Experiment No.	2
Aim	Experiment on finding the running time of an algorithm.
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:

1. Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is O(n^2), where n is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when:
- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

2. Insertion Sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Algorithm:

1. Selection Sort

Step 1: Initialize minimum value(min_idx) to location 0.
Step 2: Traverse the array to find the minimum element in the array.
Step 3: While traversing if any element smaller than min_idx is found then swap both the values.
Step 4: Then, increment min_idx to point to the next element.
Step 5: Repeat until the array is sorted.

2. Insertion Sort

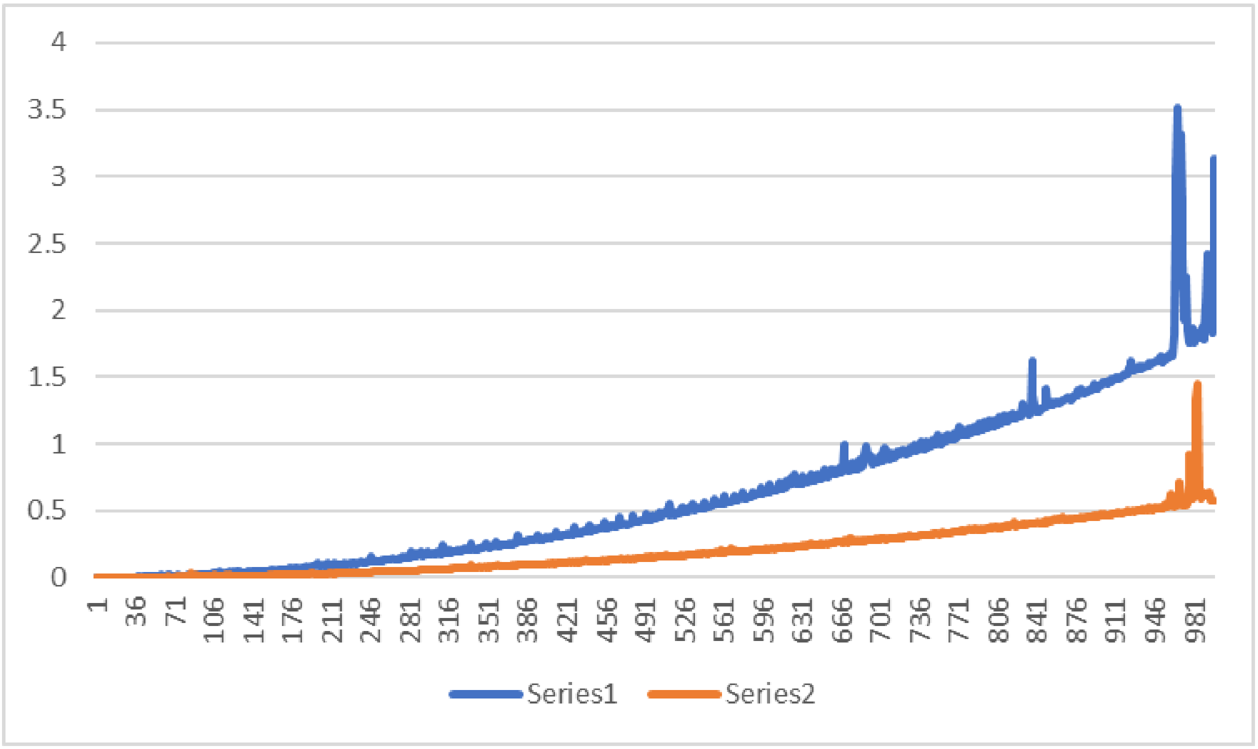
Step 1: If the element is the first element, assume that it is already sorted. Return 1.
Step 2: Pick the next element, and store it separately in a key.
Step 3: Now, compare the key with all elements in the sorted array.
Step 4: If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 5: Insert the value.
Step 6: Repeat until the array is sorted.

Program:

```c
#include<stdio.h>
#include <stdlib.h>
#include<math.h>
#include<time.h>
int main(){
    void getData(int x, int arr[]), insertionSort(int x,int arr[]);
    void selectionSort(int x,int arr[]), printArray(int x,int arr[]);
    int arr[100000];
    __clock_t start,end;
    printf("Selection Sort:\n")
    for(int i=0;i<1000;i++){
        getData(i+1,arr);
        start = clock();
        selectionSort(i+1,arr);
        end = clock();
        double duration = ((double)(end - start))/CLOCKS_PER_SEC;
        printf("%lf ",duration);
    }
    printf("Insertion Sort:\n")
    for(int i=0;i<1000;i++){
        getData(i+1,arr);
        start = clock();
        insertionSort(i+1,arr);
        end = clock();
        double duration = ((double)(end - start))/CLOCKS_PER_SEC;
        printf("%lf ",duration);
    }
    return 0;
}
void getData(int x, int arr[]){
    for(int i=0;i<x*100;i++){
        arr[i] = rand()%1000 + 1;
    }
}
void insertionSort(int x,int arr[]){
    for (int i = 1; i < x*100; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void selectionSort(int x,int arr[]){
    int i, j, min_idx;
    for (i = 0; i < x*100 -1; i++)
    {
        min_idx = i;
        for (j = i+1; j < x*100; j++){
            if (arr[j] < arr[min_idx]){
                min_idx = j;
            }
        }
        if(min_idx != i){
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
}
```

Observations:

The graph of the obtained results when plotted is as follows:



Series 1: Selection Sort Series 2: Insertion Sort

Conclusion:

From the obtained results of the experiment it can be concluded that both the algorithms perform equally good for a smaller amount of numbers. As the amount of numbers to be sorted increases, the selection sort algorithm takes more amount of time. The time taken by the insertion sort algorithm is less than that.