Homework 5

Due Wednesday, October 7 at 11:59 PM

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

1. (**Photography shop**)

Suppose you work at a photography shop, and at the start of your workday you are given a list of jobs $i = 1, 2, \ldots, n$, each with a processing time p_i and a "touchiness" factor $t_i > 0$ (corresponding to how mad the customer will get if job i is delayed). Your pay for finishing job i at time f_i is $(P - f_i)/t_i$, where P is the total processing time of all jobs ($P = p_1 + p_2 + \ldots p_n$). Assume you have enough time to finish all n jobs, but you get to choose the order in which you do them. You can only do one job at a time.

The algorithmic problem is to compute an order that maximizes your total payment.

For example, suppose you have three jobs as follows:

i	p_i	t_i
1	4.5	10
2	1.25	7
3	9	3
4	5	6

Then, the order $(1, 2, 3)$ would have finishing times $f_1 = 4.5$, $f_2 = 5.75$, $f_3 = 14.75$ and a total payoff of $4.25/10 + 9/7 + 0/3 = 1.71$ On the other hand, the ordering $(3, 1, 2)$ has finishing times $f_3 = 9, f_1 = 13.5, f_2 = 14.75$ with total payoff $5.75/3 + 1.25/10 + 0/7 = 2.07$

An idea is to find some quantity by which the tasks can be sorted, so that the optimal order is the order sorted by this quantity.

(a) Consider each of the following orders in which to schedule jobs. For **three out of the four** of them, give a counterexample showing that it does not produce the optimal task schedule:

i. Decreasing order of processing time p_i

ii. Decreasing order of touchiness t_i

iii. Increasing order of *product* of touchiness to processing time $t_i p_i$

iv. Increasing order of *sum* of touchiness and processing time $t_i + p_i$

(b) Give a polynomial-time greedy algorithm for this problem. [An exchange algorithm works well for the proof of correctness.]

ANSWER

a) If we consider the table above then add a final value 4 with $p_i = 5$ and $t_i = 6$

i. This example has order 3 4 1 2, and payout of 4.67, so it is outclassed by all other events here listed.

ii. This example has order 1 2 4 3, and payout of 5.06, so it is outclassed by both iii and iv.

iii. This example has order 2 3 4 1, and payout of 6.56, so it is not outclassed by any other events in the pile.

iv. This example has order 2 4 3 1, and payout of 6.39, so it is outclassed by iii.

b) Based on the previous accounts, the best strategy is to organize them in increasing order of the product of touchiness with processing time, so what the algorithm below does is get the array of job objects (each object is something like this: $\{i, p, t\}$) loop through each job and add them to a priority queue that is sorted based on the product value. In addition to this, while I'm in this look, I add up all the processing times to keep for later calculations.

Then find the total time, and by popping the queue it makes the earning calculations. In addition to this, it makes an ordered array of the job IDs that is also returned. It should be noted I make use of .insert to put the values into the priority queue (in the correct places).

The algorithm itself is pretty basic and makes use of the PriorityQueue data structure to assure it is always correctly organizing the jobs. As for runtime, it looks through jobs twice, so the runtime complexity is $O(2n)$, as all other operations are $O(1)$, so they do not matter.

Algorithm 1: MostIncome(arr)

Input: *arr*

```
queue = PriorityQueue < Jobs > (Comparator = (a, b) => (a.p × a.t) > (b.p × b.t)?1 :
  −1);
P = 0 for each v ∈ arr do
    queue.insert(v);
    P = P + v.p;
orderedArr = [];
payout = 0;
currentTime = 0;
while queue.size() != 0 do
    v = queue.pop;
    currentTime = currentTime + v.p;
    orderedArr.insert(v);
    payout = payout + ((P − currentTime)/v.t);
```

Output: *orderedArr, payout*

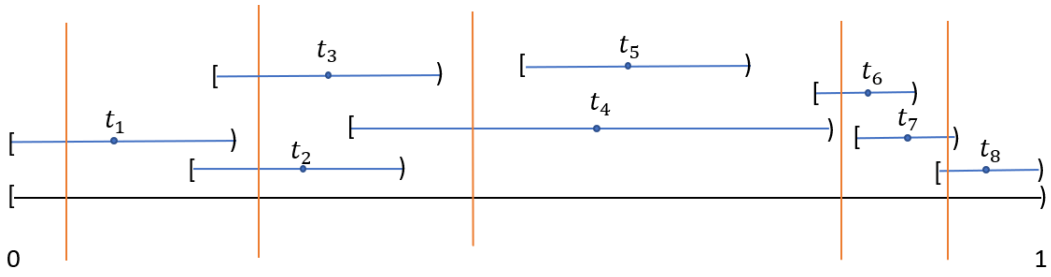
Hint: You may want to analyze the situation of two tasks that are scheduled next to each other: say $(2, 3)$. Under what condition on their parameters p_2, t_2, p_3, t_3 will be worth changing their order to $(3, 2)$? What is the sorting quantity suggested by this condition? (Compute the difference in payment resulting from the swap.)

2. (**Cell towers**) Suppose you are back on the long country road from Lab 3, which we can think of as a line segment with western and eastern endpoints. Let's identify points on the road with the interval $[0, 1)$. Now, suppose that a telecommunications company has invested in cell coverage for this road: they've placed n towers, each of which can cover a different radius in specific locations along this road such that every segment of the road is covered by *at least* one tower. We'll say $t_i \in [0, 1)$ is the location of the i-th tower along the road, and it

has radius r_i. For technical reasons, tower i's interval of coverage will be open on the right side, giving $[t_i - r_i, t_i + r_i)$.

As a surveyor for this telecommunications company, your job is to find the *maximum size set of points along the road* $[0, 1)$ *such that each tower serves at most one point in this set*; we'll call a set with this property *sparse*. The input for this problem is the list of pairs (t_i, r_i), for $i = 1, ..., n$.

Below is an example of a set of towers and orange lines indicating a sparse set. (Do not hand in) Is it the largest possible sparse set?



(a) Your friend suggests the following greedy approach: sort the intervals according to their centers t_i. Moving from left to right, add t_i (the position of tower i) to your set as long it it doesn't conflict with the previous ones (meaning no interval covers both t_i and a previously selected center). Give an example showing why this will not produce the largest sparse set. [NB: this algorithm chooses tower locations, but a sparse set doesn't have to be limited to tower locations—any subset of $[0, 1)$ is ok.]

Before we design an algorithm for finding sparse sets, let's consider a related problem: as before, you are again given the fixed locations and radii of n towers, but now suppose that the telecommunications company has decided that, in the interest of profits, they only want to keep enough cell towers running so that every point on the road $[0, 1)$ is covered by at least one tower. We will call such a set of towers *sufficient*. It's now your job to tell them the *minimum* number of towers they need to power so that the whole $[0, 1)$ interval is covered (i.e. find a sufficient set of minimum size).

(Do not hand in) How small a set of sufficient intervals can you find for the example pictured above?

(b) Prove that if there exists a sparse set $\mathcal{P}$ of locations with $|\mathcal{P}| = k$, we need at least k towers to cover $[0, 1)$ (that is, every sufficient set $\mathcal{T}$ of towers satisfies $|\mathcal{T}| \geq k$).

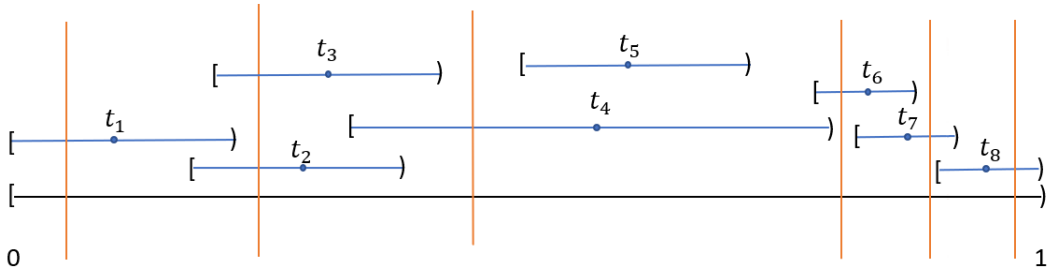
(c) Give a polynomial-time algorithm that takes as input a list of pairs (t_i, r_i) and produces both a sparse set of positions and a sufficient set of intervals, such that the two sets have the same size. (Remember to follow the usual guidelines for algorithms.)

(d) Argue that the algorithm from (c) finds both a sparse set of maximum possible size, and a sufficient set of the smallest possible size.

ANSWER

(a) If we use the set given to us as an example in this PS, and follow the suggested algorithm, we'll place the first point at t1, then the second at t2, then the third at t4, then the fourth at t6, and then the fifth at t8. All together we have 5 points, which, although it is the same amount as that which is displayed, it is not the the most amount of points we can reach.

If we take a look at the changed picture below, we can see that there is enough space between t6 and t8 inside t7 to put another point so that we can then relocate the point in t8 to later on, so still fulfilling the requirements and having 6 points.



(b) Let us assume we have a case where we have 1 tower, then we can have at most 1 point, so the statement $|\mathcal{P}| = 1$, $|\mathcal{T}| \geq 1$.

Now lets assume this statement holds for any n number of towers, then we have that $|n| \geq k$, so if we add 1 tower, one of two thing can happen, either we are able to add a brand new point (ie: the tower isn't completely intercepted by other radius), so we now have that $|n+1| \geq k+1$, which is of course also true. Or, on the other hand, we can have a tower that is totally intercepted by other radius, so we have that $|n+1| \geq k$, now since we know that n is already greater or equal than k, then n+1 will always be greater than k.

c) In order to produce the sparse set, what my algorithm would do is go from left to right, and as soon as it finds a new point on a given radius that hasn't been occupied, it would place it, but only if there it occupies no other radii. If after the first loop there are still unoccupied radii, then it runs again attempting to at most, occupy one other radii, and never one that already has been occupied. It could make use of a queue to manage with radii have and haven't been occupied. It would make use of another array data structure to track the used points. Since we couldn't possibly search all points from [0,1), what we should do is search by halves, searching first the left half, then the right half, then the left half of the left half and hence onwards, sort of like a Pre Order Search, thus giving us a runtime of O(n+m)

As for the the sufficient set, it would just go left to right and remove any tower's who's radius is completely encapsulated by another radius, thus leaving only towers who are needed to cover any given point on the range. It can do this by making a copy of the array with the place towers, and then loop through the actually given array and if it finds overlaps it deletes them from the copy array. Returning whatever is left. This would allow for a O(n) search

If we do the tasks one at a time, the runtime would be O(2n+m) or O(n+m). With n being the towers and m the radii.

As for the proof of correctness, the algorithm by design will always return the largest sparse set, as it tries minimazing the impact of any single point, does making sure it place the most amount of points. As proven before the size of this will be the same as the size of the minimum required towers, so it also fulfill that requirement. As for how it gets the minimum set itself, simply checking to make sure if any completely enclosed towers exist does the trick, as it only leaves towers that have at least 1 point where they are the only service access tower for it.

d) As I mentioned in the previous comment, the largest possible sparse set is that in which each point individually reaches the minimum amount of radii as possible, since it maximizes the amount I can fit into the timeline. As such the algorithm will produce the biggest sparse set.

As for the smallest necessary set of towers, that is also obtained by the algorithm as what it does is remove any extra towers that only serve points that are already serviced, IE: the entire radius of the tower is enclosed in other radius'. Thus keeping only the absolute essential towers to service the road.