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 9th, 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:

```cpp
#include "conio.h"
#include "ctime"
#include <iostream>
using namespace std;

void descendingSelectionSort(int arr[], int size) //to sort elements in descending order
{
    for( int i=0; i<size-1; i++ )
    {
        int maxPos = i;
        int max = arr[i];

        for( int j=i+1; j<size; j++ )
        {
            if( arr[j]>max )
            {
                maxPos = j;
                max = arr[maxPos];
            }
        }
        arr[maxPos] = arr[i];
        arr[i] = max;
    }
}

void selectionSort(int arr[], int size)
{
    for( int i=0; i<size-1; i++ )
    {
        int smallPos = i;
        int smallest = arr[i];

        for( int j=i+1; j<size; j++ )
        {
            if( arr[j]<smallest )
            {
                smallPos = j;
                smallest = arr[smallPos];
            }
        }
        arr[smallPos] = arr[i];
        arr[i] = smallest;
    }
}

void bubbleSort(int arr[], int size)
{
    bool flag = true;
    for( int i=0; i<(size-1) && flag==true; i++ )
    {
        flag =false;
        for( int j=0; j<(size-i-1); j++ ) //on each pass compares from array start
```

```
        {
            if( arr[j] > arr[j+1] )
            {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                flag = true;
            }
        }
    }
}

void insertionSort(int arr[], int size)
{
    for( int i=1;     i<(size);    i++ )
    {
        int temp    =   arr[i];
        int j    =   i-1;

        while( temp < arr[j] &&  (j>=0)   )
        {
            arr[ j+1]   =   arr[ j ];   //moves element forward
            j  =  j-1;
        }

        arr[j+1]    =   temp;
    }
}

void merge(int *a, int *b, int low, int pivot, int high)
{
    int h, i, j, k;
    h = low;
    i = low;
    j = (pivot+1);

    while( ( h<=pivot ) && ( j<=high ) )
    {
        if(a[h]<=a[j])
        {
            b[i] = a[h];
            h++;
        }
        else
        {
            b[i]=a[j];
            j++;
        }
        i++;
    }

    if(h>pivot)
    {
        for(k=j; k<=high; k++)
        {
            b[i]=a[k];
            i++;
        }
    }
    else
    {
```

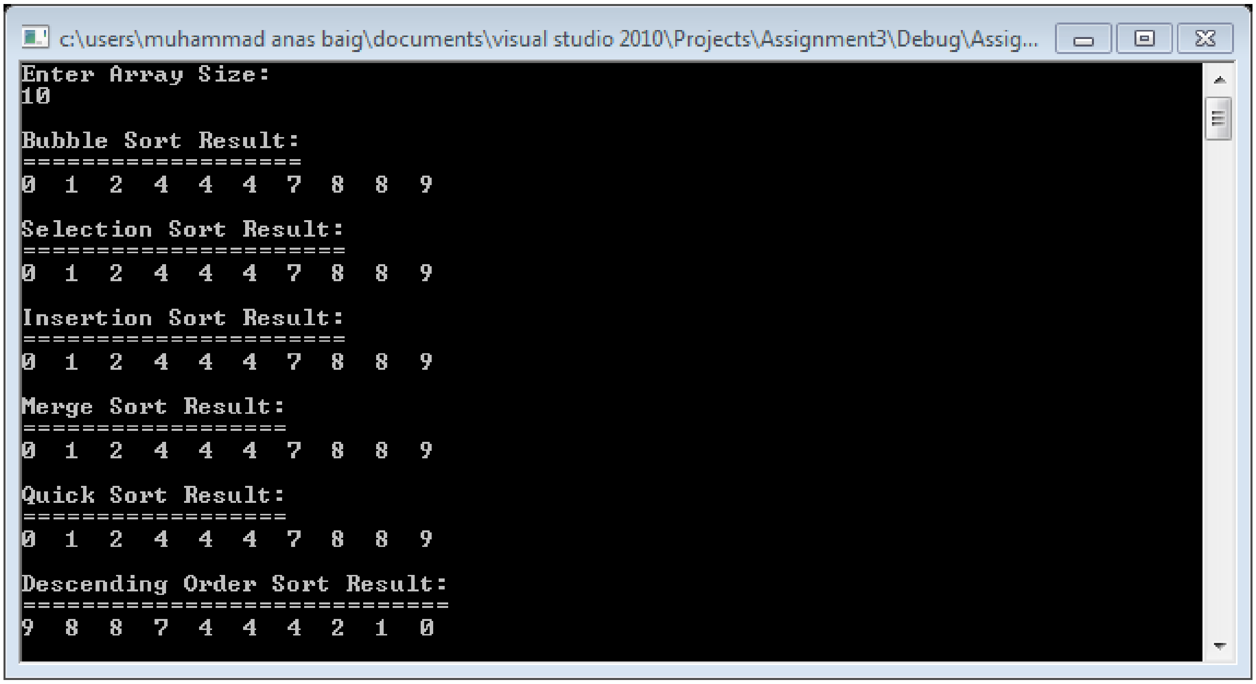
```
                for(k=h; k<=pivot; k++)
                {
                    b[i]=a[k];
                    i++;
                }
            }

            for(k=low; k<=high; k++)
            {
                    a[k]=b[k];
            }
        }

        void mergeSort(int *a, int*b, int low, int high)
        {
            int pivot;
            if( low<high )
            {
                pivot=( ( low+high )/2 );
                mergeSort( a, b, low, pivot);
                mergeSort(a, b, (pivot+1), high);
                merge(a, b, low, pivot, high);
            }
        }

        void split( int x[], int first, int last, int &pos )
        {
            int pivot = x[first];
            int left = first;
            int right = last;
            while (left < right)
            {
                while( x[right] > pivot)
                {
                    right--;
                }
                while( x[left] <= pivot && left < right )
                {
                    left++;
                }
                if (left < right)
                {
                    int temp;
                    temp = x[left];
                    x[left] = x[right];
                    x[right] = temp;
                }
            }
                x[first] = x[right];
                x[right] = pivot;
                pos = right;
        }

        void quickSort (int x[], int first, int last)
        {
            int pos;
            if ( first < last-1)
            {
                split (x, first, last, pos);
                quickSort (x, first, (pos-1));
                quickSort (x, pos + 1, last);
```

```
        }
    }
```

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

```
int main()
{
    int n;
    cout<<"Enter Number:"<<endl;
    cin>>n;
    //NOTE: Below individual arrays are declred for each sorting algorithm and then initialized with same elements to keep consistency in comparing algorithms working with same elements of array
    int *a = new int [n]; //array for bubble sorting
    int *b = new int [n]; //array for selection sorting
    int *c = new int [n]; //array for insertion sorting
    int *d = new int [n]; //array for merge sorting
    int *e = new int [n]; //array for quick sorting
    int *x = new int [n]; //array for merge sorting
    for(int i=0; i<n; i++) //initializing each array with same random numbers
    {
        a[i] = b[i] = c[i] = d[i] = e[i] = (rand() % n);
    }

    //=======================================================================================
    //bubble sorting
    //=======================================================================================
    cout<<endl;
    cout<<"Bubble Sort Result:"<<endl;
    cout<<"==================="<<endl;
    bubbleSort(a, n);
    for( int i=0; i<n; i++ )
    {
        cout<<a[i]<<"  ";
    }

    //=======================================================================================
    //selection sorting
    //=======================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Selection Sort Result:"<<endl;
    cout<<"======================"<<endl;
    selectionSort(b, n);
    for( int i=0; i<n; i++ )
    {
        cout<<b[i]<<"  ";
    }


    //=======================================================================================
    //insertion sorting
    //=======================================================================================
```

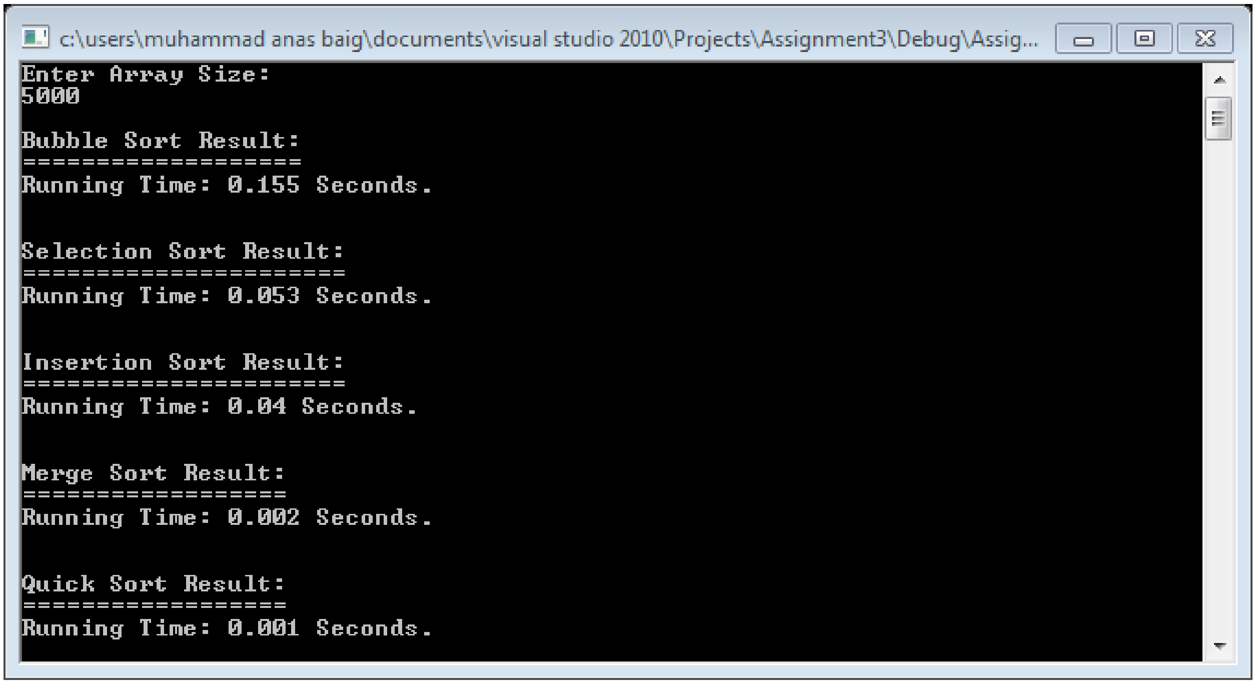
```
    cout<<endl;
    cout<<endl;
    cout<<"Insertion Sort Result:"<<endl;
    cout<<"======================"<<endl;
    insertionSort(c, n);
    for( int i=0; i<n; i++ )
    {
        cout<<c[i]<<"  ";
    }

    //=======================================================================================
    //merge sorting
    //=======================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Merge Sort Result:"<<endl;
    cout<<"=================="<<endl;
    mergeSort(d, x, 0, n-1);
    for( int i=0; i<n; i++ )
    {
        cout<<d[i]<<"  ";
    }

    //=======================================================================================
    //quick sorting
    //=======================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Quick Sort Result:"<<endl;
    cout<<"=================="<<endl;
    quickSort(e, 0, n-1);
    for( int i=0; i<n; i++ )
    {
        cout<<e[i]<<"  ";
    }

    //=======================================================================================
    //descending selection sorting
    //=======================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Descending Order Sort Result:"<<endl;
    cout<<"============================="<<endl;
    descendingSelectionSort(e, n);
    for( int i=0; i<n; i++ )
    {
        cout<<e[i]<<"  ";
    }

    getch();
}
```



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

```
int main()
{
    int n;
    cout<<"Enter Array Size:"<<endl;
    cin>>n;
    //NOTE: Below individual arrays are declred for each sorting algorithm and then initialized with same elements to keep consistency in comparing algorithms working with same elements of array
    int *a = new int [n]; //array for bubble sorting
    int *b = new int [n]; //array for selection sorting
    int *c = new int [n]; //array for insertion sorting
    int *d = new int [n]; //array for merge sorting
    int *e = new int [n]; //array for quick sorting
    int *x = new int [n]; //array for merge sorting
    for(int i=0; i<n; i++) //initializing each array with same random numbers
    {
        a[i] = b[i] = c[i] = d[i] = e[i] = (rand() % n);
    }

    //==================================================================================
    //bubble sorting
    //==================================================================================
    cout<<endl;
    cout<<"Bubble Sort Result:"<<endl;
    cout<<"=================="<<endl;
    clock_t a_time;
    a_time = clock();
    bubbleSort(a, n);
    a_time = clock() - a_time;
```

```
    cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;


    //========================================================================================
    //selection sorting
    //========================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Selection Sort Result:"<<endl;
    cout<<"======================"<<endl;
    a_time = clock();
    selectionSort(c, n);
    a_time = clock() - a_time;
    cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //========================================================================================
    //insertion sorting
    //========================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Insertion Sort Result:"<<endl;
    cout<<"======================"<<endl;
    a_time = clock();
    insertionSort(b, n);
    a_time = clock() - a_time;
    cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //========================================================================================
    //merge sorting
    //========================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Merge Sort Result:"<<endl;
    cout<<"=================="<<endl;
    a_time = clock();
    mergeSort(d, x, 0, n-1);
    a_time = clock() - a_time;
    cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //========================================================================================
    //quick sorting
    //========================================================================================
    cout<<endl;
    cout<<endl;
    cout<<"Quick Sort Result:"<<endl;
    cout<<"=================="<<endl;
    a_time = clock();
    quickSort(e, 0, n-1);
    a_time = clock() - a_time;
    cout<<"Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    getch();
}
```



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

```
int main()
{
    int n;
    cout<<"Enter Array Size:"<<endl;
    cin>>n;
    int *a = new int [n]; //dynamic array
    int *x = new int [n]; //dynamic array
    for(int i=0; i<n; i++) //initializing array with random numbers
    {
        a[i] = (rand() % n);
    }

    //======================================================================================
    //bubble sorting
    //======================================================================================
    cout<<endl;
    cout<<"Bubble Sort Result:"<<endl;
    cout<<"==================="<<endl;
    quickSort(a, 0, n-1); //make array already sorted for best case check
    clock_t a_time = clock(); //initializing clock
    bubbleSort(a, n); //checking for best case as array is already sorted
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Best Case Scenario Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds. "<<endl;
    descendingSelectionSort(a, n); //make array reversed for worst case check
    a_time = clock(); //initializing clock
```

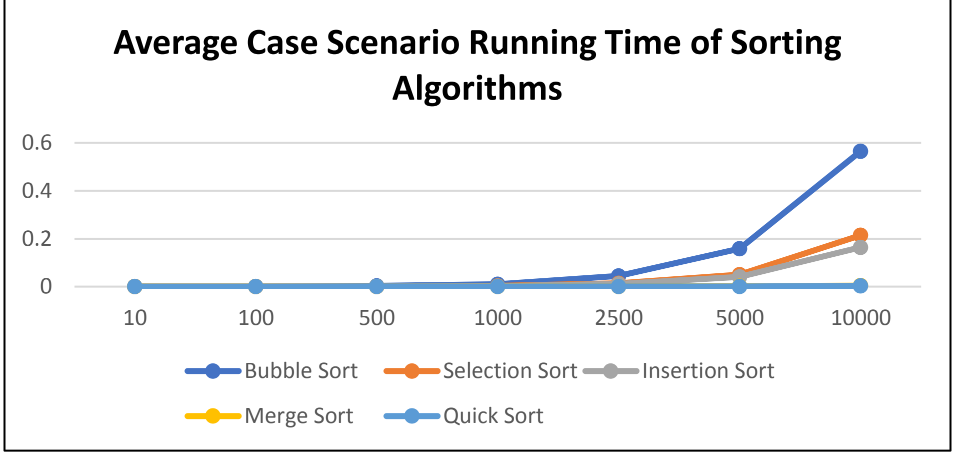
```
    bubbleSort(a, n); //checking for worst case as array is reversely ordered
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //=======================================================================================
    //selection sorting
    //=======================================================================================
    cout<<endl<<endl;
    cout<<"Selection Sort Result:"<<endl;
    cout<<"======================"<<endl;
    //array already sorted for best case check
    a_time = clock(); //initializing clock
    selectionSort(a, n); //checking for best case as array is already sorted
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
    descendingSelectionSort(a, n); //make array reversed for worst case check
    a_time = clock(); //initializing clock
    selectionSort(a, n); //checking for worst case as array is reversely ordered
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //=======================================================================================
    //insertion sorting
    //=======================================================================================
    cout<<endl<<endl;
    cout<<"Insertion Sort Result:"<<endl;
    cout<<"======================"<<endl;
    //array already sorted for best case check
    a_time = clock(); //initializing clock
    insertionSort(a, n); //checking for best case as array is already sorted
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
    descendingSelectionSort(a, n); //make array reversed for worst case check
    a_time = clock(); //initializing clock
    insertionSort(a, n); //checking for worst case as array is reversely ordered
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //=======================================================================================
    //merge sorting
    //=======================================================================================
    cout<<endl<<endl;
    cout<<"Merge Sort Result:"<<endl;
    cout<<"=================="<<endl;
    //array already sorted for best case check
    a_time = clock(); //initializing clock
    mergeSort(a, x, 0, n-1); //checking for best case as array is already sorted
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
```

```cpp
    descendingSelectionSort(a, n); //make array reversed for worst case check
    a_time = clock(); //initializing clock
    mergeSort(a, x, 0, n-1); //checking for worst case as array is reversely ordered
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    //============================================================================
    //quick sorting
    //============================================================================
    cout<<endl<<endl;
    cout<<"Quick Sort Result:"<<endl;
    cout<<"=================="<<endl;
    //array already sorted for best case check
    a_time = clock(); //initializing clock
    quickSort(a, 0, n-1); //checking for best case as array is already sorted
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Best Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;
    descendingSelectionSort(a, n); //make array reversed for worst case check
    a_time = clock(); //initializing clock
    quickSort(a, 0, n-1); //checking for worst case as array is reversely ordered
    a_time = clock() - a_time; //calculating clock difference
    cout<<"Worst Case Scenerio Running Time: "<<(float)a_time/CLOCKS_PER_SEC<<" Seconds."<<endl;

    getch();
}
```

```
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.
```

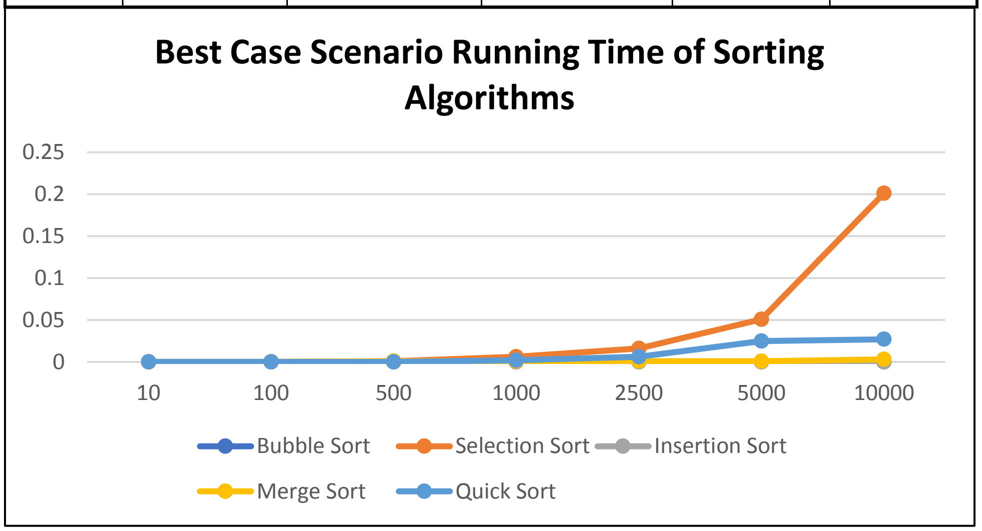
GRAPHS:

Random(Average) 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	0.003	0	0.001	0	0.001
1000	0.01	0.003	0.002	0	0
2500	0.044	0.014	0.011	0.001	0.001
5000	0.158	0.05	0.04	0.002	0.001
10000	0.565	0.214	0.164	0.004	0.003



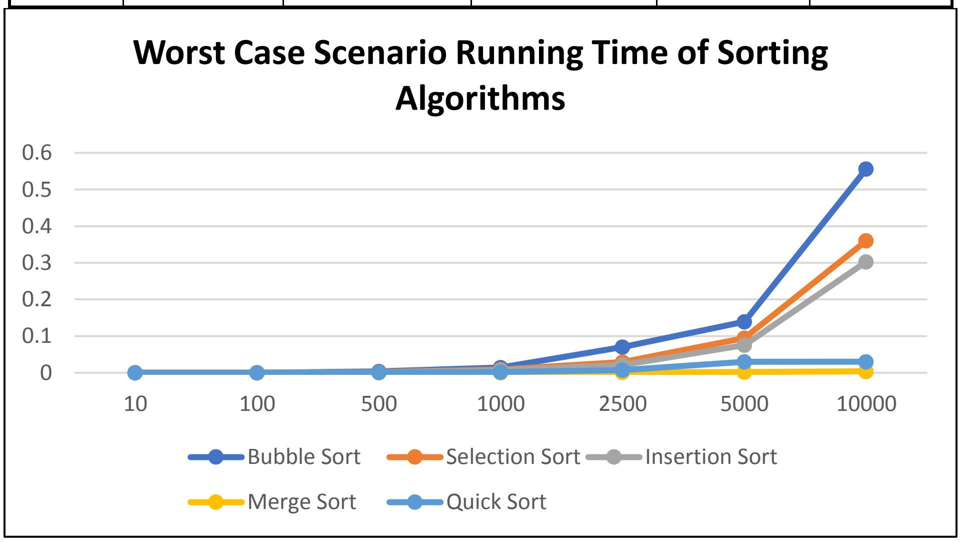
Conclusion:
As it can be seen from above graph and table that in random case scenario there is less difference between running times of sorting algorithms until 10,000 however at 10,000 Bubble Sort running time is increased gradually with respect to size.

Best 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	0.001	0	0.001	0
1000	0	0.006	0	0.001	0.002
2500	0	0.016	0	0.001	0.006
5000	0	0.051	0	0.001	0.025
10000	0	0.201	0	0.003	0.027



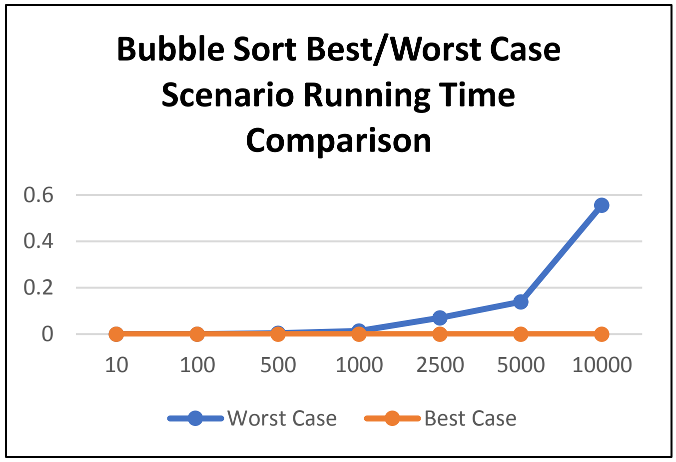
Conclusion:
In best case scenario it can be seen from above graph and table that instead of already sorted array Selection Sort is still consuming much time due to which it could be inferred that Selection Sort works almost same in either Best Case Scenario or Worst Case Scenario and is slow.

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



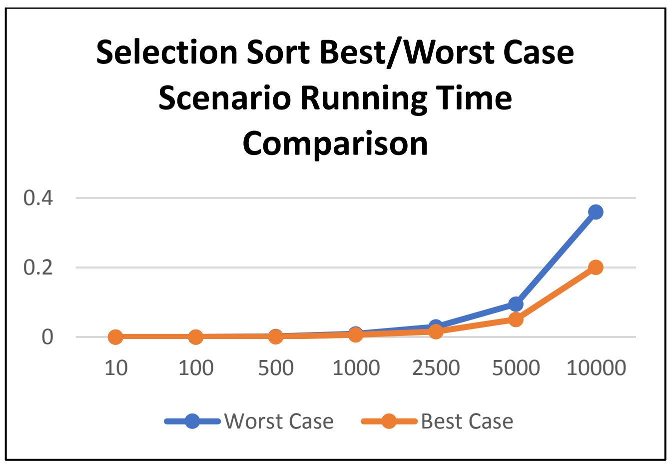
Conclusion:
In worst case scenario it can be seen from above Graph and Table that Merge Sort and Quick Sort are very much efficient and fast however Bubble Sort is slow as compared to other sorting algorithms.

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



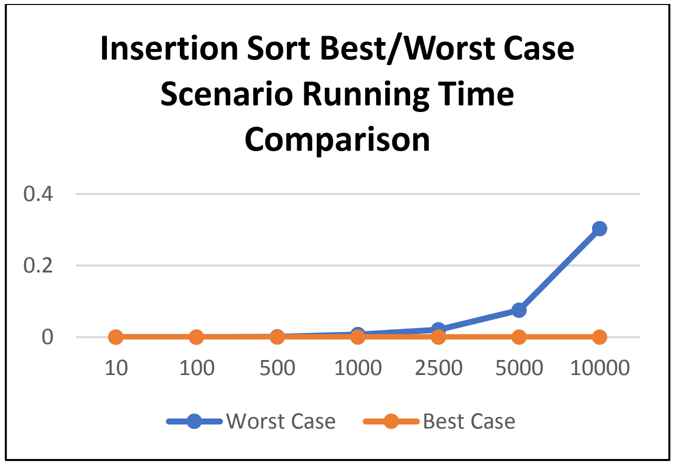
Conclusion:
In Bubble Sort it could be seen from above graph and table that Bubble Sort works efficiently for Best Case Scenario.

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



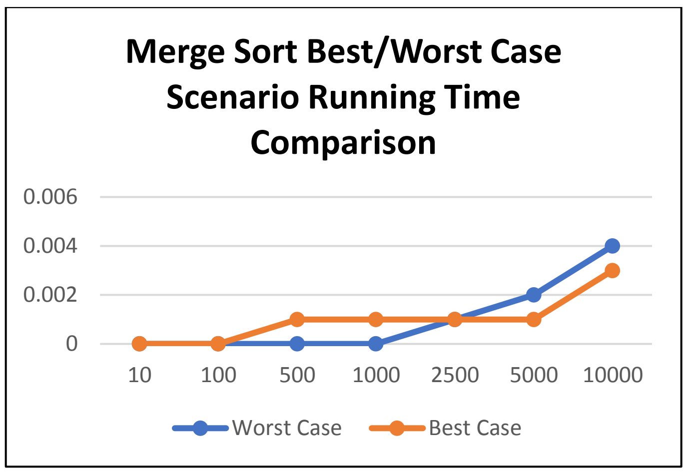
Conclusion:
In Selection Sort it can be seen from above Graph and Table that there is light difference between Worse Case Scenario and Best Case Scenario but it will affect dramatically when size of data will be too large.

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



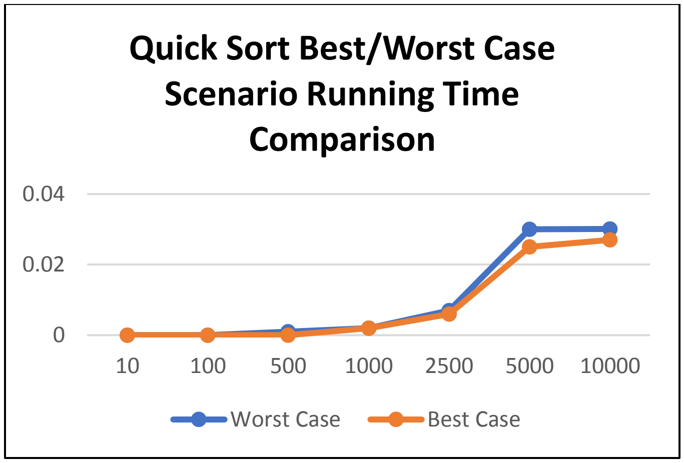
Conclusion:
It can be seen from above Graph and Table that Insertion Sort had a great efficient effect in Best Case Scenario.

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.
