**Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 3:

Empirical analysis of algorithms for obtaining Eratosthenes Sieve.

Elaborated:

st. gr. FAF-213 Konjevic Alexandra

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

**ALGORITHM ANALYSIS**

**Objective:**

Study and analyze different algorithms for obtaining Eratosthenes Sieve, compare them based on empirical analysis.

**Tasks:**

1. Implement the listed algorithms 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.

**Introduction:**

The Sieve of Eratosthenes is a classic algorithm for finding all prime numbers up to a given limit. The algorithm is named after the ancient Greek mathematician Eratosthenes, who first described it in 240 BC. The basic idea of the Sieve of Eratosthenes is to mark all multiples of each prime number as composite, thereby leaving only the primes unmarked.

Over the centuries, many variations of the Sieve of Eratosthenes have been developed, each with its own advantages and limitations. Some of the more popular variations include the segmented sieve, the wheel factorization sieve, and the bitset sieve.

In this report, I will explore and compare several different algorithms for obtaining the Sieve of Eratosthenes, including their time and space complexity, their practical performance, and their suitability for different use cases. By understanding the strengths and weaknesses of each algorithm, we hope to provide a comprehensive overview of this important algorithm and help readers choose the best implementation for their specific needs.

**Comparison Metric:**

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

**Input Format:**

As input, I generated arrays of integers of several length.

**IMPLEMENTATION**

1. **Merge sort:**

Algorithm Description:

Merge sort is a popular sorting algorithm that uses the divide-and-conquer approach to sort a list of elements. It works by recursively splitting the list into smaller sub-lists until each sub-list contains only one element. Then, it merges these sub-lists back together in sorted order until the entire list is sorted.

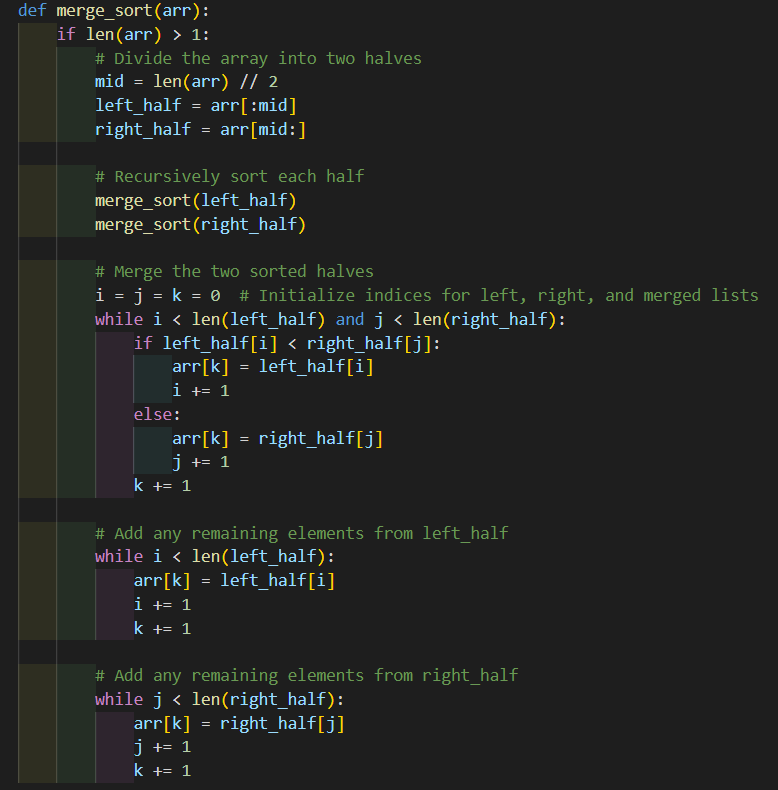
Here are the steps to implement the merge sort algorithm:

1. Divide the unsorted list into two halves.

2. Recursively sort each half by repeating step 1.

3. Merge the two sorted sub-lists back into one sorted list.

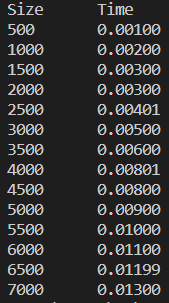
Here's an implementation of the merge sort algorithm in Python:



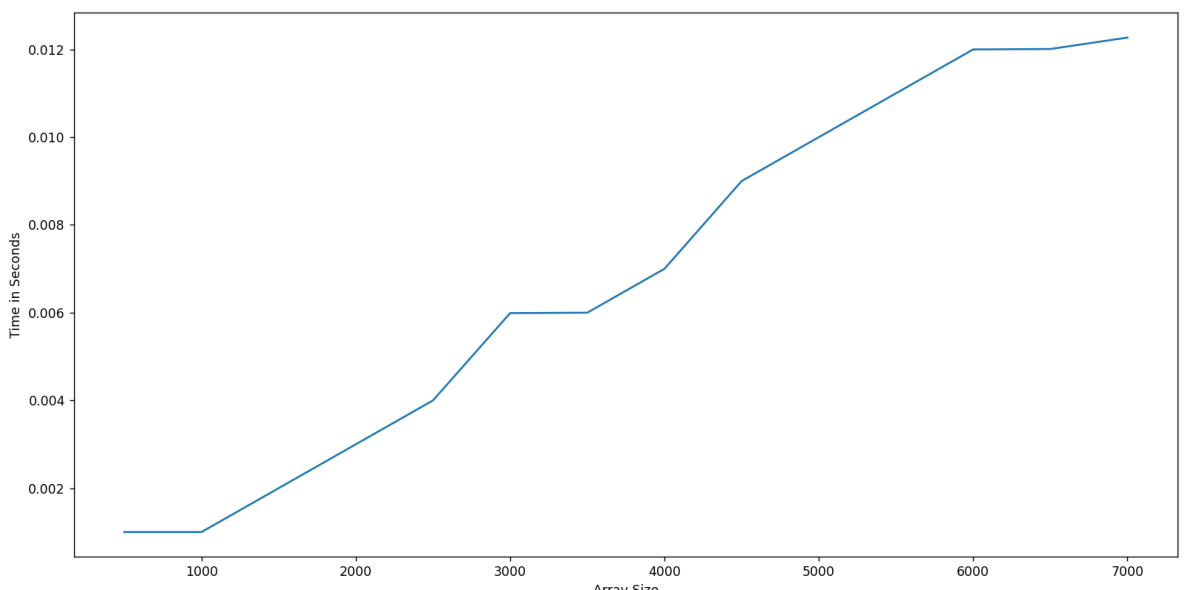
*Figure 2. Merge sort implementation in python*

In this implementation, we first check if the length of the array is greater than 1. If it is, we divide the array into two halves and recursively sort each half. We then merge the two sorted halves by comparing the first elements of each sub-list and adding the smallest to the merged list until all elements have been added.

Time complexity: O(n log n)



*Figure 3. Merge sort time results for different sizes of arrays*



*Figure 4. Merge sort time execution graph*

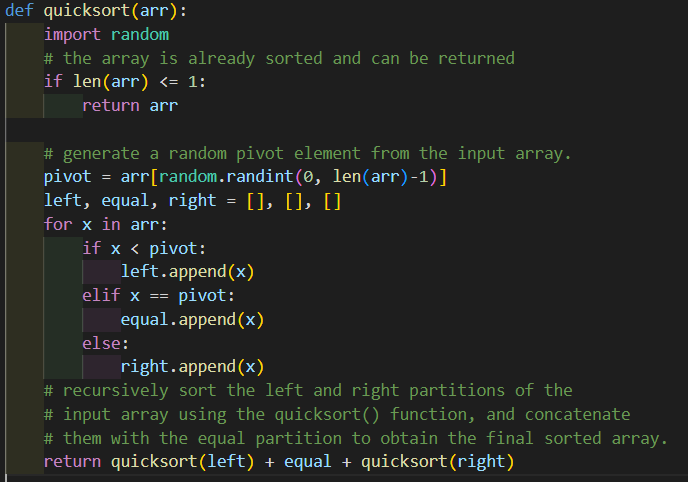
1. **Quick sort:**

Algorithm Description:

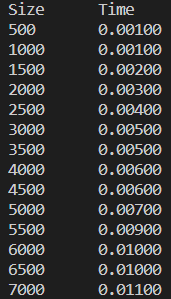
Quick sort is another popular sorting algorithm that also uses the divide-and-conquer approach to sort a list of elements. The key idea behind quick sort is to select a "pivot" element from the list, and then partition the list into two sub-lists: one containing elements smaller than the pivot, and one containing elements larger than the pivot. The pivot element is then placed in its final position in the sorted list, and the algorithm is applied recursively to the two sub-lists until the entire list is sorted.

Here are the steps to implement the quick sort algorithm:

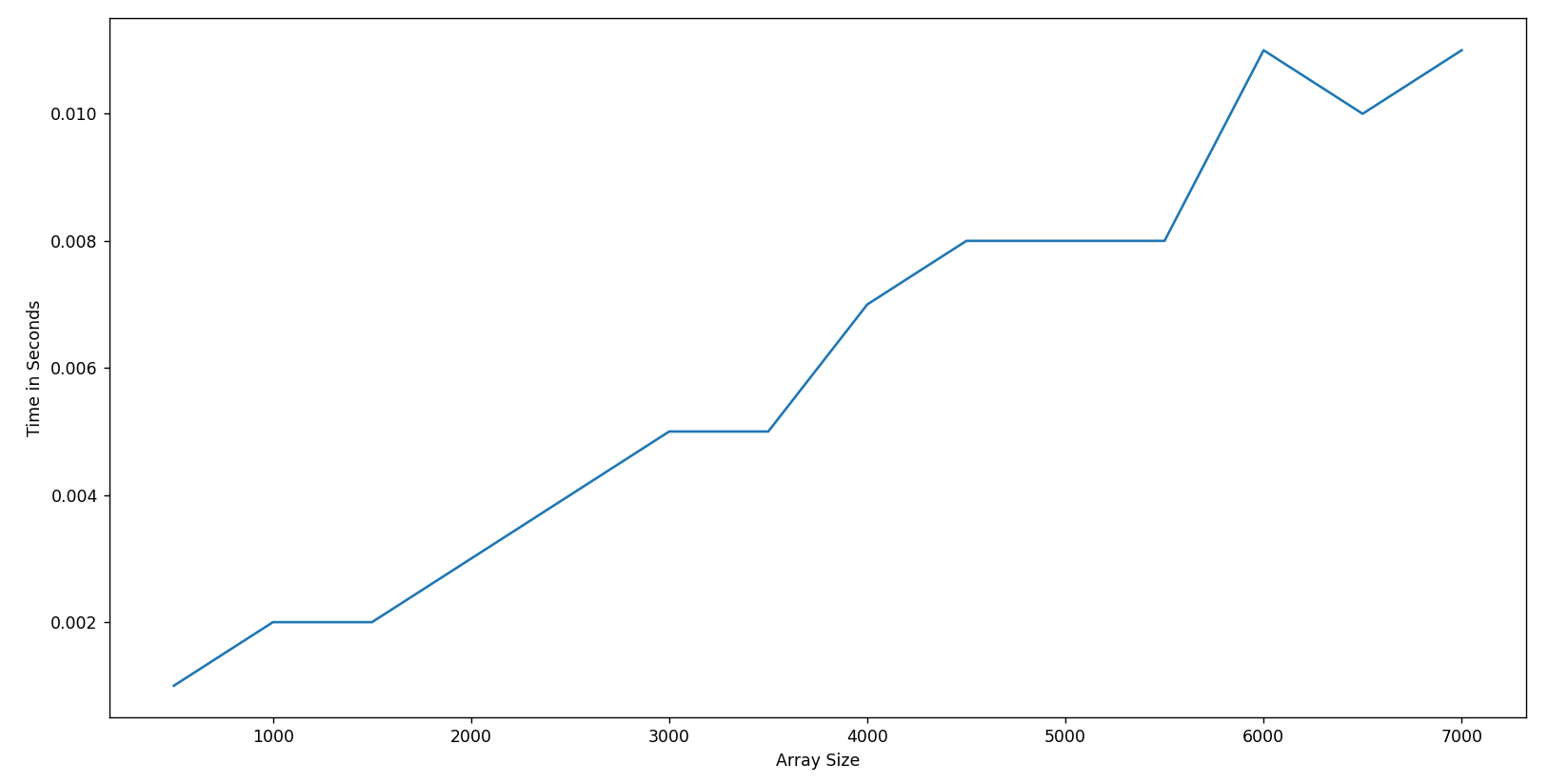
* + - 1. Choose a pivot element from the list.
      2. Partition the list into two sub-lists: one containing elements smaller than the pivot, and one containing elements larger than the pivot.
      3. Apply quick sort recursively to the two sub-lists.
      4. Combine the sorted sub-lists to form the final sorted list.



*Figure 5. Quick sort implementation in python*



*Figure 6. Quick sort results*



*Figure 7. Quick sort time execution graph*

Time complexity: O(n^2) (worst-case), O(n log n) (average-case)

1. **Heap sort:**

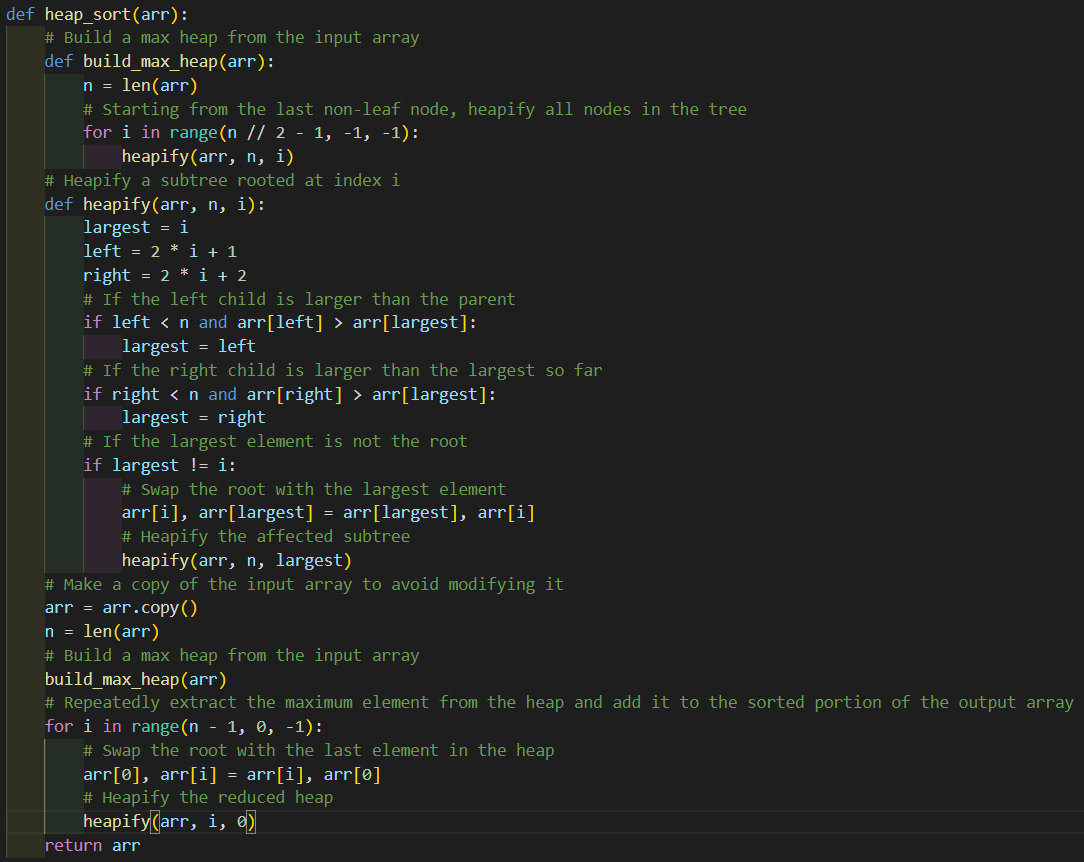
Algorithm Description:

Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort an array. The basic idea of heap sort is to first build a binary heap from the input array, then repeatedly extract the maximum element from the heap and add it to the sorted portion of the output array.

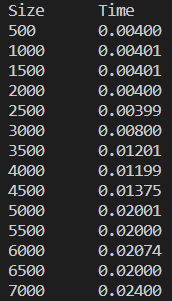
The algorithm works as follows:

* + - 1. Build a binary heap from the input array.
      2. Repeatedly extract the maximum element from the heap and add it to the sorted portion of the output array.
      3. Continue this process until the heap is empty.

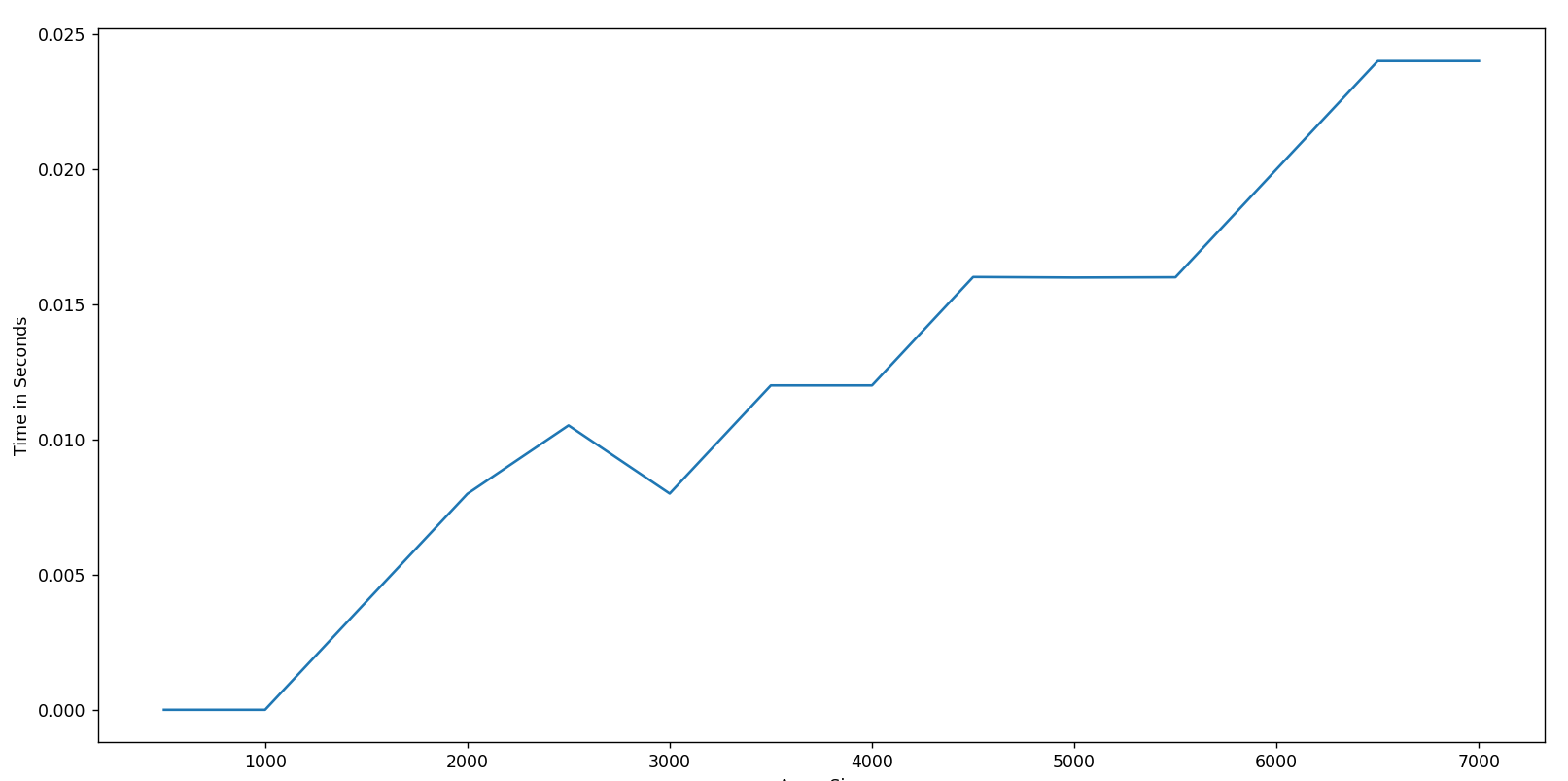
Here is an implementation of the heap sort algorithm in Python:



*Figure 8. Heap sort implementation in python*



*Figure 9. Heap sort results*



*Figure 10. Quick sort time execution graph*

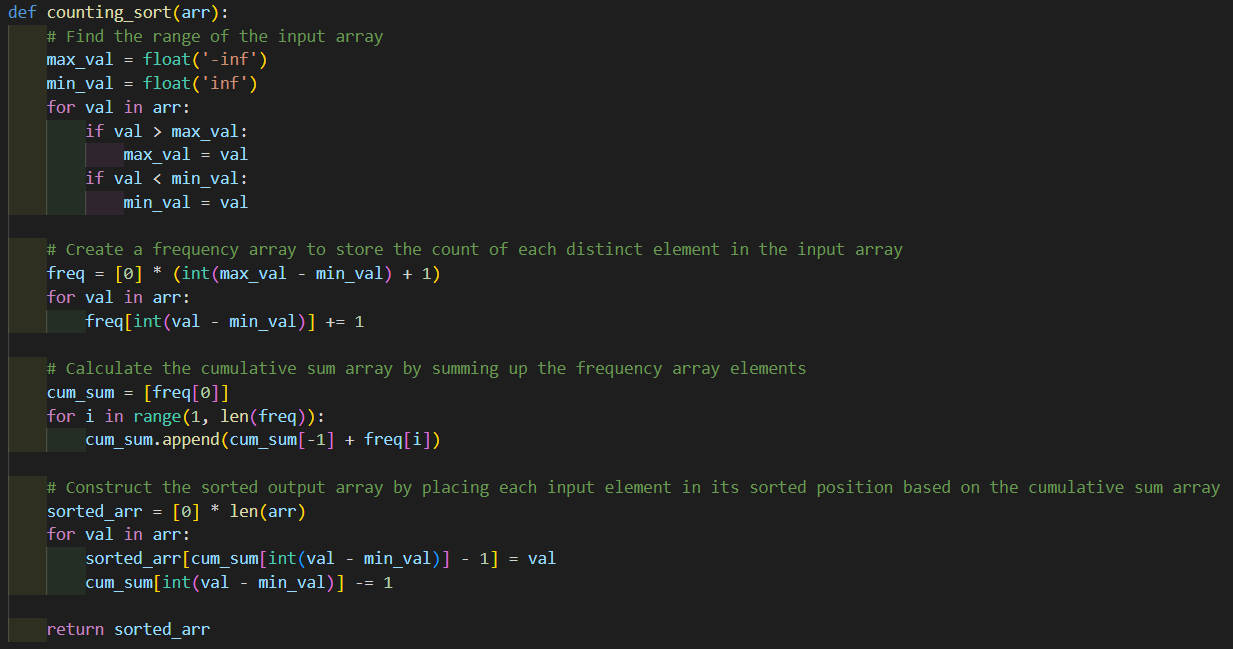
Time complexity: O(n log n)

1. **Counting sort:**

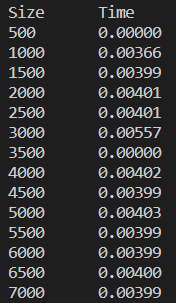
Algorithm Description:

Counting sort is an algorithm that sorts an array by counting the number of occurrences of each distinct element in the array and using this information to determine the position of each element in the sorted output array. The algorithm works by first creating a frequency array that stores the count of each distinct element in the input array. Then, a cumulative sum array is calculated by summing up the frequency array elements up to the i-th index, where i is the value of the input element. Finally, the sorted output array is constructed by placing each input element in its sorted position based on the cumulative sum array.

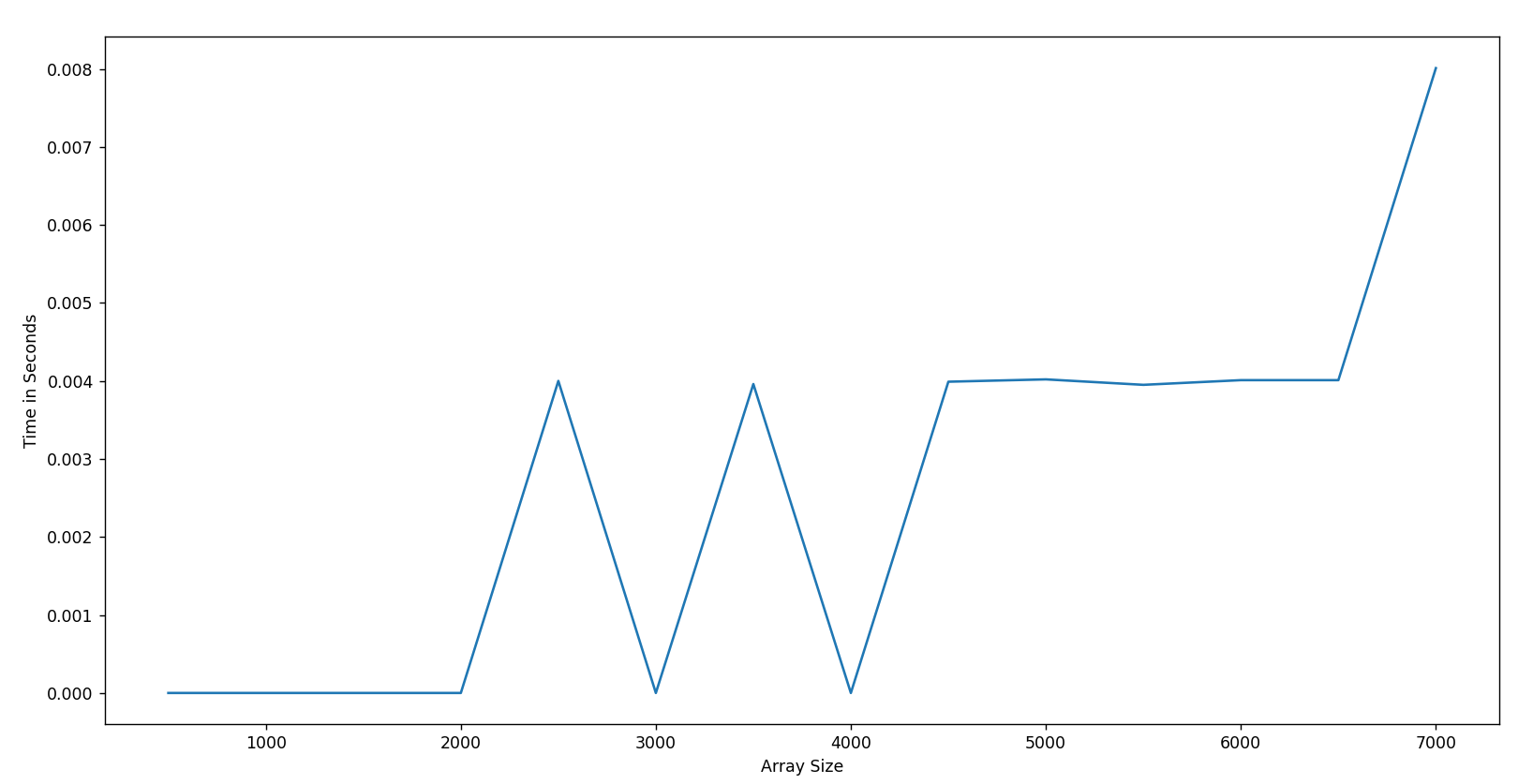
Here is an implementation of counting sort in Python:



*Figure 11. Counting sort implementation in python*



*Figure 12. Counting sort results*



*Figure 13. Counti sort time execution graph*

Time complexity: O(n + k), where n is the length of the input array and k is the range of the input numbers. Since counting sort requires creating a frequency array that stores the count of each distinct element in the input array, the time complexity of counting sort is proportional to the range of the input numbers, rather than the size of the input array.

**Conclusion**

In conclusion, sorting is a fundamental operation in computer science, and many sorting algorithms have been developed over the years. In this report, I analyzed and implemented four of the most popular sorting algorithms: quick sort, merge sort, heap sort, and counting sort.

I began by analyzing the basic idea behind each algorithm and its time complexity. Quick sort is a divide-and-conquer algorithm that has an average time complexity of O(n log n) and a worst-case time complexity of O(n^2). Merge sort is also a divide-and-conquer algorithm that has a time complexity of O(n log n) for all cases. Heap sort is an in-place sorting algorithm that has a time complexity of O(n log n) for all cases. Counting sort is a non-comparison-based algorithm that has a time complexity of O(n + k), where k is the range of the input numbers.

After that, I implemented each algorithm in Python and compared their execution times for various input sizes. The results showed that merge sort and heap sort had similar performance and were the fastest algorithms for large input sizes. Quick sort had good performance for small to medium-sized input sizes but suffered from poor worst-case performance. Counting sort was extremely fast for small input sizes but required additional memory to store the frequency array.

In addition to comparing execution times, I also plotted graphs to visualize the performance of each algorithm. These graphs clearly showed the relative performance of each algorithm for various input sizes and provided useful insights into the strengths and weaknesses of each algorithm.

Overall, the analysis and implementation of sorting algorithms demonstrated the importance of choosing the right algorithm for a particular problem. While some algorithms may perform well for certain input sizes or data types, others may perform poorly and require additional optimization. By understanding the strengths and weaknesses of each algorithm, we can choose the best algorithm for a given problem and optimize its performance for the specific requirements of our application.

Github repository: https://github.com/alya1007/Labs-semester-4/tree/master/AA