Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

REPORT

Laboratory work no.2
Study and empirical analysis of sorting algorithms.

Elaborated:
st. gr. FAF-213 Gutu Dinu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

Algorithm Analysis

Objective

1. Analysis and study of quick sort, merge sort, heap sort, counting sort
2. Empirical analysis of the aforementioned sorting algorithms

Tasks

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 Notes

Empirical analysis is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction

Sorting is a fundamental problem in computer science, and there are many different algorithms that have been developed to solve it. The basic idea of a sorting algorithm is to rearrange a set of elements in a specific order, usually in ascending or descending order.

There are many different criteria that can be used to evaluate sorting algorithms, including time complexity, space complexity, stability, adaptivity, and simplicity. Time complexity is often the most important criterion, as it determines how long the algorithm will take to sort a given input.

Some of the most popular sorting algorithms include:

- Bubble sort: a simple algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It has a time complexity of O(n^2) in the worst case.
- Selection sort: an algorithm that repeatedly selects the smallest remaining element and swaps it with the next element in the sorted portion of the array. Selection sort has a

time complexity of O(n^2) in the worst case.

- Insertion sort: an algorithm that builds the final sorted array one element at a time, by inserting each new element into its correct position. Insertion sort has a time complexity of O(n^2) in the worst case, but can be more efficient than other O(n^2) algorithms in certain cases.
- Merge sort: a divide-and-conquer algorithm that recursively divides the input array into smaller arrays, sorts them, and then merges the sorted arrays together. Merge sort has a time complexity of O(n log n) in the worst case.
- Quick sort: another divide-and-conquer algorithm that picks a pivot element, partitions the array around the pivot, and then recursively sorts the two resulting sub-arrays. Quick sort has a time complexity of O(n log n) in the average case but can degenerate to O(n^2) in the worst case.
- Heap sort: an in-place sorting algorithm that uses a binary heap to maintain the partial order of the input array, and repeatedly extracts the minimum element to build the sorted output array. Heap sort has a time complexity of O(n log n) in the worst case.
- Counting sort: a non-comparison-based algorithm that counts the number of occurrences of each input value and uses these counts to generate the sorted output array. Counting sort has a time complexity of O(n + k), where k is the range of the input values.
- Bucket sort: another non-comparison-based algorithm that distributes the input elements into a set of buckets, sorts each bucket individually, and then concatenates the sorted buckets together. Bucket sort has a time complexity of O(n + k), where k is the number of buckets.

There are many other sorting algorithms as well, each with their own strengths and weaknesses. The choice of algorithm depends on the specific requirements of the application, including the size of the input, the range of input values, and the desired time and space complexity. In this laboratory work I will be analyzing only the following algorithms: quick sort, heap sort, merge sort and counting sort.

Comparison Metric

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

Input Format

As input, the algorithms will receive parts of a random list, evidently using the seed function. The parts will grow with each iteration, that is they will cover more numbers from the list until finally the entire list needs to be sorted. The grow step will be 1/20 of the list length which is 100000, thus the step will be 5000. So starting from 5000 the arrays will have to sort for example the Array[:5000], that is up to and including the 4999th element and so forth.

Implementation

Source Code: GitHub

Quick Sort

Like Merge Sort, **Quick Sort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quick sort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.

Algorithm Description:

Basic Pseudo-code:

```
/* low -> Starting index, high -> Ending index */

quickSort(arr[], low, high) {
    if (low < high) {
        /* pi is partitioning index, arr[pi] is now at right
place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}

/* This function takes last element as pivot, places the pivot
element at its correct position in sorted array, and places
all smaller (smaller than pivot) to left of pivot and all
greater elements to right of pivot */

partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];
```

```
    i = (low - 1) // Index of smaller element and indicates
the
    // right position of pivot found so far
    for (j = low; j <= high- 1; j++){
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

In the implemented case I used a simpler version, that is the pivot is chosen as the first element and instead of partition, through an if statement it chooses the elements to the right and left side of the pivot, after which it begins to recursively call the quick_sort function until the pivot is at the right spot and the left and right sides are both sorted.

Implementation

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = []
        right = []
        for i in range(1, len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
            else:
                right.append(arr[i])
        return quick_sort(left) + [pivot] + quick_sort(right)
```

Figure 1. Quick sort algortihm

Results

Quick Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.008	0.018	0.027	0.037	0.049	0.059	0.071	0.082	0.094	0.105	0.118	0.131	0.145	0.158	0.172	0.184	0.201	0.214	0.23	0.244

Figure 2. Quick sort results

In figure 2 we can clearly see that quick sort is very efficient, thus having an average time complexity of O(n*log(n)). Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than some other sorting algorithms like Merge Sort and Heap Sort, Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.

Graphs

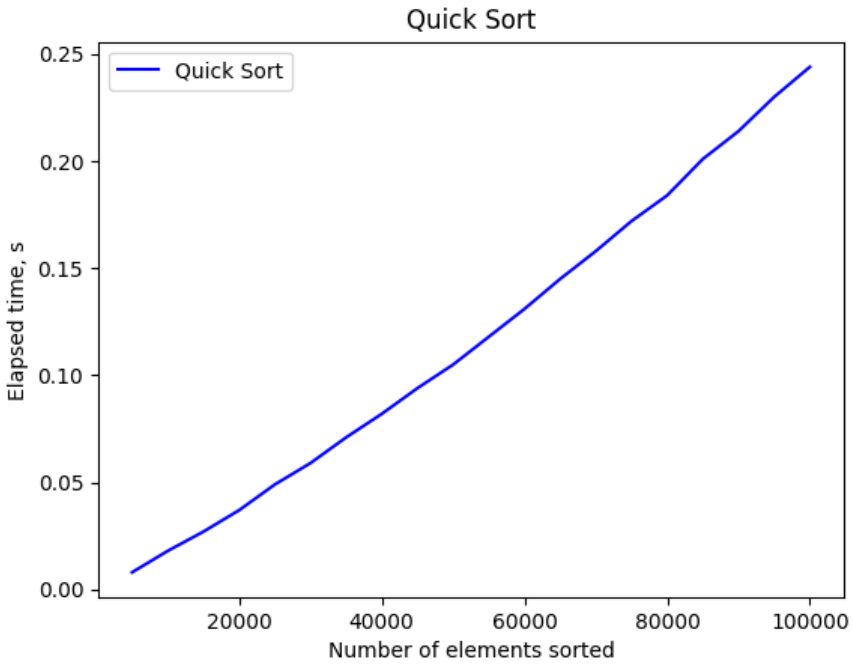

Figure 3. Quick sort graph

Figure 3 clearly shows that quick sort has an almost linear time complexity

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.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

One thing that you might wonder is what is the specialty of this algorithm. We already have several sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of **O(n log n)**, which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.
Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Algorithm Description:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
step 4: Stop
```

Implementation

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)


def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

Figure 4. Merge sort algortihm

Results

Merge Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.014	0.028	0.044	0.061	0.078	0.095	0.116	0.128	0.144	0.161	0.18	0.197	0.216	0.24	0.256	0.275	0.296	0.31	0.335	0.351

Figure 5. Merge sort results

From the graph it is suggestive that the merge sort algorithm is also very efficient, having an average time complexity of O(n*log(n)). Overall the time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves. The following graph could serve as another proof to its time complexity.

Graphs

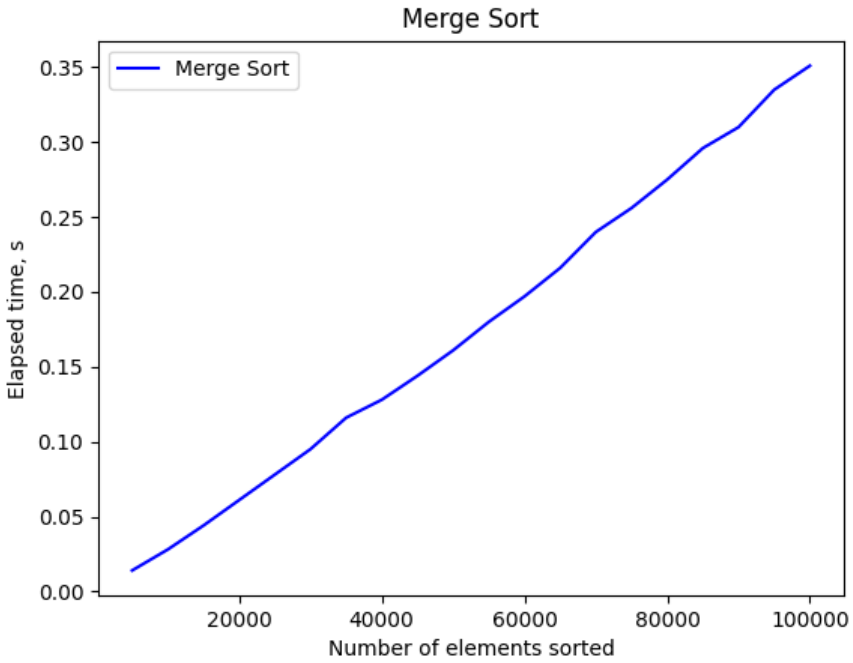

Figure 6. Merge sort graph

Heap Sort

Heap sort is a comparison-based sorting technique based on **Binary Heap** data structure. It is similar to the **selection sort** where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable
- Typically, 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of heapsort:

- **Efficiency – The** time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- **Memory Usage –** Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity – It** is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort:

- **Costly**: Heap sort is costly.
- **Unstable**: Heap sort is unstable. It might rearrange the relative order.
- **Efficient:** Heap Sort are not very efficient when working with highly complex data.

Algorithm Description:

```
First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

heapify(array)
 Root = array[0]
   Largest = largest( array[0] , array [2 * 0 + 1]/ array[2 * 0 + 2])
if(Root != Largest)
 Swap(Root, Largest)
```

Implementation

```python
def heap_sort(arr):
    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)

    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)
    return arr
```

Figure 7. Heap sort algortihm

Results

Heap Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.019	0.043	0.069	0.098	0.124	0.146	0.177	0.204	0.233	0.265	0.299	0.328	0.357	0.377	0.418	0.454	0.48	0.519	0.538	0.572

Figure 8. Heap sort results

Heap sort perfomed pretty efficinet but still worse when comparing to the previous algorithms, given that it also has a time complexity of O(n*log(n)). The benefit of heap sort is the reduced memory it takes. Figure 9 shows that heap sort has an almost linear time complexity.

Graphs

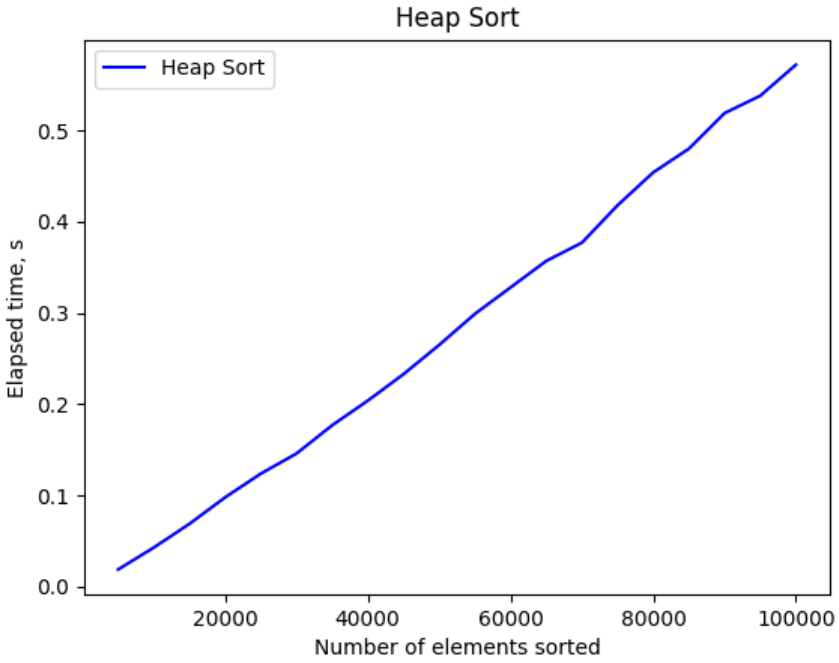

Figure 9. Heap sort graph

Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (a kind of hashing). Then do some arithmetic operations to calculate the position of each object in the output sequence.

Characteristics of counting sort:

- Counting sort makes assumptions about the data, for example, it assumes that values are going to be in the range of 0 to 10 or 10 – 99, etc., some other assumption counting sort makes is input data will be all real numbers.
- Like other algorithms this sorting algorithm is not a comparison-based algorithm, it hashes the value in a temporary count array and uses them for sorting.
- It uses a temporary array making it a non-In Place algorithm.

Algorithm Description:

```
function counting_sort(array, max_value):
    count_array = [0] * (max_value + 1)  # initialize count
array with zeros
    sorted_array = [0] * len(array)  # initialize sorted array
```

```
with zeros

    # count the number of occurrences of each element
    for value in array:
        count_array[value] += 1

    # compute prefix sums of the count array
    for i in range(1, max_value + 1):
        count_array[i] += count_array[i - 1]

    # use the count array to place each element in its correct
position
    for value in array:
        index = count_array[value] - 1
        sorted_array[index] = value
        count_array[value] -= 1

    return sorted_array
```

Implementation

```python
def counting_sort(arr):
    # Find the range of values in the array
    min_val = min(arr)
    max_val = max(arr)

    # Count the number of occurrences of each value in the array
    count = [0] * (max_val - min_val + 1)
    for val in arr:
        count[val - min_val] += 1

    # Compute the cumulative sum of the count array
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Create a sorted array
    sorted_arr = [0] * len(arr)
    for val in reversed(arr):
        sorted_arr[count[val - min_val] - 1] = val
        count[val - min_val] -= 1

    return sorted_arr
```

Figure 10. Counting sort algortihm

Results

Counting Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.002	0.003	0.005	0.006	0.007	0.009	0.01	0.011	0.012	0.013	0.015	0.016	0.018	0.019	0.021	0.022	0.023	0.025	0.026	0.028

Figure 11. Counting sort results

Figure 11 shows that counting sort performs extremely well even when compared to the previous sorting algorithms. It has a time complexity of O(N + K) where N is the number of elements in the input array and K is the range of input. Thus we can conclude that counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted.

Graphs

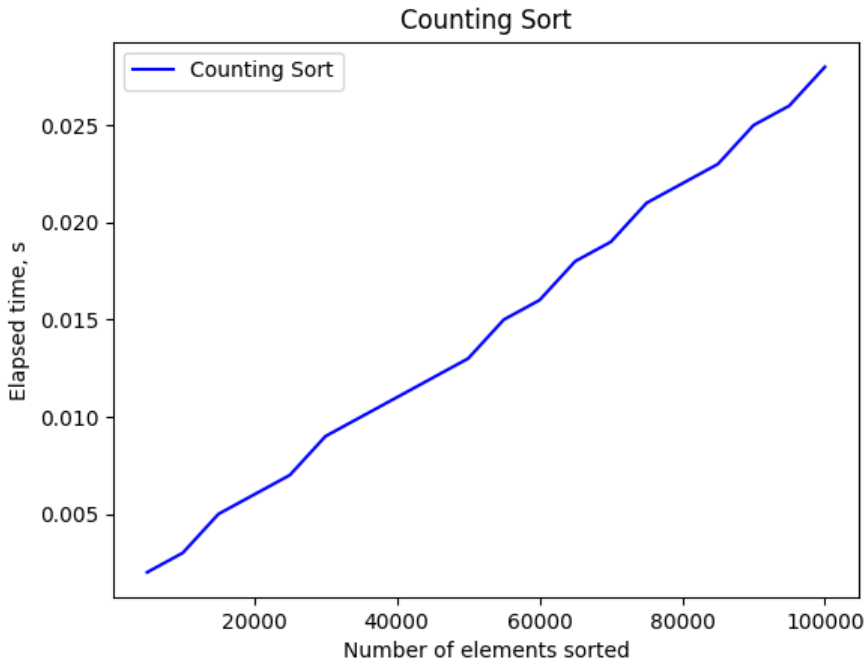

Figure 12. Counting sort graph

All algorithms

Results

Quick Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.008	0.018	0.027	0.037	0.049	0.059	0.071	0.082	0.094	0.105	0.118	0.131	0.145	0.158	0.172	0.184	0.201	0.214	0.23	0.244

Merge Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.014	0.028	0.044	0.061	0.078	0.095	0.116	0.128	0.144	0.161	0.18	0.197	0.216	0.24	0.256	0.275	0.296	0.31	0.335	0.351

Heap Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.019	0.043	0.069	0.098	0.124	0.146	0.177	0.204	0.233	0.265	0.299	0.328	0.357	0.377	0.418	0.454	0.48	0.519	0.538	0.572

Counting Sort

5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
0.002	0.003	0.005	0.006	0.007	0.009	0.01	0.011	0.012	0.013	0.015	0.016	0.018	0.019	0.021	0.022	0.023	0.025	0.026	0.028

Figure 13. All sorts results

Graphs

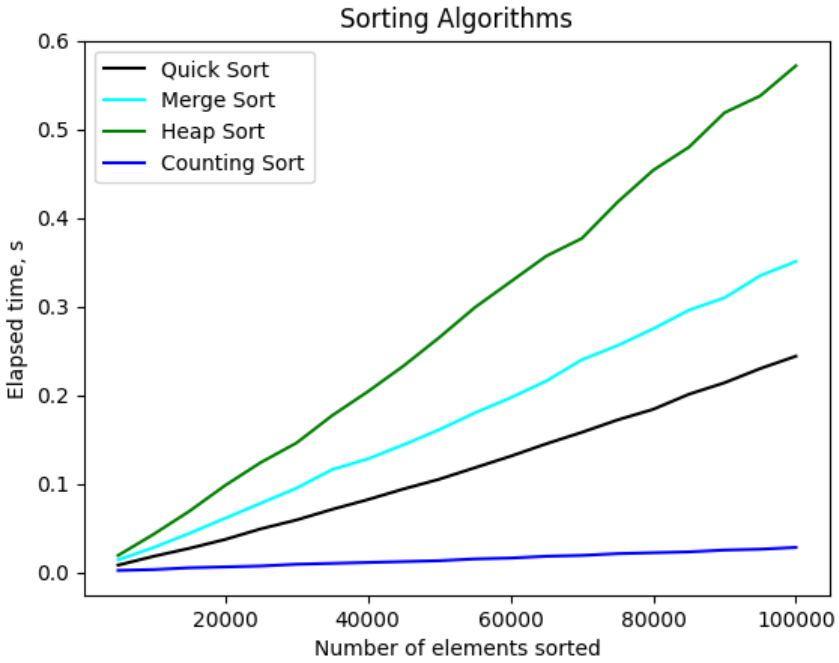

Figure 14. All sorts graph

Conclusion

Through empirical analysis, in this laboratory work were tested four sorting algorithms. Their effectiveness was determined by taking into account both their complexity and accuracy. The test cases were selected to highlight each algorithm's strengths, weaknesses, and potential for optimization while also accurately reflecting the time complexity of each approach.

Quick sort is a very popular and efficient comparison-based sorting algorithm that has an average time complexity of O(n*log n) and a worst-case time complexity of O(n^2). In addition to that, it was one of the easiest to implement and undersand. Furthermore, the algorithm works well on average and random inputs, but can be slow on inputs that are already partially sorted or contain many duplicate elements.

Merge sort is another popular and efficient comparison-based sorting algorithm that has a time complexity of O(n*log n) in the worst, best and average case. It is also stable, meaning that it preserves the relative order of equal elements. However, it requires additional memory to store the sorted sub-arrays during the merge step.

Heap sort is an in-place sorting algorithm that uses a binary heap data structure to maintain the partial order of the input array. In addition to that, it has a time complexity of O(n*log n) in the worst case, and is efficient on large inputs that can fit into memory. However, the algorithm is not stable, and can be slower than other algorithms on small inputs.

Counting sort is a non-comparison-based sorting algorithm. It has a time complexity of O(n + k), where k represents the range of the input values. It works most effectively when sorting inputs that have a small range of values, and its efficiency increases when the range is significantly smaller than the size of the input array. However, it requires extra memory to store the count array, and it may not be suitable for sorting inputs that have a large range of values.

In general, the choice of sorting algorithm depends on the specific requirements of the application, including the size and range of the input, the desired time and space complexity, and the need for stability or in-place sorting. Quick sort and Merge sort are often used for general-purpose sorting, while Heap sort and Counting sort are more specialized algorithms that work well on certain types of inputs.