Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report on "Sorting"

Report No. 04

[Code No.: COMP202**]**

SUBMITTED BY:

Sanjib Dahal (18)

Arjit Chand (17)

Group: CE

Level: II/I

SUBMITTED TO:

Dr. Rajani Chulyadyo

Department of Computer

Science and Engineering

Submission Date: 2024/06/14

Table of Contents

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:

(a) Insertion sort
(b) Quick sort

Code

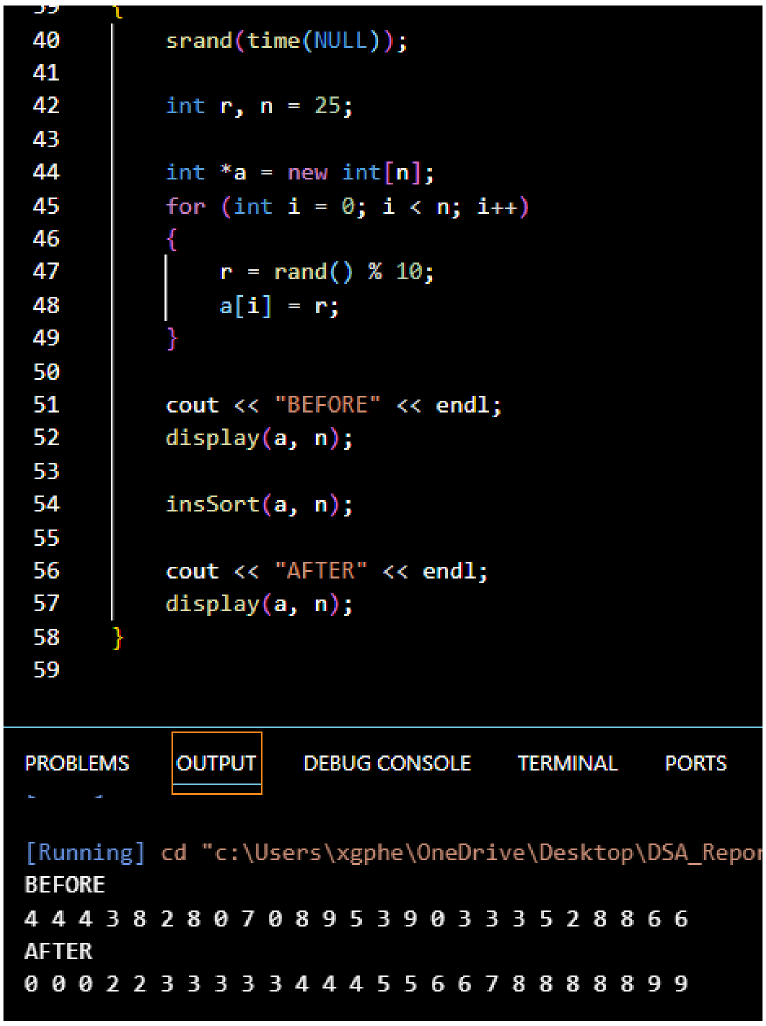
Access the source code on GitHub via:

https://github.com/sanjibdahal/CE2022_LAB4_17_18

Output

I. 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.



```
{
    srand(time(NULL));

    int r, n = 25;

    int *a = new int[n];
    for (int i = 0; i < n; i++)
    {
        r = rand() % 10;
        a[i] = r;
    }

    cout << "BEFORE" << endl;
    display(a, n);

    insSort(a, n);

    cout << "AFTER" << endl;
    display(a, n);
}
```

```
[Running] cd "c:\Users\xgphe\OneDrive\Desktop\DSA_Repor
BEFORE
4 4 4 3 8 2 8 0 7 0 8 9 5 3 9 0 3 3 3 5 2 8 8 6 6
AFTER
0 0 0 2 2 3 3 3 3 3 4 4 4 5 5 6 6 7 8 8 8 8 8 9 9
```

Figure 1. Insertion Sort Output

Time Complexity

1. Best Case: O(n) occurs when the array is already sorted.
2. Average Case: O(n^2)
3. Worse Case: O(n^2) 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.

I_Time(ms)	Input size
0.0005	10
0.0021	50
0.0057	100
0.019	200
0.1294	500
0.42285	1000
0.96995	1500
1.72225	2000
3.84805	3000
10.3534	5000
24.3352	7500
42.6053	10000
178.546	20000
1101.25	50000
4196.35	100000

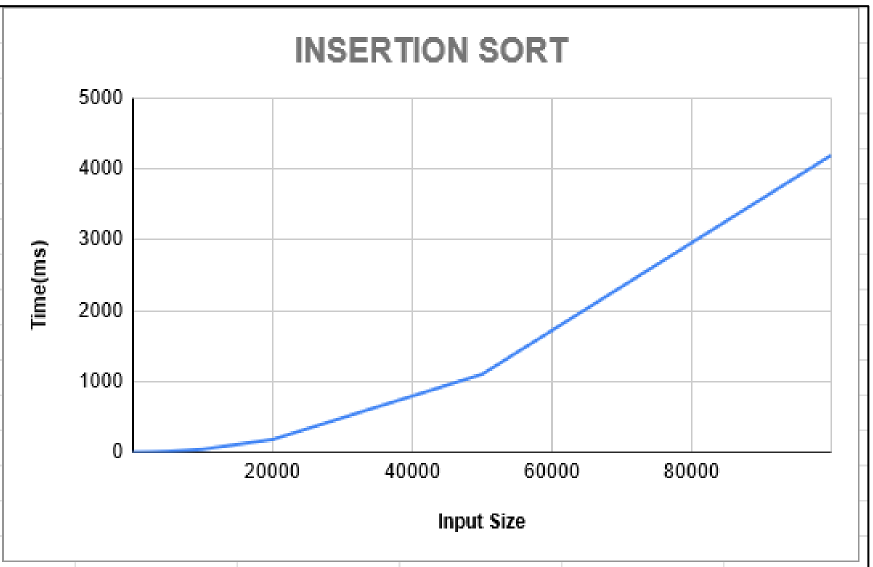


Figure 2. Insertion Sort Average Case: O(n^2)

I_Time(ms)	Input size
0.0005	10
0.0008	50
0.00095	100
0.0012	200
0.00205	500
0.00385	1000
0.00455	1500
0.00625	2000
0.0091	3000
0.01445	5000
0.02695	7500
0.03455	10000
0.069	20000
0.15795	50000
0.312	100000
0.407	150000
0.61	200000

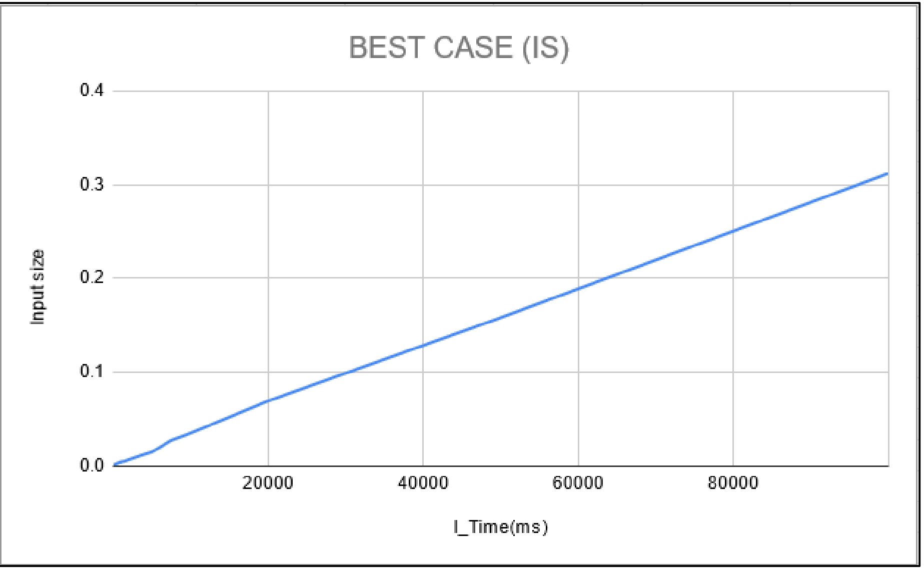
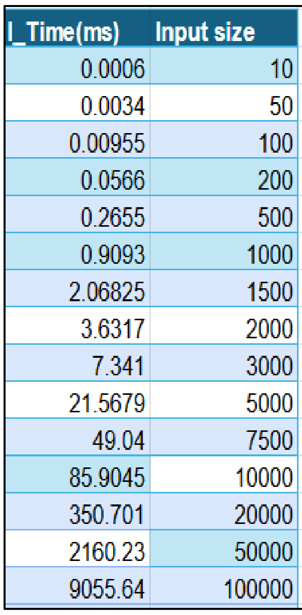


Figure 3. Insertion Sort's Best Case: O(n)



I_Time(ms)	Input size
0.0006	10
0.0034	50
0.00955	100
0.0566	200
0.2655	500
0.9093	1000
2.06825	1500
3.6317	2000
7.341	3000
21.5679	5000
49.04	7500
85.9045	10000
350.701	20000
2160.23	50000
9055.64	100000

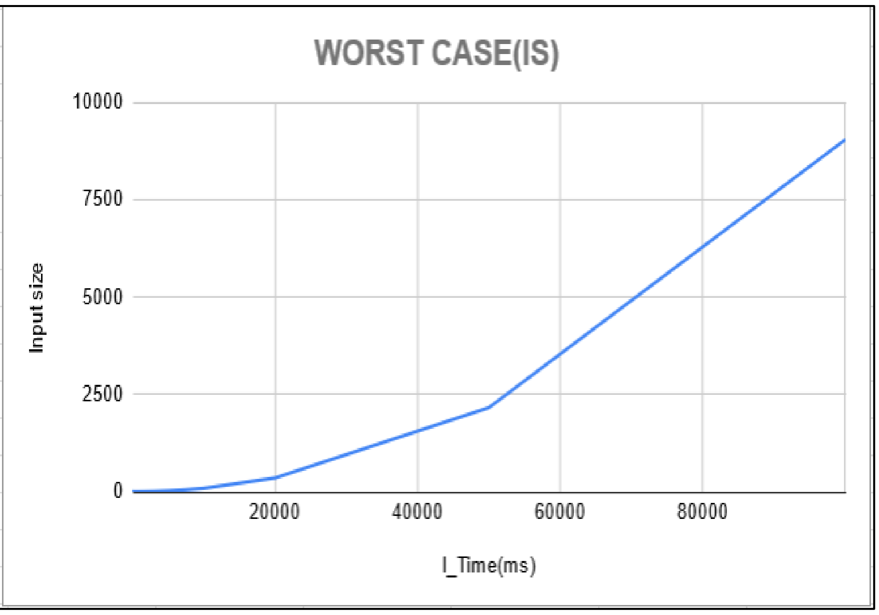


Figure 4. Insertion Sort's Worst Case: O(n^2)

In the average case of the algorithm, the graph represents a function that equals O(n^2). It forms a curve that rises sharply as the input size (n) increases.

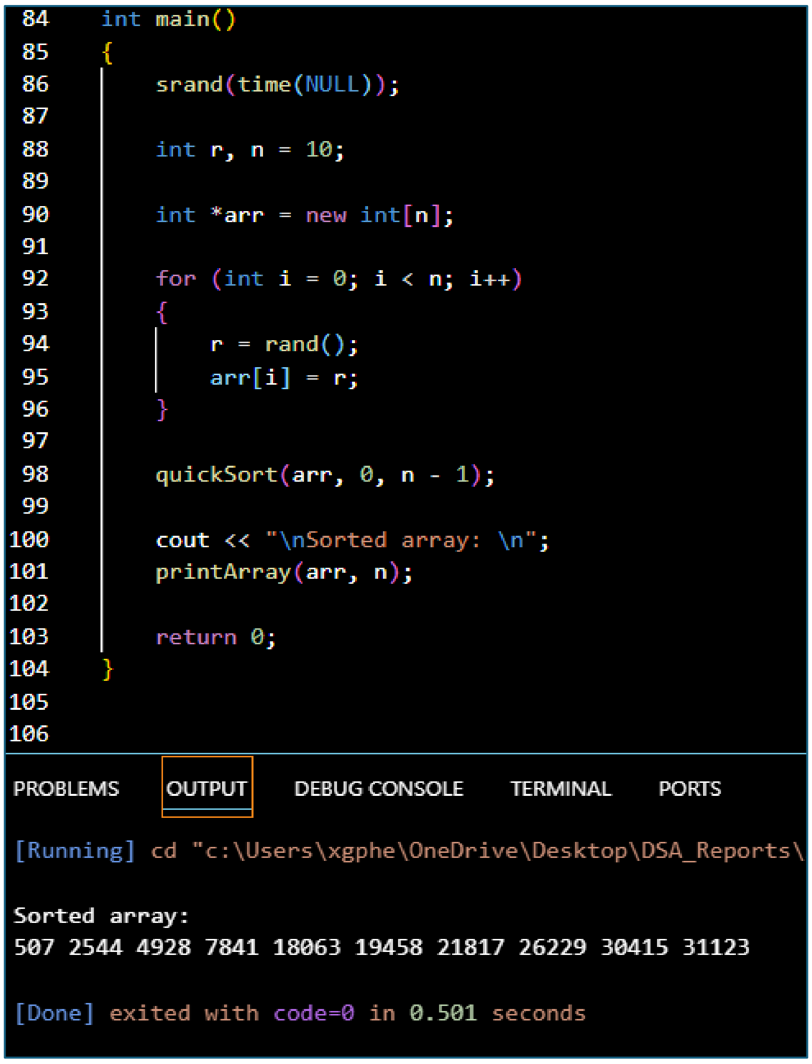
In comparison to the graph of the quicksort algorithm (n log n), this curve rises much faster. Therefore, for the same input values, computation takes longer than that for quicksort.

II. 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(n^2) occurs when the pivot at each step consistently results in highly unbalanced partitions.



```
int main()
{
    srand(time(NULL));

    int r, n = 10;

    int *arr = new int[n];

    for (int i = 0; i < n; i++)
    {
        r = rand();
        arr[i] = r;
    }

    quickSort(arr, 0, n - 1);

    cout << "\nSorted array: \n";
    printArray(arr, n);

    return 0;
}
```

```
[Running] cd "c:\Users\xgphe\OneDrive\Desktop\DSA_Reports\

Sorted array:
507 2544 4928 7841 18063 19458 21817 26229 30415 31123

[Done] exited with code=0 in 0.501 seconds
```

Figure 4. Quicksort Output

Q_Time(ms)	Input size
0.002	10
0.004	50
0.009	100
0.019	200
0.047	500
0.118	1000
0.153	1500
0.213	2000
0.372	3000
0.518	5000
0.868	7500
1.094	10000
2.63	20000
6.33	50000

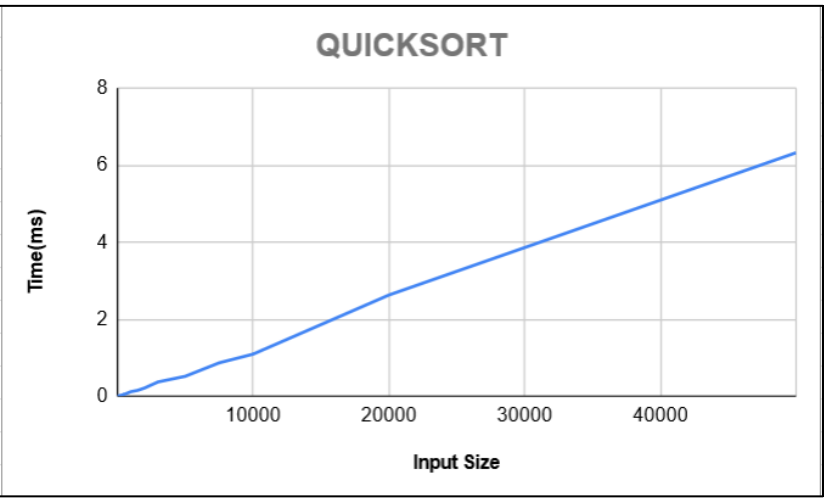
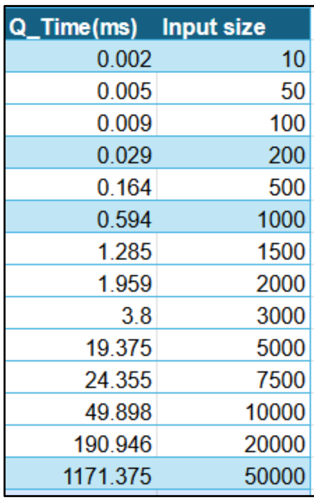


Figure 5. Quicksort Average Case



Q_Time(ms)	Input size
0.002	10
0.005	50
0.009	100
0.029	200
0.164	500
0.594	1000
1.285	1500
1.959	2000
3.8	3000
19.375	5000
24.355	7500
49.898	10000
190.946	20000
1171.375	50000

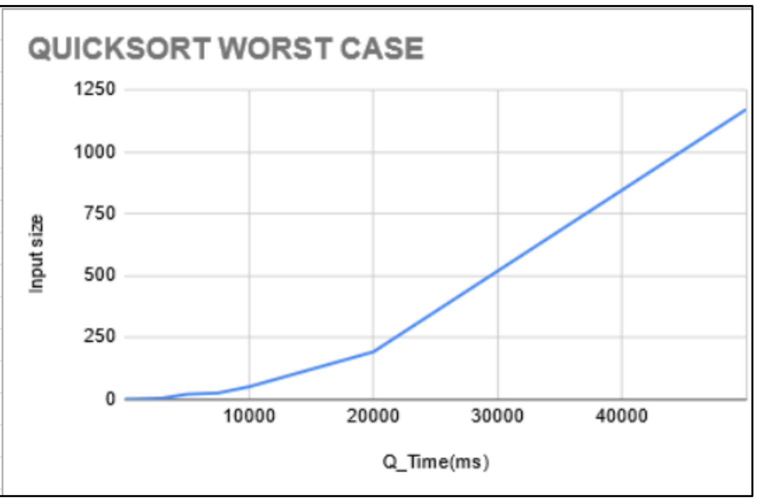


Figure 6. Quicksort Worst Case

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 n^2 graph as the input size (n) increases.

In comparison to the graph of the insertion sort algorithm (n^2), 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 $O(n^2)$ 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.