Experiment No: 01

Experiment Name: Comparison Between Bubble sort, Selection Sort and Insertion sort (Complexity Analysis)

Bubble Sort

Theory: Bubble Sort is used to order a given collection of n elements that are provided in the form of an array with n elements. Bubble Sort sorts the elements according to their values after comparing each element individually. Bubble sort will begin by comparing the first and second items of the array, swapping the components if the first element is bigger than the second, and then compare the second and third elements, and so on, if the array has to be sorted in ascending order. This method must be repeated n-1 times if there are n total items. Bubble sort contains two loops: an inner loop and an outer loop. The inner loop performs O(n) comparisons.

Worst Case: In the worst-case scenario, the outer loop runs O(n) times. As a result, the worst-case time complexity of bubble sort is O(n x n) = O(n^2).

Best Case: In the best-case scenario, the array is already sorted, but just in case, bubble sort performs O(n) comparisons. As a result, the time complexity of bubble sort in the best-case scenario is O(n).

Average Case: Bubble sort may require $n/2$ passes and O(n) comparisons for each pass in the average case. As a result, the average case time complexity of bubble sort is O($n/2$ x n) = O (n^2)

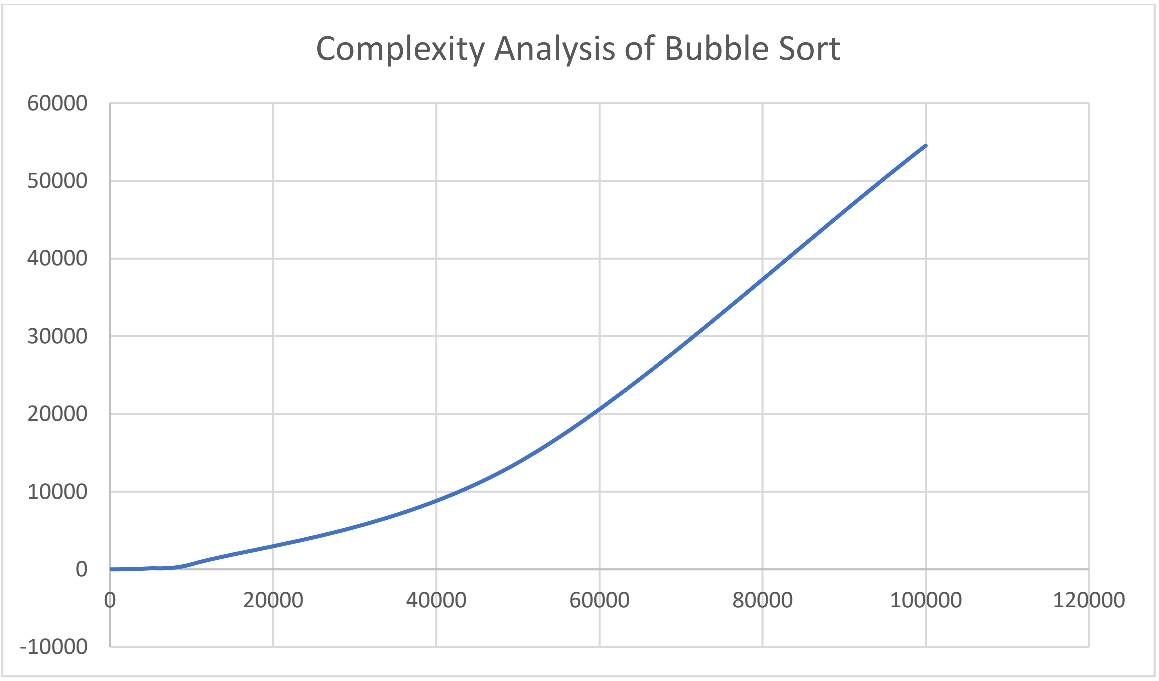
Implementation:

```
#include <bits/stdc++.h>
#include <string.h>
#include <string>
using namespace std;

void _bubbleSort(int arr[], int n){
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

int main(){
    #ifndef WORK_STATION
    freopen("in.txt","r",stdin);
    freopen("out.txt","w",stdout);
    #endif

    int n; cin >> n;
    int arr[n+1];
    for(int i=0; i <n; ++i) cin >> arr[i];

    clock_t start = clock();
    _bubbleSort(arr,n);
    clock_t finish = clock();

    for(int i=0; i <n; ++i) cout << arr[i] << " "; cout<<"\n";

    // Time taken by algorithm
    double time = finish-start;
    cout << "Time taken: " << time << "miliseconds\n";
}
/*
 @_author
 Md. Shefat Zeon
 1903147,CSE,RUET
*/
```

Input Output:

Input (Size)	Output (milliseconds)
100	1
1000	1
5000	137
10000	654
50000	13725
100000	54543

Output Analysis:



Selection Sort

Theory: Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on.
Number of comparisons:

$$(n-1) + (n-2) + (n-3) + \ldots\ldots + 1 = \frac{n(n-1)}{2}$$

nearly equals to n^2. Therefore, the selection sort algorithm shows a time complexity of $O(n^2)$.

Best Case Complexity: The selection sort algorithm has a best-case time complexity of O(n2) for the already sorted array.
Average Case Complexity: The average-case time complexity for the selection sort algorithm is O(n2), in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
Worst Case Complexity: The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.

Implementation:

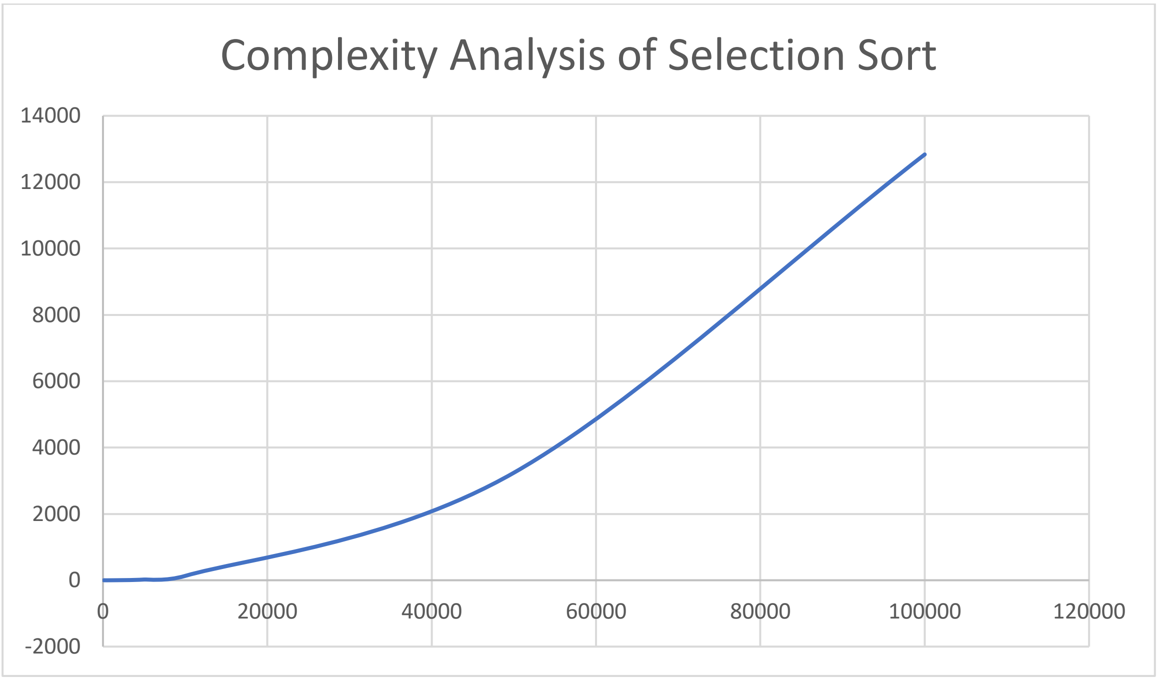
```
#include <bits/stdc++.h>
#include <string.h>
#include <string>
#include <cstdlib>
using namespace std;

void _selectionSort(int arr[], int n){
    int i, j, index;
    for (int i = 0; i < n-1; i++){
        index = i;
        for (j = i+1; j < n; j++) if (arr[j] < arr[index]) index = j;
        if(index!=i)swap(arr[index], arr[i]);
    }
}

void solve(){
    int n; cin >> n;
    int arr[n+1];
    for(int i=0; i <n; ++i) arr[i]=rand();

    clock_t startTime = clock();
    _selectionSort(arr,n);
    clock_t finishTime = clock();

    double time = finishTime - startTime;
    cout << "Input Size: " << n << "\n";
    cout << "Time taken: " << time << " miliseconds\n";
}

int main(){
    #ifndef WORK_STATION
    freopen("in.txt","r",stdin);freopen("out.txt","w",stdout);
    #endif
    int t; cin >> t;
    while(t--){
        solve();
    }

}
/*
```

```
 @_author
 Md. Shefat Zeon
 1903147,CSE,RUET
*/
```

Input Output:

Input (Size)	Output (milliseconds)
100	0
1000	0
5000	22
10000	131
50000	3241
100000	12835

Output Analysis:



Insertion Sort

Theory: Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. In the insertion sort technique, we start from the second element and compare it with the first element and put it in a proper place. Then we perform this process for the subsequent elements.

We compare each element with all its previous elements and put or insert the element in its proper position. Insertion sort technique is more feasible for arrays with a smaller number of elements.

Worst Case Complexity: Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs. Each element has to be compared with each of the other elements so, for every n_{th} element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = $n(n - 1)$ which is nearly n^2 The time complexity will be $O(n^2)$

Best Case Complexity: When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear. O(n)

Average Case Complexity: It occurs when the elements of an array are in jumbled order (neither ascending nor descending). $O(n^2)$.

Implementation:

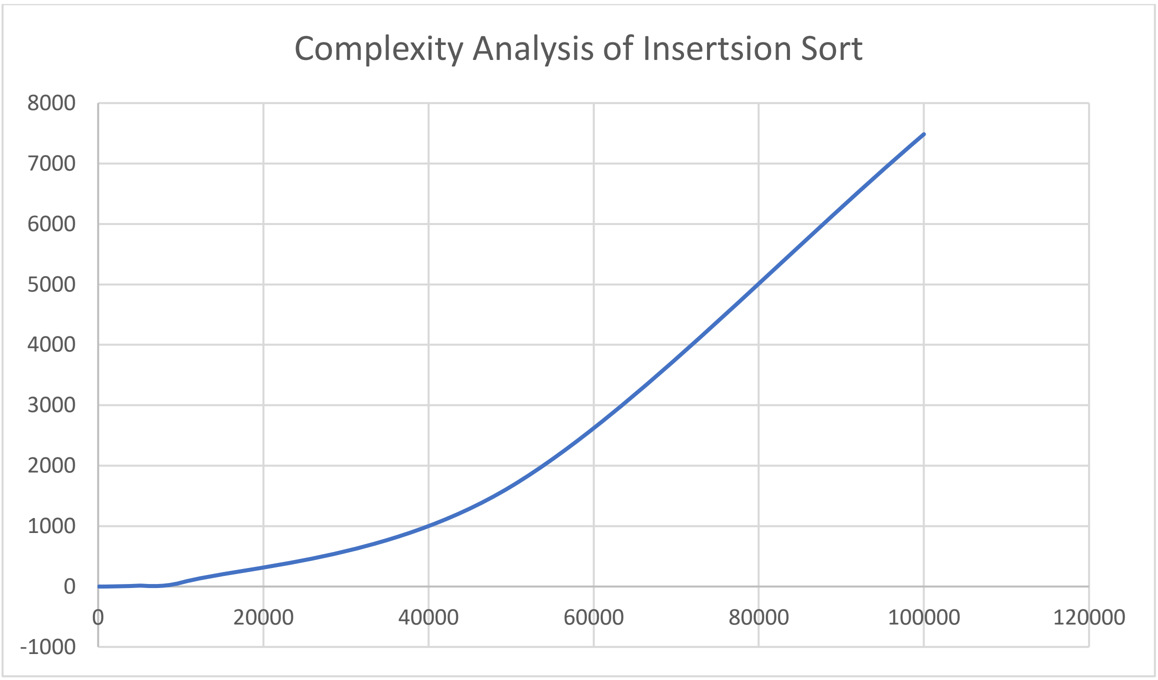
```
#include <bits/stdc++.h>
#include <cstdlib>
using namespace std;

void _insertionSort(int arr[], int n) {
    int key, j;
    // started iterating from 2nd element
    for (int i = 1; i < n; i++){
        key = arr[i];
        for (j = i - 1; j >= 0 && arr[j] > key; --j){
            arr[j + 1] = arr[j];
        }
        arr[j + 1] = key;
    }
}

void solve(){
    int n; cin >> n; int arr[n+1];
    for(int i=0; i <n; ++i) arr[i] = rand();

    clock_t startTime = clock();
    _insertionSort(arr,n);
    clock_t finishTime = clock();

    double time = finishTime - startTime;
    cout << "Input Size: " << n << "\n";
    cout << "Time taken: " << time << " miliseconds\n";
}


int main(){
    #ifndef WORK_STATION
    freopen("in.txt","r",stdin); freopen("out.txt","w",stdout);
    #endif

    int t; cin >> t;
    while(t--){
        solve();
    }
}
```

```
/*
 @_author
 Md. Shefat Zeon
 1903147,CSE,RUET
*/
```

Input Output:

Input (Size)	Output (Mili-seconds)
100	0
1000	0
5000	15
10000	63
50000	1656
100000	7485

Output Analysis:

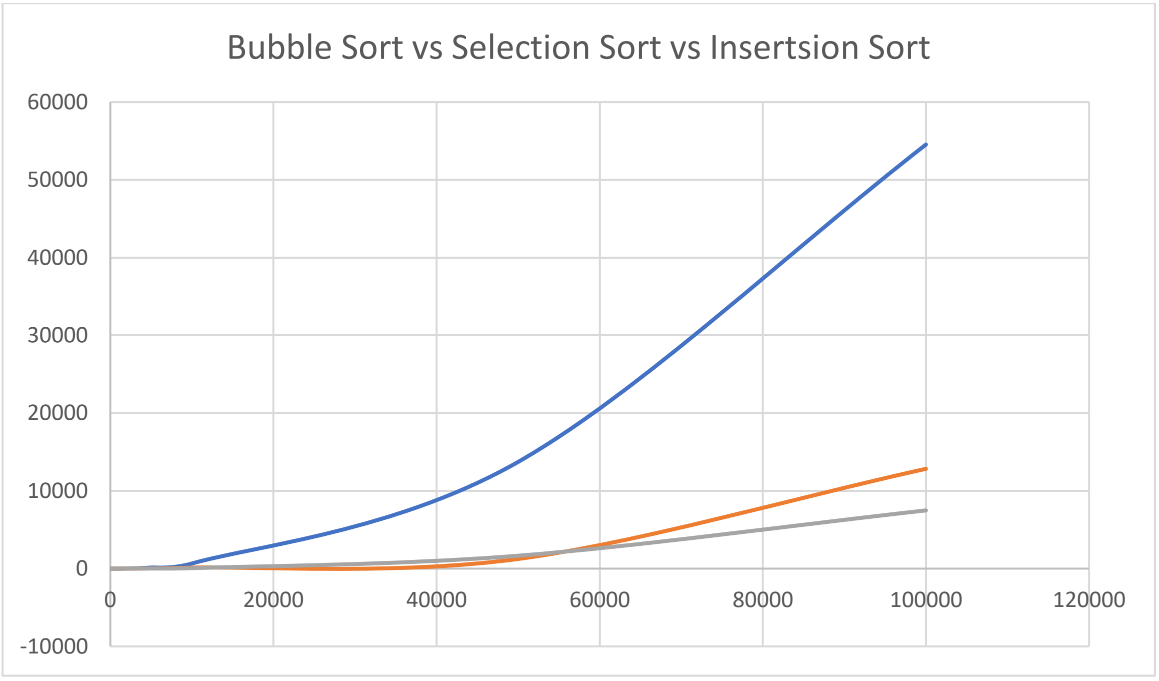


Comparison Between Bubble sort, Selection Sort and Insertion sort:

Runtime for different values:

No of Inputs	Time taken by Algorithms (milliseconds)		
	Bubble Sort	Selection Sort	Insertion Sort
100	1	0	0
1000	1	0	0
5000	137	22	15
10000	654	131	63
50000	13725	1241	1656
100000	54543	12835	7485

Graph: Blue Represents Bubble Sort, Orange Represents Selection Sort , Gray Insertion Bubble Sort



Analysis:

Here all the sorting algorithms can be used for different applications. Bubble, Selection and Insertion sort all have average time complexities O(n^2) which are quadratic in nature. When the data is almost sorted, where Bubble Sort and Insertion Sort needs few number of swaps there Selection Sort requires the same time even within almost sorted data. In case of stability, Bubble and Insertion sort is more stable than Selection sort. All three algorithms have a quadratic worst-case time complexity and thus they are slow for large datasets. Bubble sort makes many comparisons and swaps and thus is not efficient if swapping or comparison is a costly operation. Insertion sort makes a lot of swaps and thus becomes inefficient if swapping operations are costly. But, in comparison among these three sorting algorithms, Insertion sort is the most efficient one as it takes comparatively less time than other two. So, in terms of efficiency, we can conclude that Insertion sort is
the best out of the three, then selection sort and at last, bubble sort.