Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Experiment No. 1-b

Aim – Experiment on finding the running time of an algorithm.

Details – The understanding of running time of algorithms is explored by implementing two basic sorting algorithms namely Insertion and Selection sorts. These algorithms work as follows.

Insertion sort– It 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.

Selection sort– It first finds the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. 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.

Problem Definition & Assumptions – For this experiment, you need to implement two sorting algorithms namely Insertion and Selection sort methods. Compare these algorithms based on time and space complexity. Time required to sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono.

You have togenerate1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100 integers numbers with array indexes numbers A[0..99], A[0..199], A[0..299],…, A[0..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Insertion and Selection by plotting the time required to sort 100000 integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot representsthe tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.

Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers.

Important Links:

1. C/C++ Rand function Online library
   https://cplusplus.com/reference/cstdlib/rand/
2. Time required calculation Online library-
   https://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now
3. Draw 2-D plot using OpenLibre/MS Excel
   https://support.microsoft.com/en-us/topic/present-your-data-in-a-scatter-chart-or-a-line-chart-4570a80f-599a-4d6b-a155-104a9018b86e

---

Input –

1) Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100 integer numbers to Insertion and Selection sorting algorithms.

Output –

1) Store the randomly generated 100000 integer numbers to a text file.
2) Draw two 2D plot of all functions such that the x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the running time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
3) Comment on Space complexity for two sorting algorithms.

Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	1B
Aim	Experiment on finding the running time of an algorithm.
Name	Allen Andrew
UID No.	2021300006
Class & Division	SE COMPS A

Algorithms:

[A] For insertion sort

I. Start iterating the concerned array from the second element.
II. Compare the current element with the previous element.
III. If the two elements are found in proper order, move to the next element.
IV. If the two elements are not in proper order, shift the previous elements till the proper location of the current element is found.
V. Repeat steps II to IV till the end of the array.

[B] For selection sort

I. Start iterating the array from start to end and find the index value of the minimum element.
II. Swap it with the first element of the unsorted array.
III. Repeat till there are no elements left in the unsorted section of the array.

Program:

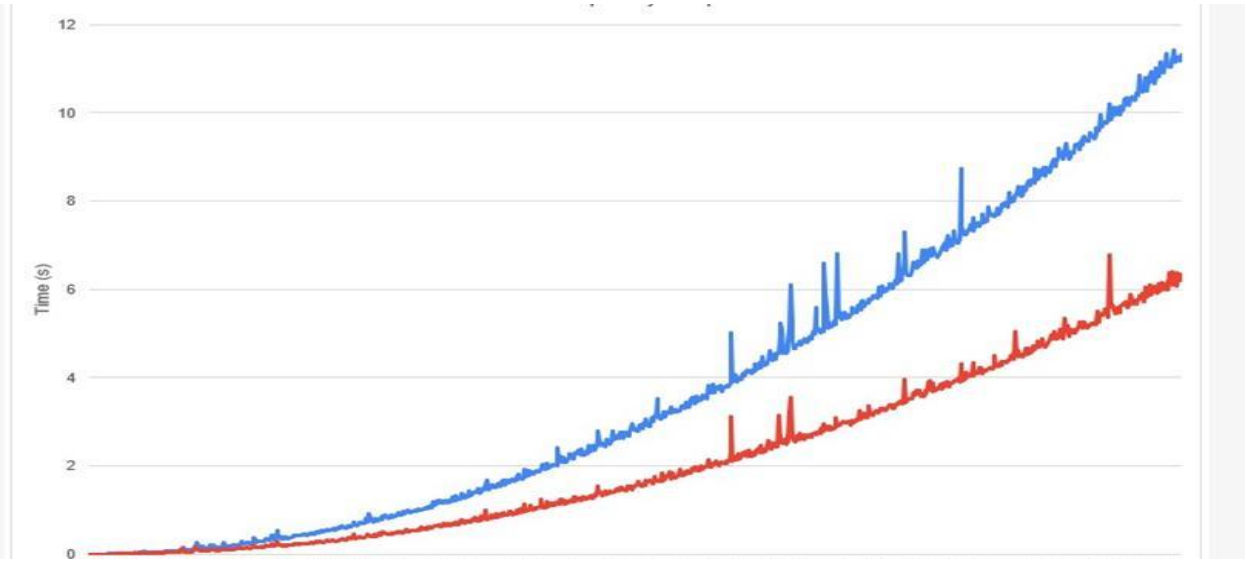
```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
void selectionsort(int arr[],int n) {
for(int i=0;i<n;i++)
{
int min_ind=i;
for(int j=i+1;j<n;j++)
{
if(arr[j]<arr[min_ind])
min_ind=j; }
if(min_ind!=i) {
int t=arr[i];
arr[i]=arr[min_ind];
arr[min_ind]=t; } } }
void insertionsort(int arr[],int n) {
for(int i=1;i<n;i++)
{
int j=i-1;
int key=arr[i];
while(j>=0 && arr[j]>key) {
arr[j + 1] = arr[j];
```

```c
int key=arr[i];
while(j>=0 && arr[j]>key) {
arr[j + 1] = arr[j];
j = j - 1; }
arr[j + 1] = key; } }
void generate_numbers()
{
FILE *ptr;
ptr=fopen("number.txt","w");
for(int i=0;i<100000;i++)
{
fprintf(ptr,"%d\n",rand() % 100000);
}
fclose(ptr);
}
void operation()
{
FILE *ptr;
ptr=fopen("number.txt","r");
for(int j=0;j<100000;j+=100) {
int arr1[j];
int arr2[j];
for(int i=0;i<j;i++)
```

```
int arr2[j];
for(int i=0;i<j;i++)
{
fscanf(ptr,"%d\n",&arr1[i]);
}
for(int i=0;i<j;i++)
{
arr2[i]=arr1[i];
}
clock_t start_selection=clock();
selectionsort(arr1,j);
clock_t end_selection = clock();
double currs=(double)(end_selection-start_selection)/CLOCKS_PER_SEC;
clock_t start_insertion=clock();
insertionsort(arr2,j);
clock_t end_insertion=clock();
double curri=(double)(end_insertion-start_insertion)/CLOCKS_PER_SEC;
printf("\n%d\t%f\t%f",j,currs,curri);
} }
int main()
{
generate_numbers();
operation();
```

```
for(int i=0;i<j;i++)
{
fscanf(ptr,"%d\n",&arr1[i]);
}
for(int i=0;i<j;i++)
{
arr2[i]=arr1[i];
}
clock_t start_selection=clock();
selectionsort(arr1,j);
clock_t end_selection = clock();
double currs=(double)(end_selection-start_selection)/CLOCKS_PER_SEC;
clock_t start_insertion=clock();
insertionsort(arr2,j);
clock_t end_insertion=clock();
double curri=(double)(end_insertion-start_insertion)/CLOCKS_PER_SEC;
printf("\n%d\t%f\t%f",j,currs,curri);
} }
int main()
{
generate_numbers();
operation();
return 0; }
```

Observation:



Selection Sort-BLUE
Insertion Sort- RED

From the above graph, it is very clear that insertion sort takes less time to sort the same configuration of elements as sorted by selection sort. The difference in the time taken by these two algorithms to complete the task is negligible initially, when the number of elements is less; however, the difference is significant as the number of elements increases to about 35000 to 40000.

Conclusion:

By performing this experiment, I was able to understand the process of determining the time an algorithm implementation takesto perform the required task. Using this, I was able to compare insertion and selection sort algorithms which led me to the conclusion that insertion sort is generally a faster sorting algorithm than selection sort.