Solved by: SHOROUK GABR AWWAD 30002030

Question 1:

(c) counting using indices like what is in count sort.

```
    int maxel = Maximum(arr,n);
     int minel = Minimum(arr,n);
     int range = maxel-minel;
     int counter[range+1];
//initiating the array
     for int i=0 to range
         counter[i]=0
//initiating the array
     For i = 0 to n
      counter[arr[i]-minel]++
     for i = 1 to range
         counter[i] += counter[i-1]
```

Solved by: SHOROUK GABR AWWAD 30002030

```
Bucket_sort (float arr[],int n)
{
    vector <float> A[n];
    int index;
    for (int i=0; i<n; i++)
    {
        index = floor(n*arr[i]);
        A[index].push_back(arr[i]);
    }
    for (int i=0; i<n; i++)
    {
        sort(A[i].begin(),A[i].end());
    }
    int ind=0;
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<A[i].size(); j++)
        {
            arr[ind++]= A[i][j];
        }
    }
}
```

The best case in bucket sort is when the elements are evenly distributed among the buckets. For example, if we have n elements in the array and their range is of k. then we will have 10 buckets, each bucket has around m elements are sorted by any mean of sort functions (insertion in most cases) of time complexity varies from log m to m^2 in worst cases (summed up in k), then we will have the total time complexity be O(kn)-> O(n).

On the other hand , the worst case in bucket sort is when all elements are in one bucket in a reversed order. In this case the time complexity is calculated like the following:

The first fir loop is repeated n times which mean its complexity of O(n)

The second for loop is repeated n times but it includes on a sort function has time complexity O(n^2) in the worst cases . while in the worst case, we suppose all the elements to be in one bucket. The sort will run only once which means the sort code is of complexity O(n^2)*1=O(n^2).

The last nested loop: the outer one will run n time and the inner one will execute only once since in all the other cases the condition for the inner one will be false. So in worst cases the complexity of the last part will be O(n)

To sum up: O(n^2) +O(n) +O(n) = O(n^2), then in worst cases the complexity will be of O(n^2).

f)

```
PointSort(A[n])
        dist =0
       timpA[n]
     for i=1 to n
                    dist= root((A[i].x-0)^2+(A[i].y-0)^2)
                    timpA[i]=dist
          vector  Arr[n]
     for i=0 to n
           index = floor (n*timpA[i])
              A[indext].push_back(A[i])
for i=0 to n
        sort(A[i].begin(),A[i].end())
     ind=0
    for i=0 to n
        for j=0 to Arr[i].size()
            A[ind++]= arr[i][j]
End function
```

Question two:

b)

Time complexity

Solved by: SHOROUK GABR AWWAD 30002030

```
void countingSortints (int arr[], int n, int k)
{
    //initiation
    int output[n];
    for (int i=0; i<n; i++)
    {
        output[i]=0;
    }
    //initiation
    int counter[10] = {0,0,0,0,0,0,0,0,0,0};

    for (int i = 0; i < n; i++)
        counter[(arr[i] / k) % 10]++;

    for (int i = 1; i < 10; i++)
        counter[i] += counter[i - 1];

    for (int i = n - 1; i >= 0; i--)
    {
        output[counter[(arr[i] / k) % 10]-1] = arr[i];
        counter[(arr[i] / k) % 10]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];

}



void radixSortints(int arr[], int n)
{
    int k;
    k= arr[0];
    for (int i=0; i<n; i++)
    {
        if (arr[i]>k)
        {

            k=arr[i];
        }
    }
    for (int i = 1; k / i > 0; i *= 10)
    {
        countingSortints(arr, n, i);
    }
}
```

Solved by: SHOROUK GABR AWWAD 30002030

For the first for loop the time complexity is O(n) and the time complexity of the second loop is O(10), and the time complexity is of O(n) and the third for loop is of time complexity O(n). the time complexity of counting sort as a total is O(kn)

in radix sort function, the time complexity of the first iteration (maximum value calculation) is of O(n) , while th time complexity of the second loop (the main loop) is of k*O(n), since counting sort (complexity of O(n)) is repeated k times(k is decreasing from an iteration to another tho, but still counted on the a constant)

the final complexity is of O(kn).

Space complexity:

the radix function has two auxiliary containers, The counter and the output. The counter container needs space of k (constant independent of number of entries) depends on the entered values themselves; sometimes k is equivalent to the range of the entered values. While the space of the other container, the output, depends on the number of entries—the space is always equal to the number of entries.

To sum up :

this implementation of radix sort is a combination of O(n)+O(k) = O(n+k)

d)

we use n as a base so every time we move from digit to another we move by n multiple like what we used for decimal sort in a, it goes the same and instead of having 10 as space for the count, we have n .

```
countSort(arr[],n, k)

   output[n]
// count of n elements
     count[n]
    for i=0 to n
       count[i] = 0

    for i=0 to n
// use n as a base
        count[ (arr[i]/k)%n ]++

    for i = 1 to n
        count[i] += count[i - 1]

    for n-1 to 0
```

```
        output[count[ (arr[i]/k)%n] - 1] = arr[i]
        count[(arr[i]/k)%n]--

    for i=0 to n
        arr[i] = output[i]

end of function

sort-function (Arr[], n)

void sort(int arr[], int n)

    countSort(arr, n, 1)

    countSort(arr, n, n)

    countSort(arr,n,n^2)

end of function
```