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

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 2:

Study and empirical analysis of sorting algorithms.

Elaborated:

st. gr. FAF-213 Konjevic Alexandra

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

**ALGORITHM ANALYSIS**

**Objective:**

Study and analyze different sorting algorithms, compare them based on empirical analysis.

**Tasks:**

1. Implement quickSort, mergeSort, heapSort and another algorithm on your choice.

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:**

Sorting algorithms are fundamental to computer science and are utilized in many applications to organize and analyze data. As a part of this laboratory work, several sorting algorithms were analyzed and implemented. Sorting algorithms play a significant role in many aspects of computer science, from database indexing and file searching to cryptography and scientific simulations. In this report, we will discuss the implementation of several sorting algorithms, including their advantages and disadvantages, performance metrics, and trade-offs. The aim of this laboratory work was to analyze and compare the performance of various sorting algorithms on different datasets and to determine their suitability for various applications. This report will provide an in-depth analysis of the algorithms used, the datasets used, and the results obtained, as well as recommendations for future improvements.

**Comparison Metric:**

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

**Input Format:**

I used arrays with random floats between -10.000 and 10.000. The arrays are generated with a size between 500 and 7.000 numbers, with a step of 500. All the arrays are stored in the “arrays” variable:



*Figure 1. Generating arrays with random numbers and sizes from 500 to 7000*

Thus, all the implemented algorithms will use the same arrays to sort, and the comparison of the execution time will be fairer.

**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.

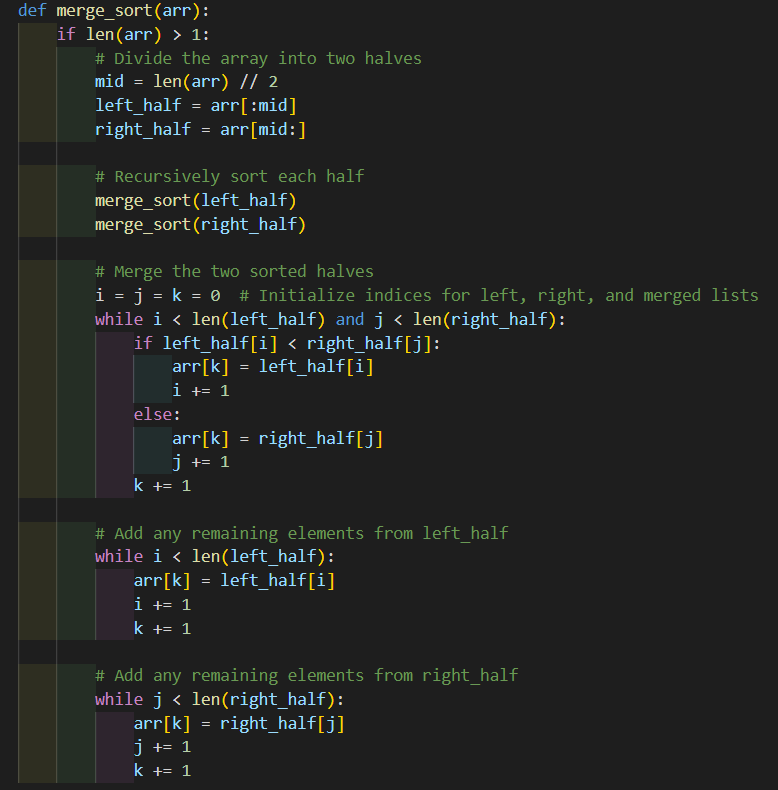
Here are the steps to implement the merge sort algorithm:

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.

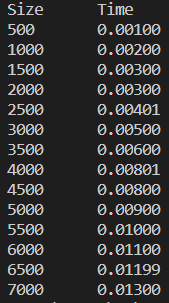
Here's an implementation of the merge sort algorithm in Python:



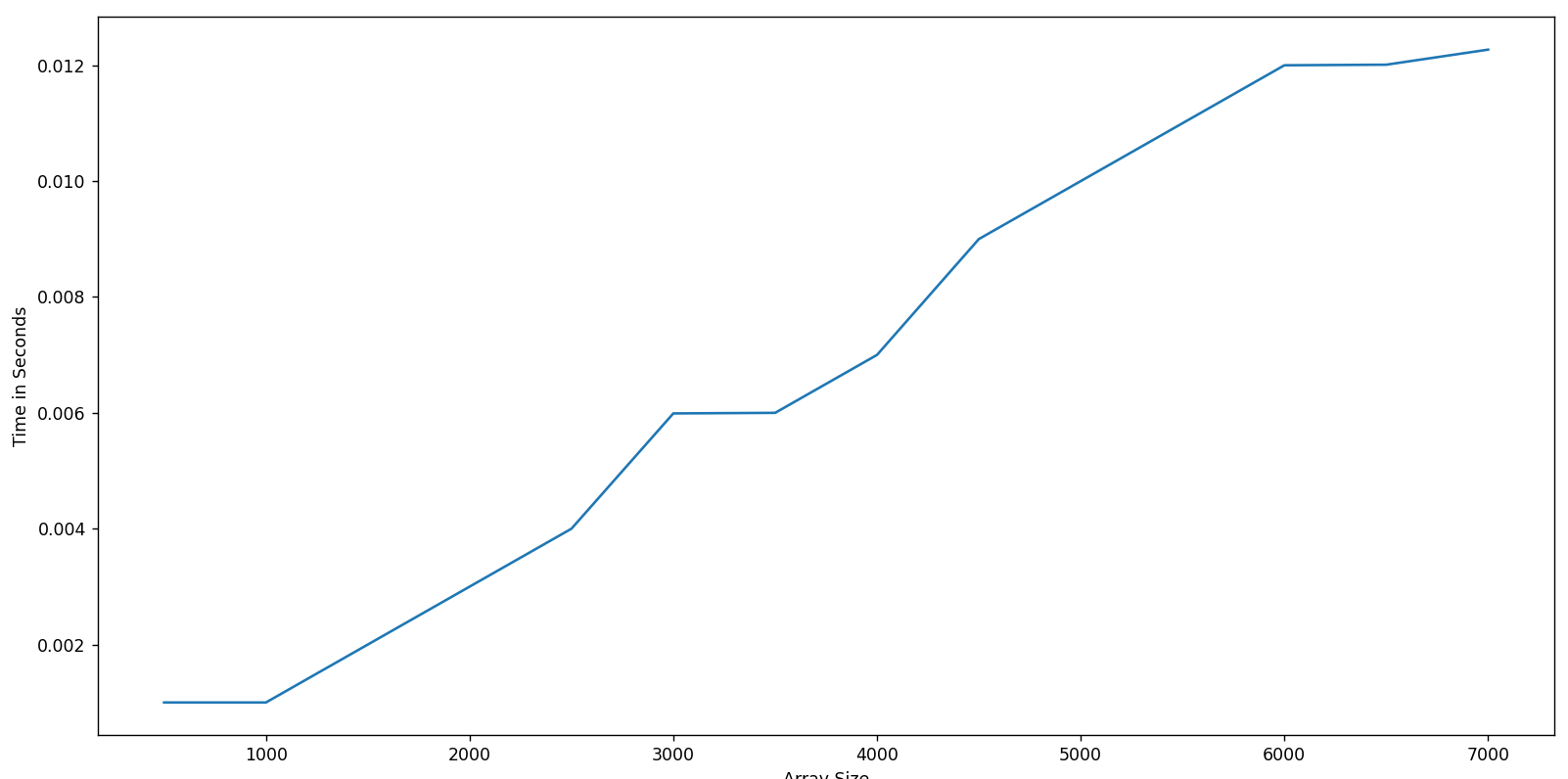
*Figure 2. Merge sort implementation in python*

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*

**Conclusion**

In conclusion, the analysis of various algorithms for finding the n-th Fibonacci number has shown that there are several efficient methods available. The methods analyzed included recursive, iterative, matrix exponentiation, Binet formula, memoization and dynamic programming solutions.

The results indicated that the Binet’s formula solution and the matrix exponentiation method have the smallest time complexity and are the most efficient among the analyzed algorithms. However, it is important to note that the Binet’s formula solution is only suitable for smaller values of n and the matrix exponentiation method requires a deeper understanding of mathematical concepts. Therefore, the choice of algorithm will depend on the specific requirements of the application. Overall, this report highlights the importance of considering different algorithms and evaluating their time complexity when solving computational problems.