Taner Giray Sönmez
2023, Fall

CS 301: Algorithms
Assignment 2

Problem 1

Part (a):

Using merge sort, I can first sort the elements of the array and then easily select the kth smallest element in the array by selecting the kth element from the sorted array. The reccurence relation for the merge sort when we divide array into almost two equal size: $T(n) = 2T(n/2) + \theta(n)$. We can prove it by using master theorem:
$a = 2 \geq 1, b = 2 \geq 1$, f(n) = $\theta(n)$ (asymptotically positive)
We need $n^{\log_b a}$:
$n^{\log_b a} = n^{\log_2 2} = n$
We need to compare $n^{\log_b a}$ with f(n):
f(n)=$\theta(n) = \theta(n^{log_b a})$
Case 2 applies, therefore:
T(n) = $\theta(n^{log_b a} logn) = \theta(nlogn)$
After sorting the array, returning the kth smallest element will take constant time which is in this case $\theta(k)$. Combining all together, total time will be $\theta(nlogn + k)$. Since $k \leq n$ we can also simply say that it is $\theta(n * logn)$.

Part (b):

I would use linear time order statistics with the "median of the medians algorithm". The steps of the algorithm are:

```
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 $\lfloor n/5 \rfloor$ group medians to be the pivot.
3-Partition around the pivot x. Let k = rank(x)
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
```

If we analyze time complexity, step 1 of the algorithm which is dividing n elements in the groups will take $\theta(n)$ time. Moving to step 2, selecting the median x of these groups recursively as a pivot will take T(n/5) time. Natural partition in step 3, takes $\theta(n)$ time. For the step 4, we should examine carefully, if we divide n elements into group of fives then we will get n/5 elements in each group. To find smaller elements in one side and bigger elements in other side we need to find median of these groups which will be (n/5)/2 = n/10. Then, we need to find how many elements we will have that are smaller than median of medians. We are aware that, for each group, at least three values fall between or equal to the median for that group. Hence, this is n/10 * 3 which is 3n/10. Again, the number 3 comes from the number of elements in one group that are smaller than equal the median. For the number of elements which are greater than median of medians is then 7n/10 because n-3n/10 = 7n/10. For the worst case, we should consider the bigger part of the partition which is 7n/10 in this case. Therefore the the time of step 4 will be T(7n/10).(For simplicity, I removed the floor operators) To sum up, total time is:
T(n) = $\theta(n)$(coming from step1) + T(n/5)(coming from step2) + $\theta(n)$(coming from step3) + T(7n/10)(coming from step 4).Hence, T(n)= T(n/5) + T(7n/10) + $\theta(n)$.
Let's solve this proof by induction:

Step 1 : Claim that T(n) =$O(n)$
Step 2 : Use induction to prove T(n) = $O(n)$
In order to prove this, we need to show: $\exists c, n_0 \geq 0$ such that $\forall n \geq n_0 : T(n) \leq cn$
Induction base: $T(1) \leq c * 1 \leq c$
We can always pick c big enough to satisfy this

Inductive step:
Assume that $T(k) \leq ck$ for all k < n (induction hypothesis)
Show that T(n) $\leq cn$
T(n)= T(n/5) + T(7n/10) + $\theta(n) \leq cn/5 + 7cn/10 + \theta(n)$
T(n)$\leq 9cn/10 + \theta(n) \leq (?)\ cn$
T(n)$\leq cn - (cn/10 - \theta(n)) \leq cn$, where $(cn/10 - \theta(n))$ is positive when we pick large enough c
Hence T(n) $\leq cn$ which means that $\mathbf{T(n) = O(}n\mathbf{)}$
After using order statistics, we can use merge sort for partitioning and sorting these numbers which takes $O(klogk)$ as I explained in the first part. Then the total time complexity will be O(n+klogk).

If k is $\lceil \log n \rceil$, then it is relatively small compared to n, and it is typically much smaller than $nlog(n)$. With that sense, part b would be more preferable and efficient since the order statistics algorithm takes O(n) which is faster than sorting the entire set in O(nlog(n)).In option (b), when we pick the k smallest numbers after partitioning, we're dealing with a smaller chunk of the data (k elements) rather than sorting the entire set. This makes it faster. So, if k is about the same as the logarithm of the total number of elements ($\lceil \log n \rceil$), option (b) is a better choice because it's more efficient for this specific situation.

Problem 2

Part (a):

To modify radix sort to lexicographically sort a list of tuples, we basically need to consider each element of the tuple like a digit in radix sort. Here are the modifications:

1. When comparing elements of tuples, if they are strings, we start the sorting process from the rightmost character, which is the least important character. Given that the available digits in a typical radix sort for numerical values are between 0 and 9, an auxiliary array with 10 elements will do. But because the English alphabet consists of 26 letters, there are 26 possible ways to order strings for each character. As such, we have to modify the auxiliary array's size appropriately. Further modification is required since the input may comprise strings of different lengths. In other words, we first make sure that the length of each string is the same before starting the comparison procedure. To ensure correct alphabetical order when comparing strings of different lengths, we can append a character like '+' to the end of the shorter strings. The '+' character is chosen because it precedes all alphabetic characters in ASCII values, preserving the original order. On the other hand, if elements are integers, we would directly sort them as in the original radix sort.Until all passes are finished, we should repeat the matching radix sort for each element, determining whether it is numerical or a string and going from the least to the most significant bit. The outcome of this thorough sorting method is that the tuples are lexicographically sorted.
2. Since we do not have least significant bit, we would compare them starting from the last elements of tuples.
3. We must sort the tuples using a stable sorting algorithm in each pass because it maintains the relative order of elements that are equal, a stable sorting algorithm is essential. With that the lexicographic order of the elements is guaranteed during the sorting process.

Part (b):

In the algorithm, we start sorting by the least significant element which is texture.Since they are strings, we first find the maximum length of the strings. Shorter strings are padded with '+' to make them equal length. We then perform count sort for each character column, starting from the last, using an auxilary space of size 26 for each letter.The rest of the sorting process is similar to that for integers(see below). The most significant bit "ids" which will be ordered in the last pass are integers. We use a count sort method with an auxiliary space of size 10 (for digits 0-9). After counting the occurrences of each digit, we convert the counts into cumulative counts, which indicate the position of each element in the sorted output. We then place each digit in a new array according to these positions, decreasing the count each time an element is placed. This process is repeated for each digit from least to most significant. After these steps, we got the final sorted list of tuples(Note that it preserves the input order among equal elements).
This is the initial list:

[⟨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⟩]
1. Sort by texture(considering as the least significant bit):
[⟨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⟩]]
2. Sort by size(considering as the second least significant bit):
[⟨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⟩,]
3. Sort by color(considering as the third least significant bit):
[⟨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⟩]
4. Sort by character(considering as the fourth least significant bit):
[⟨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⟩]
5. Sort by id(considering as the fifth least significant bit):
[⟨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⟩,]
Hence this is the final list.

Part (c):

Analyzing the radix sort algorithm's running time for tuples involves treating each member of the tuple throughout the sorting procedure. The time complexity of the modified algorithm for lexicographically sorting a list of tuples is influenced by three main factors: the number of tuples (n), the number of elements in each tuple (a), and the maximum length of the elements (b). The algorithm performs a series of passes equal to the number of elements in each tuple. If there are 'a' elements in each tuple, the algorithm will go through 'a' passes for sorting. The length of strings or the number of digits in the elements also affects the running time. The algorithm performs a count sort for each digit or character to sort tuples based on a single element. The number of count sorts performed by the algorithm for each element is determined by the maximum length of the elements (b). Therefore, the time complexity of a single pass (i.e., sorting tuples based on one element) is O(n*b), where 'n' is the number of tuples, and 'b' is the maximum length of elements. Since the modified algorithm applies this process for each element and has 'a' passes, the overall time complexity can be approximated as O(n * b * a). This means that the time complexity scales linearly with the number of tuples, the number of elements, and the maximum length of elements.