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

Laboratory work 2: Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, countingSort

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

TABLE OF CONTENTS

Contents

ALGORITHM ANALYSIS **3**
Tasks: 3
Theoretical Notes: 3
Introduction: 5
Comparison Metric: 6
Input Format: 6
IMPLEMENTATION **7**
Quick Sort Algorithm: 7
Heap Sort Algorithm 9
Merge Sort Algorithm 11
Counting Sort Algorithm 13
CONCLUSION **15**
References: 16

ALGORITHM ANALYSIS

Objective

Study and analyze different sorting algorithms.

Tasks:

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical Notes:

Empirical analysis provides an alternative approach to understanding the efficiency of algorithms when mathematical complexity analysis is impractical or insufficient. This method proves beneficial in various scenarios:

1. Initial Insights: It offers preliminary insights into an algorithm's complexity class, aiding in the understanding of its efficiency characteristics.
2. Comparative Analysis: It facilitates the comparison of multiple algorithms tackling the same problem, allowing for informed decisions regarding efficiency.
3. Implementation Comparison: Empirical analysis enables the comparison of different implementations of the same algorithm, providing insights into which may perform better in practice.
4. Hardware-specific Evaluation: It helps in assessing an algorithm's efficiency on a particular computing platform, taking into account hardware constraints and capabilities.

The empirical analysis of an algorithm typically involves the following steps:
Establishing Analysis Goals: Clearly define the objectives and scope of the analysis.

1. Choosing Efficiency Metrics: Select appropriate metrics, such as the number of operations executed or the execution time, based on the analysis goals.
2. Defining Input Data Properties: Determine the characteristics of the input data relevant to the analysis, including data size or specific attributes.
3. Implementation: Develop the algorithm in a programming language, ensuring it accurately reflects the intended logic.
4. Generating Input Data Sets: Create multiple sets of input data to cover a range of scenarios and edge cases.
5. Execution and Data Collection: Execute the program for each input data set,

recording relevant performance metrics.

6. Data Analysis: Analyze the collected data, either by computing synthetic quantities like mean and standard deviation or by plotting graphs to visualize the relationship between problem size and efficiency metrics.
7. The choice of efficiency measure depends on the analysis's objectives. For instance, if assessing complexity class or verifying theoretical estimates, counting the number of operations may be suitable. Conversely, if evaluating algorithm implementation behavior, measuring execution time becomes more relevant.

8. Post-execution, recorded results undergo analysis. This involves computing statistical measures or plotting graphs to visualize the algorithm's performance characteristics in terms of problem size and efficiency metrics. Such analyses aid in making informed decisions regarding algorithm selection and optimization strategies.

Introduction:

QuickSort:

- QuickSort is an internal sorting algorithm based on the divide and conquer strategy.
- Key features:
    - ♦ The array of elements is repeatedly divided into parts until further division is not possible.
    - ♦ It uses a pivot element for partitioning.
    - ♦ Elements smaller than the pivot are placed in one partition, and those greater in another.
- Efficiency:
    - ♦ QuickSort is efficient for smaller array sizes or datasets.
    - ♦ However, its worst-case complexity can be O(n^2) due to excessive comparisons.
    - ♦ It is an in-place sorting method, meaning it doesn't require additional storage space.
- Preferred for: Arrays.

MergeSort:

- MergeSort is an external sorting algorithm also based on divide and conquer.
- Key features:
    - ♦ The array is split into two sub-arrays repeatedly until only one element remains.
    - ♦ MergeSort uses additional storage for sorting (auxiliary arrays).
    - ♦ It recursively sorts each sub-array and then merges them.
- Efficiency:
    - ♦ MergeSort is more efficient for larger array sizes or datasets.
    - ♦ Its worst-case and average-case complexities are both O(n log n).
    - ♦ It is stable, preserving the order of equal elements.

HeapSort:

- HeapSort is a comparison-based sorting algorithm.

- Key features:
    - ♦ It builds a heap (a binary tree) from the array.
    - ♦ The largest element is moved to the end, and the heap is restructured.
    - ♦ This process is repeated until the entire array is sorted.
- Efficiency:
    - ♦ HeapSort is not as efficient as QuickSort or MergeSort.
    - ♦ It is used when memory usage is a concern.
    - ♦ It is unstable.

CountingSort:

- CountingSort is a non-comparison-based sorting algorithm.
- Key features:
    - ♦ It counts the occurrences of each element.
    - ♦ Based on these counts, it constructs the sorted output.
    - ♦ Suitable for integer data with a limited range.
- Efficiency:
    - ♦ CountingSort has a linear time complexity of O(n + k), where k is the range of input values.
    - ♦ It is stable.
    - ♦ Preferred for: Integer data.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of eachalgorithm (T(n))

Input Format:

As input, each algorithm will receive 7 series of numbers of length 1000, 5000, 10000, 20000, 30000, 40000, 50000, 100000. Each series will contain random numbers from range (0, 1000000).

IMPLEMENTATION

All 4 algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar toother experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

Quick Sort Algorithm:

QuickSort is a sorting algorithm based on the Divide and Conquer strategy. The key process in QuickSort is the partition step.It picks an element as a pivot and partitions the given array around this pivot. The goal is to place the pivot in its correct position in the sorted array. All smaller elements are placed to the left of the pivot, and all greater elements to the right. Partitioning is done recursively on each side of the pivot, eventually sorting the entire array.

Implementation:

```
    def partition(self, array, low, high):
        pivot = array[high]
        i = low - 1
        for j in range(low, high):
            if array[j] <= pivot:
                i = i + 1
                (array[i], array[j]) = (array[j], array[i])
        (array[i + 1], array[high]) = (array[high], array[i + 1])
        return i + 1

    def quicksort(self, array, low, high):
        if low < high:
            pi = Algorithms.partition(self, array, low, high)
            Algorithms.quicksort(self, array, low, pi - 1)
            Algorithms.quicksort(self, array, pi + 1, high)
```

Results:

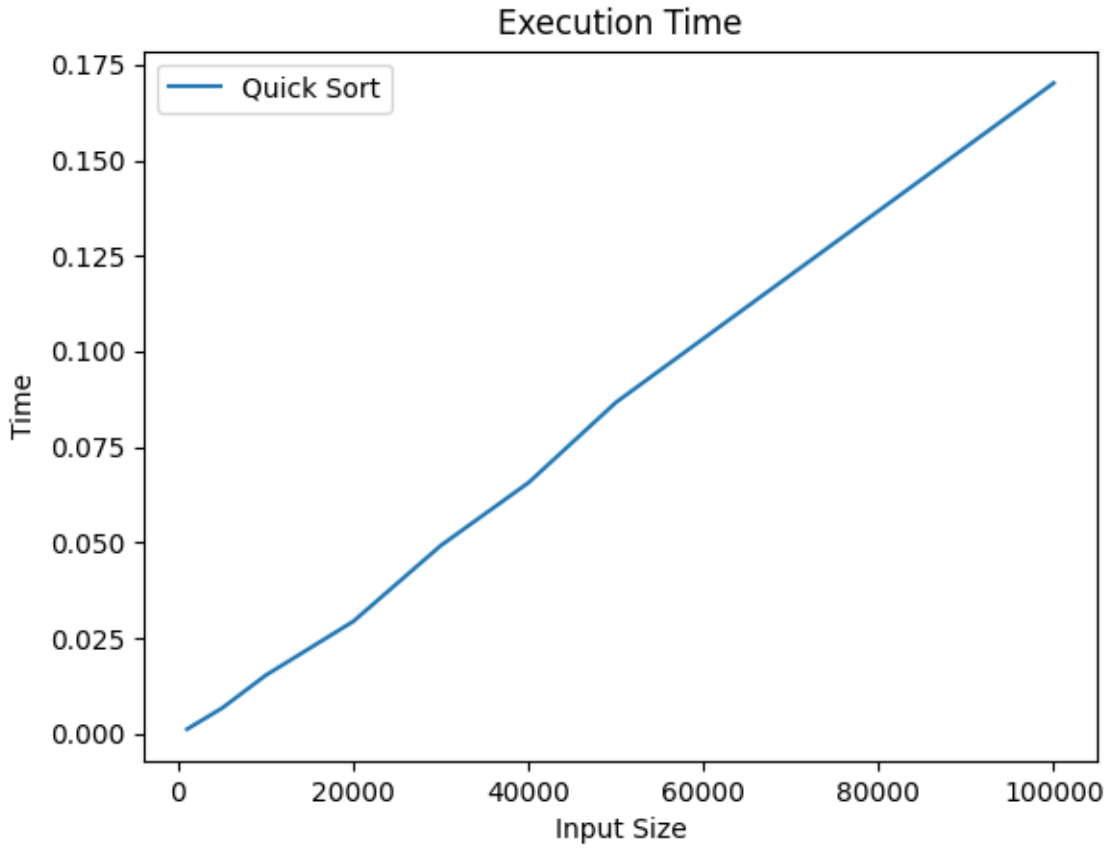

Figure 1 QuickSort Graph Results

	Input Size	QuickSort	HeapSort	MergeSort	CountSort
0	1000	0.001188	0.002965	0.007629	0.111501
1	5000	0.006696	0.020373	0.013551	0.101572
2	10000	0.015302	0.038303	0.024771	0.106595
3	20000	0.029401	0.088412	0.056632	0.154188
4	30000	0.049254	0.134657	0.078330	0.119169
5	40000	0.065668	0.179151	0.101558	0.117173
6	50000	0.086694	0.225556	0.127786	0.114396
7	100000	0.170229	0.472895	0.304942	0.130598

Figure 2 Results for all input data

In Figure 2 is represented the table of results for the inputs. The first column is the input size. Next columns represent the time needed to execute each algorithm. We may notice that the quickSort algorithm is the top 1until 50000 elements and top 2 at sorting 100000 elements.

Heap Sort Algorithm

Heap Sort is a comparison-based sorting technique that operates on a Binary Heap data structure. The key idea is to transform the unsorted array into a max heap (or a min heap, depending on the desired sorting order). A max heap ensures that the parent node is greater than or equal to its child nodes. The largest element (the root of the heap) is repeatedly removed and placed at the end of the array. The remaining elements are then restructured to maintain the max heap property. This process continues until the entire array is sorted.

Implementation:

```
def heapify(self, arr, N, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2

        if l < N and arr[largest] < arr[l]:
            largest = l

        if r < N and arr[largest] < arr[r]:
            largest = r

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]

            Algorithms.heapify(self, arr, N, largest)

    def heap_sort(self, arr):
        N = len(arr)

        for i in range(N//2 - 1, -1, -1):
            Algorithms.heapify(self, arr, N, i)

        for i in range(N-1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]
            Algorithms.heapify(self, arr, i, 0)
```

Results:

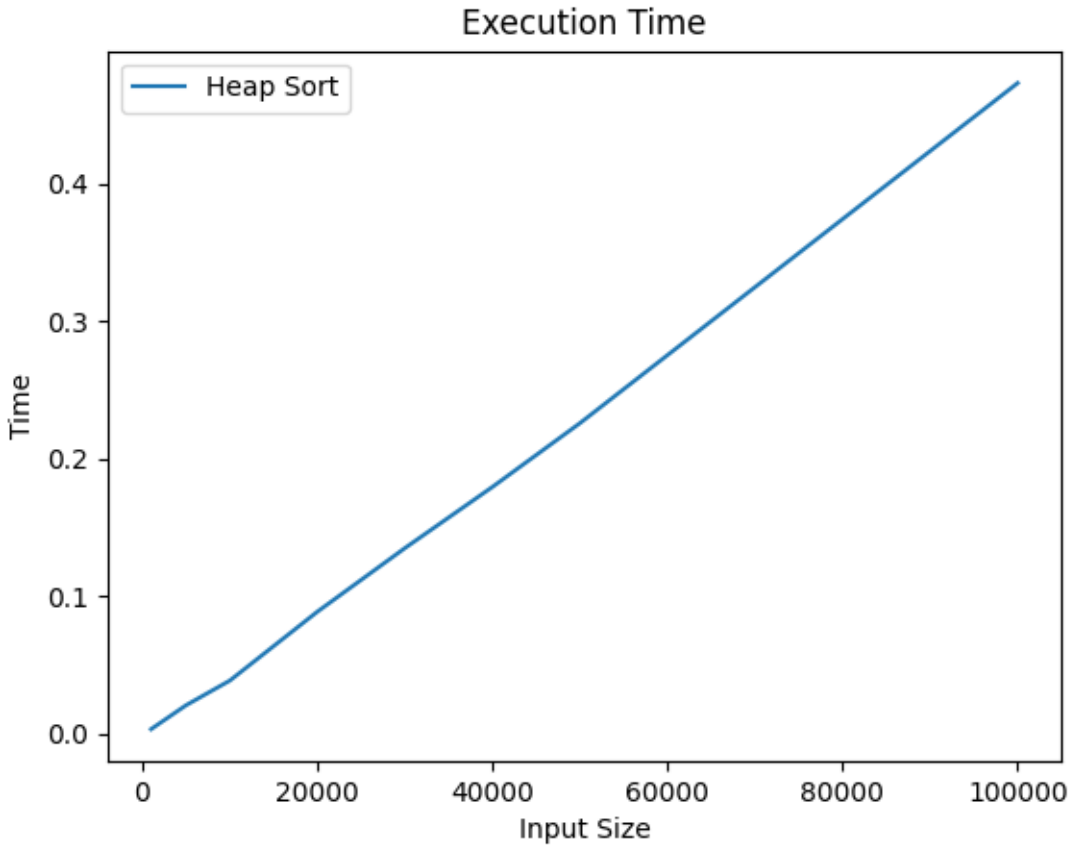

Figure 3 HeapSort Graph Results

	Input Size	QuickSort	HeapSort	MergeSort	CountSort
0	1000	0.001188	0.002965	0.007629	0.111501
1	5000	0.006696	0.020373	0.013551	0.101572
2	10000	0.015302	0.038303	0.024771	0.106595
3	20000	0.029401	0.088412	0.056632	0.154188
4	30000	0.049254	0.134657	0.078330	0.119169
5	40000	0.065668	0.179151	0.101558	0.117173
6	50000	0.086694	0.225556	0.127786	0.114396
7	100000	0.170229	0.472895	0.304942	0.130598

Figure 4 Results for all input data

In Figure 4 is represented the table of results for the inputs. The first column is the input size. Next columns represent the time needed to execute each algorithm. We may notice that the heapSort algorithm more efficient than counting sort till 20000 elements. Per general, heap sort is more inefficient than others.

Merge Sort Algorithm

Merge Sort is a classic example of a ‘divide and conquer’ algorithm. The primary idea behind Merge Sort is to divide the original array into smaller arrays until each smaller array has only one position. Then, it merges these smaller arrays in a sorted manner.

Implementation:

```
def mergeSort(self, arr):
        if len(arr) > 1:
            mid = len(arr)//2

            L = arr[:mid]

            R = arr[mid:]

            Algorithms.mergeSort(self, L)

            Algorithms.mergeSort(self, R)

            i = j = k = 0

            while i < len(L) and j < len(R):
                if L[i] <= R[j]:
                    arr[k] = L[i]
                    i += 1
                else:
                    arr[k] = R[j]
                    j += 1
                k += 1

            while i < len(L):
                arr[k] = L[i]
                i += 1
                k += 1

            while j < len(R):
                arr[k] = R[j]
                j += 1
                k += 1
```

Results:

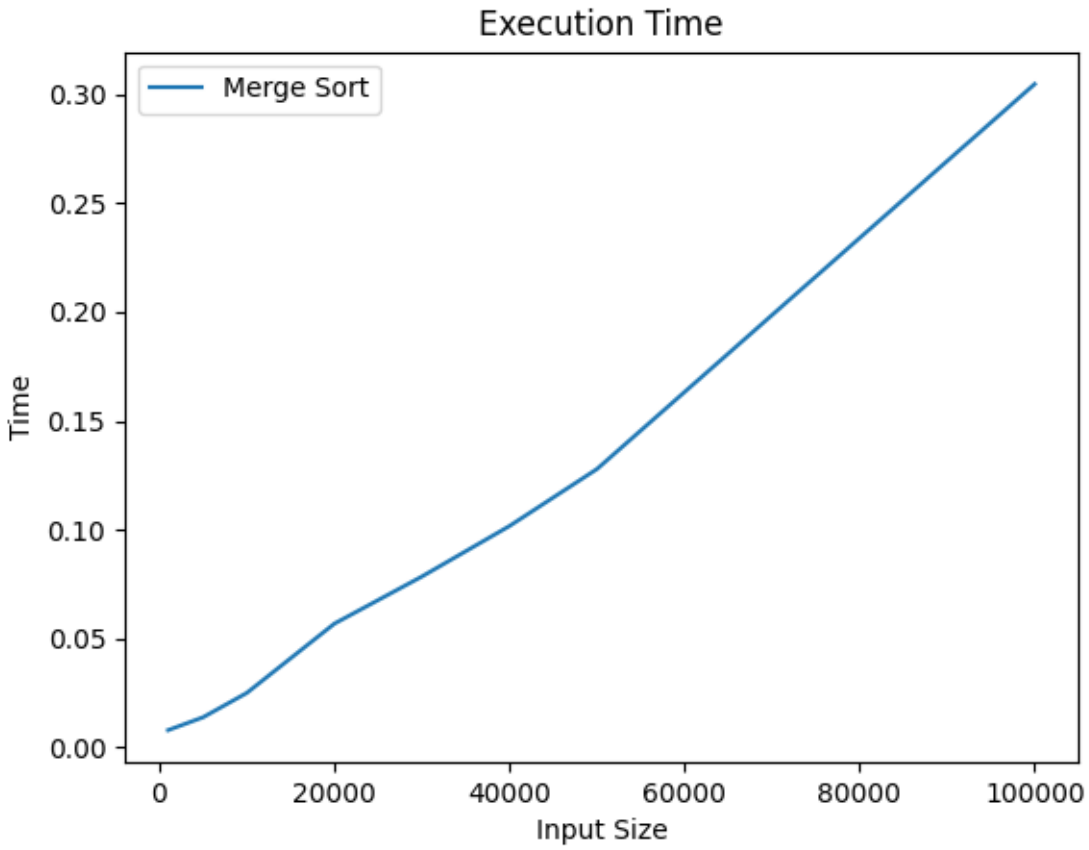

Figure 5 MergeSort Graph Results

	Input Size	QuickSort	HeapSort	MergeSort	CountSort
0	1000	0.001188	0.002965	0.007629	0.111501
1	5000	0.006696	0.020373	0.013551	0.101572
2	10000	0.015302	0.038303	0.024771	0.106595
3	20000	0.029401	0.088412	0.056632	0.154188
4	30000	0.049254	0.134657	0.078330	0.119169
5	40000	0.065668	0.179151	0.101558	0.117173
6	50000	0.086694	0.225556	0.127786	0.114396
7	100000	0.170229	0.472895	0.304942	0.130598

Figure 6 Results for all input data

In Figure 6 is represented the table of results for the inputs. The first column is the input size. Next columns represent the time needed to execute each algorithm. We may notice that the mergeSort algorithm more efficient than counting sort till 40000 elements, but less efficient compared to quickSort algorithm in terms of execution time.

Counting Sort Algorithm

Counting Sort is a non-comparison-based sorting algorithm that works well when there is a limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted.

Implementation:

```
def count_sort(self, input_array):

    M = max(input_array)
    count_array = [0] * (M + 1)

    # Mapping each element of input_array as an index of count_array
    for num in input_array:
        count_array[num] += 1

    # Calculating prefix sum at every index of count_array
    for i in range(1, M + 1):
        count_array[i] += count_array[i - 1]

    # Creating output_array from count_array
    output_array = [0] * len(input_array)

    for i in range(len(input_array) - 1, -1, -1):
        output_array[count_array[input_array[i]] - 1] = input_array[i]
        count_array[input_array[i]] -= 1

    return output_array
```

Results:

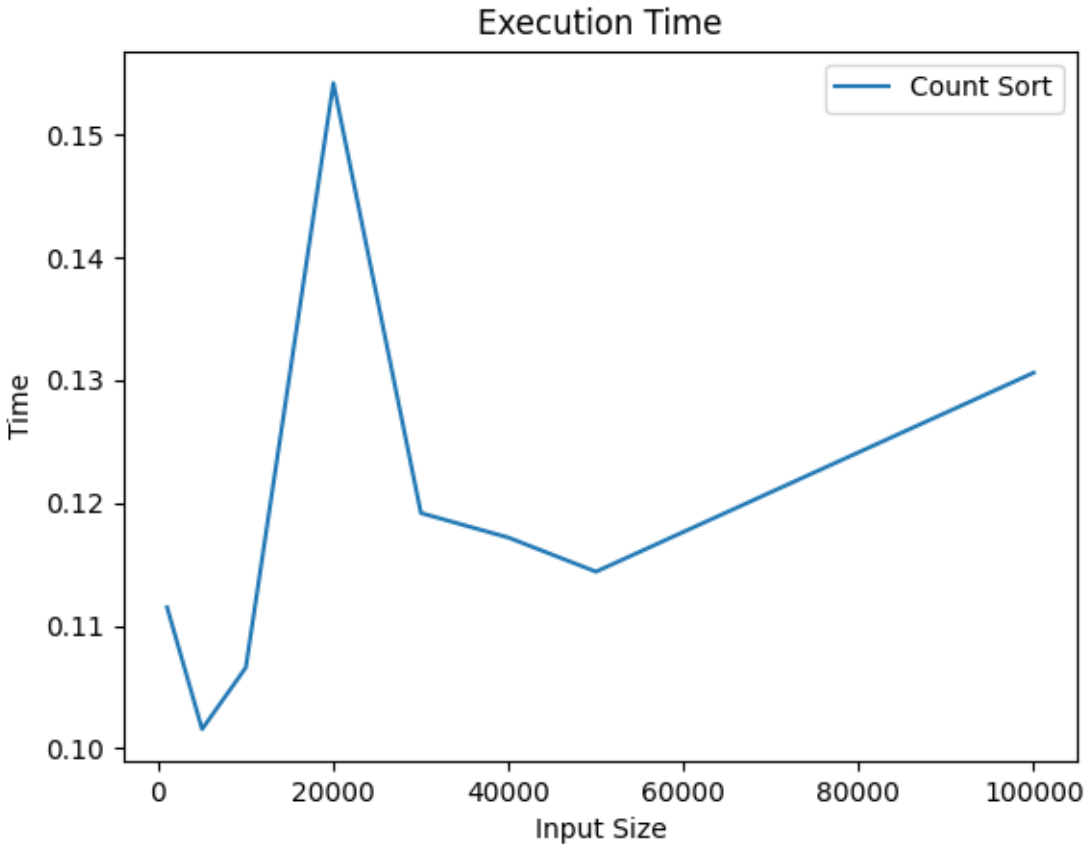

Figure 7 CountingSort Graph Results

	Input Size	QuickSort	HeapSort	MergeSort	CountSort
0	1000	0.001188	0.002965	0.007629	0.111501
1	5000	0.006696	0.020373	0.013551	0.101572
2	10000	0.015302	0.038303	0.024771	0.106595
3	20000	0.029401	0.088412	0.056632	0.154188
4	30000	0.049254	0.134657	0.078330	0.119169
5	40000	0.065668	0.179151	0.101558	0.117173
6	50000	0.086694	0.225556	0.127786	0.114396
7	100000	0.170229	0.472895	0.304942	0.130598

Figure 8 Results for all input data

In Figure 8 is represented the table of results for the inputs. The first column is the input size. Next columns represent the time needed to execute each algorithm. We may notice that the countingSort algorithm more efficient after 40000 elements and it is stable for all input data.

CONCLUSION

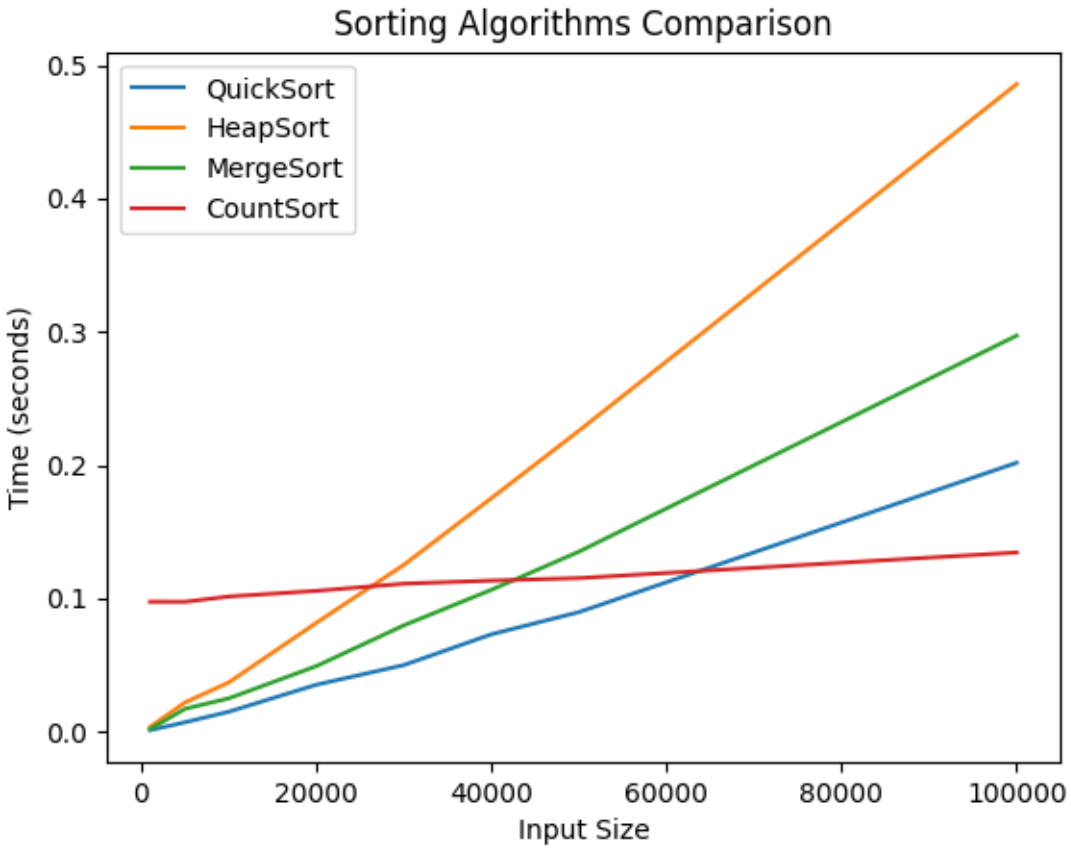

Figure 9 Results for all algorithms

Through comprehensive empirical analysis, this study has delved into the performance and characteristics of four prominent sorting algorithms: quickSort, mergeSort, heapSort, and countingSort. Each algorithm has been rigorously evaluated across various datasets, considering factors such as time complexity, space complexity, and suitability for different input scenarios.

quickSort, renowned for its efficiency and widespread use, exhibits exceptional performance with an average time complexity of O(n log n) in the best and average cases. However, its worst-case time complexity of O(n^2) warrants careful consideration, particularly with highly unbalanced input arrays.

mergeSort, a stable and predictable sorting algorithm, demonstrates consistent performance with a time complexity of O(n log n) across all cases. Its divide-and-conquer approach and efficient merging technique make it a reliable choice for sorting large datasets, although it requires additional space for auxiliary arrays.

heapSort, known for its in-place sorting and optimal worst-case time complexity of O(n log n), offers a practical solution for scenarios where space is a concern. Its inherent stability concerns and relatively slower performance in comparison to quickSort and mergeSort may limit its applicability in certain contexts.

countingSort, a non-comparative integer sorting algorithm, stands out for its linear time

complexity of O(n+k), where k represents the range of input values. It excels when sorting integers within a limited range, making it ideal for specialized use cases such as sorting non-negative integers with a small range.

In conclusion, this analysis provides valuable insights into the strengths and limitations of each sorting algorithm, aiding practitioners in selecting the most suitable algorithm based on the specific requirements of their applications. While quickSort and mergeSort offer robust performance across a wide range of scenarios, heapSort and countingSort cater to distinct use cases with their unique attributes. By understanding the trade-offs associated with each algorithm, developers can make informed decisions to optimize sorting tasks for efficiency and scalability.

References:

1. https://github.com/KaBoomKaBoom/Algorithm-Analysis.git