Algorithms and Data Structures

Problem 1

a)

At the beginning this is our array and we have the counting array set to 0

Array	9	1	6	7	6	2	1

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	0	0	0	0	0

Then we are going element by element:

Array	9	1	6	7	6	2	1
	↑						

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	0	0	0	1	0

Array	9	1	6	7	6	2	1
		↑					

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	0	0	0	0	0	0	0	1	0

Array	9	1	6	7	6	2	1
			↑				

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	0	0	0	0	1	0	0	1	0

Array	9	1	6	7	6	2	1
				↑			

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	0	0	0	0	1	1	0	1	0

Array	9	1	6	7	6	2	1
					↑		

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	0	0	0	0	2	1	0	1	0

Array	9	1	6	7	6	2	1
						↑	

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	1	0	0	0	2	1	0	1	0

Array	9	1	6	7	6	2	1
							↑

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	2	1	0	0	0	2	1	0	1	0

And now that we counted the elements we are going to start from the beginning of the counting array and lower the amount of elements till we get to 0 and thus putting that element in the output array.

Output	1						
	↑						

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	1	1	0	0	0	2	1	0	1	0

Output	1	1					
		↑					

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	1	0	0	0	2	1	0	1	0

Output	1	1	2				
			↑				

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	2	1	0	1	0

Output	1	1	2	6			
				↑			

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	1	1	0	1	0

Output	1	1	2	6	6		
					↑		

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	0	1	0	1	0

Output	1	1	2	6	6	7	
						↑	

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	0	0	0	1	0

Output	1	1	2	6	6	7	9
							↑

Count[i]	1	2	3	4	5	6	7	8	9	10
Count	0	0	0	0	0	0	0	0	0	0

And now we have the sorted array as 1, 1, 2, 6, 6, 7, 9.

b)

So we are starting the bucket sort with an array:

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1

n (number of elements) =7

So, we are creating n number of buckets, and then every element of arr[i] we are putting it in bucket nr. = n*arr[i]. Then we are sorting each bucket separately and concatenating all buckets back. And that would look like this:

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
	↑						

0.9*7=6.3

Bucket 0			
Bucket 1			
Bucket 2			
Bucket 3			
Bucket 4			
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
		↑					

0.1*7=0.7

Bucket 0	0.7		
Bucket 1			
Bucket 2			
Bucket 3			
Bucket 4			
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
			↑				

0.6*7=4.2

Bucket 0	0.1		
Bucket 1			
Bucket 2			
Bucket 3			
Bucket 4	0.6		
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
				↑			

0.7*7=4.9

Bucket 0	0.1		
Bucket 1			
Bucket 2			
Bucket 3			
Bucket 4	0.6	0.7	
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
					↑		

0.6*7=4.2

Bucket 0	0.1		
Bucket 1			
Bucket 2			
Bucket 3			
Bucket 4	0.6	0.7	0.6
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
						↑	

0.2*7=1.4

Bucket 0	0.1		
Bucket 1	0.2		
Bucket 2			
Bucket 3			
Bucket 4	0.6	0.7	0.6
Bucket 5			
Bucket 6	0.9		

arr	0.9	0.1	0.6	0.7	0.6	0.2	0.1
							↑

0.1*7=0.7

Bucket 0	0.1	0.1	
Bucket 1	0.2		
Bucket 2			
Bucket 3			
Bucket 4	0.6	0.7	0.6
Bucket 5			
Bucket 6	0.9		

And then after sorting the buckets we have this:

Bucket 0	0.1	0.1	
Bucket 1	0.2		
Bucket 2			
Bucket 3			
Bucket 4	0.6	0.6	0.7
Bucket 5			
Bucket 6	0.9		

And after the concatenation we are going to have our sorted result:

res	0.1	0.1	0.2	0.6	0.6	0.7	0.9

c)

```
def algorithm(arr,a,b):
  m=max(arr)
  count=[0]*(m+1)
  for i in range(len(arr)):
    count[arr[i]]+=1
  for i in range(1,m):
    count[i]+=count[i-1]
  return count[b]-count[a-1]
```

d)

The code for this algorithm can be found in “word.py”.

e)

Let’s say that our algorithm puts all numbers in the same bucket and then sort that bucket using insertion sort. Also, let’s say that we have those numbers in descending order and knowing the worst case for insertion sort we know that the complexity of that is going to be $\Theta(n^2)$.

f)

Let’s suppose nPoints, pointsX[ ], pointsY[ ], originX and originY are declared. Also we have allocated space for an array called dist[ ].

```
…

For n=0 to nPoints:

        Dist[n]=(sqrt((pointsX[n]-originX)**2+(pointsY[n]-originY)**2))

selectionSort(dist,pointsX,pointsY)

/*I used selection sort because we didn’t have any time complexity constrains*/


/*The declaration of the selection sort*/

selectionSort(dist,pointsX,pointsY){

for i to nPoints:

        min = dist[ i ]

        pos = i

        for j to nPoints:

                if min>dist[j]:

                        min=dist[j]

                        pos=j

        min,dist[i] = dist[i],min

        pointsX[i], pointsX[pos] = pointsX[pos], pointsX[i]

        pointsY[i], pointsY[pos] = pointsY[pos], pointsY[i]
```

Problem 2

a)

The algorithm for Hollerith’s version of the Radix Sort can be found in “modif_radix.py”.

b)

In this Hollerith's version of the Radix Sort we are starting with the most significant bit and we are going to the least significant bit. We are using a version of Bucket Sort for this algorithm and we are putting the numbers in separate buckets. When we would reach bucket of size 1 that means that there is only one element in that bucket so then when we would take elements for merging the buckets we would go in an order that it would sort them as it is going.

The best case for this Radix Sort would be the same as the best case for Bucket Sort and that would be if we had only one element in every bucket and we would have time complexity of Θ(n).

The worst case for this Radix Sort would be if we had array of elements of equal value and they would all fall into a same bucket on every bit division and that would make the time complexity of $\Theta(\mathrm{n} * \log k)$, where k would be the element that is repeating.

The space complexity of the best case would be Θ(n), because we are having n buckets. And the space complexity for the worst case would be Θ(n+k), where k would be the number of buckets which we are creating by every recursion call.

c)

Working with that big of a range, we should use Radix Sort, because all other sorts would either use too more or take more time to compute than that of Radix Sort.