Arthur Zhou

Dis 1A, 10:00am-11:50am
Parshan

1) • Sort the jobs in decreasing order of $\frac{w_i}{t_i}$
• Through doing this, we maximize the satisfaction per unit time for the customers
• Iterate thru the entire order of jobs
  ○ Perform the tasks and sum up $\sum_{i=1}^{n} w_i C_i$
• Return the order of jobs

Time Complexity: Should be $\boxed{O(n \log n)}$ because we can use heapsort to order the jobs by $\frac{w_i}{t_i}$ ( calculating takes $O(1)$ time). Iterating thru the order is $O(n)$ time which is insignificant to $O(n \log n)$

Proof: BWOC, we have i and j where we will do i before j since $\frac{w_i}{t_i} > \frac{w_j}{t_j}$, but let's assume that doing j before i is faster. NOTICE how it doesn't actually change the weights or times of either task.

C is completion time of jobs before i and j

(cont.)
Greedy: Our original $\sum_{i=1}^{n} C_i w_i$ is

$w_i(C+t_i) + w_j(C+t_i+t_j)$

Assume: Our swapped $\sum_{i=1}^{n} C_i w_i$ is

$w_j(C+t_j) + w_i(C+t_i+t_j)$

$w_i C + w_i t_i + w_j C + w_j t_i + w_j t_j > w_j C + w_j t_j + w_i C + w_i t_i + w_i t_j$

$w_j t_i > w_i t_j$, but $\frac{w_i}{t_i} > \frac{w_j}{t_j} \Rightarrow w_i t_j > w_j t_i$

so this is CONTRADICTED!

2) NOTICE: that after 24 hours things get complicated
• Iterate thru all processes and split intervals that cross into the next day as 2 intervals
• Sort all intervals (including the midnight ones that are split in 2) based on end time
  ○ For each interval that crosses 12am, we can only select one
    ▪ Choose one midnight-crossing interval at a time
    ▪ Run greedy on the rest of the schedule for all other intervals
    ▪ Record the max. # of intervals you can hold at any one time
• Return max # of intervals

Time Complexity:
• Iterating thru to check for midnight intervals is $O(n)$
• Sorting intervals is $O(n \log n)$
• Checking every midnight interval and running greedy on each is $O(n^2)$

$\boxed{O(n^2)}$ most significant

↑
$O(n \cdot n)$

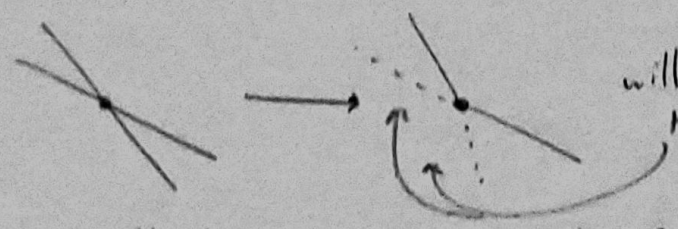
Proof: We don't have to prove greedy again since we did it so in class. Since the midnight intervals can't coexist we can only have 1 of them and use greedy on it w/ the rest of the intervals. This way we cover all possible cases and there can't be a better optimization.

3) • If we EVER have an odd # of cards, we create NULL card that doesn't match with anything
   • Partition all cards into groups of 2
   • Merge a pair of cards together, use the equivalence tester on it
      ○ IF =, move one of the cards into the next layer
      ○ IF diff, move a NULL card to the next layer
      ○ IF one value is a NULL, move the other card to the next layer
   • By the end, the last card should be the card with a majority. If it's NULL then we don't have a majority

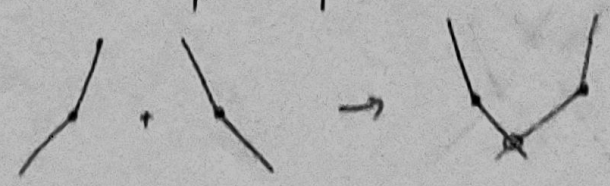
Time Complexity: It's an altered version of mergesort so it's just $\boxed{O(n \log n)}$, we iterate thru n cards each layer and there's only $\log n$ layers.

Proof: Same idea as the Majority Question - when we merge we ALWAYS keep a possible majority card (i.e. if same or if one is NULL we move the other card up). Basically we can't possibly send a minority card as our final card or we can't not show the majority. Since we proved majority in class we don't need to prove it again.

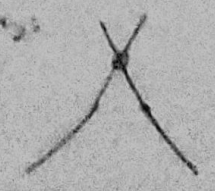
4) • Iterate thru all the lines, for parallel lines delete the one with the lower y-int since the higher one will always hide it
   • Sort all lines by slope
   • Partition all n lines into groups of one
      ○ When merging, we only care abt the visible sections of each line




• Since we sorted the lines we get exactly 1 point of intersection whenever we merge 2 lines:



▸ AT MOST one new graph traverses by 2 visible lines, be you can't end up w/ smth like this that doesn't add any lines:



• After merging is complete iterate thru the final graph and return the unique slopes

Time Complexity:

- Find all // lines is $O(n)$
- Sorting by slope is $O(n \log n)$
- Partitioning is $O(n)$
- Merging is $O(n \log n)$
- Counting one more than for unique slopes is $O(n)$

$T(n) = 3n + 2n\log n$

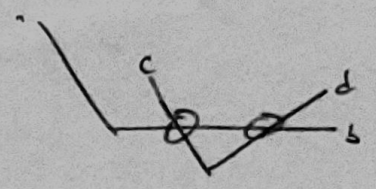
$= O(n \log n)$

Proof: We will only consider intersection points of visible line segments because the ones that are invisible/ lower will never be seen.

- When we merge groups of 2 lines we get exactly 2 intersection pts

CASE 1: We somehow get 0 intersection pts
This cannot happen since we removed all parallel lines beforehand

CASE 2: We get 2+ intersection points
This cannot happen due to the way we sorted the lines by slope then partitioned. For example:




a should've been merged with c because the slopes are closer together than the other lines.

Because we sorted the lines and we got rid of all // lines we considered these 2 cases.

5) n = total # of elements

- Initialize left = 0, right = n-1
- While left < right:
  - mid = $\frac{\text{left + right}}{2}$
  - If array[mid] is greater than both its neighbors, k = mid, return k
  - If array[mid] is greater or equal to array[left], then set left to mid+1
  - Else: right = mid
- return left which should be k at this point

Time complexity: $O(\log n)$ Since we halve the array each iteration and perform an $O(n)$ algorithm

Proof: If we shifted the array, there MUST be an element that is greater than both its neighbors (or else k = 0).

CASE 1: array[mid] > array[mid+1]
Largest element found, since the array is sorted array[mid] is larger than the #'s before it and after thus k = mid

CASE 2: array[mid] < array[left]
Less than 1st element of array, the pivot can't be in the right half of the array so right = mid

CASE 3: array[mid] ≥ array[left]
Greater than, the 1st element, pivot can't be in the left half so set left to mid+1 or = to

6) HAVE TO USE BINARY SEARCH

- Set left1, right1, mid1 as the left, right, and middle pointers to array 1
- Set left2, right2, mid2 as the left, right, and middle pointers to array 2

WHILE array1 OR array2 not fully traversed,

- IF mid1 + mid2 + 1 < k
  - If array1[mid1] < array2[mid2], then k can't be in the left side of array1
    - left1 = mid1 + 1, recalculate mid1
  - ELSE, k can't be in the left of array2
    - left2 = mid2 + 1, recalculate mid2
- ELSE
  - IF array1[mid1] < array2[mid2], then k can't be in the right side of array2
    - right2 = mid2 + 1, recalculate mid2
  - ELSE, k can't be in the right of array1
    - right1 = mid1 + 1, recalculate mid1
- If array1 is fully traversed, binary search the rest of array2 to find k
- If array2 is fully traversed, binary search the rest of array1 to find k

Time Complexity: We binary search both arrays size n and m so our time is O(log n + log m)

Proof:

Prove if array1[mid1] < array2[mid2] then k can't be a left side of array1 by contradiction

Assume k is in the left of array1, then it at most is greater than mid1 + 1 + mid2 elements since array2[mid2] > array1[mid1], but this CONTRADICTS k > mid1 + mid2 + 1

Prove if array1[mid1] < array2[mid2] then k can't be in the right of array2 BY CONTRADICTION

Assume k is in the right side of array2, then at most it must be greater than mid1 + mid2 elements since array1[mid1] < array2[mid2] < kth element. But this CONTRADICTS k ≤ M1 + M2 + 1