CS202- HW 2 Report

Halil Arda Özongun- 22202709- Section 1

05 November 2024

Question 5:

This is a heap data structure, and the only rule when building heaps is that the root must be bigger/smaller than the left and right child. But there is nothing about the relative size of the left and right child. So if we don't subtract from the heap when looking for the minimum element, we will have to look at the left and right, and if necessary their children. In the first question we looked at the first 32 elements to find the minimum 5, and since 32 is a fixed number, we can say that this algorithm ends at O(1). However, if we look for at least k elements and do it without removing them, we will have to look at every node for k height. This means looking at 2^k - 1 node enough. Since we have to look at the first 2^k - 1 node, this algorithm works at O(2^k).

If we remove it, the smallest element will be at the top in O(logn) time due to the design of the heap. Since we will do it for k elements, the algorithm works in O(k*logn).

I don't do it, but one approach would be to add the existing unused nodes in a list, and each time we find the i-th smallest element, we could add its children to the list. We could do this recursively, iterating through each element in the list for the i+1-th smallest element. This may be more efficient than mine, but for a small number like 5 this is not important, so I chose to look at 32 nodes in the first question.

Question 6

Before this I didn't know how to write a makefile, I learned how to write a makefile, what it does and how to generate executables with it. I also saw examples of heap and AVL tree data structures. I was already familiar with using heap as a priority queue, but I learned how the insertions and deletions of AVL tree have the advantage of being O(logN) cost. I also get more familiar with some technics, like two pointers and binary search on answer.

Subtask 1:

For this question I used min heap, since the question directly asked for it.

Before going into the time efficiency of the question, I want to briefly summarize each operation. Adding to heap is O(h) = O(logN) when N is the number of nodes in heap, removing is also the same. For getting the least 5, efficiency is O(1) for the same reasons I explained in question 5. There are N queries, so it has an efficiency of O(N) * O(logN + logN + 1) = O(N*logN).

Subtask 2:

Before deciding which data structure to use, we need to look at what kind of strategy we need to use to solve this problem and what kind of needs we have.

The strategy for the problem is to take a card from opponent that we can score with our own card when we take it from the opponent. So we need to get access to our largest element as quickly as possible, and we need to get access to the largest number smaller than this number in the opponent as quickly as possible. Likewise, when we discard these cards, we need to be able to quickly remove them from the deck. If we use an AVL tree, all these operations will take O(logN) time (N number of nodes in an AVL tree). Also, since there are 2N cards in total, each player draws 2 cards, so the time to play the game will be O(N). Since the number of operations in each hand is O(logN) for the reasons we just explained, the total time efficiency will be O(N*logN).

To go over the strategy again, in each turn, after choosing our highest card (O(logN)) we need to find the opponent's biggest number smaller than our number (O(logN)). After finding these two cards, we must delete them (O(logN)) and update the counters. If the opponent has no numbers smaller than ours, we need to take the largest number we can take from the opponent, and finding and deleting them has also O(logN) efficiency.

Subtask 3:

For this question I used the binary search on answer method and the heap data structure.

We are asked to find the smallest value L that satisfies the question, and if it satisfies for L, it will also satisfy for every value after L, but since L is the smallest value, no value before L will satisfy. Since we are looking for exactly one limit, we can use binary search on answer here. What we will do is to hold a left and a right pointer and check if the middle value satisfies the equation. Until the middle value is equal to the answer, we will update the left and right pointer, making the range smaller and smaller, and this approach will be equal to O(logN) since we are dividing array to half when we update left or right, where N is the length of the array.

When applying this approach, we will see if it works for a middle value so we can update our range. Suppose we find a mid value to try it out. The next thing we need to do is to find the largest M elements from the first array and the smallest M elements from the second array, then compare them. If the largest i'th element in the first array is not larger than the smallest i'th element in the second array, then it will not hold for any element smaller than the largest i'th element in the first array. But we can't use the bigger values, since we need them for other values. For these reasons, we need to keep the largest M elements in the first and the smallest M elements in the second.

To do this we can keep two heaps, and update them so that the size of these heaps is always in M. This way we get the largest M and the smallest M elements. For the first array, we will find the largest M between [0,mid]. To do this, we keep a heap and put the first M elements in the first array into this heap, and for each element between M and mid, if we see an element in the array, which is larger than the smallest element in the heap, we can include it instead of the smallest. If we keep the heap as min heap, we can access the smallest element at o(logM) and insert the new element at o(logM). If we do the opposite way for second array, hold a max heap, and while iterating add the new element if it is smaller than the biggest element in the array. We have to do these operations for every element between [0-mid], with cost logM. O(mid)*O(logM). And the equation O(mid) = O(N) is true, because mid is a number proportional to the length of the array, and can take any value between 0 and N. Therefore, it costs O(N*logM) to find the largest M and the smallest M in the heap.

Since it is O(N*logM) to compute for one mid value, and since we need to try for a total of O(logN) mid values, the time efficiency of this problem is O(N*logN*logM).

Subtask 4:

In this question we are asked to find an interval such that the M elements selected from the first array are greater than the K elements selected from the second array. To solve this effectively, we need to use a similar approach to question 3.

First of all, unlike question 3, here the first index is not fixed, both left and right indexes can change. Trying for every (l,r) pair would make the code O(N^2), but here we will use the two pointer approach. In this approach, we will use a (l,r) pair. If it doesn't provide a range, then the range is not as big as we want, we should expand the range. If it does, we should save this value somewhere and then shrink the range to see if it fits a smaller range. If we start with l = 0 and increment the right pointer when the range doesn't satisfy, increment the left pointer when it does, and so on until the right pointer exceeds the size of the array, we will end up looking at O(N) for all ranges.

But suppose we have a (l,r) pair. How do we know if this satisfies? For this, again, we need to use two data structures, and with this data structure we need to keep the largest M elements for the first array and the smallest K elements for the second array. We could keep a heap for this, but if we keep a heap, and then we change either r or l, it will be very difficult to update the heap. So we can keep an AVL tree instead. For this data structure, the insertion and deletion cost will be O(logN), and deleting any element in any index is also very simple as we can find the number directly in O(logN).

Then we need to use two AVL trees, the first one holding the largest M elements between (l,r) in the first array, and the second one holding the smallest K elements between (l,r). If we want to increase r, we should compare the r'th element in arrays and add it to the tree if necessary, and if we want to increase l, we should compare l'th element and remove it from the tree if necessary. After initially building the trees, for each update of r and l, each operation we do to them will cost O(logN). And since r and l will change by O(N) in total, the total efficiency of the algorithm will be O(N*logN).

The only small problem with this approach is that as l increases, we may need to extract values from the trees. But when we remove that value, we will not be keeping the smallest K elements or the largest M elements. The solution is to keep two more trees next to our main trees, which holdes the remaining values in the range of [l,r]. When we update r and l, we can also update these trees, and if we remove l from the main trees, we can use the values in these trees if there is a space. These are also AVL tree’s so these tree’s have same cost with our main trees, they don’t cost more time. But holding 4 trees, is not a memory efficient approach.