Jonathan Rittmayer
Mat # 30001299

Algorithms & Data Structures
Homework #6

6.1a) Implement the algorithm for counting sort and then use it to sort the sequence < 9, 1, 6, 7, 6, 2, 1 >

We have our input array

1	2	3	4	5	6	7
9	1	6	7	6	2	1

← A

and our counting sort array (which we initialize with 0s)

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

← C

Now, for every occurrence of A_i, we increment $C[A_i]$ by one. After iterating through A, C turns into:

0	2	1	0	0	0	2	1	0	1
0	1	2	3	4	5	6	7	8	9

We then iterate throug C and perform these operations:

$C_i := C_i + C_{i-1}, \ \forall i, \ 0 \leq i \leq \text{length}[C]$

thus, our array becomes:

0	2	3	3	3	3	5	6	6	7
0	1	2	3	4	5	6	7	8	9

called array ↓

We then create a new array of size length[A], (B) which will be the sorted array. Since we will need the original array A we cannot replace it for sorting. Therefore we create B and at the end we copy B into A.

$$B[C[A[i]]] := A[i]$$

$$C[A[i]] := C[A[i]] - 1$$

$$\forall i \text{ where } 0 \leq i \leq \text{length}[A]$$

The following are the steps the algorithm takes to sort A:

1.

0	0	0	0	0	0	9

← output (B)

0	2	3	3	3	3	5	6	6	6

← C array

2.

0	0	0	0	0	7	9

← B

0	2	3	3	3	3	5	5	6	6

← C

3.

0	0	0	0	6	7	9

← B

0	2	3	3	3	3	4	6	6	6

← C

4.

0	0	0	6	6	7	9

← B

0	2	3	3	3	3	3	6	6	6

← C

5.

0	0	2	6	6	7	9

← B

0	2	2	3	3	3	3	6	6	6

← C

6.

0	1	2	6	6	7	9

← B

0	1	2	3	3	3	3	6	6	6

← C

7.

1	1	2	6	6	7	9

← B

0	0	2	3	3	3	3	6	6	6

← C

Jonathan Rittmayer
Mat # 300012299

Algorithms & Data Structures
Homework #6

6.1b) Implement the algorithm for Bucket sort and then use it to sort the sequence $< 0.9, 0.1, 0.6, 0.7, 0.6, 0.3, 0.1 >$

Array A $\rightarrow$

0.9	0.1	0.6	0.7	0.6	0.2	0.1

To sort this array with Bucket-sort, we need to create an array of lists (either linkedlist or array). Since they are uniformly distributed, we can multiply each element by length [A] in order to find its position in the array of lists.

0	$\rightarrow$	0.1	0.1	/
1	$\rightarrow$	0.2	/	/
2	$\rightarrow$	/	/	/
3	$\rightarrow$	/	/	/
4	$\rightarrow$	0.6	0.7	0.6
5	$\rightarrow$	/	/	/
6	$\rightarrow$	0.9	/	/

After this table we sort the lists individually and we get this $\longrightarrow$

0	$\rightarrow$	0.1	0.1	
1	$\rightarrow$	0.2		
2	$\rightarrow$			
3	$\rightarrow$			
4	$\rightarrow$	0.6	0.6	0.7
5	$\rightarrow$			
6	$\rightarrow$	0.9		

Then we iterate through the list, starting from the first list and concatenate them into a sorted array:

$A = [0.1, 0.1, 0.2, 0.6, 0.6, 0.7, 0.9]$

6.1c) Given n integers in the range 0 to k, design an algorithm with pre-processing time $\Theta(n+k)$ that counts in $O(1)$ how many of the integers fall into the interval $[a, b]$

```
Procedure
    B[k] ← array with all 0s
    For i = 0 up to n do
        B[A[i]] ← B[A[i]] + 1
    end For
    For i = 0 up to k do
        B[i] ← B[i] + B[i-1]
    end For
    return B[b] - B[a-1]
end procedure
```

6.1e) - Given any input sequence of length n, determine the worst case time complexity for Bucket sort. Give an example of a worst-case scenario an the prove corresponding the complexity

Assume that the elements given are uniformly distributed. However the range of our set of elements is very close to 0. In Other words, the difference between the maximum and the minimum element is very small. In this case, according to Bucket-Sort algorithm all of the elements will go in one bucket. If all elements end up in one bucket, then we will have N elements in one bucket. To individually sort that bucket, we need to use our best comparison sort algorithm. We know that the lower bound for comparison sort algorithms is $\Omega(n \log n)$. Thus the sorting will cost us $\Theta(n \log n)$. Total running time will be $\Theta(n + n\log n)$ which is the worst case.

6.1f) - Given n 2D points that are uniformly randomly distributed within the unit circle, design and write down an algorithm that sorts the points by increasing Euclidean distance to the circle's origin. Write also a pseudo code function for the computation of the Euclidean distance between two 2D points.

Procedure Sort2D(A, n) ▷ Assume that A is a list of structure which contains x and y components, but also d compent, which represents the Euclidean distance from the origin, initially undefined

```
B[n] ← {{0}}
for i = 1 up to n do
    A[i].d ← $\sqrt{(A[i].x)^2 + (A[i].y)^2}$
    idx ← n × A[i].d
    Insert A[i] into B[index]
end for
for i = 1 up to n do
    Sort B[i] by distance from origin (.d) with some fast comparison so
                                                                   algo
end for
    index ← i
for i = 1 up to n do
    while B[i] has elements do
        A[index] ← element from B[i]
        index ← index + 1
    end for
    return A
end procedure
```

Jonathan Rittmayer
Mat #30001299

6.2b) Determine and prove the asymptotic time complexity and the asymptotic storage space required for your implementation

Regular Radix-sort has an asymptotic time complexity of $\Theta(d(n+b))$ where $d = \log_b k$ and k is the maximum element in array to be sorted of length "n". The reason for "d" being a term in our equation is that regular Radix-sort starts from the least significant digit and iterates, to the most significant digit, therefore making "d" operations on the "counting sort" subroutine.

However, in Hollerith's version, the sorting uses bucket sort subroutine and starts from the most significant bit. If the difference between the numbers is big, it might not be neccessary to propagate throughout the digits. The reason for this is ~~that~~ that all numbers will fall into different buckets and a bucket of size 1 is an already sorted array (an array of size 1 is always sorted) In this case (best case) the algorithm has a running time complexity of $\Theta(n)$, same as bucket sort

- Let's consider the worst case, ie all the numbers in the array are the same. In that case, all the numbers will fall into the same bucket every time bucket sort is called (no matter the exponent) In this case the running time will be $\Theta(dn)$ where $d = \log_b k$ and k is the maximum element in the sorting array, b is the base.

- In average case, half the buckets will have either size 1 or 0 and the other half will have more than 1 elements in them. Therefore the total running time will be $\Theta(n + \frac{d}{2}n) \Rightarrow \Theta(\frac{d}{2}n) \Rightarrow \Theta(dn)$

- The space complexity for the best case is $\Theta(n)$, For Average cases and worst cases, "n" buckets are initialized, however they are dynamic, therefore no buckets exceed the size they need. But for every recursive call of bucket sort, new buckets are created. The recursive bucket sort can be called a max of "d" times, thus at most "dn" buckets can be created. Thus, the space complexity is $\Theta(dn)$.