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-211 Cunev Anastasia

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS.....3
Objective 3
Tasks..... 3
Theoretical Notes: ……..... 3
Introduction: …..... 4
Comparison Metric:..... 4
Input Format: …..... 5
IMPLEMENTATION.....5
QuickSort: ……..... 5
BubleSort: ……..... 7
MergeSort:..... 9
HeapSort: ...……..……..... 11
CONCLUSION.....14

ALGORITHM ANALYSIS

Objective

Study and analyze different algorithms for sorting arrays.

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:

Empirical analysis is an alternative to mathematical analysis of complexity.

This might be beneficial for gaining preliminary information on an algorithm's difficulty class; comparing the efficiency of two (or more) methods for tackling the same issues, comparing the efficiency of several implementations of the same algorithm, and getting information on the efficiency of implementing an algorithm on a certain machine.

The following stages are often used in the empirical study of an algorithm:

1. The analysis's aim is established.

2. Determine the efficiency metric to be utilized (number of operation (s) executions or time execution of all or a portion of the algorithm).

3. The qualities of the input data on which the analysis is done are determined (data size or specific properties).

4. A programming language is used to implement the algorithm.

5. Creating numerous input data sets.

6. Run the program for each piece of input data.

7. The collected data is examined.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Sorting algorithms are a fundamental part of computer science and are used to arrange a set of items in a specific order. There are several types of sorting algorithms, each with its unique characteristics, advantages, and disadvantages. Some of the most common sorting algorithms include:

QuickSort:

Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

MergeSort:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves.

HeapSort:

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees. The initial set of numbers that we want to sort is stored in an array and after sorting, we get a sorted array. Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

BubbleSort:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements and swaps them if they are in the wrong order.

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 random arrays of different length. The arrays will be of length from 500 to 5000 numbers.

IMPLEMENTATION

All 4 algorithms will be implemented in python and analyzed empirically based on the time required for their completion. Comparing all these algorithms we can define which method is most suitable for different needs or which one is the most efficient.

QuickSort:

QuickSort is a sorting algorithm based on the divide and conquer approach where

1. An array is divided into subarrays by selecting a pivot element (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Algorithm Description:

The next pseudocode represents the quickSort algorithm for array sorting:

```
quickSort(arr):
    if len(arr) <= 1:
        return arr
    else
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot
        greater = [x for x in arr[1:] if x > pivot
        return quickSort(less) + [pivot] +
quickSort(greater)
```

Implementation:

```python
def quickSort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quickSort(less) + [pivot] + quickSort(greater)
```

Figure 1. quickSort in Python

Results:

After running the quickSort algorithm for different random arrays, I got the following results:

nr	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	0.000758171	0.00123405	0.00179601	0.00214291	0.00285101	0.00321722	0.00404716	0.00443172	0.00537586	0.00560308

Figure 2. Results for the set of inputs

In Figure 2 is represented the table of results for the set of inputs. The first row represents the lengths of the arrays, but the second one the time needed to sort them. Taking into consideration these results, we can clearly see that the time complexity of this algorithm is T(nlogn).

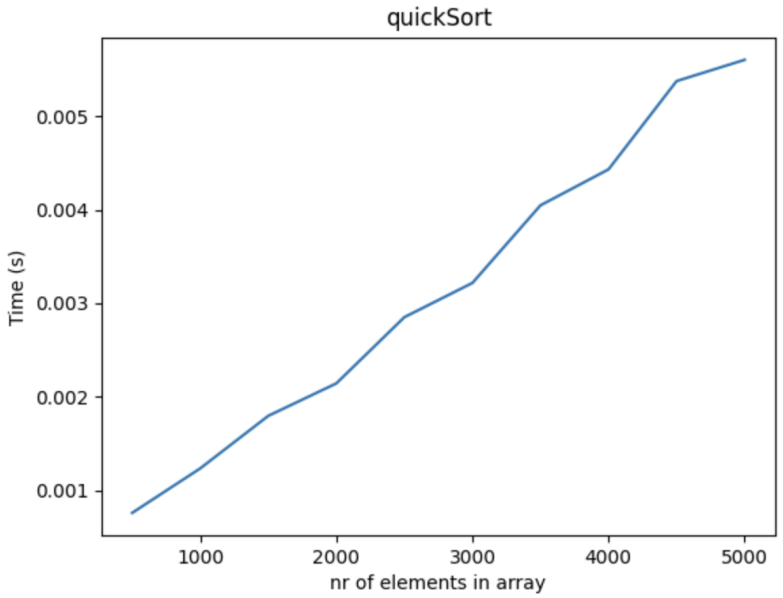

Figure 3. Graph of the quickSort Function

On the graph in Figure 3 we can also see that the Time Complexity is T(nlogn).

Time Complexity:

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **T(nlogn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **T(nlogn)**.
- **Worst Case Complexity -** In quicksort, the worst case occurs when the pivot element is either the greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **T(n^2)**.
  Since we found the median pivot, we have the best case scenario, therefore, the graph represents T(nlogn).

BubbleSort:

Bubble Sort is a sorting algorithm that repeatedly compares adjacent elements in the list and swaps them if they are in the wrong order. This process is repeated for every element in the list until the list is completely sorted. The largest element in the list "bubbles" to the end of the list after each pass. Bubble Sort has a time complexity of O(n^2), making it inefficient for large lists, but it is easy to understand and implement.

Algorithm Description:

The next pseudocode represents the bubleSort algorithm for array sorting:

```
procedure bubbleSort(A : list of sortable items)

   n = length(A)
   for i = 0 to n-1 do
       for j = 0 to n-i-1 do
           if A[j] > A[j+1] then
               swap A[j] and A[j+1]
           end if
       end for
   end for
end procedure
```

Implementation:

```
def bubbleSort(n):
    swapped = False
    for m in range(len(n) - 1, 0, -1):
        for i in range(m):
            if n[i] > n[i + 1]:
                swapped = True
                n[i], n[i + 1] = n[i + 1], n[i]
        if not swapped:
            return
```

Figure 4. bubleSort in Python

Results:

After running the bubleSort algorithm for every term from the second list of inputs, I got the following results:

nr	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	0.0198562	0.0584939	0.118232	0.204158	0.313259	0.435366	0.593962	0.750145	0.940534	1.17934

Figure 5. Results for the set of inputs

In Figure 5 is represented the table of results for the set of inputs. The first row represents the lengths of the arrays, but the second one the time needed to sort them. Taking into consideration these results, we can clearly see that the time complexity of this algorithm is T(n^2).

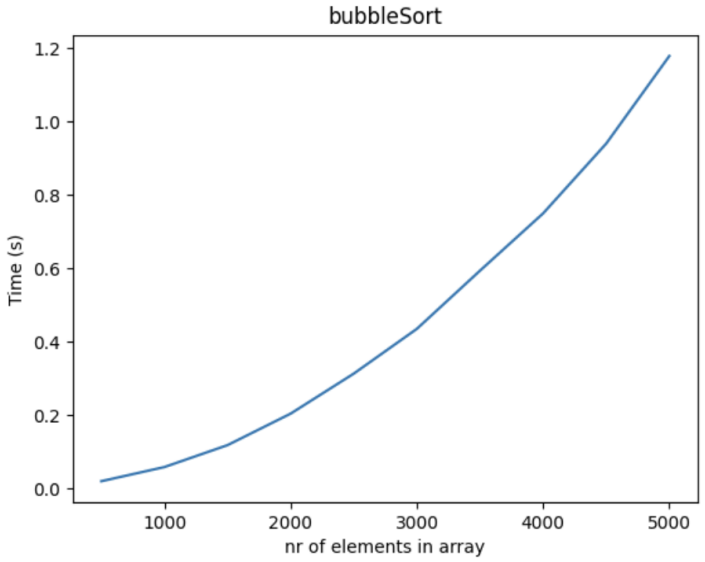

Figure 6. Graph of bubleSort Function

On the graph in Figure 6 we can also see that the Time Complexity is T(n^2), which makes it the most inefficient algorithm from the proposed ones.

MergeSort:

Merge sort is a sorting algorithm that uses a divide-and-conquer approach to sort a list of elements. It works by breaking down the list into smaller sub-lists, sorting those sub-lists, and then merging them back together in sorted order. To sort a list using this algorithm, we first divide the list into smaller sub-lists by repeatedly dividing each sub-list into halves until we reach sub-lists that contain only one element. We then combine adjacent single-element sub-lists into two-element sub-lists and sort them as we combine them. We repeat this process of merging and sorting adjacent sub-lists, doubling the size of the sub-lists at each step, until we have a single sorted list.

Algorithm Description:

The next pseudocode represents the mergeSort algorithm for array sorting:

```
merge(left_list, right_list)sorted_list = [] left_list_index
<- right_list_index <- 0

left_list_length, right_list_length <- len(left_list),
len(right_list)

for _ in range(left_list_length + right_list_length) then

if left_list_index < left_list_length and right_list_index <

right_list_length do

if left_list[left_list_index] <= right_list[right_list_index]

then

sorted_list.append(left_list[left_list_index])

left_list_index <- left_list_index + 1 otherwise:

sorted_list.append(right_list[right_list_index])
right_list_index <- right_list_index + 1

else if left_list_index are equal left_list_length then
sorted_list.append(right_list[right_list_index])
right_list_index <- right_list_index + 1

else if right_list_index == right_list_length then
sorted_list.append(left_list[left_list_index]) left_list_index
<- left_list_index + 1
```

```
return sorted_list

merge_sort(nums)

if len(nums) <= 1 then

return nums

mid = len(nums) // 2

left_list <- merge_sort(nums[:mid]) right_list <-
merge_sort(nums[mid:])

return merge(left_list, right_list)
```

Implementation:

```python
def mergeSort(arr):
    if len(arr) > 1:

        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]

        mergeSort(left)
        mergeSort(right)

        i = j = k = 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

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
```

```python
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

Figure 7. mergeSort in Python

Results:

After running the mergeSort algorithm I got the following results:

nr	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	0.00120306	0.00202608	0.00293994	0.00399399	0.00502777	0.00561786	0.00668979	0.00760388	0.00879407	0.00974202

Figure 8. Results for the set of inputs

In Figure 5 is represented the table of results for the set of inputs. The first row represents the lengths of the arrays, but the second one the time needed to sort them. Taking into consideration these results, we can clearly see that the time complexity of this algorithm is T(nlog(n)).

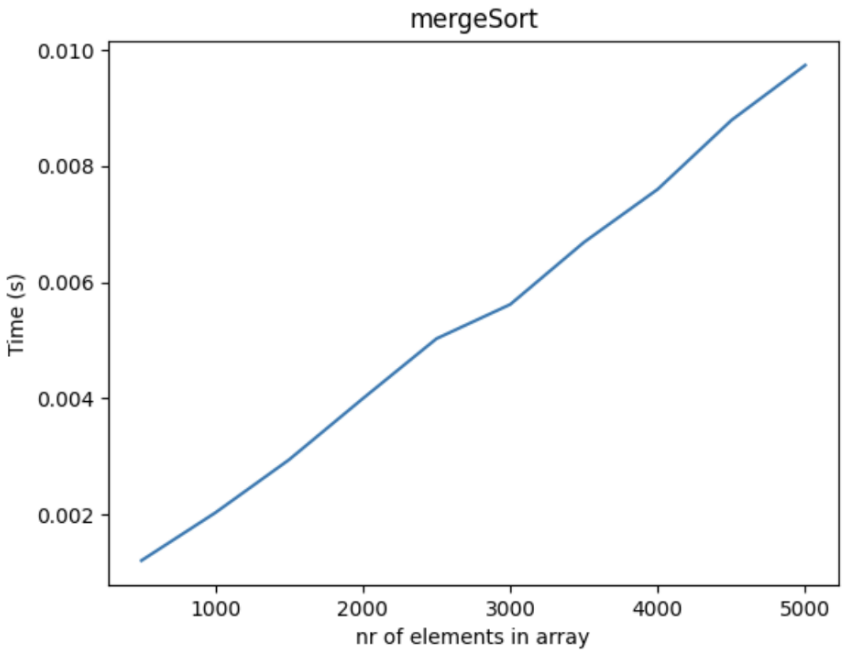

Figure 9. Graph of the mergeSort Function

On the graph in Figure 9 we can see that the time complexity for the mergeSort is T(nlog(n)).

HeapSort:

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.

4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

The next pseudocode represents the heapSort:

```
   procedure heapsort(A: array of items)
    n ← length(A)
   for i from floor(n/2) down to 1 do:
        sift_down(A, i, n)
   for i from n down to 2 do:
        swap(A[1], A[i])
        sift_down(A, 1, i - 1)
procedure sift_down(A: array of items, i, end):
    root ← i
    while root * 2 ≤ end do:
        child ← root * 2
        if child + 1 ≤ end and A[child] < A[child + 1] then:
            child ← child + 1
        if A[root] < A[child] then:
            swap(A[root], A[child])
            root ← child
        else:
            return
```

Implementation:

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)

    for i in range(n // 2, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]

        heapify(arr, i, 0)
```

Figure 10. heapSort in Python

Results:

The results for the second list of inputs for the heapSort algorithm :

nr	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time (s)	0.00152874	0.00285697	0.00417089	0.00542593	0.00690579	0.00835419	0.010031	0.0116019	0.0130591	0.0147047

Figure 11. Results for the set of inputs

In Figure 5 is represented the table of results for the set of inputs. From the following set of results we can see that the heapSort algorithm is quite efficient. Its Time Complexity is T(nlog(n)).

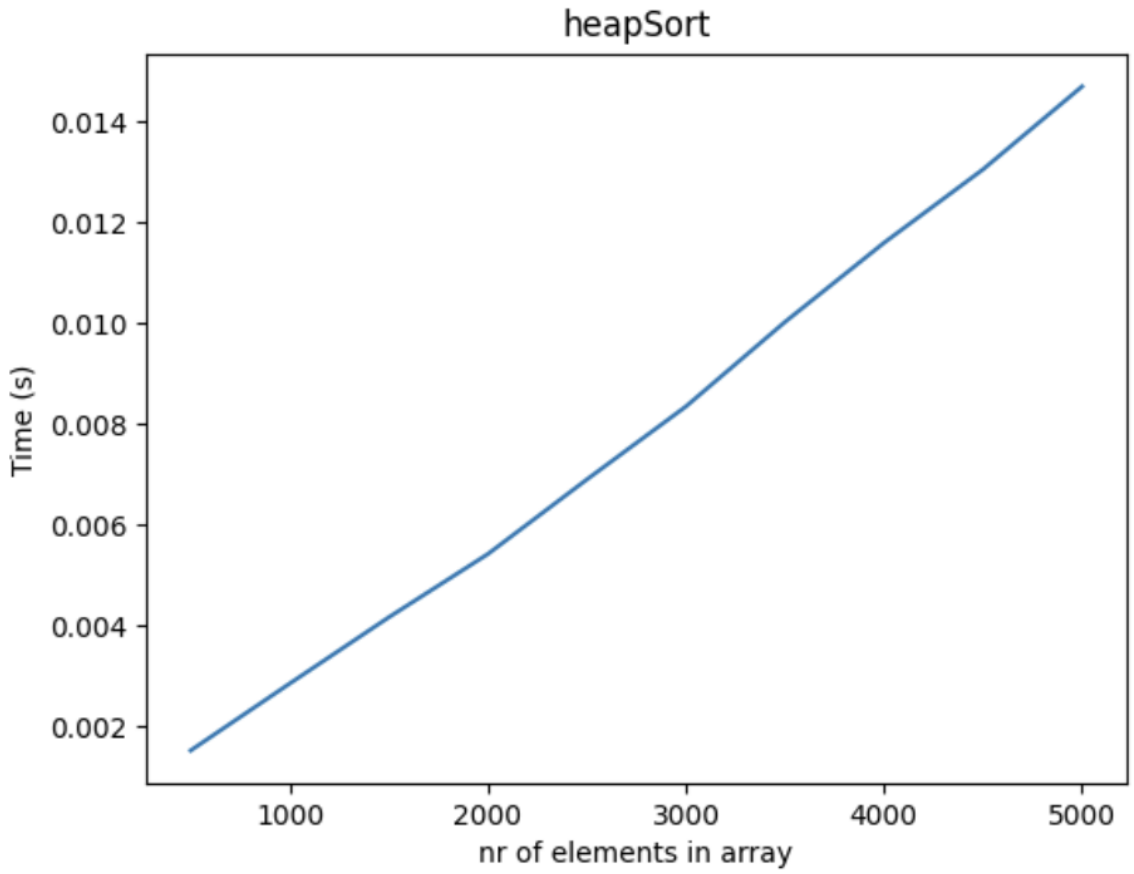

Figure 12. Graph of heapSort Function

On the graph in Figure 12 we can see that the Time Complexity for the heapSort is T(nlog(n)).

CONCLUSION

In this laboratory work we had to make the empirical analysis of the sorting algorithms. I had to test 4 algorithms: mergeSort, quickSort, heapSort and bubleSort. After performing this work we can make conclusions which algorithm is better for different needs.

QuickSort is a very efficient algorithm with an average-case time complexity of T(n log n). This sorting algorithm has a small memory requirement and operates in-place, meaning it sorts the original array without requiring additional memory for creating a separate copy. It's a good choice for sorting large datasets, especially when space is a concern.

BubbleSort is an algorithm, which has a time complexity of T(n^2), which makes it inefficient for large lists and also the most inefficient from the proposed ones.

MergeSort is a stable and reliable sorting algorithm with a time complexity of T(n log n). Compared to the quickSort, it requires extra memory for sorting, as it merges the sorted subarrays into a larger array. MergeSort is not an in-place sorting algorithm, so it requires extra memory for sorting.

HeapSort has a time complexity of T(n log n) and is an in-place sorting algorithm. The heapSort algorithm has limited uses because Quicksort and Mergesort are better in practice. HeapSort can be slower than other sorting algorithms for small datasets, due to its overhead of building a heap.

In conclusion, the choice of sorting algorithm depends on various factors such as the size of the dataset, available memory, stability requirements, and the nature of the data. If we have to sort a small array, the bubbleSort is a good option, even though it is the most inefficient one from the proposed algorithms. If memory is limited, QuickSort may be the best option. If the input data can fit into memory and a stable sort is not required, HeapSort may offer efficient performance. For large datasets and stable sorts, MergeSort is a good choice.

The link to the repository:

https://github.com/NastiaCu/AA_LABS