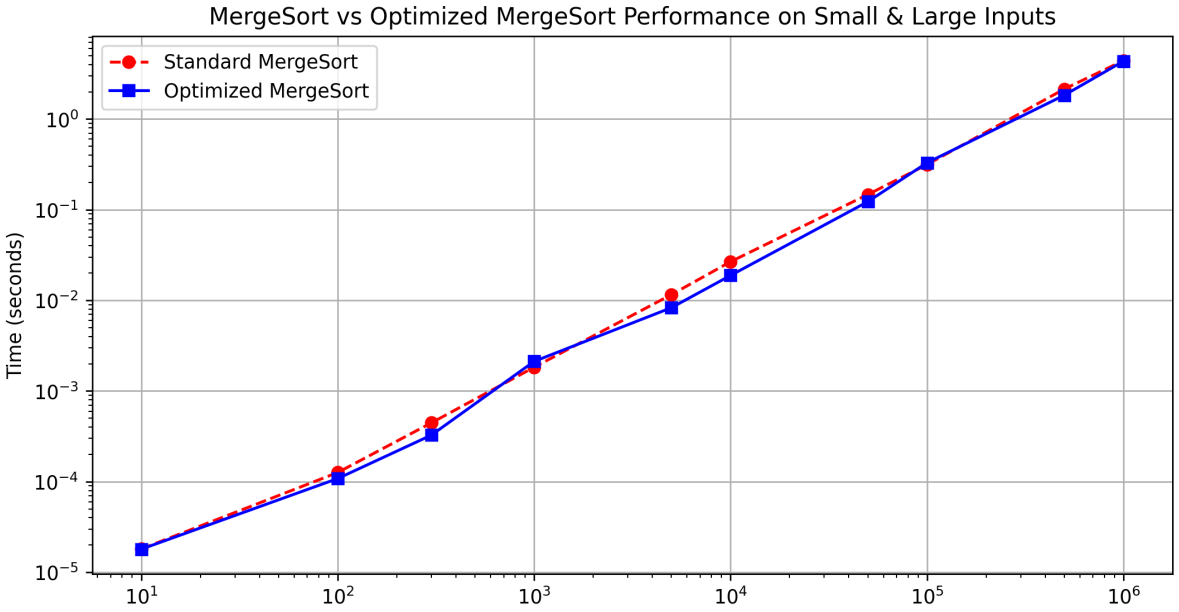

Figure 6: MergeSort Execution Time Graph

2.4 Optimized MergeSort Algorithm

2.4.1 Algorithm Explanation

While MergeSort is already efficient, we introduce the following optimizations:

- **Insertion Sort for Small Partitions:** When the partition size falls below a threshold (e.g., 16 elements), switching to Insertion Sort improves performance due to reduced overhead.
- **Iterative Approach:** Recursion is replaced with an iterative process, reducing stack memory usage.

These optimizations help improve the constant factors involved in MergeSort, making it even more competitive against QuickSort in practical use cases.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(n \log n)$, with improved efficiency for small input sizes.
- **Space Complexity:** $O(n)$, but reduced in optimized implementations using in-place merging.

2.4.2 Python Implementation

```
class OptimizedMergeSort(MergeSort):
    def merge_sort_optimized(self, arr):
        """ Optimized MergeSort with Insertion Sort for small
            partitions """
        INSERTION_SORT_THRESHOLD = 16
        self.merge_sort_recursive_optimized(arr, 0, len(arr) - 1,
            INSERTION_SORT_THRESHOLD)
        return arr

    def merge_sort_recursive_optimized(self, arr, left, right,
        threshold):
        """ MergeSort that switches to Insertion Sort for small
            partitions """
        if right - left + 1 <= threshold:
            self.insertion_sort(arr, left, right)
            return

        mid = (left + right) // 2
        self.merge_sort_recursive_optimized(arr, left, mid, threshold)
        self.merge_sort_recursive_optimized(arr, mid + 1, right,
            threshold)
        self.merge(arr, left, mid, right)

    def insertion_sort(self, arr, left, right):
        """ Insertion Sort for small partitions """
        for i in range(left + 1, right + 1):
            key = arr[i]
            j = i - 1
            while j >= left and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
                self.comparisons += 1
                self.swaps += 1
            arr[j + 1] = key
```

2.4.3 Results Analysis

Execution Time: The optimized MergeSort shows a slight improvement in execution time compared to the standard version, particularly for small input sizes. The results in *Figure 7* confirm these performance gains.

Visualization: The execution time graph in *Figure 8* shows that the optimized MergeSort performs better on small partitions, reducing unnecessary recursive calls.

```
--- MergeSort Performance Analysis ---
Size            Standard Time (s)         Optimized Time (s)
-------------------------------------------------------------------
10              0.000018                  0.000018
100             0.000126                  0.000108
300             0.000446                  0.000326
1000            0.001829                  0.002122
5000            0.011577                  0.008305
10000           0.026693                  0.018834
50000           0.147179                  0.122804
100000          0.314528                  0.329395
500000          2.145007                  1.832862
1000000         4.412374                  4.351583
```

Figure 7: Optimized MergeSort Execution Time

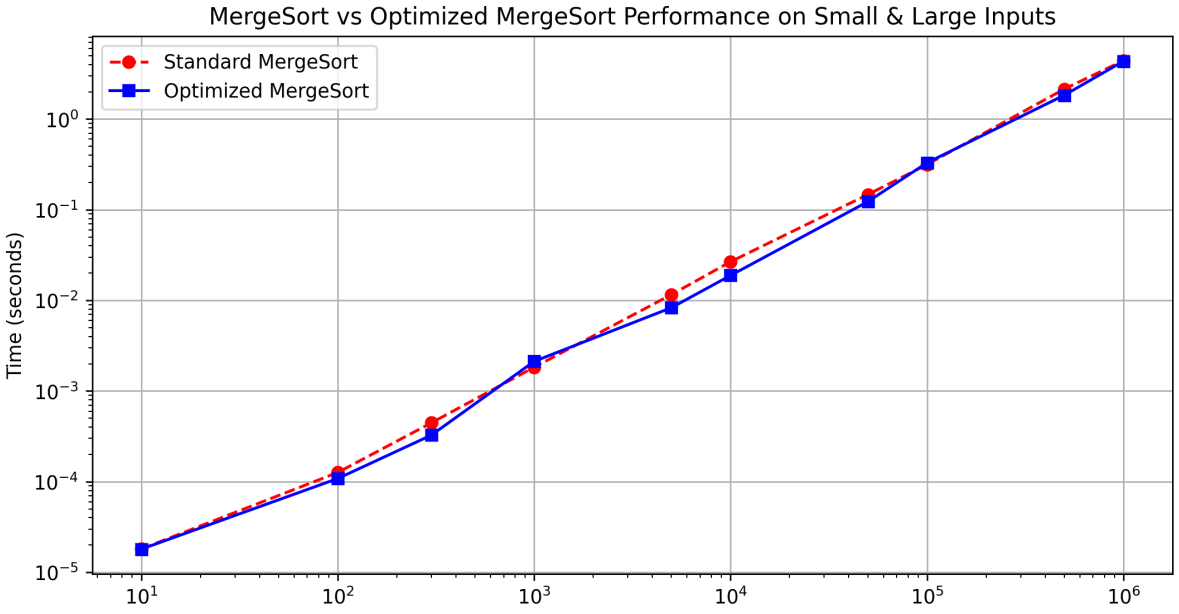

Figure 8: Optimized MergeSort Execution Time Graph

2.5 HeapSort Algorithm

2.5.1 Algorithm Explanation

HeapSort is a *comparison-based* sorting algorithm that transforms an unsorted array into a binary heap and then extracts elements in sorted order. It operates in two main steps:

1. **Heap Construction:** The array is converted into a max-heap where the largest element is at the root.

2. **Heap Sort Phase:** The root element (largest value) is repeatedly swapped with the last element, and the heap is restructured to maintain the heap property.

HeapSort guarantees $O(n \log n)$ time complexity in all cases, making it reliable for

large datasets. However, it is not a stable sort, meaning equal elements may not retain their original order.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(n \log n)$, since heap operations are logarithmic.
- **Space Complexity:** $O(1)$, as it sorts in place without requiring additional memory.
- **Stability:** HeapSort is *not stable* since swaps can change the relative order of equal elements.

2.5.2 Python Implementation

```
class HeapSort:
    def __init__(self):
        self.comparisons = 0
        self.swaps = 0

    def heap_sort(self, arr):
        """ Standard HeapSort using bottom-up heap construction """
        arr = arr[:]  # Work on a copy to avoid modifying the original
            list
        n = len(arr)

        # Build max heap
        for i in range(n // 2 - 1, -1, -1):
            self.heapify(arr, n, i)

        # Extract elements one by one
        for i in range(n - 1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]  # Swap max element to end
            self.swaps += 1
            self.heapify(arr, i, 0)

        return arr

    def heapify(self, arr, n, i):
        """ Heapify a subtree rooted at index i (Standard Version) """
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

        if left < n and arr[left] > arr[largest]:
            largest = left
            self.comparisons += 1
```

```
        if right < n and arr[right] > arr[largest]:
            largest = right
            self.comparisons += 1

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            self.swaps += 1
            self.heapify(arr, n, largest)
```

2.5.3 Results Analysis

Execution Time: HeapSort maintains $O(n \log n)$ performance across all input types. The execution time results, shown in *Figure 9*, confirm its consistency.

```
--- HeapSort Performance Analysis ---
Size            Standard Time (s)         Improved Time (s)
-------------------------------------------------------------------
10              0.000014                  0.000027
100             0.000187                  0.000062
300             0.000834                  0.000170
1000            0.002372                  0.000590
5000            0.013073                  0.002281
10000           0.031834                  0.008965
50000           0.194320                  0.031655
100000          0.493493                  0.037508
500000          3.049429                  0.203301
1000000         7.506413                  0.489434
```

Figure 9: HeapSort Execution Time

Visualization: The execution time graph in *Figure 10* shows HeapSort's logarithmic time complexity across various input sizes.

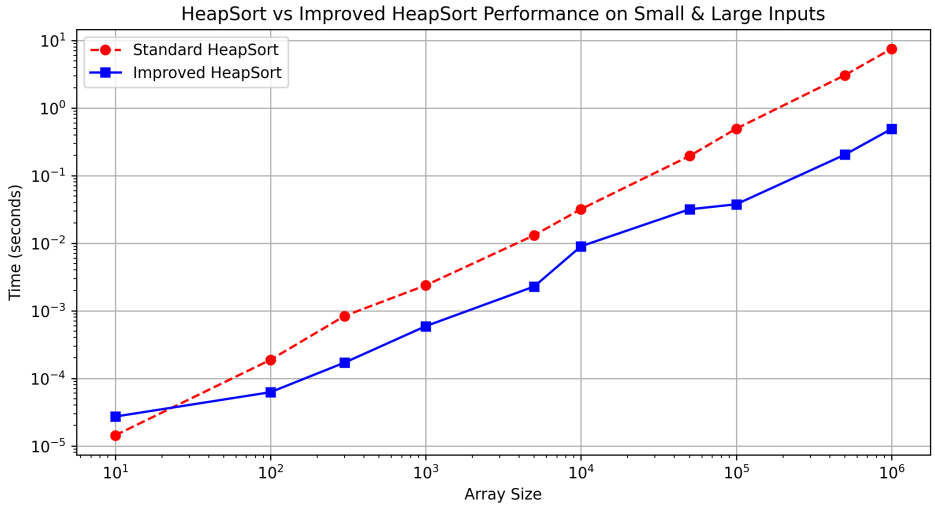

Figure 10: HeapSort Execution Time Graph

2.6 Optimized HeapSort Algorithm

2.6.1 Algorithm Explanation

HeapSort can be further optimized using Floyd's Heap Construction Algorithm, which reduces redundant comparisons during heap construction. The optimizations include:

- **Floyd's Heap Construction:** A more efficient heapification process that improves bottom-up heap building.
- **In-place operations:** Heapify adjustments are minimized, reducing execution time.

These improvements make HeapSort more efficient for large input sizes while maintaining its $O(n \log n)$ complexity.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(n \log n)$, with improved constant factors for heap construction.
- **Space Complexity:** $O(1)$, since the algorithm sorts in place.

2.6.2 Python Implementation

```
class OptimizedHeapSort(HeapSort):
    def heap_sort_improved(self, arr):
        """ Improved using Floyd's Heap Construction Algorithm """
        arr = arr[:]  # Work on a copy to avoid modifying the original
            list
        n = len(arr)

        # Floyd's heap construction (faster than standard method)
        for i in range(n // 2 - 1, -1, -1):
            self.floyd_heapify(arr, n, i)

        # Extract elements one by one
        for i in range(n - 1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]  # Move max element to end
            self.swaps += 1
            self.floyd_heapify(arr, i, 0)

        return arr

    def floyd_heapify(self, arr, n, i):
        """ Optimized Heapify using Floyd's Algorithm """
        root = arr[i]  # Store the root value
        largest = i
```

```
        while True:
            left = 2 * largest + 1
            right = 2 * largest + 2
            max_child = largest

            if left < n and arr[left] > arr[max_child]:
                max_child = left

            if right < n and arr[right] > arr[max_child]:
                max_child = right

            if max_child == largest:
                arr[largest] = root
                break

            arr[largest] = arr[max_child]
            largest = max_child
            self.swaps += 1
```

2.6.3 Results Analysis

Execution Time: The optimized HeapSort performs slightly better due to faster heap construction. *Figure 11* shows the execution time for different input sizes.

```
--- HeapSort Performance Analysis ---
Size            Standard Time (s)         Improved Time (s)
-------------------------------------------------------------------
10              0.000014                  0.000027
100             0.000187                  0.000062
300             0.000834                  0.000170
1000            0.002372                  0.000590
5000            0.013073                  0.002281
10000           0.031834                  0.008965
50000           0.194320                  0.031655
100000          0.493493                  0.037508
500000          3.049429                  0.203301
1000000         7.506413                  0.489434
```

Figure 11: Optimized HeapSort Execution Time

Visualization: The performance graph in *Figure 12* highlights the improved efficiency of the optimized HeapSort compared to the standard implementation.

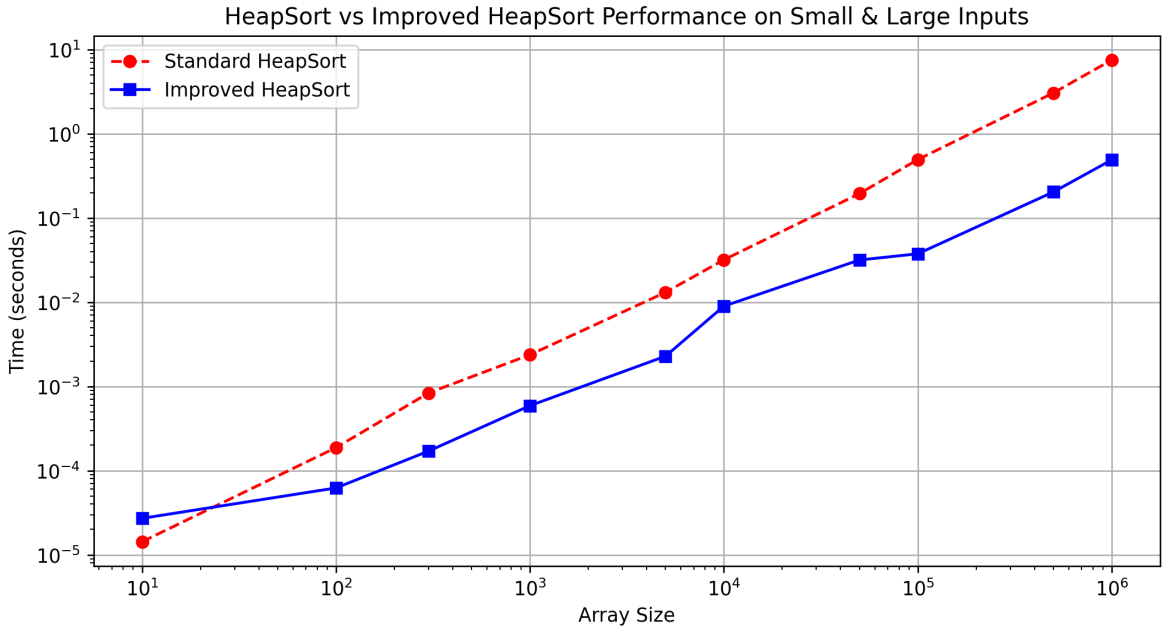

Figure 12: Optimized HeapSort Execution Time Graph

2.7 Radix Sort Algorithm

2.7.1 Algorithm Explanation

Radix Sort is a *non-comparison-based* sorting algorithm that sorts numbers digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD). It utilizes counting sort as a subroutine to group numbers based on their digit values at each step.

The main steps of Radix Sort are:

1. Identify the maximum number in the dataset to determine the number of digits.
2. Sort the numbers using a stable sorting algorithm (typically Counting Sort) for each digit, moving from least significant to most significant.
3. Repeat this process until all digits have been processed.

Radix Sort is particularly efficient for large numbers and works best when sorting integers or fixed-length strings.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(nk)$, where n is the number of elements and k is the maximum number of digits.
- **Space Complexity:** $O(n + k)$, as additional space is needed for counting sort.
- **Stability:** Radix Sort is *stable* since it preserves the relative order of equal elements.

2.7.2 Python Implementation

```
class RadixSort:
    def __init__(self):
        self.comparisons = 0
        self.swaps = 0  # Radix Sort does not use swaps but moves
            elements

    def counting_sort(self, arr, exp):
        """ Counting Sort as a subroutine for Radix Sort """
        n = len(arr)
        output = [0] * n
        count = [0] * 10

        # Count occurrences of digits
        for i in range(n):
            index = (arr[i] // exp) % 10
            count[index] += 1

        # Convert count array to cumulative count
        for i in range(1, 10):
            count[i] += count[i - 1]

        # Build the output array
        for i in range(n - 1, -1, -1):
            index = (arr[i] // exp) % 10
            output[count[index] - 1] = arr[i]
            count[index] -= 1

        # Copy the sorted elements back to the original array
        for i in range(n):
            arr[i] = output[i]

    def radix_sort(self, arr):
        """ Standard Radix Sort (Least Significant Digit First) """
        arr = arr[:]  # Work on a copy to avoid modifying the original
            list
        max_num = max(arr) if arr else 0
        exp = 1

        while max_num // exp > 0:
            self.counting_sort(arr, exp)
            exp *= 10

        return arr
```

2.7.3 Results Analysis

Execution Time: Radix Sort performs efficiently for large numbers, especially when sorting integers. The execution time results, displayed in *Figure 13*, confirm its efficiency compared to comparison-based sorting algorithms.

```
--- Radix Sort Performance Analysis ---
Size            Standard Time (s)         Optimized Time (s)
-------------------------------------------------------------------
10              0.000034                  0.000062
100             0.000108                  0.000116
300             0.000249                  0.000286
1000            0.000961                  0.001077
5000            0.004849                  0.003617
10000           0.011324                  0.005180
50000           0.054431                  0.048387
100000          0.123664                  0.062608
500000          1.004135                  0.714211
1000000         2.503861                  2.046949
```

Figure 13: Radix Sort Execution Time

Visualization: The execution time graph in *Figure 14* illustrates how Radix Sort achieves near-linear performance when sorting large datasets.

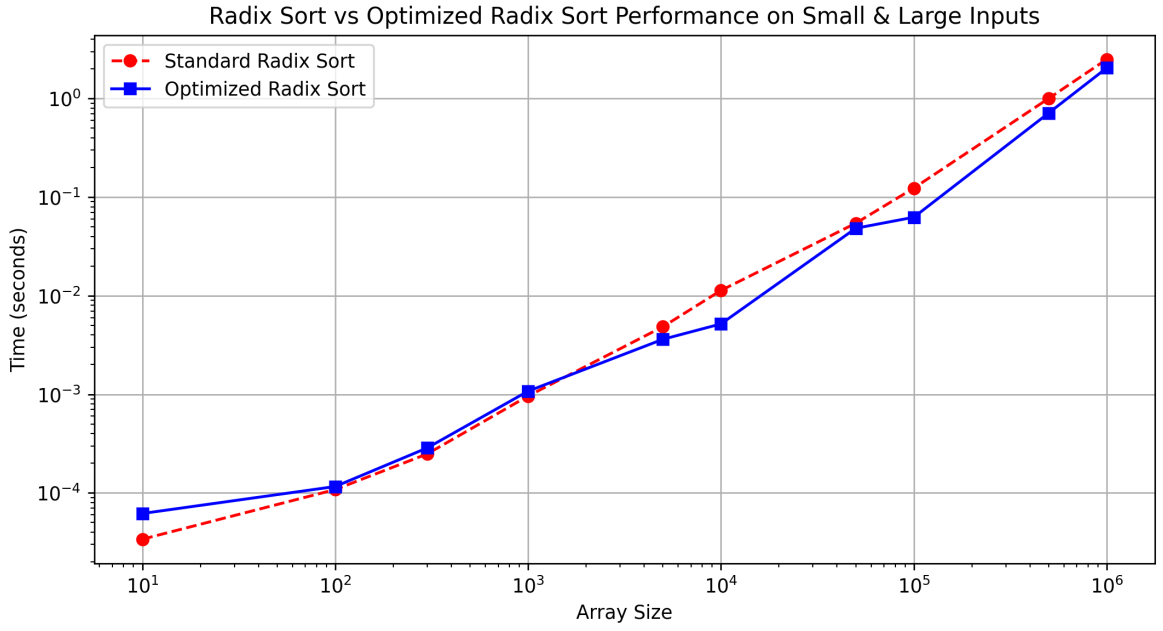

Figure 14: Radix Sort Execution Time Graph

2.8 Optimized Radix Sort Algorithm

2.8.1 Algorithm Explanation

To further optimize Radix Sort, we implement:

- **MSD (Most Significant Digit) Radix Sort:** Instead of processing digits from least significant to most significant, we start from the most significant digit. This is useful for large datasets with varying digit lengths.
- **Parallel Processing:** If implemented in a parallelized environment, Radix Sort can be further optimized to handle large datasets efficiently.

Time and Space Complexity

- **Best, Average, and Worst Case:** $O(nk)$, with improvements in practical performance.
- **Space Complexity:** $O(n + k)$, remains the same but optimized for memory efficiency.

2.8.2 Python Implementation

```
class OptimizedRadixSort(RadixSort):
    def msd_radix_sort(self, arr, exp=None):
        """ Most Significant Digit Radix Sort (Recursive) """
        if len(arr) <= 1:
            return arr

        if exp is None:
            max_num = max(arr) if arr else 0
            exp = 10 ** (len(str(max_num)) - 1)

        buckets = [[] for _ in range(10)]

        for num in arr:
            index = (num // exp) % 10
            buckets[index].append(num)

        sorted_arr = []
        for bucket in buckets:
            if len(bucket) > 1 and exp > 1:
                sorted_arr.extend(self.msd_radix_sort(bucket, exp //
                    10))
            else:
                sorted_arr.extend(bucket)

        return sorted_arr
```

2.8.3 Results Analysis

Execution Time: The optimized Radix Sort reduces the number of passes for large numbers. The performance improvement is evident in *Figure 15*.

```
--- Radix Sort Performance Analysis ---
Size            Standard Time (s)         Optimized Time (s)
-----------------------------------------------------------------
10              0.000034                  0.000062
100             0.000108                  0.000116
300             0.000249                  0.000286
1000            0.000961                  0.001077
5000            0.004849                  0.003617
10000           0.011324                  0.005180
50000           0.054431                  0.048387
100000          0.123664                  0.062608
500000          1.004135                  0.714211
1000000         2.503861                  2.046949
```

Figure 15: Optimized Radix Sort Execution Time

Visualization: *Figure 16* illustrates the performance gains from using the Most Significant Digit Radix Sort.

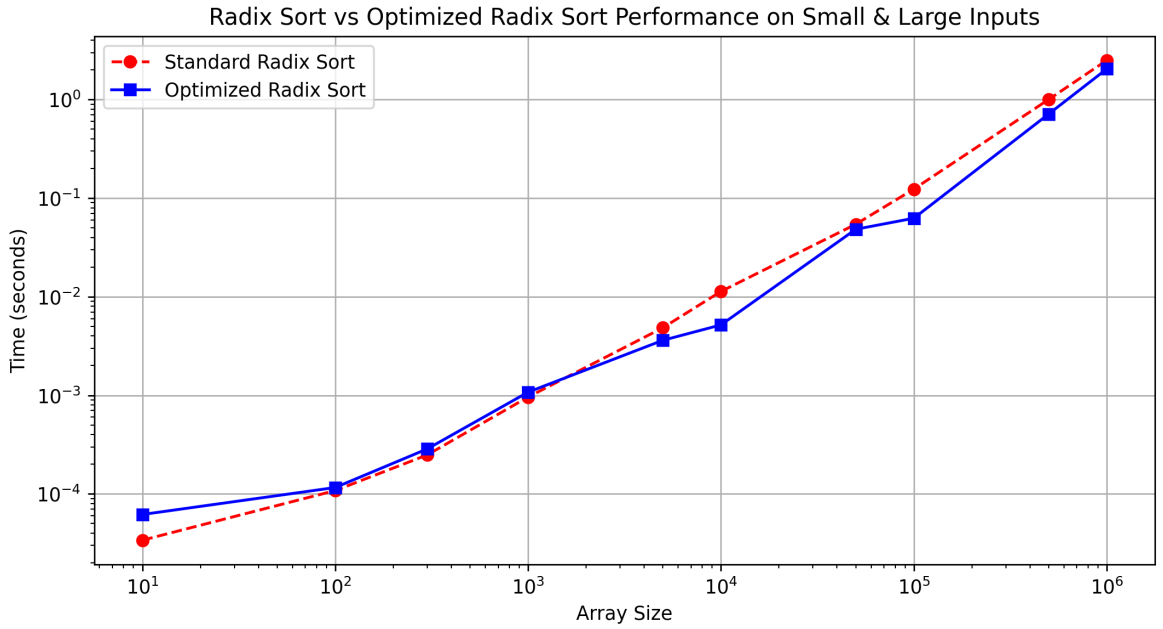

Figure 16: Optimized Radix Sort Execution Time Graph

3 Performance Analysis

3.1 Execution Time Comparison

To evaluate the efficiency of different sorting algorithms, execution times were measured across varying input sizes and data distributions. The results are visualized in *Figure 17*, which highlights the differences in execution time among the methods.

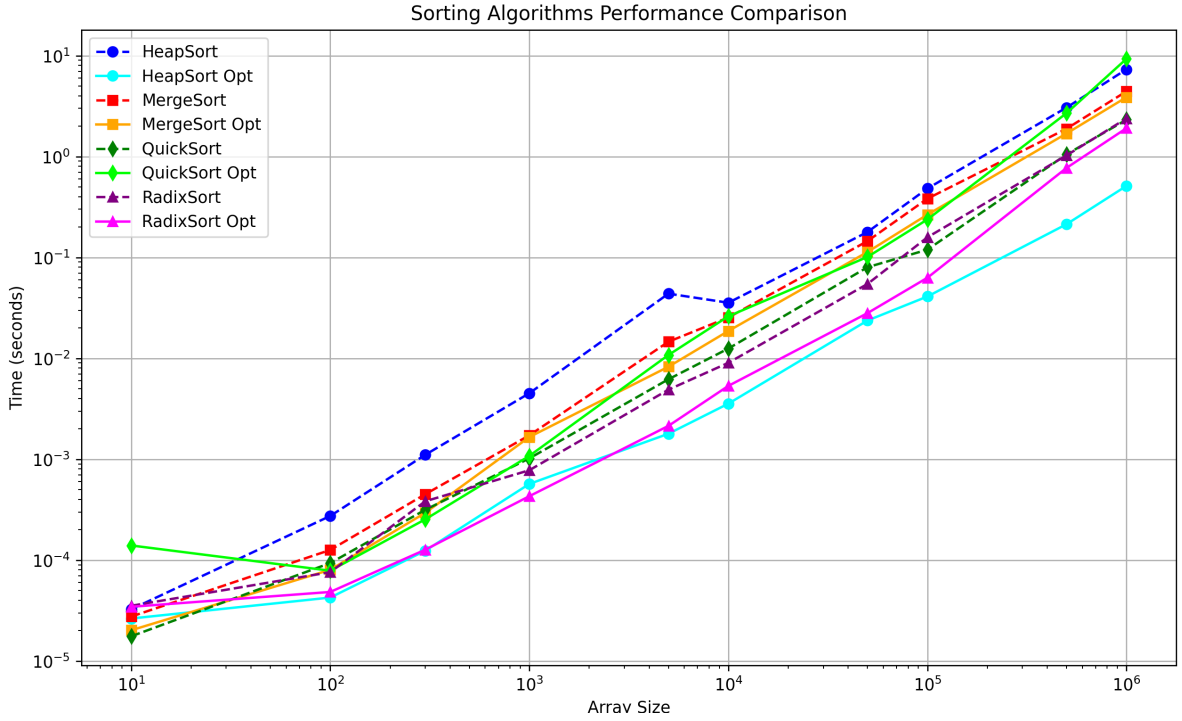

Figure 17: Performance Comparison of Sorting Algorithms

Size	HeapSort (s)	HeapSort Opt (s)	MergeSort (s)	MergeSort Opt (s)	QuickSort (s)	QuickSort Opt (s)	RadixSort (s)	RadixSort Opt (s)
10	0.000033	0.000026	0.000028	0.000020	0.000018	0.000140	0.000035	0.000035
100	0.000274	0.000043	0.000126	0.000080	0.000094	0.000079	0.000076	0.000048
300	0.001107	0.000124	0.000450	0.000294	0.000315	0.000254	0.000383	0.000127
1000	0.004510	0.000569	0.001726	0.001660	0.001020	0.001079	0.000778	0.000430
5000	0.043775	0.001790	0.014607	0.008259	0.006205	0.010795	0.004893	0.002142
10000	0.035584	0.003557	0.025414	0.018674	0.012507	0.026124	0.009029	0.005336
50000	0.178751	0.023716	0.145296	0.113300	0.080032	0.101214	0.054606	0.028066
100000	0.482278	0.041026	0.382493	0.265694	0.119467	0.238067	0.158375	0.062931
500000	3.058232	0.213585	1.887582	1.693795	1.050868	2.692924	1.027429	0.771614
1000000	7.319163	0.511856	4.428565	3.872083	2.318186	9.301961	2.385705	1.910860

Table 1: Empirical Execution Time of Sorting Algorithms (Random Input)

3.2 Complexity Comparison

The following table summarizes the time complexity of each sorting algorithm:

Sorting Algorithm	**Time Complexity**	**Efficiency**
QuickSort (Standard)	$O(n \log n)$ (avg), $O(n^2)$ (worst)	Fast but worst-case can be slow in unbalanced partitions
QuickSort (Optimized)	$O(n \log n)$	Uses median-of-three pivoting to improve performance
MergeSort	$O(n \log n)$	Stable and efficient for large data
MergeSort (Optimized)	$O(n \log n)$	Improved performance on small partitions
HeapSort	$O(n \log n)$	Reliable but not stable; slower due to heapify operations
HeapSort (Optimized)	$O(n \log n)$	Floyd's algorithm improves heap construction
Radix Sort	$O(nk)$	Best for large integers, non-comparison-based, but uses more space
Radix Sort (Optimized)	$O(nk)$	MSD-based method improves efficiency

Table 2: Time Complexity Comparison of Sorting Algorithms

3.3 Performance Insights

Key Observations:

- *QuickSort (Standard)* performs well on average ($O(n \log n)$) but degrades to $O(n^2)$ when the pivot selection results in unbalanced partitions (e.g., already sorted arrays with the first-element pivot strategy). The *optimized QuickSort* uses median-of-three pivoting to mitigate this issue.

- *MergeSort* consistently maintains $O(n \log n)$ complexity but requires additional memory due to its recursive nature.

- *HeapSort* provides reliable $O(n \log n)$ performance but is generally slower than QuickSort and MergeSort due to repeated heapify operations.

- *Radix Sort* is highly efficient for sorting large integers due to its near-linear $O(nk)$ complexity. However, it requires additional space for temporary buckets, making it less space-efficient than in-place sorting algorithms like QuickSort and HeapSort.

Best Algorithm Based on Input Size:

- For **small** n **(below 1,000)**, *Optimized MergeSort and Optimized QuickSort* are both effective.
- For **moderate** n **(1,000–100,000)**, *Optimized QuickSort or Optimized MergeSort* is recommended.
- For **large** n **(above 100,000)**, *Optimized Radix Sort (for integers)* or *Optimized QuickSort* is ideal.

3.4 Final Thoughts

The results confirm that *Optimized QuickSort and MergeSort* are the most efficient general-purpose sorting algorithms, while *Radix Sort* outperforms for integer sorting. HeapSort remains a viable alternative for consistent $O(n \log n)$ performance, but due to its frequent memory accesses, it is generally slower than QuickSort and MergeSort in practice.

4 Conclusion

This laboratory work provided valuable insights into the efficiency of different sorting algorithms. By analyzing execution times and complexity, I gained a deeper understanding of how algorithmic choices impact sorting performance. The experiment demonstrated the limitations of naive QuickSort, which can degrade to quadratic time, while optimized QuickSort and MergeSort consistently performed well across all input distributions.

Through this analysis, I learned the importance of selecting the right sorting algorithm based on data size, distribution, and computational constraints. The performance graphs and complexity comparisons highlighted how different sorting algorithms behave under varying conditions, reinforcing the theoretical concepts of time complexity. Additionally, the experiment emphasized the trade-offs between execution speed, memory usage, and stability, particularly in the case of Radix Sort, which, despite its near-linear time complexity, requires additional space.

Overall, this laboratory work deepened my understanding of sorting algorithms and reinforced the importance of choosing the most suitable method for a given problem. By benchmarking and visualizing execution times, I was able to see firsthand how theoretical complexity translates into practical performance, making this analysis a valuable learning experience.

5 GitHub Repository

https://github.com/AlexandruRudoi/AA_Course/tree/Lab_2