Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 2
Discipline: Algorithms' Analysis
Topic: Study and empirical analysis of sorting algorithms

Elaborated: Țărnă Cristina,
student group FAF-211

Verified: Fiștic Cristofor,
university assistant

Chișinău – 2023

Algorithm analysis

Objective:

Study and analyze different sorting algorithms

Tasks:

1. Implement 4 sorting algorithms
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

Introduction:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Comparison metric:

The comparison metric for this laboratory work will be considered the time of execution of eachalgorithm (T(n))

Input format:

As input, each algorithm will receive a random array of 100000 numbers

IMPLEMENTATION

All four algorithms will be implemented in python an analyzed empirically based on the time required for their completion. The particular efficiency in rapport with input will vary depending on memory of the device used.

Quick Sort

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved.

Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log_n)$ comparisons to sort *n* items. In the worst case, it makes $O(n^2)$ comparisons.

Implementation:

```python
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# function to perform quicksort
def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)
```

Figure 1. Quick sort implementation

Results:

	array length	execution time (milliseconds)
0	10000	158.717632
1	20000	131.485462
2	40000	338.089228
3	80000	1161.364555
4	160000	1941.892147
5	320000	4552.301645
6	640000	14566.983938
7	1000000	30666.592121

Figure 2. Quick sort result table

The plot:

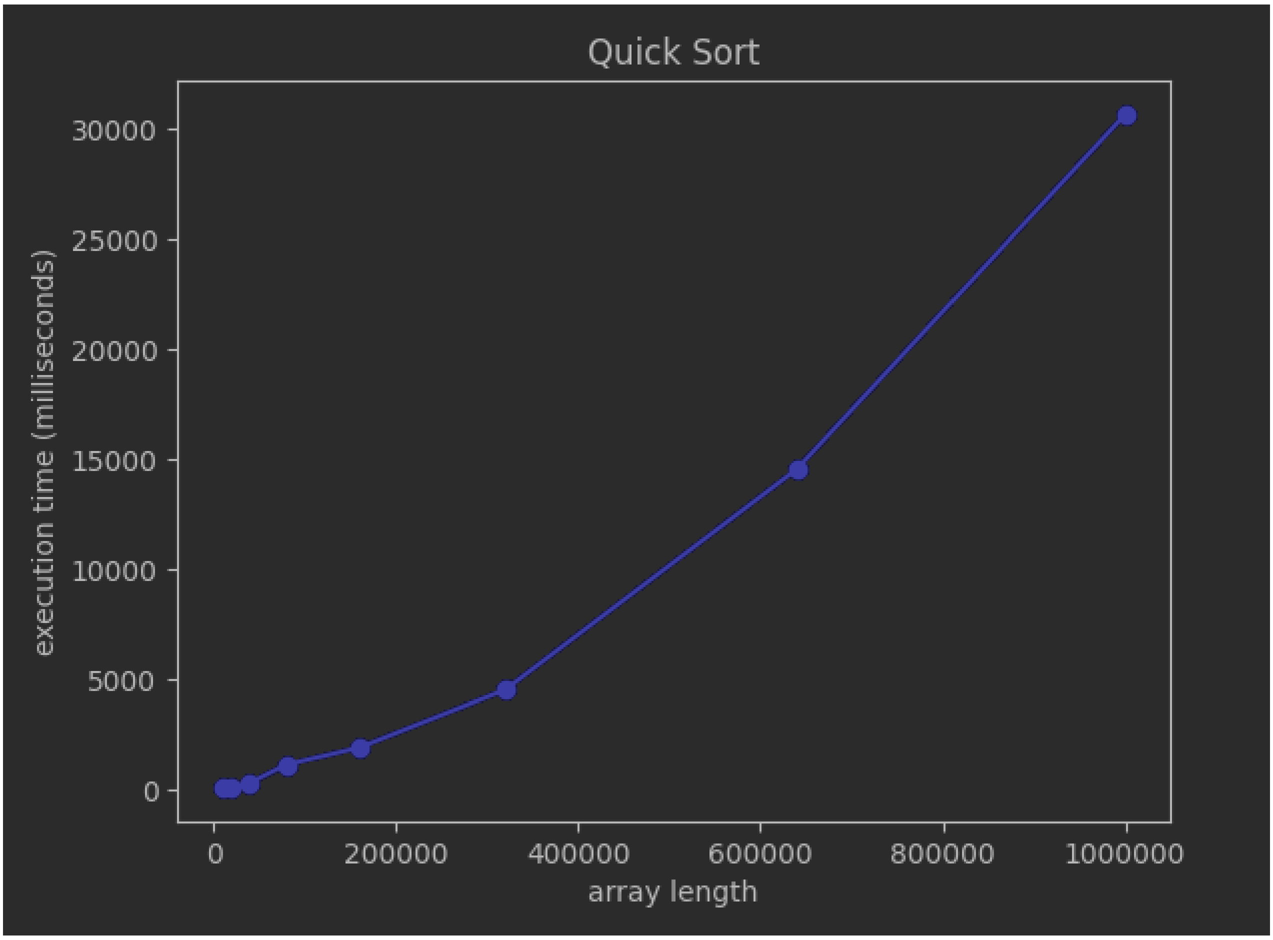

Figure 3. Quick sort plot result

Space complexity: O(1)

Best case: O (n*logn)

Worst case: O (n**2)

Stable: No

Merge Sort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Implementation:

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0    # Initial index of first subarray
    j = 0    # Initial index of second subarray
    k = l    # Initial index of merged subarray

    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
```

Figure 4. Merge sort implementation 1

```
def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
```

Figure 5. Merge sort implementation 2

Results:

	array length	execution time (milliseconds)
0	10000	155.855417
1	20000	170.241833
2	40000	239.865303
3	80000	739.741325
4	160000	1385.490179
5	320000	2778.556585
6	640000	4769.016981
7	1000000	7761.406898

Figure 6. Merge sort result table

The plot:

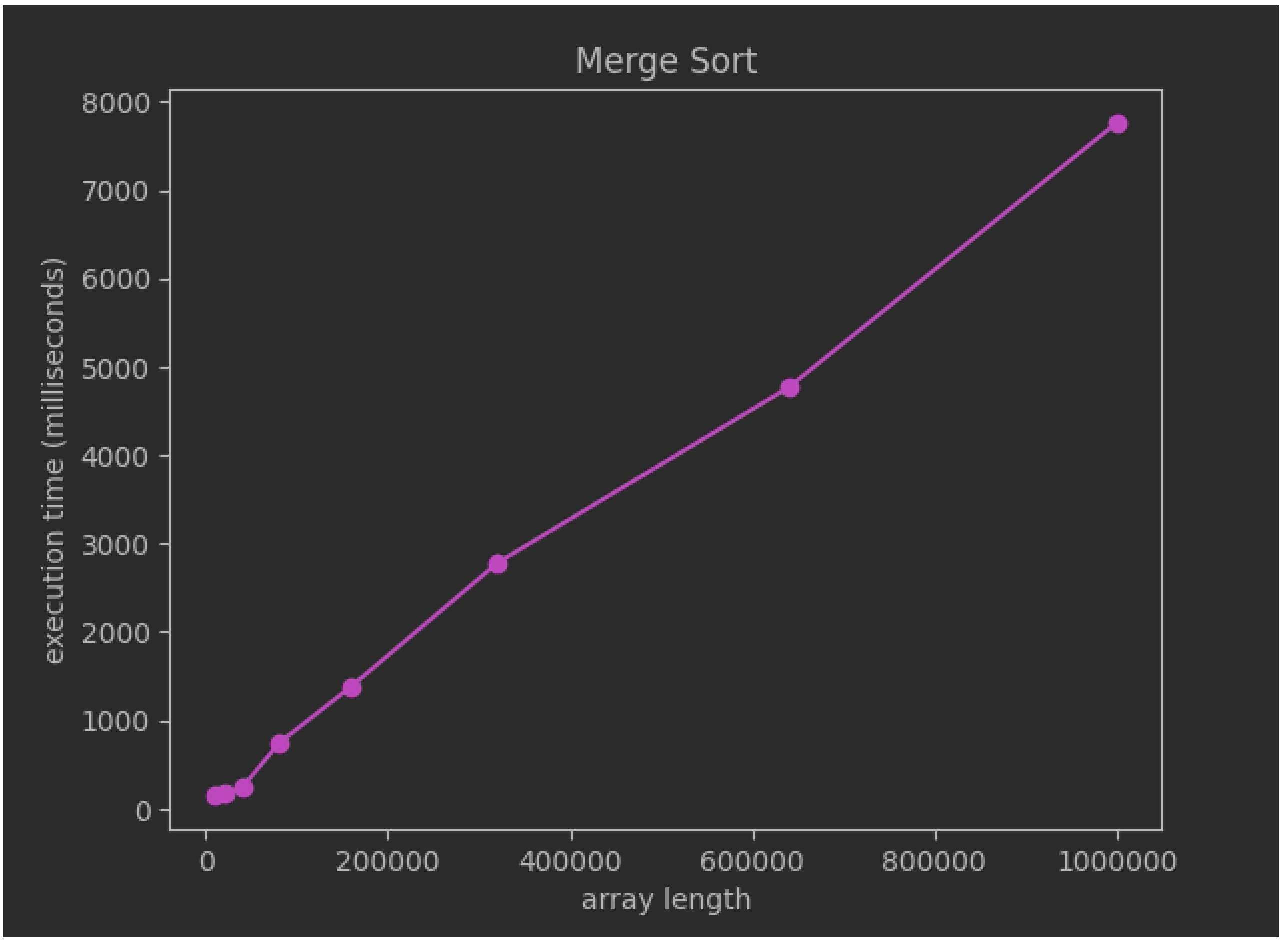

Figure 7. Merge sort plot result

Space complexity: O(n)

Best case: O (n*logn)

Worst case: O (n*logn)

Stable: Yes

Heap Sort

Heapsort is a comparison-based sorting algorithm. It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.

Implementation:

```python
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

# See if left child of root exists and is
# greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

# See if right child of root exists and is
# greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

# Change root, if needed
    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i]) # swap

# Heapify the root.
        heapify(arr, n, largest)


# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

# Build a maxheap.
# Since last parent will be at ((n//2)-1) we can start at that location.
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

# One by one extract elements
    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i]) # swap
        heapify(arr, i, 0)
```

Figure 8. Heap sort implementation

Results:

	array length	execution time (milliseconds)
0	10000	140.676498
1	20000	246.214867
2	40000	461.436272
3	80000	1058.922291
4	160000	2124.501228
5	320000	4748.028994
6	640000	11152.838230
7	1000000	16665.899038

Figure 9. Heap sort table result

The plot:

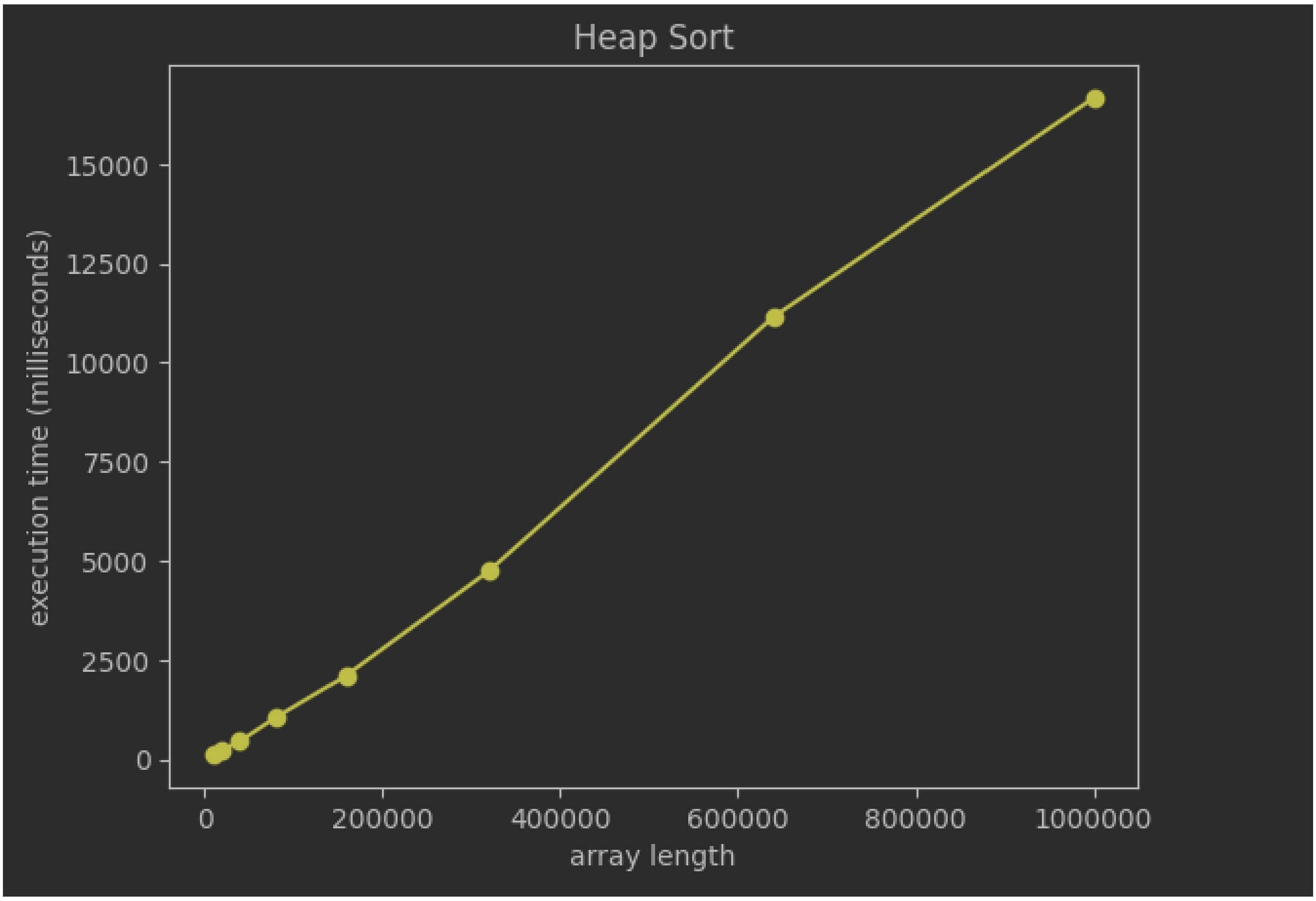

Figure 10. Heap sort plot result

Space complexity: O(n)

Best case: O (n)

Worst case: O (n*logn)

Stable: Yes

Counting Sort

Counting sort is an algorithm for sorting a collection of objects according to keys that are small positive integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that possess distinct key values, and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

Implementation:

```python
def countingSort(arr):
    size = len(arr)
    output = [0] * size

    # count array initialization
    count = [0] * size

    # storing the count of each element
    for m in range(0, size):
        count[arr[m]] += 1

    # storing the cumulative count
    for m in range(1, 10):
        count[m] += count[m - 1]

    # place the elements in output array after finding the index of each element of original
     array in count array
    m = size - 1
    while m >= 0:
        output[count[arr[m]] - 1] = arr[m]
        count[arr[m]] -= 1
        m -= 1

    for m in range(0, size):
        arr[m] = output[m]
```

Figure 11. Counting sort implementation

Results:

	array length	execution time (milliseconds)
0	10000	12.528419
1	20000	24.437904
2	40000	45.604944
3	80000	75.562954
4	160000	167.082787
5	320000	445.294380
6	640000	463.932276
7	1000000	893.347740

Figure 12. Counting sort table result

The plot:

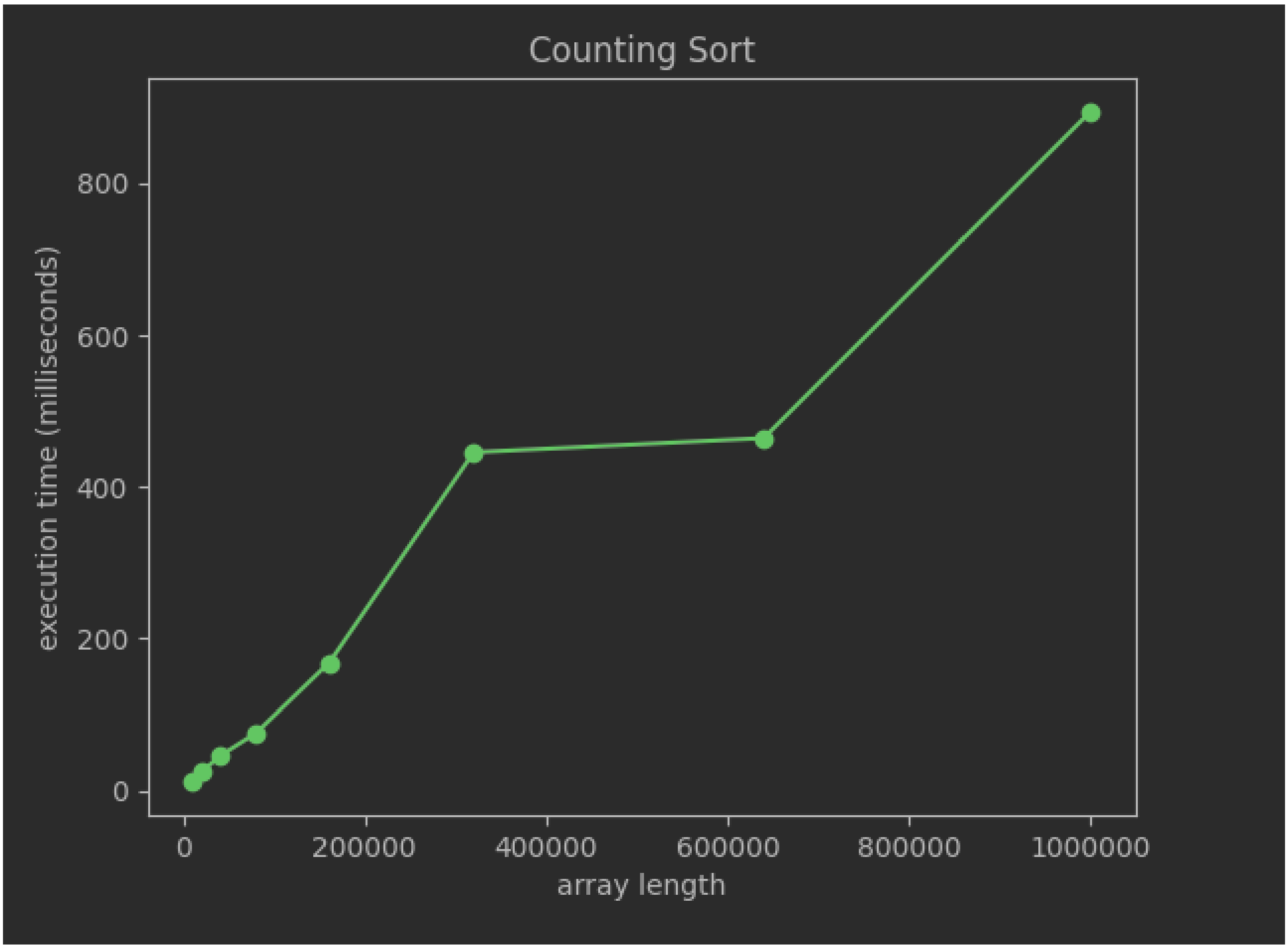

Figure 13. Counting sort plot result

Space complexity: O(k)

Best case: O (n+k)

Worst case: O (n+k)

Stable: Yes

n is the nr of elements in the array, k is the range of input

Conclusion:

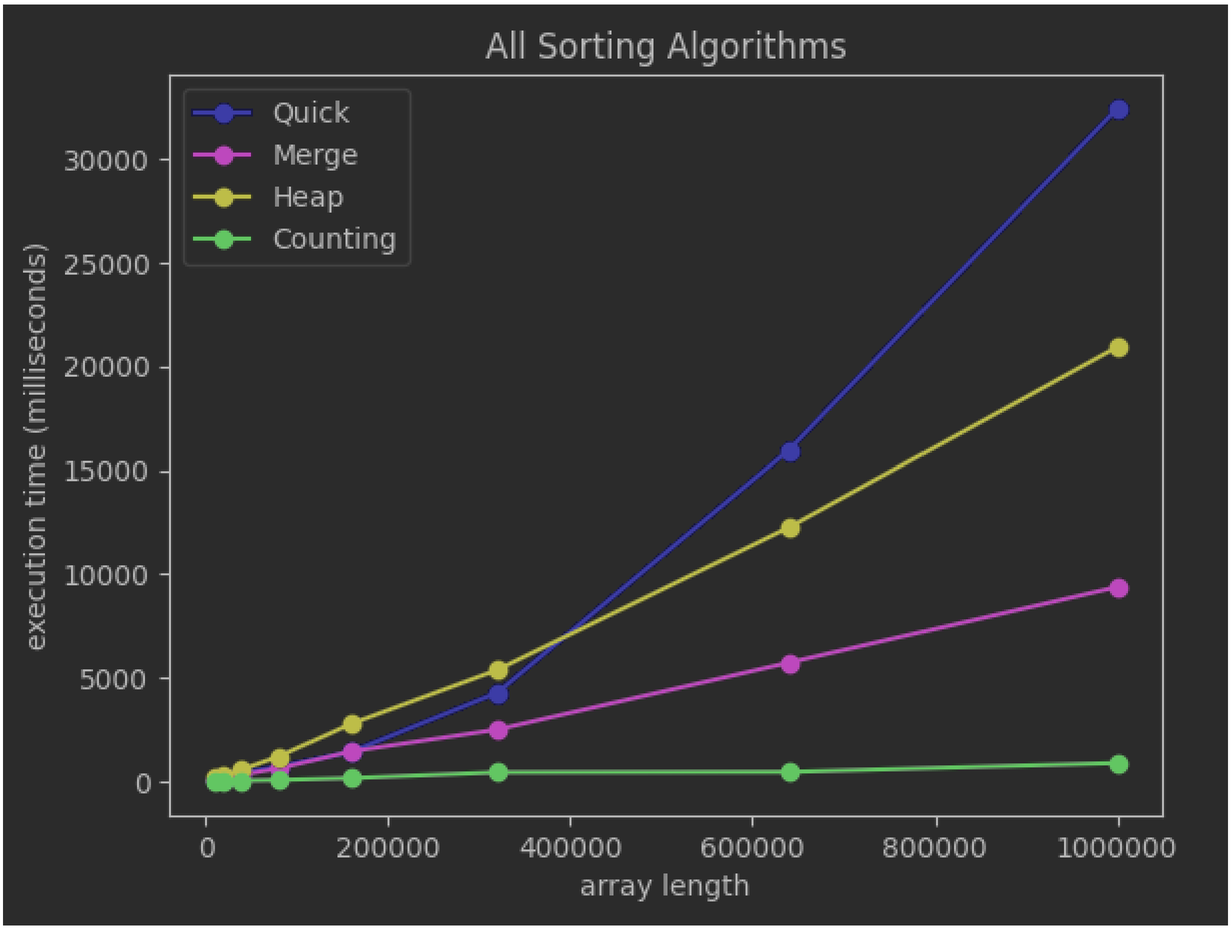

Figure 14. All sorting algorithms

The slowest algorithm, ironically, turned out to be Quick sort. The fastest is Counting Sort.

Link to GitHub: https://github.com/CristinaT21/APA_LABS