**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics, Department of Software Engineering and Automation**

**Calculability and complexity**

**Laboratory work Nr.2**

**STUDY AND EMPIRICAL ANALYSIS OF SORTING ALGORITHMS**

**Group: FAF-211**

**Created by: Iuliana Stețenco**

**Examined by: Fiștic Cristofor**

**Chișinău, 2023**

**TABLE OF CONTENTS**

[1 ALGORITHM ANALYSIS 3](#_Toc96206818)

[1.1 Scope and tasks 3](#_Toc96206819)

[1.2 Algorithm execution time 4](#_Toc96206820)

[1.3 Empirical analysis of the complexity of algorithms 5](#_Toc96206821)

[1.4 Introduction to Sorting Algorithms 7](#_Toc96206822)

[2 IMPLEMENTATION 8](#_Toc96206823)

[2.1 Merge Sort 8](#_Toc96206824)

[2.2 Quick Sort 10](#_Toc96206825)

[2.3 Heap Sort 14](#_Toc96206826)

[2.4 Cocktail Sort 17](#_Toc96206827)

[2.5 Comparative analysis of a common graph 20](#_Toc96206828)

[3 CONCLUSION 22](#_Toc96206829)

### 1 ALGORITHM ANALYSIS

In this chapter will be presented the scope and the tasks for this laboratory work, theoretical notes taken for this laboratory work, which will contain information about algorithm analysis, algorithm execution time and empirical analysis of the complexity of algorithms. Also, there will be a short introduction on sorting algorithms.

In the first subchapter will be listed the scope and the tasks which will be performed in order to complete this laboratory work.

In the second one will be mentioned the factors on which depends the running time of a program, the characteristics of a computational model also called a random-access computing machine, the purpose of the runtime calculation, the most unfavorable case, the most favorable case and the average execution time.

In the third subchapter will be discussed the purpose of the empirical analysis, the steps which are followed in order to perform this analysis, some explanations and advice about the performance of empirical analysis and how important it is.

In the fourth subchapter will be a short introduction about sorting algorithms.

#### 1.1 Scope and tasks

The main scope of this laboratory work is to study and to do an empirical analysis of sorting algorithms.

The tasks that must be performed in order to complete this laboratory work are:

1. Implement quickSort, mergeSort, heapSort and one more sorting algorithm;
2. Determine the properties of the input data in relation to which the analysis is performed;
3. Choose the metrics to compare the algorithms;
4. Perform the empirical analysis of the proposed algorithms;
5. Make a graphical presentation of the obtained data;
6. Make a conclusion about the laboratory work.

#### 1.2 Algorithm execution time

Often, in order to solve a problem, an algorithm must be chosen from several possible ones, two main selection criteria being contradictory:

* the algorithm should be easy to understand, code and debug;
* the algorithm should efficiently use computer resources, to have a short execution time.

If the program being written has to run a small number of times, the first requirement is more important. In this situation, the time to set up the program is more important than its running time so the simplest version of the program should be chosen. Otherwise, if the program is to be run a large number of times, with a large number of data to be processed, the algorithm that leads to a faster execution must be chosen. Moreover, in this situation, the simpler algorithm should be implemented first and we need to calculate the reduction of the execution time that the implementation of the complex algorithm would have.

The running time of a program depends on the following factors:

- input data;

- the quality of the code generated by the compiler;

- the nature and speed of execution of the program instructions;

- the complexity of the algorithm that underlies the program.

So, the running time is a function of its input, most of the time, not depending on the values ​​at the input, but on the amount of data.

By T(n) is denoted the execution time of an algorithm intended to solve a problem of size “n”. In order to estimate the execution time, a calculation model and a unit of measurement must be established.

A computational model also called a random-access computing machine is characterized by:

* Processing is performed sequentially;
* Basic operations are performed in a constant time regardless of the value of the operators;
* The access time to the information does not depend on its position (there are no differences between the processing of the first element and that of the last element of an array).

Establishing a unit of measurement means determining which are basic operations and considering as a unit of measure their execution. In this way the execution time will be expressed by the number of elementary operations performed. The basic operations are the arithmetic ones (addition, subtraction, multiplication, division), comparisons and the logical ones (negation, conjunction and disjunction).

The purpose of the runtime calculation is to allow the comparison of algorithms, sometimes it is sufficient to count only certain types of basic operations, called basic operations (for example, in the case of a search or sort algorithm, only comparison operations can be counted) and/or to consider that their execution time is unitary (although the operations of multiplication and division are more expensive than those of addition and subtraction in the analysis can be considered that they have the same cost).The execution time of the whole algorithm is obtained by summing the execution times of the component processing.

In assessing and comparing algorithms, the most unfavorable case is of particular interest because it provides the longest execution time relative to any fixed-size input data. On the other hand, for some algorithms the worst case is relatively common.

For the most favorable case analysis, it provides a lower time frame and can be useful for identifying inefficient algorithms (if an algorithm has a high cost in the most favorable case, then it cannot be considered a solution acceptable).

Sometimes, extreme cases (the most unfavorable and the most favorable) are rare, so the analysis of these cases does not provide enough information about the algorithm. In these situations, another measure of the complexity of the algorithms is useful, named the average execution time. This is an average value of the execution times calculated in relation to the probability distribution corresponding to the input data space.

#### 1.3 Empirical analysis of the complexity of algorithms

An alternative to mathematical analysis of complexity is the empirical analysis. It may be useful for:

* obtaining preliminary information on the complexity class of an algorithm;
* to compare the efficiency of two (or more) algorithms for solving the same problem;
* to compare the efficiency of several implementations of the same algorithm;
* to obtain information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established;

2. Choose the efficiency metric to be used (number of executions of an operation(s) or time

execution of all or a part of the algorithm);

3. Are established the properties of the input data in relation to which the analysis is performed;

4. The algorithm is implemented in a programming language;

5. Generating multiple sets of input data;

6. Run the program for each input data set;

7. The obtained data is analyzed.

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 to verify the accuracy of a theoretical estimate, then it is appropriate to use the number of operations performed. However, if the goal is to evaluate the behavior of the implementation of an algorithm, then the execution time is appropriate.

To perform an empirical analysis, a single set of input data is not enough, but several, which highlight the different characteristics of the algorithm. In general, it is good to choose data of different sizes so as to cover the range of all dimensions that will appear in practice. On the other hand, the analysis of different values ​​or configurations of the input data is also important. If an algorithm is analyzed that checks whether a number is prime or not and the test is done only for non-prime numbers or only for numbers that are prime then a relevant result will not be obtained. The same can happen for an algorithm of whose behavior depends on the degree of sorting of a table (if is chosen only the table almost sorted according to the desired criterion or tables ordered in reverse direction the analysis will not be relevant).

In order to empirically analyze the implementation of the algorithm in a programming language, sequences will have to be introduced, whose purpose is to monitor the execution. If the efficiency metric is the number of executions of an operation, then a counter is used which is incremented after each execution of the respective operation. If the metric is the execution time, then the time of entry into the analyzed sequence and the time of exit must be recorded. Most programming languages ​​offer time-lapse measurement functions. It is important, especially if several tasks are active on a computer, to count only the time allotted for the execution of the analyzed program. Especially when it comes to measuring time, it is advisable to run the test program several times and calculate the average time.

After the execution of the program for the test data, the results are recorded and for the purpose of the analysis, either synthetic quantities (average, standard deviation, etc.) are calculated or pairs of shape points are graphically represented (problem size, efficiency measure).

#### 1.4 Introduction to Sorting Algorithms

A sorting algorithm in computer science is an algorithm that arranges the elements of a list. The most common orders are numerical and lexicographical, and they can be ascending or descending.

Sorting algorithms can be classified according to different criteria, there are some of them:

* Computational complexity: Refers to the best, worst and average case behaviours in terms of the size of the list. Good behaviour for standard serial sorting algorithms is O(n log n), with parallel sort in O(log2n), and bad behaviour is O(n2).The ideal behaviour for a serial sort is O(n).The optimal parallel sorting a is O(log n).
* The number of swaps: This method categorizes sorting algorithms based on the number of swaps they;
* Memory usage: Some sorting algorithms are "in place" and they use O(1) memory beyond the items being sorted or O(log n) memory to establish temporary sorting sites;
* Recursion: Sorting algorithms are either recursive, for example Quick Sort or non-recursive like Selection Sort and Insertion Sort, with some algorithms combining the two as Merge Sort;
* Stability: Stable sorting algorithms maintain the relative order of records with equal keys. To put it another way, stable sorting keeps the position of two equal components that are comparable to one another. Insertion Sort, Bubble Sort, and Radix Sort are some examples of this kind of sorting;
* Adaptability: Whether or not the pre-sorted input affects the running time. Algorithms that take this into account are known to be adaptive. Some adaptive sorting algorithms are: Bubble Sort, Insertion Sort and Quick Sort. On the other hand, some non-adaptive sorting algorithms are: Selection Sort, Merge Sort, and Heap Sort;
* Internal Sorting: Internal sorting algorithms use the main memory exclusively during the sorting process. Bubble Sort, Insertion Sort, and Quick Sort are some of the common algorithms that exploit this sorting characteristic.
* External sorting: Sorting algorithms that utilise external memory during the sorting process. They are comparatively slower than internal sorting algorithms. For example, merge sort algorithm. It sorts chunks that each fit in RAM, then merges the sorted chunks together.

### 2 IMPLEMENTATION

There are several sorting algorithms, for this laboratory work were chosen four of them and the empirical analysis was performed by measuring and comparing the programs’ execution time in seconds, giving different sets of data as input and running the programs, in order to see which implementation method is the best one in terms of timing and efficiency.

To do a proper empirical analysis and implementation must be established some characteristics which were listed as tasks in the first chapter:

1. Input data:

In order to determine the properties of the input data in relation to which the analysis is performedhas been prepared a set of 10 arrays with different number of elements from 100 to 4500 with different steps, their values are represented as integers in the range from -100 to 100 as the input data, because having more input data with different ranges/steps grants the possibility to point out some features of the algorithms.

1. Comparison metric:

As for comparison metric the time (measured in seconds) of program’s execution was chosen to evaluate the behavior of the algorithms. The algorithm should efficiently use computer resources, to have a short execution time. Our programs are to be run a pretty number of times, with a set of a decent number of data to be processed, so the algorithm that leads to a faster execution must be chosen. In Python’s “time” library, there is a method “time()”, which can be used to measure the execution time taken by the given code. All the manipulations used for determining the time taken for execution of the program can be observed in the code which will be presented further in this report.

#### 2.1 Merge Sort

Merge sort is one of the most prominent divide-and-conquer sorting algorithms in the modern era. It can be used to sort the values in any traversable data structure such as a list. Merge sort works by splitting the input list into two halves, repeating the process on those halves, and finally merging the two sorted halves together.

The implementation of Merge Sort can be observed in the Figure 1 below.

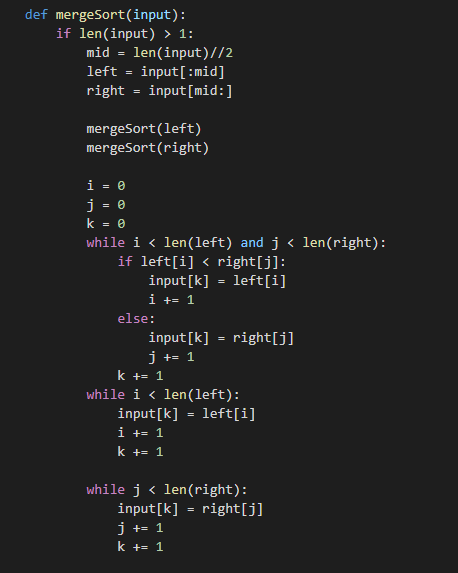


Figure 1 – Merge Sort Implementation

In the Figure 1 can be seen the function “mergeSort” that performs the algorithm and takes as the parameter the input array. It finds the middle of the array and calls recursively the left and right halves of it. Iterators “i” and “j” to transverse these two halves and the “k” one is the iterator for the main array. In the first while function is copied the data to the left and right arrays, in the last two ones is checked if there were left any elements.

Table 1 provides the results of the data used for analysis, the input column will be presented the number of elements in the input array, and the last column is for the run time for each input.

Table 1. Input size and the execution time for Merge Sort

|  |  |  |
| --- | --- | --- |
| Nr. | Input size(n) | Time (s) |
| 1.  2. | 100  500 | 0.0  0.0009982585906982422 |

Continuation of Table 1.

|  |  |  |
| --- | --- | --- |
| Nr. | Input size(n) | Time (s) |
| 3.  4.  5.  6.  7.  8.  9.  10. | 1000  1500  2000  2500  3000  3500  4000  4500 | 0.0003266334533691406  0.0016598701477050781  0.003270864486694336  0.0051801204681396484  0.007105827331542969  0.009255170822143555  0.011675596237182617  0.013243913650512695 |

From Table 1 can be observed that the Merge Sort performs very well on arrays that size is 100, but starting from this point its execution time increases considerably, as bigger is the number of elements in the array list as much rises the amount of execution time for performing the sort. The graphic representation of the data provided in the table can be observed in Figure 2.

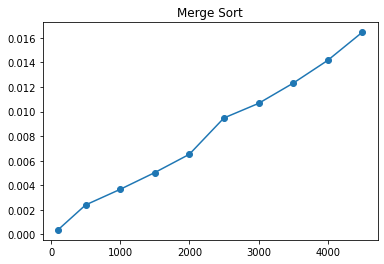


Figure 2 – Execution time of Merge Sort

#### 2.2 Quick Sort

Quicksort is one of the most popular sorting algorithms which is used very often, right alongside Merge Sort. Its worst-case running time is as bad as Selection Sort's and Insertion Sort's: O(n2). When Quick Sort is implemented well, it can be somewhat faster than the merge sort and about two or three times faster than heapsort.

It's a good example of an efficient sorting algorithm, with an average complexity of O(n log n). Part of its popularity also derives from the ease of implementation.

Quicksort is a representative of three types of sorting algorithms:

* Divide and conquer: Quicksort splits the array into smaller arrays until it ends up with an empty array, or one that has only one element, before recursively sorting the larger arrays.
* In place: Quicksort doesn't create any copies of the array or any of its subarrays. It does however require stack memory for all the recursive calls it makes.
* Unstable: A stable sorting algorithm is one in which elements with the same value appear in the same relative order in the sorted array as they do before the array is sorted. An unstable sorting algorithm doesn't guarantee this, it can of course happen, but it isn't guaranteed.

The process fundamental to the ‘QuickSort’ algorithm is the partition. The way partition works is by first selecting a pivot. Options for pivots include: first element, last element, random element and middle element.

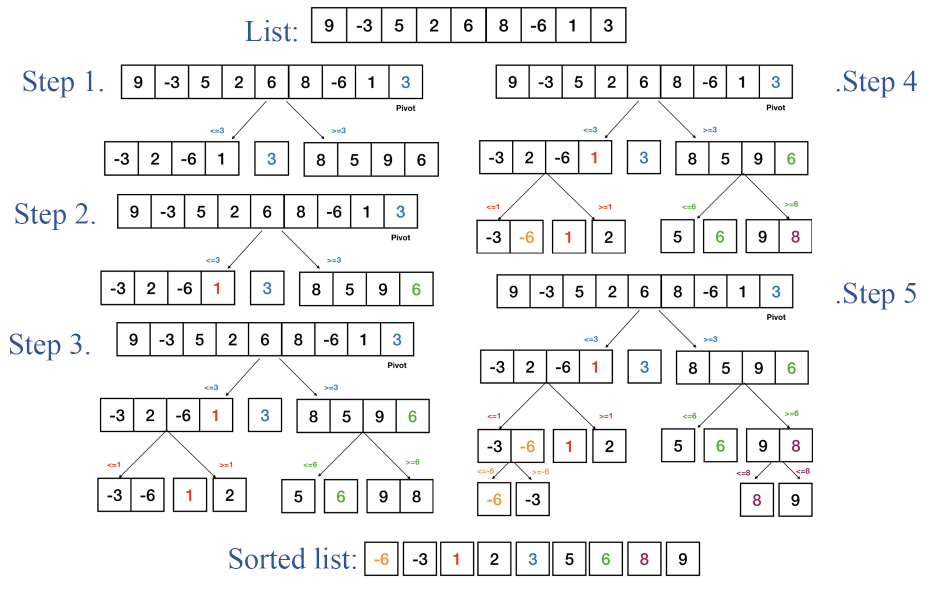
Upon selecting the pivot, the elements of the array are partitioned based on the pivot value. Namely, for an input array and pivot, put all elements smaller than the pivot before the pivot and all elements greater than the pivot after it. To see this process an example will be provided in Figure 1.

Figure 3 - Example of Quick Sort algorithm performance

For the list shown in Figure 3, in Step 1 is selected element “3” as a pivot, in the next step are created two sublists, the left one contains values smaller than the pivot one and the right sublist contains values which are greater, then is starting the process of finding the pivots for the sublists, for the left one it is “1” and for the right one “6”. In Step 3 the sublists are partitioned around their pivots, and then in Step 4 are selected the pivots for the last set of subsets “-6” for the left one and “8” for the right. In the last step is completed the final partition and a sorted list is elaborated.

The implementation in Python programming language can be seen below in Figure 3.

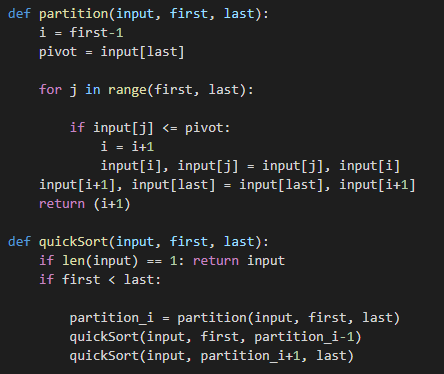


Figure 3 – Implementation of Quick Sort

In the Figure 2, can be seen function “partition” that includes as parameters the input, first and last elements, the last element is selected as pivot as shown in 6th line of the code, next if the current element is smaller or equal to the selected pivot the index “i” of smaller element will increment and the swap will be performed. The function “quickSort” implements the Quick Sort, if the lists’ length is equal to 1 it will return the list, in case the first element is smaller than the last one , take “partition\_i” as partitioning index that uses the function “partition” and then sort separately elements before and after partition using “quickSort” function.

Table 1 provides the results of the data used for analysis, the input column will be presented the number of elements in the input array, and the last column is for presenting the run time for each input.

Table 2. Input size and the execution time for Quick Sort

|  |  |  |
| --- | --- | --- |
| Nr. | Input size(n) | Time (s) |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | 100  500  1000  1500  2000  2500  3000  3500  4000  4500 | 0.0016100406646728516  0.029088258743286133  0.10688161849975586  0.24027490615844727  0.4336569309234619  0.835277795791626  1.6683614253997803  1.4447643756866455  1.8400378227233887  2.353222608566284 |

Observing Table can be seen that Quick Sort has a great performance on lists which size is smaller than 500 and starting with the list with 500 elements it starts to take more time to perform.

In order to see the changes in the execution time of the program using Quick Sort algorithm for sorting lists is created a graph that is presented in Figure 4. It shows how the running time of the algorithm changes on all the input sizes given in Table 2 and using the time shown also in that table.

In Figure 4 is clearly seen that the execution time starts to rise considerably starting with the array list that has 1000 elements, reaching its peak at the one with 4500 values.

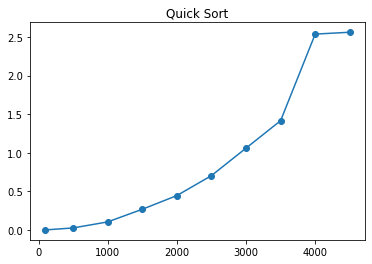


Figure 4 – Execution time of Quick Sort

#### 2.3 Heap Sort

Heap Sort is another example of an efficient sorting algorithm, it represents a comparison-based sorting technique based on Binary Heap data structure. Its main advantage is that it has a great worst-case runtime of O(n log n) regardless of the input data. As the name suggests, Heap Sort relies heavily on the heap data structure. Without a doubt, Heap Sort is one of the simplest sorting algorithms to implement and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations.

Heap Sort works by "removing" elements from the heap part of the array one-by-one and adding them to the sorted part of the array. It is an in-place algorithm, meaning that it requires a constant amount of additional memory, i.e. the memory needed doesn't depend on the size of the initial array itself, other than the memory needed to store that array. For example, no copies of the original array are necessary, and there is no recursion and recursive call stacks. The simplest implementation of Heap Sort usually uses a second array to store the sorted values.

Heap Sort is unstable, meaning that it does not maintain the relative order of elements with equal values. This is not an issue with primitive types (like integers, characters etc.) but it can be a problem when complex types like objects are sorted.

For example, imagine a custom class “Student” with the “age” and “name” fields, and several objects of that class in an array, including a student called "Sebastian" aged 20 and "Angela", also 20 years old appearing in that order. If the array intents to be sorted by age, there would be no guarantee that "Sebastian" would appear before "Angela" in the sorted array, even though they appeared in that order in the initial array, it can happen, but it's not guaranteed.

A heap data structure is a complete binary tree that meets the heap property, in which every given node is always:

* greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**;
* smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property.**

For example, simply put, a min-heap is a tree-based data structure in which every node is smaller than all of its children. Most often a binary tree is used.

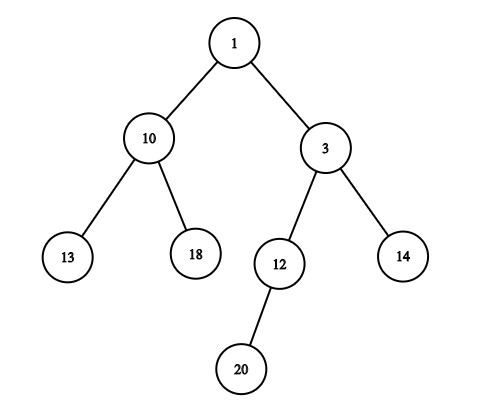
In Python heaps have three supported operations: delete\_minimum(), get\_minimum() and add(). Only the first element in the heap can be deleted, after which it's re-sorted. Heaps re-sort themselves after an element is added or removed, so that the smallest element is always in the first position. But this doesn’t mean that heaps are sorted arrays, the fact that every node is smaller than its children isn't enough to guarantee that the whole heap is in ascending order. An example of heap can be seen below in Figure 5.

Figure 5 – Heap example

As can be seen in the above example from Figure 5***,*** it does fit the description of a heap but is not sorted. The crucial advantage of the heap data structure is leveraged when using it in Heap Sort is that the next smallest element is always the first element in the heap.

Due to the way heaps sort elements after an element is removed, the complexity of the next smallest element moving to the first position, while keeping the array a heap still, takes O(log n) time, which is a highly efficient operation.

Python programming language provides methods for creating and using heaps so there is no need to implement manually:

* heappush(list,item): Adds an element to the heap, and re-sorts it afterward so that it remains a heap. Can be used on an empty list;
* heappop(list): Pops (removes) the first (smallest) element and returns that element. The heap remains a heap after this operation, so we don't have to call heapify();
* heapify(list): Turns the given list into a heap. It is worth noting that this method exists even though we won't be using this since we don't want to change our original array.

The implementation of Heap Sort in Python programming language can be seen in Figure 6 from below.

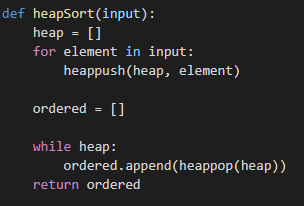


Figure 6 – Heap Sort implementation

In Figure 6 can be seen the function “heapSort” function that performs the sorting of the input array. As mentioned before the “heappush(input)” function adds an element to the heap, and re-sorts it afterward so that it remains a heap. Can be used on an empty list, it was declared before. The “heappop(heap)” function removes the first smallest element and returns it. The heap remains a heap after this operation.

Table 3 provides the results of the data used for analysis, the input column will be presented the number of elements in the input array, and the last column is for presenting the run time for each input.

Table 3. Input size and the execution time for Heap Sort

|  |  |  |
| --- | --- | --- |
| Nr. | Input size(n) | Time (s) |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | 100  500  1000  1500  2000  2500  3000  3500  4000  4500 | 0.0006656646728515625  0.002288341522216797  0.004593849182128906  0.006410837173461914  0.009936094284057617  0.014779329299926758  0.015372514724731445  0.017462491989135742  0.020421504974365234  0.025890111923217773 |

The graphic representation of the dates from this table can be seen in Figure 7 from below.

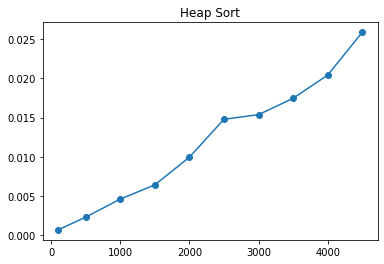


Figure 7 – Execution time of Heap Sort

#### 2.4 Cocktail Sort

Cocktail Sort is a variation ofBubble [sort](http://quiz.geeksforgeeks.org/bubble-sort/) however it still retains the same worst case computational complexity O(n2), best case O(n) occurs when the array is already sorted.

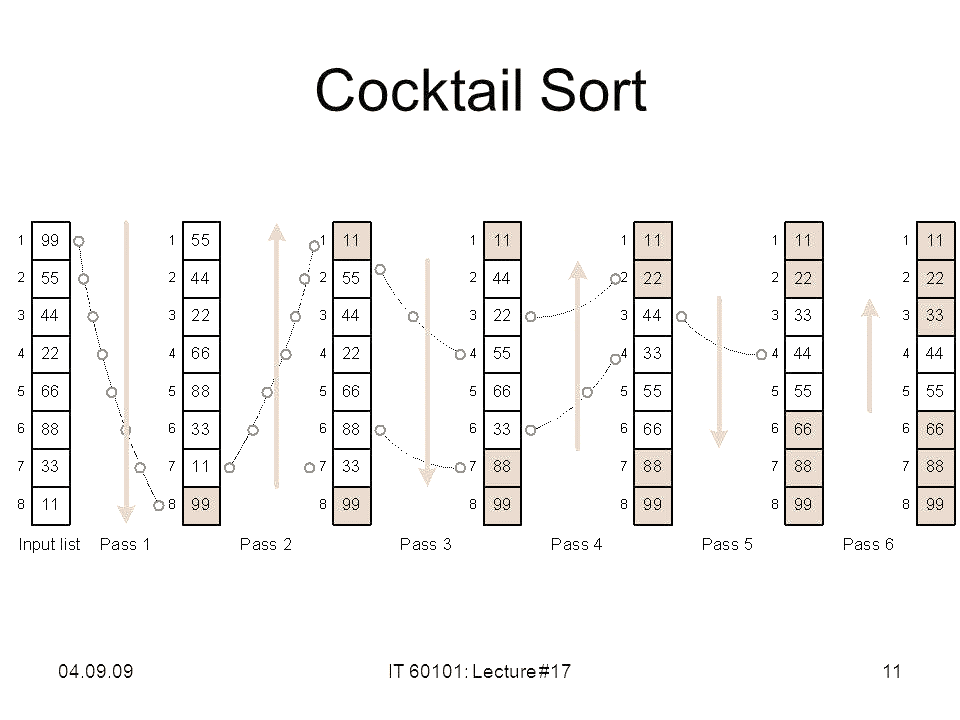
The Cocktail algorithm differs from a bubble sort in that it sorts in both directions on each pass through the list as shown in Figure 10. This sorting algorithm is more difficult to implement than the bubble sort. The first rightward pass will shift the largest element to its correct place at the   
end, and the following leftward pass will shift the smallest element to its correct place at the beginning. The second complete pass will shift the second largest and second smallest elements to their correct places, and so on.

Figure 8 – Example of Cocktail Sort process of sorting

Each iteration of the Cocktail Sort’s algorithm is broken up into 2 stages:

1. The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if value on the left is greater than the value on the right, then values are swapped. At the end of first iteration, largest number will reside at the end of the array.
2. The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

Much like bubble sort, cocktail sort has very little relevance in the real world and is mainly used to teach algorithms.

The Cocktail Sort implementation in Python programming language can be seen below in Figure 9.

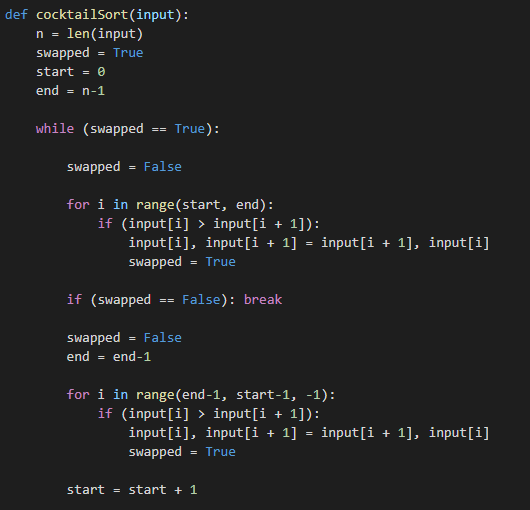


Figure 9 – Cocktail Sort implementation

In Figure 9 is presented the function “cocktailSort” that performs the sorting algorithm. At the start of the while loop is reset the “swapped” flag, because it might be true from one of the previous iterations, then there is a loop from the left to the right, in case nothing was moved then the array is sorted, otherwise the flag is reset to be used in the next stage.

There are performed movements from the end point back by one, because the item at the end is in the rightful spot. Then, in the for loop, from the right to left are performed the same comparison processes, after this the starting point is increased due to the fact that the last stage would have moved the next smallest number to its rightful spot.

Table 4 provides the results of the data used for analysis, the input column will be presented the number of elements in the input array, and the last column is for presenting the run time for each input.

Table 4. Input size and the execution time for Cocktail Sort

|  |  |  |
| --- | --- | --- |
| Nr. | Input size(n) | Time (s) |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | 100  500  1000  1500  2000  2500  3000  3500  4000  4500 | 0.0008733272552490234  0.0021352767944335938  0.0031518936157226562  0.005372524261474609  0.0066509246826171875  0.008533239364624023  0.01082611083984375  0.012476921081542969  0.01411747932434082  0.01627182960510254 |

Observing Table 4 can be seen that this algorithm has worse performance in terms of time that the other three, it performs well to the strings with less that 200 values, but starting with this point the time spent on performing it starts to increase, it is similar to the range 200-300, but after it the increase is seen more considerably, especially in the range from 2000 to 4500 it increases by several times.

To see the changes in execution time that occurs in the performance of the programm of Cocktail Sort algorithm a graph was plotted. It can be seen in Figure 12 on the next page. In it can be seen that the more substantial changes in the execution time start with the array with more than 1000 values, after this it rises exponentially.

Figure 12 from below and dates from the tables shows clearly that the performance of the Cocktail Sort is the worst one from the algorithms mentioned before, in terms the execution time.

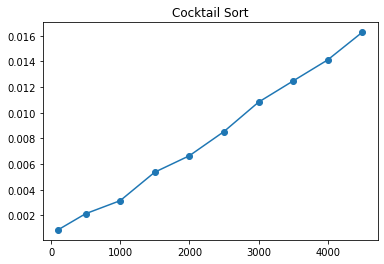


Figure 10 – Execution time of Cocktail Sort algorithm

#### 2.5 Comparative analysis of a common graph

To see clearly and compare in an easy way the execution time of all the sorting algorithms mentioned before a graph was plotted with the data collected from the Tables 1,2,3 and 4. This graph can be viewed in form of Figure 13.

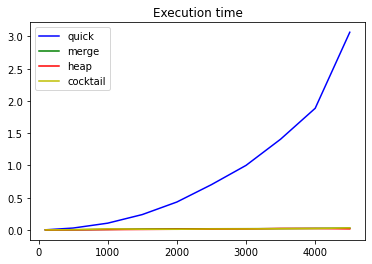


Figure 11 – Execution time of Merge Sort, Heap Sort, Cocktail Sort and Quick Sort

Analyzing Figure 11 can be seen clearly quick sort is the worst in terms of execution time. In order to be able to compare the rest 3 algorithms, let’s eliminate the quick sort and look at the graph one more time in Figure 12.

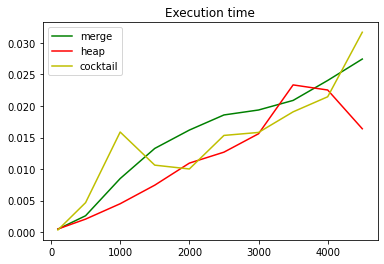


Figure 12 – Execution time of Merge Sort, Heap Sort, Cocktail Sort

The execution time is pretty non-liniar but we can see that Heap sort performs overall faster than Merge sort or Cocktail sort. The slowest sorting algorithm is the Cocktail sorting, because it sorts in both directions on each pass through the array list making the process of execution slower. It performs pretty good with smaller arrays but the execution time starts to increase exponentially.

### 3 CONCLUSION

To summarize, in this laboratory work were analyzed and implemented 4 sort algorithms in order to find the fastest one in performance. It was an interesting experience to use the empiric analysis on these implementations to see which one will have better results in term of execution time.

To perform the empiric analysis several steps were followed such as establishing the purpose of the analysis, in this case, to evaluate the behavior of the implementation of an algorithm, by finding what method performs better in term of execution time, the efficiency metric was defined as time of program’s execution in seconds, the proprieties of the input data was chosen as array lists of integer numbers in the range from -100 to 100, with different steps, the number of elements of these arrays were from 50 to 5000. The algorithms were implemented in one programming language “Python”, the multiple sets of input data was generated (10 arrays), all programs were executed for the established input data set and the obtained data was placed in form of tables and graphs and then analyzed.

This work helped to understand that there are different factors that can influence the execution time of a program such as input data, the quality of the code generated by the compiler, the nature and speed of execution of the program instructions and the complexity of the algorithm that underlies the program.

Implementing and analyzing four sorting algorithms can be said that the slower one turned out to be the Quick sorting. The most efficient method turned out to be the Heap Sort, showing the fastest results.

As final words in this report, can be said that this laboratory work was an useful one, because it familiarized the students with important concepts of empiric analysis and complexity of the sorting algorithms which will help to compare and determine which algorithms and implementations are most efficient ones and what are the factors that can influent their efficiency, for example the number of elements in the array.