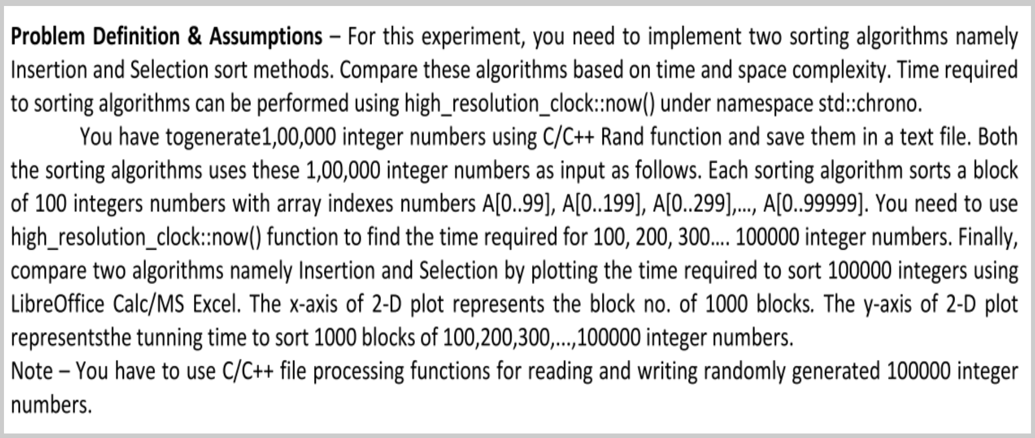
NAME:	Pratham Jain
UID:	2021300051- COMPS A (C-batch)
SUBJECT	DAA
EXPERIMENT NO :	1-B
DATE OF PERFORMANCE	06/02/23
DATE OF SUBMISSION	13/02/23
AIM:	Experiment on finding the running time of an algorithm.

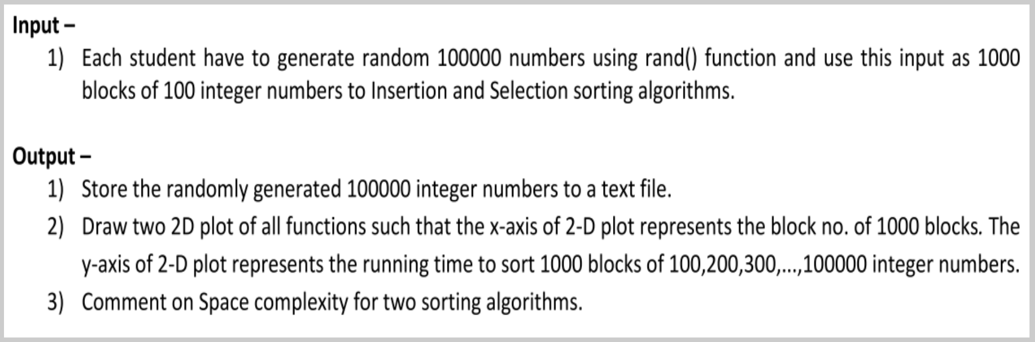
PROBLEM STATEMENT 1:



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.



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.

THEORY

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.

	Insertion Sort	**Selection Sort**
1.	Inserts the value in the presorted array to sort the set of values in the array.	Finds the minimum / maximum number from the list and sort it in ascending / descending order.
2.	It is a stable sorting algorithm.	It is an unstable sorting algorithm.
3.	The best-case time complexity is Ω(N) when the array is already in ascending order. It have Θ(N^2) in worst case and average case.	For best case, worst case and average selection sort have complexity Θ(N^2).
4.	The number of comparison operations performed in this sorting algorithm is less than the swapping performed.	The number of comparison operations performed in this sorting algorithm is more than the swapping performed.
5.	It is more efficient than the Selection sort.	It is less efficient than the Insertion sort.
6.	Here the element is known beforehand, and we search for the correct position to place them.	The location where to put the element is previously known we search for the element to insert at that position.
7.	The insertion sort is used when: • The array is has a small number of elements • There are only a few elements left to be sorted	The selection sort is used when • A small list is to be sorted • The cost of swapping does not matter • Checking of all the elements is compulsory • Cost of writing to memory matters like in flash memory (number of Swaps is O(n) as compared to O(n2) of bubble sort)
8.	The insertion sort is Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is **O(kn)** when each element in the input is no more than **k** places away from its sort	Selection sort is an in-place comparison sorting algorithm

ALGORITHM

INSERTION SORT

To sort an array of size N in ascending order:

1. Iterate from arr[1] to arr[N] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

SELECTION SORT

1. Set Min to location 0 in Step 1.
2. Look for the smallest element on the list.
3. Replace the value at location Min with a different value.
4. Increase Min to point to the next element
5. Continue until the list is sorted.

PROGRAM:

SORTING PROGRAM :

```c
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

void insertionSort(int array[], int size) {
    for (int step = 1; step < size; step++) {
        int key = array[step];
        int j = step - 1;
        while (key < array[j] && j >= 0) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = key;
    }
}
```

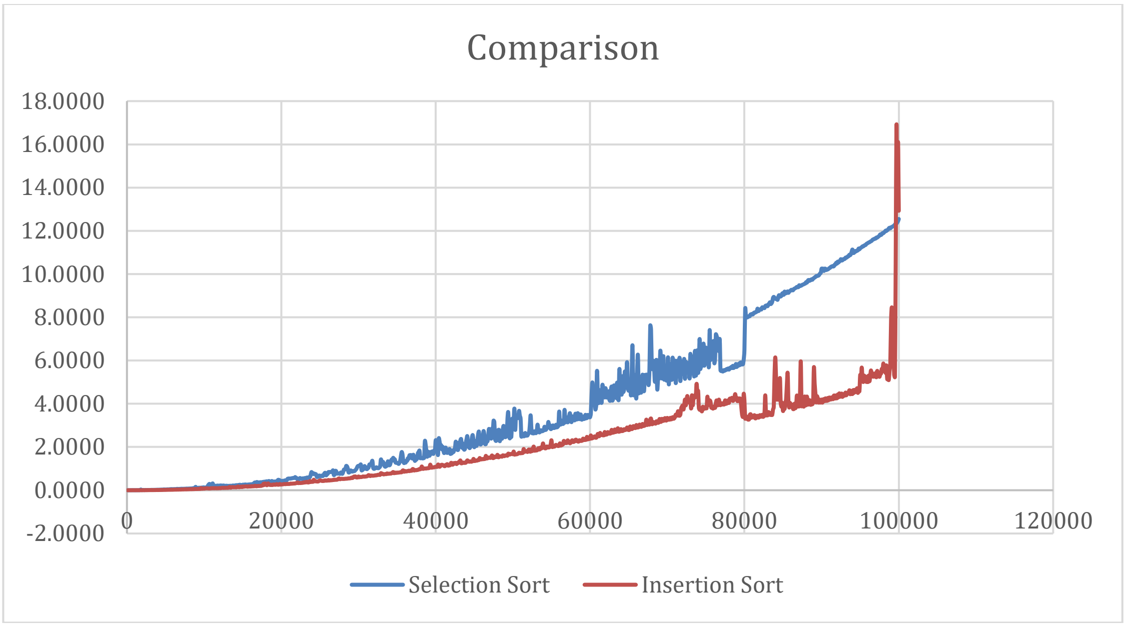
```c
void selectionSort(int arr[], int len){
    int minIndex, temp;
    for(int i=0; i<len; i++){
        minIndex = i;
        for(int j=i+1; j<len; j++){
            if(arr[j] < arr[minIndex]){
                minIndex = j;
            }
        }
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

/*  the same code was edited to generate both the values
    for selection and insertion sort. The were saved to a
    text file then copied and pasted to excel sheet to plot
    the graph.
*/

int main(){

    FILE *fptr, *sPtr;
    int index=99;
    int arrNums[100000];
    clock_t t;
    fptr = fopen("Random.txt", "r");
    sPtr = fopen("iTimes.txt", "w");
    for(int i=0; i<=999; i++){
        for(int j=0; j<=index; j++){
            fscanf(fptr, "%d", &arrNums[j]);
        }
        t = clock();
        insertionSort(arrNums, index+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(sPtr, "%lf\n", time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        index = index + 100;
        fseek(fptr, 0, SEEK_SET);
    }
    fclose(sPtr);
    fclose(fptr);
    return 0;
}
```

To create random numbers:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

int main(){

    FILE *fptr;
    fptr = fopen("Random.txt", "w");
    srand(time(NULL));
    int random;
    for(int i=1; i<=100000; i++){
        random = rand()%100000 + 1;
        fprintf(fptr, "%d ", random);
    }
    fclose(fptr);

    return 0;
}
```

RESULT:

Comparison between Selection and Insertion Sort

	
CONCLUSION:	Through this experiment I learned the algorithm, implementation and comparison of Insertion sort & Selection sort. <u>Insertion sort</u> :The best-case time complexity is Ω(N) It have Θ(N2) in worst case and average case. <u>Selection sort:</u> For best case, worst case and average selection sort have complexity Θ(N2).