Engincan Varan 25050

CS301 Homework 2

Question 1:

a. To sort N elements in a set, we can use a max heap. In order to create the max heap, we can use BuildHeap-heapify- function which takes O(n) time. After we create the max heap, we have a sorted list in a decreasing order. Then, we can use the GetMax() method which returns the maximum element in our maxheap and adjusts the heap. Since our maximum value will be on the index [1], GetMax() method will take O(1) time to return the maximum value but we need to maintain the heap properties, so the function also needs to fix this. This fixing operation would take -in worst case- O(logn) time. All in all, building the max heap would take O(n) time and getting the max element will take O(logn) time but we will call GetMax() function k times and add the elements into the result list. In the end, we will have a complexity which will be O(n) + O(klogn) time.

b. In this case we need to select the k^{th} biggest (or n-k^{th} smallest) number as our pivot for partitioning. To select k^{th} biggest element in our unordered list we can use Rand-Select() -QuickSelect()- functions to set our pivot. This would take for the best-case O(n) time, but in the worst case it would take O(n^2). However, using sorting based select would be better since it will have O(nlogn) time not matter what by sorting the list first than taking the k^{th} element. Anyway, after we choose our k^{th} biggest element we can use this number as our pivot in partitioning function. The partitioning function will divide our list into 2 parts from the pivot where left part is the smaller numbers and the right ones are bigger than pivot. The partitioning function would take O(n) time since it traverses the list just once. After the partitioning, we can get the right part of the list with our pivot, which would bring us the k^{th} biggest elements in an unordered list. Up until now we always had unordered lists. At the end we can use an efficient comparison-based sort algorithm such as MergeSort or QuickSort. Sorting the last k^{th} biggest element in these sorting algorithms would take O(klogk) time. In memory wise it depends on the sorting algorithm we use. I would prefer MergeSort since the worst-case is O(klogk) but it requires O(k) memory. To sum up, first we need to select the k^{th} biggest element in our unordered list by using Rand-Select function which takes O(n) for its average case. After that we need to partition the list by setting the pivot we selected from Rand-Select function, this would take O(n) time. After all of this, we will end up with k biggest elements in an unordered list. Then we need to sort this list by using MergeSort which takes O(klogk) time and O(k) memory. In the end, we will have O(n) + O(n) + O(klogk) as our total time complexity which is essentially O(n+klogk). -O(nlogn) + O(n) + O(klogk) if we use sorting base selection-

All logarithms are in base 2.

Which method is better?

We have 2 different algorithms to find the kth biggest elements in a given list of n elements. One of them takes O(n+klogn) time whereas the other one takes O(n+klogk) time. As we can observe if “k” is very close to “n” we will have the same result at the end but if k is not close enough we have a candidate. To compare, first they both take O(n) time at first then they have a “k” time something. The only difference is the “logn” and “logk”. So, if “k” is much smaller than “n”, “logk” is also much smaller than “logn” and it will reveal that the second algorithm, which uses order statistics and partitioning, is better than the first one. For example, if we want “n-2”th biggest elements in the list of n elements, both algorithms take nearly the same time, there won’t be such difference to separate them. However, taking some number much smaller than “n”, second algorithm is more efficient.

Question 2:

a. First, we need to pick a range to sort the elements. For example, in integer Radix Sort the range is 0-9. To do this we can use all lowercase characters or uppercase characters. Since the example given to us requires uppercase characters our range will be ‘A’ – ‘Z’. After that we need an element to add to the short strings to make all strings equal length. Normally, integer Radix Sort add “0” to the beginning of the integers to make them same length since 0748 is the same as 748. To modify this to our string Radix Sort, we can use ‘*’ since in ASCII table it appears before the alphabet. Also, this time instead of putting ‘*’ at the beginning of the string we are going to put that at the end of the string. For example, instead of ‘apple’ we will write ‘apple**’ if we want all the strings have length 7. After that we don’t have to change anything at all, we can start checking the strings from the end and apply the algorithm normally. Picking the indices and putting them into the buckets that we have for each character, then empty the buckets and go on until we are at the first index. Modifications are as follows:
   i. Instead of the range 0 – 9, we will have ‘A’-‘Z’ or ‘a’-‘z’ + ‘*’. (total 27 buckets)
   ii. We will add ‘*’ at the end of the strings to make them all equal length, before we begin the sorting.
   iii. Start from the last index and continue to apply the algorithm.

All logarithms are in base 2.

b. [VEYSEL , EGE , SELIN , YASIN] is the given list.
   i. Prepare the buckets. (In this case an array sized 27, from ‘A’ to ‘Z’ + ‘*’.
   ii. First find the maximum length. Then add ‘*’ to smaller string’s end.
      a) [VEYSEL , EGE*** , SELIN* , YASIN*]
   iii. Choose the strings according to their last index and add to the buckets.
      a) [VEYSEL , EGE*** , SELIN* , YASIN*]
      b) Adding to the list according to the character we checked and the index that character has. Then, extract from the buckets from the beginning.
      c) [EGE*** , SELIN* , YASIN* , VEYSEL] → returned version of the list.
   iv. Continue to do the step III till the first index.
   v. This is how the algorithm works.

[VEYSEL , EGE , SELIN , YASIN] -> Given list

[VEYSEL , EGE*** , SELIN* , YASIN*] -> ‘*’s are added

[EGE*** , SELIN* , YASIN* , VEYSEL] -> Order according to their last index (*,*,*,L)

[EGE*** , VEYSEL , SELIN* , YASIN*] -> Order according to their length – 1 index (*,E,N,N)

[EGE*** , SELIN* , YASIN* , VEYSEL] -> Order according to their length – 2 index (*,I,I,S)

[EGE*** , SELIN* , YASIN* , VEYSEL] -> Order according to their length – 3 index (E,L,S,Y)

[YASIN* , SELIN* , VEYSEL , EGE***] -> Order according to their length – 4 index (A,E,E,G)

[EGE*** , SELIN* , VEYSEL , YASIN*] -> Order according to their first index (E,S,V,Y)

[EGE , SELIN , VEYSEL , YASIN] -> Resulting list, sorted in increasing order

c. Before the algorithm begins, it should check for the longest string in the given list. It would take O(n) time. After that we need to add ‘*’(s) at the end of the smaller strings. This would take O(n) time since it needs to iterate over each element and if the string’s length is not enough it will add ‘*’. After the modifications are done we can start taking the strings according to the indices we want. This operation will take O(n) time because it will iterate over each element again, also the algorithm needs to check every indices that strings has so the length of the strings are also a part of time complexity. Let k denote the maximum length of the strings, then we will have O(kn) time. To sum all this up, O(n) + O(n) + O(kn) will be the final time complexity which is O((k+2) n) which is essentially means O(kn) where k is the maximum length of the strings. Since k cannot be very large like million or billion we can basically say the algorithm takes O(n) time.

All logarithms are in base 2.