K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: B-2 **Roll No.: 16010122151**

Experiment No.___1____

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Implementation of selection sort/ Insertion sort

Objective: To analyse performance of sorting methods

CO to be achieved:

CO 1 Analyze the asymptotic running time and space complexity of algorithms.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Insertion_sort**
4. **http://www.sorting-algorithms.com/insertion-sort**
5. **http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion_sort.html**
6. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm**
7. **http://en.wikipedia.org/wiki/Selection_sort**
8. **http://www.sorting-algorithms.com/selection-sort**
9. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm**
10. **http://courses.cs.vt.edu/~csonline/Algorithms/Lessons/SelectionCardSort/selectioncardsort.html**

Pre Lab/ Prior Concepts:

Data structures, sorting techniques.

Historical Profile:

There are various methods to sort the given list. As the size of input changes, the performance of these strategies tends to differ from each other. In such case, the priori analysis can helps the engineer to choose the best algorithm.

New Concepts to be learned:
Space complexity, time complexity, size of input, order of growth.

Topic: Sorting Algorithms

Theory: Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, 1< k ≤n, yielding k sub problems. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and-conquer design are the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

Algorithm Insertion Sort

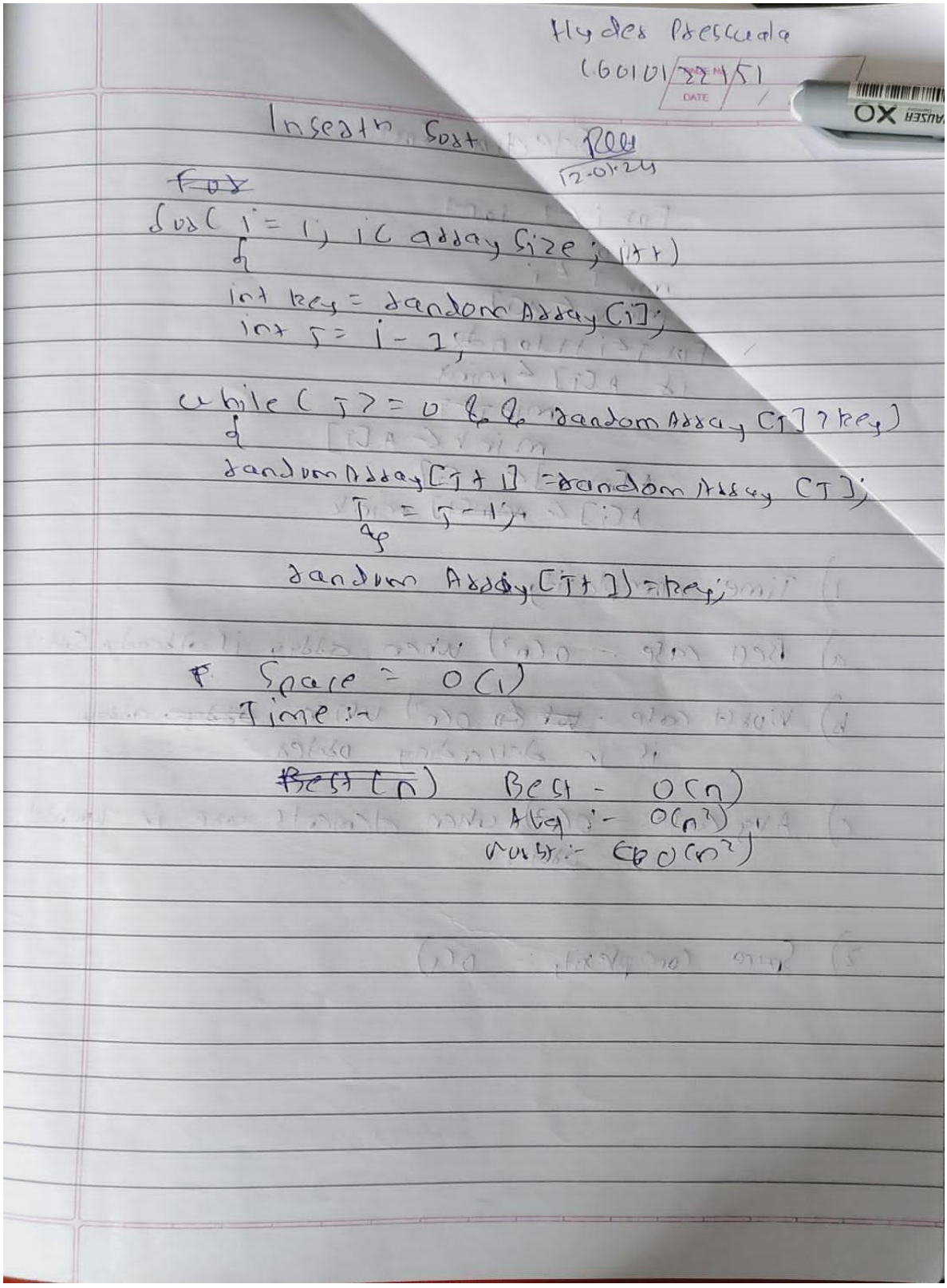
INSERTION_SORT (*A*,*n*)
//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.
//The array *A* is sorted in place: the numbers are rearranged within the array
// A[1..n] of eletype, n: integer

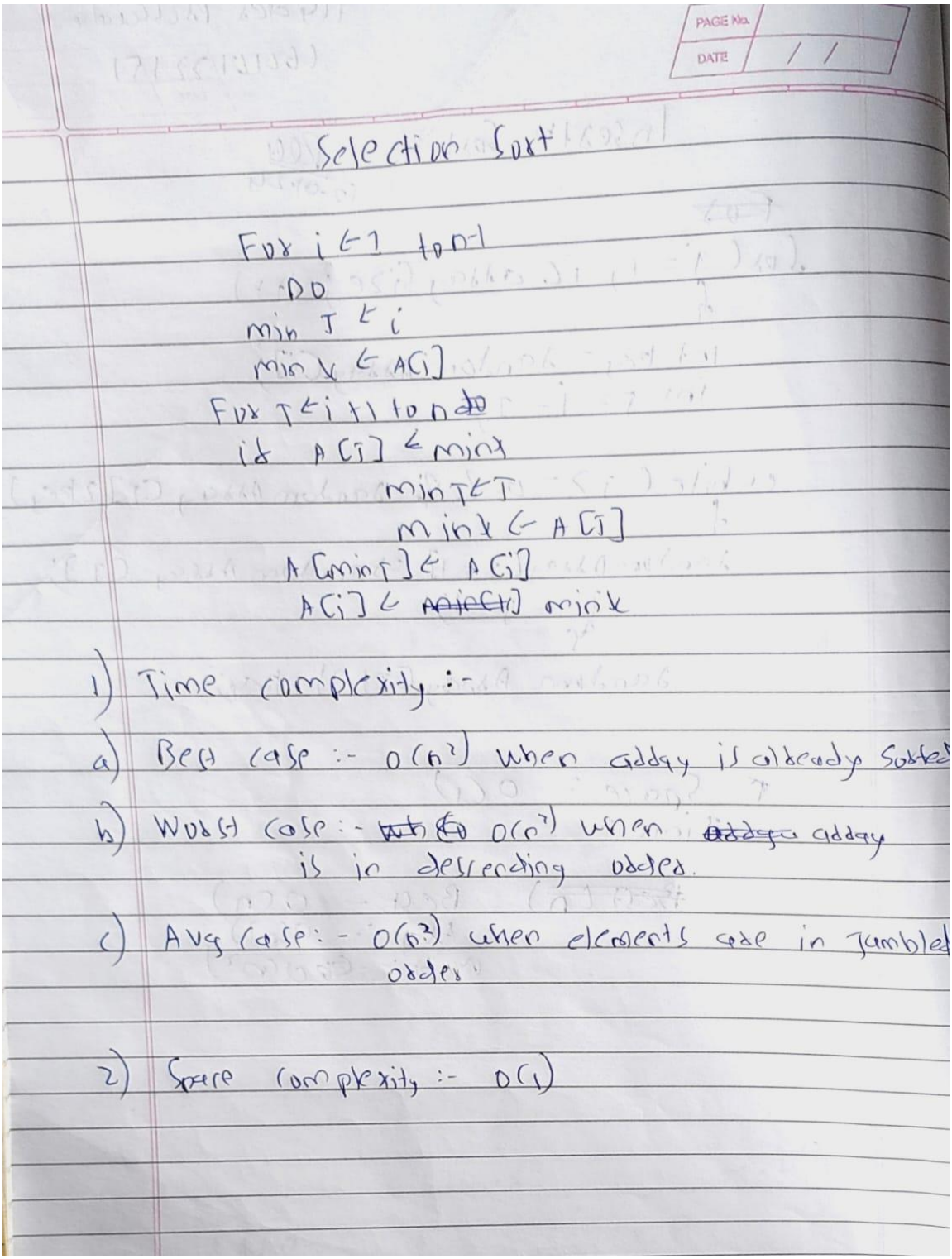
```
FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
            i ← j − 1
            WHILE i > 0 and A[i] > key
                DO A[i +1] ← A[i]
                    i ← i − 1
             A[i + 1] ← key
```

Algorithm Selection Sort

SELECTION_SORT (A,n)
//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.
//The array *A* is sorted in place: the numbers are rearranged within the array
// A[1..n] of eletype, n: integer

```
FOR i ← 1 TO n-1 DO
   min j ← i;
   min x ← A[i]
  FOR j ← i + 1 to n do
     IF A[j] < min x then
        min j ← j
        min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```


Insertn Sort
for(i = 1; i < array size; i++)
{
int key = random Array[i];
int j = i - 1;
while(j >= 0 && random Array[j] > key)
{
random Array[j+1] = random Array[j];
j = j - 1;
}
random Array[j+1] = key;
Space = O(1)
Time :-
Best - O(n)
Avg :- O(n^2)
Worst :- O(n^2)



Selection Sort

For i ← 1 to n-1
DO
min I ← i
min V ← A[i]
For I ← i + 1 to n ~~do~~
if A[I] < min
minI ← I
minV ← A[I]
A[minI] ← A[i]
A[i] ← ~~A[i+1]~~ minV

1) Time complexity :-

a) Best case :- $O(n^2)$ when ~~a~~dday is already sorted

b) Worst case :- ~~worst~~ $O(n^2)$ when ~~addaya~~ dday is in descending order.

c) Avg case :- $O(n^2)$ when elements are in jumbled order.

2) Space complexity :- $O(1)$

Code Implementation of Selection Sort:-

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    int swapCount = 0;
    int comparisonCount = 0;

    srand(static_cast<unsigned int>(time(0)));

    int arraySize, Maxi;
    cout << "Enter Size of Array: ";
    cin >> arraySize;
    int randomArray[arraySize];

    cout << "Enter Maximum value You want in Array: ";
    cin >> Maxi;
    for (int i = 0; i < arraySize; ++i)
    {
        randomArray[i] = rand() % Maxi;
    }

    cout << "Original array: ";
    for (int i = 0; i < arraySize; ++i)
    {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < arraySize - 1; ++i)
    {
        int minIndex = i;
        for (int j = i + 1; j < arraySize; ++j)
        {
            comparisonCount++; // Counting comparisons

            if (randomArray[j] < randomArray[minIndex])
            {
                minIndex = j;
            }
        }

        if (minIndex != i)
```

```
        {
            // Swap elements
            int temp = randomArray[i];
            randomArray[i] = randomArray[minIndex];
            randomArray[minIndex] = temp;
            swapCount++; // Counting swaps
        }
    }

    cout << "Sorted array: ";
    for (int i = 0; i < arraySize; ++i)
    {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    cout << "Comparisons: " << comparisonCount << endl;
    cout << "Swaps: " << swapCount << endl;

    return 0;
}
```

Output:-

```
Enter Size of Array: 8
Enter Maximum value You want in Array: 50
Original array: 12 45 23 5 34 2 48 10
Sorted array: 2 5 10 12 23 34 45 48
Comparisons: 28
Swaps: 7
```

Code Implementation for Insertion Sort:-

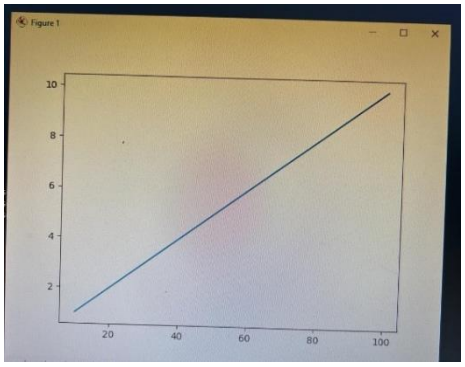
```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    int swapCount = 0;
    int comparisonCount = 0;

    srand(static_cast<unsigned int>(time(0)));
```

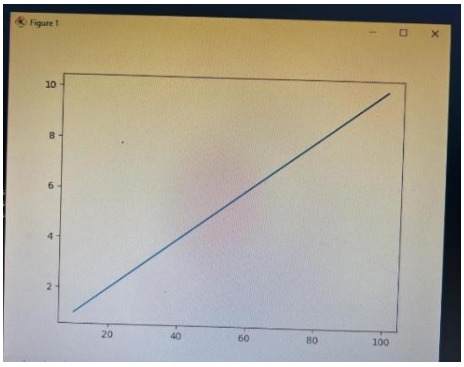
```
    int arraySize, Maxi;
    cout << "Enter Size of Array: ";
    cin >> arraySize;
    int randomArray[arraySize];

    cout << "Enter Maximum value You want in Array: ";
    cin >> Maxi;
    for (int i = 0; i < arraySize; ++i)
    {
        randomArray[i] = rand() % Maxi;
    }
    for (int i = 0; i < arraySize; ++i)
    {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    for (int i = 1; i < arraySize; ++i)
    {
        int key = randomArray[i];
        int j = i - 1;

        while (j >= 0 && randomArray[j] > key)
        {
            randomArray[j + 1] = randomArray[j];
            j = j - 1;
        }
        randomArray[j + 1] = key;
    }

    cout << "Sorted array: ";
    for (int i = 0; i < arraySize; ++i)
    {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    return 0;
}
```

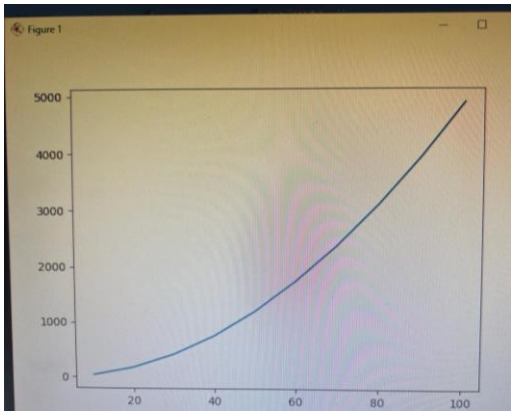
Output:-

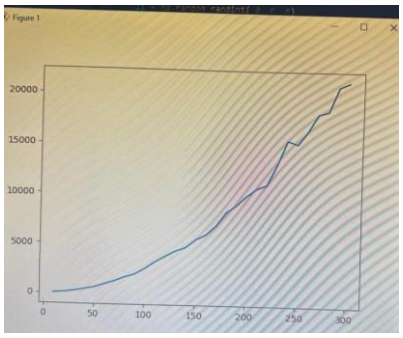
```
Enter Size of Array: 10
Enter Maximum value You want in Array: 100
78 12 45 89 23 67 1 56 34 90
Sorted array: 1 12 23 34 45 56 67 78 89 90
```

Graphs for varying input sizes:









CONCLUSION:-

We have explored the various sorting algorithms via this assignment. Looking at their effectiveness, usability, and flexibility with different data sets. The decision between the two methods ultimately comes down to the particular needs of the work at hand, even if both have advantages and disadvantages, such as insertion sort's flexibility to virtually sorted arrays and selection sort's constant time complexity. When creating algorithms for practical uses, we may make more informed choices if we have a solid grasp of the performance subtleties of various sorting methods.