Homework 5

Dushan Terzikj

due 20 Mar, 2018

1 Problem 1

a) Given the sequence [9, 1, 6, 7, 6, 2, 1], sort the sequence by Counting Sort.

Solution:

1	2	3	4	5	6	7	
9	1	6	7	6	2	1	←A

Let us define another array called *count* and initialize it with 0s.

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

Now for every occurrence of A_i, we increment $count[A_i]$ by one. After iterating through A, *count* becomes something like this:

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

Then we iterate through *count* and perform the following operation:
$count_i := count_i + count_{i-1}, \forall i, 0 \leq i \leq length[count]$
We should get an array like this:

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

At this point we create a new array of size $length[A]$, i.e., the sorted array. Since we need the original structure of A we cannot use it for sorting. Therefore we need to create a new array and then just copy it into A. Let's call the sorted array B. We have:

$$B[count[A[i]]] := A[i]$$
$$count[A[i]] := count[A[i]] - 1$$
$$\forall i, where 0 \leq i \leq length[A]$$

Here are the steps:

I	$output->$	0	0	0	0	0	0	9		
	$count->$	0	2	3	3	3	3	5	6	6	6
II	$output->$	0	0	0	0	0	7	9		
	$count->$	0	2	3	3	3	3	5	5	6	6
III	$output->$	0	0	0	0	6	7	9		
	$count->$	0	2	3	3	3	3	4	6	6	6
IV	$output->$	0	0	0	6	6	7	9		
	$count->$	0	2	3	3	3	3	3	6	6	6
V	$output->$	0	0	2	6	6	7	9		
	$count->$	0	2	2	3	3	3	3	6	6	6
VI	$output->$	0	1	2	6	6	7	9		
	$count->$	0	1	2	3	3	3	3	6	6	6
VII	$output->$	1	1	2	6	6	7	9		
	$count->$	0	0	2	3	3	3	3	6	6	6

b) Given the sequence $< 0.9, 0.1, 0.6, 0.7, 0.6, 0.2, 0.1 >$, sort the sequence by Bucket Sort.

Solution:

Let us denote array A:

$A \rightarrow$	0.9	0.1	0.6	0.7	0.6	0.2	0.1

In order to sort it with Bucket-Sort, we need to create an array of lists (either linked list 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	$->$	0.1	0.1	/
1	$->$	0.2	/	/
2	$->$	/	/	/
3	$->$	/	/	/
4	$->$	0.6	0.7	0.6
5	$->$	/	/	/
6	$->$	0.9	/	/

After we get this table, we sort the lists individually and we get something like this:

0	$->$	0.1	0.1	
1	$->$	0.2		
2	$->$			
3	$->$			
4	$->$	0.6	0.6	0.7
5	$->$			
6	$->$	0.9		

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

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

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

Solution:

```
1: procedure F(A, n, a, b)
2:     B[k] ← array with all 0s
3:     for i = 0 up to n do
4:         B[A[i]] ← B[A[i]] + 1
5:     end for
6:     for i = 0 up to k do
7:         B[i] ← B[i] + B[i − 1]
8:     end for
9:     return B[b] − B[a − 1]
10: end procedure
```

d) Given a sequence of n English words of length k, implement an algorithm that sorts them in $\Theta(n)$. Let k and n be flexible, i.e., automatically determined when reading the input sequence.

Solution:

Please check 'sort_words.cpp'.

e) Given any input sequence of length n, determine the worst-case time complexity for Bucket Sort. Explain your answer!

Solution:

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 $\Omega(nlgn)$, therefore the sorting will cost us $\Theta(nlgn)$. Total running time will be $\Theta(n + nlgn)$, which is the worst case.

f) Given n 2D points that are uniformly randomly distributed within the unit circle, design an algorithm (only pseudocode) that sorts the points by increasing Euclidean distance to the circle's origin.

Solution:

```
1: procedure SORT2D(A, n)        ▷ Assume that A is a list of structure which contains x and
   y components, but also d component, which represents the Euclidean distance from the origin,
   initially undefined
2:     B[n] ← {{0}}                                          ▷ Initialize buckets, list of lists
3:     for i = 1 up to n do
4:         A[i].d ← sqrt((A[i].x)^2 + (A[i].y)^2)
5:         idx ← n × A[i].d
6:         Insert A[i] into B[idx]
7:     end for
8:     for i = 1 up to n do
9:         Sort B[i] by distance from origin (.d) with some fast comparison sort algorithm
10:    end for
11:    index ← 1
12:    for i = 1 up to n do
13:        while B[i] has elements do
14:            A[index] ← element from B[i]
15:            index ← index + 1
16:        end while
17:    end for
18:    return A
19: end procedure
```

2 Problem 2

Consider Hollerith's version of the Radix Sort, i.e., a Radix Sort that starts with the most significant bit and propagates iteratively to the least significant bit (instead of vice versa).

a) Implement Hollerith's version of the Radix Sort.

Solution:

Please check 'hollerith_radix_sort.cpp'.

b) Derive the asymptotic time complexity and the asymptotic storage space required for your implementation.

Solution:

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 necessary to propagate through the digits. The reason for this is 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, i.e., 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 cases, half of the buckets will have either size 1 or 0, and the other half will have more than 1 element 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 maximum d times, therefore at most dn buckets can be created. Therefore, the space complexity is $\Theta(dn)$.

c) Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Solution:

Best way to sort n integers in such big range is to use *radix sort*. *Counting sort* will not work since it only works for integers in smaller range and *bucket sort* will not work since the numbers might not be uniformly distributed. Therefore, radix sort with average time and space complexity of $\Theta(nd)$, where $d = log_b k$ and b is the base of the numbers and k is the max element in the array to be sorted, is the solution to our question.