SARDAR PATEL INSTITUTE OF TECHNOLOGY

DAA EXPERIMENT 1-B

Name	OM CHANDRA
UID no.	2021700014
Experiment No.	1-B

AIM:	Sorting Methods
Program 1	
PROBLEM STATEMENT :	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.

ALGORITHM THEORY:	Selection sort is a simple sorting algorithm that works by repeatedly selecting the minimum element from the unsorted portion of the list and swapping it with the first

element of the unsorted portion. The process is repeated until the entire list is sorted.
Insertion sort is another simple sorting algorithm that works by maintaining a sorted portion of the list and inserting each subsequent element in the correct position in the sorted portion. The process is repeated until the entire list is sorted.
Both selection sort and insertion sort have a time complexity of O(n^2) in the worst-case scenario, making them inefficient for large lists. However, they are simple to understand and implement, making them useful for small lists or as building blocks for more efficient algorithms.

Selection sort algorithm:

1. Start with an unsorted list of elements.
2. Select the minimum element from the list.
3. Swap the minimum element with the first element of the list.
4. Repeat steps 2 and 3 for the remaining unsorted portion of the list.
5. Repeat steps 2 to 4 until the entire list is sorted.

Insertion sort algorithm:

1. Start with an unsorted list of elements.
2. Consider the first element as a sorted portion of the list.
3. For each subsequent element, compare it with the elements in the sorted portion of the list.
4. If the element is smaller than any of the elements in

	the sorted portion, insert it in the correct position. 5. Repeat steps 3 and 4 for each element in the unsorted portion of the list. 6. Repeat steps 3 to 5 until the entire list is sorted.

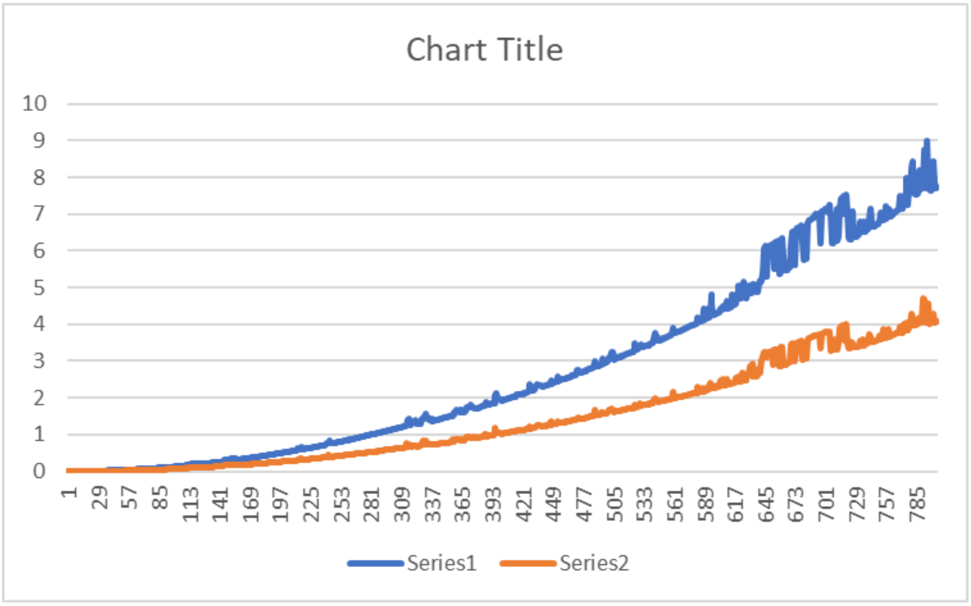
PROGRAM:	See code below

```
#include <stdio.h>
#include<time.h> #include<stdlib.h>
void selection_sort(int arr[],int size){

for(int i=0;i<size-1;i++){
int min_idx=i;     for(int
j=i+1;j<size;j++){
if(arr[j]<arr[min_idx]){
min_idx=i;
        }
    }
    int temp=arr[min_idx];
arr[min_idx]=arr[i];     arr[i]=temp;
}

}


void insertion_sort(int arr[],int size){

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

```
        }
        arr[j + 1] = key;
    }

}


int       main(void)       {
clock_t  a,b,c;
    int arr[100000],arr2[100000];
    double          d,e;
FILE *fptr;
  int num;


  fptr = fopen("integers", "w");

  if (fptr != NULL) {
    printf("File created successfully!\n");
  }
else {
    printf("Failed to create the file.\n");

    return -1;
  }


int i=0;   while
(i!=100000) {
num=rand()%100000;
if (num != -1) {
      putw(num, fptr);
    }
else {
```

```
      break;
    }
    ++i;
  }


  fclose(fptr);


  fptr = fopen("integers", "r");

i=0;
  while ( (num = getw(fptr)) != EOF ) {
arr[i]=num;      arr2[i]=num;
     i++;
  }

  printf("\nEnd of file.\n");


  fclose(fptr);


  for(int j=100;j<=100000;j+=100){
a=clock();       selection_sort(arr,j);
b=clock();

      d=(double)(b-a)/CLOCKS_PER_SEC;
printf("%f\n",d);        c=clock();
printf("\t");
       insertion_sort(arr2,j);
       e=(double)(c-b)/CLOCKS_PER_SEC;
       printf("%f\n",e);


  }
     d=(double)(b-a)/CLOCKS_PER_SEC;
printf("%f",d);

  return 0;
}
```

RESULT:
Blue: Selection Sort
Red : Insertion Sort



Observations:

Both selection sort and insertion sort have a worst-case time complexity of O(n^2), where n is the number of elements in the list. This means that as the number of elements in the list increases, the time required to sort the list will increase quadratically.
However, the difference between insertion sort and selection sort lies in their average-case time complexity. Insertion sort has an average-case time complexity of O(n^2), which is the same as its worst-case time complexity. However, selection sort has an average-case time complexity of O(n^2 / 2), which is slightly better than its worst-case time complexity.
In practice, the difference between the average-case time complexity of insertion sort and selection sort is not significant, and both algorithms are generally considered to be inefficient for large lists. More efficient sorting algorithms, such as quicksort or merge sort, should be used in these cases.

CONCLUSION:	In this experiment I was able to understand basic sorting algorithms (insertion sort and selection sort) and their time and space complexities by plotting their execution time and analysing it. Both insertion sort and selection sort have time complexity O(n^2) which is not efficient enough.