Ministry of Education, Research and Culture
Technical University of Moldova
Software Engineering and Automation Departments

REPORT

Laboratory work No. 2

Course: Analysis and Design of Algorithms

Theme: Study and empirical analysis of sorting algorithms.

Author: Corneliu Catlabuga
FAF-213

Checked by: univ. assist.
Cristofor Fiștic

Chisinau 2023

Objective:

Study and empirical analysis of sorting algorithms. Analysis of the quick sort, merge sort, heap sort and shell sort sorting algorithms on similar random arrays at different array sizes.

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.

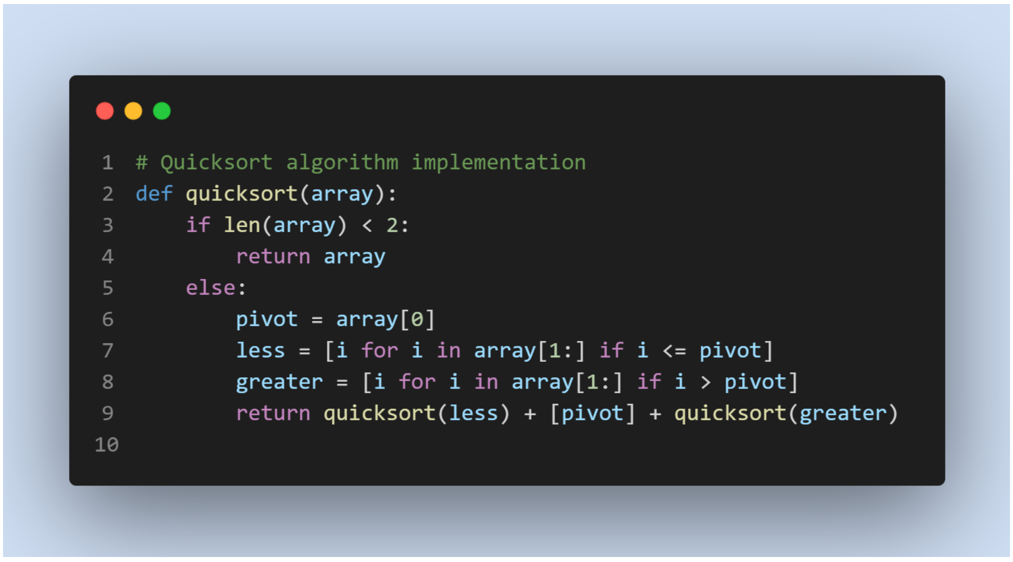
Theoretical considerations:

The following algorithms have been implemented in Python programming language. The used external libraries are: matplotlib (used for graph plotting), random (used for array generation) and time (used for execution time measurement).

Implementation, practical results:

Quicksort algorithm

Quicksort is a divide-and-conquer algorithm that sorts an array by selecting a "pivot" element and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted using the same process until the base case of a single element or empty array is reached.



```python
# Quicksort algorithm implementation
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Figure 1: Quicksort python function

Best-case complexity: O(n log(n))

Worst-case complexity: O(n^2)

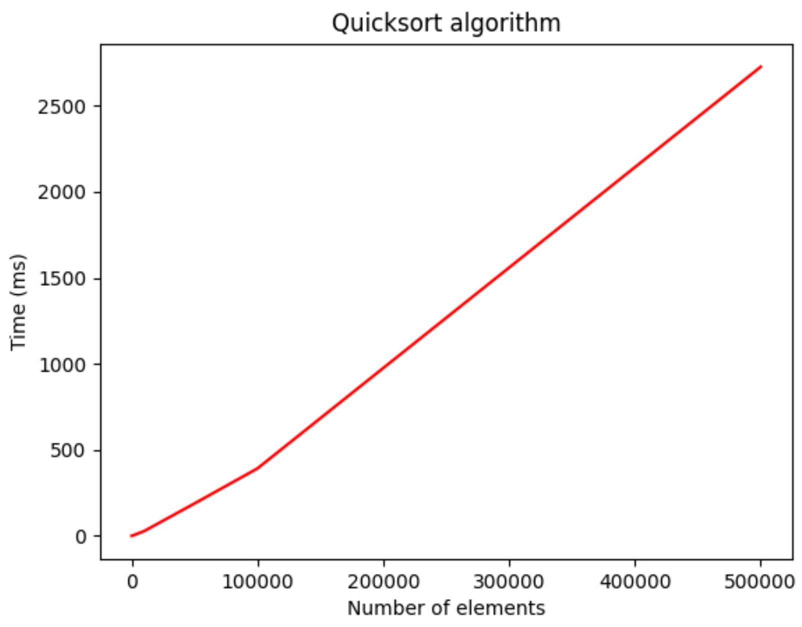
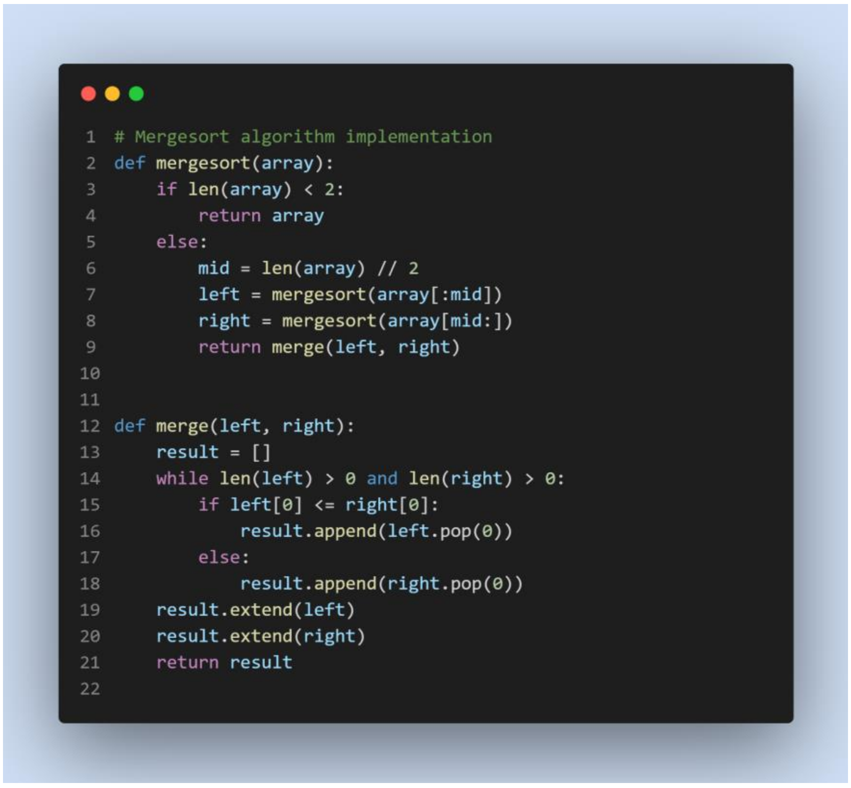


Figure 2: Quicksort execution time graph

Mergesort algorithm

Merge sort is a popular divide-and-conquer sorting algorithm that works by recursively dividing an unsorted array into two halves, sorting each half, and then merging the two sorted halves back together to form a single sorted array. The process is repeated until the entire array is sorted. During the merge phase, the algorithm compares the first element of each sub-array and places them in sorted order, continuing until all elements have been merged. Merge sort has a time complexity of O(n log n) and is often used as a benchmark for other sorting algorithms.



```python
# Mergesort algorithm implementation
def mergesort(array):
    if len(array) < 2:
        return array
    else:
        mid = len(array) // 2
        left = mergesort(array[:mid])
        right = mergesort(array[mid:])
        return merge(left, right)


def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left)
    result.extend(right)
    return result

```

Figure 3: Mergesort python function

Complexity: O(n log(n))

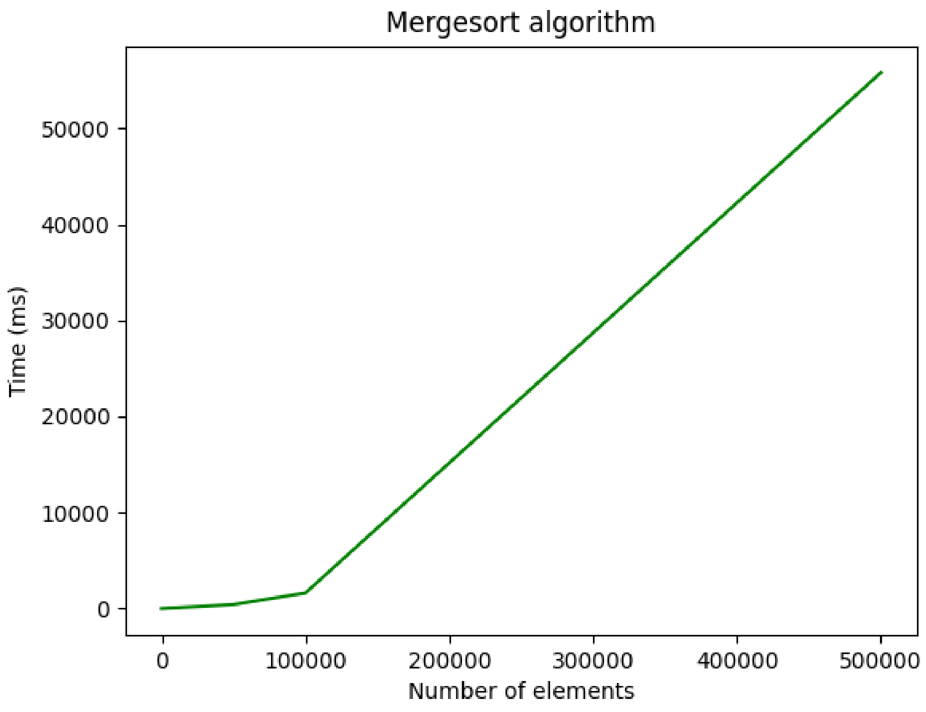
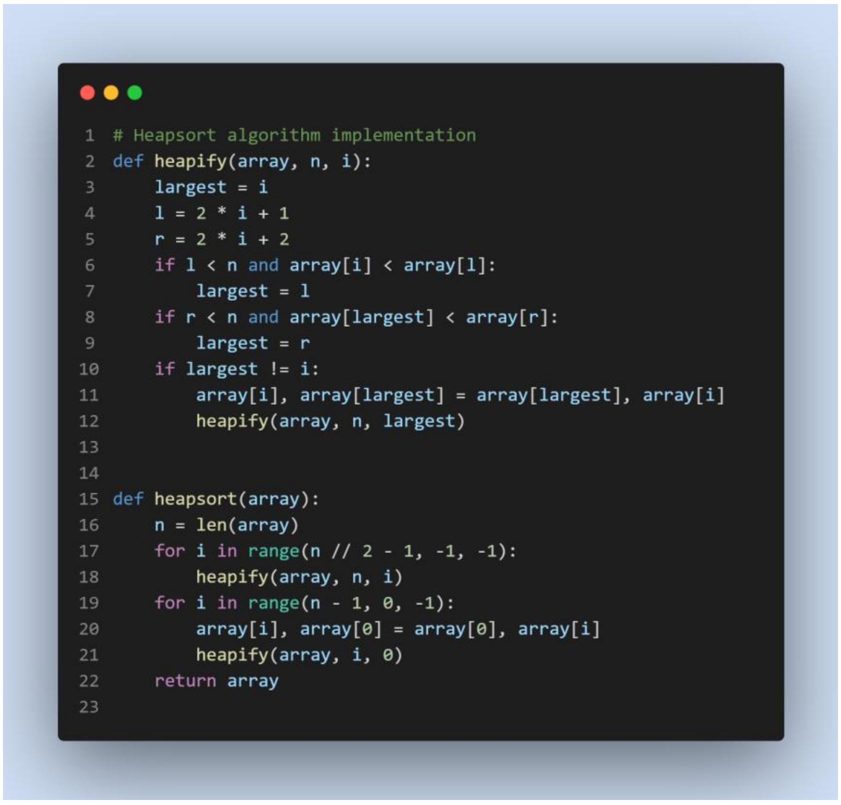


Figure 4: Mergesort execution time graph

Heapsort algorithm

Heap sort is a comparison-based sorting algorithm that works by building a binary heap data structure from the elements in the input array, and then repeatedly extracting the largest (or smallest) element from the heap and placing it at the end of the array. This process is repeated until the entire array is sorted.



```python
# Heapsort algorithm implementation
def heapify(array, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and array[i] < array[l]:
        largest = l
    if r < n and array[largest] < array[r]:
        largest = r
    if largest != i:
        array[i], array[largest] = array[largest], array[i]
        heapify(array, n, largest)


def heapsort(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)
    return array

```

Figure 5: Heapsort python function

Complexity: O(n log n)

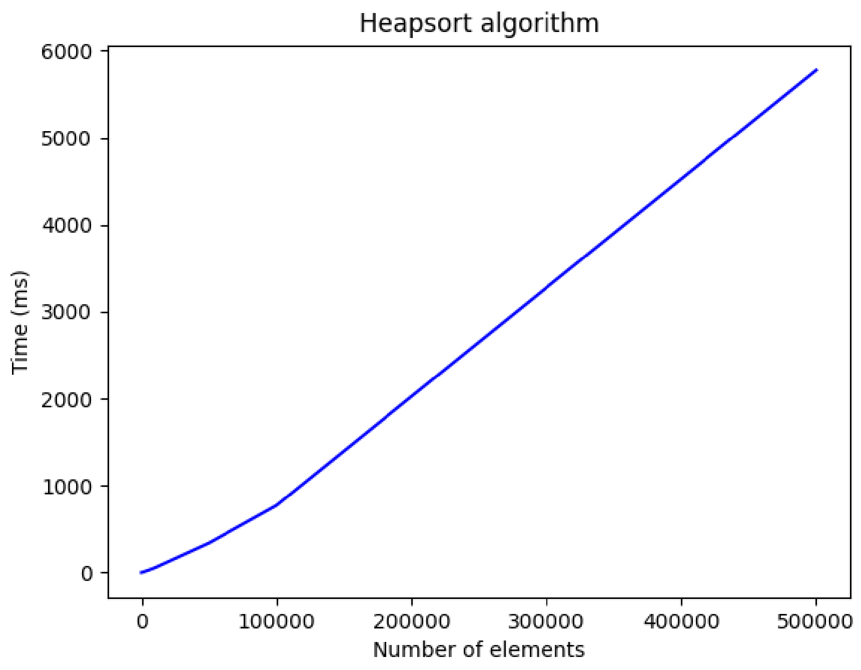
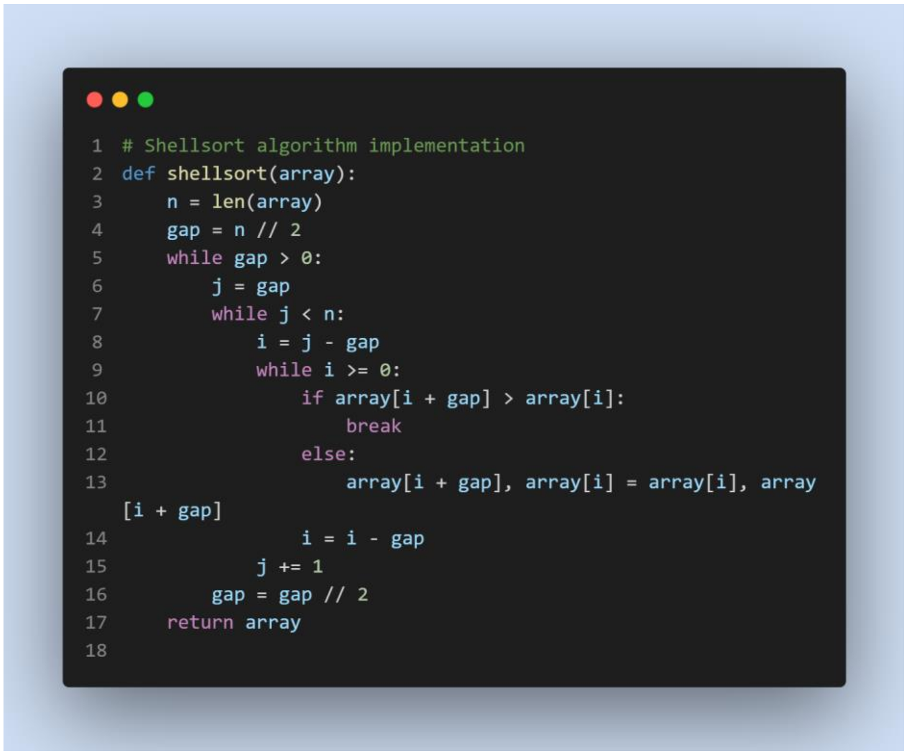


Figure 6: Heapsort execution time graph

Shellsort algorithm

Shell sort is a sorting algorithm that works by breaking down a large list into smaller sublists, sorting these sublists using insertion sort, and then combining them back together. The algorithm uses a sequence of gap values to determine the size of the sublists and gradually reduces the gap until it is equal to one, resulting in a fully sorted list.



```python
# Shellsort algorithm implementation
def shellsort(array):
    n = len(array)
    gap = n // 2
    while gap > 0:
        j = gap
        while j < n:
            i = j - gap
            while i >= 0:
                if array[i + gap] > array[i]:
                    break
                else:
                    array[i + gap], array[i] = array[i], array
[i + gap]
                i = i - gap
            j += 1
        gap = gap // 2
    return array

```

Figure 7: Shellsort python function

Best-case complexity: O(n log(n))

Worst-case complexity: O(n^2)

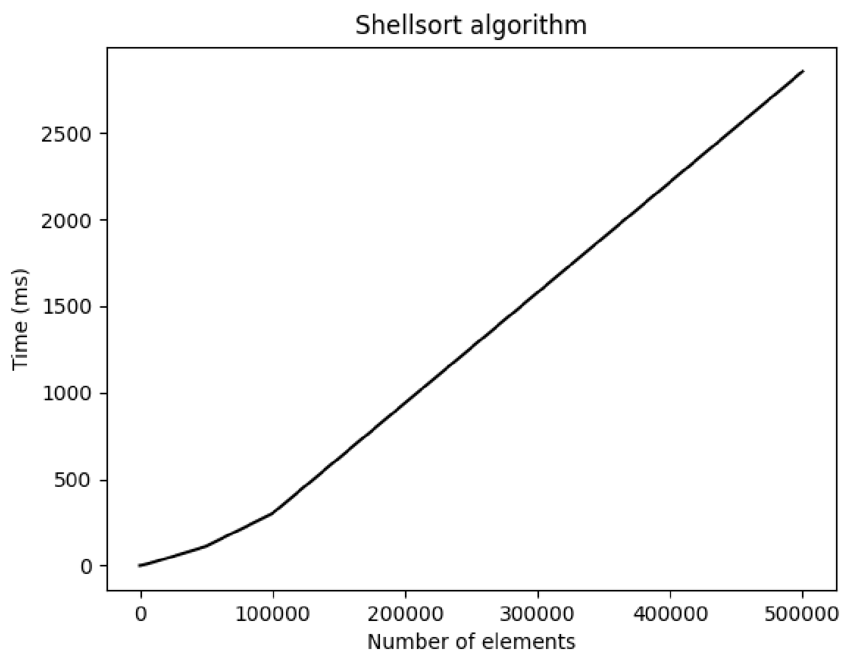


Figure 8: Shellsort execution time graph

Conclusions:

While all the algorithms are viable for arrays with under 5*10^4 elements, the mergesort algorithm, due to its recursive nature the execution time increases at an exponential rate. The best algorithms for sorting with lengths between 10^4 and 10^6 are the quicksort, heapsort and shellsort algorithms. For arrays with lengths bigger than 10^6 the quicksort and shellsort algorithms are the most time efficient.

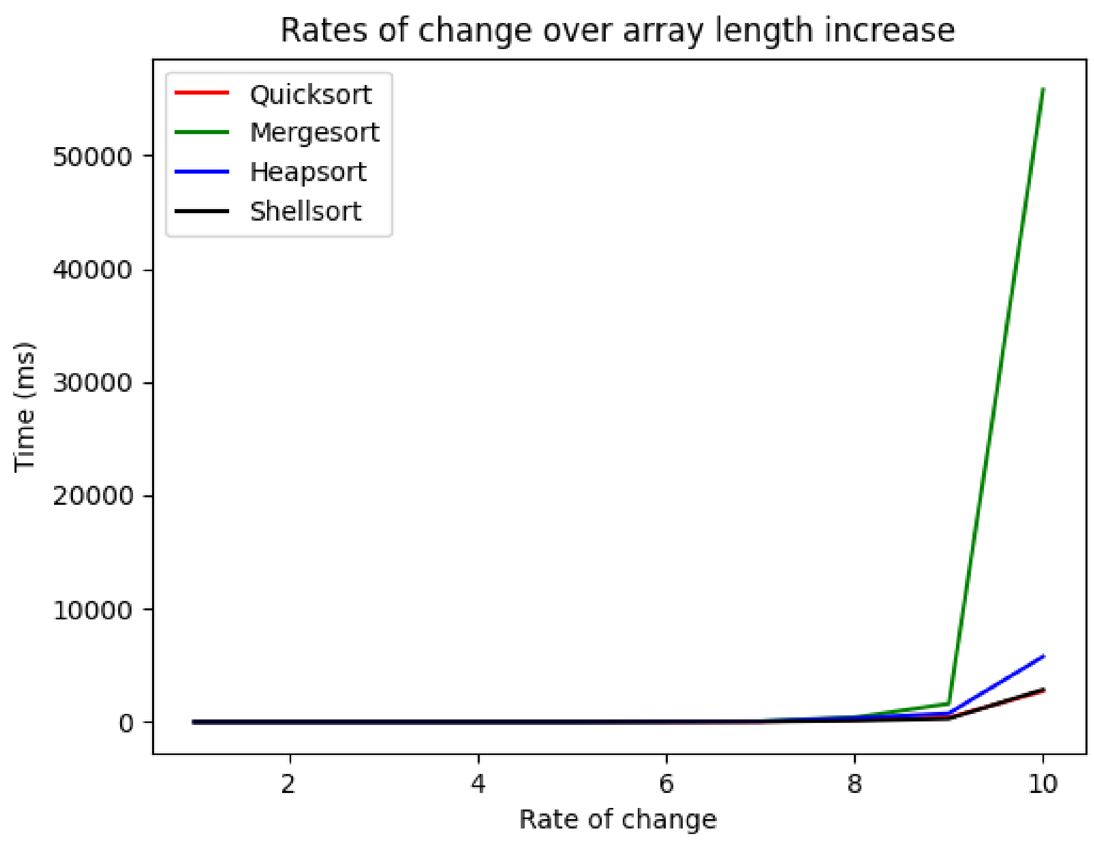


Figure 9: Rates of execution time

References:

1. https://github.com/muffindud/AA_Lab_2
2. https://www.geeksforgeeks.org/shellsort/
3. https://www.geeksforgeeks.org/sorting-algorithms/