CS301-HW2

Ahmet Furkan Ün

October 29, 2023

Problem 1: Order Statistics

(a)

In the lecture we proved with binary tree that the best possible asymptotic worst-case running time of a comparison based sorting algorithm is $\Theta(n*logn)$. This time can be achieved by using different sorting algorithms but I want to use Merge Sort because of its stability and reliable performance. It accomplishes this through a recursive process where the array is divided into smaller, equal sized sorted subarrays, and then merged together. The recurrence relation for its time complexity, $T(n) = 2*T(n/2)+\Theta(n)$, reflects the need to sort two subarrays with length $n/2$ and merge them, with the merging step contributing $\Theta(n)$ time.

We can find time complexity of this recurrence with Master theorem. $T(n) = aT(n/b) + f(n)$, so $a = b = 2$ and $f(n) = \Theta(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 2})$. Therefore, time complexity will be $\Theta(f(n) * logn) = \Theta(n * logn)$. After sorting all the elements, it will take $\Theta(k)$ time to return the first k elements to get smallest k elements.Then, total time complexity of algorithm will be $\Theta(n*logn+k)$. We know that $k \leq n$, therefore we can express the time complexity as $\Theta(n*logn)$

(b)

To determine the k'th smallest number in an array, the Median of Medians order statistics algorithm can be useful. First, the array is divided into groups of 5 elements, creating approximately $n/5$ groups. This initial division takes $\Theta(n)$ time. Within each group, the median is identified. Then, the median of these group medians is selected recursively to serve as the pivot, denoted as x, which takes $T(n/5)$ time. Subsequently, a partitioning step is carried out around this pivot taking $\Theta(n)$ time. We know that there are $\lfloor n/5 \rfloor$ groups and group medians. At least half of the medians are less then or equal to the pivot x, which makes $\lfloor n/10 \rfloor$ group medians. For each group we know that at least 3 values are less then or equal to the group median. Therefore there are at least $\lfloor 3*n/10 \rfloor$ elements that are $\leq x$ and at most $\lfloor 7 * n/10 \rfloor$ elements that are $\geq x$. Thus, the recurrence for running time can assume that selection of k'th smallest number recursively takes time $T(3n/4)$ in the worst case. Considering all this times, the recurrence relation is:
$T(n) = T(n/5) + T(7n/10) + \Theta(n)$
By substitution method we can calculate the time complexity. We need to show that $T(n) \leq cn$.
$T(n) \leq cn/5 + 7cn/10 + \Theta(n) = 9cn/10 + \Theta(n)$
$= cn - (cn/10 + \Theta(n))$ (for large enough c, $cn/10$ dominates $\Theta(n)$)
$\leq cn$ therefore $T(n) = \Theta(n)$ After finding the k'th smallest element, sorting k elements using merge sort will take $\Theta(k * logk)$ time. Therefore the total time will be $\Theta(n + k * logk)$

As k approaches to n, the time complexity of the algorithms discussed in (a) and (b) will be getting closer. However when k is smaller compared to n, choosing the algorithm in (b) will be more efficient. I would prefer algorithm (b) for $k = logn$.

Problem 2: Linear-time sorting

(a)

First, we need to identify the keys within each tuple. These keys, such as k1, k2, k3, and so on, are used to determine the lexicographic order. We must also establish the order of importance for these keys, for example ensuring that k1 is more significant than k2, and k2 is more significant than k3 Next, we should create multiple passes, one for each key. Each pass corresponds to a single key and sorts the tuples based on that key. We start with the least significant key, e.g., k3, and work our way up to the most significant key, e.g., k1.

In each pass, we need to use a stable sorting algorithm to sort the tuples based on the current key. A stable sorting algorithm is crucial because it preserves the relative order of equal elements. This means that if two tuples have the same value for the current key (e.g., k2), they should retain their relative order based on the sorting from previous passes (e.g., k1). This ensures that the sorting process respects the lexicographic order of the keys.For this we need to use radix sort in a modified way. The standard radix sort algorithm employs counting sort to efficiently arrange numerical keys. When it comes to sorting string keys, the approach diverges. In this case, we initiate the process by sorting the least significant character, which is the rightmost character. In the context of standard radix sort for numerical values, the available digits range from 0 to 9, making a 10-element auxiliary array sufficient. However, when it comes to sorting strings, there are 26 possibilities for each character due to the 26 letters in the English alphabet. Consequently, we must adjust the size of the auxiliary array accordingly. Additionally, since the input may contain strings of varying lengths, another adjustment is necessary. To address this, we ensure that all strings have the same length before initiating the comparison process. This entails equalizing the sizes of the shorter strings in our array with the longest string. One way to achieve this is by padding the shorter strings with "a" characters to match the length of the longest string. These least significant "a" characters ensure that shorter words with identical initial letters are correctly ordered, as exemplified by the comparison between "life" and "lifelong."

We should repeat the corresponding radix sort for each key by looking at whether it is integer or string, moving from the least significant to the most significant key, until all passes are completed. This comprehensive sorting approach results in the tuples being lexicographically sorted according to the specified key order.

(b)

The importance of the keys in the sorting is the most important part of the algorithm that I described in (a). The ordering of the keys should be decided based on the problem definition and what kind of sorting is needed. Therefore the ordering should consider the need for the sorting problem. In this case I will follow the ordering that is given, which means that the id is least significant and texture is the most significant key.

The sorting algorithm starts with the least significant key to sort and then continue with other keys in order iteratively. In this case we start with id. Since the id is integer, we perform the traditional radix sort. We need to count sort all the ids based on their least significant to most significant digit. While sorting we use an auxilary space of length 10 (from 0 to 9) to keep the counts. After keeping the counts, the count sort algorithm converts the counts array to an array that stores number of elements smaller than each element. After this, starting from the last element, we put the elements into new array with the index having the count value and decrement the count value by 1. At the end of this process, the ids will be sorted.

For string keys, similar approach is applied. This time we need to pass all the tuples to get the max length of the string key. After getting that we pad the short elements with "a" to make all elements same size. Starting from the last letter, for each letter in character column, we perform

count sort with an auxilary space of length 26 (one for each letter) to keep the counts. the rest of the sorting for strings are the same.

After performing the relevant radix sort for each keys by looking at whether it is integer or string, we obtained the final sorted list of tuples.

(i) Sorting by ID

⟨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⟩

(ii) Sorting by Character

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

(iii) Sorting by Color

⟨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⟩

(iv) Sorting by Size

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

(v) Sorting by Texture

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

(c)

The time complexity of the modified algorithm for lexicographically sorting a list of tuples depends on several factors. First, the number of keys in each tuple (k) plays a significant role, as the algorithm will perform a number of passes equal to this. If there are 'k' keys in each tuple, the algorithm will go through 'k' passes for sorting. Second, the length of strings or the number of digits in the keys also affects the algorithm's running time since the algorithm performs a count sort for each digit or character to sort tuples based on a single key. The number of count sorts performed by the algorithm fore each key is determined by maximum length of the keys (l)

Therefore, the time complexity of the radix sort for a single pass, in other words, sorting the tuples based on a key takes O(n*l) time, where 'n' is the number of elements being sorted, and 'l' is the number of digits (or characters) in each element. Since the modified algorithm applies radix sort for each key and has 'k' passes, the overall time complexity can be approximated as O(n * l * k).