****

**NAME: -** Darji Akshatkumar Hiteshbhai

**RollNo: -** 23MIT3301

**Branch: -** M.tech-CSE**(Data Science)**

**Subject: -** Complexity Theory & Algorithms

**Practical-1**

**Aim:** Perform Selection Sort, Quick Sort, Bubble Sort and Insertion Sort for the input size 10000, 50000 and 100000 for Ascending, Descending & Random order array. Plot the chart of the output data and do the analysis which algorithm is best and justify your reason.

**Code-**

    #include <bits/stdc++.h>

    using namespace std;

    using namespace std::chrono;

    void random\_array(int \*arr, int n)

    {

        for (int i = 0; i < n; i++)

        {

            arr[i] = rand() % n;

        }

    }

    // For Quick Sort

    void random(vector<int> &array, int n)

    {

        for (int i = 0; i < n; i++)

        {

            array[i] = rand() % n;

        }

    }

    void acending\_array(int \*arr, int n)

    {

        for (int i = 0; i < n; i++)

        {

            arr[i] = i;

        }

    }

    // For Quick Sort

    void acending(vector<int> &array, int n)

    {

        for (int i = 0; i < n; i++)

        {

            array[i] = i;

        }

    }

    void decending\_array(int \*arr, int n)

    {

        for (int i = 0; i < n; i++)

        {

            arr[i] = n - i - 1;

        }

    }

    // For Quick Sort

    void decending(vector<int> &array, int n)

    {

        for (int i = 0; i < n; i++)

        {

            array[i] = n - i - 1;

        }

    }

    // BubbleSort Algorithm

    void bubblesort(int \*arr, int n)

    {

        for (int i = n - 1; i >= 0; i--)

        {

            for (int j = 0; j <= i - 1; j++)

            {

                if (arr[j] > arr[j + 1])

                {

                    int temp = arr[j + 1];

                    arr[j + 1] = arr[j];

                    arr[j] = temp;

                }

            }

        }

    }

    void bubblesortcode\_a(int \*arr, int n)

    {

        acending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        bubblesort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Bubble sort for " << n << " elements in "

            << "Acending Order is: " << total\_time.count() << endl;

    }

    void bubblesortcode\_d(int \*arr, int n)

    {

        decending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        bubblesort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Bubble sort for " << n << " elements in "

            << "Decending Order is: " << total\_time.count() << endl;

    }

    void bubblesortcode\_r(int \*arr, int n)

    {

        random\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        bubblesort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Bubble sort for " << n << " elements in "

            << "Random Order is: " << total\_time.count() << endl;

    }

    // InsertionSort Algorithm

    void insertionsort(int \*arr, int n)

    {

        for (int i = 0; i < n - 1; i++)

        {

            int j = i;

            while (j > 0 && arr[j - 1] > arr[j])

            {

                int temp = arr[j - 1];

                arr[j - 1] = arr[j];

                arr[j] = temp;

                j--;

            }

        }

    }

    void insertionsort\_a(int \*arr, int n)

    {

        acending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        insertionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Insertion sort for " << n << " elements in "

            << "Acending Order is: " << total\_time.count() << endl;

    }

    void insertionsort\_d(int \*arr, int n)

    {

        decending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        insertionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Insertion sort for " << n << " elements in "

            << "Decending Order is: " << total\_time.count() << endl;

    }

    void insertionsort\_r(int \*arr, int n)

    {

        random\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        insertionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Insertion sort for " << n << " elements in "

            << "Random Order is: " << total\_time.count() << endl;

    }

    // Selection Sort Algorithm

    void selectionsort(int \*arr, int n)

    {

        for (int i = 0; i <= n - 2; i++)

        {

            int min = i;

            for (int j = i; j <= n - 1; j++)

            {

                if (arr[j] < arr[min])

                {

                    min = j;

                }

            }

            int temp = arr[min];

            arr[min] = arr[i];

            arr[i] = temp;

        }

    }

    void selectionsort\_a(int \*arr, int n)

    {

        acending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        selectionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Selection sort for " << n << " elements in "

            << "Acending Order is: " << total\_time.count() << endl;

    }

    void selectionsort\_d(int \*arr, int n)

    {

        decending\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        selectionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Selection sort for " << n << " elements in "

            << "Decending Order is: " << total\_time.count() << endl;

    }

    void selectionsort\_r(int \*arr, int n)

    {

        random\_array(arr, n);

        auto start = high\_resolution\_clock::now();

        selectionsort(arr, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Selection sort for " << n << " elements in "

            << "Random Order is: " << total\_time.count() << endl;

    }

    // Quicksort Algorithm

    int part(vector<int> &arr, int low, int high)

    {

        int midIndex = low + (high - low) / 2;

        int pivot = arr[midIndex];

        int i = low;

        int j = high;

        while (i < j)

        {

            while (arr[i] <= pivot && i <= high - 1)

            {

                i++;

            }

            while (arr[j] > pivot && j >= low + 1)

            {

                j--;

            }

            if (i < j)

            {

                swap(arr[i], arr[j]);

            }

        }

        swap(arr[midIndex], arr[j]);

        return j;

    }

    void qs(vector<int> &array, int low, int high)

    {

        if (low < high)

        {

            int pIndex = part(array, low, high);

            qs(array, low, pIndex - 1);

            qs(array, pIndex + 1, high);

        }

    }

    vector<int> quicksort(vector<int> array, int n)

    {

        qs(array, 0, array.size() - 1);

        return array;

    }

    void quicksort\_a(vector<int> &array, int n)

    {

        acending(array, n);

        auto start = high\_resolution\_clock::now();

        quicksort(array, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Quick sort for " << n << " elements in "

            << "Acending Order is: " << total\_time.count() << endl;

    }

    void quicksort\_d(vector<int> &array, int n)

    {

        decending(array, n);

        auto start = high\_resolution\_clock::now();

        quicksort(array, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Quick sort for " << n << " elements in "

            << "Decending Order is: " << total\_time.count() << endl;

    }

    void quicksort\_r(vector<int> &array, int n)

    {

        random(array, n);

        auto start = high\_resolution\_clock::now();

        quicksort(array, n);

        auto end = high\_resolution\_clock::now();

        duration<double> total\_time = end - start;

        cout << endl

            << "Total Time taken by Quick sort for " << n << " elements in "

            << "Random Order is: " << total\_time.count() << endl;

    }

    // Main Function

    int main()

    {

        int n;

        cout << "Enter the Array Size: ";

        cin >> n;

        int \*arr = new int[n];

        //For QUickSort

        vector<int> array(n);

        for (int i = 1; i <= 3; i++)

        {

            if (i == 1)

            {

                bubblesortcode\_a(arr, n);

                insertionsort\_a(arr, n);

                selectionsort\_a(arr, n);

                quicksort\_a(array, n);

            }

            else if (i == 2)

            {

                bubblesortcode\_d(arr, n);

                insertionsort\_d(arr, n);

                selectionsort\_d(arr, n);

                quicksort\_d(array, n);

            }

            else if (i == 3)

            {

                bubblesortcode\_r(arr, n);

                insertionsort\_r(arr, n);

                selectionsort\_r(arr, n);

                quicksort\_r(array, n);

            }

        }

        return 0;

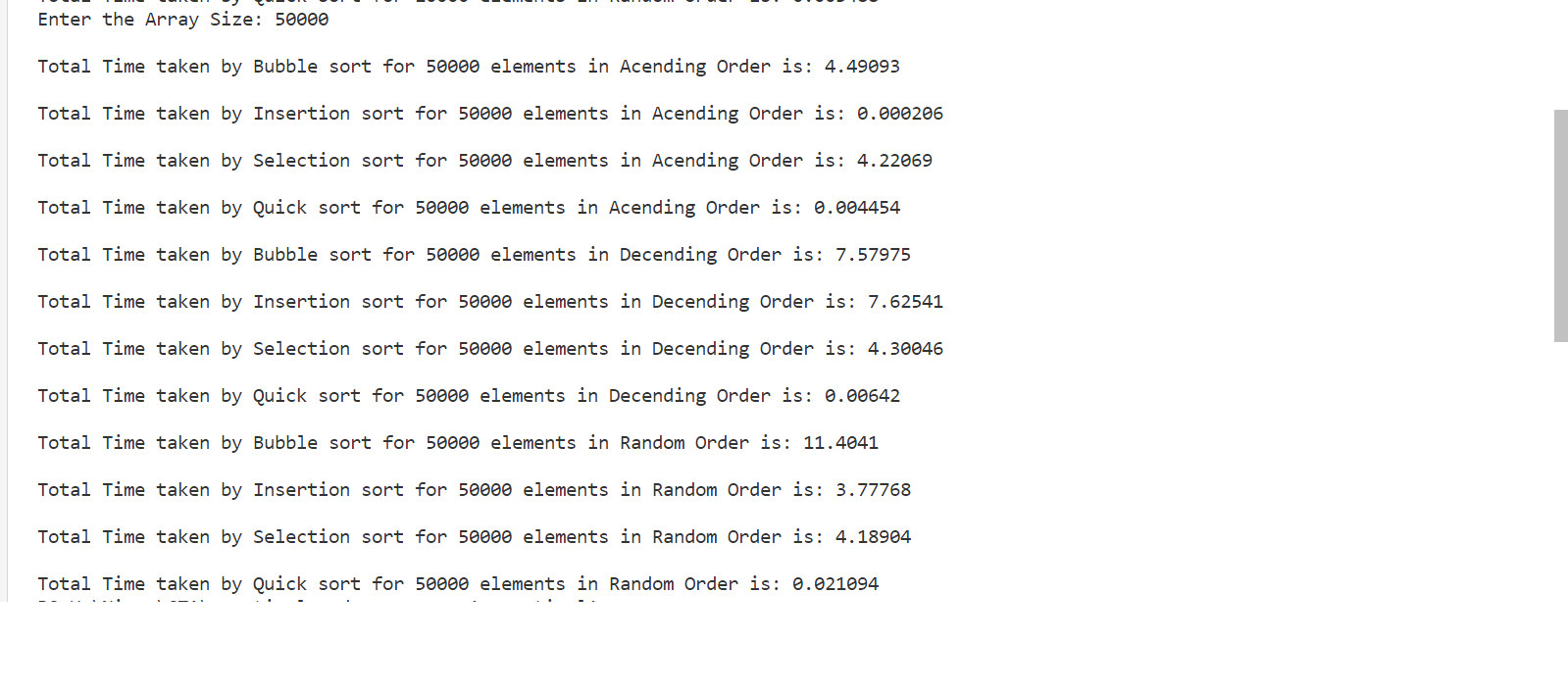
    }

* **Output**

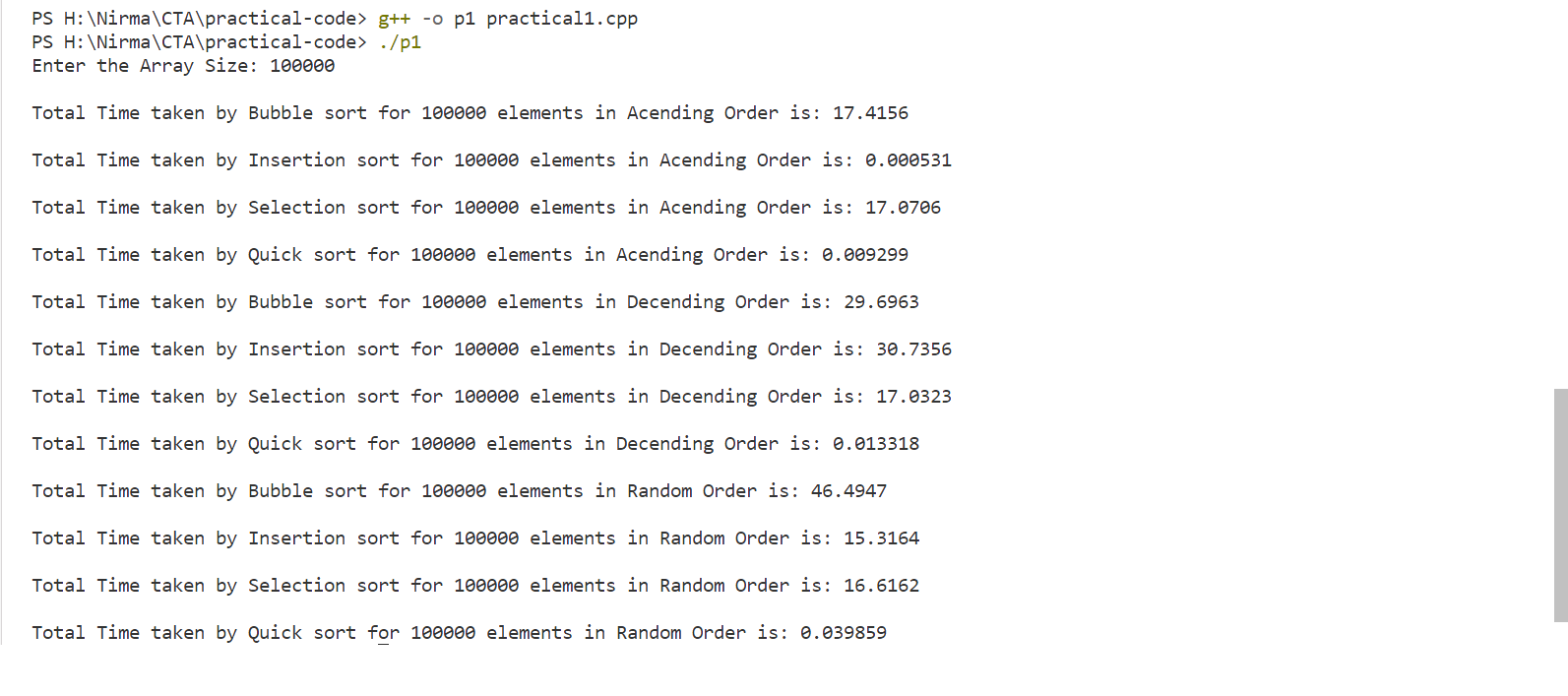
For array size = 10000



For array size = 50000



For array size = 100000



* **Graphs & Output Data**

Here are the three tables that describes the output generated by the above code for ascending, descending and random array for size 10000, 50000 & 100000.

**ASCENDING ORDER:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BubbleSort** | **InsertionSort** | **SelectionSort** | **QuickSort** |
| **10000** | **0.1866** | **0.0000041** | **0.1733** | **0.00112** |
| **50000** | **4.4903** | **0.000206** | **4.2206** | **0.004454** |
| **100000** | **17.4156** | **0.000531** | **17.07** | **0.009299** |

* In above graph x-axis contains the time in second, y-axis contains the sorting algorithms and the graph compares the time taken by sorting algorithms for 10000, 50000 & 100000 array size in **ascending order** only.

**DESCENDING ORDER:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BubbleSort** | **InsertionSort** | **SelectionSort** | **QuickSort** |
| **10000** | **0.3092** | **0.2915** | **0.1736** | **0.00118** |
| **50000** | **7.5797** | **7.6254** | **4.3004** | **0.00642** |
| **100000** | **29.6963** | **30.7356** | **17.0323** | **0.01331** |

* In above graph x-axis contains the time in second, y-axis contains the sorting algorithms and the graph compares the time taken by sorting algorithms for 10000, 50000 & 100000 array size in **descending order** only.

**RANDOM ORDER:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BubbleSort** | **InsertionSort** | **SelectionSort** | **QuickSort** |
| **10000** | **0.3888** | **0.1604** | **0.1744** | **0.00348** |
| **50000** | **11.4041** | **3.7777** | **4.189** | **0.021** |
| **100000** | **46.4967** | **15.3164** | **16.6162** | **0.0398** |

* In above graph x-axis contains the time in second, y-axis contains the sorting algorithms and the graph compares the time taken by sorting algorithms for 10000, 50000 & 100000 array size in **random order** only.
* **Analysis**

The first thing I notice that the running time of all four algorithm increases as the input size increases. The reason behind this is as the array size is bigger the number of elements in the array are more so the number of comparisons and swaps that the algorithm need to sort an array is increases with the size of input.

**BUBBLE SORT**

The bubble sort algorithm works by repeatedly comparing the neighboring elements in the array and swap the largest element at the end of an array. In our case the worst-case time complexity is O(n^2), the reason behind this is in the worst-case array is in the **descending order** so all the elements in the array are in reverse order, so the bubble sort algorithm will have to make n comparisons and n-1 swaps for each n elements in the array. So, the bubble sort takes more time to sort an array in descending order.

In **ascending order** bubble sort require only comparisons there is no swap required because in ascending order array the array elements are already in sorted manner so there is no need to swap the elements so it takes less time than the descending order array.

In **Random order,** the elements are not placed in any specific order, so generally bubble sort is worse for random order array, as it needs to perform continuous number of swaps and comparisons to rearrange the array elements and sometimes it compares the element twice due to random array, so it takes more time than the descending order array.

So, from our data also we can clarify that for the 10000 elements the bubble sort takes 0.1866s in ascending order, 0.3092s in descending order and 0.3888s in the random order.

**INSERTION SORT**

Insertion sort performs comparatively faster then the bubble sort in all cases because in insertion sort it compares only its adjacent elements it consider the first element as already sorted element and from index-1 it checks its previous elements and put the smaller element at the first position of an array, so in the ascending order array is already sorted so it takes less amount of time, in descending order it takes more amount of time because of the array is in reverse order so it needs more comparisons and in random order it is dependent on the randomness.

Insertion sort is more efficient when array is already sorted. Because insertion sort is stable algorithm.

**SELECTION SORT**

Selection sort behaves same as the bubble sort. Because it considers first element as a minimum and then check every element in the array and find the minimum which is less than the default minimum so the number of comparisons and swaps are similar to the bubble sort because it checks all the elements in the array and then swaps.

**QUICK SORT**

Quick sort is the fastest algorithm among the selection, bubble and insertion sort. In some **cases,** **insertion sort** is better than the **quick sort**, when the array is smaller and partially sorted in such cases insertion sort is better than the quick sort because the insertion sort is stable algorithm means it does not swap the equal elements so it takes less time than quick sort while quicksort recursively solves the problem so it takes slightly more time than the insertion sort for the smaller size array.

But in other cases, and the array size increases quick sort is best searching algorithm among all others. Because quick sort is a divide and conquer technique, it divides the larger array into smaller subarrays and recursively sort that array so that’s the reason that quick sort is better algorithm then other for the larger arrays. And other algorithms not perform well because of the other algorithms like bubble, selection and insertion are sequential sorting algorithms means sequentially iterations performs while quick sort recursively sort the array.

Pivot element selection is also matters for sorting the array.