**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-213Bardier Andrei

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

**TABLE OF CONTENTS**

[ALGORITHM ANALYSIS 3](#_Toc127483466)

[Objective 3](#_Toc127483467)

[Tasks: 3](#_Toc127483468)

[Theoretical Notes: 3](#_Toc127483469)

[Introduction: 4](#_Toc127483470)

[Comparison Metric: 4](#_Toc127483471)

[Input Format: 4](#_Toc127483472)

[IMPLEMENTATION 5](#_Toc127483473)

[Merge Sort Algorithm 5](#_Toc127483474)

[CONCLUSION 6](#_Toc127483475)

# ALGORITHM ANALYSIS

## Objective

Study and analyze different sorting algorithms, including merge sort, quick sort, heap sort and one for my choice.

## 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:

An alternative to mathematical analysis of complexity is empirical analysis, which can provide useful information about the efficiency of algorithms. The stages of empirical analysis typically include:

1. Establishing the purpose of the analysis.
2. Choosing an efficiency metric, such as the number of operations executed or the execution time of the algorithm or a portion of it.
3. Defining the properties of the input data that the analysis will be based on, such as size or specific properties.
4. Implementing the algorithm in a programming language.
5. Generating multiple sets of input data.
6. Running the program for each set of input data.
7. Analyzing the obtained data.

The choice of the efficiency measure depends on the purpose of the analysis. For example, if the goal is to obtain information about the complexity class or verify a theoretical estimate, the number of operations performed is appropriate, while the execution time is more relevant if the aim is to evaluate the implementation of the algorithm.

It is important to use multiple sets of input data to reflect the different characteristics of the algorithm, covering a range of sizes and different values or configurations of input data. To obtain meaningful results, it is crucial to use input data that reflects the range of situations the algorithm will encounter in practice.

To perform empirical analysis of an algorithm's implementation in a programming language, monitoring sequences are introduced to measure the efficiency metric. If the efficiency metric is the number of operations executed, a counter is used to increment after each execution. If the metric is execution time, the time of entry and exit must be recorded. Most programming languages have functions to measure the time elapsed between two moments, and it is important to only count the time affected by the execution of the analyzed program, especially when measuring time.

After the program has been run for the test data, the results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated or the results are represented graphically as pairs of points in the form of (problem size, efficiency measure).

## Introduction:

Sorting algorithms are essential in programming, and they have been in use for centuries. Sorting is a process of arranging data or information in a specific order, either in ascending or descending. Sorting algorithms are used in various applications like data processing, searching, and decision-making. There are many sorting algorithms available, and the choice of algorithm depends on the size of the dataset, the distribution of the data, and the available memory.

The history of sorting algorithms dates back to ancient times when people used to sort goods in markets. In the early 1940s, computer scientists started working on sorting algorithms for computers. One of the first sorting algorithms was Bubble Sort, which was introduced in 1956. It is a simple sorting algorithm that is easy to understand and implement, but it has poor time complexity and is not suitable for large datasets.

In the 1960s, Quick Sort and Merge Sort were introduced. Quick Sort is a fast algorithm with good time complexity, but it is less stable than other sorting algorithms. Merge Sort, on the other hand, is a stable sorting algorithm that has a better time complexity than Bubble Sort and Quick Sort.

Heap Sort was introduced in the 1970s and is based on the Heap data structure. Heap Sort has an efficient time complexity and is suitable for large datasets.

In summary, sorting algorithms are crucial in programming, and they have evolved over time. The choice of sorting algorithm depends on various factors like the size of the dataset, the distribution of the data, and the available memory. Understanding the characteristics of different sorting algorithms can help programmers choose the most appropriate one for their 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, each algorithm will receive five series of numbers:

* Small array of 100 random elements
* Medium array of 10’000 random elements
* Large array of 1’000’000 random elements
* Large sorted array
* Large partially sorted array (1 : 1 sorted and unsorted parts)

# 

# IMPLEMENTATION

The five algorithms will be implemented without optimization in Python and evaluated based on their completion time. The results obtained may resemble those of other similar experiments, but the specific performance in relation to the input size may vary depending on the memory specifications of the device being used.

## Bubble Sort Algorithm

Bubble Sort is a simple sorting algorithm that is used to sort arrays of elements in ascending or descending order. It is one of the most basic sorting algorithms and is often used in educational contexts to teach programming students about sorting algorithms. The algorithm works by repeatedly swapping adjacent elements if they are in the wrong order until the array is sorted.

The bubble sort algorithm gets its name from the way that elements "bubble up" to their correct positions in the sorted array. In each pass through the array, the algorithm compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the array is fully sorted.

While bubble sort is easy to understand and implement, it is not very efficient for large arrays. Its worst-case time complexity is O(n^2), which means that its performance degrades quickly as the size of the input array grows.

However, bubble sort can still be useful in certain scenarios, such as when sorting small arrays or when the array is already mostly sorted. It is also sometimes used as a building block for more complex sorting algorithms.

Here's the implementation of the bubble sort algorithm in Python:

def bubble\_sort(arr):  
 n = len(arr)  
 # Traverse through all array elements  
 for i in range(n - 1):  
 # optimization  
 swapped = False  
 for j in range(0, n - i - 1):  
 # Swap if the element found is greater than the next element  
 if arr[j] > arr[j+1]:  
 arr[j], arr[j+1] = arr[j+1], arr[j]  
 swapped = True  
 # if no swaps happened => array already sorted  
 #if not swapped:  
 # break  
 return arr

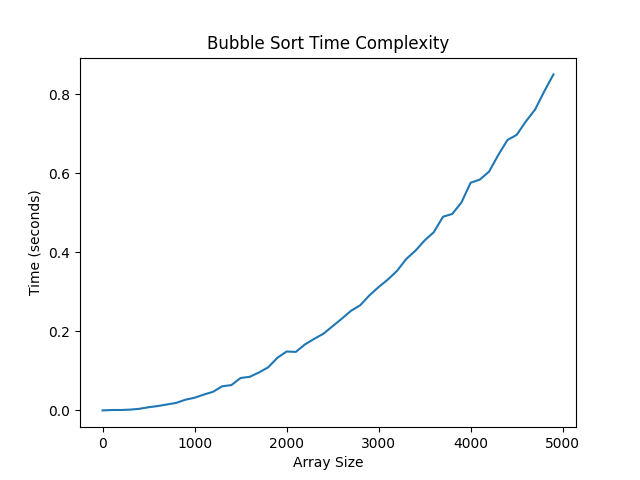


Figure 1. Bubble Sort Time Complexity

# CONCLUSION