**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**

A black and white logo

Description automatically generated

**A Project Report**

**on**

**“Sorting”**

**Report No.** 04

**[Code No.:** COMP202**]**

**SUBMITTED BY: SUBMITTED TO:**

Sanjib Dahal (18) Dr. Rajani Chulyadyo

Arjit Chand (17) Department of Computer

Group: CE Science and Engineering

Level: II/I

**Submission Date:** 2024/06/14

**Table of Contents**

[Introduction 3](#_Toc169240230)

[Task 3](#_Toc169240231)

[Code 3](#_Toc169240232)

[Output 4](#_Toc169240233)

[I. Insertion Sort 4](#_Toc169240234)

[Time Complexity 5](#_Toc169240235)

[II. Quicksort 6](#_Toc169240236)

[Time Complexity 6](#_Toc169240237)

[Conclusion 8](#_Toc169240238)

**Introduction**

Lab exercise no. 4 covers implementing two major sorting algorithms – quick sort and insertion sort in code and studying their individual time complexities by plotting a graph of varying input sizes against the execution time for each algorithm.

Sorting refers to arranging the elements of a given list into some defined order, for instance, arranging a set of numbers in ascending order. The most common rearrangements involve numerical order and lexicographical order that can either be ascending or descending. Sorting algorithms in computing aim to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator plays a big role in determining the relative order of elements in sorting algorithms. Programming languages allow the use of the comparison operators - ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’. The order is based on the type of operator used in an algorithm, and it can either be ascending (when ‘<’ is used) or descending (when ‘>’ is used).

Some commonly used sorting algorithms are bubble sort, selection sort, insertion sort, merge sort, quicksort, heap sort, tree sort etc. This lab exercise involves implementing insertion sort and quicksort and plotting time vs input size graphs for each algorithm.

**Task**

The following sorting algorithms are to be implemented in code and their respective input-size vs execution time graphs are to be plotted:

1. Insertion sort
2. Quick sort

# **Code**

Access the source code on GitHub via:

<https://github.com/sanjibdahal/CE2022_LAB4_17_18>

**Output**

## **Insertion Sort**

Insertion sort is a basic sorting algorithm that arranges a list of elements by performing comparisons one element at a time. It is an “in-place” sorting algorithm, and thus doesn’t require any extra memory space beyond what is already allocated for the original array. To sort an array of size ‘n’ in ascending order, the algorithm iterates over the array starting with the second element of the array. It performs comparisons between an element and its predecessors and places the former into its correct position until the whole array is sorted.

A screen shot of a computer program

Description automatically generated

Figure 1. Insertion Sort Output

### **Time Complexity**

1. Best Case: O(n) occurs when the array is already sorted.
2. Average Case: O(n2)
3. Worse Case: O(n2) occurs when the input array is sorted in reverse order.

To observe this, a graph between various array inputs against the execution time for each input was plotted.

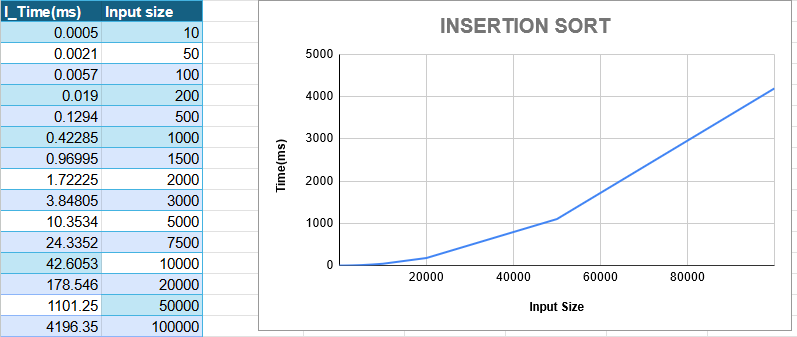


Figure 2. Insertion Sort Average Case: O(n2)

A graph with a line drawn on it

Description automatically generated

Figure 3. Insertion Sort's Best Case: O(n)

A graph with a line going up

Description automatically generated

Figure 4. Insertion Sort's Worst Case: O(n2)

In the average case of the algorithm, the graph represents a function that equals O(n2). It forms a curve that rises sharply as the input size (n) increases.

In comparison to the graph of the quicksort algorithm (n log n), this curve rises much faster. Therefore, for the same input values, computation takes longer than that for quicksort.

## **Quicksort**

Quicksort is a highly efficient sorting algorithm that employs a divide-and-conquer strategy. For an input array, the algorithm chooses a 'pivot' element and partitions the array into two sub-arrays, with elements less than the pivot on one side and greater on the other. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position. This is done until the array gets sorted.

### **Time Complexity**

1. Best Case: O (n log n) occurs when the pivot chosen at each step divides the array into almost equal halves.
2. Average Case: O (n log n)
3. Worst Case: O(n2) occurs when the pivot at each step consistently results in highly unbalanced partitions.

A screen shot of a computer program

Description automatically generated

Figure 4. Quicksort Output

A graph with a line

Description automatically generated

Figure 5. Quicksort Average Case

A graph on a white sheet

Description automatically generated

Figure 6. Quicksort Worst Case

One of the worst-case scenarios of the quicksort algorithm is when the input is either a sorted array or a nearly sorted array.

The average case graph represents a function that equals O (n log n). It forms a curve that grows slower than the quadratic n2 graph as the input size (n) increases.

In comparison to the graph of the insertion sort algorithm (n2), its performance is relatively smoother and more efficient when it comes to dealing with larger datasets. Thus, quicksort is generally preferred for large datasets while insertion sort is preferred for much smaller or already sorted datasets.

# **Conclusion**

Comparing the graphs of quicksort and Insertion Sort is helpful in analyzing the performance and efficiency of each algorithm. Quicksort has an average time complexity of O (n log n), and thus produces a graph resembling a logarithmic shape, showing a more gradual rise in execution time as the input size increases. This is a slower rate of increase compared to the quadratic curve of Insertion Sort. Insertion Sort, in contrast, has time complexity of O(n2) and exhibits a steep rise in its graph as the input size grows. This shows that quicksort, as its name implies, is much quicker and more efficient than insertion sort, especially for larger datasets as with the growing input size, its execution time grows much more slowly compared to Insertion Sort.

However, plotting the graphs for each algorithm was a tedious task. Input sizes had to be manually increased. Also, achieving precise measurements proved to be difficult as the time taken for execution depends on the input values and the system the code is running on. The randomizer in the code led to fluctuations in measurements even for the same input size. All in all, implementing and graphing showed their time complexities and helped in analyzing the algorithms.