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. Analysis of quickSort, mergeSort, heapSort, (one of your choice)

Elaborated:
st. gr. FAF-21X Cristian Brinza

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS **3**
Tasks: 3
Theoretical Notes: 3
Introduction: 4
Comparison Metric: 5
Input Format: 5
IMPLEMENTATION **6**
Quick Sort Algorithm 6
Merge Sort Algorithm 7
Heap Sort Algorithm 8
Counting Sort 10
Results 11
CONCLUSION **13**

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, (one of your 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:

Sorting algorithms are an essential part of computer science, and they help us arrange data in a specific order. The aim of this report is to perform an empirical analysis of quickSort, mergeSort, heapSort, and one additional sorting algorithm of our choice. The primary objective is to determine the efficiency, stability, and memory usage of each algorithm.

Algorithm Implementation:

The first step in this project is to implement quickSort, mergeSort, heapSort, and one additional algorithm in a programming language such as Python. The implementation of these algorithms must be done correctly to ensure that the results obtained from them are accurate.

Properties of Input Data:

The properties of the input data against which the analysis will be performed must be established. The input data must be in a format that is easily recognizable by the implemented algorithms. It is essential to ensure that the input data is diverse and includes edge cases to provide a comprehensive analysis of the algorithms.

Metrics for Comparing Algorithms:

To compare the performance of the sorting algorithms, we must select suitable metrics. Common metrics for comparing algorithms include time complexity, space complexity, and stability. Time complexity refers to the amount of time it takes for an algorithm to complete its task, while space complexity refers to the amount of memory an algorithm uses. Stability refers to whether an algorithm preserves the relative order of equal elements in the sorted output.

Empirical Analysis:

The next step is to perform empirical analysis on the implemented algorithms. Empirical analysis involves measuring the actual running time and memory usage by testing the algorithm on real or

simulated inputs. This step will provide us with real-world results that we can use to compare the performance of each algorithm.

Graphical Presentation of Data:

After performing the empirical analysis, it is necessary to present the results in a way that is easy to understand. The results can be presented graphically to provide a clear comparison of the algorithms' performance. The graphs can show the running time, memory usage, and stability of each algorithm.

Conclusion:

In conclusion, the empirical analysis of sorting algorithms helps determine the efficiency, stability, and memory usage of each algorithm. The implementation of quickSort, mergeSort, heapSort, and one additional algorithm, establishing the properties of input data, selecting suitable metrics, performing empirical analysis, and presenting the data graphically are the key steps to obtain valuable insights into the performance of sorting algorithms. This information can be used to choose the best algorithm for a specific task, depending on the requirements of the project.

Introduction:

Sorting is an essential operation in computer science, and it plays a significant role in many applications. The importance of sorting algorithms has led to the development of numerous algorithms that can sort data effectively and efficiently. In this report, we will focus on the study and empirical analysis of three popular sorting algorithms: QuickSort, MergeSort, and HeapSort. In addition, we will choose one more algorithm to analyze and compare against these three algorithms.

The primary objective of this report is to investigate the efficiency and effectiveness of these sorting algorithms in various scenarios. We will implement these algorithms in Python and analyze them empirically. Our analysis will focus on the execution time and memory usage of the algorithms on various input data types and sizes.

The first step in our analysis is to establish the properties of the input data against which the algorithms will be analyzed. We will generate various input data types, including random, sorted, and reverse-sorted arrays, to simulate different scenarios. We will also test the algorithms on different input sizes to observe the effect of data size on the performance of the algorithms.

To compare the algorithms' performance, we will use several metrics, including execution time and memory usage. These metrics will help us determine the most efficient algorithm for different input data types and sizes. We will also make a graphical presentation of the data obtained to provide a visual representation of the algorithms' performance.

In this laboratory work, we will analyze 4 sorting algorithms.

Comparison Metric:

Using the empirical approach, we will analyze and compare the algorithms by their execution runtime in seconds. We will use 5 random generated array, and will compute the average runtime, for more precise comaparison of the algorithms.

Input Format:

Because the Coutning Sort algorithm depends on the input range, we will have 2 main cases: array with small input range: vaues from 0 to 100, and array with large input range: values from 0 to 1000000. Each array will have 50000 elements, so we test on large inputs. Also, each array will be created randomly, so for more precise results, we will have 5 arrays of each type, and we will compute the average runtime on array of each type (with small or large range of values)..

IMPLEMENTATION

All four algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion.

The execution runtime for each experiment was measured using the time library in python. Because the computer has various tasks besides our code compilation, the execution time may differ from time to time. Although a little error exists, the obtained results seemed to be persistent. The error margin determined will constitute 2.5 seconds as per experimental measurement.

Code link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab2

Quick Sort Algorithm

This algorithm uses recursion (function calls itself) to sort the array. It picks an element as a pivot an partitions the array around it.

We choose an element as the pivot and arrange the rest of the elements in the array, such that:

1) The pivot element has a correct position in the final sorted array
2) All the items on the left of the pivot are smaller
3) All the items on the right of the pivot are larger

This algorithm is described by the following

pseudocode:

```
# Define the quick sort algorithm
'''
This function is an implementation of the quicksort algorithm, a divide and conquer
sorting algorithm.
It takes an array (arr) to be sorted, as well as two indices (low and high) that determine
the subarray of arr to be sorted by this recursive call.
'''
def quick_sort(arr, low, high):
# Check if there are at least two elements in the current subarray to sort. If there is
only one or zero,
# then there is nothing to sort and we return.
    if low < high:

        # Use the partition function (not shown here) to find the pivot index that splits
the subarray into
        # two subarrays, one containing elements smaller than the pivot and one containing
elements larger than
        # the pivot.
        partition_index = partition(arr, low, high)
```

```
        # Recursively call quicksort on the left and right subarrays of the pivot index.

        quick_sort(arr, low, partition_index - 1)
        quick_sort(arr, partition_index + 1, high)
```

The ‘low’ and ‘high’ variables represent the starting and ending indices of the subbaray to be sorted. At the first function call we need the whole array, so initially we pass ‘low = 0’ and high = array length – 1’. If ‘low’ is smaller that ‘high’ then there are still elements to be sorted, and we call the partition() function to partition the array into 2 parts and use recursion.
The partition() method always takes the last element as the pivot. There still exist other methods to choose the pivot, such as taking the first array element, or taking the median element. Then, the function iterates through the subarray from ‘low’ to ‘high-1’ and checks if each element is less than the pivot. If it is, it swaps that element with the element at the current index of the smaller element, i, and increments i. After the loop completes, it swaps the pivot element with the element at index i+1, which is the correct position for the pivot in the sorted array. Finally, it returns the index of the pivot element.

The time complexity of Quick Sort is O(n*log(n)) on average, and O(n^2) in the worst case. The worst case occurs when the pivot element chosen is either the smallest or the largest element in the array, leading to unbalanced partitions (one full partition and other is an empty one).

Merge Sort Algorithm

Like Quick Sort, this algorithm also uses recursion to sort the array. The main concept is to divide the array in halves, until we are left with indvidual items. We merge individual items in temporary arrays of 2 elements, and sort them. Then we merge the temporary arrays and sort them, and so on.

This algorithm is described by the following pseudo code:

```
# Define the merge sort algorithm
# This is a function called merge_sort that takes an array as input

def merge_sort(arr):
# The function takes an input array "arr".
# It first checks if the length of the array is greater than 1.
# If the array length is 1 or less, it is already sorted, so the function simply returns
the array.
    if len(arr) > 1:
        # If the array length is greater than 1, the function recursively divides the
array into two halves.
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        # After the array is divided, the function then merges the two halves back
together in sorted order.
```

```
        # It does this by comparing the first elements of each half and putting the
smaller element in the first position of a new array.
        # It then repeats this process for the second elements, third elements, etc. until
the entire array is sorted.
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
                k += 1
        # After comparing and merging all elements in both halves, the function checks if
any elements remain in either half.
        # If elements remain in the left half, they are added to the new array.
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        # If elements remain in the right half, they are added to the new array.
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
    # Finally, the function returns the sorted array.
    return arr
```

The merge_sort function keeps dividing the array until the length of the array becomes 1. Then, it merges the two sorted halves by calling the merge function.

The merge function takes two sorted arrays as inputs and merges them into a single sorted array. The function

first initializes three pointers i, j, and k to zero. Then, it compares the elements of both arrays, and puts the smaller element into the k-th index of the resulting array and increments the corresponding pointers. The function continues the process until all elements are sorted and merged. Finally, the function returns the merged and sorted array.

The time complexity for this algorithm is O(n*logn).

Heap Sort Algorithm

This algorithm first builds a max heap from the input array and then extracts elements from the heap one by one and places them at the end of the array.

The logic of the algorithm is represented by the following pseudocode:

```
# Define the heap sort algorithm
'''
```

```
This code defines a heap_sort function that takes an array as an input and returns the sorted array. It sorts the array in place using the heap sort algorithm, which uses a max heap data structure.
The heap_sort function first finds the length of the input array and builds a max heap out of it using the build_max_heap function.
Then, the function loops through the array from the end to the beginning using a for loop. At each iteration, it swaps the first element (which is the largest element in the heap) with the ith element, where i is the current index of the loop. It then heapifies the array again using the heapify function.
Finally, the function returns the sorted array.
The build_max_heap and heapify functions are not defined in this code snippet and should be defined elsewhere in the code or imported from a library. These functions are common in heap sort implementations and are used to create and maintain the heap structure.
'''
# Define a function called heap_sort that takes an array as an argument
def heap_sort(arr):
    # Find the length of the array
    n = len(arr)
    # Build a max heap out of the array
    build_max_heap(arr, n)
    # Loop through the array from the end to the beginning
    for i in range(n - 1, 0, -1):
        # Swap the first element with the ith element
        arr[0], arr[i] = arr[i], arr[0]
        # Heapify the array again
        heapify(arr, i, 0)
    # Return the sorted array
    return arr


# Define the build_max_heap function, used in the heap sort algorithm
def build_max_heap(arr, n):
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

# Define the heapify function, used in the build_max_heap function and the heap sort algorithm
'''
This is a function that implements the heapify operation on an array
It takes three arguments:
arr: the array to be heapified
n: the size of the array
i: the index of the current node being heapified
'''
def heapify(arr, n, i):
    largest = i # initialize the largest node as the current node
    left_child = 2 * i + 1 # calculate the index of the left child
    right_child = 2 * i + 2 # calculate the index of the right child

    # if left child is larger than largest, update largest
    if left_child < n and arr[left_child] > arr[largest]:
        largest = left_child
```

```
    # if right child is larger than largest, update largest
    if right_child < n and arr[right_child] > arr[largest]:
        largest = right_child

    # if largest is not the current node, swap the values of largest and current node
    # and recursively call heapify on the affected child node
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

The heap_sort function takes the input array as an argument and first builds a max heap from the array using the build_max_heap function. It then extracts elements from the heap one by one and places them at the end of the array using the heapify function.

The build_max_heap procedure iterates over all parent nodes in the tree and applies the heapify procedure to each one to create the max heap. Starting at a given node, the heapify function compares the node with its left and right children and finds the largest element. If the largest element is not the node itself, then the node is swapped with the largest element and the procedure is recursively called on the subtree rooted at the largest element.

The time complexity for this algorithm is O(n*logn), which makes it a pretty efficient algorithm. The
build_max_heap has a complexity of O(n) and the heapify function a complexity of O(logn).

Counting Sort

The Counting Sort Algorithm works by counting the number of objects having distinct key values. It then uses mathematical calculations to determine the placement of each element in the final sorted array.

The pseudocode describing the Counting Sort Algorithm:

```
# This function is called counting_sort and takes in an array as input.
def counting_sort(arr):
    # Find the maximum element in the input array and cast it to an integer.
    max_element = int(max(arr))

    # Create a list called count with length of the maximum element + 1, and fill it with
zeros.
    count = [0 for _ in range(max_element + 1)]

    # Iterate through the input array, incrementing the count of the current element in
the count list.
    for i in arr:
        count[i] += 1

    # Create a temporary variable and initialize it to zero.
    temp = 0
```

```
    # Iterate through each element in the count list and for each non-zero value,
    # set the corresponding number of elements in the input array to the current element
value.
    for i in range(max_element + 1):
        for j in range(count[i]):
            arr[temp] = i
            temp += 1

    # Return the sorted array.
    return arr
```

In the first iteration, the function counts the number of occurrences of each element in the array and stores it in a count array. In the second iteration, it fills the output array with the elements from the input array in the order determined by the count array.

The time complexity of counting sort is O(n+k), where n is the number of elements in the input array and k is the range of input. Counting sort is efficient when the range of input is not significantly greater than the number of elements in the array, making it useful for sorting arrays with a small range of values, such as integers. However, it can become inefficient when the range is very large, as it requires allocating an array of size k.

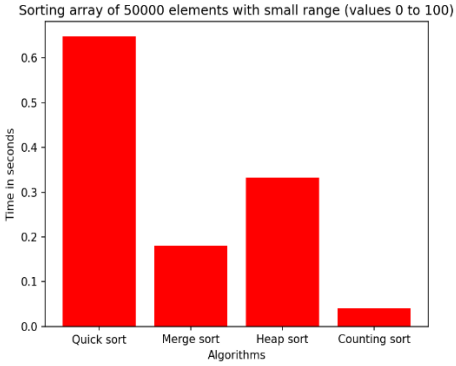
Results

The algrtihm were tested on 5 arrays of each of the 2 types of array (with small and large input range). Then the average runtime was taken, just by dividing the whole time with 5.

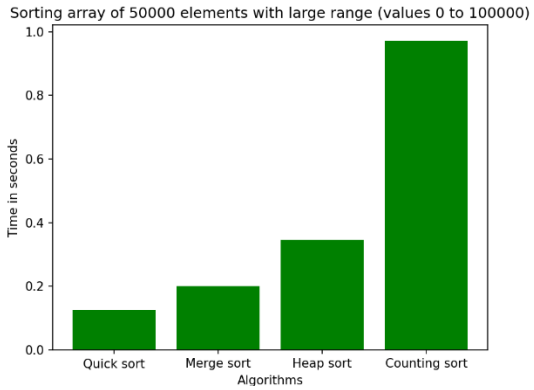
For simplicity and convience, the results were put in a bar chart diagram.

Runing the code for arrays with small range of values, we obtain the results:

As we can see, the slowest one is the Quick Sort, having a worst-case time complexity of O(n^2). The Merge Sort and the Heap Sort gave similar results, both having a time complexity of O(n*logn). And the fastest one is Counting Sort, because the input range is pretty small, resulting in a time complexity close to O(n).



But runing the code for arrays with large range of values, we obtain the results:



As we can see, the Quick Sort gave similar results to Merge and Heap Sorts, because it reached the average time complexity case of O(n*logn). The slowest algorithm is the Counting Sort, because for such lurge values of k, O(n+k) is bigger than O(n*logn).

CONCLUSION

During this laboratory work, we analyzed four algorithms for sorting arrays and compared their efficiency through empirical analysis, based on their average execution time.

The Quick Sort algorithm, while versatile, can have poor runtime results due to its worst-case time complexity of O(n^2). It is therefore crucial to use suitable methods for finding the pivot element to avoid worst-case time complexity and achieve an average case of O(n*logn) time.

The Merge Sort and Heap Sort algorithms produced similar results because of their comparable time complexity, and the choice of algorithm depends on implementation preferences.

The Counting Sort Algorithm demonstrated fast runtime results when working with inputs of a small range, making it an excellent choice for sorting arrays with limited input ranges.

However, as the range increases, so does the runtime efficiency, making it less suitable for larger input ranges.

Various sorting algorithms exist, each with its strengths and weaknesses in terms of efficiency, stability, and memory usage.

Empirical analysis is necessary to compare them and make the optimal choice in different situations.

Code link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab2