CS301 Assignment 2

Alpay Naçar

October 30, 2023

1 Problem 1

1.a

Best possible worst-case time complexity we can get from a comparison-based sorting is O(nlogn) because when we take a look at the decision tree of that algorithm, there should be n! leaves and this tree has less than or equal to 2^h leaves, and when we simplify $n! \leq 2^h$, it becomes h=Ω(nlogn). In decision trees, h shows worst case time complexity of that algorithm.

For the first part of the question, I would use merge sort. In merge sort, we will call two recursive functions with half the size and merge them. Merging part will be linear time. Therefore we can write it as T(n) = 2*T(n/2) + O(n). Using master theorem we can find that T(n) = O(nlogn).

After sorting, returning k elements will take O(k) time. We will just iterate through k elements.

Asymptotic complexity of this algorithm is O(nlogn + k).

1.b

We can use median of medians with groups of 5 for order statistics. This algorithm involves several steps (let's call this algorithm T(n)), first dividing all elements into groups of 5 which takes O(n) time, then recursively selecting median of, medians we found in previous step which takes T(n/5) time, then partition around the pivot which takes O(n) time, then check rank of median of median we found, (if it is the rank that we are searching for, just skip to sorting k elements part) if its not in the rank that we are trying to get, call function recursively (with updated k value) to get the kth element, and this last step will be T(3n/4) time, because there is $\lfloor n/5 \rfloor$ groups and half of these groups medians are bigger than median of median, therefore $\lfloor n/10 \rfloor$*3 elements is definitely bigger than median of median, similarly $\lfloor n/10 \rfloor$*3 elements are smaller than the median of median. In worst-case scenario, rest of the elements, are in the same group (either all bigger than median of median or all bigger than median of median), which results in bigger part being n - $\lfloor n/10 \rfloor$*3. We can use 3n/4 because it is already bigger than 7n/10. If we use substitution method, we can find that this whole algorithm is O(n).
Substitution method: assume that: T(n) ≤ c*n for all k smaller than n, it satisfies
T(n) = O(n) + T(n/5) + O(n) + T(3n/4)
T(n) ≤ O(n) + c*n/5 + O(n) + c*3n/4 = O(n) + 19c*n/20 ≤ c*n
T(n) = O(n)

The order statistics part will be O(n). Then sorting (we can use merge sort again for that) this k element will be O(klogk). After that, getting the last k element will be O(k). Therefore finding kth element, partitioning around kth element, and sorting k elements will be O(n + klogk) time.

Which method would you use if k is $\lceil lgn \rceil$? Please explain why.

If k equals to $\lceil lgn \rceil$ (in worst case we can say logn+1 and it will become logn in big-o), first method (sorting the whole array) with O(nlogn + k) will become O(nlogn) because nlogn dominates logn. Second method (order statistics) with O(n + klogk) will become O(n) because n dominates $logn*log^2n$. Therefore it is more efficient to use first solution (order statistics), instead of second solution (sorting the whole array). While sorting the whole array we spend some unnecessary time sorting elements that we are not concerned about.

2 Problem 2

2.a

First I need to prioritize properties, for simplicity, let's say that priority decreases from left to right in the given order. id, character, color, size, and texture are the prioritized respectively.

We need to use radix sort for each property, each radix sort will use that property only and does counting sort for each character of that property. This will work because we know that radix sort is a stable sort algorithm.

To make all strings equal-sized, so that we can use counting sort for them, we can put a symbol, let's say * at the end of strings till all strings of that property become same sized, so that smaller strings will become smaller.

2.b

Initially: [⟨7, bird, blue, small, soft⟩,⟨4, fish, red, medium, hard⟩,⟨3, bear, blue, big, soft⟩,⟨6, rabbit, red, small, hard⟩,⟨5, fish, blue, medium, soft⟩]

After editing properties (to make their size the same): [⟨7, bird**, blue, small*, soft⟩,⟨4, fish**, red*, medium, hard⟩,⟨3, bear**, blue, big***, soft⟩,⟨6, rabbit, red*, small*, hard⟩,⟨5, fish**, blue, medium, soft⟩]

After first Radix Sort (by texture): [⟨4, fish**, red*, medium, hard⟩,⟨6, rabbit, red*, small*, hard⟩,⟨7, bird**, blue, small*, soft⟩,⟨3, bear**, blue, big***, soft⟩,⟨5, fish**, blue, medium, soft⟩]

After second Radix Sort (by size): [⟨3, bear**, blue, big***, soft⟩,⟨4, fish**, red*, medium, hard⟩,⟨5, fish**, blue, medium, soft⟩,⟨6, rabbit, red*, small*, hard⟩,⟨7, bird**, blue, small*, soft⟩]

After third Radix Sort (by color): [⟨3, bear**, blue, big***, soft⟩,⟨5, fish**, blue, medium, soft⟩,⟨7, bird**, blue, small*, soft⟩,⟨4, fish**, red*, medium, hard⟩,⟨6, rabbit, red*, small*, hard⟩]

After fourth Radix Sort (by character): [⟨3, bear**, blue, big***, soft⟩,⟨7, bird**, blue, small*, soft⟩,⟨5, fish**, blue, medium, soft⟩,⟨4, fish**, red*, medium, hard⟩,⟨6, rabbit, red*, small*, hard⟩]

After fifth Radix Sort (by id) (Optional Step, I am not sure if id is something we want to sort accordingly or not): [⟨3, bear**, blue, big***, soft⟩,⟨4, fish**, red*, medium, hard⟩,⟨5, fish**, blue, medium, soft⟩,⟨6, rabbit, red*, small*, hard⟩,⟨7, bird**, blue, small*, soft⟩]

2.c

Let's say that p is property amount (in this given example p is 4, character, color, size, texture), and s is the maximum size of strings, and c is the number of characters a character can get (in our example they are all lowercase english letters, so we can say that it is 26), n is amount of elements given (in this example there is 5 elements given). There will be p radix sorts, and every radix sort runs s counting sorts, and each counting sort runs on O(n + c) time. Therefore all of this algorithm runs in O(p*s*(n+c)).