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

Laboratory work 2:
Study and Empirical Analysis of Algorithms for Determining Sorting Algorithms

Elaborated:
st. gr. FAF-212 Moraru Dumitru

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

Table of Contents

ALGORITHM ANALYSIS

Objective

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

Tasks:

1. Implement 4 algorithms for sorting arrays;
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.

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 part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
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 are 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 checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Sorting algorithms are fundamental tools in computer science and are used to organize data in a specific order. Sorting algorithms take a set of unordered data and arrange them in a specified order, such as ascending or descending order. Sorting is an essential operation that is required in many applications, such as database management, searching, and data analysis.

There are many different types of sorting algorithms, each with its own advantages and disadvantages. Some algorithms are more efficient for small data sets, while others are better suited for large data sets. Sorting algorithms can be categorized into two primary groups: comparison-based sorting algorithms and non-comparison-based sorting algorithms.

Comparison-based sorting algorithms are the most common type of sorting algorithm and compare elements of the data set to each other to determine their relative order. Examples of comparison-based sorting algorithms include bubble sort, insertion sort, selection sort, merge sort, quicksort, and heapsort. Non-comparison-based sorting algorithms, on the other hand, do not compare elements of the data set and instead rely on the properties of the data to determine their order. Examples of non-comparison-based sorting algorithms include counting sort, radix sort, and bucket sort.

Understanding sorting algorithms is essential for computer science students and professionals as it is a fundamental operation that underpins many computer science applications. The choice of sorting algorithm can greatly impact the performance of a program, and choosing the right algorithm for the task at hand is critical for optimal performance.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing 4 algorithms empirically.

Comparison Metric:

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

Input Format:

The input format for most sorting algorithms is typically an array or a list of integers that are randomly generated or provided by the user. The size of the array or list can vary, ranging from small to very large data sets. The integers in the array or list are usually unsorted, and the objective of the sorting algorithm is to rearrange them in a specific order, such as ascending or descending order.

IMPLEMENTATION

All four algorithms will be implemented in their naive form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending o memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Quick Sort Method:

Quick sort is a popular sorting algorithm that works by dividing a large unsorted list into smaller sub-lists. It follows a divide-and-conquer approach, which means that it breaks the list into smaller parts and sorts them individually. The basic idea behind the algorithm is to select a pivot element from the list, partition the list around the pivot element, and recursively apply the same process to the sub-lists.

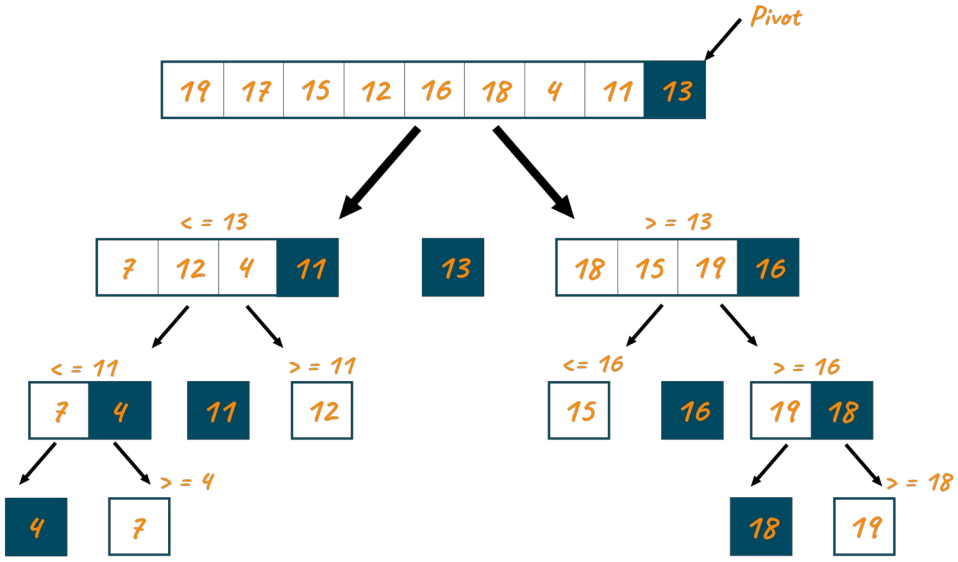

Figure 1: Quick sort explanation

Algorithm Description:

To implement the quick sort algorithm, we first choose a pivot element from the list. We then partition the list around the pivot element, such that all elements smaller than the pivot are placed to the left of the pivot, and all elements larger than the pivot are placed to the right of the pivot. This is done by comparing each element in the list with the pivot element and swapping elements as necessary to achieve the partition.

Once the partition is complete, we recursively apply the quick sort algorithm to the left and right sub-lists created by the partition until the entire list is sorted. The recursion terminates when we reach sub-lists of length one, which are already sorted.

Implementation:

```python
def quick_sort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

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

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

Figure 2: Quick sort algorithm

```python
# Function to find the partition position
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
    # e 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
```

Figure 3: Quick sort partition function

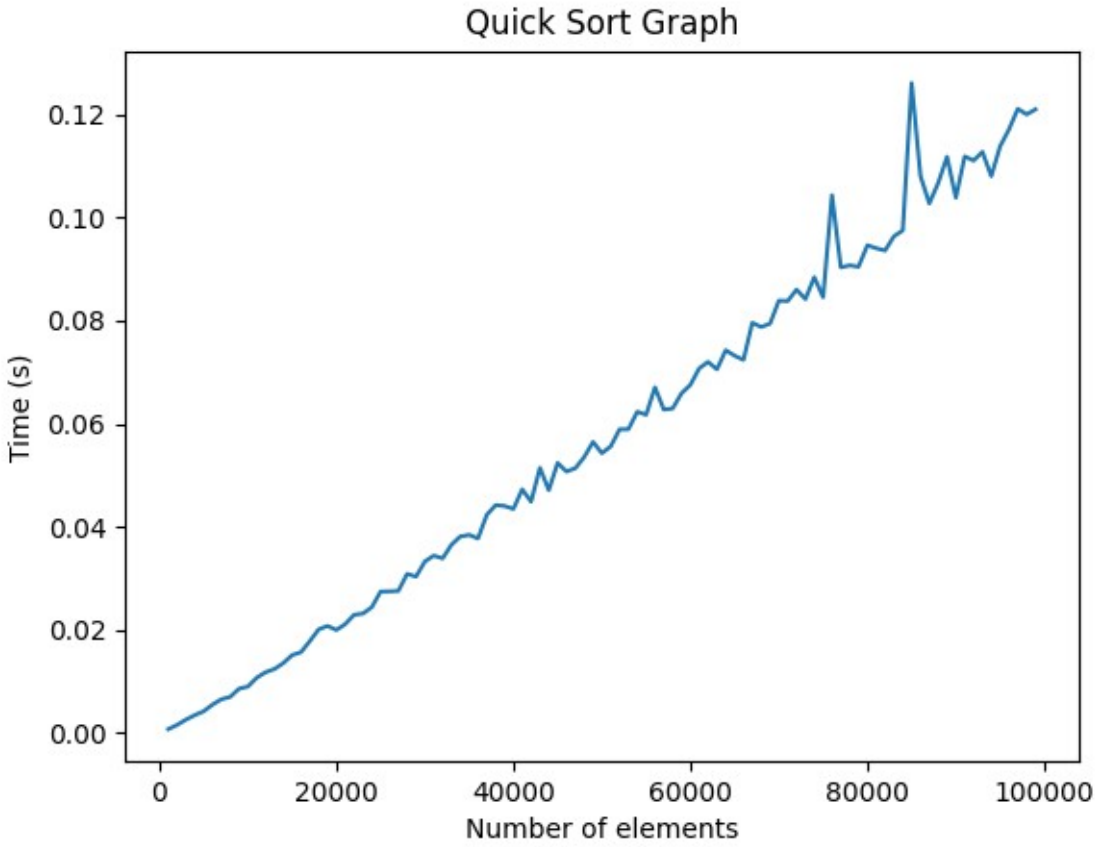

Figure 4: Graph of Quick Sort

The graph in Figure 4 that shows that Quick sort is an efficient sorting algorithm with an average time complexity of O(n *log* n) for a list of n elements. However, its worst-case time complexity can be $O(n^2)$ if the pivot element is poorly chosen and causes the partition to be highly unbalanced.

Merge Sort Method:

Merge sort works by dividing an unsorted list into smaller sub-lists, sorting those sub-lists, and then merging them back together to form a sorted list. It follows a divide-and-conquer approach, which means that it breaks the list into smaller parts and sorts them individually.

Algorithm Description:

To implement the merge sort algorithm, we first divide the unsorted list into two equal-sized sub-lists. We then recursively apply the merge sort algorithm to each sub-list until we reach sub-lists of length one, which are already sorted.

Once we have sorted the sub-lists, we merge them back together into a single sorted list. This is done by comparing the first elements of each sub-list and selecting the smaller one to be added to the final merged list. We continue this process, selecting the next smallest element from either sub-list and adding it to the merged list, until we have merged all elements from both sub-lists.

Implementation:

```
def mergeSort(arr):
    if len(arr) > 1:

        # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Figure 5: Merge sort algorithm

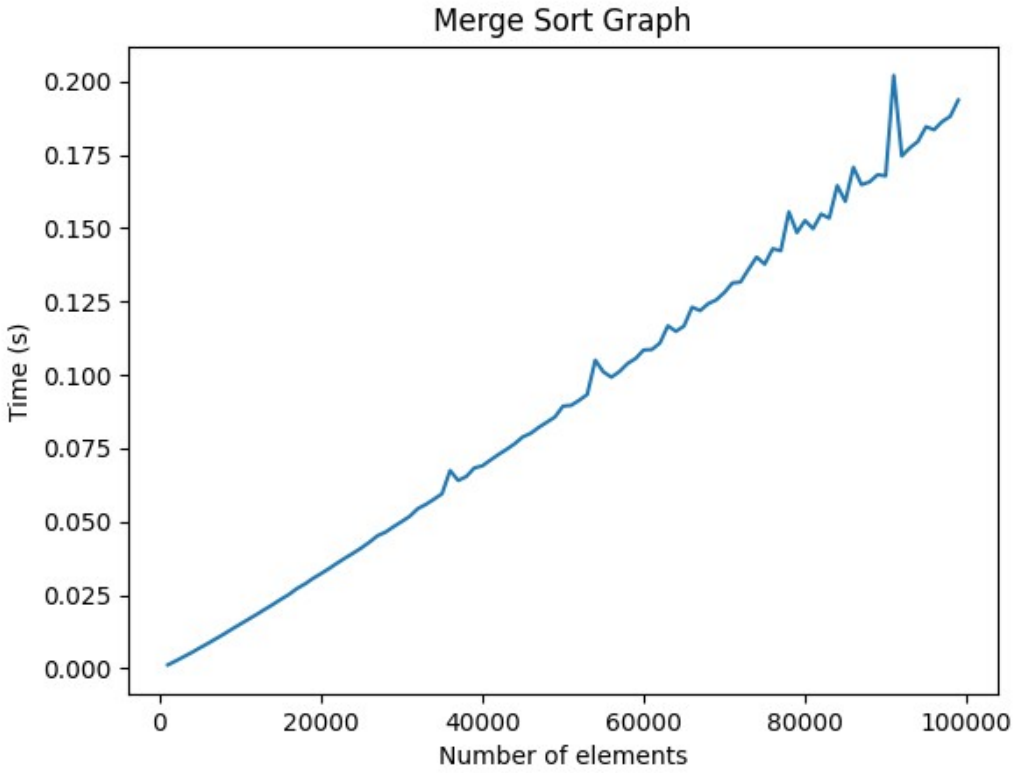

Figure 6: Merge Sort algorithm graph

Merge sort is an efficient sorting algorithm with a guaranteed worst-case time complexity of O(n *log* n) for a list of n elements. It is also a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the sorted list. However, merge sort has a space complexity of O(n), which means that it requires extra memory to store the sub-lists during the sorting process.

Heap Sort Method:

Heap sort is a popular sorting algorithm that works by creating a binary heap data structure from the unsorted list and then repeatedly extracting the maximum (or minimum) element from the heap and placing it at the end of the sorted list. It is an in-place sorting algorithm, which means that it does not require additional memory to store the sorted list.

Algorithm Description:

To implement the heap sort algorithm, we first build a binary heap from the unsorted list. A binary heap is a tree-like data structure in which each node has at most two children, and the value of each node is greater than (or less than) the values of its children. The heap can be either a max heap (in which the root node has the maximum value) or a min heap (in which the root node has the minimum value).

Once we have built the heap, we repeatedly extract the maximum (or minimum) element from the heap and place it at the end of the sorted list. We then restore the heap property by sifting down the root node to maintain the heap structure. We continue this process until all elements have been extracted from the heap and placed in the sorted list.

Implementation:

The implementation of the driving function in Python is as follows:

```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[largest] < 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.
    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 7: Heap sort implementation with help functions

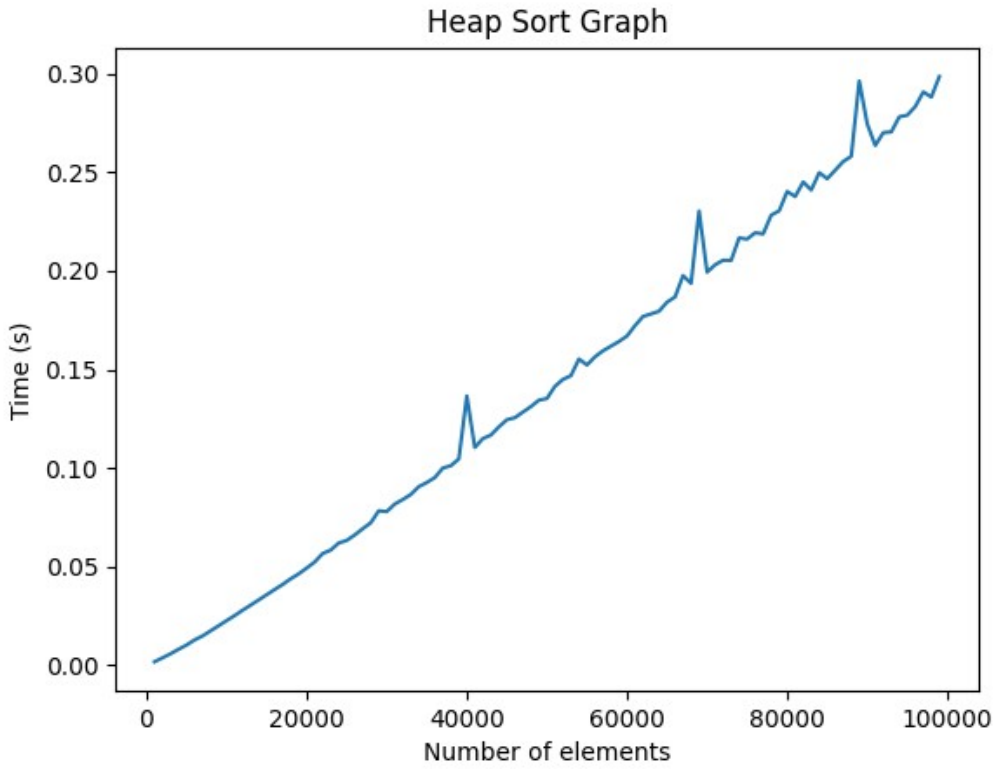

Figure 8: Heap Sort algorithm graph

Heap sort has a guaranteed worst-case time complexity of O(n log n) for a list of n elements, making it an efficient sorting algorithm. However, it has a relatively high constant factor and is not stable, meaning that it may change the relative order of equal elements in the sorted list.

Shell Sort Method:

Shell sort is a sorting algorithm that is an extension of insertion sort. It works by sorting elements that are far apart from each other, and progressively reducing the gap between the elements being compared until the gap becomes one, at which point it performs a standard insertion sort.

Algorithm Description:

To implement the shell sort algorithm, we first choose a gap sequence, which is a sequence of integers that determines the distance between the elements being compared. We then use this gap sequence to divide the unsorted list into smaller sub-lists and sort each sub-list using insertion sort.

After sorting each sub-list, we reduce the gap between elements being compared and repeat the process until the gap becomes one. At this point, the algorithm performs a standard insertion sort, which is efficient for small lists.

Implementation:

The implementation of the function in Python is as follows:

```
def shellSort(arr, n):
    # code here
    gap = n // 2


    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap # This will keep help in maintain gap value

            while i ≥ 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:

                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap # To check left side also
                            # If the element present is greater than current element
            j += 1
        gap = gap // 2
```

Figure 9: Shell Sort implementation

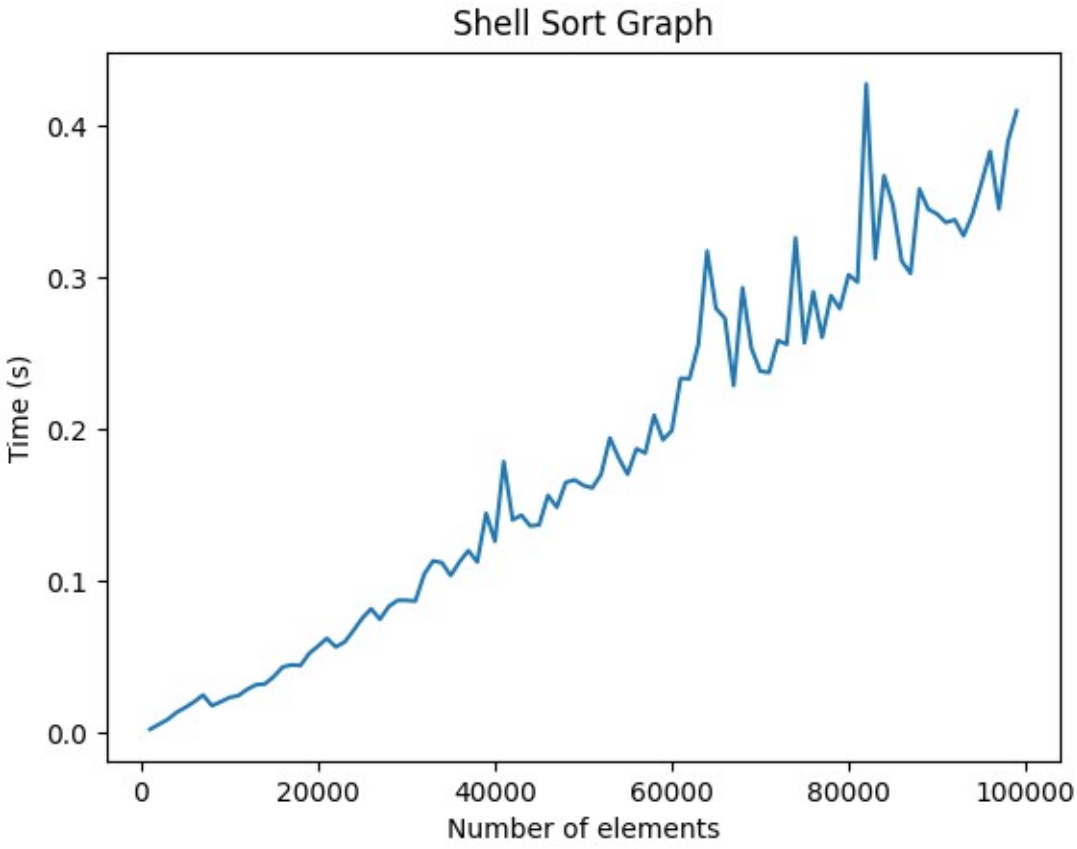

Figure 10: Shell Sort algorithm graph

Shell sort has a time complexity that depends on the gap sequence used, but it typically ranges between O(n log n) and O(n^2) for a list of n elements. It is an efficient sorting algorithm for medium-sized lists and can be faster than other O(n^2) algorithms like selection sort and insertion sort. However, it is not stable, meaning that it may change the relative order of equal elements in the sorted list.

CONCLUSION

In this paper, four categories of techniques have undergone empirical analysis to determine their efficiency in producing accurate results and the time complexity required for their execution. The study aims to identify the specific contexts in which each technique can be utilized and identify possible enhancements that can be made to improve their feasibility.

Quick sort is a efficient sorting algorithm that uses a divide-and-conquer approach to sort an unsorted list of elements. With an average time complexity of O(n log n) and a relatively low space complexity, it is widely used in computer science and is often the preferred sorting algorithm for large datasets.

Merge sort works by dividing an unsorted list into smaller sub-lists and merging them back together to form a sorted list. With a guaranteed worst-case time complexity of O(n log n), it is efficient and effective for sorting large datasets.

Heap sort is an in-place sorting algorithm that works by creating a binary heap data structure from an unsorted list and repeatedly extracting the maximum (or minimum) element from the heap and placing it at the end of a sorted list. With a guaranteed worst-case time complexity of O(n log n), it is an efficient sorting algorithm for large datasets. However, it is not stable and may change the relative order of equal elements in the sorted list.

Shell sort is an extension of insertion sort that sorts elements that are far apart from each other, gradually reducing the gap between the elements being compared until the gap becomes one, at which point it performs a standard insertion sort. With a time complexity that depends on the gap sequence used, it is an efficient sorting algorithm for medium-sized lists and can be faster than other $O(n^2)$ algorithms like selection sort and insertion sort. However, it is not stable and may change the relative order of equal elements in the sorted list.