

Subject

Assignment 1 (Sorting Techniques)

Done By

Name	Id	Group	Section
Omar Hossam Eldin Ebraheem	6667	1	1

Supervisor

Prof Dr. Amr El Masry

2020 – 2021

Introduction :

In this Assignment it is required to implement three $O(n^2)$ sorting algorithms which are (Selection Sort, Bubble Sort and Insertion Sort) and three $O(n \log(n))$ sorting algorithms which are (Merge Sort, Heap Sort and Quick Sort algorithm in the average case) , after that it is required to compare the running time performance of the implemented algorithms against each other , and in order to test the implementation and analyse the running time performance, it is required to generate a dataset of random numbers, and plot the relationship between the execution time of the sorting algorithm versus the input size in a graph using Excel sheet , all the six sorting , three of them are $O(n^2)$ and the other three are $O(n \log(n))$, all of them are implemented in (C Programming Language) using Code Blocks Software , also it is required to include in a PDF Report the following (Description of the program , Pseudo code for each algorithm , Sample Runs and Graph described in the Assignment Manual) .

Discussion :

The parts required to be included in this PDF Report are as the following (Description of the program , Pseudo code for each algorithm , Sample Runs and Graph described in the Assignment Manual) :

1. Description of the Program

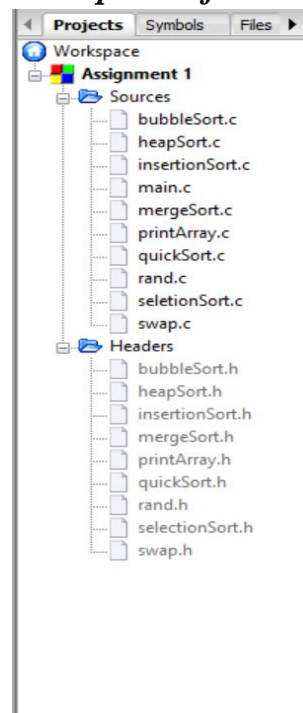


Figure 1.1




- This Program has been divided into separate files in order to be easy in tracing and be more readable , it have been divided to 10 (.C Files) with their Headers :
 - i. main.c : includes the main function which controls the program , and includes the interfacing with the user by taking the Array Size and the desired Sorting Algorithm.
 - ii. rand.c : includes generateArray() , which generates a random array of integers within a range in order to be sorted using one of the Sorting Algorithms .
 - iii. printArray.c : includes printArray() function , that prints the given array .

- iv. swap.c : includes the swap() function that swaps the given variables , it has been used by some Sorting Algorithms such as (Selection Sort , Bubble Sort , Heap Sort and Quick Sort) .
- v. selectionSort.c : includes the selectionSort() function that selects the first smallest element , swaps it with the first element in an unordered list , selects the second smallest element , swaps it with the second element in an unordered list and so on , until the list have been sorted .
- vi. bubbleSort.c : includes bubbleSort() function , that bubbles the largest element from un sorted part to sorted part , after that a wall moves one element before increasing the number of sorted elements and decreasing the number of unsorted ones .
- vii. insertionSort.c : includes insertionSort() function , that picks up the first element in an unordered list and to be transferred to the ordered part and inserted in an appropriate place , insertion sort is more simple and very easy especially for small numbers .
- viii. mergeSort.c : includes merge() function , that merges the two sub arrays , also it includes the mergeSort() function , that makes the recursive call with the two sub arrays in order to be sorted .
- ix. heapSort.c : includes heapify() function , that heapify the heap according to Max Heap Concept , heapsort() function , that includes the logic of build MaxHeap , and sorts the array by swapping the root with the last child , and includes a call to heapify() function , in order to re heapify the affected heap.
- x. quickSort.c : includes partition() function , that selects the pivot element as the last element in the array , places the pivot in the appropriate place , and places the smaller elements (smaller than pivot) to the left of the pivot , and places the greater elements (greater than the pivot) to the right of the pivot , and it includes quickSort() function that picks the index of the pivot , and make the recursive call of two sub arrays around the pivot .

2. Pseudo Code for each Algorithm

- In this section firstly it is going to briefly illustrate and describe each algorithm , then Pseudo Code for each Algorithm will be mentioned .
- Algorithm is defined as a sequence of steps that are be followed in order to solve a specific problem .
- While program is defined as the implementation of an Algorithm in some programming languages .

2.1 Selection Sort Algorithm :

- It selects the first smallest element , swaps it with the first in an unordered list , selects the second smallest element swaps it with the second element in an unordered list , and it repeats the operation until the array is sorted .
- Time Complexity Analysis :
 -  Best Case → $O(n^2)$
 -  Average Case → $O(n^2)$
 -  Worst Case → $O(n^2)$
- Space Complexity Analysis :
 - Selection Sort Algorithm is an in place algorithm , it performs all the computations on the original Array , so it doesn't need any extra array .
 - So Space Complexity is about to be → $O(1)$

- Note that Selection Sort Algorithm is not a very efficient Algorithm when data sets are large and it can be viewed from average and worst case complexities and from the graph below (*Figure 4.1*) .
- Note that selection sort algorithm uses minimum number of swap operations ($O(n)$) among all sorting Algorithms .
- Pseudo Code for Selection Sort Algorithm :
Assume array A , size =n
SelectionSort(A,n)

```

{
    for i ← 0 to n-2
    {
        iMin ← i
        for j ← i+1 to n-1
        {
            if(A[j] < A[iMin])
                iMin ← j
        }
        Swap(A[i] ,A[iMin])
    }
}

```

2.2 Bubble Sort Algorithm :

- In this algorithm the array is divided into two sub lists : sorted part and unsorted part .the largest element is bubbled from unsorted part to sorted part , after that a wall moves one element before increasing the number of sorted elements and decreasing the number of unsorted ones .
- Each time an element moves from unsorted part to sorted part , one sort pass is completed , so if given a list of n elements , it requires up to n-1 passes to sort all the data .
- Time Complexity Analysis :
 - 🚦 Best Case → $O(n)$ [in case of elements are already sorted]
 - 🚦 Average Case → $O(n^2)$
 - 🚦 Worst Case → $O(n^2)$
- Space Complexity Analysis :
 - Bubble Sort Algorithm is an in place algorithm , it performs all the computations on the original Array , so it doesn't need any extra array .
 - So Space Complexity is about to be → $O(1)$
- Note that Bubble Sort Algorithm is not a very efficient Algorithm when data sets are large and it can be viewed from average and worst case complexities and from the graph below (*Figure 4.1*) .
- Bubble Sort Algorithm is adaptive for the nearly sorted array , number of comparisons are less if the array is partially sorted .
- Best case is when the array is sorted .

- Pseudo Code for Bubble Sort Algorithm :

Assume array A , size =n

BubbleSort(A,n)

```
{
    for i ← 0 to n-2
    {
        for j ← 0 to n-i-2
        {
            if(A[j] > A[j+1])
            {
                Swap(A[j] , A[j+1])
            }
        }
    }
}
```

2.3 Insertion Sort Algorithm :

- The array is divided into two sub lists : sorted part and unsorted part .
- In each pass the first element in the unsorted part is picked up and transferred to sorted part , and placed in an appropriate place .
- For a given list of n elements it requires up to n-1 passes to sort all the data .
- Time Complexity Analysis :
 - 🚦 Best Case → $O(n)$ [in case of elements are already sorted]
 - 🚦 Average Case → $O(n^2)$
 - 🚦 Worst Case → $O(n^2)$
- Space Complexity Analysis :
 - Insertion Sort Algorithm is an in place algorithm , it performs all the computations on the original Array , so it doesn't need any extra array .
 - So Space Complexity is about to be → $O(1)$
- Note that Insertion Sort Algorithm is not a very efficient Algorithm when data sets are large and it can be viewed from average and worst case complexities and from the graph below (Figure 4.1).
- Insertion Sort Algorithm is adaptive for the nearly sorted array , number of comparisons are less if the array is partially sorted .
- Best case is when the array is sorted .

- Pseudo Code for Insertion Sort Algorithm :

Assume array A , size =n

InsertionSort(A,n)

```
{
    for i ← 1 to n -1
    {
        key = A[i]
        j ← i-1
        while(j >= 0 && A[j] > key)
        {
            A[j+1] = A[j]
            j ← j-1
        }
        A[j+1] ← key
    }
}
```

2.4 Merge Sort Algorithm :

- Merge Sort Algorithm is one of two important divide and conquer sorting algorithms , the other one is the Quick Sort algorithm .
- It is a recursive algorithm .
- It divides the list into two halves , sorts each halve separately , then merge the two sorted halves into one sorted array .
- Time Complexity Analysis :
- Since merge sort algorithm is a recursive algorithm so it must conclude firstly a Recurrence Relation .
 - $T(n) = 2T(n/2) + n$.
 - By solving the Recurrence Relation using any of the studied methods (Recurrence Tree , Mathematical Substitution or Master Theorem) .
 - After that the average and worst case complexities is about to be → $O(n \log(n))$
- Space Complexity Analysis :
- Merge Sort Algorithm requires an extra array whose size is equal to size of the original array , so space complexity is $O(n)$.

➤ Pseudo Code for Merge Sort Algorithm :

merge (Arr,left,mid,right)

```
{  
    // Find sizes of two subarrays to be merged  
    nL← mid - left + 1 (Length of left Array)  
    nR← right – mid (Length of right Array)  
    Create Left array of size nL → L[nL]  
    Create Right array of size nR → R[nR]  
    //Copy data to temp arrays  
    for i = 0 to nL-1  
        L[i] = Arr[left + i];  
    for j = 0 to nR-1  
        R[j] = Arr[mid + 1 + j];  
    i= 0;  
    j= 0;  
    k= 1;  
    while (i< nL && j< nR)  
    { if (L[i]<=R[j])  
        {  
            Arr[k] = L[i];  
            i =i+1;  
            k =k+1;  
        }  
        else  
        {  
            Arr[k] =R[j];  
            j =j+1;  
            k =k+1;  
        }  
    }  
}
```

```

    While (i< nL)
    {
        Arr[k] =L[i];
        i =i+1;
        k =k+1;
    }
    While (j< nR)
    {
        Arr[k]=R[j];
        j =j+1;
        k =k+1;
    }
}

mergesort(Arr, first,last)
    if (first < last) {
        mid = first+ (last-first)/2; // index of midpoint
        mergesort(Arr, first, mid);
        mergesort(Arr, mid+1, last);
        merge (Arr,first,mid,last);
    }

```

2.5 Heap Sort Algorithm :

- Binary Heap data structure is an array object that we can view as a nearly complete binary tree .
- It contains two functions which are heapsort() , and heapify() function .
- Time Complexity Analysis :
 - 🌈 Average Case ➔ $O(n \log(n))$.

- Pseudo Code for Heap Sort Algorithm :



```

heapify ( A , n , i )
{
    Largest = i ;
    L = leftchild(i);
    R = rightchild(i);
    If(l<heapsize&&A[l] >A[largest])
        Largest = l;
    If(R<heapsize&&A[R] >A[largest])
        Largest =R;
    If(largest!=i)
        Swap(A[i],A[largest]);
    heapify(A,n,largest)
}

heapSort(A,n)
{
    for i ➔ n/2-1 down to 0
        heapify(A,n,i)
    for i ➔ n-1 down to 1
        swap(A[0],A[i]);
        heapify(A,i,0);
}

```

2.6 Quick Sort :

- Quick Sort Algorithm is one of two important divide and conquer sorting Algorithms , the other one is the Merge Sort Algorithm .
- It contains two functions , partition() function that selects the pivot element as the last element in the array , places the pivot in the appropriate place , and places the smaller elements (smaller than pivot) to the left of the pivot , and places the greater elements (greater than the pivot) to the right of the pivot , and it includes quickSort() function that picks the index of the pivot , and make the recursive call of two sub arrays around the pivot.
- Time Complexity Analysis :
 - The Average case when the array is partitioned to a nearly two halves , so the Recurrence Relation will be $T(n) = 2T(n/2) + n$.
 - By solving the Recurrence Relation using any of the studied methods (Recurrence Tree , Mathematical Substitution or Master theorem) .
 Average Case ➔ $O(n \log(n))$
 - The worst case when the pivot element is chosen and it appears to be max or min so , all the elements will be moved left to the pivot (in case of pivot is the maximum) , or all the elements will be moved right to the pivot (in case of pivot is minimum) , so the Recurrence Relation will be , $T(n) = T(n-1) + T(0) + n$.
 - By solving the Recurrence Relation using any of the studied methods (Recurrence Tree , Mathematical Substitution or Master theorem) .
 Worst Case ➔ $O(n^2)$

- Pseudo Code for Quick Sort Algorithm :

Partition(A,low,high)

```
{
    Pivot = A[high];
    i ← low-1;
    for j ← low to high -1
        if ( A[j]<pivot)
        {
            i ← i+1 ;
            swap ( A[i],A[j]);
        }
    swap(A[i+1],A[high]);
    return i+1;
}
```

quickSort(A,low,high)

```
{
    If (low<high)
    {
        pi = partition(A,low,high);
        quickSort(A,low,pi-1);
        quickSort(A,low,pi+1);
    }
}
```

- Finally all the six Algorithms , three of them are $O(n^2)$, and three of them are $O(n \log(n))$ have been implemented using C Programming Language using Code Blocks Software .

3. Sample Runs

- In this part it is required sample runs from the implemented Code of the six Sorting Algorithms , so for simplicity in this part it will attach some Sample Runs of the six Sorting Techniques on random Array of integers of 10 and 100 input size elements .
- As it required to Generate random integer arrays of sizes (10, 100, 1000, 10000, 100000) then sort each of them using the 6 different sorting techniques and compute the running time for each algorithm.
- But for the large input size (1000 , 10000 , 100000) , it couldn't be take a single Screenshot of a sample runs of these large size .
- So it is recommended for the instructor in order to enjoy sorting Huge array of integers with input size (10 , 100 ,1000 , 5000 , 100000 , 50000 , 100000 , 150000 , 200000 , ...) It is recommended to run the code it self with any desirable input size , and enjoy watching the Sorted Array .
- Note that the random Array of integers have been generated using rand() , function which generates different random arrays each time it has been called .
- Also all the following Screenshots display the generated random array after Sorting it with the selected Sorting Algorithm from the Sorting Algorithms Menu .

3.1 Selection Sort Algorithm :

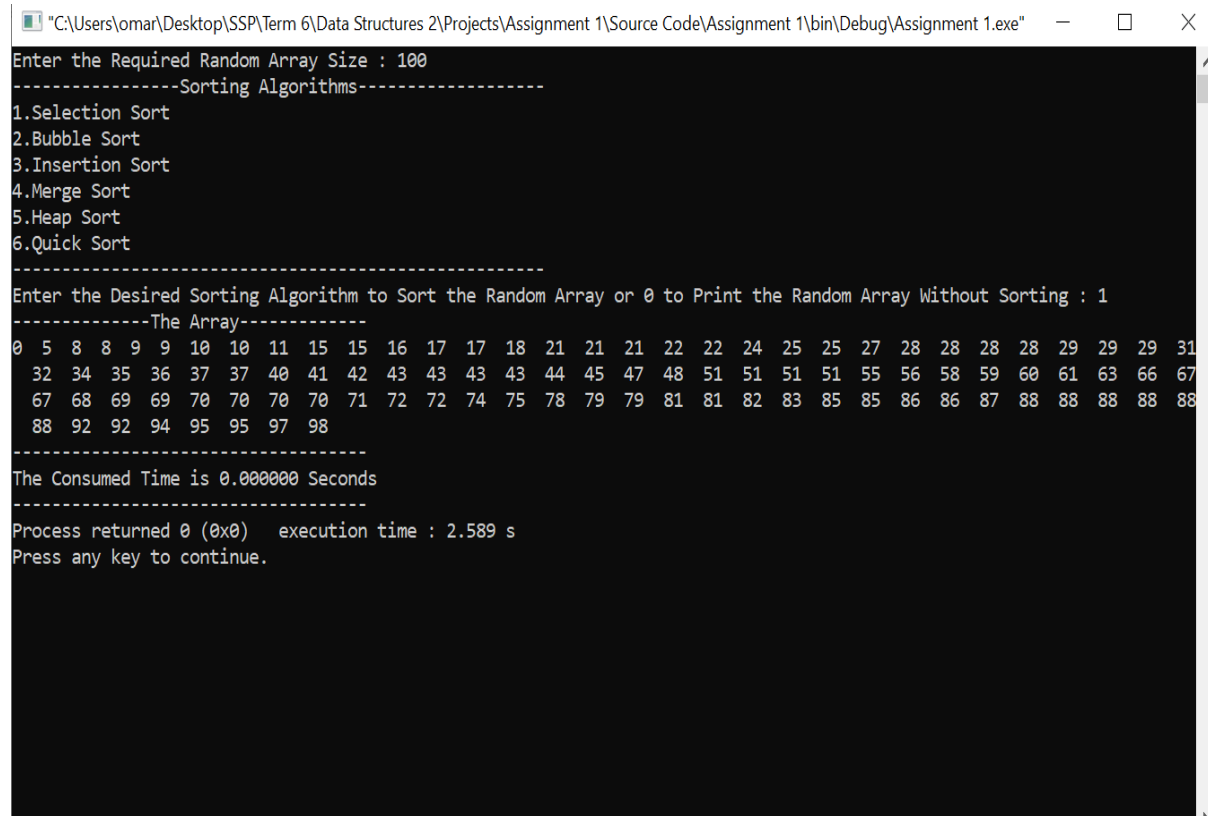
- Sorting random Array of 10 elements as shown in the figure below (*Figure 3.1.1*) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 1
-----The Array-----
26 26 34 38 47 47 68 72 74 86
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 5.317 s
Press any key to continue.
```

Figure 3.1.1

- Sorting random Array of 100 elements as shown in the figure below (*Figure 3.1.2*) :

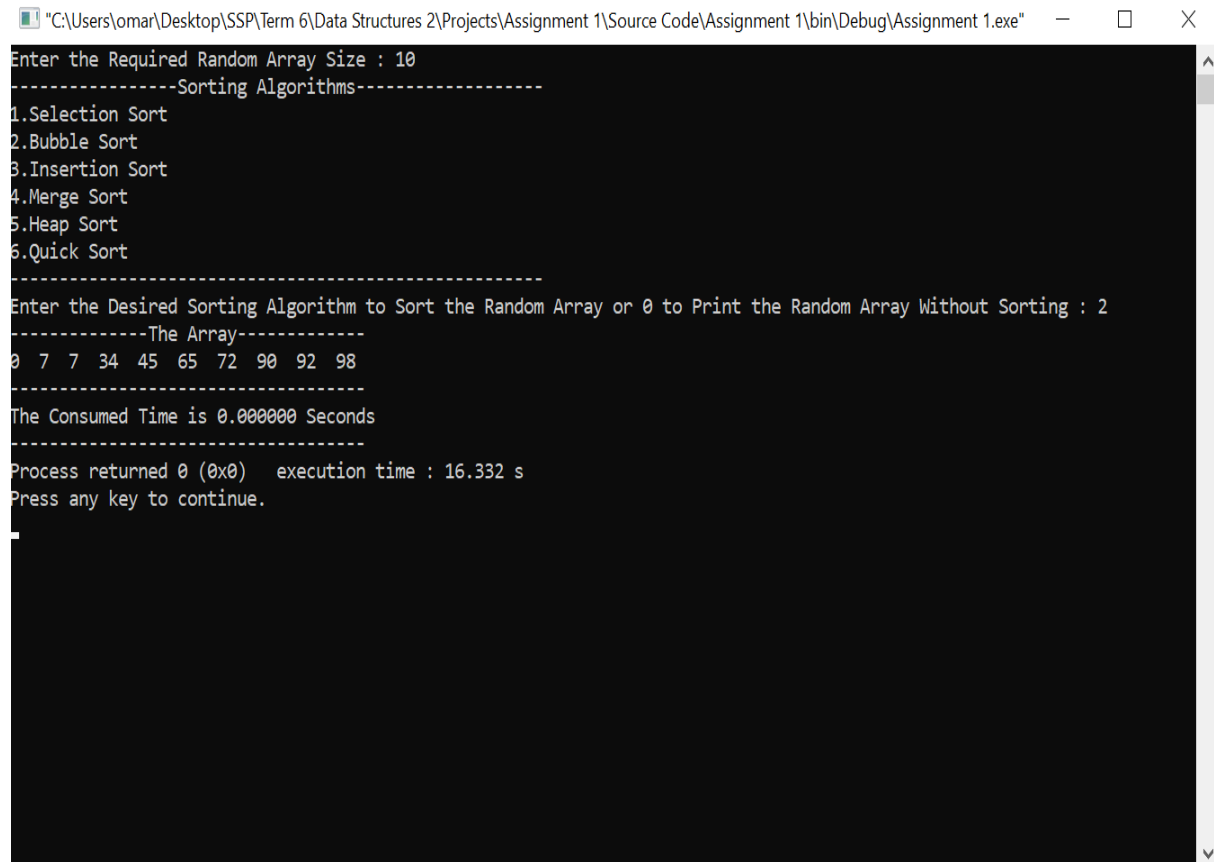


```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 1
-----The Array-----
0 5 8 8 9 9 10 10 11 15 15 16 17 17 18 21 21 21 22 22 24 25 25 27 28 28 28 28 29 29 29 31
32 34 35 36 37 37 40 41 42 43 43 43 43 44 45 47 48 51 51 51 51 55 56 58 59 60 61 63 66 67
67 68 69 69 70 70 70 70 71 72 72 74 75 78 79 79 81 81 82 83 85 85 86 86 87 88 88 88 88 88
88 92 92 94 95 95 97 98
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.589 s
Press any key to continue.
```

Figure 3.1.2

3.2 Bubble Sort Algorithm :

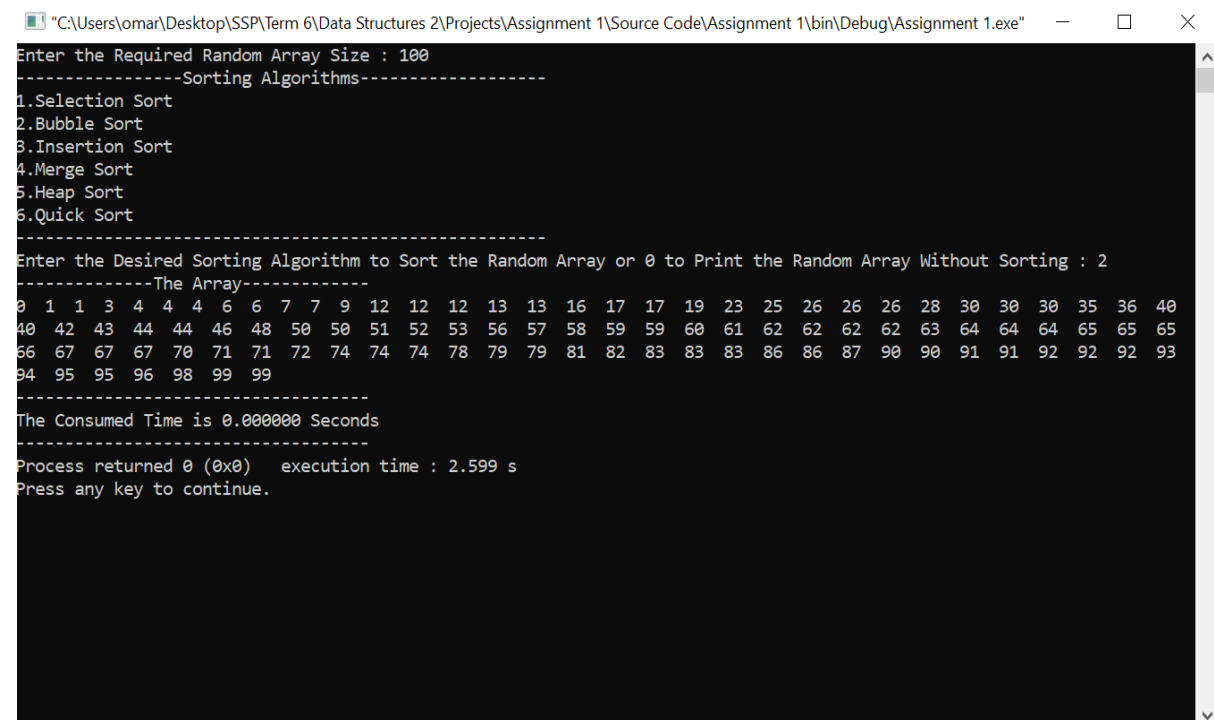
- Sorting random Array of 10 elements as shown in the figure below (Figure 3.2.1) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 2
-----The Array-----
0 7 7 34 45 65 72 90 92 98
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 16.332 s
Press any key to continue.
```

Figure 3.2.1

- Sorting random Array of 100 elements as shown in the figure below (Figure 3.2.2) :

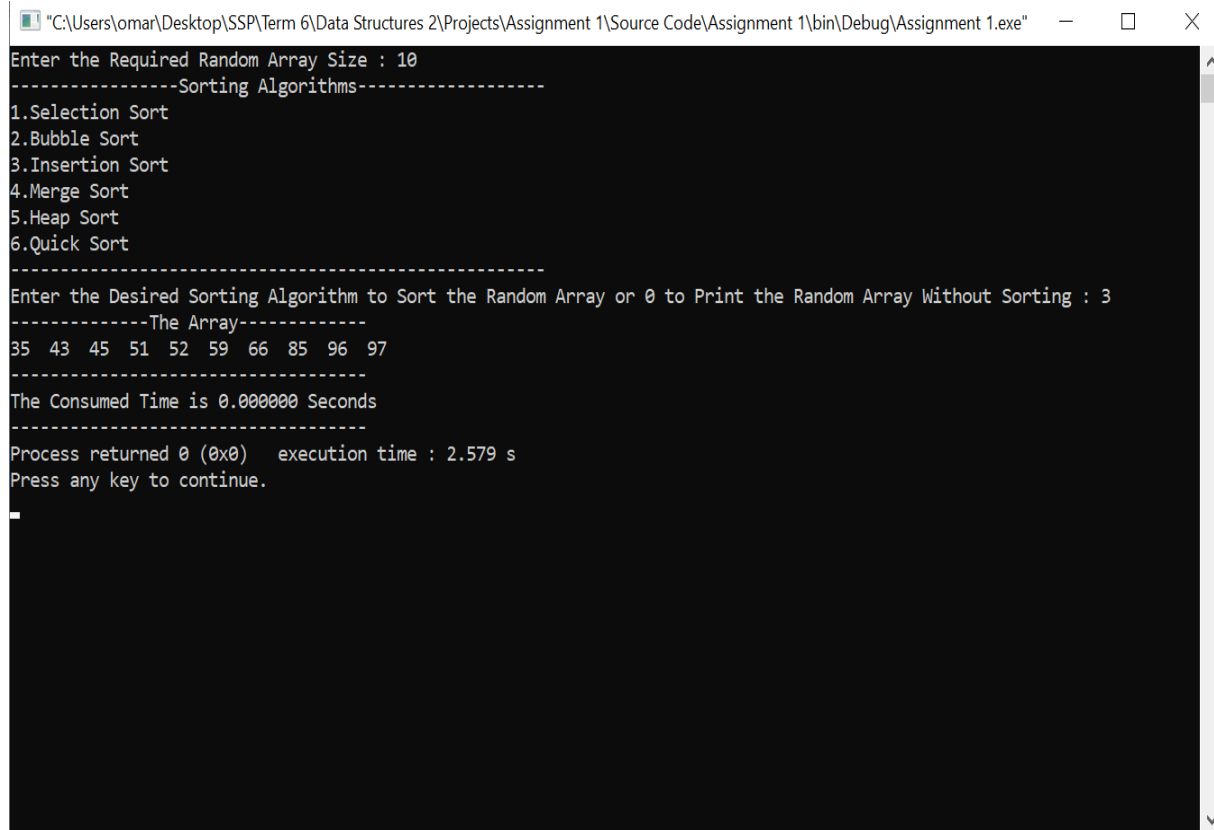


```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 2
-----The Array-----
0 1 1 3 4 4 4 6 6 7 7 9 12 12 13 13 16 17 17 19 23 25 26 26 26 28 30 30 30 35 36 40
40 42 43 44 44 46 48 50 50 51 52 53 56 57 58 59 59 60 61 62 62 62 62 63 64 64 64 65 65 65
66 67 67 67 70 71 71 72 74 74 74 78 79 79 81 82 83 83 83 86 86 87 90 90 91 91 92 92 92 93
94 95 95 96 98 99 99
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.599 s
Press any key to continue.
```

Figure 3.2.2

3.3 Insertion Sort Algorithm :

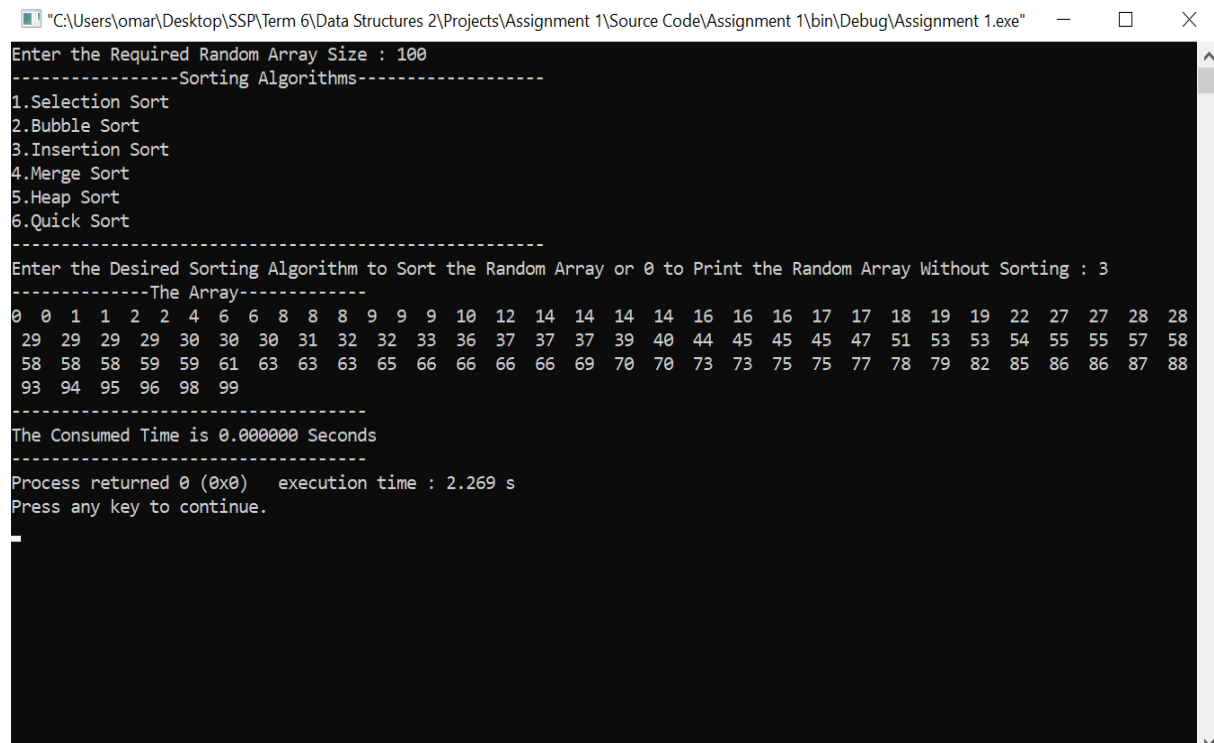
Sorting random Array of 10 elements as shown in the figure below (Figure 3.3.1) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 3
-----The Array-----
35 43 45 51 52 59 66 85 96 97
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.579 s
Press any key to continue.
```

Figure 3.3.1

➤ Sorting random Array of 100 elements as shown in the figure below (Figure 3.3.2) :

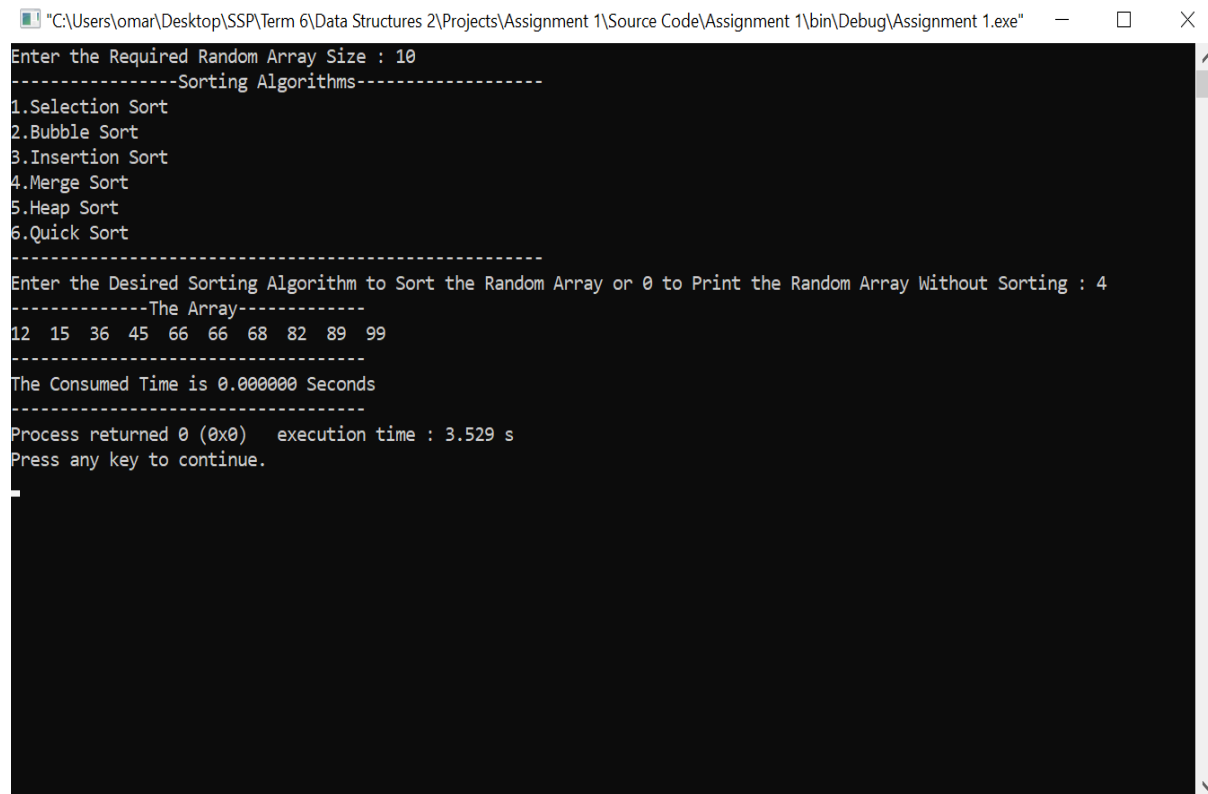


```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 3
-----The Array-----
0 0 1 1 2 2 4 6 6 8 8 8 9 9 9 10 12 14 14 14 14 16 16 16 17 17 18 19 19 22 27 27 28 28
29 29 29 29 30 30 30 31 32 32 33 36 37 37 37 39 40 44 45 45 45 45 47 51 53 53 54 55 55 57 58
58 58 58 59 59 61 63 63 63 65 66 66 66 66 69 70 70 73 73 75 75 77 78 79 82 85 86 86 87 88
93 94 95 96 98 99
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.269 s
Press any key to continue.
```

Figure 3.3.2

3.4 Merge Sort Algorithm :

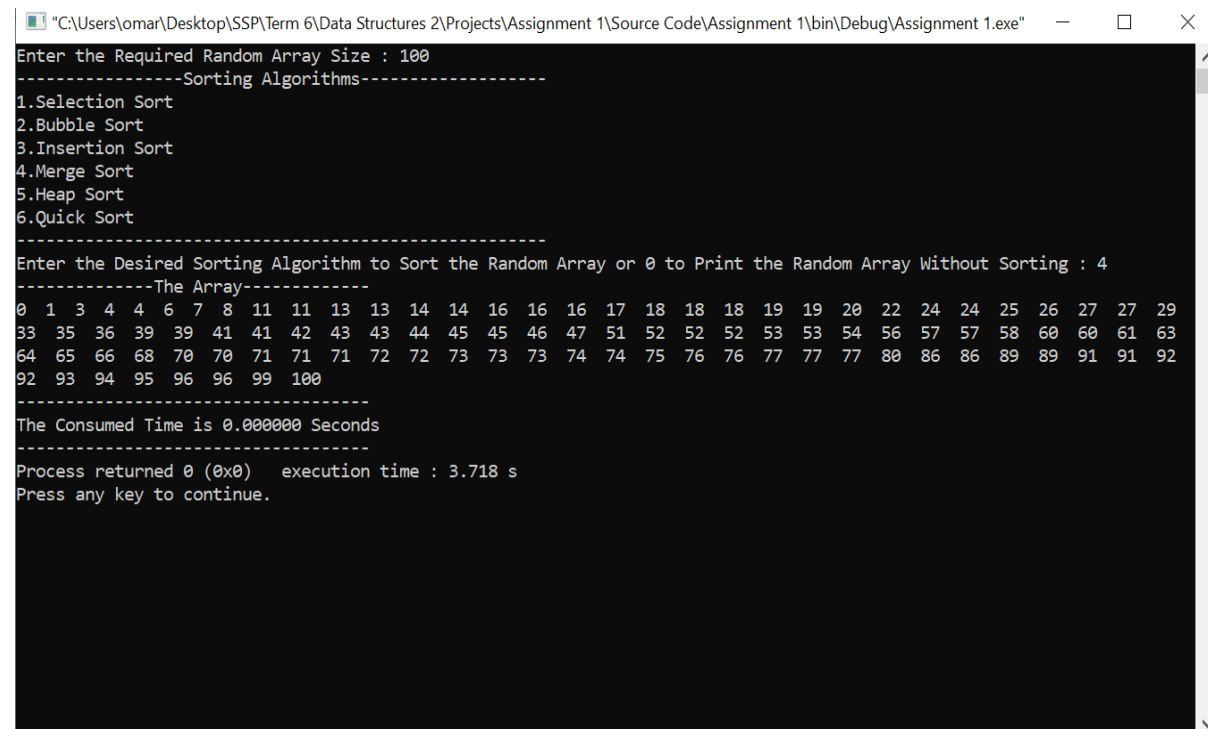
- Sorting random Array of 10 elements as shown in the figure below (Figure 3.4.1) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 4
-----The Array-----
12 15 36 45 66 66 68 82 89 99
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 3.529 s
Press any key to continue.
```

Figure 3.4.1

- Sorting random Array of 100 elements as shown in the figure below (Figure 3.4.2) :

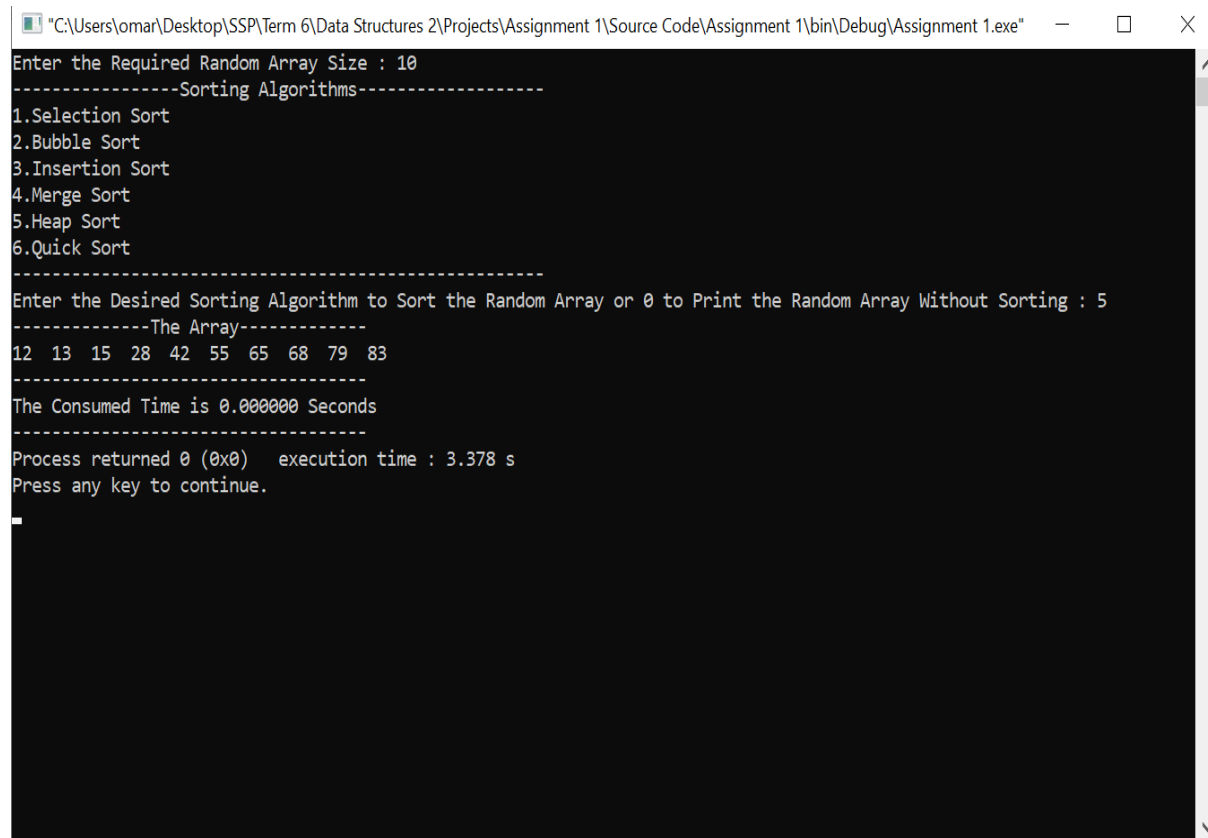


```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 4
-----The Array-----
0 1 3 4 4 6 7 8 11 11 13 13 14 14 16 16 16 17 18 18 18 19 19 20 22 24 24 25 26 27 27 29
33 35 36 39 39 41 41 42 43 43 44 45 45 46 47 51 52 52 52 53 53 54 56 57 57 58 60 60 61 63
64 65 66 68 70 70 71 71 71 72 72 73 73 73 74 74 75 76 76 77 77 77 80 86 86 89 89 91 91 92
92 93 94 95 96 96 99 100
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 3.718 s
Press any key to continue.
```

Figure 3.4.2

3.5 Heap Sort Algorithm :

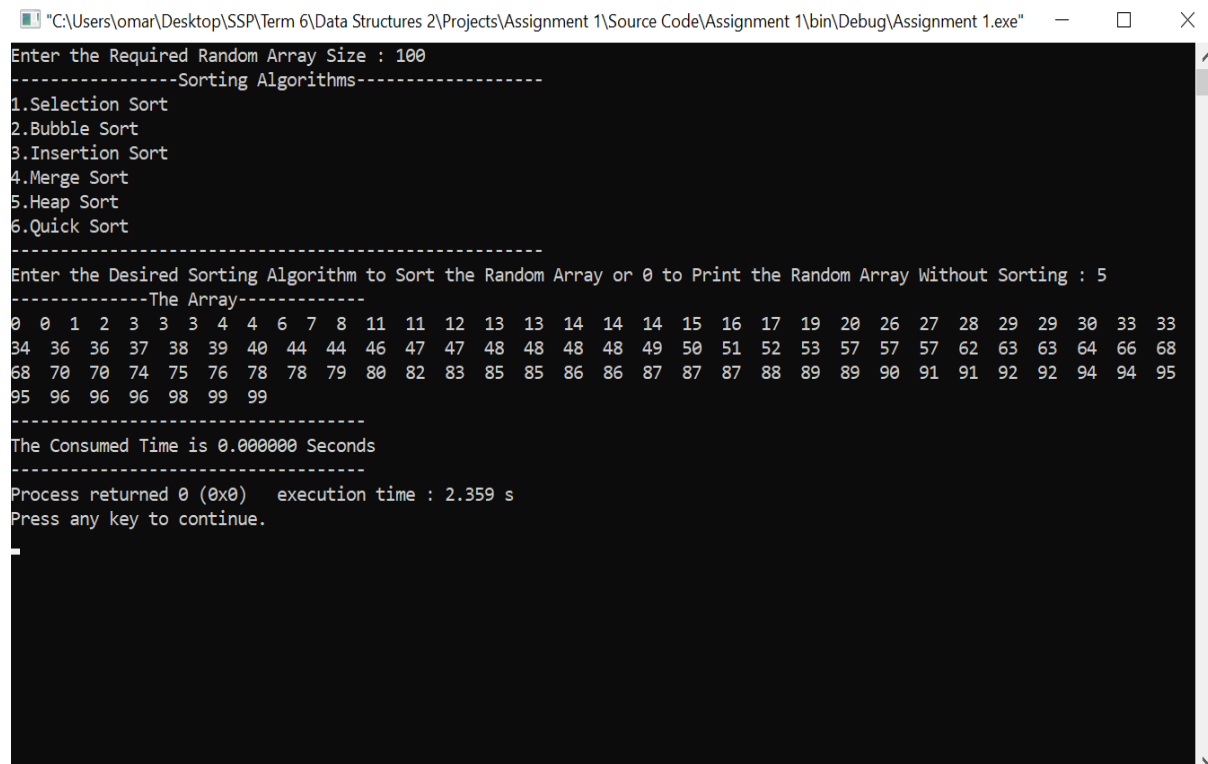
- Sorting random Array of 10 elements as shown in the figure below (Figure 3.5.1) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe" - □ ×
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 5
-----The Array-----
12 13 15 28 42 55 65 68 79 83
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 3.378 s
Press any key to continue.
-
```

Figure 3.5.1

- Sorting random Array of 100 elements as shown in the figure below (Figure 3.5.2) :

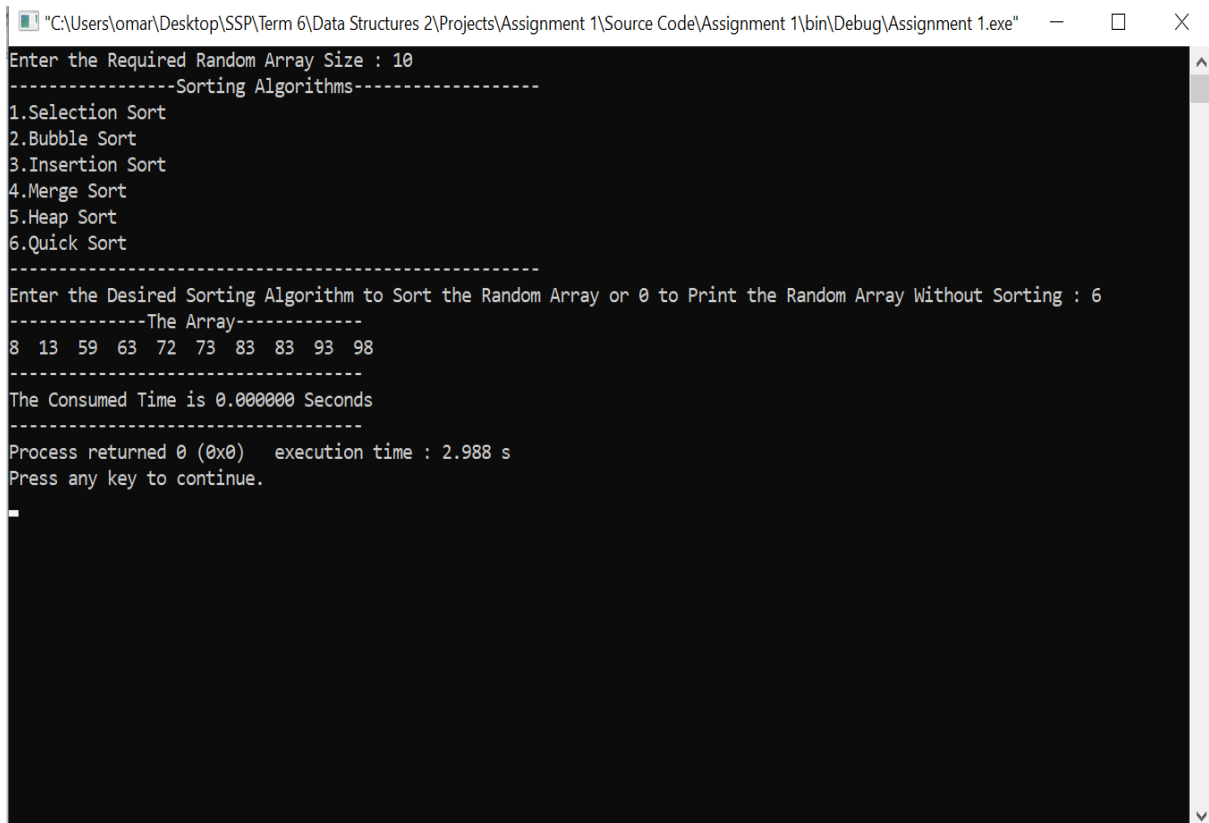


```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe" - □ ×
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 5
-----The Array-----
0 0 1 2 3 3 3 4 4 6 7 8 11 11 12 13 13 14 14 14 15 16 17 19 20 26 27 28 29 29 30 33 33
34 36 36 37 38 39 40 44 44 46 47 47 48 48 48 48 49 50 51 52 53 57 57 57 62 63 63 64 66 68
68 70 70 74 75 76 78 78 79 80 82 83 85 85 86 86 87 87 87 88 89 89 90 91 91 92 92 94 94 95
95 96 96 96 98 99 99
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.359 s
Press any key to continue.
-
```

Figure 3.5.2

3.6 Quick Sort Algorithm :

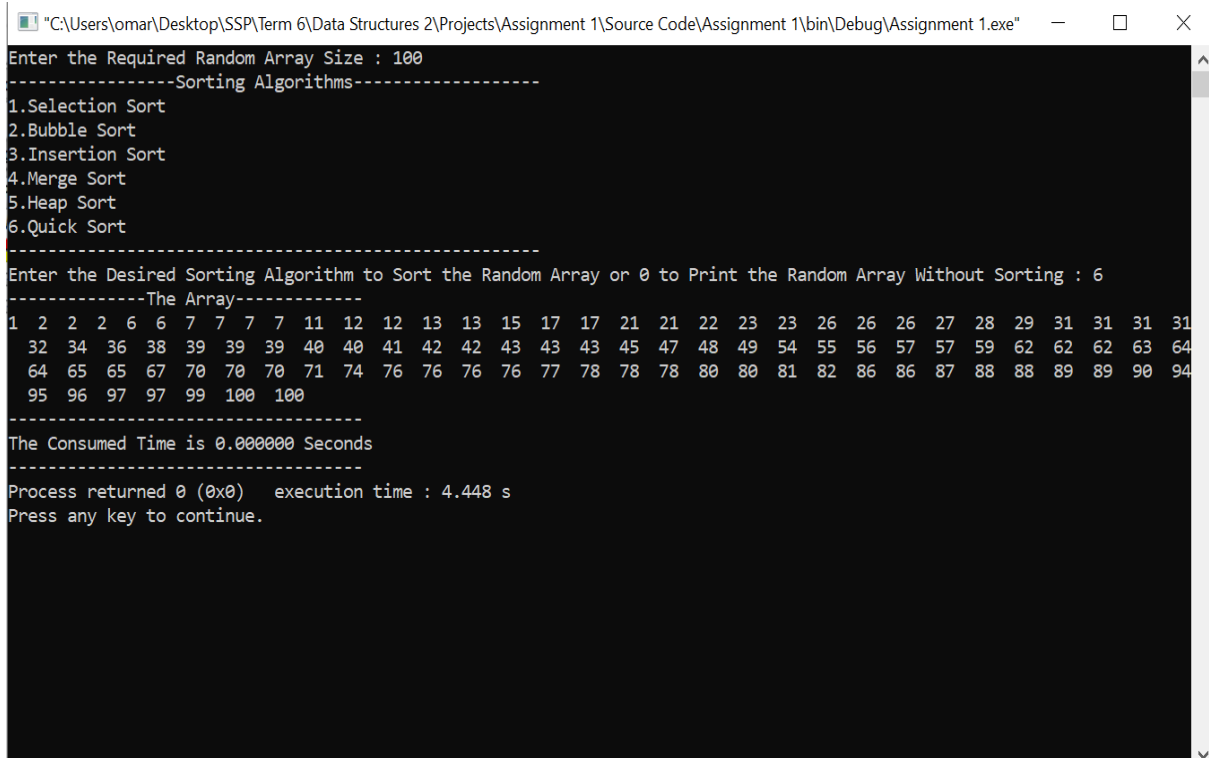
- Sorting random Array of 10 elements as shown in the figure below (*Figure 3.6.1*) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 10
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 6
-----The Array-----
8 13 59 63 72 73 83 83 93 98
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 2.988 s
Press any key to continue.
```

Figure 3.6.1

- Sorting random Array of 100 elements as shown in the figure below (*Figure 3.6.2*) :



```
"C:\Users\omar\Desktop\SSP\Term 6\Data Structures 2\Projects\Assignment 1\Source Code\Assignment 1\bin\Debug\Assignment 1.exe"
Enter the Required Random Array Size : 100
-----Sorting Algorithms-----
1.Selection Sort
2.Bubble Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
-----
Enter the Desired Sorting Algorithm to Sort the Random Array or 0 to Print the Random Array Without Sorting : 6
-----The Array-----
1 2 2 2 6 6 7 7 7 7 11 12 12 13 13 15 17 17 21 21 22 23 23 26 26 26 27 28 29 31 31 31 31
32 34 36 38 39 39 39 40 40 41 42 42 43 43 43 45 47 48 49 54 55 56 57 57 59 62 62 62 63 64
64 65 65 67 70 70 70 71 74 76 76 76 76 77 78 78 78 80 80 81 82 86 86 87 88 88 89 89 90 94
95 96 97 97 99 100 100
-----
The Consumed Time is 0.000000 Seconds
-----
Process returned 0 (0x0)   execution time : 4.448 s
Press any key to continue.
```

Figure 3.6.2

4. Graph

- A Graph is required between Time versus Array Size (Input Size) for the six different Sorting Algorithms, it is required to generate random arrays of different sizes and calculate the time required to sort it using the implemented Algorithms , and plot the results to a graph as shown in the figure below (Figure 4.1) :

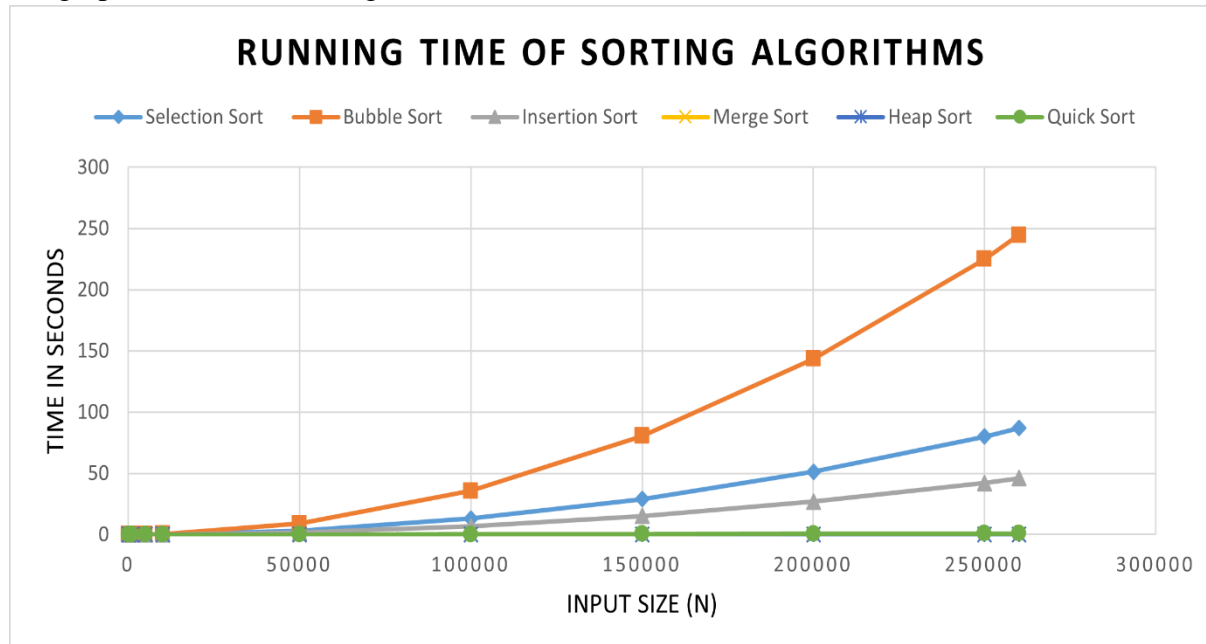


Figure 4.1

- From the figure above (Figure 4.1) , it can be concluded that the best Sorting Techniques to be used are (Quick Sort , Merge Sort and Heap Sort) since they are identical on each other on the same line as shown from the last line , as there running time in seconds are nearly to each others with the average case complexity $\rightarrow O(n \log(n))$, so they approximately lies with in the same range of Running Time .
- Also from the figure above (Figure 4.1) , it can be concluded that the worst Sorting Technique is Bubble Sort Algorithm as it consumes the more time among all the algorithms as input size gets larger .
- After that the second worst technique is Selection Sort Algorithm , as shown from the figure above (Figure 4.1) .
- The third worst one is the Insertion Sort Algorithm , as shown from the figure above (Figure 4.1) .
- And it is very logic as the worst three algorithms in running time (Bubble Sort , Selection Sort and Insertion Sort) consumes the $\rightarrow O(n^2)$ Running Time .
- While the best three algorithms in running time (Quick Sort , Merge Sort and Heap Sort) consumes in average case $\rightarrow O(n \log(n))$
- Note that the Input Size (N) of the figure above (Figure 4.1) has been tested upon the values (10 , 100 , 1000 , 5000 , 10000 , 50000 , 100000 , 150000 , 200000 , 250000 , 260000) .

Conclusion :

In this Assignment it is required to implement three $O(n^2)$ sorting algorithms which are (Selection Sort, Bubble Sort and Insertion Sort) and three $O(n \log(n))$ sorting algorithms which are (Merge Sort, Heap Sort and Quick Sort algorithm in the average case) , after that it is required to compare the running time performance of the implemented algorithms against each other , and in order to test the implementation and analyse the running time performance, it is required to generate a dataset of random numbers, and plot the relationship between the execution time of the sorting algorithm versus the input size in a graph using Excel sheet , all the six sorting , three of them are $O(n^2)$ and the other three are $O(n \log(n))$, all of them are implemented in (C Programming Language) using Code Blocks Software , also it is required to include in a PDF Report the following (Description of the program , Pseudo code for each algorithm , Sample Runs and Graph described in the Assignment Manual) , all the Sorting Techniques have been implemented using (C Programming Language) using Code Blocks Software , also it have been attached in the PDF Report the required features , also the Source Code of the Assignment have uploaded to Google Drive as a compressed file with the following Link .

Source Code Link

<https://drive.google.com/file/d/1rJenPcNlwEltbbKRNBfimG2I0DJp39Cy/view?usp=sharing>

References

1. Assignment 1 Manual
2. <https://www.youtube.com/watch?v=dB-EwQ9EhM>
3. <https://www.youtube.com/watch?v=0jdX22qM8JA&t=415s>
4. <https://www.youtube.com/watch?v=3PwVWX28dEE>
5. https://www.youtube.com/watch?v=7_tj54ZZGqU
6. <https://www.youtube.com/watch?v=TfkNkrKMF5c&t=253s>
7. <https://www.youtube.com/watch?v=yImI8GFrGuI>
8. <https://www.youtube.com/watch?v=gN-HG1O-Dfk&t=349s>
9. <https://www.youtube.com/watch?v=U4nQDEeOrqU&t=3s>
10. <https://www.youtube.com/watch?v=idXIb84gsqU&t=88s>