Preyasi Gaur Disc 1A

Time: 8:00AM - 9:50 AM

TA: Vincent Li

Computer Science 180

Homework 4

**Question 1**

Algorithm:

* For each job i, we calculate the ratio
* Next we sort the jobs in non-increasing order with respect to their ratios .
* Schedule the jobs in this order.
* Pseudocode:

def schedulejobs(jobs):

for each job in jobs:

job.ratio = job.weight / job.time

sorted\_jobs = sort(jobs, key = lambda job: job.ratio, reverse = True)

current\_time = 0

for job in sorted.jobs:

job.start\_time = current\_time

job.finish\_time = current\_time + job.time

current\_time += job.time

return sorted\_jobs

Time Complexity:

* Most time is taken in the sorting step which is O(nlogn)

Proof:

* The proof that this algorithm minimizes the weighted sum of completion times is based on an exchange argument. Assume that there is an optimal schedule that does not follow the WSPT rule. This means there exist two consecutive jobs, i and j in the optimal schedule such that < (i.e., i has a smaller weight to processing time ratio but is scheduled before j). If we swap these two jobs, the increase in the completion time for job i is less than the decrease in the completion time for job j due to the ratios. This swap reduces the weighted sum of completion times, which contradicts the assumption that we had an optimal schedule. Therefore, by contradiction, the WSPT rule must yield an optimal schedule.

**Question 2**

Algorithm:

* We first split all the intervals across midnight in two parts.
* Next, we will sort the current intervals based on their ending time.
* The point is that for intervals across midnight, only one can be included as they overlap at midnight. For all these kinds of intervals that overlap at midnight, we select one, say j, and exclude all others.
* Then, with j, we can implement the interval scheduling algorithm that we have previously studied.
* For all intervals across midnight, j = 1, 2… k, we do the same thing and record the maximum number of intervals we can carry and return.

Time complexity:

* Splitting the intervals across midnight: O(n)
* Sorting the intervals: O(nlogn)
* Interval Scheduling with sorted intervals: O(n)
* Interval scheduling for all intervals across midnight: O()
* Total: O()

Proof:

* As the intervals across midnight cannot occur at the same time, we pick every one of them and draw a solution independently.
* When we are picking one of them, we run the greedy algorithm for that one interval.
* As we did the greedy algorithm for all intervals across midnight, we have checked all the results, and thus no other solution can be better.

**Question 3**

Algorithm:

* For this problem, we will employ the divide and conquer algorithm.
* We will start by partitioning the whole list having n cards to n lists of 1 card each.
* As we will merge, we will implement an equivalence tester.
* If we have two sets of cards from left side and right side, there are three situations possible:
  + Case 1: No card shows to be the majority between left and right.
    - Here we return no majority to the next level of merging.
  + Case 2: If one card, say Card 1, shows to be the majority of either left or right.
    - There is no majority card on the other side, and then we use the tester to see if Card 1 is the majority among the set of cards we have.
    - If Card 1 is still the majority, we return Card 1, else we return no majority to the next level of merging.
  + Case 3: We have majority cards in both left and right, say Card 1 and Card 2.
    - Here, we will do a double check on the basis of Card 1 and 2 being different or the same.
      * Subcase 1: If they are different.
        + Use a tester to check if Card 1 or Card 2 is the majority, among the set of cards.
        + If either is the majority, return the majority.
        + If neither is the majority, return no majority to the next level of merging
      * Subcase 2: If they are the same.
        + Return either card to the next level of merging.
* If at the final level in which either side has n/2 cards, we will still use the above rule to see for the majority card.
* If our algorithm returns one card, then we get the majority card.
* If nothing is returned then there is no majority card.

Time Complexity:

* Partitioning: O(n)
* For the levels of merging, the worst case would be if we use the tester for both the returned cards from left and right, and for each one we did a linear scan for all the cards in the left and right: O(n)
* We have O(logn) levels.
* Thus, in total we have O(nlogn).

Proof:

* We have n cards → there is a majority card when:
  + The card is the majority card of ether left or right stack
  + The card is the majority for both
* The first statement is easy to prove, and thus we prove the second one using proof by induction.
* Base Case:
  + For n=1, when we have only one card, it is the majority card and we return to the next level.
* Inductive step:
* Assume that for n = k, our algorithm works correctly. Then we would like to show that it works correctly for n = k + 1. Let’s consider the three possible cases:
  + Case 1: Card 1 is the majority card in the left m cards and Card 2 is the majority card of the right side.
    - Here we will just need the tester to compare both cards to (k+1-1) cards.
    - If Card 1 or Card 2 have more than (k+1)/2 matching, then we can return that card → as this is the current majority in k+1 level.
    - We also know from our initial assumption that no card except for Card 1 and 2 can be the majority card as that for a card to be the majority card, it has to be one majority in at least half the cards. As Card 1 and 2 are majorities in left and right respectively, there can’t be any other card with numbers more that k/2 on each side.
  + Case 2: There is only one side that has a majority card → say Card 1 is the majority for the left side, and the right side has no majority.
    - Now here, we need to use the tester compare Card 1 to (k+1-1) cards.
    - If Card 1 has more that (k+1)/2 matching, then we can return Card 1 as it is the current majority in the k+1 level.
    - We also know from our assumption, that there can be no card but A that can be the majority, as we have discussed above.
  + Case 3: There is no majority card on either side.
    - There will be no majority card in the k+1 level.
    - Also according to our assumption, there can be no other card that is the majority.

**Question 4**

Algorithm:

* For this question, we will use the divide and conquer algorithm.
* For all the parallel lines, we will start by deleting the one with the lower y-intercept.
* We delete the one with the lower y-intercept, as the one with the higher y-intercept will always be above it, and thus make it invisible.
* We will then sort all the lines with their slopes in a non decreasing order.
* We will partition them into n groups, each with just one line.
* Now in each partition, the line is visible as it is the only one there. Thus, we treat it as a base case,
* For every partition, we will see the visible lines, the visible intervals and the intersection points.
* As we merge two groups of lines together, more than one intersection point will be found. Here, we will only care about the new intersection points.
* Now as we know that the lines are sorted, the visible intervals from both the sides can only add one new intersection point.
* Before the merge, we have segments represented as ([(seg1, -∞, point1), (seg2, point1, ∞)]\) on one side of the partition and ([(seg3, -∞, point2), (seg4, point2, ∞)]) on the other side. After merging, the outcomes will vary based on the relationships among different points, denoted as (pointA), (pointB), and (pointC). For example, if (pointC < pointA) and (pointC < pointB), the new sequence will be ([(seg1, -∞, point1), (seg2, point1, pointC), (seg3, pointC, point2), (seg4, point2, ∞)]), as described above. Conversely, if (pointC > pointA) and (pointC > pointB), we would end up with a sequence such as ([(seg1, -∞, point1), (seg3, point1, ∞)]), and so on for other possible relationships among the points.
* There can be other situations with different relationships among pointA, pointB and pointC, but at maximum the list will increase two segments, as one new intersection point is being added.
* After all the merge steps, the returned recorded list is the answer.

Time Complexity:

* Partitioning: O(n)
* Merging at each level: O(n)
* Thus, T(n) = T(n/2) + T(n/2) + Cn. Solving this we get, T(n) → O(nlogn)

Proof:

* Here we only notice the intersection point caused by the visible line segments. We do this as the line segments that are not visible right now, will never become visible in the future.
* Moreover, the future visible line segments will inevitably have a higher y intercept as compared to the current invisible line segments, given the same x coordinate.
* There will also only be one new intersection point from two groups, as we initially got rid of all parallel lines in the very beginning. So there will not be a case when no new intersection point is added.
* Moreover, as we have initially sorted all the lines by slope, then partitioned and merged, there cannot be a case where as we merge, we will add more than one intersection point.

**Question 5**

Algorithm:

* To solve this problem, we can employ binary search.
* We start by initiating three variables:
  + l: to track the leftmost element
  + m: to track the middle element
  + r: to track the rightmost element
* Now let’s consider the possible cases.
  + Case 1: If arr[l] > arr[m]
    - This means that the inversion is done in the first half (upto m) of the array.
    - Thus, the r pointer is moved to the element currently pointed by m and m is pointed to the middle of l and r
  + Case 2: If arr[r] < arr[m]
    - This means that the inversion is done in the second half of the array (m → r).
    - Thus, the l pointer is moved to where m was pointing ad m is moved to the midpoint of r and l.
* We will continue this until we get
  + arr[m-1]>arr[m] or,
  + arr[m]>arr[m+1]
* Thus, m is the value we want.

Time Complexity:

* Halving the number of elements: O(logn) times
* Constant time checking in each half: O(1)
* Total: O(logn)

Proof:

* We will prove this case by case.
* If our array shifts, then this means that there is a pair of neighboring elements where one is the biggest in our array, and one is the smallest.
* In each step, we check if arr[m] < arr[l] or if arr[m] > arr[r], then we know that in the respective half, there is a pair of neighboring elements with order reversed.
* Thus, we continuously halve the subarray that we will be searching for, to find the pair of neighbor elements, and subsequently the index of the element in the left is the number of steps the array has been shifted, i.e. k.

**Question 6**

Algorithm and Proof:

* To solve this problem, we can employ binary search.
* Here, we will keep 6 variables such that:
  + For the first array:
    - l1: to track leftmost element
    - m1: to track middle element
    - r1: to track righmost element
  + For the second array:
    - l2: to track leftmost element
    - m2: to track middle element
    - r2: to track righmost element
* Now assume that the
* Suppose that we are given an input of two arrays: arr1 and arr2
* if m1 + m2 + 1 < k:
  + if arr2[m2] > arr1[m1], then p cannot in the left partition of arr1.
    - Proof by contradiction: Say p is on the left of arr1[m1].
    - In this case, p can at maximum be more that m1 + m2 - 1 elements as p < arr1[m1] < arr2[m2].
    - This is contradictory as m1+m2+1 < k. Thus we can simply ignore the first half of arr1, and rerun the algorithm. When we rerun we will make the input arr1[m1+1: end] and arr2.
    - else, similar to above, p cannot be in the left of arr2[m2]. Then again, we will just let the first half of arr2 be, and rerun the algorithm with the input arr1 and arr2[m2+1 : end].
* else:
  + We use similar to what we did in the previous case, except here we do for the left side.
  + if arr1[m1] < arr2[m2], then p cannot be in the right of arr2[m2]. Similar to above, we will prove this using contradiction.
    - Suppose p is in the right of arr2[m2].
    - Then it must be greater than m1+m2 elements in total as we know that x > arr2[m2] > arr1[m1].
    - This is contradictory as m1+m2+1 k.
  + Thus we can disregard the second half of arr2, and rerun the algorithm with inputs as arr1 and arr2[0:m2+1]
  + Else, similarly to above, p can’t be in the right of arr1[m1]. Thus, we disregard the second half of arr1, and rerun the algorithm with the inputs arr1[0:m1+1] and arr2.
* If either array has been completely scanned, then we just need to employ a binary search for the remaining elements of the array until we find p.

Time Complexity:

* We have to split the number of elements everytime, in which the worst case is splitting them by O(logn + logm) times.
* For every half, checking time: O(1)
* Total time: O(logn + logm)