Ministry of Education, Culture and Research
Technical University of Moldova

# REPORT

Laboratory Work No. 2

Sort algorithm analysis (Execution time of an algorithm)

Student: Dvorac Ana

Group: FAF - 193

# ABSTRACT

The purpose of this laboratory work is to compare execution time of 4 different sorting algorithms: Quick Sort, Merge Sort, Heap Sort and one of our choice. I chose Radix algorithm.

Briefly, I described process of sorting an array using different Sort Algorithms. For comparing their time complexity, I took 10 different random generated arrays of 1000 digits from 1 to 100, and plot the results. Furthermore, I introduced obtained values into a table and found an average time for each sort. The result showed that the fastest algorithm is Quick Sort with average time complexity of O(n log(n)) and space complexity of O(n).

# 1. INTRODUCTION

## 1.1 Definition

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order.
Sorts are most commonly in numerical or a form of alphabetical (called lexicographical) order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order.

Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.

When using different algorithms some questions have to be asked. How big is the collection being sorted? How much memory is at disposal to be used? Does the collection need to grow?
The answers to these questions may determine what algorithm is going to work best for the situation. Some algorithms like merge sort may need a lot of space to run, while insertion sort is not always the fastest but it doesn't require many resources to run.

## 1.2 Some of sorting algorithms

Some of the most common sorting algorithm:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort

## 1.3 Classification of a Sorting Algorithm

Sorting algorithms can be categorized based on the following parameters:

1. Based on Number of Swaps or Inversion. This is the number of times the algorithm swaps elements to sort the input. `Selection Sort` requires the minimum number of swaps.

2. Based on Number of Comparisons. This is the number of times the algorithm compares elements to sort the input. Using Big-O notation, the sorting algorithm examples listed above require at least O(nlogn) comparisons in the best case and O(n^2) comparisons in the worst case for most of the outputs.

3. Based on Recursion or Non-Recursion. Some sorting algorithms, such as Quick Sort , use recursive techniques to sort the input. Other sorting algorithms, such as Selection Sort or Insertion Sort , use non-recursive techniques. Finally, some sorting algorithm, such as Merge Sort , make use of both recursive as well as non-recursive techniques to sort the input.

4. Based on Stability. Sorting algorithms are said to be stable if the algorithm maintains the relative order of elements with equal keys. In other words, two equivalent elements remain in the same order in the sorted output as they were in the input.

5. Insertion sort , Merge Sort , and Bubble Sort are stable.

6. Heap Sort and Quick Sort are not stable.

7. Based on Extra Space. Requirement Sorting algorithms are said to be in place if they require a constant O(1) extra space for sorting.

8. Insertion sort and Quick-sort are in place sort as we move the elements about the pivot and do not actually use a separate array which is NOT the case in merge sort where the size of the input must be allocated beforehand to store the output during the sort.

9. Merge Sort is an example of out place sort as it require extra memory space for its operations.

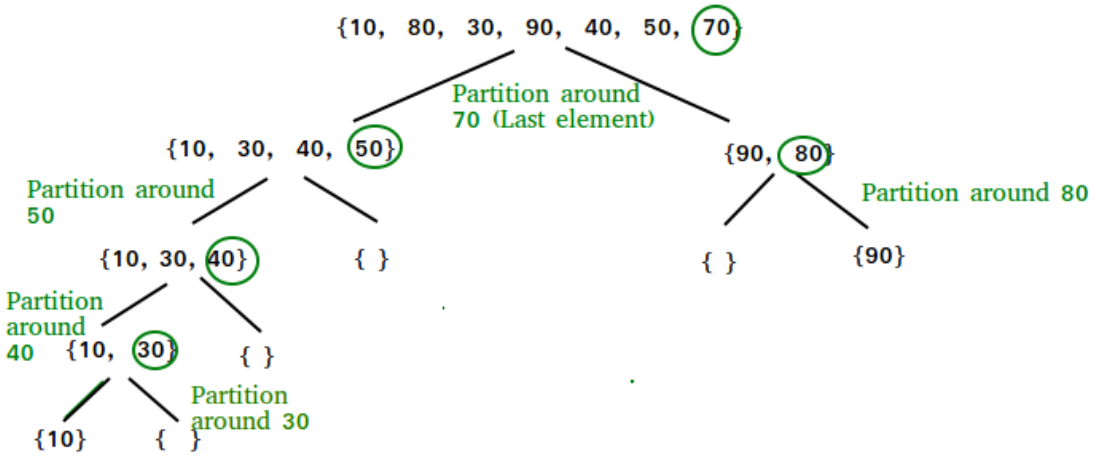
# 2. SORTING ALGORITHMS

## 2.1 Quick Sort

Quick sort is an efficient divide and conquer sorting algorithm. Average case time complexity of Quick Sort is O(nlog(n)) with worst case time complexity being O(n^2) depending on the selection of the pivot element, which divides the current array into two sub arrays.

For instance, the time complexity of Quick Sort is approximately O(nlog(n)) when the selection of pivot divides original array into two nearly equal sized sub arrays.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.




### 2.1.1 Source code:

Anexa 1, Anexa A, Anexa B

### 2.1.2 Time Complexity:

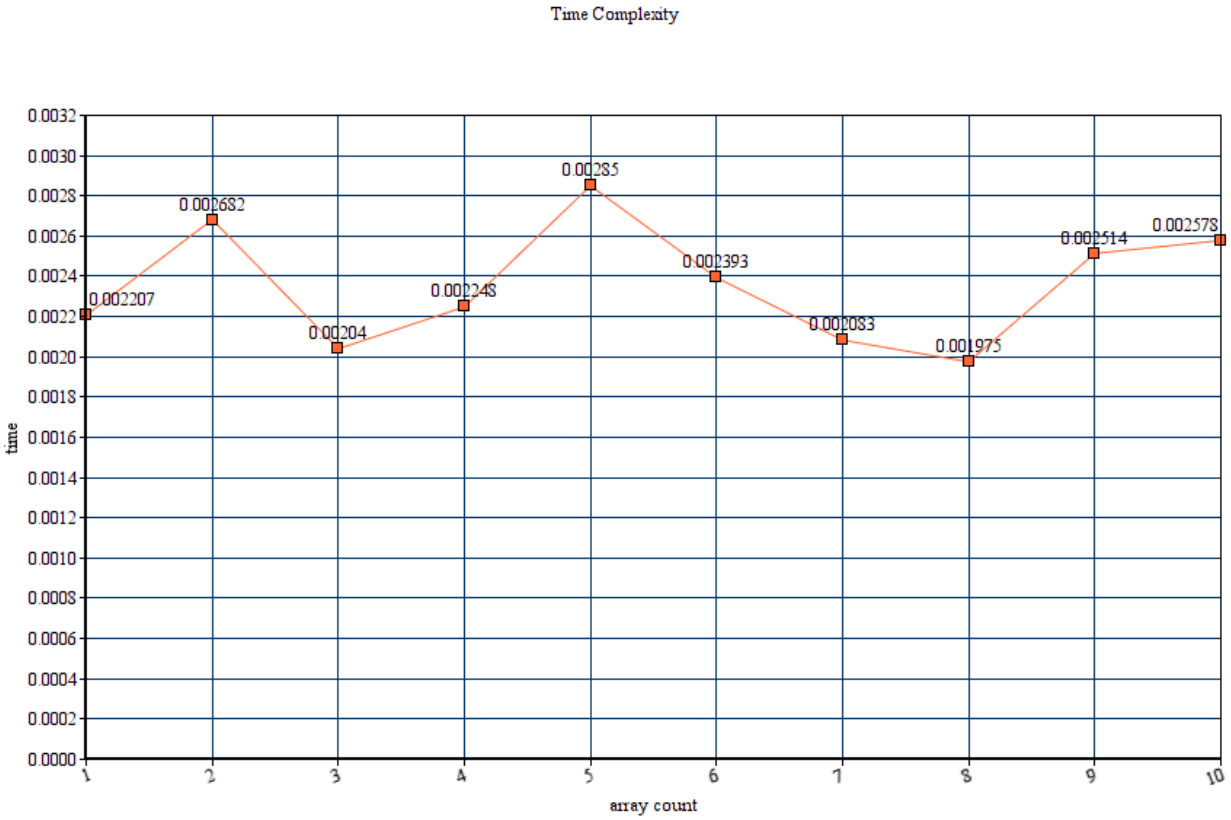
Best memory - O(n log(n))

Average - O(n log(n))

Worst - O(n^2) and O(log n)

### 2.1.3 Space Complexity

The space complexity of quick sort is O(n). This is an improvement over other divide and conquer sorting algorithms, which take O(n log(n)) space.

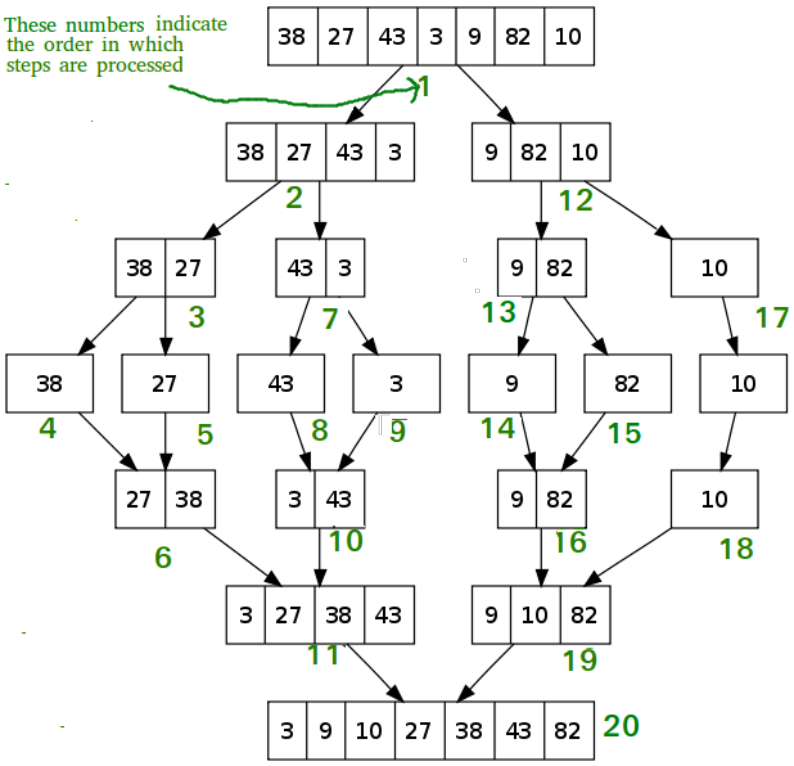



```
quickSort
"/Users/anadvorac/Documents/FAF/anul
Time of execution: 0.00220704
Time of execution: 0.00268197
Time of execution: 0.00204015
Time of execution: 0.00224805
Time of execution: 0.00285006
Time of execution: 0.00239301
Time of execution: 0.00208306
Time of execution: 0.00197506
Time of execution: 0.00251389
Time of execution: 0.00257802
```

## 2.2 Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The major portion of the algorithm is given two sorted arrays, and we have to merge them into a single sorted array. The whole process of sorting an array of N integers can be summarized into three steps:

1. Divide the array into two halves;
2. Sort the left half and right half using the same recurring algorithm;
3. Merge the sorted halves.




### 2.2.1 Source code

Anexa 2, Anexa A, Anexa B

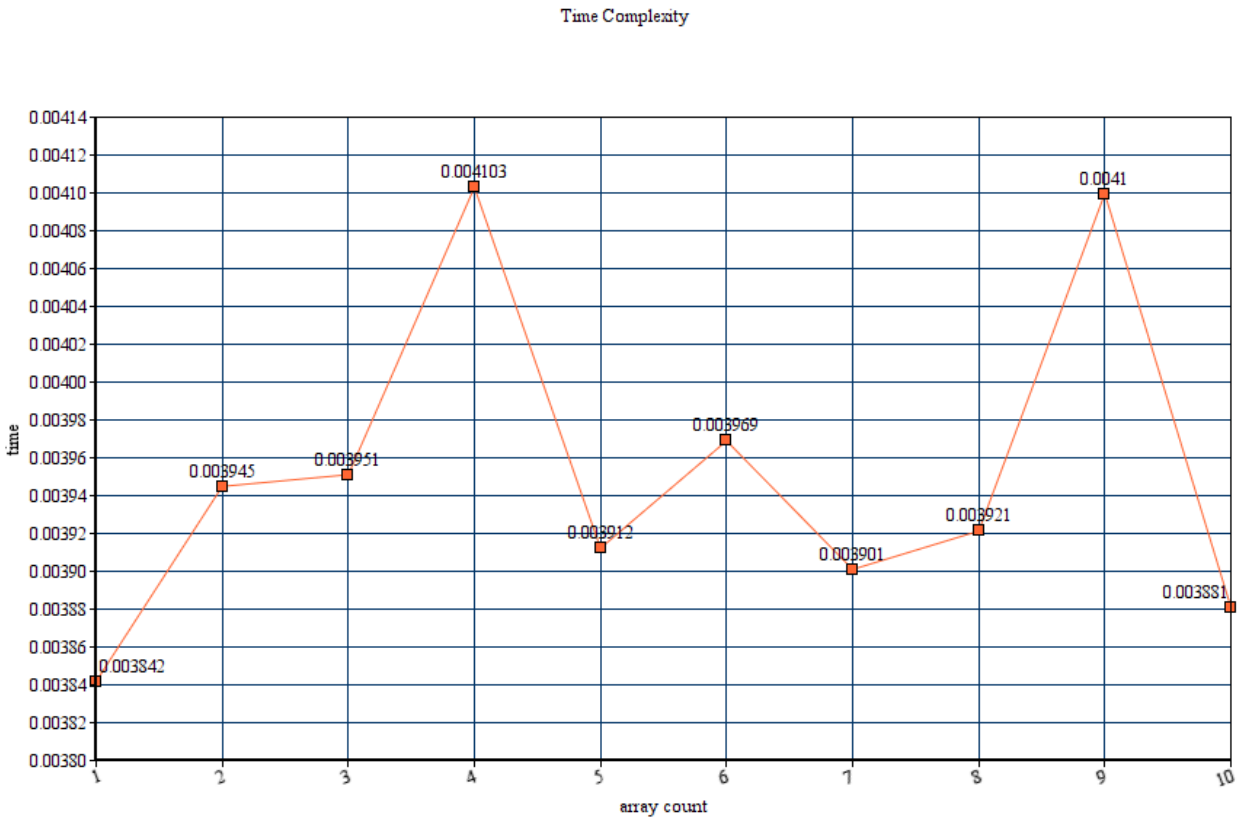
### 2.2.2 Time Complexity

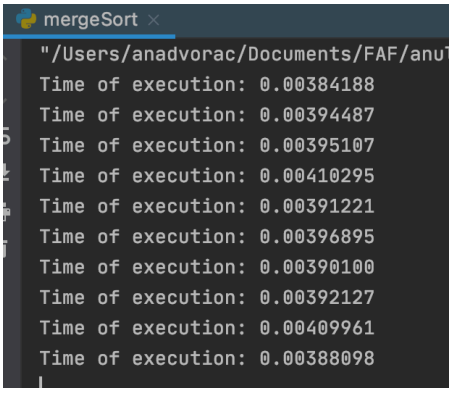
O(n*log(n)).

The time complexity for the Merge Sort might not be obvious from the first glance. The log(n) factor that comes in is because of the recurrence relation we have mentioned before

### 2.2.3 Space Complexity

O(n)




## 2.3 Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

What is Binary Heap?

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

The process of sorting an array can be summarised in 3 steps:

1. Build a MaxHeap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree;
3. Repeat step 2 if the ice of heap is greater than 1.

### 2.3.1 Source Code

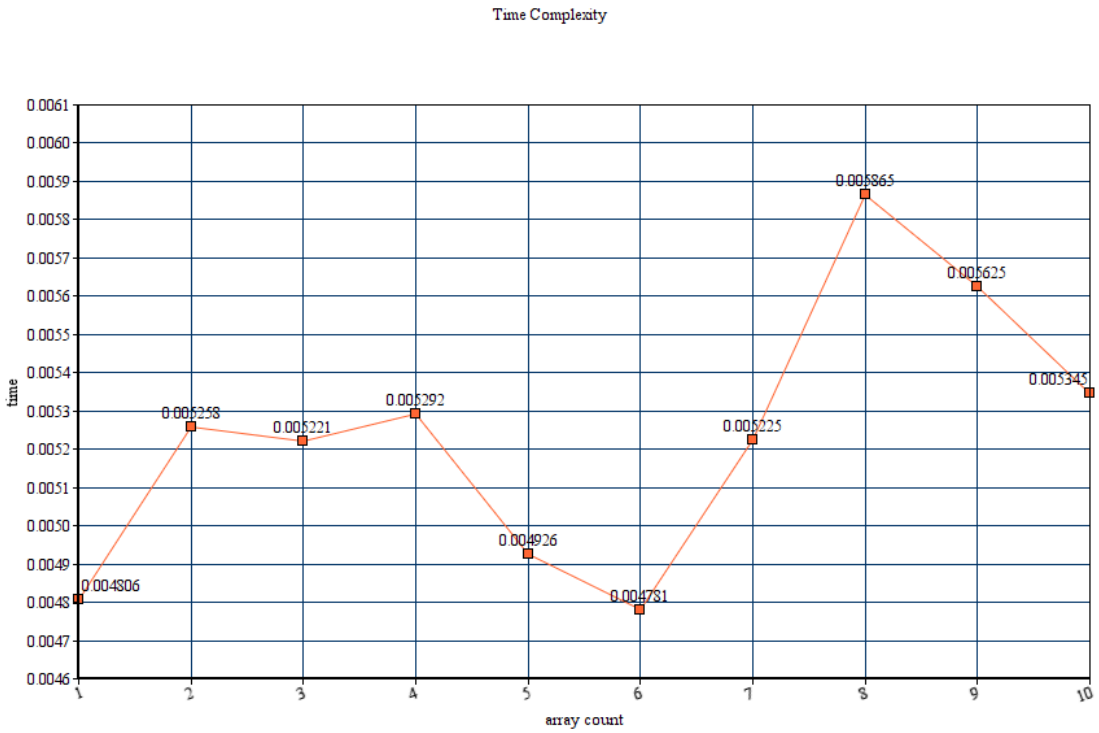
Anexa 3,  Anexa A, Anexa B

### 2.3.2 Time Complexity

Time complexity of heapify is O(log n). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(n log n).

### 2.3.3 Space Complexity

O(1)



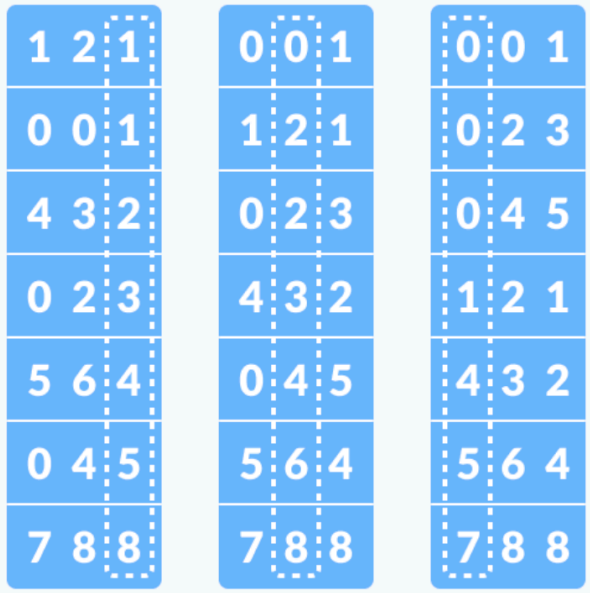

```
heapSort
"/Users/anadvorac/Documents/FAF/
Time of execution: 0.00480580
Time of execution: 0.00525808
Time of execution: 0.00522113
Time of execution: 0.00529218
Time of execution: 0.00492597
Time of execution: 0.00478101
Time of execution: 0.00522494
Time of execution: 0.00586486
Time of execution: 0.00562501
Time of execution: 0.00534511
```

## 2.4 Radix Sort

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.




sorting the integers according to units, tens and hundreds place digits

How radix sort works?

1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements;
2. Now, you go through each significant place one by one. Use any stable sorting techniques to sort digits at each significant place;
3. Sort the elements based on units place digits (X = 0);
4. Sort the elements based on digits at tens place;
5. Finally, sort the elements based on digits at hundreds place.

### 2.4.1 Source Code

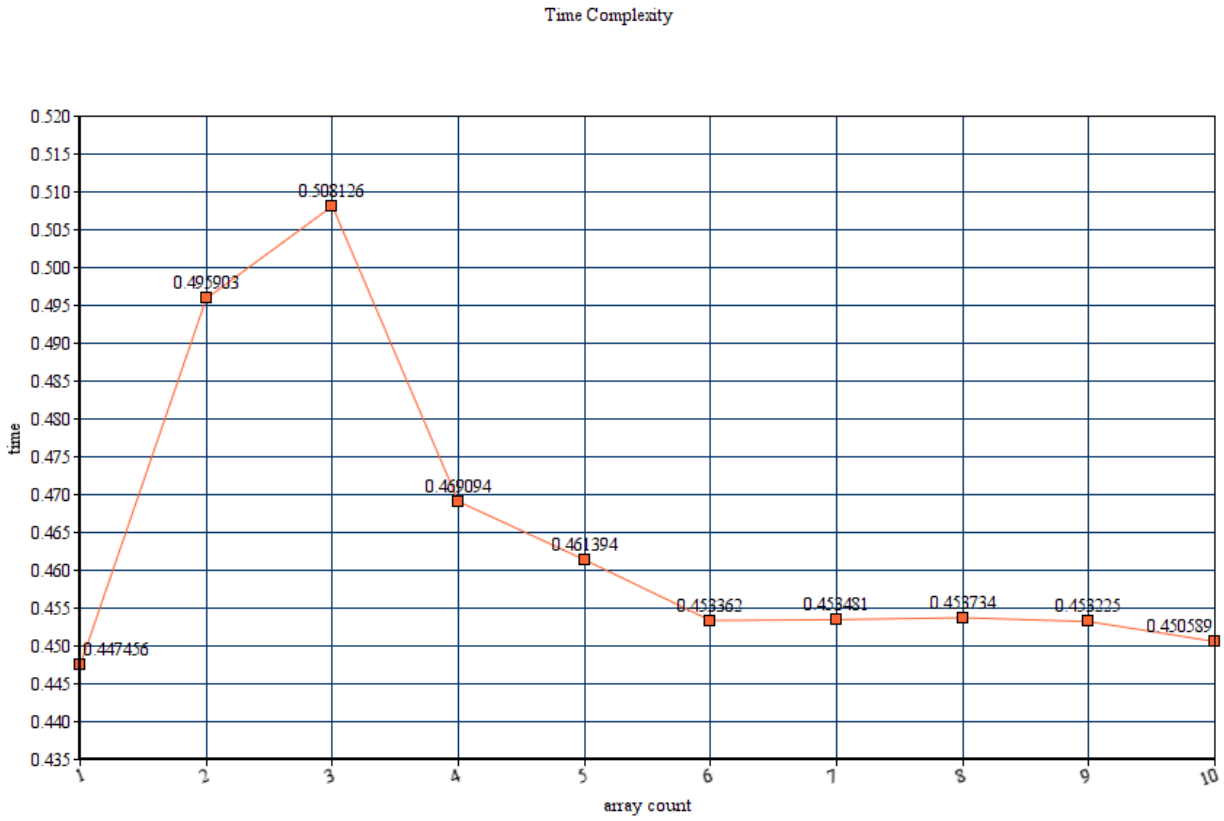
Anexa 4, Anexa A, Anexa B

### 2.4.2 Time Complexity

O(d(n + k), where d is number of cycles and O(n + k) is the time complexity of counting sort

### 2.4.3 Space Complexity

O(n + 2^d)




```
radixSort
"/Users/anadvorac/Documents/FAF/an
Time of execution: 0.44745588
Time of execution: 0.49590302
Time of execution: 0.50812602
Time of execution: 0.46909428
Time of execution: 0.46139383
Time of execution: 0.45336199
Time of execution: 0.45348096
Time of execution: 0.45373416
Time of execution: 0.45322490
Time of execution: 0.45058918
```

# 3. RESULTS REVIEW

| | Quick Sort | Merge Sort | Heap Sort | Radix Sort |
|---|---|---|---|---|
| 1 | 0.00220704 | 0.00384188 | 0.00480580 | 0.44745588 |
| 2 | 0.00268197 | 0.00394487 | 0.00525808 | 0.49590302 |
| 3 | 0.00204015 | 0.00395107 | 0.00522113 | 0.50812602 |
| 4 | 0.00224805 | 0.00410295 | 0.00529218 | 0.46909428 |
| 5 | 0.00285006 | 0.00391221 | 0.00492597 | 0.46139383 |
| 6 | 0.00239301 | 0.00396895 | 0.00478101 | 0.45336199 |
| 7 | 0.00208306 | 0.00390100 | 0.00522494 | 0.45348096 |
| 8 | 0.00197506 | 0.00392127 | 0.00586486 | 0.45373416 |
| 9 | 0.00251389 | 0.00409961 | 0.00562501 | 0.45322490 |
| 10 | 0.00257802 | 0.00388098 | 0.00534511 | 0.45058918 |
| Average time | 0.002357031 | 0.003952479 | 0.005234409 | 0.464636422 |

In the table above, are stored all the values of time execution, that were obtained running the sorting algorithms. From these results, we can observe that Quick Sort is the fastest from these 4 sorts.

# 4. CONCLUSION

During this laboratory work, I learnt about sorting algorithms. I found out how these algorithms are classified based on number of swaps, number of comparisons, extra space, recursion or non-recursion, stability. I mastered writing code for sorting algorithms and measuring their execution time. Then, by comparing obtained results for execution time, I deduce that the fastest from these 4 algorithms is QuickSort - an efficient divide and conquer sorting algorithm with average time complexity of O(n log(n)) and space complexity of O(n)

# 5. ANEXA

## 5.1 Anexa A

```
import time
import random


def array_time_execution(sort_function, array_size = 10, array_count = 1):


    for i in range(array_count):
        array = [random.randint(1, 100) for j in range(array_size)]


        start = time.time()
        sort_function(array)
        stop = time.time()


        print("Time of execution: {:.8f}".format(stop - start))
```

## 5.2 Anexa B

```
import time


def execution_of_array(sort_function):
    array_count = [[34, 57, 19, 83, 13, 73, 72, 81, 95, 56],
            [90, 94, 71, 88, 70, 20, 19, 73, 62, 92, 80, 45, 7, 89, 57, 76, 93, 82, 97, 78, 11, 75, 74, 47,
85,],
            [71, 38, 89, 80, 66, 56, 77, 53, 33, 44, 51, 61, 10, 3, 43, 98, 75, 68, 74, 63, 21, 23, 14, 73,
57, 67, 94, 59, 39, 25, 35, 83, 99, 96, 64, 97, 79, 8,37, 91]]
    for i in array_count:
        start = time.time()
        sort_function(array_count)
        stop = time.time()


        print("Time of execution: {:.8f}".format(stop - start))
```

## 5.3 Anexa 1

```
from arrayTimeExecution import array_time_execution
from executionOfArray import execution_of_array


def partition(array, low, high):
    i = low - 1
    pivot = array[high]

    for j in range(low, high):
        if array[j] < pivot:
            i += 1
            array[i], array[j] = array[j], array[i]

    array[i+1], array[high] = array[high], array[i+1]
    return i + 1

def quick_sort(array, low, high):
    size = high - low + 1
    stack = [0] * (size)

    top = -1

    top = top + 1
    stack[top] = low
    top = top + 1
    stack[top] = high

    while top >= 0:
        high = stack[top]
        top = top - 1
        low = stack[top]
```

```
        top = top - 1

        p = partition(array, low, high)

        if p - 1 > low:
            top = top + 1
            stack[top] = low
            top = top + 1
            stack[top] = p - 1

        if p + 1 < high:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = high

def wrapped_call(array):

    low = 0
    high = len(array) - 1
    quick_sort(array, low, high)

array_time_execution(wrapped_call, 1000, 10)
# execution_of_array(wrapped_call)
```

## 5.4 Anexa 2

```
from arrayTimeExecution import array_time_execution
from executionOfArray import execution_of_array

def merge_sort(array):
    if len(array) > 1 :
        middle = len(array) // 2
```

```
            left = array[:middle]
        right = array[middle:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                array[k] = left[i]
                i += 1
            else:
                array[k] = right[j]
                j += 1
            k += 1

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

        while j < len(right):
            array[k] = right[j]
            j += 1
            k += 1

array_time_execution(merge_sort, 1000, 10)
# execution_of_array(merge_sort)
```

## 5.5 Anexa 3

```
from arrayTimeExecution import array_time_execution
from executionOfArray import execution_of_array


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

    if left < n and array[largest] < array[left]:
        largest = left

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

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

def heap_sort(array):
    n = len(array)
    for i in range(n // 2 - 1, -1, -1):
        heapify(array, n, i)

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

array_time_execution(heap_sort, 1000, 10)
# execution_of_array(heap_sort)
```

## 5.6 Anexa 4

```
from executionOfArray import execution_of_array
from arrayTimeExecution import array_time_execution


def counting_sort(array, exp1):
    n = len(array)
    output = [0] * (n)
    count = [0] * (10)

    for i in range(0, n):
        index = (array[i] / exp1)
        count[int(index % 10)] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]
    i = n - 1
    while i >= 0:
        index = (array[i] / exp1)
        output[count[int(index % 10)] - 1] = array[i]
        count[int(index % 10)] -= 1
        i -= 1
    i = 0
    for i in range(0, len(array)):
        array[i] = output[i]

def radix_sort(array):
    max1 = max(array)
    exp = 1
    while max1 / exp > 0:
        counting_sort(array, exp)
        exp *= 10
```

```
# execution_of_array(radixSort)
array_time_execution(radix_sort, 1000, 10)
```