Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #2

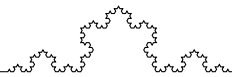
Study and empirical analysis of sorting algorithms. Analysis of quick sort, merge sort, heap sort and bubble sort

---

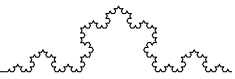
Author: Timur Cravtov
std. gr. FAF-231

Verified by: Cristofor Fistic
asist. univ

Chișinău
2025



Contents



1 Algorithm Analysis

1.1 Objective

Study and analyze different sorting algorithms.

1.2 Task

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.

1.3 Introduction

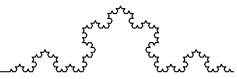
Data sorting is one of the most used algorithms in any software development process. Businesses need to sort their users based on income, countries, etc. Despite the fact that most programming languages already have default sorting algorithms implemented, understanding their theoretical part is crucial for analysing the global project algorithm analysis, as well as for possible manual improvements of the built-in algorithms for a certain set of data. For example, data might follow some patterns, for which the default algorithms can either result in a worst-case scenario or can be slower than a user-defined algorithm.

In current lab, the analysis of four sorting algorithms will be performed: quick sort, bubble sort, merge sort and heap sort.

Each algorithm will be performed on three types of the array: Random, Sorted, and Partially Sorted, with different sample sizes. This will help determine the worst case scenario of each algorithm and help undertand the use cases on each, based on the expected data.

2 Implementation

All the algorithms will be implemented in Python with no optimization. All the algorithms will use integers in sample data, however they are not limited to that since Python defines *number* datatype as both float and integer.



All the algorithms are designed to perform ascending sorting. It is not hard, but rather too tedious and unimportant for this course to make descending sorting a feature.

2.1 Bubble sort

Algorithm description

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

In the code snipped bellow, the pseudo code of the algorithm is provided.

```
1   FUNCTION BUBBLE_SORT(ARRAY)
2       SET N TO LENGTH OF ARRAY
3       FOR I FROM 0 TO N - 1 DO
4           FOR J FROM 0 TO N - I - 2 DO
5               IF ARRAY[J] > ARRAY[J + 1] THEN
6                   SET TEMP TO ARRAY[J]
7                   SET ARRAY[J] TO ARRAY[J + 1]
8                   SET ARRAY[J + 1] TO TEMP
9               END IF
10          END FOR
11      END FOR
12      RETURN ARRAY
13  END FUNCTION
```

Since in this algorithm we iterate throuh the whole array, and during the iteration we iterate again, the complexity is $n \times n - n$, or $O(n^2)$.

Implementation

```
1  def bubble_sort(a: list):
2      n = len(a)
3      for i in range(n - 1):
4          for j in range(n - i - 1):
5              if a[j] > a[j + 1]:
6                  a[j], a[j + 1] = a[j + 1], a[j]
7      return a
```

Listing 1: Bubble sort function

Results

The listing below shows the raw data lof algorithm execution

```
1  Bubble Sort Execution Times (seconds):
```

Array Size	Random	Sorted	Partially Sorted
500	0.011567	0.014354	0.009018
1000	0.066485	0.027019	0.043776
2000	0.195238	0.134995	0.153636
2500	0.328435	0.270692	0.224337
4000	1.047527	0.629422	0.658559
5000	1.313900	0.843947	0.956303

Listing 2: Bubble sort raw data

From figure 1, it becomes pretty obvious that the complexity of bubble sort in this form is $O(n^2)$,

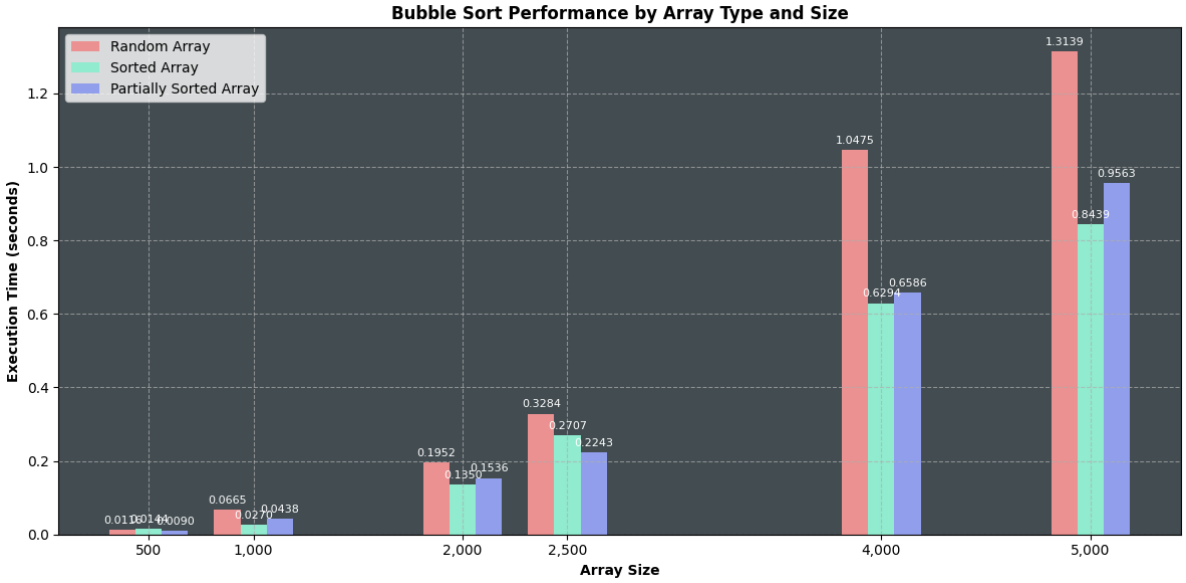


Figure 1: BubbleSort time execution

Possible improvements

If the array is already sorted, the bubble sort still performs in its worst-case scenario. However, if we track of the number of swaps in inner cycle, then we can stop the execution of bubble sort if no swaps are performed.

The result of the exection with this improvement, presented on figure 2 suggests that the improvement drastically decreases time of bubble sort for already sorted or almost sorted array.

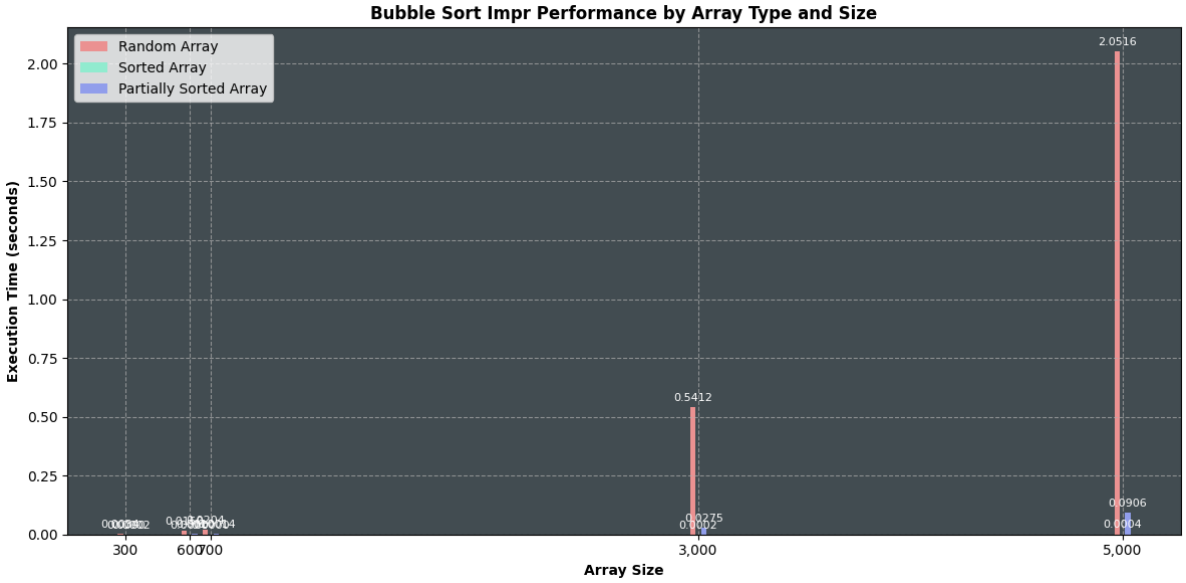


Figure 2: BubbleSort Improved time execution

2.2 Quick sort

Algorithm description

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The algorithm complexity for best case and average case is $O(n \log n)$. However, the worst case is $O(n^2)$, and example of that is provided below in this section.

There are different way to choose the pivot in Quick Sort:

1. *The last element*: Simply take the pivot as the last element of the array
2. *Random*: Each pivot is selected randomly
3. *Mean of the data*: Before selecting the pivot, the mean of the subarray is calculated and set as pivot. In this approach, it is guaranteed that the left and right subarrays have the same size.

For this implementation, I choose the first option.

```
1 FUNCTION QUICK_SORT(ARRAY, LOW, HIGH)
2     IF LOW < HIGH THEN
3         SET PIVOT_INDEX TO PARTITION(ARRAY, LOW, HIGH)
4
5         CALL QUICK_SORT(ARRAY, LOW, PIVOT_INDEX - 1)
6         CALL QUICK_SORT(ARRAY, PIVOT_INDEX + 1, HIGH)
```

```
    END IF
END FUNCTION
```

Implementation

```
def partition(arr, low, high):

    pivot = arr[high]

    i = low - 1

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

    swap(arr, i + 1, high)
    return i + 1

def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]


def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)

def _quick_sort(arr, low, high ):
    if low < high:

        # pi is the partition return index of pivot
        pi = partition(arr, low, high)

        _quick_sort(arr, low, pi - 1)
        _quick_sort(arr, pi + 1, high)
```

Listing 3: Quick sort function

Results

During the exection of dataset consisted with around 800 elements, the Python Compiler threw the exection, meaning the recursion is too deep. Since no improvement were made in this algorithm, the error was ignored, and sample data is adjusted.

```
Quick Sort Execution Times (seconds):

Array Size      Random          Sorted          Partially Sorted
-------------------------------------------------------------------
30              0.000026        0.000054        0.000031
100             0.000072        0.000614        0.000174
200             0.000182        0.002938        0.000507
300             0.000256        0.007163        0.001050
400             0.000385        0.012486        0.001708
500             0.000648        0.019454        0.002551
600             0.000689        0.029312        0.003890
700             0.000894        0.048410        0.005669
```

Listing 4: Quick sort raw data

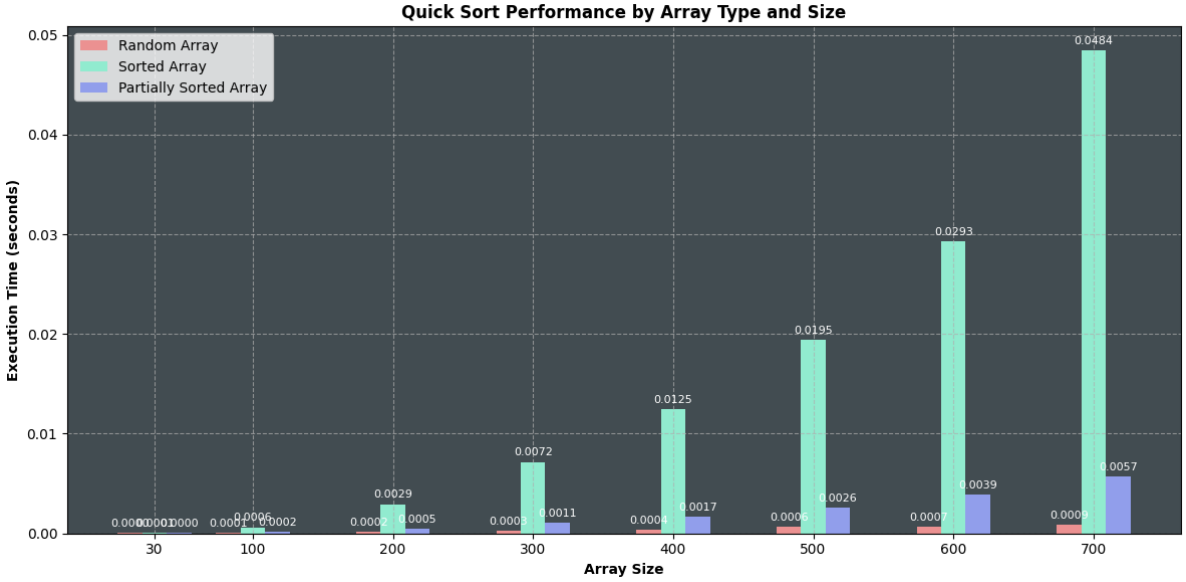
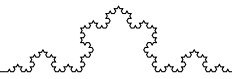


Figure 3: Quick Sort time execution

It becomes clear from the data that the more sorted the data is, the more times it needs, being $O(n^2)$ in worse case scenario of sorted array. It can be explain in the following way: the pivot element, taken each time the last, devides the subarray in the element itself and the rest of the array. It doesn't performs any checks which doesn't stop the execution.



2.3 Heap sort

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

Implementation

```
def heapify(arr, n, i):

    largest = i

    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)

def heap_sort(arr):

    n = len(arr)

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

    for i in range(n - 1, 0, -1):

        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
```

Listing 5: Heap sort function

Results

```
Heap Sort Execution Times (seconds):

Array Size      Random          Sorted          Partially Sorted
-----------------------------------------------------------------
1000            0.003995        0.006371        0.017070
10000           0.034197        0.035542        0.045354
20000           0.083229        0.097711        0.079125
50000           0.234774        0.223257        0.227682
200000          0.998416        0.983055        1.076050
```

Listing 6: Heap sort raw data

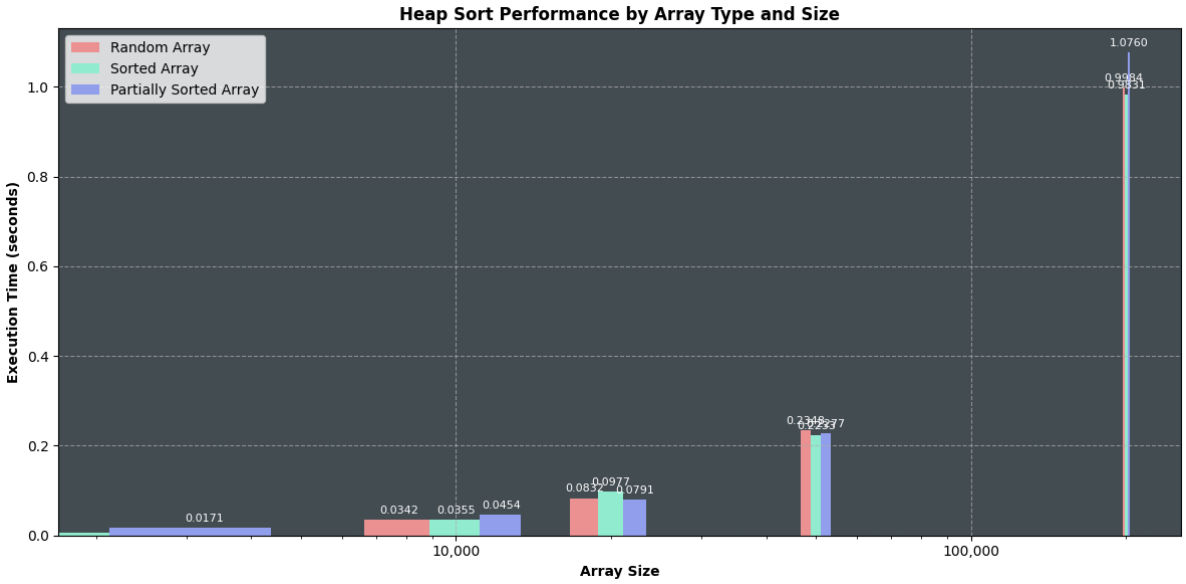


Figure 4: Heap Sort time execution

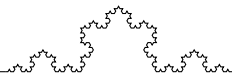
2.4 Merge sort

Algorithm description

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In listing below there is provided pseudocode of the algorithm.

```
FUNCTION MERGE_SORT(ARRAY, LEFT, RIGHT)
```



```
    IF LEFT < RIGHT THEN
        SET MID TO (LEFT + RIGHT) / 2

        CALL MERGE_SORT(ARRAY, LEFT, MID)
        CALL MERGE_SORT(ARRAY, MID + 1, RIGHT)

        CALL MERGE(ARRAY, LEFT, MID, RIGHT)
    END IF
END FUNCTION

FUNCTION MERGE(ARRAY, LEFT, MID, RIGHT)
    CREATE LEFT_SUBARRAY FROM ARRAY[LEFT TO MID]
    CREATE RIGHT_SUBARRAY FROM ARRAY[MID + 1 TO RIGHT]

    MERGE LEFT_SUBARRAY AND RIGHT_SUBARRAY BACK INTO ARRAY[LEFT
        TO RIGHT]
END FUNCTION
```

Implementation

```python
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = left  # Initial index of merged subarray

    # Merge the temp arrays back
    # into arr[left..right]
```

```
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[],
    # if there are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1


def merge_sort(arr: list):
    _merge_sort(arr, 0, len(arr) - 1)

def _merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        _merge_sort(arr, left, mid)
        _merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)
```

Listing 7: Merge sort function

Results

It is from from the diagram 5 that the algorithm has $O(n \log n)$ compexity. Its compexity doesn't depend on the type of the array, so it's worst, best, and average

complexity altogether.

Its space complexity is $O(n)$, since it requires storing of the divided arrays.

```
Merge Sort Execution Times (seconds):

Array Size      Random          Sorted          Partially Sorted
-------------------------------------------------------------
1000            0.002917        0.002677        0.003391
10000           0.039135        0.033806        0.041604
20000           0.082450        0.079612        0.077128
50000           0.245989        0.268167        0.316710
200000          1.111926        0.874895        0.885635
```

Listing 8: Merge sort raw data

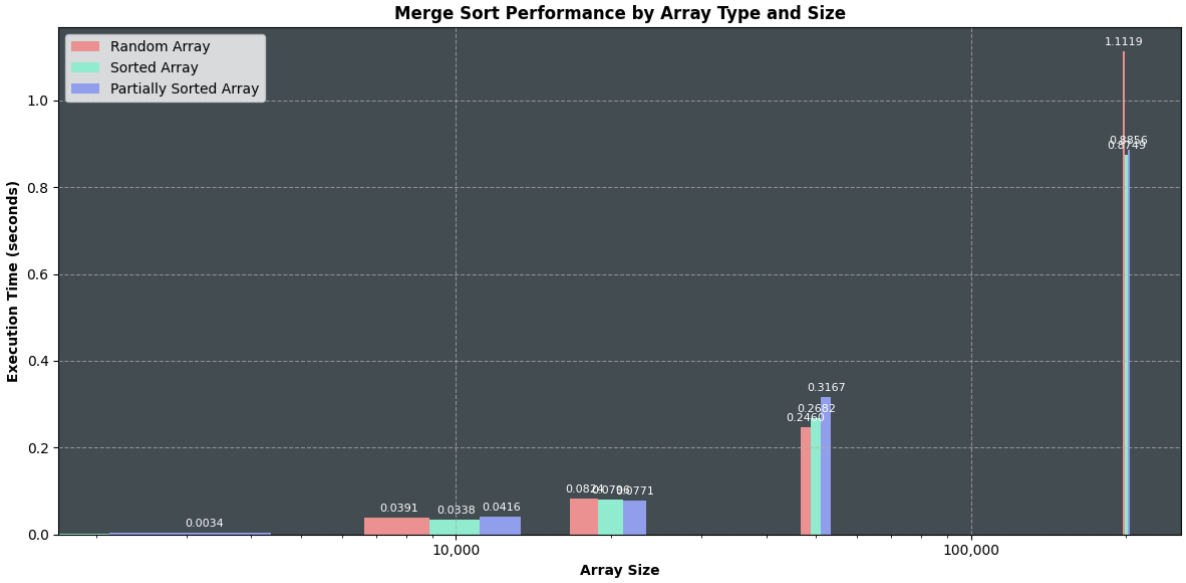


Figure 5: Merge Sort time execution

3 Bonus: visualization

The visualization service was made as generic as possible: it takes an array and the sorting function which must accept only one argument: the array.

To track each update of the array modification (such as swapping), the array was wrapped into a new class which stops the thread on each array element setting (swapping counts for two).

The full code is avaliable on GitHub repository[1] (lab2/code/visualizr.py). The image below is a frame of the visualization process of Heap Sort.

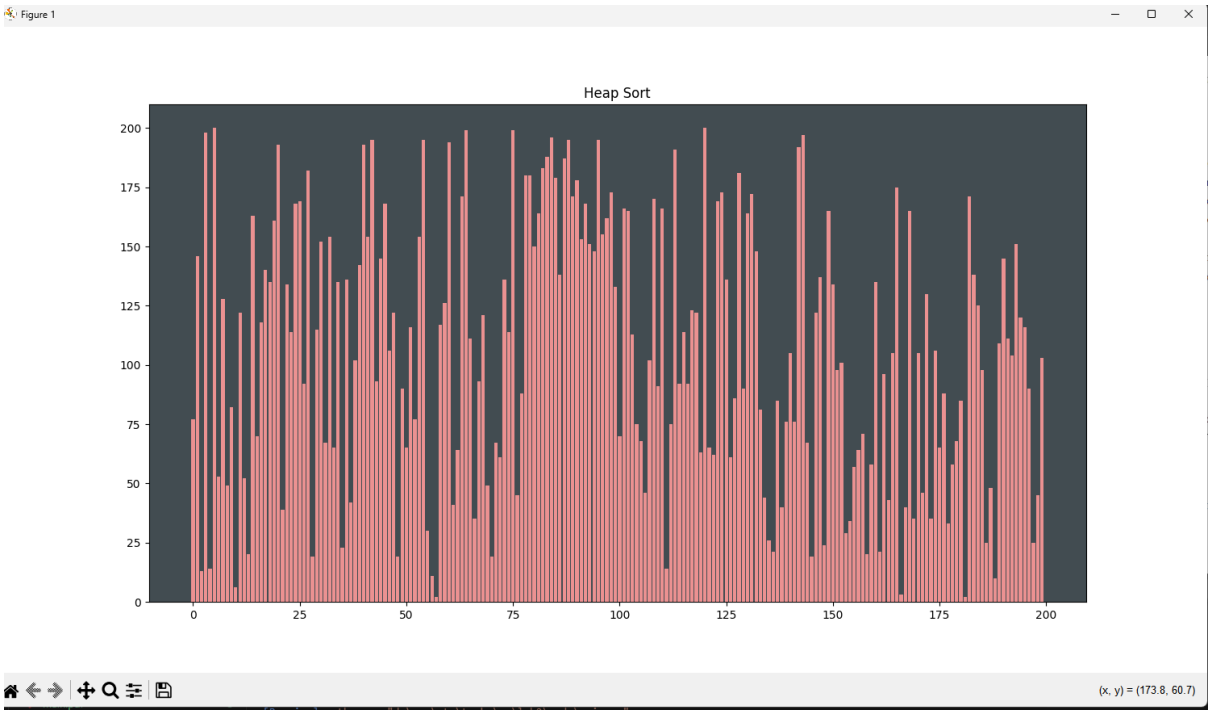


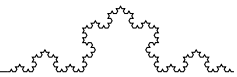
Figure 6: Heap Sort visualization frame

4 Conclusions

During this laboratory work, the analysis of four sorting algorithms were performed.

Each algorithm was tested on different dataasets, such as random, sorted and partially sorted to determine their complexity in dependence of the data (to find out average, worst and best case complexity), as well as on different datasets to track the algorithm complexity growth with growth of the size of the array.

The conclusitons upon the algorithm are made in the table 1.



Name	Best	Average	Worst	When to Use
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Only for small or nearly sorted datasets
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	General-purpose sorting, but avoid worst-case by choosing pivot wisely
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Best for linked lists and stable sorting needs
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Useful when constant time retrieval of max/min is needed

Table 1: Sorting Algorithm Complexity and Use Cases

References

[1] https://github.com/TimurCravtov/AA-labs