DEPARTMENT OF CYBER SECURITY

DESING AND ANALYSIS OF ALGORITHM ASSINGMENT	
SUBMITTED BY	211042 Noman Masood Khan A
SEMESTER	Fifth
SUBJECT	Design and analysis of algorithm
SUBMITTED TO	Dr. Ammar Masood

AIR University Islamabad

a. Explanation for the Insertion Sort and Merge Sort Code

i. Insertion Sorting Code

Programming language:

C++

```
#include<iostream>
using namespace std;

int main(){

    int A[10]={6,4,8,9,1,5,7,2,3,10},key,j;

    for (int i = 0; i<10; i++){

        key = A[i];

        j = i-1;

        while (j >= 0 && A[j]>key){

            A[j+1] = A[j];
            j=j-1;

        }
    A[j+1] = key;
    }

    for (int j = 0; j<10; j++){
        cout<<A[j]<<"\n";
    }

}
```

The program is basically working on the user initialized array.

The working principal of the program is that it is working on the key and the element of an array and their comparison to sort the element.

For loop is being used to iterate over all the elements of the array. In each iteration, A[i] element is being taken as a key and the element behind that key is being taken as j.

While the j does not reach the 0 and we found out the greater element than the key, till then we will quit the loop else the loop will continue.

In the while loop condition, we are checking that if the element at the j = i-1 position is greater than the key, that means that it is already sorted and quit the loop.

In the body of the while loop, when we have found that element less than the key, than put this element in the sorted array. At the j+1 position.

a. Merge Sorting code

The merge sorting code is working on the divide and conquer technique which is being done on the recursion.

The program has three parts:

i. The main function
ii. mergeSort function
iii. merge function

i. Main function

```
int main() {

  int arr[] = {6, 5, 12, 10, 9, 1};

  int size = sizeof(arr) / sizeof(arr[0]);

  mergeSort(arr, 0, size - 1);

  cout << "Sorted array: \n";

  for (int i = 0; i < size; i++)
    cout << arr[i] << " "<<endl;

}
```

The main function has the initialized array and it is calculating the size of the given array.

Then it is calling the mergeSort function will run the recursion and to divide the arrays to the pair to twos.

At-last the sorted array will be displayed to the user.

ii. MergeSort Function

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

// m is the point where the array is divided
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}
```

This function is taking the array and the index of the right and left sub-arrays.

Then mergeSort function is starting the recursion to break the sub-arrays to the sub-arrays and passing the indexes of the sub-arrays to the itself function.

First the left array is being sent to the mergeSort function, which it is dividing it to the more sub-parts of left and right arrays.

Then the mergeSort function is being run on the right sub-array to divide it to the sub-arrays.

When the both of them have been divided then the code will call the merge function to merge the left and right arrays.

iii. Merge function

```
void merge(int arr[], int p, int q, int r) {

  int n1 = q - p + 1, n2 = r - q;
  int L[n1], M[n2], i=0, j=0, k=p;

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];

  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}
```

The function is making a new array for the left sub array and right sub arrays and filling the elements with the for loops.

Then while loop is being run to compare the element of the newly created arrays to sort them in the in the arr array. The comparison is being done on the newly created left and right arrays L and M. Then the sorted element is being found and that is being put to the arr array. The arr array will be sorted array which will be merge and sorted.

The elements left out in any L or M array will instead to the arr array because the left out elements will be the one which were left out because the loop ended.

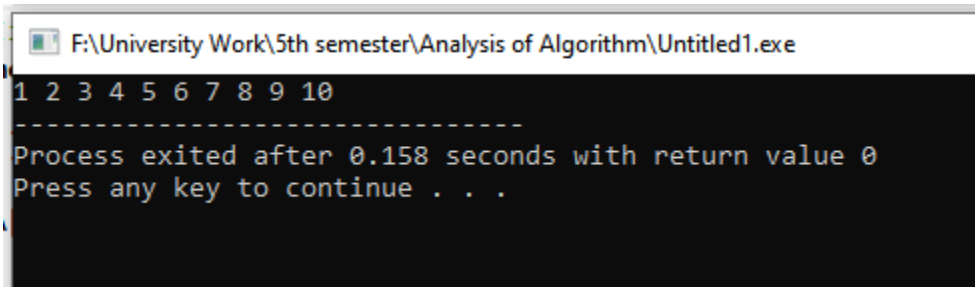
b. Test cases used to check the code

In the test cases I have provided the input to the to the code by initializing the array with the int numbers.

i. First test case for the insertion sorting

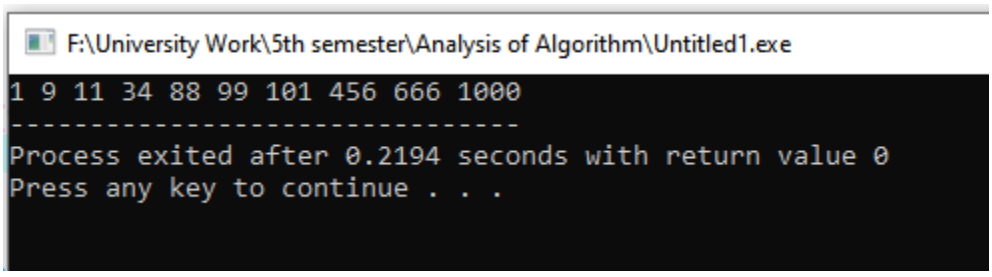
Input : {6,4,8,9,1,5,7,2,3,10}

Output :



Input: int A[10]={9,101,88,1,34,11,99,1000,666,456}

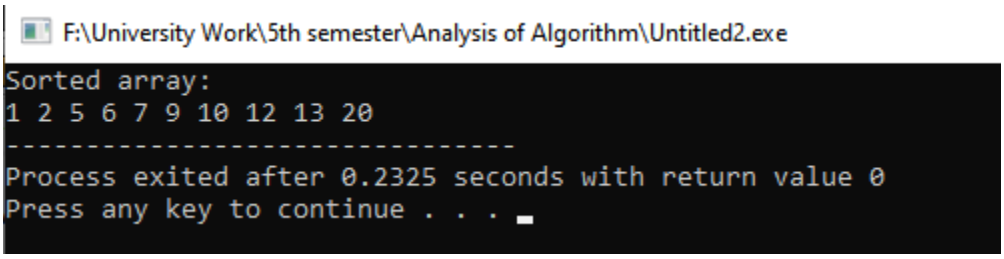
Output :



ii. Insertion Sorting

Input: int arr[] = {6, 5, 12, 10, 9, 1,7,2,13,20}

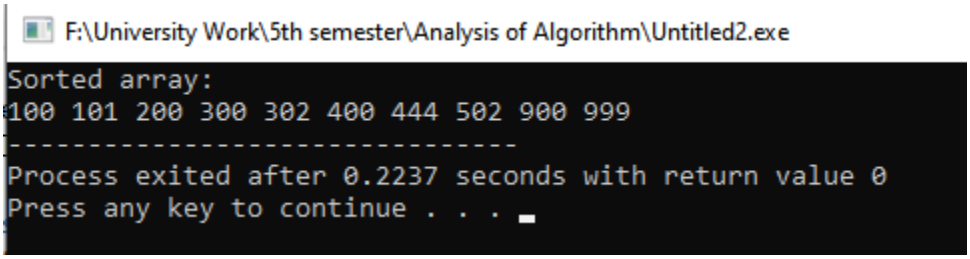
Output:



Input :

int arr[] = {200,300,100,400,900,502,444,302,101,999}

Output:



c. Procedure for generation of inputs

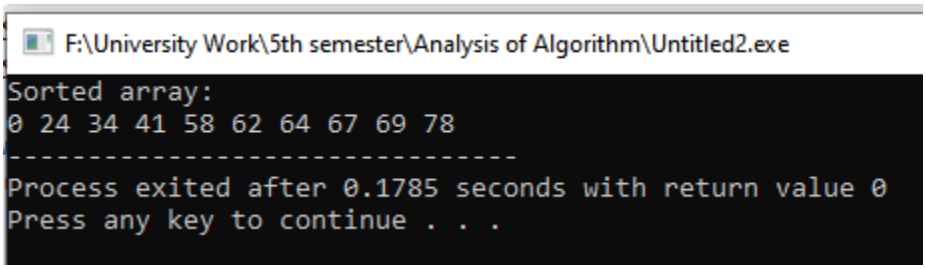
The method for the input generation is the random input generator function

I am using the random generator in the code to generate the 10 elements and store them inside the array. Here the size of the input being generated is 10 and the numbers are less than the 100.

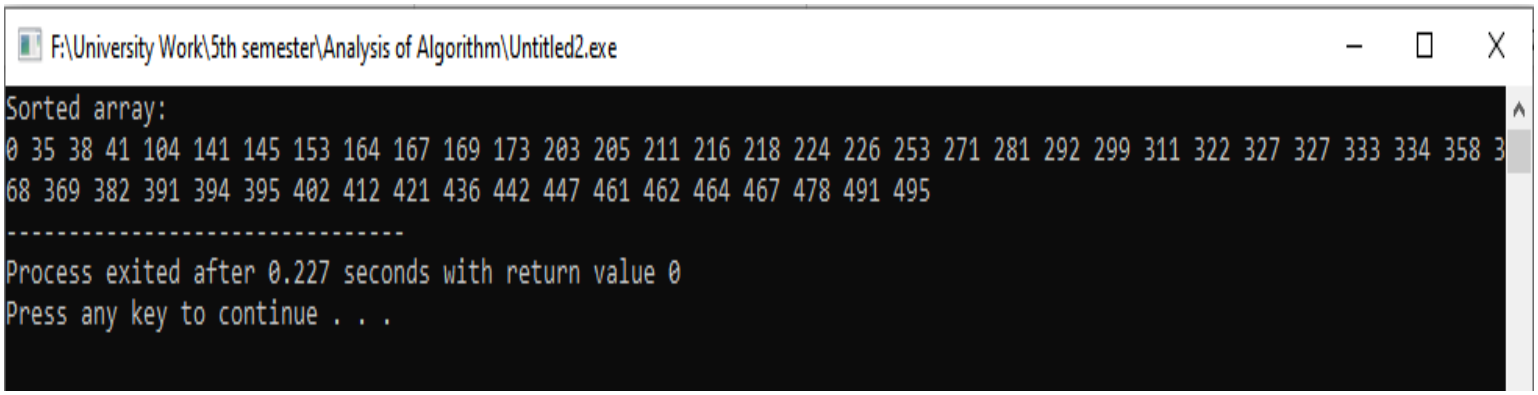
The **code for the random input generator** is :

```
int length=10;

int arr[length];

 for (int i=0;i<length;i++){

        arr[i] = rand() % 100;

 }
```

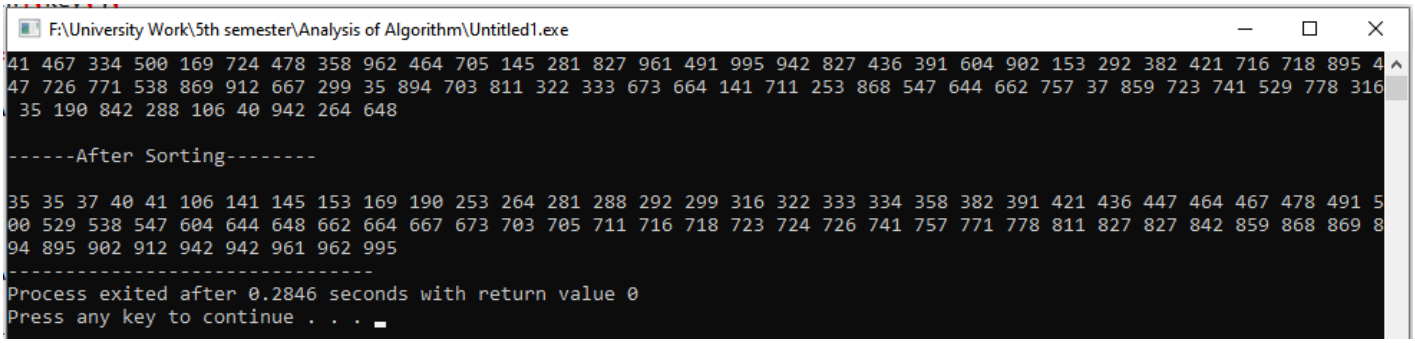
The **output** of the **merge sorting** for the **10 random numbers less than 100** being sorted is



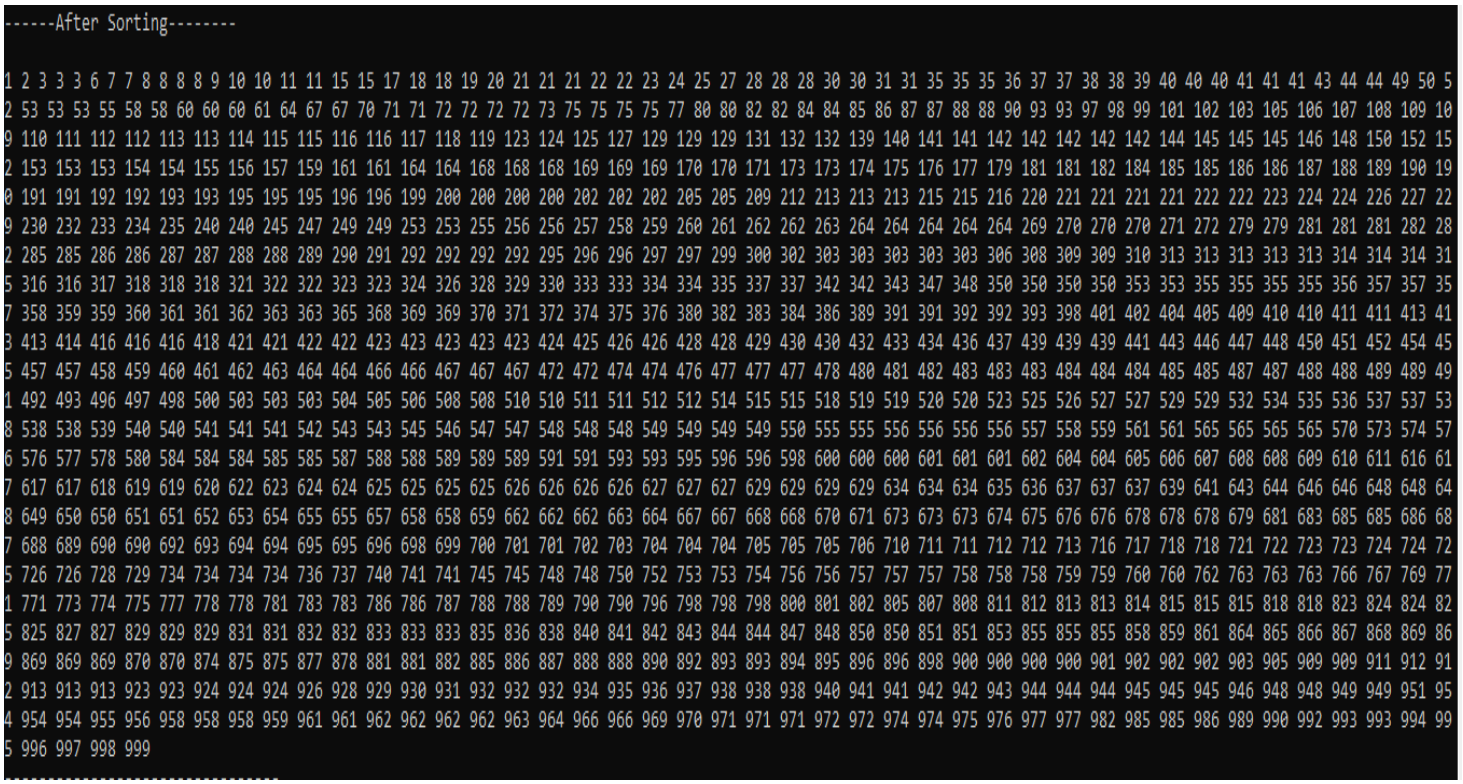
Now I am going to test the program on the 50 inputs which are less than 500, then the all the numbers where sorted perfectly



For the **insertion sorting** I am also doing the same and giving the **70 inputs** to the code which will be numbers between 0 to 1000. So the **merge sort result** before and after sorting can be seen is :



I have checked both the algorithms with the input size of 1000 and both the algorithms were performing very well as they were sorting the random inputs into the correct order.



d. Comparison of the time and memory complexity

Time Complexity for Insertion Sorting:

Best Case:

The best running time is **O(n)**

Average Case:

The average running time is **O(n^2).**

Worst Case:

The Best case running time is **O(n^2).**

Time complexity for the merge sorting

Best Case:

The best running time is **O(nlogn)**

Average Case:

The average running time is **O(nlogn).**

Worst Case:

The Best case running time is **O(nlogn).**

Memory Complexity for insertion sorting

Memory or space complexity for the insertion sorting is O(1). This is because the space required by the insertion sorting is constant and does not depend on the size of an input.

Memory Complexity for merge sorting

Memory or space complexity for the merge sorting is O(n). Merger sort being recursive takes up the additional space to store the elements in the memory.

Conclusion

I compared the result time of the both the algorithm by giving both the input size of the 5000 (five thousand). The merge sort gave back the result in very short time but the insertion sorting took time. This is because of the reason that the merge sorting takes nlogn and the insertion sort takes n^2 time complexity. With the inputs growing to bigger and bigger, the time to sort is growing for the insertion sort while it does not effects merge sort pretty much. Merge sort takes memory but it does not affect the efficiency of the merge sort algorithm.