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-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:

```
for i in range(500, 7000, 500):
    arrays.append(generate_random_array(i))
```

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:

```python
def merge_sort(arr):
    if len(arr) > 1:
        # Divide the array into two halves
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort each half
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the two sorted halves
        i = j = k = 0  # Initialize indices for left, right, and 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
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Add any remaining elements from left_half
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # Add any remaining elements from right_half
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

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)

Size	Time
500	0.00100
1000	0.00200
1500	0.00300
2000	0.00300
2500	0.00401
3000	0.00500
3500	0.00600
4000	0.00801
4500	0.00800
5000	0.00900
5500	0.01000
6000	0.01100
6500	0.01199
7000	0.01300

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

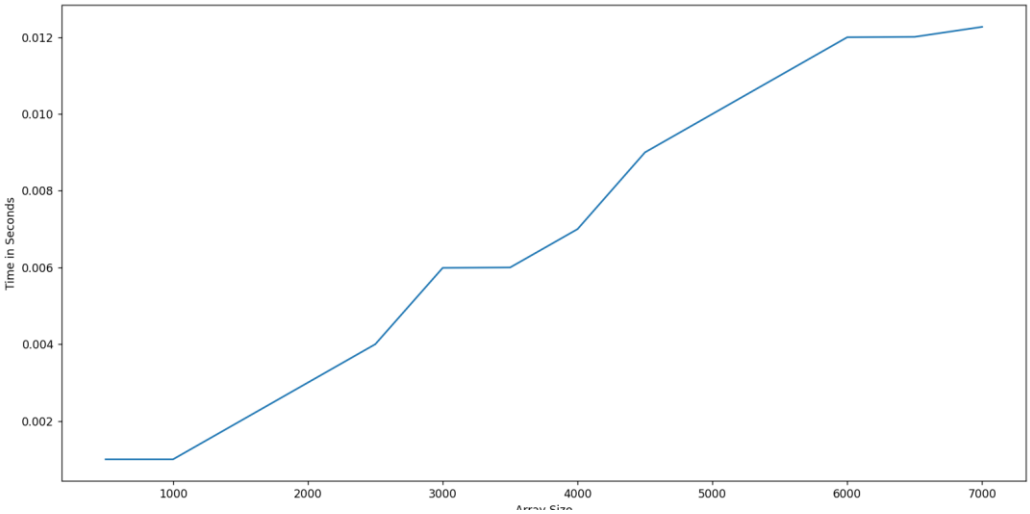

Figure 4. Merge sort time execution graph

2. 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.

```python
def quicksort(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)
    # recursively sort the left and right partitions of the
    # input array using the quicksort() function, and concatenate
    # them with the equal partition to obtain the final sorted array.
    return quicksort(left) + equal + quicksort(right)
```

Figure 5. Quick sort implementation in python

Size	Time
500	0.00100
1000	0.00100
1500	0.00200
2000	0.00300
2500	0.00400
3000	0.00500
3500	0.00500
4000	0.00600
4500	0.00600
5000	0.00700
5500	0.00900
6000	0.01000
6500	0.01000
7000	0.01100

Figure 6. Quick sort results

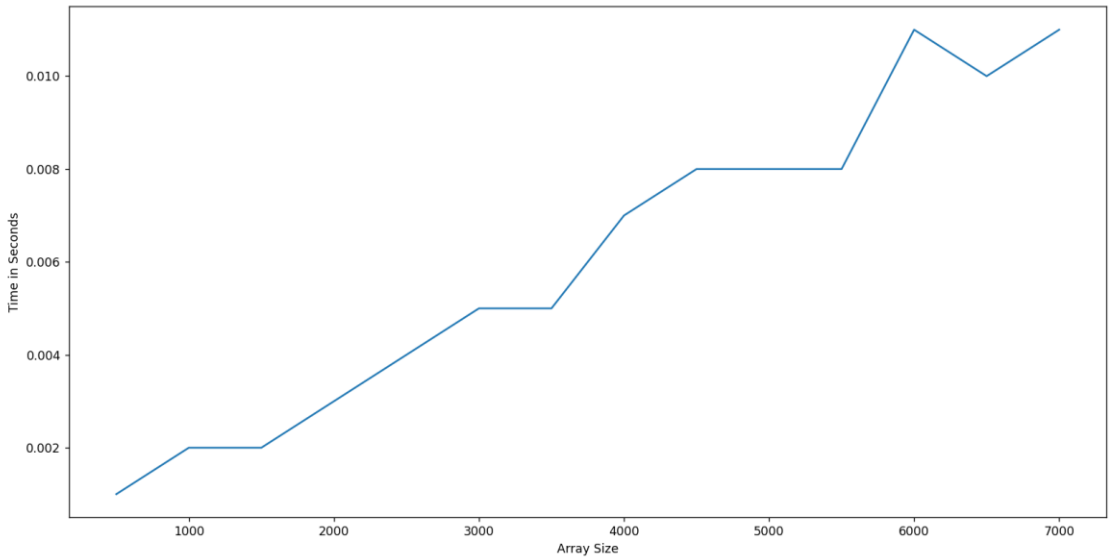

Figure 7. Quick sort time execution graph

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

3. 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:

```python
def heap_sort(arr):
    # Build a max heap from the input array
    def build_max_heap(arr):
        n = len(arr)
        # Starting from the last non-leaf node, heapify all nodes in the tree
        for i in range(n // 2 - 1, -1, -1):
            heapify(arr, n, i)
    # Heapify a subtree rooted at index i
    def heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        # If the left child is larger than the parent
        if left < n and arr[left] > arr[largest]:
            largest = left
        # If the right child is larger than the largest so far
        if right < n and arr[right] > arr[largest]:
            largest = right
        # If the largest element is not the root
        if largest != i:
            # Swap the root with the largest element
            arr[i], arr[largest] = arr[largest], arr[i]
            # Heapify the affected subtree
            heapify(arr, n, largest)
    # Make a copy of the input array to avoid modifying it
    arr = arr.copy()
    n = len(arr)
    # Build a max heap from the input array
    build_max_heap(arr)
    # Repeatedly extract the maximum element from the heap and add it to the sorted portion of the output array
    for i in range(n - 1, 0, -1):
        # Swap the root with the last element in the heap
        arr[0], arr[i] = arr[i], arr[0]
        # Heapify the reduced heap
        heapify(arr, i, 0)
    return arr
```

Figure 8. Heap sort implementation in python

Size	Time
500	0.00400
1000	0.00401
1500	0.00401
2000	0.00400
2500	0.00399
3000	0.00800
3500	0.01201
4000	0.01199
4500	0.01375
5000	0.02001
5500	0.02000
6000	0.02074
6500	0.02000
7000	0.02400

Figure 9. Heap sort results

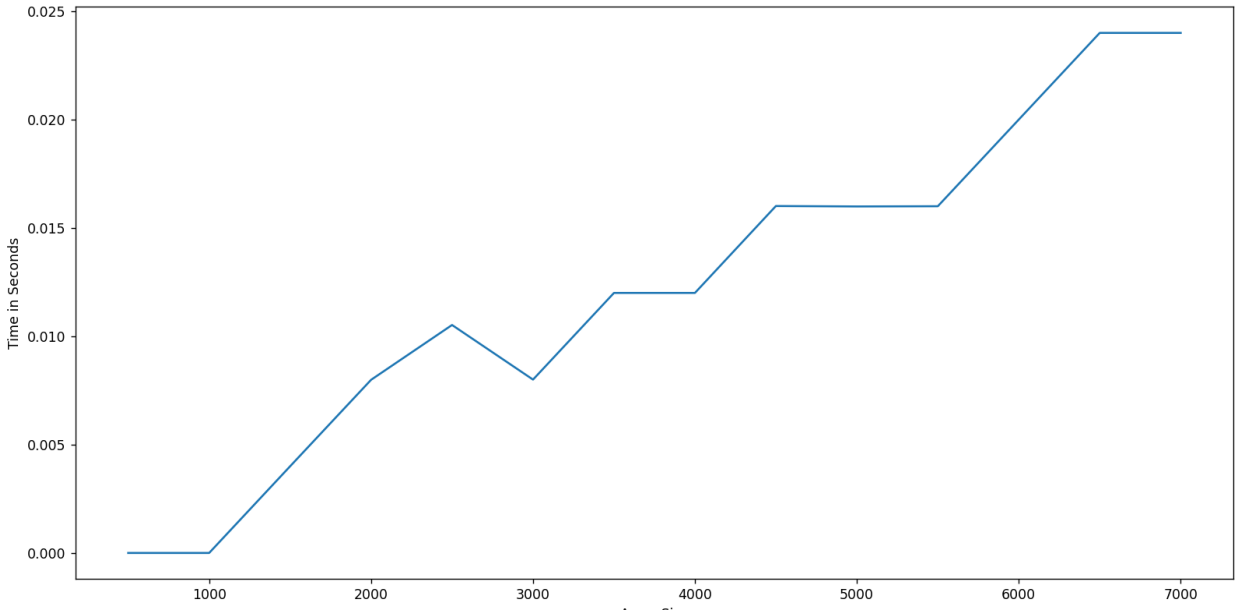

Figure 10. Quick sort time execution graph

Time complexity: O(n log n)

4. 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:

```python
def counting_sort(arr):
    # Find the range of the input array
    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

    # Create a frequency array to store the count of each distinct element in the input array
    freq = [0] * (int(max_val - min_val) + 1)
    for val in arr:
        freq[int(val - min_val)] += 1

    # Calculate the cumulative sum array by summing up the frequency array elements
    cum_sum = [freq[0]]
    for i in range(1, len(freq)):
        cum_sum.append(cum_sum[-1] + freq[i])

    # Construct the sorted output array by placing each input element in its sorted position based on the cumulative sum array
    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
```

Figure 11. Counting sort implementation in python

Size	Time
500	0.00000
1000	0.00366
1500	0.00399
2000	0.00401
2500	0.00401
3000	0.00557
3500	0.00000
4000	0.00402
4500	0.00399
5000	0.00403
5500	0.00399
6000	0.00399
6500	0.00400
7000	0.00399

Figure 12. Counting sort results

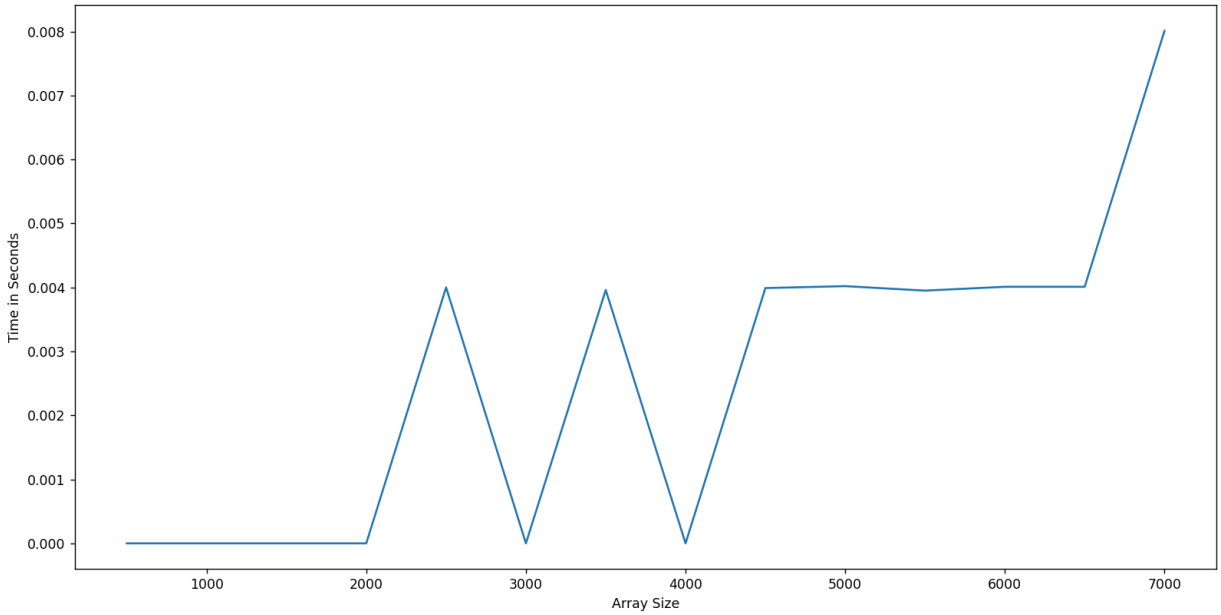

Figure 13. Counti sort time execution graph

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