CS310: Homework 3

Scott Fenton

March 19, 2017

Exercise (1). Professor P. has an odd habit: He only buys red, green, and blue books, and he keeps them in completely unordered stacks. However, he does keep track of the total number of books he owns. One day he decides to get a bookshelf so that he can put all books of one of the colors onto it. Unfortunately, when he is at the shelf store, he only knows that he has a total of 271 books, but he does not know how many books of each color he owns. What is the minimum number of books that the new shelf must be able to hold so that Professor P. can be sure he can put all his books of one of the colors onto it? He does not care which color it will be.

Solution (1). Well, this is not exactly the pigeonhole principle that applies, but the idea is very similar. The question is: If a set of 271 books is divided into 3 disjoint subsets, what is the maximum size that the smallest subset could possibly have? The smallest subset will be maximal if the three subsets are equal in size. The closest we can get to equal size is having 90, 90, and 91 books in the three sets, which means that the smallest set can never contain more than 90 books. Thus, Professor P. needs a shelf that can hold at least 90 books.

Exercise (1b). Give a reccurence relation for I(n)

Solution (1b). The reccurence relation for I(n) is I(2n + 1) = 2I(n) + 1. We can see from the table above that the penultimate winner, and last winner must always be an odd number, because each even number gets killed the first rotation through. The n comes from the power of 2 relationship between n = 3, n = 6, n = 12, and so on.

$$\begin{aligned} I(1) &= 1 \\ I(2n) &= 2I(n) - 1 \\ I(2n+1) &= 2I(n) + 1 \end{aligned}$$

Exercise (2). Give an O($n \log k$)-time algorithm that merges k sorted lists with a total of n elements into one sorted list.

Solution (2). We solve this by using a min-heap to merge k sorted lists. We pick the smallest element in each list, and insert them into a minimum heap. When we insert it into the min-heap we keep track of what index of the list it came from. Then we deleteMin from the heap and insert that into the new array that is used to merge all three lists. Each time we delete a value from the min-heap, we insert an element from the corresponding list that the element came from. In this algorithm it takes O(k) to build the heap, for every element it takes O($\log k$) time to DeleteMin and O($\log k$) time to insert the next element from the corresponding list. The total time is O($k + n \log k$), which can be reduced to O($n \log k$)

Exercise (3). There are many duplications in the input sequence of n numbers, such that only O($\log n$) numbers are distinct. Give an O($n \log \log n$) worst-case time algorithm to sort such a sequence.

Solution (3). We can solve this in O($n \log \log n$) by using a balanced binary tree of all the distinct numbers. Each node then contains a list containing the count of how many of that distinct number was seen. For each element in he input sequence, we try to input it into the tree, and if it is already in the tree we increment

the list within that node. After we have go through the entire input sequence, we traverse the tree in order, and combine the lists within the node. Since there are only $\log n$ distinct numbes, all of the operations are performed in $\log \log n$ time, and to sort the sequence we only have to traverse the tree and print each number the required Count of times. Reading each value takes O(n) time, and the operations on the binary search tree are performed in $O(\log \log n)$ time. So the total time is $O(n \log \log n)$.

Exercise (4). Design an O(n log n) worst-case time algorithm that counts the number of inversions in an array of n numbers. Hint: Modify mergesort.

Solution (4). We can design this algorithm to determine the number of inversions in an array of n numbers by modifying mergesort to count the number of inversions, while sorting the array. Merge sort works like normal, dividing the array intil it reaches the atomic value, and cannot be divided any further. Then it compares the elements in each subarray, and if it makes a switch, it increments the inversion counter. The process continues sorting the array, and incrementing the inversion count until the array is fully sorted. The algorithm is $O(n \log n)$. because it utilizes mergesort's performance. The initial division into subarrays is on the order of O(1), and can be ignored. Once the mergesort is split into the subarrays, it must iterate through each array O(n) times, and it does that for each subarray until it combines into one array. There are $O(\log n)$ number of subarrays, since each merge reduces the subarrays by half, resulting in a $O(\log n)$ relationship. Below is the proof for mergesort.

Proof.

$$
\begin{aligned}
T(n) &= 2 * T(n/2) + O(n) \\
&= 2 * (2 * T(n/4) + O(n/2)) + O(n) \\
&= 4 * T(n/4) + O(n) + O(n) \\
&= 4 * (2 * T(n/8) + O(n/4)) + O(n) + O(n) \\
&= 8 * T(n/8 + O(n) + O(n) + O(n) \\
&= \vdots \\
&= 2^{\log n} * T(\frac{n}{2^{\log n}}) + O(n) + O(n) + ... + O(n) \\
&= n * O(1) + O(n) * \log n \\
&= n \log n \\
&O(n \log n)
\end{aligned}
$$

□

Exercise (5). The k-th quantiles of a set of n numbers are the $k1$ numbers that divide the sorted set into k equal-sized sets (to within 1). For example, if the set is $1, 2, 3, ..., 99$, the 10-th quantiles are 10, 20, . . . ,90. Design an O(n log k) time algorithm to find the k-th quantiles of a set. Hint: Use median of medians.

Solution (5). This problem is split into two situations, either k is even, or k is odd. If k is even, we find the median, partition around that median, and recursivly call the left and right subarray. If k is odd, we chose two index values close to the median, then recursivly call the left and right subarray. To find the median in constant time, we want to use the median of medians algorithm. This can be used as a pivot strategy to find the k-th quantilies in the above problem. The median of medians divides the n numbers into n/5 groups of 5 elements, and one group of the remaining elements. It finds the median of each of the groups recursivly, and uses the median of those five groups as the pivot. The median of medians method avoids the $O(n^2)$ run time of other methods by calculating a good pivot that falls between the 30th and 70th percentile. By the time we find them median of all five medians, we have searched through $\frac{3}{10}$ of the set. Each iteration splits the array in half, so the depth is $O(\log k)$, with k being the number of quantiles. A Recurssion relation of Median of Medians is given as $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10} + 6) + O(n)$.