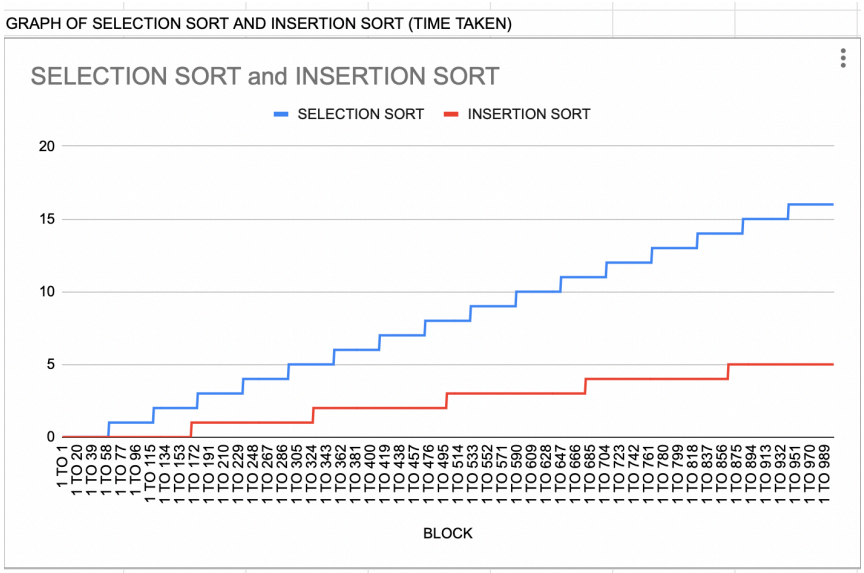
NAME:	PRIYANKA SUNIL KADAM
UID NO.:	2021300052
EXPERIMENT NO.:	2

AIM:	Experiment on finding the running time of an algorithm 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.
PROGRAM:	#include <bits/stdc++.h>
	#include <fstream>
	#include <time.h>
	#include <chrono>
	using namespace std;
	volatile int sink;
	vector<int>selectionSort(vector<vector<int>>&a);
	vector<int>insertionSort(vector<vector<int>>&a);
	void printTime(vector<int>timer_selection, vector<int>timer_insertion);
	int main()
	{
	clock_t begin, end;
	vector<vector<int>>a(1000,vector<int>(100,0));
	int offset=0;
	srand(0);
	begin = clock();
	for (int i=0; i<1000; ++i) {
	for(int j=0;j<100;j++){
	a[i][j]=(rand()%100000+offset);
	}
	offset+=100;
	}

```
    for (int i = 0; i < 1000; I++){
      cout<<"BLOCK "<<i+1<<"\n";
        for (int j = 0; j < 100; j++){
            cout<<a[i][j]<<“\t";}
      cout<<“\n";}
    ofstream fout("Myfile.txt");
    if(fout.is_open()){
        for(int i = 0;i<a.size(); i++){
            fout<<"BLOCK\t"<<i+1<<endl;
            for(int j=0;j<a[0].size();j++){
                fout<<a[i][j]<<"\t";
            }
            fout<<endl;
        }
        cout << "Success!" << endl;
    }
    else {
        cout << "File could not be opened." << endl;
    }
    printTime(insertionSort(a),selectionSort(a));
    fout.close();
    end=clock();
    double time_to_generate_print = ((double)end - begin) / CLOCKS_PER_SEC;
    cout<<"Time in generating and printing :"<<time_to_generate_print<<endl;
    return 0;
}
vector<int> insertionSort(vector<vector<int>>& a){
```

```
    clock_t begin, end;
    begin = clock();
    vector<int>timer;
    int n=a.size();
for (int k = 0; k < 1000; k++)
{
     for (int j = 0; j < n; j++)
        {
            int i, key, m;
            for (i = 1; i < n; i++)
            {
                key = a[k][i];
                m = i - 1;
                while (m >= 0 && a[k][m] > key)
                {
                    a[k][m + 1] = a[k][m];
                    m = m - 1;
                }
                a[k][m + 1] = key;
            }
        }
    end = clock();
    double timeinsert = ((double)end - begin) / CLOCKS_PER_SEC;
    timer.push_back(timeinsert);
}
    return timer;
}
vector<int>selectionSort(vector<vector<int>>&a){
```

```
    clock_t begin, end;
    begin = clock();
    int n=a.size();
    vector<int>timer;
    for (int k = 0; k < 1000; k++){
        int i, j, min_idx;
        for (i = 0; i < n - 1; i++)
        {  // Find the minimum element in unsorted array
            min_idx = i;
            for (j = i + 1; j < n - 1; j++)
                if (a[k][j] < a[k][min_idx])
                    min_idx = j;
            // Swap the found minimum element with the first element
            if (min_idx != i)
                swap(a[k][min_idx], a[k][i]);
        }
        end = clock();
        double timeselection = ((double)end - begin) / CLOCKS_PER_SEC;
        timer.push_back(timeselection);
    }
    return timer;
}
void printTime(vector<int>timer_selection, vector<int>timer_insertion){
    cout<<"BLOCK \tTIME TO SELECTION SORT \tTIME TO INSERTION SORT"<<"\n";
    for (int i = 0; i < 1000; i++){
       cout<<"1 TO "<<(i+1)<<"\t"<<timer_selection[i]<<"\t"<<timer_insertion[i]<<endl;
    }
```

	timer_selection.clear();
	timer_insertion.clear();
	}
RESULT:	GRAPH OF SELECTION SORT AND INSERTION SORT (TIME TAKEN)  1. TIME GRAPH: https://docs.google.com/spreadsheets/d/13jYgiFegM2D6TFBZQ-dUiZVnZgs1A0qitdqtO99X6b4/edit?usp=sharing 2. DATA VALUES: https://docs.google.com/spreadsheets/d/1i_51CCrRm3NI5iyUvIS8bfjZZ8PZVXoT_tKH17M7-gE/edit?usp=sharing
CONCLUSION:	The speed of insertion sort and selection sort can vary based on several factors, such as the size of the input array, the order of the elements in the array, and the specific implementation of the algorithm. However, in general, insertion sort tends to be faster than selection sort. This is because insertion sort has the advantage of sorting elements in place and it is more efficient when the array is partially sorted. On the other hand, selection sort has to repeatedly find the minimum element in the unsorted portion of the array, which can be time-consuming for larger arrays. Both insertion sort and selection sort have a space complexity of O(1), meaning that the algorithms use a constant amount of extra memory regardless of the size of the input array. This is because both algorithms are considered to be in-place sorting algorithms, meaning that they sort the elements within the original array and do not require any additional memory to store the sorted result.