HW 4 rubrics

1. (15 points) Exercise 11 on page 193

Conclusion (5 points): For any graph G, and any MST T of G, there is a valid execution of Kruskal's algorithm on G that produces T as output.

Proof/Algorithm (10 points):

- We can modify the edges for the assumption that there could be edges with the same weight.

- Assume there is an edge e and e', with the same weight $w_e = w_{e'} = a$, and δ is the least weight difference across all pair of edges.

- Then, we can add a very small number ϵ, $\epsilon \ll \delta$, onto $w_{e'}$ to let $w_{e'} = a + \epsilon$.

- We do this for every pair of nodes that share the same weights (of course the $\epsilon's$ are different), to make all edges have distinct values.

- Then we can use the Kruskal's algorithm to output a valid MST since now the weights are distinct, which satisfy the running prerequisite of Kruskal's algorithm.

Time Complexity: Optional since this question explicitly asks for only proof. Kruskal's algorithm $O(mlogm)$. Finding δ takes $O(m)$ since we only need to compare the pairs of weights in order after sorting done in Kruskal. Adding small numbers takes $O(m)$. In total $O(mlogm)$. Optional, no credits.

2. (15 points) Exercise 17 on page 197

Algorithm (7 points):

- The difference between this problem and the interval scheduling problem we discussed during lectures is there are schedules across the midnight and make it complicated.

- We split all intervals that across the midnight into two parts. – $O(n)$

- We sort the current intervals (intervals across midnight now become two intervals) based on their ending time – $O(nlogn)$

- We notice that for the intervals across midnight, only one can be included into the final schedule since they at least overlapped at the midnight.

- Therefore, for each such intervals i that across midnight, we select it, and exclude other intervals that across midnight.

- Then, with i include, we can run the interval scheduling problem using the algorithm we discussed during the lectures for all the current intervals. (students are allowed to just call the greedy algorithm of the interval scheduling problem we mentioned during the classes without step-by-step descriptions) – $O(n)$ given sorted intervals

- For every interval across midnight, $i = 1 \dots k$, we do the same thing, and record the maximum number of intervals we can carry and return. - $O(n * n) = O(n^2)$

Time Complexity (3 points):

- Detailed time complexity analysis is included in the algorithm description.

In total it should be $O(n^2)$

Proof (5 points):

- Since the intervals that across midnight cannot co-exist at the same time, we pick each of them and then consider the solution independently. When we pick one of them, we actually running the greedy algorithm of interval scheduling that taught on the class, which has been proved its correctness. We did the greedy algorithm for every interval across midnight, which means we have checked all possible cases, and no solutions can be better than the current results. Proved.

3. (15 points) Exercise 3 on Page 246

Algorithm (10 points):

- We will use divide and conquer to solve this problem.

- We first partition the entire list of n cards into list size of 1 to begin with.

- During merging, we use the equivalence tester. Given two cards, there are three situations:

  1) One value is None. Then we return the other card to next level

  2) Two values are tested equal, then we return one of the cards to next level.

  3) Two values are tested different, then we return None to next level.

- If there are odd number of cards, we create a None to pair the last one card not paired.

- The final returned card is the requested card. If the final return is None then that means there is no such card satisfies the requirement.

Time Complexity (3 points):

- Partition takes ~ $O(n)$

- At each level of the partition, the worst case is when you compare $\frac{n}{2}$ pairs of cards. After the comparison, the algorithm will do merge step and then go to upper level, with only half of the cards following the algorithm we designed (Either return None or one card). Therefore, from bottom to the top, the merge step costs $\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots \rightarrow O(n)$

- In total $O(n)$

Proof (2 points):

- The same as the majority problem we covered in Week 1. During the merging step, it is always true that either there is a cancellation or we keep a potential majority card. Basic idea is that it is not possible to destroy a majority by canceling out one card from the majority cards with one card from a non-majority card. This can be shown during the class and can be directly called.

4. (20 points) Exercise 5 on page 248

Algorithm (10 points):

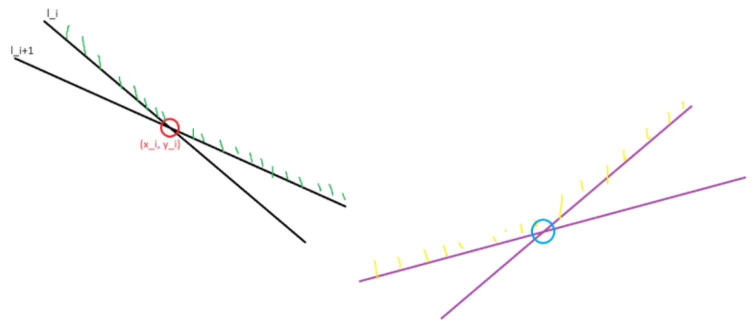
- We will use divide and conquer to solve this problem.

- For all parallel lines, we delete the one with lower y-intercept since the one with higher y-intercept will be always above it and make it invisible.

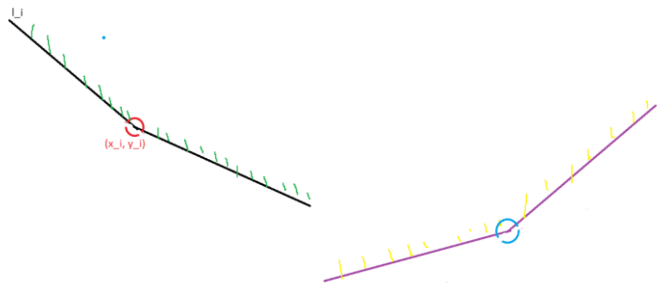
- We then sort all lines with respect to their slopes.

- Partition: We partition them into n groups, which contains only one line. Now within each group, the line is obviously visible since there is only one there, and we treat this as the base case.

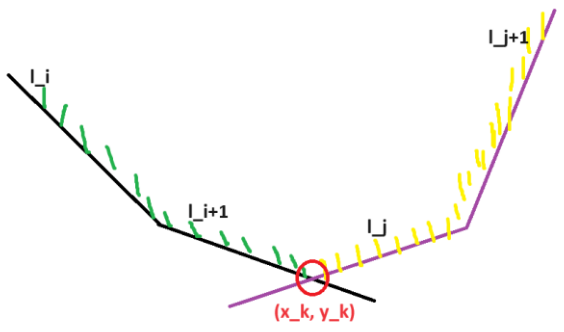
- For each partition, we will record visible lines, and also the intersection points. For example, if there are two lines l_i, l_{i+1}, and the intersection point is (x_i,y_i), then we have $[(l_i, -\infty,\ x_i)$, $(l_{i+1},\ x_i,\infty)]$ indicates l_i is visible during $(-\infty,\ x_i)$, and l_{i+1} is visible during (x_i,∞). Below, two black lines have an intersection point at (x_i,y_i). The visible intervals are shown with green and yellow color for left and right figure respectively.



- Merge: During the step we merge two groups of lines, multiple intersection points will be introduced. We only care about the newly introduced intersection points that generated by the visible intervals from each side (***prove later – P1***).



- Given the lines are sorted, the visible intervals from both sides will only add one new intersection point. (***prove later– P2***).



- This means, before merging, we have segments represented by $[(l_i, -\infty, x_i), (l_{i+1}, x_i, \infty)]$, and segments represented by $[(l_j, -\infty, x_j), (l_{j+1}, x_j, \infty)]$ from the two sides of the partitions. After merging, there will be some situations, depending on relationships among y_i, y_j, y_k. For example, if $y_k < y_i$ and $y_k < y_j$, then we will have the recorded list as $[(l_i, -\infty, x_i), (l_{i+1}, x_i, x_k), (l_j, x_k, x_j), (l_{j+1}, x_j, \infty)]$, which is shown above. For example, if $y_k > y_i$ and $y_k > y_j$, then we will have the recorded list as $[(l_i, -\infty, x_k), (l_{j+1}, x_k, \infty)]$, etc.

- There can be other situations with different relationship among y_i, y_j, y_k, but at most the recorded list will increase two segments since there at most one new intersection point added. This means at each level during the merging, the operations will cost $\sim O(n)$ for adding all original segments together.

- After all the merging steps, the returned recorded list is the requested answer.

Time Complexity (3 points):

- Partition takes ~ $O(n)$

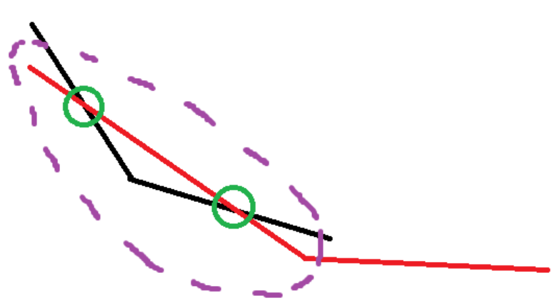
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + Cn \rightarrow T(n) \sim O(nlogn)$

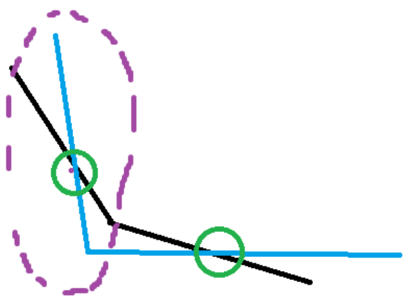
Proof (7 points):

- 1) **P1:** We only care the intersection point caused by the visible line segments. This is because the line segments that are currently invisible will never become visible in the future. The future visible line segments will always have higher y coordinates compared with the current invisible line segments given the same x coordinate.

- 2) **P2**: Why there will be only one new intersection point from two groups of visible line segments? First, only when two groups of lines are exactly the same then there will be no new intersection point. However, we have got rid of all parallel lines before running the algorithm, and thus this situation cannot happen.

Second, we want to show there cannot be more than one new intersection point added. The two situations that two visible line segments can have multiple new intersections can be seen below. Both of them are conflict with the truth that all lines are sorted based on the slope first, and then partition and merge. For example, the red one has two intersection points with the black one. The left line segment of the red one has a slope in between of the two black line segments. However, if following the algorithm, both of the two red lines should either have larger or smaller slope than both black line segments. (No need rigorous math proof, descriptions like this will be fine)





5. (15 points) Suppose you are given an array of distinct 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 position to the right you get (5 7 1 3 4). Design an efficient algorithm for finding K.

Algorithm (7 points):

- We will use binary search to solve this problem.

- We keep three indices, L indicates the current leftmost element, M indicates the current middle element and R indicates the current rightmost element. Some situations:

  - If Arr[L]>Arr[M], then we know the inversion is in the lhs of the arr. Then R point to the element current pointed by M, and M pointed to the middle one of L and R.

  - If Arr[R]<Arr[M], then we know the inversion is in the rhs of the arr. Then L point to the element current pointed by M, and M pointed to the middle one of L and R.

- We do this until we have Arr[M]>Arr[M+1] or Arr[M-1]>Arr[M] (finding a pair of neighbor elements such that the left one is greater than the right one), and then M is the requested value.

Time Complexity (3 points):

- Every time we halve the number of elements, in total we can halve $\sim O(logn)$ times.

- Then each result half, we did constant time checking ~ $O(1)$

- In total it is $O(logn)$

Proof (5 points):

- Proof by case analysis. If the array shifts, then there has to be a pair of neighbor elements such that one is the largest in the Arr and the other one is the least in the Arr. In every step, If Arr[L]>Arr[M], or Arr[R]<Arr[M], then we can know that in the corresponding half, there exists a pair of neighbor elements such that the order is reversed. This is because by default, we should have the element to the left is less than the element to the right. We iteratively halve the subarray that we will be searching, and finally we can find the pair of neighbor elements, and the index of the element in the left is the number of steps the array shifted.

6. (20 points) 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

Algorithm (10 points):

- We will use binary search to solve this problem, similar as Q5.

- We keep 6 indices. For array 1, L_1 indicates the current leftmost element, M_1 indicates the current middle element and R_1 indicates the current rightmost element. For array 2, L_2 indicates the current leftmost element, M_2 indicates the current middle element and R_2 indicates the current rightmost element. Assume the k_{th} element has a name of x.

- When given input array Arr_1 & Arr_2

- if $M_1 + M_2 + 1 < k$

  - if $Arr_1[M_1] < Arr_2[M_2]$, then x cannot be in the lhs of $Arr_1[M_1]$. Proof the above statement by contradiction. Assume the x in the lhs of $Arr_1[M_1]$. Then it can at most be greater than $M_1 - 1 + M_2$ elements in total given $x < Arr_1[M_1] < Arr_2[M_2]$. This contradicts the fact that "$M_1 + M_2 + 1 < k$". Proved. Therefore, we can just ignore the first half of Arr_1, and then run the algorithm again, exchange the input from Arr_1 & Arr_2 to be $Arr_1[M_1 + 1: end]$&Arr_2.

  - else, following similar logic as above, then x cannot be in the lhs of $Arr_2[M_2]$. Therefore, we can just ignore the first half of Arr_2, and then run the algorithm again, exchange the input from Arr_1 & Arr_2 to be Arr_1&$Arr_2[M_2 + 1: end]$.

- else

  - if $Arr_1[M_1] < Arr_2[M_2]$, then x cannot be in the rhs of $Arr_2[M_2]$. Similar logic as above. Proof the above statement by contradiction. Assume the x in the rhs of $Arr_2[M_2]$. Then it has to be greater than $M_1 + M_2$ elements in total given $x > Arr_2[M_2] > Arr_1[M_1]$. This contradicts the fact that "$M_1 + M_2 + 1 \geq k$". Proved. Therefore, we can just ignore the second half of Arr_2, and then run the algorithm again, exchange the input from Arr_1 & Arr_2 to be Arr_1&$Arr_2[0: M_1 + 1]$.

  - else, following similar logic as above, then x cannot be in the rhs of $Arr_1[M_1]$. Therefore, we can just ignore the second half of Arr_1, and then run the algorithm again, exchange the input from Arr_1 & Arr_2 to be $Arr_1[0: M_1 + 1]$&Arr_2.

- if given the Arr_1 $(or\ Arr_2)$ has been fully scanned, then we just need to do binary search on the rest of the elements on Arr_2 $(or\ Arr_1)$ until we find the request x.

Time Complexity (3 points):

- Every time we halve the number of elements for both arrays. In the worst case we can only halve the two arrays $\sim O(logn + logm)$ times. In each result half, we did constant time checking. In total, $\sim O(logn + logm)$

Proof (7 points): Proof is along with the pseudo-code.