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

Batch: E-2 Roll No.: 16010123325

Experiment No._1__

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

Signature of the Staff In-charge with date

Title: Study, Implementation, and Performance Comparison of Selection Sort, Insertion Sort to analyze sorting strategies.

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 algorithms",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 a case, the priori analysis can help 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

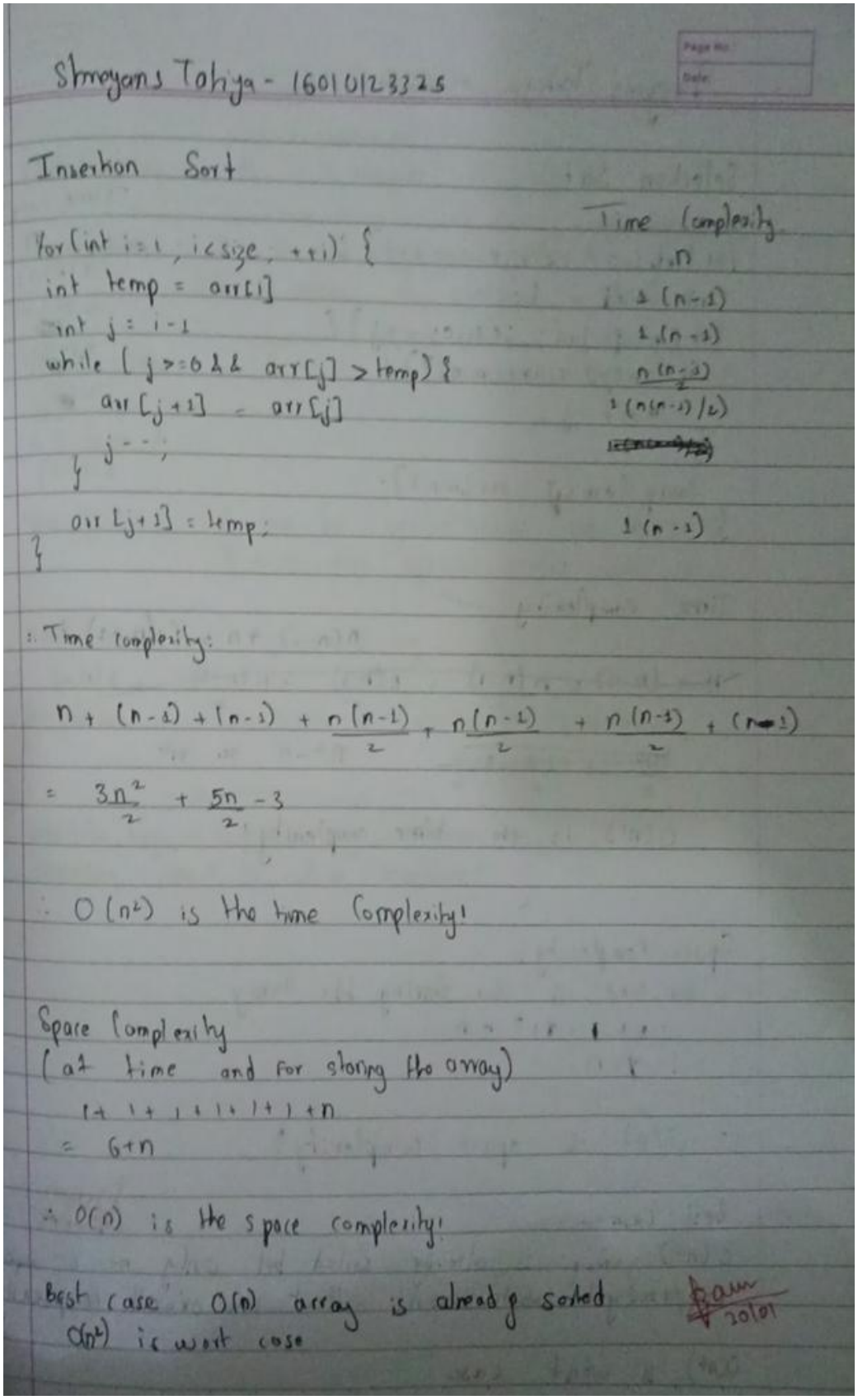
```
FOR j ← 2 TO length[A]
    DO  key ← A[j]
        {Put A[j] into the sorted sequence A[1 . . j − 1]}
        i ← j − 1
        WHILE i > 0 and A[i] > key
                DO A[i +1] ← A[i]
                    i ← i − 1
        A[i + 1] ← key
```

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
```

The space complexity of Insertion sort & Time complexity for Insertion sort:



Shreyans Tohiya - 16010123325

Insertion Sort

```
for(int i=1, i<size, ++i) {
int temp = arr[i]
int j = i-1
while [ j>=0 && arr[j] > temp) {
    arr[j+1] = arr[j]
    j--;
}
arr[j+1] = temp;
}
```

Time complexity

	n
	1 + (n-1)
	1 + (n-1)
	n(n-1)/2
	1 + (n(n-1)/2)
	[illegible]
	1 (n-1)

∴ Time complexity:

$$n + (n-1) + (n-1) + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + (n-1)$$

$$= \frac{3n^2}{2} + \frac{5n}{2} - 3$$

∴ $O(n^2)$ is the time complexity!

Space complexity
(at time and for storing the array)

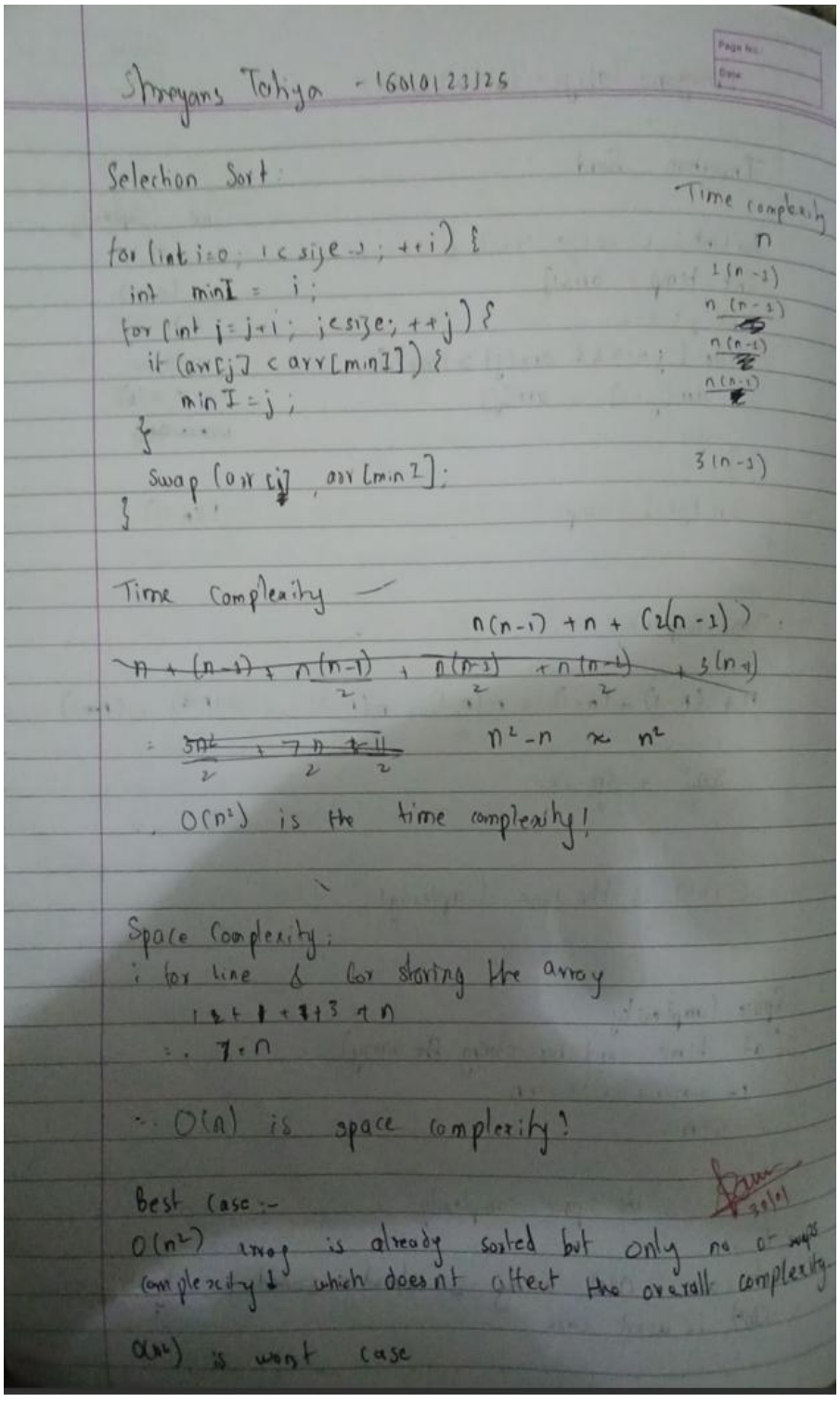
1 + 1 + 1 + 1 + 1 + 1 + n

= 6 + n

∴ $O(n)$ is the space complexity!

Best case : O(n) array is already sorted
$O(n^2)$ is worst case

The space complexity of Selection sort & Time complexity for selection sort:



Graphs for varying input sizes: (Insertion Sort & Selection sort)

Code-

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <fstream>
#include <cstdlib>
#include <algorithm>
using namespace std;


void selectionSort(vector<int> &arr) {
    for (int i = 0; i < arr.size(); ++i) {
        int mini = i;
        for (int j = i + 1; j < arr.size(); ++j) {
            if (arr[j] < arr[mini]) {
                mini = j;
            }
        }
        swap(arr[i], arr[mini]);
    }
}

void insertionSort(vector<int> &arr) {
    for (int i = 1; i < arr.size(); ++i) {
        int j = i;
        while (j > 0 && arr[j] < arr[j-1]) {
            swap(arr[j], arr[j-1]);
            --j;
        }
    }
}

vector<int> generateRandomArray(int n) {
    vector<int> arr(n);
    for (int i = 0; i < n; ++i) {
        arr[i] = rand() % 100;
    }
    return arr;
}
```

```cpp
int main() {
    ofstream file("output.csv");
    file << "n,selection,insertion \n";

    for (int i = 100; i <= 1000; i+=100) {
        vector<int> arr = generateRandomArray(i);
        vector<int> arr1 = arr;
        vector<int> arr2 = arr;
        vector<int> arr3 = arr;

        auto start1 = chrono::high_resolution_clock::now();
        selectionSort(arr1);
        auto end1 = chrono::high_resolution_clock::now();
        double time_taken1 = chrono::duration_cast<chrono::milliseconds>(end1
- start1).count();

        auto start2 = chrono::high_resolution_clock::now();
        insertionSort(arr2);
        auto end2 = chrono::high_resolution_clock::now();
        double time_taken2 = chrono::duration_cast<chrono::milliseconds>(end2
- start2).count();

        file << i << "," << time_taken1 << "," << time_taken2 <<"\n";
    }
    file.close();
}
```

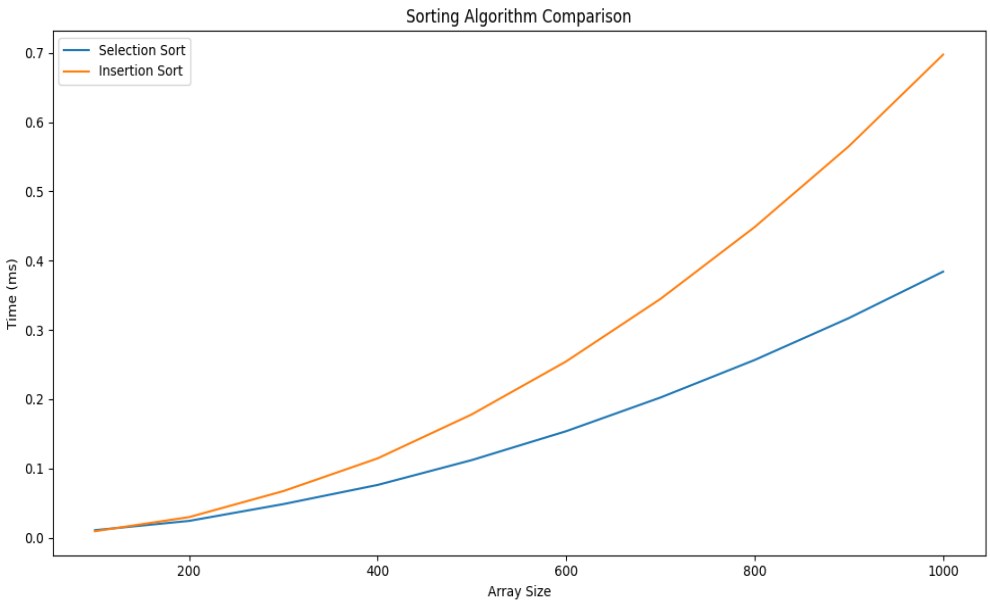
Python File (For Plotting) -

```python
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file
df = pd.read_csv('output.csv')

plt.figure(figsize=(10, 6))
plt.plot(df['n'], df['insertion'], 'r-o', label='Insertion Sort')
plt.plot(df['n'], df['selection'], 'b-o', label='Selection Sort')
plt.xlabel('Input Size (n)')
plt.ylabel('Time (milliseconds)')
plt.title('Time Complexity Comparison')
plt.legend()
plt.grid(True)
plt.savefig('OG.png')
plt.show()
```

Table-

Array Size	Selection Sort Time (ms)	Insertion Sort Time (ms)
100	0.010857871600000001	0.009316088
200	0.024290941	0.029769848
300	0.04853659	0.067329045
400	0.076211703	0.114586979
500	0.11213531260000001	0.17817942550000002
600	0.153805689	0.25459909
700	0.202407541	0.344780625
800	0.256657125	0.448617322
900	0.317190743	0.5653845655999999
1000	0.384270045	0.6977024285000001

Graph-



CONCLUSION:

Sorting algorithms have varying efficiency based on input size and order. While simpler methods like Insertion Sort and Selection Sort work well for small datasets. Understanding their time and space complexity helps in choosing the right sorting strategy.