CS 180
Dis 1B->Parshan
Krish Patel

Algorithms and Complexity, Homework 4->

Q1-> Exercise 11 pg 193
Problem 1)
Currently in class, we've been taught about using Kruskal's Algo only with weighted graphs where no two edges have the same weights. However, with a simple modification to the algorithm, we could guarantee it to work with graphs with weighted graphs having similar edges.
There is a way to get the valid minimal spanning tree as the output. We can do this in the following manner:
First we sort the list of edges in term
Find the shortest difference between any two edges provided that they aren't similar. We take this value, and divide it by around n^2, and call this constant C. This is helpful as n^2 is strictly greater than or equal to n when n>= 0. Then, for each of the values in the vertices, we increment the value of each distance by C*i, where I represents the indexing of the array in which it is stored. Based on this new ordering, now each of the edges have a different weight, and the weighted ordering of the edges should not change.
We then run Kruskal's Algorithm on this new sorted list, giving a time complexity of O(Elog E) where E represents the amount of time to sort the edges. As now all the weights are distinct, the algorithm would create the same MST every time it is ran.
However, as E could be a maximum of N^2, we could say that the time complexity is O(V^2log V) after simplification

Q2)
This would be a variation of the scheduling problem.
We first convert all the times to 24 hour format.
We then list all of the tasks in terms of first completion time.
Steps:

- We first start by converting all of the times to 24 hour format so we can carry out direct comparisons. However, if the start time for on of the time slots is before 24 and ends on the next day, for instance 1 am the next day, then we should label it as 25 and so on
- We then sort the tasks out using a quick sort algorithm, based on first completion time.
- We then use a greedy algorithm, and take pick the time slot that has the shortest completion time, and delete all the tasks that run during that interval. We then pick the next task, and so on. For the tasks that end after 2400, we linearly go through the array and if the completion time is before the start time of the first task of the day, then we add it to the schedule, else we terminate and return the list. This is because the completion times are in ascending order, and if we find a task that ends after the first time slot start, then all tasks after that would end after that time, thus adding them the schedule would cause conflicts.

Time complexity-> This would take O(n log n) because of the sorting algorithm we would use. The most time efficient sorting algo has a time complexity of O(n log n).
Proof that it works:
The greedy algorithm efficiently maximizes task scheduling by selecting the earliest finishing tasks first, ensuring there is no overlap and optimal use of time. Tasks ending after midnight are adjusted to maintain the continuous schedule. The algorithm's effectiveness is based on leaving the most room for the future tasks, given that tasks are independent.

Q3)
For this algorithm, we can only make comparisons between two cards.
Step wise implementation of the algorithm:

- We start with a list of cards, and we pair each card with the next one, until a pair is formed. We do this until we have n/2 pairs (and if there is an extra card, we keep it by itself and label it an alone class).
- Each of these pairs go into the tester. If there is a match between the two cards, take the matching card and add it to another list(or remove one of the cards from the list) If there is a difference in the cards, discard both of them. If only left with one card, then add that to the next list. However, if there is a comparison between a normal pair and an alone value, then the alone pair should hold lower precedence and should be discarded if they are different.If they are the same, keep it
- Keep repeating this process for the new list, until you are left with one or no cards.
- If there is only 1 card remaining, it means that the same card is has more than n/2 duplicates. If there is none in the end, there is no card that occurs more than n/2 times.

According to the algo, we reduce the list by at least a half, and thus this algorithm has a n log n time complexity(where the log is base two). The total time complexity for this entire algo would be O(n log n)
Proof that it works: Suppose we have two worst case scenarios.
In the worst-case scenario, even if the majority element is always paired with a different element, the majority element will still never be completely eliminated because there is always one more of it than all other elements combined. Thus, by the end of the process, it must be the last card remaining if a majority exists. If no majority exists, the algorithm correctly concludes with no remaining card.
IF all the common cards are paired with each other, the number of common cards would be greater than n/2 if there is a majority, and thus it would still be the majority in the next round of comparisons.

Q4)
For this algorithm
To solve this problem efficiently, we want an algorithm that runs in *O*(*n*log*n*) time. One approach is to use a divide and conquer algorithm similar to the way the merge sort algorithm works.
Here are the steps for the algorithm using Divide and conquer:

1. **Divide:**Split the set of lines into two halves.
2. **Conquer**: We can recursively find the uppermost line in each half. This step will work by recursively dividing each subset until you reach subsets that contain only one line or can be trivially compared directly.
3. We then combine the two halves back together by comparing the uppermost lines from each subset. This step can be done by considering the points of intersection between lines. By sorting these points of intersection along the x-axis, we can then find the sequence of uppermost lines across the set of all lines.
4. As we merge the subsets, we can eliminate lines that are never the uppermost line at any x-coordinate. If a line in one subset is always below another line at every point of intersection, it can be discarded.
5. After the merge step, we have a set of lines that are uppermost at least at one point. We can then traverse this list to build the final listof visible lines.
6. To ensure that the merging step is efficient, we store the lines in a structure that allows us to quickly determine the uppermost line at any given x-coordinate, for this purpose we could use a balanced binary search tree).

Q5)
For this algorithm, we have an sorted array that is shifted towards the right.
We implement the following algorithm:

- We take the start and end indexes of the array and initialize them to two variables: start and end. I
- We then take the midpoint value by calculating the index as end+start/2. We then make a comparison as follows: Compare the value at mid and end positions. If mid < end, then this means that the smallest value in the array is on the left. Change the end to be mid. IF mid is greater than end, then we update the start variable to be mid, as the min value lies towards the right of the partition. If the start = end, then the min value lies at that index. Return that index.
- Repeat this till you have found the minimum value, and the index of that value should give you the value for K.

The time complexity for the algorithm is log n, as this divides the array into two parts, and continuously does that. Thus, as the array halves at every iteration, it has a time complexity of O(log n) and thus very efficient
Proof of working :
Let's consider the rotated array [4, 5, 6, 7, 0, 1, 2].

- Start is 0, End is 6 (array length - 1).
- Mid is (0 + 6) / 2 = 3, and array[3] is 7. Since 7 > array[6] (which is 2), we update start to mid + 1 which is 4.
- Now, Start is 4, End is 6.
- Mid is (4 + 6) / 2 = 5, and array[5] is 1. Since 1 < array[6], we set end to mid, which is 5.
- Now, Start is 4, End is 5.
- Mid is (4 + 5) / 2 = 4, and array[4] is 0. Since 0 < array[5], we set end to mid, which is 4.
- Now, Start and End both equal 4.
- The loop ends, and the minimum value is at index 4, which is 0.

Thus, the array was rotated k = 4 positions to the right

Q6)

Step wise algorithm for the problem:

- If one of the arrays is empty, return the k-th element from the full array.
- If k == 1, return the minimum of the first elements of both arrays.
- If k == m + n, return the maximum of the last elements of both arrays.

Binary Search on the Smaller Array:

- Identify the smaller array (let's call it A) and the larger array (let's call it B). This is to ensure we perform the binary search on the smaller array to achieve O(log(min(m, n))) complexity.
- Set two pointers for this, start = max(0, k – length of B) and end = min(length of A), k)

While start <= end, perform the following steps:

```
    Calculate the mid-point of array A, midA = (start + end) / 2.
    Calculate the corresponding cut on array B, midB = k- midA.
    Get the elements just before and after the cut from both arrays:
        leftA = A[midA - 1] if midA > 0, else -infinity
        leftB = B[midB - 1] if midB > 0, else -infinity
        rightA = A[midA] if midA < len(A), else infinity
        rightB = B[midB] if midB < len(B), else infinity
```

Now we have two elements on the left (leftA and leftB) and two on the right (rightA and rightB).
If leftA <= rightB and leftB less than or equal to rightA, wehave found the correct partition.

- If leftA > rightB, move towards the left in array A (end = midA - 1).
- If leftB > rightA, move towards the right in array A (start = midA + 1).

Finding the k-th Element:

- Once the middle values (for the partition) are correctly positioned, the k-th element is the maximum of leftA and leftB.

Time Complexity:

Each iteration of the while loop reduces the search space by half, so the time complexity is O(log(min(m, n))), where min(m,n) is the minimum value between m and n)