Moin Khan, CS202, Section 1, HW2, 22101287

Question 5:

If I am asked to get the least k elements instead of the least 5 elements, the time complexity will remain the same in my approach as I am maintaining a separate static array of size 5 of the least five elements and maintain it with the least 5 elements by updating the members as new data is inserted into the heap. The time complexity to retrieve these elements is O (1); the logic will remain the same even if, instead of 5, we have k elements. But if the k starts to get very large, even though the time complexity to get k elements would be the same, it would start to become inefficient as the larger k will take more time to maintain the static array with the least k elements, a lot of sorting will need to be done, which for small numbers is acceptable but for larger number is very time-consuming.

Question 6:

Question 1:

It makes us familiar with working with min-Heaps, and the requirement to retrieve the smallest 5 elements in O (1) time without removing elements highlights the importance of data structure choice and auxiliary structures. It teaches us to maintain a separate array to avoid unnecessary heap operations. To achieve O (1) complexity of retrieval, we also show a trade-off between constant-time retrieval and incremental maintenance costs, which shows O (1) retrieval complexity does not come free but is the result of managing the static array and time spent in managing that, but this trade-off is acceptable as management of a static small sized array is much more efficient and faster than going through the whole heap for our target values. It also shows the difference between theoretical and practical complexity, as while theoretically, for a larger number of elements, time complexity might remain O (1), it will not necessarily remain as efficient as it is for a smaller number of elements. My solution for this was based on min-heaps for data structure as suggested in the question, while I maintained a static array of size five, which after each insertion stores the least 5 value if there are less than or equal to 5 values all the available values are stored in this static array, as the number increases it makes sure to maintain least 5 digits, and also make use of selection sort on it every time a number

is inserted so that it is easy to do comparison and changes in static array as a new insertion in min heap take place.

This static array allows the retrieval of at least 5 digits in O (1) complexity.

Question 2:

This question teaches the use of balanced trees for optimal strategies in competitive scenarios where both players aim to maximize their scores. It also taught me how to apply real-world thinking like one used in this card game as an algorithm, a concept useful in advanced algorithms. It also teaches you to be aware of time complexity while making your code and to create efficient code. It also enforced the importance of choosing the correct data structure based on your needs. It also familiarised me with the AVL tree and its methods of insertion and deletion in different scenarios. I made use of 2 AVL trees to store all the cards for each player; on each player's turn, it chooses the biggest card from his deck and chooses the biggest card smaller than his card from the opponent's deck if it exists. AVL trees made it easy to find the largest card, and the largest card was smaller than the player's own card from the opponent's deck while keeping storage balanced, fast, and capable of efficiently removing cards from trees after each turn. If it wasn't an AVL tree, it is possible for the tree to degrade to a Linked List-like structure. The time complexity for my code is O (N log N) as AVL tree node insertion and AVL tree node deletion take place within a loop of size N, where these insertions and deletions and search take only O (log N) time.

Question 3:

In this question, I had a similar application to that of Question 2; I made use of AVL trees, and I learned how to use them and how balanced trees are faster and more efficient than non-balanced trees in terms of traversal, insertion, and deletion. To solve this, I slightly modified the code to find the optimal card from your opponent in Question 2 to handle situations where numbers are equal. Then I started with two subarrays of size M and inserted them into two AVL trees, and I applied the algorithm of the card game to select the largest number from first tree and use that largest number to find the largest number smaller than that number in second tree if it exists after which those numbers will be removed from AVL tree, and whenever the number from first tree was larger than number from second, I counted it, if this count became

at least equal to m, the size of that array was my answer. While I could not achieve O (N log N log M) complexity, my code is still efficient with the complexity of O (N^2 log N). This also taught me to be aware of time complexity while coding to make my code efficient whenever possible.

Match Details:

On November 6, 2024, Beşiktaş won 2-1 against Malmö in the UEFA Europa League at Vodafone Park in Istanbul, Turkey. The first half featured no goals, neither of the teams was able to make a breakthrough. In the 76th minute, Beşiktaş took advantage, with Ernest Muci giving the team a close-range finish. Semih Kılıçsoy doubled the advantage in the 85th minute, which saw his shot from outside the penalty box to the net. A late goal from Malmö's Soren Rieks in stoppage time reduced the gap, but that did not change the outcome. The win pushed Beşiktaş up to six points in 14th place while Malmö remained on three points in 26th place.