Joyce Chen Dis 1F 2:00PM Parshan Teimouri

CS180 Homework #4

1. **Exercise 11 on page 193**
   Claim: Yes, there is a valid execution of Kruskal's Algorithm on G that produces a unique minimum spanning tree T, given that G contains edges of the same cost.
   Proof:
   - Suppose that running Kruskal's on G doesn't produce T.
   - This means that we added an edge e' that doesn't create an MST.
   - Another e exists such that cost(e) < cost(e') and doesn't produce a cycle.
   - This means that we encountered e' before e, causing us to add e' to the MST. But this is a contradiction, since we iterate through all the edges in G in nondecreasing order of cost. ><

   - If cost(e') = cost(e), then both edges e and e' would be considered part of the minimum spanning tree,if adding both wouldn't create any cycle.
   - If we wanted to obtain T that includes e' and not e, we can order e' first in the initial sorted list of edges.
     - This can be done by setting a specific ordering of the edges, whether that be subtracting from each edge to obtain edges with all distinct weights (while still retaining a valid ordering).
     - The ordering of edges with the same cost wouldn't matter in terms of creating a valid MST, but it does matter if we want to create a unique MST out of it.

2. **Exercise 17 on page 197**
   Algorithm:
   - Pick an arbitrary job interval, J.
   - Delete all intervals that overlap with J, such that we split the time at some point in J, say t.
   - Consider the remaining job intervals that run between t and t + 24 hours.
     - Sort the job intervals by end time.
     - Greedily select the job interval with the earliest end time, and delete all overlapping job intervals out of the remaining ones.
     - After we finish picking all the job intervals, add J back to the solution. The result will be the optimal job schedule if we explicitly include J and start counting time from t.
   - Repeat this process for all job intervals.
   - Select the job schedule with the maximum # of jobs.

   Proof of Correctness:

- Assume that there exists an optimal algorithm S* that chooses more jobs than our algorithm S.
- Base case: 1st accepted job request.
    - S goes through all possible job intervals, and ends up selecting 1.
    - S* also selects 1 job interval. Whatever S* chooses, S will have considered before.
    - Both S and S* pick 1 job.
- Inductive hypothesis: kth accepted job request.
    - Assume that our algorithm S picks at least as many jobs as S* given k-1 job requests.
    - On the kth job request, S selects the job request with the earliest end time that doesn't overlap with previous selected job requests.
    - Therefore, S will always remain ahead of S* because we are selecting the job that makes for maximum remaining time to choose other job intervals.
    - We repeat the process for all job requests given, so we're considering all possible optimal job schedules in S.
- S will choose the most job intervals within a certain 24-hour frame.

<u>Time Complexity Analysis</u>:

- Choosing an arbitrary job J and deleting all overlapping intervals with J takes O(1) time.
- Sorting the job requests by end time takes O(NlogN) time.
- Greedily picking the job requests to form a valid schedule takes O(N) time.
- We repeat the process of choosing an arbitrary job and sorting by end time for all N jobs.
- Total: $O(N^2)$ time.

3. **Exercise 3 on Page 246**

   <u>Algorithm</u>:

   - Divide and conquer: break the set of cards into subsets. build up to a solution by checking whether each subset meets the requirement of having more than half of the cards being equivalent.
   - Order the set of n cards arbitrarily. Recursively call this function on the left and right halves of the n cards until we reach 1 card on both sides.
   - After reaching the base case (1 card on each half), do the steps below on the 2 halves to determine whether there are >c/2 equivalent cards in the current set of c cards.
       - Base case:
           - Plug the pair of cards into the machine.

        - If the machine returns different, return NO.
        - If the machine returns the same, return YES.
    - If 1 half is NO and the other half is NO:
        - there is no possibility for the combined halves to have more than half the cards equivalent to each other
        - ie. if card A != B, C != D, then at most only 2 cards out of ABCD will be equivalent (A = D, B = C).
        - returns NO.
    - If 1 half is YES and the other half is NO:
        - pick a card from the set of equivalent cards in the YES half
        - test this card with all the cards in the NO half
        - if the equivalent cards in the YES half + cards in the NO half that return true > c/2, return YES.
        - otherwise, return NO.
    - If 1 half is YES and the other half is YES:
        - each half of c cards has >c/2 equivalent cards
        - plug 1 card from the equivalent cards on each side into the machine.
        - return NO if different, YES if same.
- If the function returns YES on the last merge, where both halves have n/2 cards, then there are >n/2 equivalent cards in the set of n cards.

<u>Proof of Correctness</u>:

- Base case: 2 cards.
    - If they're the same, 2/2 > ½ of them are equivalent, satisfying the condition. If they're different, then 1 out of 2 cards are equivalent but ½ isn't > ½, meaning that the condition isn't satisfied.
    - Our algorithm correctly returns true if the 2 cards are the same, false if they're different.
- Inductive hypothesis: > 2 cards.
    - Assume that our algorithm produces the correct result for n-1 cards. That is, out of n-1 cards, if >(n-1)/2 cards are equivalent, our algorithm returns YES and otherwise NO.
    - For n cards, we say that they produce a majority of equivalent cards if we merge the left and right halves of the n cards and return YES.
    - Our algorithm returns NO if the 2 halves both don't have a majority of equivalent cards, or if 1 half contains a majority and the other doesn't but combined they don't produce a majority.

        - Therefore, our algorithm correctly returns NO if there isn't a majority in the n cards.
    - Our algorithm returns YES if the 2 halves both have a majority, or if the 2 halves combined produce a majority.
        - Suppose we have A cards in the left half, and A/2 + 1 cards are the same. We have A cards in the right half, and A/2 + 1 cards are the same.
        - Combined, we have 2A cards and A+2 cards that are the same. A+2 > A, so we still maintain a majority. Therefore, 2 halves that return YES will return YES when combined.
        - Our algorithm correctly returns YES if there is a majority in the n cards.
- Therefore, our algorithm produces the correct result.

Time Complexity Analysis:
- Recursively calling the function on both halves of the n cards takes O(logN) time.
- For each of the 2 halves produced, we must merge them to get a solution.
    - The comparison operations are all O(1).
    - Worst case, we would have to compare a card in 1 half with all cards in the other half which takes O(N) time.
- Total = O(NlogN) time.

4. **Exercise 5 on page 248**

Idea:
- Use divide and conquer to get the # of visible lines in a subset of n lines, then merge these sub-solutions to get the actual # of visible lines in the total set of n lines.
- To find the # of visible lines in a subset, we can observe the following:
    - Given a subset of 3 lines:
        - The 2 lines with the min and max slope will always be visible at some point x.
        - The remaining line will be visible if its intersection point w/ the min-slope line is to the *left* of that w/ the max-slope line.
- To merge the of 2 subsets (find the visible lines):
    - We look at the visible lines in the 2 subsets, and ignore those that are marked invisible already (they will never be visible, because we are only adding more lines to the solution at this point).
    - Again, we know that the visible lines with min/max slopes will still be visible, respectfully.

    - Consider each intersection point between 2 visible lines in the combined set.
        - Check for lines that lie between the 2 visible lines that are hidden by them.
        - Eliminate these lines from the set.

Algorithm:

- Order the lines in increasing slope, {L1, L2, … LN}
- Recursively call our function on the first N/2 lines and last N/2 lines.
- When we reach a base case of 3 lines {Li, Li+1, Li+2}:
    - Li and Li+2 will always be visible at some point x.
    - Li+1 will be visible if it intersects with Li to the left of where it intersects with Li+2.
    - Otherwise, eliminate Li+1 from the final set of visible lines.
    - Add the 2 or 3 visible lines to the “visible” subset.
- Merge function:
    - For 2 subsets of visible lines, the 1st and last line of the combined set will always be visible at some point x.
    - The remaining lines will be visible if its intersection point with the min-slope line is to the left of that with the max-slope line.

Proof of Correctness:

- Suppose we don’t find the the total # of visible lines by the end of our algorithm.
- This would mean that at some point, we didn’t correctly count the # of visible lines in a subset, such that when we merged it with another subset, the # of visible lines wasn’t correct.
- In the base case, we established that a line will be marked invisible if it is “squeezed” in between 2 lines.
- Thus, we will get the correct # of visible lines for the base case.
- When we merge the 2 subsets of base cases, we are checking whether there are any lines that become hidden when they were previously marked visible within their own subset.
- Thus, we are always getting the correct # of visible lines for each set of lines broken down from the N lines.
- Our algorithm correctly calculates the total # of lines by induction.

Time Complexity Analysis:

- Recursively calling the function takes O(logN) time, since we’re halving the problem each time.

- Merging the function (determining how many visible lines result from combining the 2 subsets of visible lines) takes O(N) time.
- We are doing the merge for each pair of halves in our set of N lines.
- Total time: O(NlogN) time.

5. **Suppose you are given an array of sorted integers that has been circularly shifted k positions to the right. For example taking (1 3 4 5 7) and circularly shifting it 2 positions to the right you get (5 7 1 3 4). Design an efficient algorithm for finding K.**

   <u>Algorithm</u>:
   - Let the first element of the original array be A. To get k, we just need to find the index of A in the circularly shifted array.
   - Perform a binary search on the circularly shifted array.
   - Create a “low” and “high” pointer for the first and last element of the shifted array, respectively.
   - While the low and high pointer don’t overlap:
     - Select the element in the middle of low and high, M.
     - If M < the element right before M:
       - M is A. Return M’s index.
     - If M >= low:
       - This means that the entire left half of the shifted array is in sorted order. So A must lie in the right half of the array because A breaks the sorted order.
       - Set low = element right after M to check the right half.
     - Otherwise, if M < low:
       - Since M is not A, A must lie somewhere before M in the left half of the array. This is because A is the minimum element of the shifted array.
       - Set high = element right before M to check the left half.
   - If we haven’t returned by now, the array isn’t shifted (k = 0).

   <u>Proof of Correctness</u>:
   - Our algorithm claims that k = index of min. element in the shifted array.
   - If the min.element M isn’t the first element, then it’s the only element in the array whose previous element is greater than it.
   - Elements in the subarray to the left of M are all smaller than elements in the subarray to the right of M.
   - Using this logic, we can deduce that:
     - If element A > element B, then A and B must be in the same subarray (either left or right).

    - If element A < element B, then A could be either to the right of B (A is part of M's right subarray and B is in M's left subarray) or left of B (A and B are both in the left subarray).
- Let element A = the middle element and element B = the low element:
    - If mid > low, then they both must be in the left subarray of M because low != M. Therefore, M lies to the right of mid.
    - If mid < low, then mid must be in the right subarray and low in the left subarray, because if they were in the same subarray then low = M. However, we already established that low != M in the initial check. Therefore, M would like to the left of mid.
- By updating low recursively such that the array is halved each time, we narrow down M to be either on the right or left half of the array and eventually figure out its location.

<u>Time Complexity Analysis</u>:
- Each time we are comparing M, we narrow down M's position to be either in the right or left half of the subarray by updating the low/high pointers.
    - This binary search for M takes O(logN) time.
- All comparisons take constant O(1) time since we're just doing array accesses.
- Therefore, the total time is O(logN).

6. **Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k-th position of the final sorted array. Note that a linear time algorithm is trivial (and therefore we are not interested in).**
Input : Array 1 - 2 3 6 7 9
Array 2 - 1 4 8 10
k = 5
Output : 6
<u>Idea:</u>
- Given that the arrays are sorted, we can just do a binary search for a point P in the smaller array.
- All elements up to P in the smaller array + all elements up to k-P in the larger array = the k smallest elements in the final sorted array.
- The max of the last elements in the 2 subarrays is our answer (k-th element).
- A valid partition of k elements in the 2 arrays is as follows:
    - The last element in the 1st subarray must be less than the element right after the last element of the 2nd subarray.

- The last element in the 2nd subarray must be less than the element right after the last element of the 1st subarray.
- This is how we know that these are truly the k smallest elements in the 2 arrays.

Algorithm:

- Maintain a low and high pointer for the smaller array.
    - These serve as the range (upper/lower bound) for the location of P, the index we decide to partition the smaller array.
- Set the low pointer to *max(0, k - length of larger array)*.
    - If k > length of the larger array, then P must be at least k-length positions into the smaller array (all elements in the larger array are smaller than those in the smaller array).
    - If k < length of larger array, P could very well be the first element of the smaller array, at index 0 (all k elements are in the larger array).
- Set the high pointer to *min(length of smaller array, k)*.
    - If k < length of smaller array, then obviously P can't be past k.
    - Otherwise all elements of the smaller array are in the partition.
- While low < high:
    - Let mid = the position halfway low and high. We consider whether mid can be the position P.
    - Let mid2 = k - mid - 2, or the position of the corresponding partition in the larger array.
    - If smaller_arr[mid] <= larger_arr[mid2 + 1] and larger_arr[mid2] <= smaller_arr[mid + 1]:
        - We've found a valid partitioning.
        - P = mid, but the actual kth element is the maximum between smaller_arr[P] and larger_arr[k-P-2].
    - If smaller_arr[mid] > larger_arr[mid2 + 1]:
        - P must lie somewhere to the left of mid.
        - Update high to be mid - 1.
    - If larger_arr[mid2] > smaller_arr[mid + 1]:
        - P must lie somewhere to the right of mid.
        - Update low to be mid + 1.
- The k-th element is the last element of the larger array.

Proof of Correctness:

- Assume that we don't find the k-th element of the final sorted array.
- This means that the element we found > the actual k-th element.
- There are 2 cases in which this could happen:

    - Element found = last element in the larger array’s partition, and it was bigger than values not included in the smaller array’s partition.
        - We check that larger_arr[mid2] <= smaller_arr[mid + 1] before finalizing larger_arr[mid2] as the last element in the larger array’s partition.
        - ><
    - Element found = last element in the smaller array’s partition, and it was bigger than values not included in the larger array’s partition.
        - However, we check that larger_arr[mid2] <= smaller_arr[mid + 1] before finalizing larger_arr[mid2] as the last element in the larger array’s partition.
        - ><
- Therefore, our algorithm finds the k-th element of the sorted array.

<u>Time Complexity Analysis</u>:

- Trying to find a valid partitioning using binary search on the smaller array costs O(logN) time, where N = size of the smaller array.
- Comparing the elements of the array takes constant O(1) time.
- Maintaining and updating the low / high pointers takes constant O(1) time.
- Total: O(logN) time.