Batch: B4 Roll. No.: 16010122828

Experiment: 9

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of sorting algorithms

Objective: To understand various searching methods

Expected Outcome of Experiment:

CO	Outcome
CO3	Demonstrate sorting and searching methods.

Websites/books referred:

1. Geekforgeeks

2. Stackoverflow

3. Javatpoint

4. Lecture PPT

Abstract: -

Sorting is a technique that is implemented to arrange the data in a specific order. Sorting is required to ensure that the data which we use is in a particular order so that we can easily retrieve the required piece of information from the pile of data.

Related Theory:

A **Stable Sort** is one which preserves the original order of input set, where the comparison algorithm does not distinguish between two or more items. A Stable Sort will guarantee that the original order of data having the same rank is preserved in the output.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some Sorting Algorithms are stable by nature, such as Bubble Sort, Insertion Sort, Merge Sort, Count Sort etc.

An **in-place** algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed. In Place: Bubble sort, Selection Sort, Insertion Sort, Heapsort.

Not In-Place: Merge Sort. Note that merge sort requires O(n) extra space.

Number of passes in a sorting algo is the number of iterations in the loop required to sort an array completely. A sorting algorithm goes through a list of data a number of times, comparing two items that are side by side to see which is out of order. It will keep going through the list of data until all the data is sorted into order. Each time the algorithm goes through the list it is called a 'pass'.

Quicksort uses extra space for recursive function calls. It is called in-place according to broad definition as extra space required is not used to manipulate input, but only for recursive calls.

Assigned Sorting Algorithm: Insertion/counting

a) Insertion sort Step

1 - Start

Step 2 - If the element is the first element, assume that it is already sorted. Return 1.

Step 3 - Pick the next element and store it separately in a key. Step 4

- Now, compare the key with all elements in the sorted array.

Step 5 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 6 - Insert the value.

Step 7 - Repeat until the array is sorted.

Step 8 - Stop

b) Counting sort:

Step 1: Start

Step 2: Call the function using countsort(array, size)

Step 3: The value of Max is the largest element in array and initialize the count array with all zeros

Step 4: For j ← 0 to size, find the total count of each unique element and

Step 5: Store the count at jth index in the count array

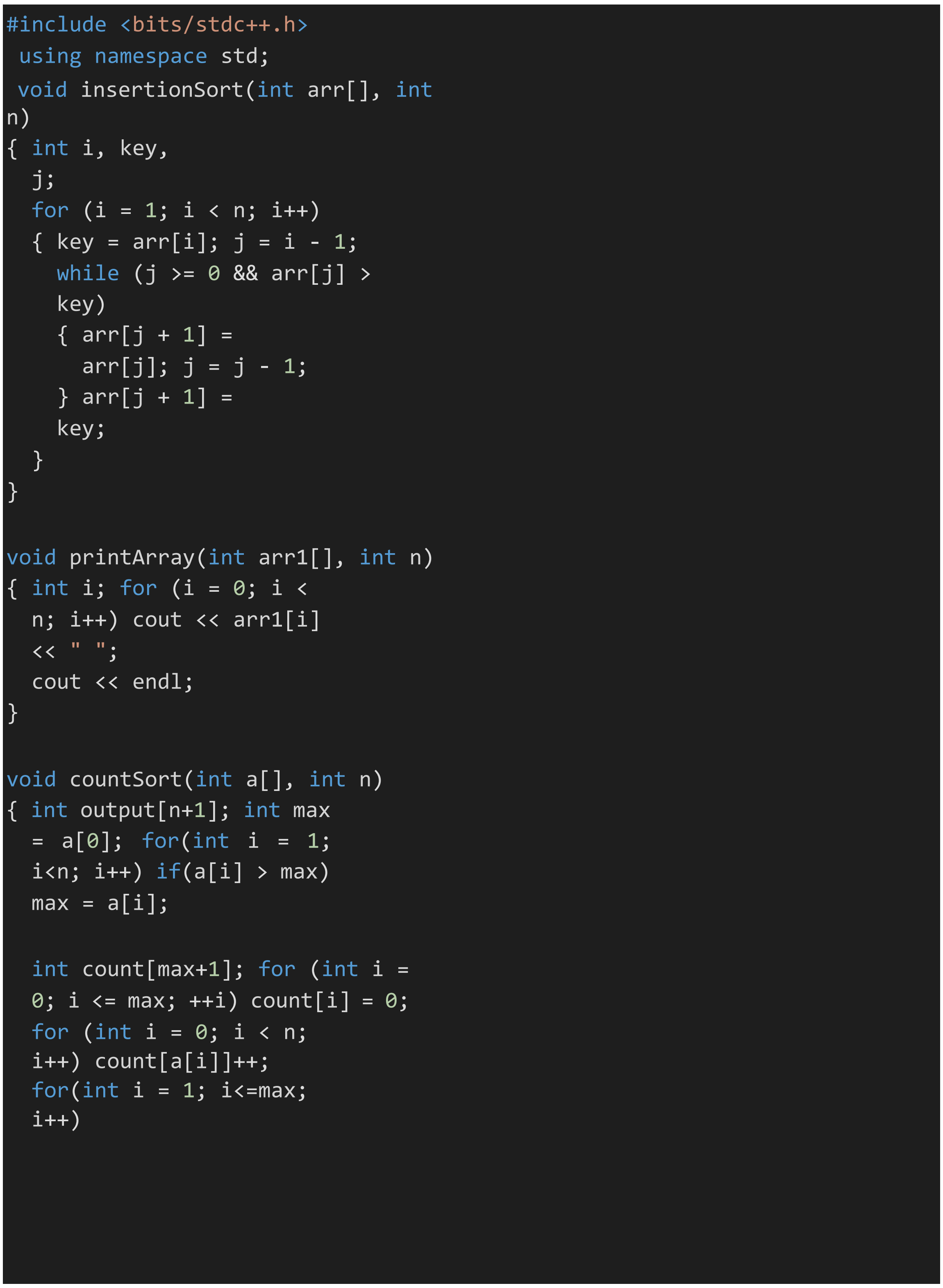
Step 6: For i ← 1 to max, find the cumulative sum and store it in count array itself

Step 7: For j ← size down to 1, restore the elements to array

Step 8: Decrease count of each element restored by 1 Step 9:

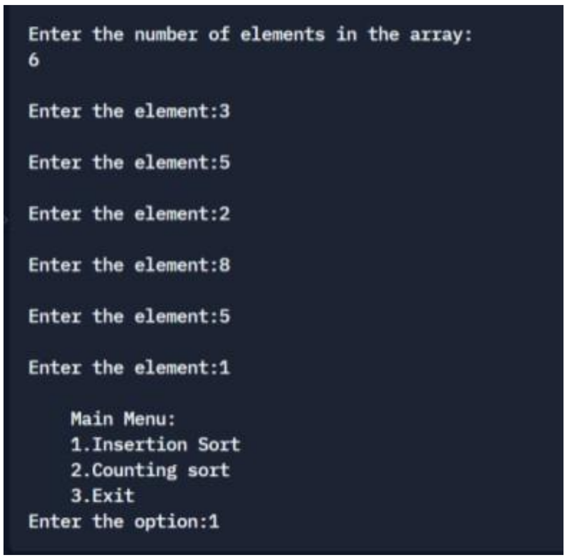
Stop

Code and output screenshots for assigned sorting algorithm:

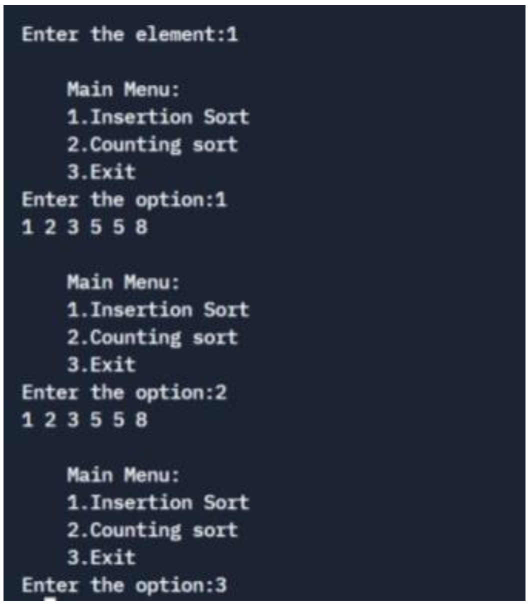


```
#include <bits/stdc++.h>
 using namespace std;
 void insertionSort(int arr[], int
n)
{ int i, key,
  j;
  for (i = 1; i < n; i++)
  { key = arr[i]; j = i - 1;
    while (j >= 0 && arr[j] >
    key)
    { arr[j + 1] =
      arr[j]; j = j - 1;
    } arr[j + 1] =
    key;
  }
}

void printArray(int arr1[], int n)
{ int i; for (i = 0; i <
  n; i++) cout << arr1[i]
  << " ";
  cout << endl;
}

void countSort(int a[], int n)
{ int output[n+1]; int max
  = a[0]; for(int i = 1;
  i<n; i++) if(a[i] > max)
  max = a[i];

  int count[max+1]; for (int i =
  0; i <= max; ++i) count[i] = 0;
  for (int i = 0; i < n;
  i++) count[a[i]]++;
  for(int i = 1; i<=max;
  i++)
  count[i] += count[i-1];
  for (int i = n - 1; i >= 0; i--)
  { output[count[a[i]] - 1] = a[i]; count[a[i]]--;
  } for(int i = 0; i<n; i++)
  { a[i] = output[i];
  } }
int main()
{ int arr[25],arr1[25]; int no,no1; int element; cout<<"\nEnter the
  number of elements in the array:\n"; cin>>no; no1=no; int i;
  for(i=0;i<no;i++)
    { cout<<"\nEnter the element:"; cin>>element;
      arr[i]=element;
    } for(i=0;i<no;i++)
    { arr1[i]=arr[i];
    } int option; do
    { cout<<"\n\tMain Menu:\n"; cout<<"\t1.Insertion
        Sort\n"; cout<<"\t2.Counting sort\n";
        cout<<"\t3.Exit";
        cout<<"\nEnter the option:"; cin>>option;
        switch(option)
        { case 1:
                insertionSort(arr, no); printArray(arr, no);
                break;
            case 2:
                countSort(arr1,no1);
                printArray(arr1,no1);
                break;
            case 3:
            break;

        }

    }
    while(option!=3);
    return 0;
}
```

Output



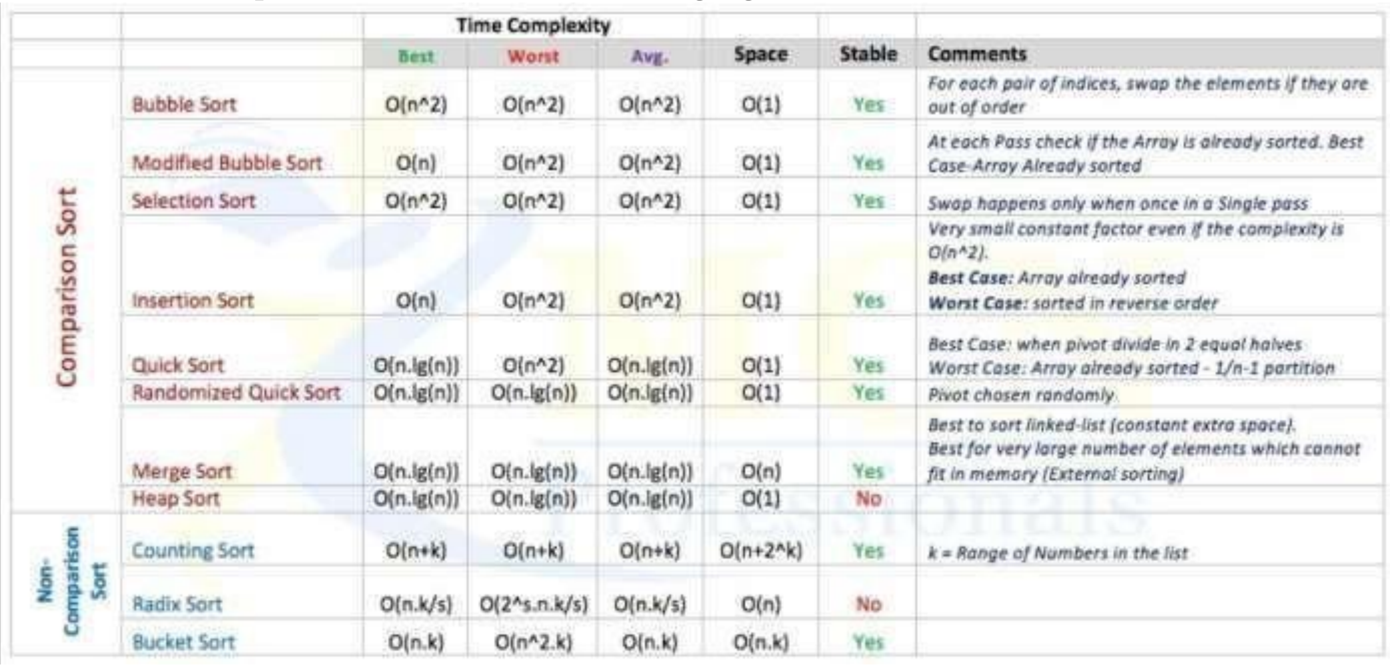
```
Enter the number of elements in the array:
6

Enter the element:3

Enter the element:5

Enter the element:2

Enter the element:8

Enter the element:5

Enter the element:1

    Main Menu:
    1.Insertion Sort
    2.Counting sort
    3.Exit
Enter the option:1
```



```
Enter the element:1

    Main Menu:
    1.Insertion Sort
    2.Counting sort
    3.Exit
Enter the option:1
1 2 3 5 5 8

    Main Menu:
    1.Insertion Sort
    2.Counting sort
    3.Exit
Enter the option:2
1 2 3 5 5 8

    Main Menu:
    1.Insertion Sort
    2.Counting sort
    3.Exit
Enter the option:3
```

Post lab questions-

a. **Compare and contrast various sorting algorithms.**



		Time Complexity					
		Best	Worst	Avg.	Space	Stable	Comments
Comparison Sort	Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*For each pair of indices, swap the elements if they are out of order*
	Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*At each Pass check if the Array is already sorted. Best Case-Array Already sorted*
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*Swap happens only when once in a Single pass*
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*Very small constant factor even if the complexity is O(n^2).* ***Best Case:*** *Array already sorted* ***Worst Case:*** *sorted in reverse order*
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	*Best Case: when pivot divide in 2 equal halves Worst Case: Array already sorted - 1/n-1 partition*
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	*Pivot chosen randomly*
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	*Best to sort linked-list (constant extra space). Best for very large number of elements which cannot fit in memory (External sorting)*
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	
Non-Comparison Sort	Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+2^k)	Yes	*k = Range of Numbers in the list*
	Radix Sort	O(n.k/s)	O(2^s.n.k/s)	O(n.k/s)	O(n)	No	
	Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	Yes	

b. **Comment on the input (sorted in ascending order/descending order/random) and time required for execution of sorting algorithm.**

<u>For count sort</u>

Input can be random
Let, N=Number of elements in given array K=maximum value present in given array.
Therefore, <u>Time Complexity</u> = O(N+K)
In all cases time complexity will be O(N+K) as the range of for loops does not changes.
<u>For Insertion sort</u>

Input can be random
<u>Time Complexity</u>= O (n^2) in all cases, where n is the number of elements in the array

Conclusion: - With the help of the above experiment, the various sorting algorithms have been successfully understood and implemented