CS301 Algorithms-Assignment 2

Alper Kaan Odabaşoğlu - 28147

November 2022

1 Question 1

1.1 First sort the numbers using a comparison-based sorting algorithm, and then return the k smallest numbers.

Comparison based sorting algorithms: Insertion sort, bubble sort, selection sort, merge sort, heap sort and quick sort. We can use merge sort to sort n elements since it's one of the fastest comparison based sorting algorithms. I did not prefer to use other sorting methods like quick sort and insertion sort because in worst cases the quadratic behavior dominates the complexity which we do not want.

Merge sort recurrence relation: $T(n) = 2 \cdot T(n/2) + \Theta(n)$. Because we divide the array into two sub-arrays from the half of the array, we recursively process two n/2 elements. Thus we have 2T(n/2) in the equation. Also there is $\Theta(n)$ because we merge these arrays recursively up to the root array. At the end when we solve this recurrence relation with master's theorem:

Case 2 applies because f(n) = $\Theta(n) = \Theta(n^{log_2^2})$. Thus the complexity coming from the merge sort is T(n) = $\Theta(n \cdot logn)$. Because it's theta, we can say that the worst case is O($n \cdot logn$)

After we sort these n number with merge-sort, we can use for loop in our algorithm to store numbers up to k in a built-in array and we can return this array as intended. As it's just an iteration up to the number k, it has complexity O(k).

In total the complexity is O($k+n \cdot logn$). However we can also say that $k \leq n$ in all conditions because we choose a number k in a sorted array and it can be maximum n. So the complexity coming from our iteration can be maximum O(n). Thus the term O($n \cdot logn$) dominate our complexity relation and we can

say that the complexity of this overall algorithm directly depends on the best worst case running time complexity of merge sort which is O($n \cdot logn$).

1.2 First use an order-statistics algorithm to find the k'th smallest number, then partition around that number to get the k smallest numbers, and then sort these k smallest numbers using a comparison-based sorting algorithm.

To find k^{th} smallest number we can use worst case linear time order statistics which has an algorithm given in CS301-Algorithms lecture as:

1.Divide n elements into groups of 5. Find the median of each 5-element group by rote.
2.Recursively Select the median x of the $\lfloor n/5 \rfloor$
3.Partition around x, k= rank(x)
4.
if(i = k) then return x
else if i<k:
    then recursively select the i^{th} smallest element in lower part
else:
    recursively select the (i-k)th smallest element in upper part

Complexity analysis of this algorithm:
Dividing the n elements into group of 5 and finding their median costs $\Theta(n)$ steps. After that, recursive selection of medians across $\lfloor n/5 \rfloor$ elements cost T(n/5) in recurrence relation. After that we partition around the pivot which is $\Theta(n)$. Since at least half of the group medians will be $\leq x$ which is $\lfloor n/10 \rfloor$ group medians we can say that there will be at least $\lfloor 3n/10 \rfloor$ elements $\leq x$. As mentioned in Algorithms course when $n \geq 50 \Rightarrow \lfloor 3n/10 \rfloor \geq n/4$. Then the algorithm should cover $\leq 3n/4$ numbers to find the k^{th} smallest number. For that it adds T(3n/4) to our recurrence relation.
Total recurrence relation will: $T(n) = T(n/5) + T(3n/4) + \Theta(n)$
Our guess is that this recurrence will be $O(n)$. So lets check with substitution method:
What we will prove: $T(n) \leq c \cdot n$
Inductive proof:
Base Case: n = 1 where $T(1) \leq c \cdot 1$. We can pick c big enough to handle this equation.
Inductive hypothesis: $T(k) \leq c \cdot k$ where $\exists c > 0,\ 0 < k < n$.
Because n/5 and 3n/4 are smaller than n we can use inductive hypothesis:

$$\begin{aligned} T(n) &\leq c \cdot n/5 + c \cdot 3n/4 + c_1 \cdot n. \\ &\leq 19c \cdot n/20 + c_1 \cdot n \\ &\leq c \cdot n - (c \cdot n/20 - c_1 \cdot n) \\ &\leq c \cdot n \end{aligned}$$

We can pick c_1, c big enough and c_1 small enough to make residual part positive. Thus this equation is O(n). So we can find the k smallest element with linear complexity. To find the k smallest elements it is adequate to partition around this k smallest element and create an array by iteration. This partition step will also O(n). So, it does not affect the overall complexity since its still O(n).

In sorting part, again we can use merge sort to sort these k smallest numbers since it's one of the fastest algorithms amongst comparison-based algorithms. I did not prefer to use other sorting methods like quick sort and insertion sort because in worst cases the quadratic behavior dominates the complexity which we do not want. As it was analyzed above, the merge sort best asymptotic worst case running time complexity is $O(n \cdot logn)$ with n elements. As we have k elements, the best asymptotic worst case running time complexity is $O(k \cdot logk)$.

When we combine the order statistics part and sorting part, best asymptotic worst case running time complexity is $O(n + k \cdot logk)$. In this equation, it depends on n and k that which part of the equation will dominate. Thus we can keep the best asymptotic worst case running time complexity as $O(n + k \cdot logk)$.

I would prefer to use second algorithm because when k is smaller, the n dominates in the equation and the complexity will become linear which is better than the $O(n \cdot logn)$ coming from 1st algorithm. If k is closer to n than $k \cdot logk$ dominates and we can say that it will still be smaller than $n \cdot logn$ which we found in first algorithm. Since the complexity of the first algorithm is $O(n \cdot logn)$ and complexity of second algorithm depends on the behavior of k it will be logical to use the second algorithm. In each case it will be better option.

2 Question 2

2.1 Modification of Radix sort to sort strings

To sort strings with radix sort algorithm, we can use the ASCII values of the strings. Like in integers, we can start from least significant character (most right char) to most significant one (most left char). First of all, we can equalize the character count of strings by finding the maximum string length amongst the string set and adding particular amount of space character to the end of the strings one by one iteratively up to the point where all strings have equal number of characters. The reason behind this notion is starting from same least significant index in all strings to facilitate our sorting algorithm. Because the space character has an ASCII value equal to 32 and it's the smallest ASCII character that a string can directly contain without a manipulation, it will not affect the sorting mechanism. In that sense, we can sort the strings like "BATU"

and "BATUHAN". For instance, in "BATU" and "BATUHAN" example, we

can add 3 space characters after the most right character of "BATU" ("BATU ") and we can use radix sort starting from $6^{th} \Rightarrow 0^{th}$ indices afterwards.

Since radix sort utilizing auxiliary sort which is counting sort, we need to modify the counting sort as well. Our string can include 95 characters starting from ' ' (space char) to '~'. So we can store an array that includes 95 indices. While we are comparing the characters, and their ASCII values, we can subtract 32 from their ASCII value and then assign to the corresponding indices. For instance, space character has a value 32 in ASCII table and whenever we encounter this character, we subtract 32 from it (32-32=0) and assign it to the 0^th index. After that we will use counting sort mechanism with these decimal ASCII values. Counting sort steps:

(This part is also referenced from CS301 Algorithms counting sort slides)
Input array is A which has an index starting from 0 to number of strings - 1. C is an array starting from index 0 to 94. B is an output array with sorted strings and it has index starting from 0 to number of strings -1.
1- for i from 0 to 94
    do C[i] = 0
2- for j from 0 to number of strings - 1
    do C[A[j]] = C[A[j]] + 1
3- for i from 1 to 94
    do C[i] = C[i] + C[i-1]
4- for j from number of strings-1 to 0
    do B[C[A[j]] - 1] = A[j]

I use B[C[A[j]] - 1] in the last step since our array starts from the index 0.
After these alterations steps we can use radix sort on strings.

2.2 Illustrate how your algorithm sorts the following list of strings ["BATURAY", "GORKEM", "GIRAY", "TAHIR", "BARIS"]. Please show every step of your algorithm.

First of all, my algorithm will find the max length of these strings which is 7 ("BATURAY"). After that iteratively add spaces to strings up to the point their lengths are equal.

Pseudocode of this:
foreach string in array
    if string.length < max length
        add max length - string length " " (space) to string

So iteratively:
1st step: [“BATURAY”, “GORKEM”, “GIRAY”, “TAHIR”, “BARIS”].
2nd step: [“BATURAY”, “GORKEM ”, “GIRAY”, “TAHIR”, “BARIS”].
3rd step: [“BATURAY”, “GORKEM ”, “GIRAY ”, “TAHIR”, “BARIS”].
4th step: [“BATURAY”, “GORKEM ”, “GIRAY ”, “TAHIR ”, “BARIS”].
5th step: [“BATURAY”, “GORKEM ”, “GIRAY ”, “TAHIR ”, “BARIS ”].

After this step we can start sorting this array with our modified radix sort.

In all of the below steps, it is assumed that the radix sort is stable sort. The stability property of radix sort can be tracked below as well.

Radix sort steps:
for 6^{th} index: Inıtial A = [’Y’, ’ ’, ’ ’, ’ ’, ’ ’]
Inıtial C = [0, 0, 0, ... , 0] (There are 0-94 indices)
After iterating A and executing the step C[A[j]] = C[A[j]] + 1 $\Rightarrow$ C becomes [4,0,0, ..., 1 ...,0]. The 1 is in the 57th index because the ascii value of Y=89 and 89-32 = 57
After the step C[i] = C[i] + C[i-1] $\Rightarrow$ C= [4,4,4,....,5,5,5,5,5 .... ,5]

2.2.1 1st step For index = 6

B = [, , , ’ ’,] $\longrightarrow$ C= [3,4,4,....,5,5,5,5,5 .... ,5] and A[4] = ’ ’ (we started from last index of starting array)
B = [, ’ ’ , ’ ’ , ’ ’,] $\longrightarrow$ C= [2,4,4,....,5,5,5,5,5 .... ,5] and A[3] = ’ ’
B = [,’ ’ , ’ ’ , ’ ’,] $\longrightarrow$ C= [1,4,4,....,5,5,5,5,5 .... ,5] and A[2] = ’ ’
B = [’ ’ ,’ ’ , ’ ’ , ’ ’,] $\longrightarrow$ C= [0,4,4,....,5,5,5,5,5 .... ,5] and A[1] = ’ ’
B = [’ ’ ,’ ’ , ’ ’ , ’ ’, ’Y’] $\longrightarrow$ C= [0,4,4,....,4,5,5,5,5 .... ,5] and A[0] = ’Y’

After these steps our resulting array will be: [“GORKEM ”, “GIRAY ”, “TAHIR ”, “BARIS ”, “BATURAY”].

For other steps I will directly use C value without calculating again and again. However the calculation logic is same with above.

2.2.2 2nd step For index = 5

Starting A=[’M’, ’ ’, ’ ’, ’ ’, ’A’]
Starting C=[3,3,3 ..., 4,4,4, ... , 5, ...]
B= [, , , ,’M’] $\longrightarrow$ C = [3,3,3 ..., 4,4,4, ... , 4, ...] A[4] = ’M’
B= [, , ’ ’ , , ’M’] $\longrightarrow$ C = [2,3,3 ..., 4,4,4, ... , 4, ...] A[3] = ’ ’
B= [, ’ ’ , ’ ’ , , ’M’] $\longrightarrow$ C = [1,3,3 ..., 4,4,4, ... , 4, ...] A[2] = ’ ’

B= [' ' , ' ', ' ' , , 'M'] ⟶ C = [0,3,3 ..., 4,4,4, ... , 4, ...] A[1] = ' '
B= [' ' , ' ', ' ', 'A' , 'M'] ⟶ C = [0,3,3 ..., 3,4,4, ... , 4, ...] A[0] = 'A'

After these steps our resulting array will be: ["GIRAY ", "TAHIR ", "BARIS ","BATURAY" ,"GORKEM "].

2.2.3 3rd step For index = 4

Starting A=['Y', 'R', 'S', 'R', 'E']
Starting C=[0,0,0 .. 1, ... ,3,4, ... 5, ...]
B = ['E' , , , ,] ⟶ C=[0,0,0 .. 0, ... ,3,4, ... 5, ...]
B = ['E' , , 'R' , ,] ⟶ C=[0,0,0 .. 0, ... ,2,4, ... 5, ...]
B = ['E' , , 'R' , 'S' ,] ⟶ C=[0,0,0 .. 0, ... ,2,3, ... 5, ...]
B = ['E' ,'R', 'R' , 'S' ,] ⟶ C=[0,0,0 .. 0, ... ,1,3, ... 5, ...]
B = ['E' ,'R', 'R' , 'S' ,'Y'] ⟶ C=[0,0,0 .. 0, ... ,1,3, ... 4, ...]

After these steps our resulting array will be: ["GORKEM ", "TAHIR ", "BATURAY" , "BARIS ", "GIRAY "].

2.2.4 4th step For index = 3

Starting A=['K', 'I' , 'U', 'I', 'A']
Starting C=[0,0,0,0,0,...,1,1,1,...,3,3,4,...,5,...]
B = ['A', , , ,] ⟶ C=[0,0,0,0,0,...,0,1,1,...,3,3,4,...,5,...]
B = ['A', ,'I', ,] ⟶ C=[0,0,0,0,0,...,0,1,1,...,2,3,4,...,5,...]
B = ['A', ,'I', ,'U'] ⟶ C=[0,0,0,0,0,...,0,1,1,...,2,3,4,...,4,...]
B = ['A','I','I', ,'U'] ⟶ C=[0,0,0,0,0,...,0,1,1,...,1,3,4,...,4,...]
B = ['A','I','I','K','U'] ⟶ C=[0,0,0,0,0,...,0,1,1,...,1,3,3,...,4,...]

After these steps our resulting array will be: ["GIRAY ", "TAHIR ", "BARIS ", "GORKEM ", "BATURAY"].

2.2.5 5th step For index = 2

Starting A=['R', 'H' , 'R', 'R', 'T']
Starting C=[0,0,0,0,0,...,1,1,1 ...,1, 1,4,4,4,5,...]
B = [, , , , 'T'] ⟶ C=[0,0,0,0,0,...,1,1,1 ...,1, 1,4,4,4,4,...]
B = [, , ,'R','T'] ⟶ C=[0,0,0,0,0,...,1,1,1 ...,1, 1,3,4,4,4,...]
B = [, ,'R','R','T'] ⟶ C=[0,0,0,0,0,...,1,1,1 ...,1, 1,2,4,4,4,...]
B = ['H', ,'R','R','T'] ⟶ C=[0,0,0,0,0,...,0,1,1 ...,1, 1,2,4,4,4,...]

B = ['H','R','R','R','T'] ⟶ C=[0,0,0,0,0,...,0,1,1 ...,1, 1,1,4,4,4,...]
After these steps our resulting array will be: ["TAHIR ", "GIRAY ", "BARIS ", "GORKEM ", "BATURAY"].

2.2.6 6th step For index = 1

Starting A=['A', 'I' , 'A', 'O', 'A']
Starting C=[0,0,0,0,0,...,3,3,3,...,4,...,5,...]
B = [, ,'A', ,] ⟶ C=[0,0,0,0,0,...,2,3,3,...,4,...,5,...]
B = [, ,'A', ,'O'] ⟶ C=[0,0,0,0,0,...,2,3,3,...,4,...,4,...]
B = [,'A','A', ,'O'] ⟶ C=[0,0,0,0,0,...,1,3,3,...,4,...,4,...]
B = [,'A','A','I','O'] ⟶ C=[0,0,0,0,0,...,1,3,3,...,3,...,4,...]
B = ['A','A','A','I','O'] ⟶ C=[0,0,0,0,0,...,0,3,3,...,3,...,4,...]
After these steps our resulting array will be: ["TAHIR ", "BARIS ", "BATURAY", "GIRAY ", "GORKEM "].

2.2.7 7th step For index = 0

Starting A=['T', 'B' , 'B', 'G', 'G']
Starting C=[0,0,0,0,0,...,0,2,...,4,...,5,...]
B = [, , ,'G',] ⟶ C=[0,0,0,0,0,...,0,2,...,3,...,5,...]
B = [, ,'G','G',] ⟶ C=[0,0,0,0,0,...,0,2,...,2,...,5,...]
B = [,'B','G','G',] ⟶ C=[0,0,0,0,0,...,0,1,...,2,...,5,...]
B = ['B','B','G','G',] ⟶ C=[0,0,0,0,0,...,0,0,...,2,...,5,...]
B = ['B','B','G','G','T'] ⟶ C=[0,0,0,0,0,...,0,0,...,2,...,4,...]
After these steps our resulting array will be: ["BARIS ", "BATURAY", "GIRAY ", "GORKEM ", "TAHIR "].

As we can see, our algorithm work perfectly with this strings. The resulting array is sorted: ["BARIS ", "BATURAY", "GIRAY ", "GORKEM ", "TAHIR "].

2.3 Analysis of the modified algorithm.

It would be beneficial to analyze this algorithm step by step. In first step we decide on the maximum length amongst the strings. If there is n string exist in array, then finding the maximum will be O(n).

After that, for each string, we add space characters at the end of the strings

up to the point where all strings have equal number of characters which is the maximum string length. If the max length is m, then in worst case we need to add (m-1) space characters to (n-1) strings. This sumation will be done in a for loop. So the running time will be O((m-1)·(n-1)) = O(m·n) in worst case.

After adding spaces we use radix sort. In each step of radix sort we use counting sort and for the counting sort there are steps. In first step we allocate an array with k indices and assign 0 to each index. This step will cost O(k). In our algorithm k will always be 94. So O(k) = O(94) = O(1). After that for filling the C with char values from A, we iterate over A and it costs O(n). After this step, we iterate over C and add consecutive element values which will cost again O(k) = O(1) and last of all we create our resulting B array which costs again O(n) due to the iteration over input array. So, cumulatively O(1)+O(n)+O(1)+O(n) which will also O(n).

This step will be repeated m times for radix sort which is our maximum length. Thus, it costs O(n) · m = O(m·n). Combining the radix sort and space addition to the end of the strings, O(m·n) + O(m·n) = O(m·n). In the worst case m=n result in the complexity O(n^2). So the worst case running time become quadratic.