**Ministerul Educaţiei și Cercetării al Republicii MoldovaUniversitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 2:

Study and empirical analysis of sorting algorithms

Elaborated:

st. gr. FAF-212 Lupascu Felicia

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

# ALGORITHM ANALYSIS

### Objective:

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

### Tasks:

1. Implement quickSort, mergeSort, heapsort, and another algorithm of 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.

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 generated arrays of random floating-point values between -10.000 and 10.000. The for loop in Main iterates over a range of input sizes from 500 to 6000 (inclusive) in increments of 500. All the arrays are stored in the “arrays” variable:

for s in range(500, 6000+1, 500):  
 arrays.append(generate\_random\_array(s))  
 x.append(s)

# IMPLEMENTATION

* **Merge sort:**

Algorithm Description:

The merge\_sort function is a recursive implementation of the merge sort algorithm, which is a popular sorting algorithm that has a worst-case **time complexity** of O(n log n).

The function takes a list as input and creates a copy of it (arr) to avoid modifying the original list. If the length of the array is greater than 1, it divides the array into two halves (left\_half and right\_half) and recursively calls merge\_sort on each half.

Once both halves are sorted, the function merges them back together into a single sorted array by iterating through the two halves and comparing their elements. The smaller element is added to the merged list, and the corresponding index is incremented. If one of the halves has no more elements to add, the remaining elements from the other half are added to the merged list.

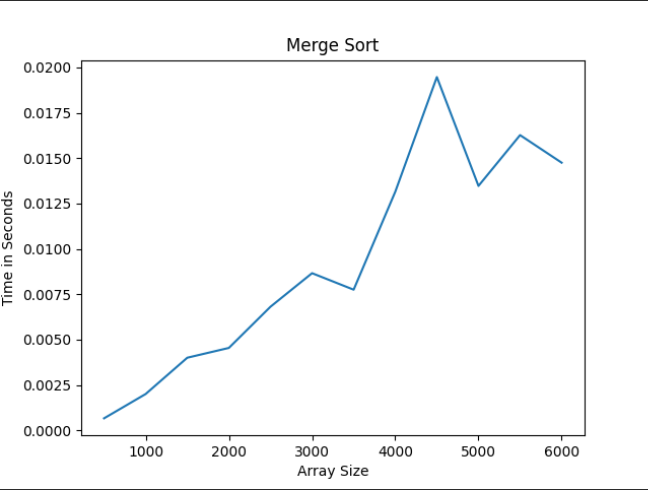
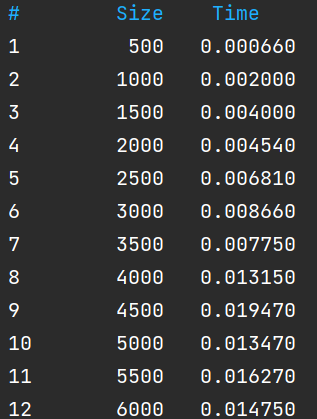
Finally, the function returns the sorted array.

Overall, this implementation of merge sort is correct and efficient. However, note that it uses extra space to create a copy of the original array. If space is a concern, an in-place implementation of merge sort could be used instead.

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. Implementation of the merge sort algorithm in Python:

def merge\_sort(a):  
 arr = a[:]  
 if len(arr) > 1:  
 mid = len(arr) // 2  
 left\_half = arr[:mid]  
 right\_half = arr[mid:]  
 merge\_sort(left\_half)  
 merge\_sort(right\_half)  
  
 i = j =k=0 #Initialize indices for left/right,merged lists  
 while i < len(left\_half) and j < len(right\_half):  
 if left\_half[i] < right\_half[j]:  
 arr[k] = left\_half[i]  
 i += 1  
 otherwise:  
 arr[k] = right\_half[j]  
 j += 1  
 k += 1  
 while i < len(left\_half):  
 arr[k] = left\_half[i]  
 i += 1  
 k += 1  
 while j < len(right\_half):  
 arr[k] = right\_half[j]  
 j += 1  
 k += 1



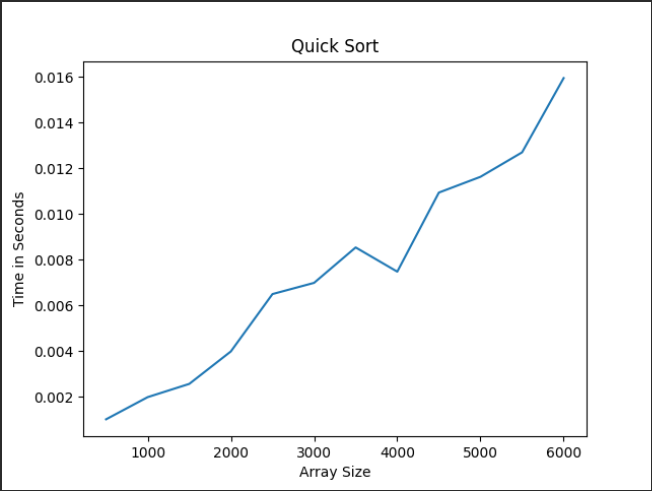
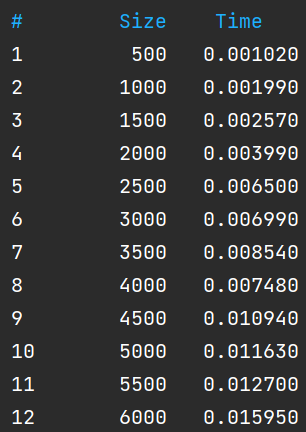
* **Quick Sort**

This code implements the quicksort algorithm to sort an array of numbers. Here's a step-by-step breakdown of how it works:

1. Check if the length of the input array is less than or equal to 1. If it is, return the array as it is already sorted.
2. Choose a pivot element randomly from the input array.
3. Divide the input array into three partitions - left, equal, and right - based on the pivot element. All elements less than the pivot go into the left partition, all elements greater than the pivot go into the right partition, and all elements equal to the pivot go into the equal partition.
4. Recursively sort the left and right partitions using the quicksort function.
5. Concatenate the sorted left partition, equal partition, and sorted right partition to obtain the final sorted array.

The **time complexity** of the quicksort algorithm is O(n\*log(n)) on average and O(n^2) in the worst case when the pivot is the minimum or maximum element of the array. The space complexity of the algorithm is O(log(n)) in the average case and O(n) in the worst case when the recursion depth is equal to the size of the input array.

def quick\_sort(arr):  
 import random  
 # the array is already sorted and can be returned  
 if len(arr) <= 1:  
 return arr  
 # generate a random pivot element from the input array.  
 pivot = arr[random.randint(0, len(arr)-1)]  
 left, equal, right = [], [], []  
 for x in arr:  
 if x < pivot:  
 left.append(x)  
 elif x == pivot:  
 equal.append(x)  
 else:  
 right.append(x)  
 return quicksort(left) + equal + quicksort(right)



* **Heap Sort**

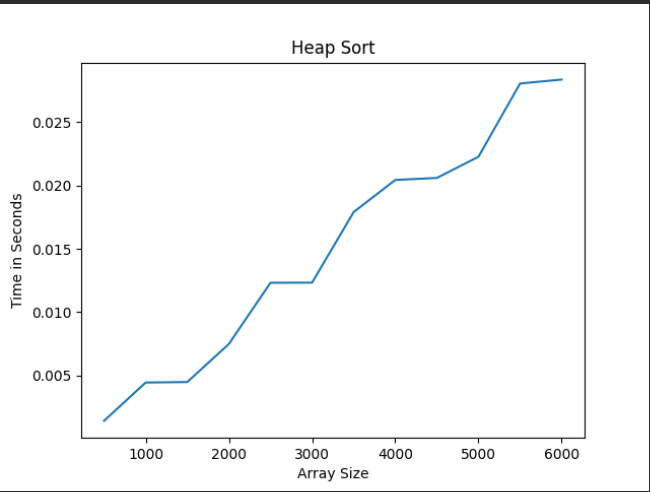
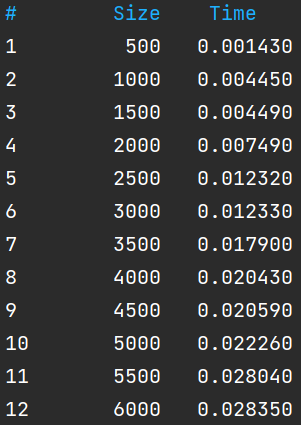
Heap sort is a comparison-based sorting technique based on the Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

# Heap Sort  
def heap\_sort(arr):  
 def build\_max\_heap(arr):  
 n = len(arr)  
 for i in range(n // 2 - 1, -1, -1):  
 heapify(arr, n, i)  
 def heapify(arr, n, i):  
 largest = i  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
 if left < n and arr[left] > arr[largest]:  
 largest = left  
 if right < n and arr[right] > arr[largest]:  
 largest = right  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 heapify(arr, n, largest)  
 arr = arr.copy()  
 n = len(arr)  
 build\_max\_heap(arr)  
 for i in range(n - 1, 0, -1):  
 arr[0], arr[i] = arr[i], arr[0]  
 heapify(arr, i, 0)  
 return arr

This code implements the Heap Sort algorithm, which is a sorting algorithm that uses a binary heap data structure to sort an array.The algorithm first builds a max heap from the input array, then repeatedly extracts the maximum element from the heap and adds it to the sorted portion of the output array.

The build\_max\_heap() function is used to build the max heap from the input array. It starts from the last non-leaf node in the heap and heapifies all nodes in the tree.The heapify() function is used to maintain the heap property of the binary heap data structure. It takes as input the array, the size of the heap, and the index of the node to heapify. It compares the node with its left and right children, and if necessary, swaps the node with the largest child.

Overall, the **time complexity** of Heap Sort is O(nlogn) in the worst case, and the space complexity is O(1), making it an efficient sorting algorithm for large arrays.



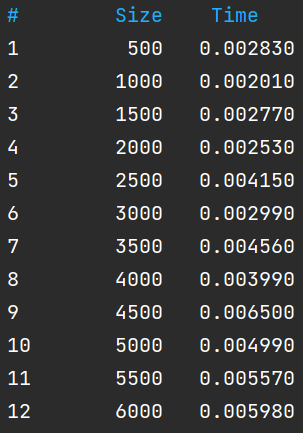
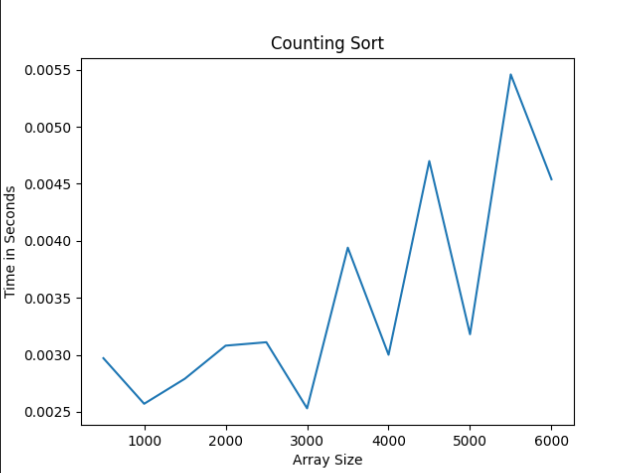
* **Counting Sort**

Counting sort is an efficient algorithm for sorting an array of integers, in which the range of input is not significantly greater than the size of the array. It works by counting the number of occurrences of each value in the input array and using this information to construct a sorted output array.

def counting\_sort(arr):  
 max\_val = float('-inf')  
 min\_val = float('inf')  
 for val in arr:  
 if val > max\_val:  
 max\_val = val  
 if val < min\_val:  
 min\_val = val  
 freq = [0] \* (int(max\_val - min\_val) + 1)  
 for val in arr:  
 freq[int(val - min\_val)] += 1  
 cum\_sum = [freq[0]]  
 for i in range(1, len(freq)):  
 cum\_sum.append(cum\_sum[-1] + freq[i])  
 sorted\_arr = [0] \* len(arr)  
 for val in arr:  
 sorted\_arr[cum\_sum[int(val - min\_val)] - 1] = val  
 cum\_sum[int(val - min\_val)] -= 1  
 return sorted\_arr

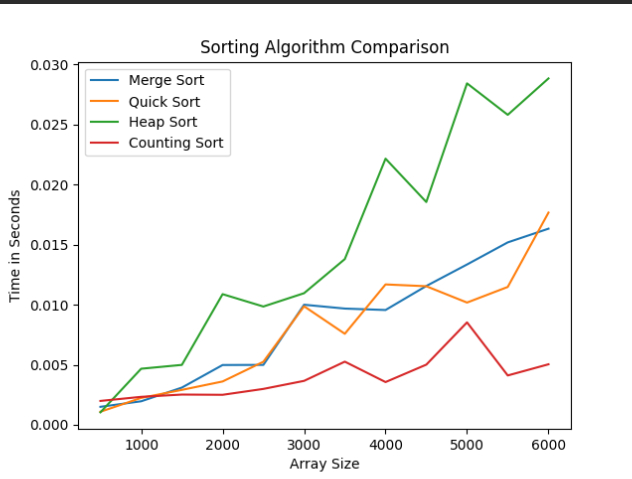
The function first finds the range of the input array by iterating over its elements. It then creates a frequency array to store the count of each distinct element in the input array. The frequency array is initialized to all zeros, and each element of the input array is used to increment the corresponding frequency count in the frequency array.

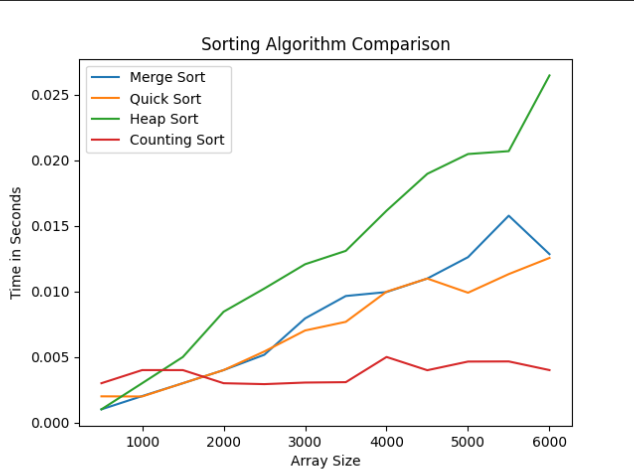
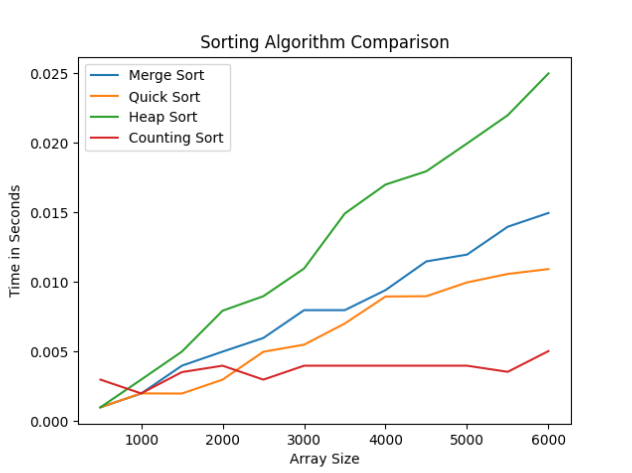
Next, the function calculates the cumulative sum array by summing up the frequency array elements. The cumulative sum array is used to determine the final sorted position of each element in the output array. The sorted array is constructed by iterating over the input array again and placing each element in its sorted position based on the cumulative sum array **Time complexity**: O(n + k), where n is the length of the input array and k is the range of the input numbers..Note that this implementation assumes that the input array only contains integer values. It also assumes that the input array is non-empty.

**Sorting Algorithms Comparison:**

I will attach 3 cases, in which the program was executed, for an analysis of Sorting Algorithms.

****

**** 

A brief comparison of the sorting algorithms mentioned for sorting numbers from 1 to 6000:

Heap sort: Heap sort has an average time complexity of O(n log n) and a worst-case time complexity of O(n log n). Heap sort is an in-place sorting algorithm, meaning it doesn't require additional memory. However, it may not perform as well as other sorting algorithms when dealing with small data sets.

Counting sort: Counting sort is an efficient sorting algorithm for small integers, making it a good choice for sorting numbers from 1 to 6000. It has a time complexity of O(n + k) where n is the number of elements to sort and k is the range of the input data. Counting sorting requires additional memory to store the counts of each element, making it less efficient than in-place sorting algorithms for larger data sets.

Merge sort: Merge sort has a time complexity of O(n log n) and can handle large data sets efficiently. It is also a stable sorting algorithm, meaning it maintains the relative order of equal elements in the input. Merge sort requires additional memory to merge the sub-arrays, making it less efficient than in-place sorting algorithms for large data sets.

Quick sort: Quick sort has a time complexity of O(n log n) on average, making it a good choice for sorting numbers from 1 to 6000. Quick sort is also an in-place sorting algorithm, meaning it doesn't require additional memory. However, its worst-case time complexity is O(n^2), making it less efficient than other sorting algorithms when dealing with nearly sorted data.

However, we can see that for my array, on my device, Counting Sort is the most efficient and fast, in all 3 cases. And the slowest becomes Heap Sort. But Quick and Merge Sort alter (having approximately the same values), that's why they rank in the middle!!!

**Conclusions:**

In conclusion, sorting is an essential computer science process, and several sorting algorithms have been developed over time. I examined and used four of the most well-liked sorting algorithms in this report: quick sort, merge sort, heap sort, and counting sort. I started by examining each algorithm's overall principle and time complexity. Then, I implemented it in code. The code provides a way to measure the time taken for each algorithm to sort a randomly generated array of numbers. The resulting times can be used to compare the performance of the algorithms and determine which algorithm is the most efficient for a particular use case. The code also includes some ANSI color codes to add some color to the output when running the code.

The sorting algorithms are defined as follows:

* Merge Sort: this algorithm recursively divides the input array into two halves, sorts each half separately, and then merges the sorted halves.
* Quick Sort: this algorithm selects a pivot element from the input array, and partitions the remaining elements into two sub-arrays based on whether they are less than or greater than the pivot. It then recursively sorts the sub-arrays using the same method.
* Heap Sort: this algorithm builds a max-heap from the input array, and repeatedly extracts the maximum element from the heap to build the sorted output array.
* Counting Sort: this algorithm calculates the range of the input array, creates a frequency array to count the occurrence of each distinct element, calculates the cumulative sum array by summing the frequency array elements, and constructs the sorted output array by placing each input element in its sorted position based on the cumulative sum array.

To see how each method performed, I generated graphs in addition to comparing execution times. These graphs gave helpful insights into the advantages and disadvantages of each method by displaying the relative performance of each algorithm for a range of input sizes.

For large input sizes – the Counting Sorting Algorithm is the best. The results showed that Merge sort and Quick sort had similar performance. For small array size Counting Sort is not very efficient, the most suitable would be Quick and Heap Sort.

The efficiency of an algorithm depends on several factors, including the input size, the characteristics of the input (e.g., sorted, partially sorted, random, or almost sorted), the implementation of the algorithm, and the hardware on which the algorithm is executed.

The analysis and implementation of sorting algorithms highlighted the significance of selecting the appropriate algorithm for a given question. While certain algorithms could work well with particular input sizes or data types, others might not work well and need more optimization. Understanding the advantages and disadvantages of each algorithm enables us to select the most effective algorithm for a particular issue and improve its efficiency to fulfill the requirements.