

**Name: M. Anas Baig**

**Subject: Data Structure & Algorithms**

**Enrollment No.: 01-134152-037**

**Section: BS(CS)-4A**

**BAHRIA UNIVERSITY ISLAMABAD, PAKISTAN**

### **Assignment # 3**

#### **Data Structures and Algorithms**

**May 9<sup>th</sup>, 2017**

### **SOLUTION:**

#### **C++ Concepts Used:**

Following C++ concepts are used in solving the problem:

- **Data Structure** (dynamic array).
- **Sorting Algorithms** (bubble sort, selection sort, insertion sort, merge sort, quick sort).

#### **Functionalities\Features of Program:**

Following functionalities are provided in the program:

Program: -

- Could **sort Array of any size**. (dynamic memory allocation to array)
- Could **compute the Running Time of Sorting Algorithms**.
- Could **compute the Running Time of Sorting Algorithms for Best Case Scenario and Worst Case Scenario** so we can determine and use respective sorting algorithms according to our requirements.

**CODE SOLUTION:****Source.cpp File:**

```
1. #include "conio.h"
2. #include "ctime"
3. #include <iostream>
4. using namespace std;
5.
6. void descendingSelectionSort(int arr[], int size) //to sort elements in descending order
7. {
8.     for( int i=0; i<size-1; i++ )
9.     {
10.         int maxPos = i;
11.         int max = arr[i];
12.
13.         for( int j=i+1; j<size; j++ )
14.         {
15.             if( arr[j]>max )
16.             {
17.                 maxPos = j;
18.                 max = arr[maxPos];
19.             }
20.         }
21.         arr[maxPos] = arr[i];
22.         arr[i] = max;
23.     }
24. }
25.
26. void selectionSort(int arr[], int size)
27. {
28.     for( int i=0; i<size-1; i++ )
29.     {
30.         int smallPos = i;
31.         int smallest = arr[i];
32.
33.         for( int j=i+1; j<size; j++ )
34.         {
35.             if( arr[j]<smallest )
36.             {
37.                 smallPos = j;
38.                 smallest = arr[smallPos];
39.             }
40.         }
41.         arr[smallPos] = arr[i];
42.         arr[i] = smallest;
43.     }
44. }
45.
46. void bubbleSort(int arr[], int size)
47. {
48.     bool flag = true;
49.     for( int i=0; i<(size-1) && flag==true; i++ )
50.     {
51.         flag = false;
52.         for( int j=0; j<(size-i-1); j++ ) //on each pass compares from array start
```

```
53.     {
54.         if( arr[j] > arr[j+1] )
55.         {
56.             int temp = arr[j];
57.             arr[j] = arr[j+1];
58.             arr[j+1] = temp;
59.             flag = true;
60.         }
61.     }
62. }
63. }
64.
65. void insertionSort(int arr[], int size)
66. {
67.     for( int i=1; i<(size); i++ )
68.     {
69.         int temp = arr[i];
70.         int j = i-1;
71.
72.         while( temp < arr[j] && (j>=0) )
73.         {
74.             arr[ j+1 ] = arr[ j ]; //moves element forward
75.             j = j-1;
76.         }
77.
78.         arr[j+1] = temp;
79.     }
80. }
81.
82. void merge(int *a, int *b, int low, int pivot, int high)
83. {
84.     int h, i, j, k;
85.     h = low;
86.     i = low;
87.     j = (pivot+1);
88.
89.     while( ( h<=pivot ) && ( j<=high ) )
90.     {
91.         if(a[h]<=a[j])
92.         {
93.             b[i] = a[h];
94.             h++;
95.         }
96.         else
97.         {
98.             b[i]=a[j];
99.             j++;
100.        }
101.        i++;
102.    }
103.
104.    if(h>pivot)
105.    {
106.        for(k=j; k<=high; k++)
107.        {
108.            b[i]=a[k];
109.            i++;
110.        }
111.    }
112.    else
113.    {
```

```
114.         for(k=h; k<=pivot; k++)
115.         {
116.             b[i]=a[k];
117.             i++;
118.         }
119.     }
120.
121.     for(k=low; k<=high; k++)
122.     {
123.         a[k]=b[k];
124.     }
125. }
126.
127. void mergeSort(int *a, int*b, int low, int high)
128. {
129.     int pivot;
130.     if( low<high )
131.     {
132.         pivot=( ( low+high )/2 );
133.         mergeSort( a, b, low, pivot);
134.         mergeSort(a, b, (pivot+1), high);
135.         merge(a, b, low, pivot, high);
136.     }
137. }
138.
139. void split( int x[], int first, int last, int &pos )
140. {
141.     int pivot = x[first];
142.     int left = first;
143.     int right = last;
144.     while (left < right)
145.     {
146.         while( x[right] > pivot)
147.         {
148.             right--;
149.         }
150.         while( x[left] <= pivot && left < right )
151.         {
152.             left++;
153.         }
154.         if (left < right)
155.         {
156.             int temp;
157.             temp = x[left];
158.             x[left] = x[right];
159.             x[right] = temp;
160.         }
161.     }
162.     x[first] = x[right];
163.     x[right] = pivot;
164.     pos = right;
165. }
166.
167. void quickSort (int x[], int first, int last)
168. {
169.     int pos;
170.     if ( first < last-1)
171.     {
172.         split (x, first, last, pos);
173.         quickSort (x, first, (pos-1));
174.         quickSort (x, pos + 1, last);
```

```

175.     }
176. }

```

## Main1(): (To test and validate Sorting Algorithms)

```

1.  int main()
2.  {
3.      int n;
4.      cout<<"Enter Number:"<<endl;
5.      cin>>n;
6.      //NOTE: Below individual arrays are declred for each sorting algorithm and then ini
           tialized with same elements to keep consistency in comparing algorithms working with sa
           me elements of array
7.      int *a = new int [n]; //array for bubble sorting
8.      int *b = new int [n]; //array for selection sorting
9.      int *c = new int [n]; //array for insertion sorting
10.     int *d = new int [n]; //array for merge sorting
11.     int *e = new int [n]; //array for quick sorting
12.     int *x = new int [n]; //array for merge sorting
13.     for(int i=0; i<n; i++) //initializing each array with same random numbers
14.     {
15.         a[i] = b[i] = c[i] = d[i] = e[i] = (rand() % n);
16.     }
17.
18.     //=====
19.     //bubble sorting
20.     //=====
21.     cout<<endl;
22.     cout<<"Bubble Sort Result:"<<endl;
23.     cout<<"===== "<<endl;
24.     bubbleSort(a, n);
25.     for( int i=0; i<n; i++ )
26.     {
27.         cout<<a[i]<<" ";
28.     }
29.
30.     //=====
31.     //selection sorting
32.     //=====
33.     cout<<endl;
34.     cout<<endl;
35.     cout<<"Selection Sort Result:"<<endl;
36.     cout<<"===== "<<endl;
37.     selectionSort(b, n);
38.     for( int i=0; i<n; i++ )
39.     {
40.         cout<<b[i]<<" ";
41.     }
42.
43.
44.     //=====
45.     //insertion sorting
46.     //=====

```

```
47.     cout<<endl;
48.     cout<<endl;
49.     cout<<"Insertion Sort Result:"<<endl;
50.     cout<<"====="<<endl;
51.     insertionSort(c, n);
52.     for( int i=0; i<n; i++ )
53.     {
54.         cout<<c[i]<<" ";
55.     }
56.
57.     //=====
58.     //merge sorting
59.     //=====
60.     cout<<endl;
61.     cout<<endl;
62.     cout<<"Merge Sort Result:"<<endl;
63.     cout<<"====="<<endl;
64.     mergeSort(d, x, 0, n-1);
65.     for( int i=0; i<n; i++ )
66.     {
67.         cout<<d[i]<<" ";
68.     }
69.
70.     //=====
71.     //quick sorting
72.     //=====
73.     cout<<endl;
74.     cout<<endl;
75.     cout<<"Quick Sort Result:"<<endl;
76.     cout<<"====="<<endl;
77.     quickSort(e, 0, n-1);
78.     for( int i=0; i<n; i++ )
79.     {
80.         cout<<e[i]<<" ";
81.     }
82.
83.     //=====
84.     //descending selection sorting
85.     //=====
86.     cout<<endl;
87.     cout<<endl;
88.     cout<<"Descending Order Sort Result:"<<endl;
89.     cout<<"====="<<endl;
90.     descendingSelectionSort(e, n);
91.     for( int i=0; i<n; i++ )
92.     {
93.         cout<<e[i]<<" ";
94.     }
95.
96.     getch();
97. }
```

```

c:\users\muhammad anas baig\documents\visual studio 2010\Projects\Assignment3\Debug\Assig...
Enter Array Size:
10
Bubble Sort Result:
=====
0 1 2 4 4 4 7 8 8 9
Selection Sort Result:
=====
0 1 2 4 4 4 7 8 8 9
Insertion Sort Result:
=====
0 1 2 4 4 4 7 8 8 9
Merge Sort Result:
=====
0 1 2 4 4 4 7 8 8 9
Quick Sort Result:
=====
0 1 2 4 4 4 7 8 8 9
Descending Order Sort Result:
=====
9 8 8 7 4 4 4 2 1 0

```

## Main2(): (To compute running time of Sorting Algorithms)

```

1.  int main()
2.  {
3.      int n;
4.      cout<<"Enter Array Size:"<<endl;
5.      cin>>n;
6.      //NOTE: Below individual arrays are declared for each sorting algorithm and then ini
           tialized with same elements to keep consistency in comparing algorithms working with sa
           me elements of array
7.      int *a = new int [n]; //array for bubble sorting
8.      int *b = new int [n]; //array for selection sorting
9.      int *c = new int [n]; //array for insertion sorting
10.     int *d = new int [n]; //array for merge sorting
11.     int *e = new int [n]; //array for quick sorting
12.     int *x = new int [n]; //array for merge sorting
13.     for(int i=0; i<n; i++) //initializing each array with same random numbers
14.     {
15.         a[i] = b[i] = c[i] = d[i] = e[i] = (rand() % n);
16.     }
17.
18.     //=====
19.     //bubble sorting
20.     //=====
21.     cout<<endl;
22.     cout<<"Bubble Sort Result:"<<endl;
23.     cout<<"===== "<<endl;
24.     clock_t a_time;
25.     a_time = clock();
26.     bubbleSort(a, n);
27.     a_time = clock() - a_time;

```

```

28.     cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
29.
30.
31.     //=====
32.     //selection sorting
33.     //=====
34.     cout<<endl;
35.     cout<<endl;
36.     cout<<"Selection Sort Result:"<<endl;
37.     cout<<"===== "<<endl;
38.     a_time = clock();
39.     selectionSort(c, n);
40.     a_time = clock() - a_time;
41.     cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
42.
43.     //=====
44.     //insertion sorting
45.     //=====
46.     cout<<endl;
47.     cout<<endl;
48.     cout<<"Insertion Sort Result:"<<endl;
49.     cout<<"===== "<<endl;
50.     a_time = clock();
51.     insertionSort(b, n);
52.     a_time = clock() - a_time;
53.     cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
54.
55.     //=====
56.     //merge sorting
57.     //=====
58.     cout<<endl;
59.     cout<<endl;
60.     cout<<"Merge Sort Result:"<<endl;
61.     cout<<"===== "<<endl;
62.     a_time = clock();
63.     mergeSort(d, x, 0, n-1);
64.     a_time = clock() - a_time;
65.     cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
66.
67.     //=====
68.     //quick sorting
69.     //=====
70.     cout<<endl;
71.     cout<<endl;
72.     cout<<"Quick Sort Result:"<<endl;
73.     cout<<"===== "<<endl;
74.     a_time = clock();
75.     quickSort(e, 0, n-1);
76.     a_time = clock() - a_time;
77.     cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
78.
79.     getch();
80. }

```



```

c:\users\muhammad anas baig\documents\visual studio 2010\Projects\Assignment3\Debug\Assig...
Enter Array Size:
5000

Bubble Sort Result:
=====
Running Time: 0.155 Seconds.

Selection Sort Result:
=====
Running Time: 0.053 Seconds.

Insertion Sort Result:
=====
Running Time: 0.04 Seconds.

Merge Sort Result:
=====
Running Time: 0.002 Seconds.

Quick Sort Result:
=====
Running Time: 0.001 Seconds.

```

## Main3(): (To compute running time for Best and Worst Case Scenario of Sorting Algorithms)

```

1. int main()
2. {
3.     int n;
4.     cout<<"Enter Array Size:"<<endl;
5.     cin>>n;
6.     int *a = new int [n]; //dynamic array
7.     int *x = new int [n]; //dynamic array
8.     for(int i=0; i<n; i++) //initializing array with random numbers
9.     {
10.        a[i] = (rand() % n);
11.    }
12.
13.    //=====
14.    //bubble sorting
15.    //=====
16.    cout<<endl;
17.    cout<<"Bubble Sort Result:"<<endl;
18.    cout<<"===== "<<endl;
19.    quickSort(a, 0, n-1); //make array already sorted for best case check
20.    clock_t a_time = clock(); //initializing clock
21.    bubbleSort(a, n); //checking for best case as array is already sorted
22.    a_time = clock() - a_time; //calculating clock difference
23.    cout<<"Best Case Scenario Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds.
    "<<endl;
24.    descendingSelectionSort(a, n); //make array reversed for worst case check
25.    a_time = clock(); //initializing clock

```

```

26.     bubbleSort(a, n); //checking for worst case as array is reversely ordered
27.     a_time = clock() - a_time; //calculating clock difference
28.     cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds
    ."<<endl;
29.
30.     //=====
    =====
31.     //selection sorting
32.     //=====
    =====
33.     cout<<endl<<endl;
34.     cout<<"Selection Sort Result:"<<endl;
35.     cout<<"===== "<<endl;
36.     //array already sorted for best case check
37.     a_time = clock(); //initializing clock
38.     selectionSort(a, n); //checking for best case as array is already sorted
39.     a_time = clock() - a_time; //calculating clock difference
40.     cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds.
    "<<endl;
41.     descendingSelectionSort(a, n); //make array reversed for worst case check
42.     a_time = clock(); //initializing clock
43.     selectionSort(a, n); //checking for worst case as array is reversely ordered
44.     a_time = clock() - a_time; //calculating clock difference
45.     cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds
    ."<<endl;
46.
47.     //=====
    =====
48.     //insertion sorting
49.     //=====
    =====
50.     cout<<endl<<endl;
51.     cout<<"Insertion Sort Result:"<<endl;
52.     cout<<"===== "<<endl;
53.     //array already sorted for best case check
54.     a_time = clock(); //initializing clock
55.     insertionSort(a, n); //checking for best case as array is already sorted
56.     a_time = clock() - a_time; //calculating clock difference
57.     cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds.
    "<<endl;
58.     descendingSelectionSort(a, n); //make array reversed for worst case check
59.     a_time = clock(); //initializing clock
60.     insertionSort(a, n); //checking for worst case as array is reversely ordered
61.     a_time = clock() - a_time; //calculating clock difference
62.     cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds
    ."<<endl;
63.
64.     //=====
    =====
65.     //merge sorting
66.     //=====
    =====
67.     cout<<endl<<endl;
68.     cout<<"Merge Sort Result:"<<endl;
69.     cout<<"===== "<<endl;
70.     //array already sorted for best case check
71.     a_time = clock(); //initializing clock
72.     mergeSort(a, x, 0, n-1); //checking for best case as array is already sorted
73.     a_time = clock() - a_time; //calculating clock difference
74.     cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds.
    "<<endl;

```

```

75.     descendingSelectionSort(a, n); //make array reversed for worst case check
76.     a_time = clock(); //initializing clock
77.     mergeSort(a, x, 0, n-1); //checking for worst case as array is reversely ordered
78.     a_time = clock() - a_time; //calculating clock difference
79.     cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds
    ."<<endl;
80.
81.     //=====
    =====
82.     //quick sorting
83.     //=====
    =====
84.     cout<<endl<<endl;
85.     cout<<"Quick Sort Result:"<<endl;
86.     cout<<"===== "<<endl;
87.     //array already sorted for best case check
88.     a_time = clock(); //initializing clock
89.     quickSort(a, 0, n-1); //checking for best case as array is already sorted
90.     a_time = clock() - a_time; //calculating clock difference
91.     cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds.
    "<<endl;
92.     descendingSelectionSort(a, n); //make array reversed for worst case check
93.     a_time = clock(); //initializing clock
94.     quickSort(a, 0, n-1); //checking for worst case as array is reversely ordered
95.     a_time = clock() - a_time; //calculating clock difference
96.     cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds
    ."<<endl;
97.
98.     getch();
99. }

```

```

c:\users\muhammad anas baig\documents\visual studio 2010\Projects\Assignment3\Debug\Assig...
Enter Array Size:
5000

Bubble Sort Result:
=====
Best Case Scenerio Running Time: 0 Seconds.
Worst Case Scenerio Running Time: 0.148 Seconds.

Selection Sort Result:
=====
Best Case Scenerio Running Time: 0.054 Seconds.
Worst Case Scenerio Running Time: 0.096 Seconds.

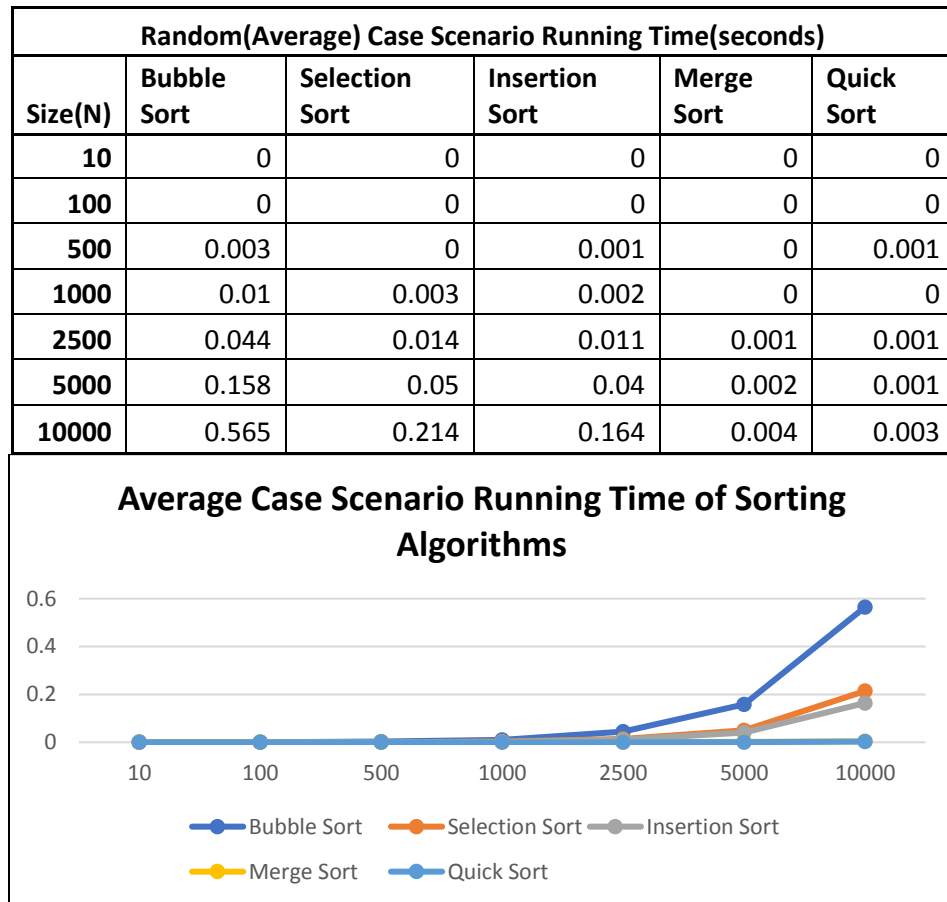
Insertion Sort Result:
=====
Best Case Scenerio Running Time: 0 Seconds.
Worst Case Scenerio Running Time: 0.076 Seconds.

Merge Sort Result:
=====
Best Case Scenerio Running Time: 0.002 Seconds.
Worst Case Scenerio Running Time: 0.002 Seconds.

Quick Sort Result:
=====
Best Case Scenerio Running Time: 0.025 Seconds.
Worst Case Scenerio Running Time: 0.03 Seconds.

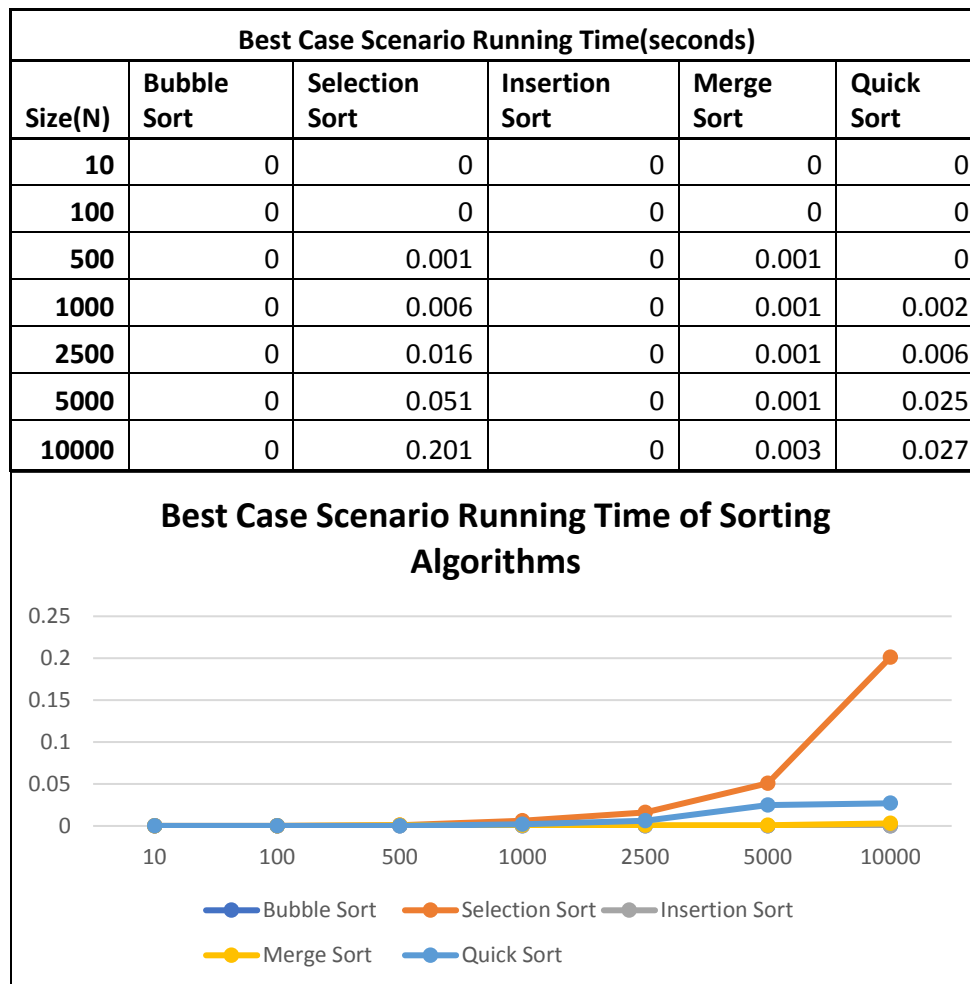
```

## GRAPHS:



### Conclusion:

As it can be seen from above graph and table that in random case scenario there is less difference between running times of sorting algorithms until 10,000 however at 10,000 Bubble Sort running time is increased gradually with respect to size.



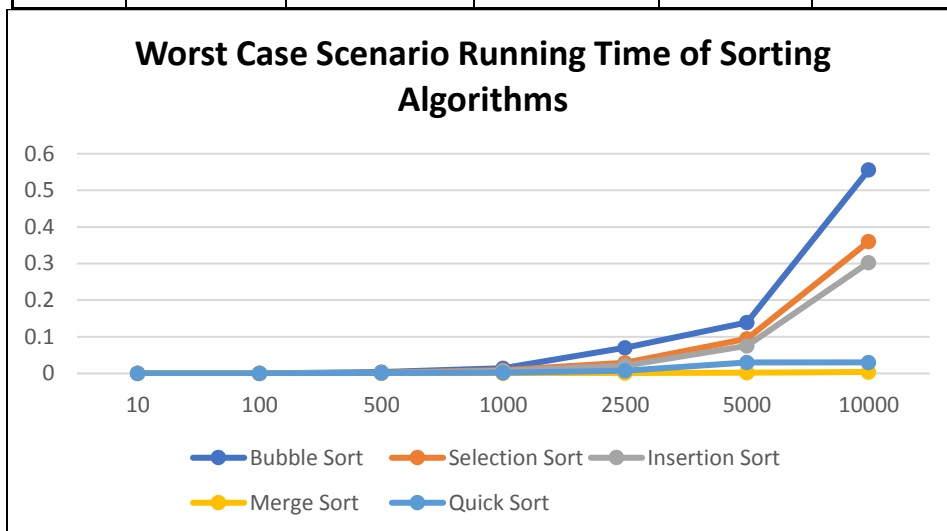

---

### Conclusion:

In best case scenario it can be seen from above graph and table that instead of already sorted array Selection Sort is still consuming much time due to which it could be inferred that Selection Sort works almost same in either Best Case Scenario or Worst Case Scenario and is slow.

---

Worst Case Scenario Running Time(seconds)					
Size(N)	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
10	0	0	0	0	0
100	0	0	0	0	0
500	0.004	0.002	0.001	0	0.001
1000	0.014	0.009	0.007	0	0.002
2500	0.07	0.029	0.021	0.001	0.007
5000	0.139	0.095	0.075	0.002	0.03
10000	0.556	0.36	0.303	0.004	0.0301



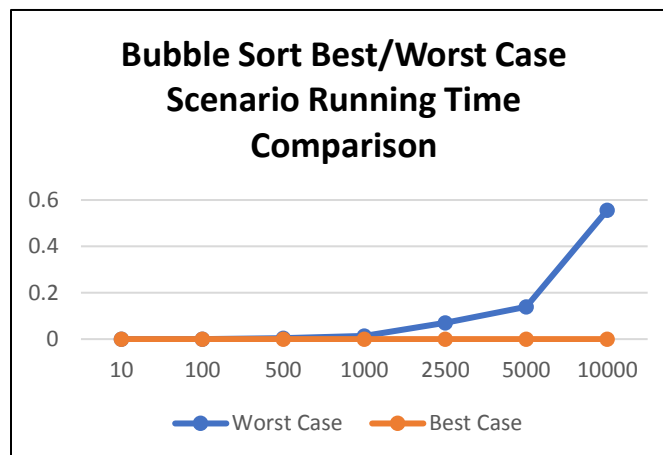

---

### Conclusion:

In worst case scenario it can be seen from above Graph and Table that Merge Sort and Quick Sort are very much efficient and fast however Bubble Sort is slow as compared to other sorting algorithms.

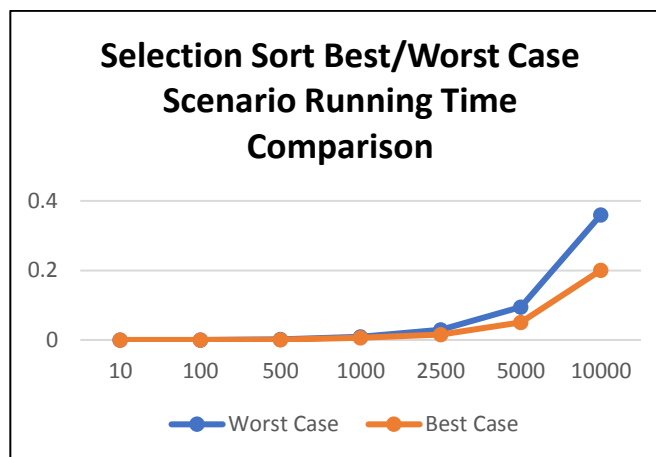
---

Bubble Sort Case Scenario Time Comparison		
Size(N)	Worst Case	Best Case
10	0	0
100	0	0
500	0.004	0
1000	0.014	0
2500	0.07	0
5000	0.139	0
10000	0.556	0

**Conclusion:**

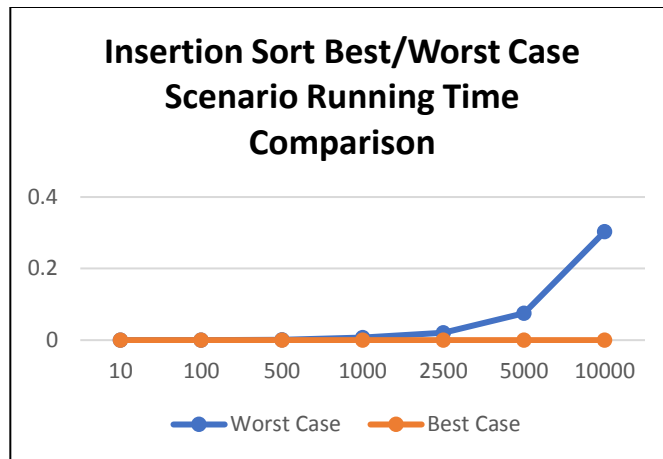
In Bubble Sort it could be seen from above graph and table that Bubble Sort works efficiently for Best Case Scenario.

Selection Sort Case Scenario Time Comparison		
Size(N)	Worst Case	Best Case
10	0	0
100	0	0
500	0.002	0.001
1000	0.009	0.006
2500	0.029	0.016
5000	0.095	0.051
10000	0.36	0.201

**Conclusion:**

In Selection Sort it can be seen from above Graph and Table that there is light difference between Worst Case Scenario and Best Case Scenario but it will affect dramatically when size of data will be too large.

Insertion Sort Case Scenario Time Comparison		
Size(N)	Worst Case	Best Case
10	0	0
100	0	0
500	0.001	0
1000	0.007	0
2500	0.021	0
5000	0.075	0
10000	0.303	0

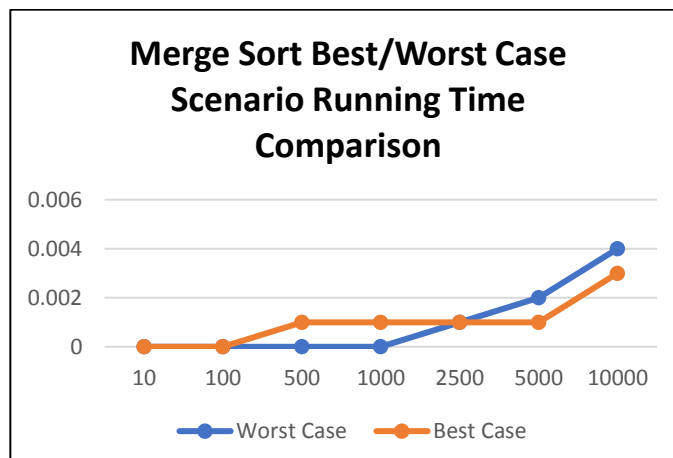


### Conclusion:

It can be seen from above Graph and Table that Insertion Sort had a great efficient effect in Best Case Scenario.

---

Merge Sort Case Scenario Time Comparison		
Size(N)	Worst Case	Best Case
10	0	0
100	0	0
500	0	0.001
1000	0	0.001
2500	0.001	0.001
5000	0.002	0.001
10000	0.004	0.003



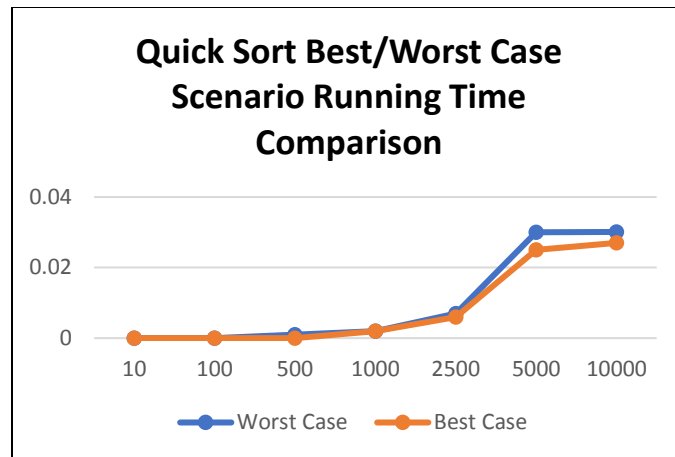
### Conclusion:

It can be seen from above Graph and Table that in Merge Sort Best Case Scenario and Worst Case Scenario had almost the same affect.

---



Quick Sort Case Scenario Time Comparison		
Size(N)	Worst Case	Best Case
10	0	0
100	0	0
500	0.001	0
1000	0.002	0.002
2500	0.007	0.006
5000	0.03	0.025
10000	0.0301	0.027

**Conclusion:**

It can be seen from above Graph and Table that Quick Sort got negligible difference in Best Case Scenario and Worst Case Scenario and is also the most efficient and less time consuming sorting algorithm.

---