CS 301 - Assignment 2

Oytun Kuday Duran / 28357

November 9, 2022

1 Problem 1 (Order statistics)

1.1 (a) Sorting and returning k smallest elements

Merge sort can be used to sort the array of size of n. Then, we could find the k smallest number by iterating through the array. This algorithm would be T(n) = 2*T(n/2) + $\Theta(n)$ since in each recursion, we would create 2 subproblems of half size of the current recursions' size. To find best worst-case asymptotic time complexity of this algorithm, we could use masters theorem. If we take f(n) = $\Theta(n)$, a=2 and b=2, we would find f(n) = Θ(n) = n^{lna} = n. Therefore, the algorithmic complexity would be Θ(f(n)*log n) = Θ(n*log n). It matches with what we knew from previous lectures, merge sort takes Θ(n*log n) time in both worst, best and average cases.

After we sorted the array, we would need to find k smallest elements. We could simply iterate through the array for k times, which would take Θ(k) time. So, the total complexity would be Θ(n*log n + k). Since k is smaller or equal to n (size of array), in worst case, complexity would be Θ(n*log n + n) = Θ(n*log n).

1.2 (b) Order Statistics and returning k smallest elements

We need to find the k'th smallest element first and then find elements lower than k'th smallest element. For the finding k'th smallest element part, I would use the order statistics method we saw in the lectures with 5-medians pivot:

```
SELECT(i, n)
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the n/5 group medians to be the pivot.
3. Partition around the pivot x. Let k = rank(x).
if i = k then return x
else if i smaller than k
- then recursively SELECT the ith smallest element in the lower part
- else recursively SELECT the (i–k)th smallest element in the upper part
```

Analyzing first part:

The first step would take $\Theta(n)$ time since we would randomly partition the elements into groups of 5 and find medians.
The second step would take $\Theta(T(n/5))$ time.
Third step (Partitioning around pivot) would take $\Theta(n)$ time.
Last step, as discussed in class would take T(3n/4) time. So in total, it would be:
T(n) = $\Theta(n)$ + T(n/5) + $\Theta(n)$ + T(3n/4)

Using substitution method, there exists c>0 and n0>0 and n>n0 s.t. T(n) < c*n
Assuming T(k) <= c k for all k ¡ n:
T(n) = $\Theta(n)$ + T(n/5) + $\Theta(n)$ + T(3n/4)
T(n) = $\Theta(n)$ + cn/5 + bn + 3cn/4 (b>0) T(n) = 19cn/20 +bn
T(n) = $cn - (cn/20 - bn)$

$n0 > 1, c > 1, b < c/20$ conditions satisfy this similar to second quiz. We achieved something we want minus something residual.
We know that in worst case, T(n) = $\Theta(n)$ which shows that it runs in worst case. Therefore, with satisfying n0, c an b values, we can find the kth smallest element in linear time ($\Theta(n)$). Here, you can realise that T(n) is O(n) and $\Omega(n)$.

Analyzing second part:
We can find the smaller elements in linear time simply by getting previous elements of the arrays from last recursive steps since the way we are considering medians in the order statistics algorithms. After finding k smallest element by this way, we can simply sort those using merge sort. Since size is k, complexity would be $\Theta(kl * ogk)$ as explained in previous problem. So overall complexity is $\Theta(n) + \Theta(k * logk)$.

Which is better?
The first merge sort algorithm takes $\Theta(n)$ while the second algorithm using order statistics take $\Theta(n) + \Theta(k * logk)$ time. As k increase to n, second algorithms' complexity would be similar/same to first algorithm. However, on small k, second algorithm would have a complexity of $\Theta(n)$ which is linear and better than the merge sorting alone.

2 Problem 2 (Linear-time sorting)

2.1 (a) Modifying radix sort to sort strings:

First of all, to sort strings, we need to modify the auxiliary stable counting sort algorithm which radix sort uses to sort digits with almost linear time because of non-comparison method it uses. For the sake of simplicity, assuming that all characters are uppercase as shown in the question (Even if they were not, we could simply .toUpper() them and then reverse back). Since there is 26 available characters in English alphabet, the second array we use for count sort would be size 26 if they were in equal length. However, In terms of radix sort, because of the different lengths of strings, there is a problem in index ranges of strings since we start from the least significant character to the most significant character. To solve this, since we will compare based on characters, we could fill all the strings to be the equal length in order for them to be valid for sorting. To do this, since we sort based on ASCII value of characters, a character with a lower ASCII value than uppercase English alphabet characters would be necessary. I wouldn't also use any character might occur in the strings. We could use "@" ASCII value 64 to overcome this problem. Adding "@" to our second array that we use on count sort, our second array's (Auxiliary storage array) size would be of size 27 in such assumptions. Since "@" has a much smaller ASCII value than alphabet characters, it would be at the index 0 of the second array. After filling all shorter strings to match the length of the longest string, we could move on.

After these modifications, counting sort would simply initialize all 27 spots on storage array to 0, Iterate over the input array and increase indexes according to elements. Here, the index of element would be **asciiVal - 64** since "@" would correspond to 0 (64-64), "A" would correspond to 1 (65-64) etc. Then we would iterate over the auxiliary storage array like we would normally do and continue the algorithms as usual. At the end, after sorting, we would need to delete the "@" characters that we added at the end of shorter strings.

2.2 (b) Steps on input array consisting of: BATURAY", "GORKEM", "GIRAY", "TAHIR", "BARIS"

Firstly, we would add ”@” at the end of shorter strings to match the longest string ”BATURAY”, then the array would be: [“BATURAY”, “GORKEM@”, “GIRAY@@”, “TAHIR@@”, “BARIS@@”].

For the sake of simplicity, assume C is the extra array for counting sort, A as the input and B as the resulting array. Then, we would simply start from the rightmost character and start to sort accordingly:

Step 1: (7th characters of strings)
We would initialize C as [0,0,0,0....] (Size of 27)
Checking last characters we would observe: A= [’Y’,’@’,’@’,’@’,’@’]
Counting sort array C would be = [4,0,1,..,0]
After looping through array and doing addittion phase, C would be C=[4,4,5...,5]
We then would start from the end of C and update result array:
A = [”Y”, ”@”, ”@”, ”@”, ”@”] C = [4, 4, 5, 5, ..., 5]
B = [””, ””, ””, ””, ”Y”] C = [4, 4, 4, 5, ...,5]
B = [””, ””, ””, ”@”, ”Y”] C = [3, 4, 4, 5, ...,5]
B = [””, ””, ”@”, ”@”, ”Y”] C = [2, 4, 4, 5, ...,5
B = [””, ”@”, ”@”, ”@”, ”Y”] C = [1, 4, 4, 5, ...,5]
B = [”@”, ”@”, ”@”, ”@”, ”Y”] C = [0, 4, 4, 5, ...,5]
Current resulting array would be: [“GORKEM@”, “GIRAY@@”, “TAHIR@@”,“BARIS@@”,BATURAY”]

Step 2: (6th characters of strings)
We would initialize C as [0,0,0,0....] (Size of 27)
Checking 6th characters we would observe: A= [’M’,’@’,’@’,’@’,’A’]
Counting sort array C would be = [3,1,0,..,1,.,0]
After looping through array and doing addittion phase, C would be C=[3,4,4,..5,...5,5]
We then would start from the end of C and update result array:
A = [”M”, ”@”, ”@”, ”@”, ”A”] C = [3, 4, 4, 4, .,5,5,]
B = [””, ””, ””, ””, ”M”] C = [4, 4, 4, 5, .5..,5]
B = [””, ””, ””, ”A”, ”M”] C = [3, 4, 4, 5, ...,5]
B = [””, ””, ”@”, ”A”, ”M”] C = [2, 4, 4, 5, ...,5
B = [””, ”@”, ”@”, ”A”, ”M”] C = [1, 4, 4, 5, ...,5]
B = [”@”, ”@”, ”@”, ”A”, ”M”] C = [0, 4, 4, 5, ...,5]
Current resulting array would be: [“GIRAY@@”, “TAHIR@@”,“BARIS@@”,BATURAY”,“GORKEM@”]

Step 3: (5th characters of strings)
We would initialize C as [0,0,0,0....] (Size of 27)
Checking 5th characters we would observe: A= [’Y’,’R’,’S’,’R’,’E’]
Counting sort array C would be = [0,0,..1,0..2,0..,1,..1.0]
After looping through array and doing addittion phase, C would be C=[0..,1,..3..,3,...4..5]
We then would start from the end of C and update result array:
A = [’Y’,’R’,’S’,’R’,’E’] C = [0...,1,1... 3,3..,4..5,5,5]
B = [””, ””, ””, ””, ”Y”] C = 0...,1,1... 3,3..,4..4,5,5]
B = [””, ””, ””, ”S”, ”Y”] C = [0,...1, 3,3..,3..,4 , ...,5]
B = [””, ””, ”R”, ”S”, ”Y”] C = [0...,1, 2, 3, ...,5
B = [””, ”R”, ”R”, ”S”, ”Y”] C = [0...1,1...,0...4,..5, ...,5]
B = [”E”, ”R”, ”R”, ”S”, ”Y”] C = [0,,,,0,1,... 3.., 4.., 5.., ...,5]
Current resulting array would be: [“GORKEM@”,“TAHIR@@”,BATURAY”,“BARIS@@”,“GIRAY@@”]

Step 4: (4th characters of strings)
We would initialize C as [0,0,0,0....] (Size of 27)
Checking 4th characters we would observe: A= [’K’,’I’,’U’,’I’,’A’]
Counting sort array C would be = [0,1,..1,...2,..1,.,0]
After looping through array and doing addittion phase, C would be C=[0..,1,..2..,4,...5]

We then would start from the end of C and update result array:
A = ['K','I','U','I','A'] C = [0...,1,1... 2,2..,4.,5,5,5]
B = ["", "", "", "", "U"] C = 0.1..,1,1... 2,2..,4..4,5,5]
B = ["", "", "", "I", "U"] C = [0,1...1, 2,2..,3..,4 , ...,5]
B = ["", "", "I", "I", "U"] C = [0,1..,1, 2,.. 2, ...,5
B = ["", "K", "I", "I", "U"] C = [0,1..1,...,2...3,..5, ...,5]
B = ["A", "K", "I", "I", "U"] C = [0,0,1,... 2.., 3.., 5.., ...,5]
Current resulting array would be: ["GIRAY@@","GORKEM@","TAHIR@@","BARIS@@","BATURAY"]

Step 5: (3rd characters of strings)
Since the idea is very simple and repetitive, I will show shortly how the input gets affected:

We would repeat just like previous steps:
Checking 3rd characters we would observe: A= ['R','R','H','R','T']:
...
...

After count sort:
B = ["H", "R", "R", "R", "T"] C = [....]
Current resulting array would be: ["TAHIR@@","GIRAY@@","GORKEM@","BARIS@@","BATURAY"]

Step 6: (2nd characters of strings)

We would repeat just like previous steps:
Checking 2nd characters we would observe: A= ['A','I','O','A','A']:
...
...

After count sort:
B = ['A','A','A','I','O'] C = [....]
Current resulting array would be: ["TAHIR@@","BARIS@@","BATURAY","GIRAY@@","GORKEM@"]

Step 7: (1st characters of strings)
We would repeat just like previous steps:
Checking 1st characters we would observe: A= ['T','B','B','G','G']:
...
...

After count sort:
B = ['B','B','G','G','T'] C = [....]
Current resulting array would be: ["BARIS@@","BATURAY","GIRAY@@","GORKEM@","TAHIR@@"]

After removing the extra character that we added to equalize length, at the end, we would get :
.["BARIS","BATURAY","GIRAY","GORKEM","TAHIR"] which is sorted correctly.

2.3 (c) Analysis

Finding longest string takes Θ(n) time. Adding the extra char "@" to the end of shorter strings takes Θ(n.p) in worst case where p is the length of the longest string. So the addition of extra char takes Θ(n+n.p) = Θ(n) time.

During counting that is used in radix sort, initialization of first input array (Also I called as A) takes Θ(n) time. Initializing the counting sort array (I called as C) takes Θ(k) time where k is the max value in range (27 in our case since we substract 64). Counting the values in input array and modifying the counting sort array C takes Θ(n) time. After, summing the values in C takes Θ(k) time. Rechecking C from end to start and forming the sorted array takes Θ(n) time since we need to change values according to the size of input array. So counting sort part takes Θ(n+n+n+k) = Θ(n+k) time. We call the counting sort for every char which is the length of longest string(p), which totally takes Θ((n+k)*p) time. After sorting is done, clearing the extra character "@" takes Θ(n*p) which is same as inserting it.

All included, T(n)= Θ(n*p +p(n+k) + n*p) = Θ(d*(n+k)).

If we stay with our assumptions of uppercase English alphabet and 1 extra char (@), the k is 26+1=27 which is constant. So, we can get rid of it and conclude T(n) = Θ(p*n) where p is the length of longest string.