# CS301-A2

Görkem Yar (Student)

March 2022

## 1 Problem-1

### 1.1 Sorting and Finding the k Smallest Numbers

For the first part of this question, I would use merge sort to sort the array of size=n, after that I would find the first k smallest numbers. The algorithm of the merge sort is as follows: T(n) = 2*T(n/2) + $\Theta(n)$. Because in each iteration we create 2 subproblems of size 2 and we are examining all of the array. If we examine the best worst-case asymptotic time complexity of the merge sort by using master theorem: We can take f(n) = $\Theta(n), n^{\log_b a} = n^{\log_2 2} = \Theta(n)$ then since f(n) = $\Theta(n) = n^{\log_b a}$ the algorithmic complexity of the problem would be: $\Theta(f(n) * \log n) = \Theta(n * \log n)$

The next step is to find the k smallest numbers. This step will take $\Theta((k))$ time since the array is sorted and the only thing we are going to do is iterate till the element at position k.

The total algorithmic complexity of the problem is $\Theta(n \log n + k)$ since $k \leq n$ Complexity $= \Theta(n \log n + n) = \Theta(n \log n)$

### 1.2 Finding the k Smallest Numbers with Order Statistics Algorithm

The question is consist of two parts. Firstly, finding the K'th smallest elements and elements that are lower than this element. Secondly, sorting these k elements.

For the first part, I would use 5-medians pivot order statistics algorithm to find k'th smallest number. The intuition behind this algorithm is to find a better pivot to make complexity of our algorithm $\Theta(n)$ The algorithm works as follows:

**Step 1:** Divide n elements into group of 5. Find the median of each 5-element group by rote.
**Step 2:** Recursively SELECT the median x of the $\lfloor \frac{n}{5} \rfloor$ group medians to be the pivot.
**Step 3:** Partition around the pivot x. Let k = rank(x).
**Step 4:**

```
if i = k
  then return x
elseif i < 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
```

**Complexity Analyzation:**

**Step:1** This step will take $\Theta(n)$ time because we randomly divide n elements into group of 5 then find the median of 5 elements n/5 times.

**Step:2** Since we want to find median of the n/5 elements this will take $\Theta(T(n/5))$ time.

**Step:3** Partitioning around a pivot will take $\Theta(n)$.

**Step:4** Since in the step-2 we find the median of n/5 groups there are $\lfloor \frac{n}{10} \rfloor$ medians that $\leq$ than our pivot x. Also in each group, median is bigger or equal to 3 elements in their group. Therefore x $\geq 3 * \lfloor \frac{n}{10} \rfloor$ . Similarly, $x \leq 3 * \lfloor \frac{n}{10} \rfloor$ of the list. When we take n $\geq$ 50, $3 * \lfloor n/10 \rfloor$ will always $\leq \frac{n}{4}$. So, from now on we will assume that pivot-x is greater than $\frac{n}{4}$ elements in each iteration.

Since we are looking for the best, worst-case's Algorithmic complexity. We can assume that $\frac{3n}{4}$ (since $\frac{n}{4}$ of them is smaller) of the list is in each iteration bigger than our pivot x.

Therefore overall algorithmic complexity is: T(n)= $\Theta(n) + \Theta(T(\frac{n}{5})) + \Theta(T(\frac{3n}{4}))$.

If we use substitution method: Assume that $T(n) \leq c * n$

T(n)= $\Theta(n) + \Theta(T(\frac{n}{5})) + \Theta(T(\frac{3n}{4})) \leq \Theta(n) + \frac{3cn}{4} + \frac{cn}{5} = \Theta(n) + \frac{19cn}{20}$

$\rightarrow T(n) \leq c * n - (\frac{cn}{20} - \Theta(n))$

In which, we can find a large c that makes residue negative. Therefore T(n) is $O(n)$. Similarly, one can see that T(n) is also $\Omega(n)$ by just checking this part T(n)= $\Theta(n) + others$ because already this part is at least $\Omega(n)$

Since T(n) is O(n) and $\Omega(n)$, it is $\Theta(n)$.

**Result of the Second Algorithm:**Now, we find the k'th smallest element and the elements that are smaller than this element in this $\Theta(n)$ complexity. The second part of the problem is sorting k elements. To sort these elements I would use merge sort, which I explained complexity in the subsection 1.1. Then the comlexity of this sorting operation would be $\Theta(k \log k)$. As a result overall complexity would be $\Theta(n) + \Theta(k \log k)$

**Comparing Algorithms :** I would use second algorithm for finding the smallest k'th numbers because the first algorithm will always run in $\Theta(n \log n)$ without depending on k. On the other hand, the second algorithm will run in $\Theta(n) + \Theta(k \log k)$, if k is closer to n then their complexity is same whereas the second algorithm runs in $\Theta(n)$ time complexity for small k's.

## 2 Problem-2 (Linear-time sorting)

### 2.1 Radix Sort for Strings

**Modifications in Radix Sort:** To use Radix Sort, one needs an auxiliary stable algorithm which is the counting sort. To sort strings, we need to start from the most right (least significant) character and iterate to the most left (most important) character like the Radix sort for integers. However, there is an issue in the length of the strings because all of the strings do not have the same length; as a result, we cannot directly start from the least significant character. Firstly, we need to add arbitrary characters to the end of shorter strings. This arbitrary character should have precedence over all other characters. In our case, I am choosing this arbitrary character as +. For instance, if one of the strings is 'AYSU' and the longest string is 'DILARA', we will add two + characters to the end of 'AYSU' and it will became 'AYSU++'.

These were the modifications for Radix sort. Further we need to make some modifications in our auxiliary-stable sorting algorithm 'Counting Sort' as well.

**Modifications in Count Sort:** Auxiliary storage (array C) of the Counting sort should be size of 27 because there are 26 English characters assuming that all of them is either upper case or lower case and there is the character that we created arbitrary. So, 26+1 is 27. And following steps are:

**Step 1:** Like the normal counting sort all indexes of the array C should be initially 0 (+ element is at the index 0).

**Step 2:** Iterate over the input array A then increment corresponding indexes like the normal Counting Sort

**Step 3:** Iterate over array C such that each index will be count of that character and the count of characters which precede that particular character. (basically $C[i]+=C[i-1]$)

**Step 4:** Iterate over array A (reverse iteration) to find correct (by using array C, in each time we arrive some index of C we need to decrement that index for the following iterations) places for the output (B).

After these we can use Count and Radix sort for strings starting from least significant (most right) character.

## 2.2 How Radix Sort Works for String

First Array: [“MERT”, “AYSU”, “SELIN”, “ERDEM”, “DILARA”].

Firstly, we find the longest string in A which is 'DILARA' then modify other elements according to that.

Modified Array: [“MERT++”, “AYSU++”, “SELIN+”, “ERDEM+”, “DILARA”].

From now on, I will write the Counting Sort C after the first three steps of the counting sort then I will show the step 4 of counting sort one by one. Also, when I am showing the step 4 the array C that I will write next to array B is changed version of array C with respect to B (basically decreasing an index of C in step 4).

**Step 1: Take the right-most characters (6'th character):**

Input array A= ['+','+','+','+','A']

Counting Array C after the first 3 steps of counting sort: [4,5,.........,5]

```
        B=[   ,   ,   ,   , A] → C=[4,4,.....,5]
       B=[   ,   ,   , + , A] → C=[3,4,.....,5]
      B=[   ,   , + , + , A] → C=[2,4,.....,5]
     B=[   , + , + , + , A] → C=[1,4,.....,5]
    B=[ + , + , + , + , A] → C=[0,4,.....,5]
```

End of the this step is [“MERT++”, “AYSU++”, “SELIN+”, “ERDEM+”, “DILARA”]

**Step 2: 5’th character**

Input array A= [’+’,’+’,’N’,’M’,’R’]

Counting Array C after the first 3 steps of counting sort: [2,...,3,4,...,5,...]

```
        B=[  ,  ,  ,  , ’R’] → C=[2,...,3,4,...,4,...]
       B=[  ,  , M ,  , ’R’] → C=[2,...,2,4,...,4,...]
      B=[  ,  , M , N , ’R’] → C=[2,...,2,3,...,4,...]
     B=[  , + , M , N , ’R’] → C=[1,...,2,3,...,4,...]
    B=[ + , + , M , N , ’R’] → C=[0,...,2,3,...,4,...]
```

End of the this step is [“MERT++”, “AYSU++”, “ERDEM+”, “SELIN+”, “DILARA”]

**Step 3: 4’th character**

Input array A= [’T’,’U’,’E’,’I’,’A’]

Counting Array C after the first 3 steps of counting sort: [..1,..,2,..,3,..,4,..,5,..]

```
        B=[ A ,  ,  ,  , ] → C=[..0,..,2,..,3,..,4,..,5,..]
       B=[ A ,  , I ,  , ] → C=[..0,..,2,..,2,..,4,..,5,..]
      B=[ A , E , I ,  , ] → C=[..0,..,1,..,2,..,4,..,5,..]
     B=[ A , E , I ,  , U] → C=[..0,..,1,..,2,..,4,..,4,..]
    B=[ A , E , I , T , U] → C=[..0,..,1,..,2,..,3,..,4,..]
```

End of the this step is [“DILARA”, “ERDEM+”, “SELIN+”, “MERT++”, “AYSU++”]

**Step 4: 3’rd character**

Input array A= [’L’,’D’,’L’,’R’,’S’]

Counting Array C after the first 3 steps of counting sort: [0..,1,..,3,...,4,...,5,..,5]

```
        B=[  ,  ,  ,  , S] → C=[..,1,..,3,..,4,...,4,..]
       B=[  ,  ,  , R , S] → C=[..,1,..,3,..,3,...,4,..]
      B=[  ,  , L , R , S] → C=[..,1,..,2,..,3,...,4,..]
     B=[ D ,  , L , R , S] → C=[..,0,..,2,..,3,...,4,..]
    B=[ D , L , L , R , S] → C=[..,0,..,1,..,3,...,4,..]
```

End of the this step is [“ERDEM+”, “DILARA”, “SELIN+”, “MERT++”, “AYSU++”]

**Step 5: 2’nd character**

Input array A= [’R’,’I’,’E’,’E’,’Y’]

Counting Array C after the first 3 steps of counting sort: [..,2,..,3,..,4,..,5,..]

```
    B=[  ,  ,  ,  , Y] → C=[..,2,..,3,..,4,..,4,..]
   B=[  , E ,  ,  , Y] → C=[..,1,..,3,..,4,..,4,..]
  B=[ E , E ,  ,  , Y] → C=[..,0,..,3,..,4,..,4,..]
 B=[ E , E , I ,  , Y] → C=[..,0,..,2,..,4,..,4,..]
B=[ E , E , I , R , Y] → C=[..,0,..,2,..,3,..,4,..]
```

End of the this step is [“SELIN+”, “MERT++”,“DILARA”, “ERDEM+”, “AYSU++”]

**Step 6: 1’st character**

Input array A= [’S’,’M’,’D’,’E’,’A’]
Counting Array C after the first 3 steps of counting sort: [..,1,..,2,..,3,..,4,..5,..]

```
    B=[ A ,  ,  ,  , ] → C=[..,0,..,2,..,3,..,4,..5,..]
   B=[ A ,  , E ,  , ] → C=[..,0,..,2,..,2,..,4,..5,..]
  B=[ A , D , E ,  , ] → C=[..,0,..,1,..,2,..,4,..5,..]
 B=[ A , D , E , M , ] → C=[..,0,..,1,..,2,..,3,..5,..]
B=[ A , D , E , M , S] → C=[..,0,..,1,..,2,..,3,..4,..]
```

End of the this step is [“AYSU++”, “DILARA”, “ERDEM+”, “MERT++”, “SELIN+”]

Sorted Array after Radix is completed: [“AYSU++”, “DILARA”, “ERDEM+”, “MERT++”, “SELIN+”]

## 2.3 Running Time Analyzation

At the beginning of our Algorithm we find the longest string. Say the length of the longest string is m and there is n words. Finding the longest string and modifying other strings takes O(m*n) time.

After this, in each iteration of Radix Sort we used a count sort. First step of count sort will take O(k) times where the k is length of the Count Array C in our case it was 27. Second step of count sort will take $\Theta(n)$ time since we iterate all of the input array A. Third step of Count Sort will take O(k) time as well since we just modify the C array. The last step of count sort will take $\Theta(n)$ time as well since we iterate all of the input Array A and creating an output array B.

Total Algorithmic Complexity of the Count Sort is $\Theta(n + k)$ and we used count sort for m times which is the length of the longest string.

Total Algorithmic Complexity of the Radix Sort is O(m*n) + $\Theta(m * (n + k)) = \Theta(m * (n + k))$ .
( O(m*n) comes from the finding the longest string part)

For this example: n=5, k=27 and m=6 so k dominated n and the algorithmic complexity became $\Theta(m * k)$. However, for other cases when n is much bigger n will dominate k since k will not change, it will always be 27 because number of upper case + arbitrary element = 27. As a conclusion, we can say that if n is big enough the overall algorithmic complexity will be $\Theta(m * n)$.