**CS202 Homework 2**

Hilal Coşgun

22201571

Section 1

**Question 5:**

In question 5, to find the smallest 5 elements in the heap without any removals, I scanned all the nodes within the first 5 levels of the heap. Since I only compare the first $2^5$ nodes regardless of the heap size, this approach works in O(1). However, if I were asked for to find the smallest k elements instead of 5, I would need to compare $2^k$ nodes, meaning the complexity would grow exponentially. Therefore, this approach cannot be considered efficient for large values of k.

**Question 6:**

**Q1:**

For question 1, as required, I implemented a minimum heap structure. Since a heap is structured as a complete binary tree, I chose an array-based design. I implemented the specified actions such as insertion and removal of top. When it comes to finding least 5 elements without any removals, I used this logic:

The kth smallest element can be found at most on the kth level. Because every node at the (k+1)th level has k parents which are less than the node itself. In a min heap, the smallest element is root. The 2nd smallest element is at most on the 2nd level, the 3rd smallest element is at most on 3rd level and so on. Therefore, spanning the first 5 levels is sufficient to find the smallest 5 elements. If the heap has fewer levels, then searching within those levels is sufficient. To implement this logic, I copied first $2^5 - 1$ element of the heap into an array and compared them, found the smallest element, and then repeated the process for five iterations to obtain the five smallest values. Since 5 and $2^5 - 1$ are constant numbers independent of heap size or height, this solution has constant number of comparisons and O(1) complexity. However, as I mentioned in the 5th question, for larger k values, search range exponentially grows. Comparing $2^k$ elements to find only k elements is not very efficient, thus not very applicable.

**Q2:**

For 2nd question, I implemented an AVL tree structure because I needed a function that could find the largest key smaller than a given key to achieve the optimal strategy, which cannot be easily done with a heap. AVL tree is good for such operations due to its balanced and recursion-friendly structure. My implementation also supports the presence of duplicate keys. To handle this, I added a count variable within each node. When a duplicate key is inserted, instead of creating a new node, the count of the existing node is incremented. Similarly, during removal, if an element has a count greater than one, it is not fully deleted; instead, its count is decremented by one.

I inserted both players' cards into AVL trees. To play optimally, in each round, both players need to find the largest card of their AVL trees. For this,I wrote a function that recursively reaches the rightmost node of the AVL tree to find the largest card. This function works in O(logN) since the AVL tree is balanced. Then, they find the largest card from the opponent's AVL tree that is smaller than their chosen card and gain a point. For this, I wrote another recursive method which recursively searches right and left subtrees to find this card. If key of the current root is smaller than key, the key of the root is recorded and search continues with right subtrees. Otherwise, method recursively calls itself for left subtrees. This function also works in O(logN). Removing cards from the AVL trees is O(logN) too. If a player cannot find a card smaller than their maximum in the opponent's AVL, it means they will not be able to win any of the remaining rounds. In this case, the remaining turns are added to opponent's score and the game ends. Since there are N rounds in total, for loop runs N times, with a constant number of O(logN) operations inside it, the total complexity is O(NlogN).

**Q3:**

For 3rd question, I again used AVL tree structure because I needed access to both the maximum and minimum values of the same subarray. While solving this question, I first wrote a helper function that, for a given L, finds if there are at least M indices such that Ai>Bi. To accomplish that, I created two separate AVL trees for A and B, and limited the size of each tree to M. I did this to keep the cost of operations such as insertion, removal, finding minimum and maximum at O(logM). Tree A keeps the largest M elements of its subarray, while tree B keeps the smallest M elements of its subarray.

In order to keep the size limited, the first M elements of the subarrays are directly inserted into the AVL trees. For the remaining elements of tree A, I used the following method: if the upcoming element of the subarray was greater than the minimum of tree A, then, that minimum element was no longer in the greatest M elements. So, remove it from tree A and insert the new one. Make the vice versa for the tree B. Finally, compare A with the B and if all of the corresponding M elements satisfy the condition, then length L is suitable and return true. The size of the trees is M; therefore, a single operation is O(logM). Since insertion and deletion are performed around L times, and L is equal to N at most, the helper function's complexity becomes O(NlogM).

In order to find the minimum length L, I first tried incrementing L one by one and stopping at the first success, but this approach is O(N) and exceeds the required complexity for the assignment. Then I tried a new approach, where I first checked half of N, and if that length was sufficient, I halved it further, narrowing down the possible values of L like in binary search. In this way, I find the minimum valid length in logN iterations.

The total complexity is O(logN)xO(NlogM), which results in O(NlogMlogN).

**Q4:**

In the 4th question, I used a similar approach to the 3rd question. I wrote a function that checks whether there are M elements in subarray A that are all greater than K elements of subarray B. To achieve this, I kept the largest M elements of the subarray A[left:right] in tree

A and the smallest K elements of subarray B[left:right] in tree B. This limited-size logic is the same as in the 3rd question, but this time I do not have a fixed length variable; rather, I span the subarray starting from the left index to the right index.

This spanning takes N iterations at most, and AVL tree operations have a complexity of O(logM) or O(logK). Since M>=K, M is the upper bound, making the overall complexity of function O(NlogM).

When determining the left and right indexes, I used a sliding window approach, which works as follows: Start left index from zero and right index from M. Using a helper function, check if the current values of left and right satisfy the condition. If this valid subarray is shorter than the shortest valid subarray so far, update the output. Then, increment the left index to check if any shorter valid subarray exists. If the helper function returns false with the current values of left and right, increment the right index to continue checking with a larger subarray.

This sliding window approach takes O(N). Therefore, the total complexity becomes O(NlogM) x O(N), which is equal to $O(N^2 logM)$.

In this assignment, I learned how to implement heap and AVL classes and how to use these structures to access maximum and minimum values or the predecessor of a given value. I also learned how to manage duplicate keys in binary search trees and explored different approaches to subarray problems, such as the sliding window technique.