CS 577: Introduction to Algorithms Fall 2022

Homework 5

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers the greedy-stays-ahead-paradigm. **Problem 3 must be submitted for grading by 2:29pm on 10/25**. Note that you have two weeks for this assignment because of the first midterm exam. Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

1. You are given a knapsack with a weight limit of W and n items with nonnegative weights $w_1, w_2, ..., w_n$. You want to fill your knapsack as close to the weight limit W as possible but without exceeding it. You can see this as a specific case for the general knapsack problem for which each item has the same value as its weight.

   Consider the case where the weight of each item is at least as large as the weight of all previous items combined, i.e. $w_i \geq \sum_{j=1}^{i-1} w_j$ for each $1 < i \leq n$. Design an algorithm to solve this problem that performs no more than $O(n)$ elementary operations. Arithmetic operations like the addition of two numbers and the comparison of two numbers are considered elementary for this problem. Your algorithm should output an optimal list of items to put in your knapsack.

2. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box which arrived later than their own make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

   They wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking: Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

   Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed.

Regular problems

3. [Graded] Given n half-closed intervals $(s_1, f_1], (s_2, f_2], \ldots (s_n, f_n]$, we are looking for the smallest number of points in $\mathbb{R}$ such that each of the given intervals contains at least one of the points.

   (a) You decide that you should pick your first point to be one that fall inside the largest number of intervals. Then you remove these intervals and pick your second point with the same rule. You continue until no more intervals remain. Construct a counter-example where this greedy strategy fails to compute an optimal solution.

   (b) Design a greedy algorithm that solves the problem in $O(n \log n)$ time.

4. Your summer job is to drive kayakers from the parking lot to the kayak launch platform. The bus you are driving can take up to k kayakers; a round-trip between the parking lot and the launch platform lasts m minutes. You are given an alphabetical list with the names of all n kayakers for a given day, together with the times they will arrive at the parking lot. The kayakers need to be served in the order they arrive but you can decide for each roundtrip when the bus leaves and how many kayakers you take along.

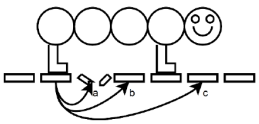
   You would like to organize your schedule in such a way that you are donefor the day as early as possible. Design an algorithm that computes the earliest time you can be done in $O(n \log n)$ steps.

   For example, if $k = 20$, $m = 30$, and there is only a group of 25 people that day, all arriving at noon, then the earliest you can be done is 1:00pm; if in addition there is a group of 5 people arriving at 12:45pm, then the earliest you can be done is 1:15pm.

5. For the opening scene of a computer game, you want the main character, Wormly, to cross a bridge. Wormly is a worm made of k equal circular bubbles and ℓ legs. At all times each leg has to be under one of the bubbles, and under each bubble there can be at most one leg. The bridge was supposed to be composed of n planks with the width of each plank equal to the diameter of each of Wormly's bubbles. However, some of the planks are missing.

   At every moment, Wormly can do exactly one of the following:

   - Move one of its legs forward over any number of (possibly missing) planks. After the move, the leg should be on a plank and underneath one of Wormly's bubbles. A leg isn't allowed to overtake other legs.
   - Move all of its bubbles forward one plank while its legs remain on the same planks. After the move each leg must still be under one of Wormly's bubbles.

   In the above figure, the only possible move for the last leg is to position b. This is because the plank at position a is missing, so the leg cannot move there; to get to position c, the last leg would have to overtake the first leg. Also, in this example, moving all the bubbles forward is not allowed because Wormly's last leg would end up without a bubble over it.

   Initially Wormly's bubbles are directly above the leftmost k planks of the bridge and its legs are on the leftmost ℓ planks. At the end of the animation Wormly's bubbles have to be directly above the rightmost k planks and its legs have to be on the rightmost ℓ planks. The left- and rightmost ℓ planks of the bridge are not missing.

   Design an algorithm to determine the smallest number of steps for the animation when given zk, ℓ, and a binary string of length n where the ith bit indicates whether the ith position has a plank. Your algorithm should run in time $O(n)$.

Challenge problem

6. In some courses you can choose a certain number k of the n assignments that will be dropped in the calculation of your grade. If all the assignments counted equally, the choice would be easy: simply drop the assignments with the lowest scores. However, each assignment may have a different maximum score. Your final homework grade will be the percentage ratio of your total score to the maximum possible score for the retained assignments. This leads to the following problem. You are given a value of k and a list of n assignment results (s_i, m_i), $1 \leq i \leq n$, where s_i denotes your score on the ith assignment and m_i denotes the maximum possible score on that assignment. Your goal is to find a set $I \subseteq \{1, 2, \ldots, n\}$ with $|I| = k$ such that $\sum_{i \notin I} s_i / \sum_{i \notin I} m_i$ is as large as possible.

   Design an algorithm that runs in time $O(n^2 \log n)$. For starters, aim for an algorithm that runs in time polynomial in the number of bits in the input.

Programming problem

7. SPOJ problem I AM VERY BUSY (problem code BUSYMAN).

CS 577: Introduction to Algorithms Fall 2022

Homework 5 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

You are given a knapsack with a weight limit of W and n items with nonnegative weights $w_1, w_2, ..., w_n$. You want to fill your knapsack as close to the weight limit W as possible but without exceeding it. You can see this as a specific case for the general knapsack problem for which each item has the same value as its weight.

Consider the case where the weight of each item is at least as large as the weight of all previous items combined, i.e. $w_i \geq \sum_{j=1}^{i-1} w_j$ for each $1 < i \leq n$. Design an algorithm to solve this problem that performs no more than $O(n)$ elementary operations. Arithmetic operations like the addition of two numbers and the comparison of two numbers are considered elementary for this problem. Your algorithm should output an optimal list of items to put in your knapsack.

A greedy approach to solve this problem is to try to fit the heavier items in the knapsack first and then use the lighter items to get as close as possible to W. More precisely, we consider the items in order of non-increasing weight and put them in the knapsack unless that would violate the weight limit W.

This greedy approach works because of the restriction that an item must weigh as much as all the previous items combined. It is a good exercise to try and show that this approach fails in general by finding a counterexample. Now consider the following greedy algorithm:

Algorithm 1

Input: n items with non-negative weights $w_1, w_2, \ldots, w_n$ such that $w_i \geq \sum_{j=1}^{i-1} w_j$ for each $1 < i \leq n$; non-negative number W.
Output: K, an optimal subset of items for a knapsack with weight limit W.

```
1: procedure FillKnapsack(w_1, w_2, ... w_n, W)
2:     K ← ∅
3:     w ← 0
4:     for i = n to 1 do
5:         if w + w_i ≤ W then
6:             K ← K ∪ {i}
7:             w ← w + w_i
8:     return K
```

To argue correctness, notice that our greedy algorithm always produces a valid solution on valid inputs, i.e., a subset of the items whose total weight does not exceed the limit W. To prove optimality we develop a similar argument as the one we used for interval scheduling – we show that our greedy algorithm stays ahead of any other algorithm.

Consider any valid solution S (i.e. a subset of the items that does not exceed the weight limit). If our algorithm packs the same items in the knapsack as S ($K = S$) then we are done. If not, consider the items in the order as our algorithm tried to put them in, and let i denote the first

item for which our greedy algorithm makes a different decision than S. Since the greedy algorithm always puts an item in if there is room for it and S agrees with all decisions the greedy algorithm has made before, it has to be the case that the greedy algorithm puts item i in the knapsack whereas S does not.

Let us call the weight of the knapsack just before S and our algorithm disagree w. Then after the disagreement, the weight of the knapsack which our algorithm produces is $w+w_i$. Now, whatever S decides to do with the remaining items, it cannot add more weight than the combined weight of the remaining items, i.e., $\sum_{j=1}^{i-1} w_j$. Therefore, the solution S can produce a knapsack of weight at most $w+\sum_{j=1}^{i-1} w_j \leq w+w_i$, where the last inequality follows from the input restriction. Therefore, our algorithm stays ahead of S. The algorithm performs a constant amount of elementary operations per item which sum to a total of $O(n)$ elementary operations.

Problem 2

You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box which arrived later than their own make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

They wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking: Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed.

Suppose n boxes arrive in the order $b_1, b_2, ..., b_n$ and each box b_i has weight w_i, where $w_i > 0$. To preserve the arriving order of boxes, the greedy algorithm assigns each box to one of the trucks $T_1, T_2, ..., T_N$ such that:

- No truck is overloaded: the total weight of all boxes in each truck is no larger than the weight limit W.
- The arriving order is preserved: if box b_i arrived earlier than box b_j (i.e. $i < j$), then box b_i must be sent before box b_j. In other words, if box b_i and box b_j are assigned to truck T_x and T_y, respectively, it must be the case that $x < y$.

We prove that the greedy algorithm uses the minimum amount of trucks for sending boxes $b_1, b_2, \ldots, b_n$ by applying the greed-stays-ahead framework via the following claim.

Claim 1. *If the greedy algorithm fits boxes $b_1, b_2, ..., b_j$ into the first k trucks, and an arbitrary solution S fits boxes $b_1, b_2, ..., b_i$ into the first k trucks, then $j \geq i$.*

Proof. We prove the result by induction on k.

1. Base case, $k = 1$. In this case, since the greedy algorithm fits as many boxes as possible into the first truck, it cannot be the case that an arbitrary solution S fits more boxes into the first truck than the greedy algorithm. We conclude $j \geq i$ and the claim holds.
2. Inductive step, $k > 1$. In this case, let $b_1, b_2, \ldots, b_{j'}$ and $b_1, b_2, \ldots, b_{i'}$ be the boxes fit into the first $k-1$ trucks by the greedy algorithm and an arbitrary solution S, respectively. By the induction hypothesis, since $k-1 < k$, we have that $j' \geq i'$. Then, the greedy algorithm puts boxes $b_{j'+1}, b_{j'+2}, \ldots, b_j$ (total of $j-j'$) into the k-th truck and S puts boxes $b_{i'+1}, b_{i'+2}, \ldots, b_i$ (total of $i-i'$) into the k-th truck. Since $i' \leq j'$, the greedy algorithm G can at least fit boxes $b_{j'+1}, b_{j'+2}, ..., b_i$ in the k-th truck, totaling at least i boxes in all k trucks. Thus $j \geq i$.

□

Claim 1 then implies the optimality of the greedy algorithm by setting k to be the number of trucks used by the greedy algorithm.

CS 577: Introduction to Algorithms Fall 2022

Homework 5 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 3

Given n half-closed intervals $(s_1, f_1], (s_2, f_2], \ldots (s_n, f_n]$, we are looking for the smallest number of points in $\mathbb{R}$ such that each of the given intervals contains at least one of the points.

(a) You decide that you should pick your first point to be one that falls inside the largest number of intervals. Then you remove these intervals and pick your second point with the same rule. You continue until no more intervals remain. Construct a counter-example where this greedy strategy fails to compute an optimal solution.

(b) Design a greedy algorithm that solves the problem in $O(n \log n)$ time.

Part (a)

A counter example is

$$L = [\ [(0,2],\ (1,4],\ (1,4],\ (1.5,4],\ (3,5],\ (3,5],\ (4,6],\ (4,7]\ ]$$

The proposed greedy strategy in this case would first pick an element in the intersection of $(1,4], (1,4], (1.5,4], (3,5], (3,5]$, e.g. $p_1 = 3.5$. Then it would pick the next point $p_2 \in (4,6] \cap (4,7]$, e.g. $p_2 = 5$ and finally some $p_3 \in (0,2]$, $p_3 = 0.5$. The optimal number of visits in this example is 2, for example $p_1 = 2$ and $p_2 = 5$.

Part (b)

Intuitively, for this problem it makes sense to place the first (or leftmost) point as far to the right as possible in the number line. After placing this point, we would like to be able to worry only about placing points to its right, which means that this first point needs to cover all intervals to its left. This suggests the following greedy strategy.

Find the leftmost right endpoint of an uncovered interval, and place a point at this position.

Correctness Similar to the interval scheduling problem seen in class, there are different ways to view this problem. If we consider the input intervals as the components, then a natural greedy-stays-ahead claim is that the number of points required to cover the first i intervals used by the greedy strategy is less than or equal to the number of points required to cover the same intervals for any other solution S. Optimality of the greedy strategy then follows right away. To state this claim more formally, we introduce some notation. Let $L = [(s_1, f_1], (s_2, f_2], \ldots, (s_n, f_n]]$ be the list of half-open intervals sorted by non-decreasing right endpoint. Let also L_i be the union of the first i intervals in this order (or all of them if $i > n$). Finally. let $G = (g_1, \ldots, g_k)$ be the list of points in the greedy solution and $S = (s_1, \ldots, s_\ell)$ be an arbitrary solution (both in nondecreasing order). The claim can then be stated as

Claim 1. *For any valid solution S and any positive integer i, $|G \cap L_i| \leq |S \cap L_i|$*

Proof. We prove the claim by induction on i. The base case for $i = 1$ holds because G uses only a single point to cover the first interval, and any other solution needs to use at least one point for that as well. For the inductive step, let $i > 1$ and notice that the inductive hypothesis guarantees that $|G \cap L_{i-1}| \leq |S \cap L_{i-1}|$. Then, if G does not need an additional point to cover the i-th interval, we are done since $|G \cap L_i| = |G \cap L_{i-1}| \leq |S \cap L_{i-1}| \leq |S \cap L_i|$. On the other hand, if $|G \cap L_i| = |G \cap L_{i-1}| + 1$, then we show that $|S \cap L_i| > |S \cap L_{i-1}|$. Since G needs an extra point to cover interval $(s_i, f_i]$, it must be the case that $s_i \geq g_m$, where $m \doteq |G \cap L_{i-1}|$. Moreover, by G's operation it is also the case that $g_m = f_j$ for some interval $(s_j, f_j]$ with $1 \leq j < i$, and the inductive hypothesis with parameter j guarantees that S needs at least m points to cover the first j intervals, and thus uses at least m points at or before f_j. As $f_j = g_m \leq s_i$, none of these points can cover $(s_i, f_i]$ and S needs at least $m + 1$ points to cover L_i. □

Alternatively, we may consider the output points as the components. In this case, our notion of "ahead" is that the i-th point in the greedy solution is greater than or equal to the i-th point in any solution S. Recall that we defined $G = (g_1, \ldots, g_k)$ as the list of points in the greedy solution and $S = (s_1, \ldots, s_\ell)$ as the list of points for an arbitrary solution S, both sorted in nondecreasing order. To make our proofs smoother, we also define $g_0 = s_0 = -\infty$ and $g_i = s_j = \infty$ for $i > k$ and $j > \ell$. Our "greedy-stays-ahead" claim is then as follows.

Claim 2. *For any valid solution S and any positive integer i, $g_i \geq s_i$.*

Before we prove the claim we note that the optimality of G follows from Claim 2 as it implies that if some valid solution S consists of less than i points (indicated by $s_i = \infty$) then so does G.

Proof. We establish the claim by induction on i

1. Base case, $i = 0$. This trivially holds because $g_0 = s_0 = -\infty$.
2. Induction step, $i > 0$. The key observation is that for any valid solution S, the intervals that are not yet covered by the first i points (in the order considered) are exactly those whose left endpoint is at or after s_i. This is true because we consider the two solutions G and S in non-decreasing order of right endpoints. If an interval starts before s_i and is not covered by the first i points, this means that this interval will remain uncovered in the end. From the inductive hypothesis, we have that $g_{i-1} \geq s_{i-1}$. Using the above observation, we have that all intervals of G (resp. S) that remain uncovered after considering its leftmost $i - 1$ points start at or after g_{i-1} (resp. s_{i-1}). Since $g_{i-1} \geq s_{i-1}$, the set of uncovered intervals of G after considering its $i - 1$ leftmost points is a subset of the uncovered intervals of S after considering its $i - 1$ leftmost points. Then, because g_i is, by definition, the right endpoint of the first interval still uncovered by G, if $s_i > g_i$ then there exists a interval that S does not cover, which is a contradiction since we assumed that S is a valid solution. This means that $g_i \geq s_i$.

□

Implementation and runtime We now show how to implement the above greedy strategy. We first sort the list L of tuples $(s_i, f_i]$, in non-decreasing order of f_i. This step takes time $O(n \log n)$. We can then find the optimal number of points by doing a linear scan of the list where we keep track of the last point placed, which we update if we find some interval starting at or after that point. This step takes only $O(n)$ time. Therefore, the overall run time is $O(n \log n)$. We also present pseudocode for this solution in Algorithm 1.

Algorithm 1

Input: A list L of n tuples $(s_i, f_i]$.
Output: The smallest number of points in $\mathbb{R}$ that covers all intervals.

```
1: procedure CoverPoints((s_1, f_1], (s_2, f_2], ..., (s_n, f_n])
2:     Sort L in non-decreasing order of finish time
3:     p ← −∞
4:     k ← 0
5:     for i = 1 to n do
6:         if s_i ≥ p then                ▷ If (s_i, f_i] is not covered by point p
7:             p ← f_i                    ▷ Then update the value of p to f_i
8:             k ← k + 1
9:     return k
```

Alternate solution This problem is actually dual to the interval scheduling problem from class: the minimum number of points equals the maximum number of pairwise disjoint intervals. That the former is at least the latter follows from the definitions; the other inequality is harder to argue but also holds. One possible argument uses a result that we will cover later in class, namely that the maximum number of edge disjoint paths from s to t in a digraph equals the minimum number of edges that need to be removed to make it impossible to go from s to t. Once the equality is established, it suffices to call the algorithm from class for interval scheduling. However, rigorously establishing the equality does not seem simpler than solving the problem from scratch as in the above model solution.

Problem 4

Your summer job is to drive kayakers from the parking lot to the kayak launch platform. The bus you are driving can take up to k kayakers; a round-trip between the parking lot and the launch platform lasts m minutes. You are given an alphabetical list with the names of all n kayakers for a given day, together with the times they will arrive at the parking lot. The kayakers need to be served in the order they arrive but you can decide for each roundtrip when the bus leaves and how many kayakers you take along.

You would like to organize your schedule in such a way that you are done for the day as early as possible. Design an algorithm that computes the earliest time you can be done in $O(n \log n)$ steps.

We build a greedy strategy around the following idea: once we have decided when a bus trip leaves, filling any extra room with kayakers who show up before that time doesn't delay departure. Our greedy strategy is to consider the kayakers in order from latest arrival time to earliest; we keep grouping together the next k to form a bus trip until we have no more than k left, who will form the earliest bus trip. This strategy stays ahead of any other strategy in the following sense.

Claim 3. *For each bus trip of our greedy schedule, the last person on that trip leaves the parking lot no later than in any other schedule.*

Proof. For the first bus trip, this follows immediately — the bus can leave as soon as this last person arrives, a situation which cannot be improved with respect to that individual. Now assume that Claim 3 holds for the $(i-1)^{th}$ bus trip, and consider the i^{th} bus trip. Let P_i be the last person on the i^{th} bus trip, and P_{i-1} be the last person on the $(i-1)^{th}$ bus trip. If the i^{th} trip leaves as soon as P_i arrives, our claim holds for the same reason as in the base case. Otherwise, the trip was delayed because of having to wait for the bus to return from the preceding trip. Now, since our greedy strategy fills the bus on every trip after the first, there are $k - 1$ kayakers between P_i and P_{i-1}. Thus, any schedule must place P_{i-1} on an earlier bus than P_i . But by our induction hypothesis, P_{i-1} cannot have been on a bus trip which left earlier than in our greedy schedule, so any schedule must have a bus trip which leaves at least as late as the bus before the one that P_i is on in our greedy schedule. Hence, it is impossible for P_i to be on a bus trip that leaves earlier. □

The optimality of the algorithm follows from Claim 3. It implies that the last person on the last bus cannot have been on a bus trip that left earlier, so we cannot be done earlier.

If we have grouped the kayakers using the given greedy strategy, we can compute the departure time of each bus trip by looking at the last kayaker on each of them; every bus trip leaves either when the last kayaker on it arrives, or m minutes after the previous bus trip left, whichever is larger. The dominating factor in our algorithm is the initial sort — everything thereafter can be computed with a linear scan of the kayakers; hence, the runtime for this algorithm is $O(n \log n)$.

Problem 5

For the opening scene of a computer game, you want the main character, Wormly, to cross a bridge. Wormly is a worm made of k equal circular bubbles and ℓ legs. At all times each leg has to be under one of the bubbles, and under each bubble there can be at most one leg. The bridge was supposed to be composed of n planks with the width of each plank equal to the diameter of each of Wormly's bubbles. However, some of the planks are missing.

At every moment, Wormly can do exactly one of the following:

- Move one of its legs forward over any number of (possibly missing) planks. After the move, the leg should be on a plank and underneath one of Wormly's bubbles. A leg isn't allowed to overtake other legs.
- Move all of its bubbles forward one plank while its legs remain on the same planks. After the move each leg must still be under one of Wormly's bubbles.

Initially Wormly's bubbles are directly above the leftmost k planks of the bridge and its legs are on the leftmost ℓ planks. At the end of the animation Wormly's bubbles have to be directly above the rightmost k planks and its legs have to be on the rightmost ℓ planks. The left- and rightmost ℓ planks of the bridge are not missing.

Design an algorithm to determine the smallest number of steps for the animation when given k, ℓ, and a binary string of length n where the ith bit indicates whether the ith position has a plank. Your algorithm should run in time $O(n)$.

The greedy strategy for this problem is simply to move the legs as far forward as possible, and then to move the body as far forward as possible, repeating until the end is reached or we detect that no further progress is possible. There are two main parts to this solution: the proof that the greedy strategy is correct, and the actual algorithm which calculates the number of steps the strategy takes. The latter part is not trivial, since the number of steps in the greedy strategy can be $\Omega(n^2)$, for example, when n is even, $k = n/2, \ell = k - 1$, and all planks are present. So, merely simulating the legs' movements would result in an algorithm that is not $O(n)$.

Optimality of the greedy strategy We use the greedy stays ahead scheme for proving the optimality of the greedy strategy. We don't need to worry about the cost of the moves that move the bubbles forward, since this cost will be the same no matter what strategy we use (assuming there is a solution). We consider each of the contributions to the cost of getting to the end by each leg. We claim that these contributions per leg are minimized by the greedy strategy. Our claim is as follows.

Claim 4. *For every leg and every integer $i \geq 0$, the position of the leg after the i-th time it is moved is at least as far under the greedy schedule as in every other valid schedule.*

Proof. We only need to consider schedules in which each leg is moved at most once between two bubble moves, and either all legs or no legs are moved, and if moved they are moved in order from the first to the last. If we are given a schedule where this is not true we can convert to this kind of schedule by combining the moves of the same leg that occur between bubble moves into a single move, starting from the first leg and moving our way back to the last one, and moving every leg

over at least one position if we move the first one. The conversion keeps the schedule valid and makes the new leg positions at least as far as before. We now prove the claim by induction on i.

1. Base case, $i = 0$. In this case the claim trivially holds for any leg.
2. Inductive step, $i > 0$. Let S be another strategy and note that by the inductive hypothesis every leg has been moved at least as much in the greedy schedule (which we call G) as in S until time $i - 1$. Right before the i-th time the legs are moved, G has as the bubbles as far right as possible, limited by the position of the last leg. Note that the induction hypothesis implies the bubbles for S cannot be further forward than G's. Now G moves each leg as far as possible limited by the position of the first bubble and any missing planks the bubbles are currently over. Since the position of the first bubble in S is no further than G's, it cannot move the first leg further than G does, and the same then holds for any other leg, then the claim holds for i as well.

□

With this we can conclude that, if there is a solution, the number of moves per leg to get all the legs to the end is minimized by the greedy strategy, and so the greedy strategy is optimal.

Algorithm The idea behind the algorithm is that since every leg is moved the same number of times we can trace the path of the back leg and multiply the number of moves required for the back leg by the number of legs. If there were no missing planks, this would be simple. The absence of some planks complicates the simulation of the back leg, though, because the position of the back leg after moving it is not simply the front bubble position minus the number of legs plus one.

To solve this problem of positioning the last leg correctly, observe that for any set of leg moves in between bubble moves, if we don't count positions with missing planks the last leg will move the same number of planks that the first leg does. So if we can keep track of the number of planks the first leg moves, we can move the last leg that many planks as well to place it in the correct new position. To keep track of the number of planks the first leg will move in a given leg move set, we count the number of planks in the interval starting with the position just in front of the first leg's current position and ending on the position of the front bubble. For the first iteration we will need to compute this number explicitly; in subsequent iterations we can count these planks as we move the bubbles forward.

In the subsequent code, *body_front* is the position of the front bubble, *last_leg* is the position of the last leg, and *num_planks* is the above count.

Note that it is possible for *num_planks* to be zero. In that case, there is no further movement possible. If this happens before the head reached the end, i.e., before $body\_front = n$, this means that there is no solution to the problem. Once the head has reached the end, we may still need to move the legs one last time, namely when $num\_planks > 0$.

The correctness of the algorithm follows from our proof that the greedy strategy is optimal and the fact that the algorithm correctly calculates the number of steps required for the greedy algorithm, which we explained above. To argue the running time, we note that the algorithm does a constant amount of work each time it accesses each position in the *planks* array, and it accesses each position at most twice. So the running time of the algorithm is $O(n)$.

Algorithm 2

Input: The number of positions n, bubbles k, legs ℓ, and the binary array $planks[1, \ldots, n]$ indicating the planks.
Output: The minimum number of steps required to move wormly across the bridge.

```
1:  procedure WORMLY(k, l, planks[1, ..., n])
2:      last_leg ← 1
3:      body_front ← k
4:      num_planks ← the number of 1's in planks[ℓ + 1...k]
5:      steps ← 0
6:      while body_front < n and num_planks > 0 do
7:          ▷ Move the last leg forward num_planks planked positions
8:          while num_planks > 0 do
9:              last_leg ← last_leg + 1
10:             if planks[last_leg] = 1 then
11:                 num_planks ← num_planks − 1
12:         steps ← steps + ℓ
13:         ▷ Move body as far forward as possible and update num_planks
14:         while body_front < last_leg + k − 1 and body_front < n do
15:             body_front ← body_front + 1
16:             if planks[body_front] = 1 then
17:                 num_planks ← num_planks + 1
18:             steps ← steps + 1
19:     if body_front < n then
20:         ▷ No more available planks
21:         return "Not possible"
22:     else
23:         if num_planks > 0 then
24:             return steps + ℓ
25:         else
26:             return steps
```


COMP SCI 577 Homework 05 Problem 3

Greed

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

25 October 2022

Counter-example

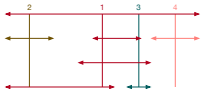


Figure 1: Wrong

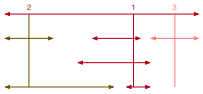


Figure 2: Correct

Let us have intervals $(0, 100], (0, 225], (0, 400], (150, 300], (180, 280], (250, 300], (300, 400]$ which are plotted and discussed below.

Wrong (Figure 1): The first pick is the one that falls inside the largest number of intervals (4), by breaking the tie for selecting the first satisfying all conditions. The remaining are following the order from left to right, with all ties containing 1 interval and the same tie-breaking rule.

Correct (Figure 2): the first pick is another one that falls inside the largest number of intervals (4), which is in a different order than the wrong tie-breaking rule. Then we have no more ties: the second pick contains 2 intervals, and the third pick contains 1 interval.

Algorithm

Explanation

We first sort the intervals, by the end (`intervals[i][1]`) and then the start (`intervals[i][0]`) for every interval. Then we loop through the sorted intervals and check whether

the last picked point covers the next intervals with the same ends (i.e., only need to check the start of the last interval). If the check fails or there is no picked point (i.e., at the beginning), we pick the common end of the next intervals. If there are no next intervals (i.e., the last interval), we just do the check again, as it is the same situation to split a set of intervals with the same ends. By picking like this, we get the smallest number of points.

Code (Python) of $O(n \log n)$

```python
import typing
import input

if __name__ == "__main__":
    input.openFile("test0.in")
    n: int = input.nextInt()
    intervals: typing.List[typing.Tuple[int, int]] = []
    last_cut: int = -1
    cuts: int = 0
    for i in range(n):
        intervals.append((input.nextInt(), input.nextInt()))
    intervals.sort(key=lambda x: (x[1], x[0]))
    for i in range(n):
        if i == n - 1 or intervals[i][1] != intervals[i + 1][1]:
            if last_cut <= intervals[i][0]:
                last_cut = intervals[i][1]
                cuts += 1
    print(cuts)
```

Proof

Induction

Measure m_k: the smallest number of points to cover the interval from 0 to the k^{th} distinct interval end.

Claim: The algorithm is correct for m_k of the first n distinct ends of intervals for all $k \in [n], n \in \mathbb{N}$.

Base Case: $n = 0$, no loop is executed, and the number of cuts m_0 is obviously 0.

Inductive Step: Suppose we have the algorithm correct for $n \leq k-1$ distinct ends of intervals, then we want to prove for $n = k$. Denote the set of intervals with the k^{th} distinct interval as SI_k. There are two cases:

1. $\forall si \in SI_k, \forall i \in [0, k-1], si.start \geq SI_i.end$**:** Namely, the cut of all previous intervals does not affect any of this interval, which is equivalent to $SI_k[0].start \geq SI_i.end$ with $SI_k[0]$ be with the one with the smallest start, or $SI_k - \cup_{i \in [0,k-1]} SI_i = SI_k$. Then all the intervals in SI_k should be handled, and we increment m_k for 1. As if we pick the end, as the range of the pick an optimal solution could make must include the end (at least one common point).

2. does not satisfy 1: Namely, $SI_k - \cup_{i \in [0,k-1]} SI_i \neq SI_k$.

2. (a) $SI_k - \cup_{i \in [0,k-1]} SI_i \neq \emptyset$**:** Namely, there are still intervals ending with $SI_k.end$ which does not cover by previous picks; in our algorithm, we can check this situation by only $SI_{k-1}.end$ (i.e., `last_cut`), as it is the most recent pick which should capture all elements in the set difference, as any interval which overlaps with previous intervals should have a start in previous intervals, which must pass $SI_{k-1}.end$, plus it is the inverse of the situation discussed in **1**. In this case, we must pick a number in the interval left (for greedy, the end) again by $m_k = m_{k-1} + 1$, and any intersection with the next intervals will be covered in future induction cases.

2. (b) $SI_k - \cup_{i \in [0,k-1]} SI_i = \emptyset$**:** The check for this situation is the inverse to which is already discussed in **2. (a)**. We just skip these intervals like the base case, as we are already done, given that they are already covered in the first $k-1$ numbers from the induction hypothesis. Thus, $m_k = m_{k-1}$.

As the greedy solution stays ahead of an arbitrary optimal solution in all cases, the greedy solution is optimal. □

Termination

The algorithm must terminate, as it is in loops without any recursion. Also, edge cases (i.e., beginning and ending) are taken care of.

Complexity

Data reading costs $O(n)$ time, sorting costs $O(n \log n)$ time, and cutting costs $O(n)$ time. The overall time cost is $O(n \log n)$.

Test Cases

Input: $Intervals \leftarrow (0,100], (0,225], (0,400], (150,300], (180,280], (250,300], (300,400]$;
Output: 3;
Explanation: Same as the counter-example.

Appendix

Code (Python) for utility `input`

```
from typing import List, Optional

file = None
queue = []

def openFile(filename: Optional[str]):
    global file
    if filename != None:
        file = open(filename, "r")
    else:
        file = None

def next() -> Optional[str]:
    while len(queue) == 0:
        if file == None:
            line: str = input()
        else:
            line: str = file.readline()
        if len(line) == 0:
            return None
        lineArr: List[str] = line.split(" ")
        for lineElem in lineArr:
            queue.append(lineElem)
    return queue.pop(0)

def nextInt() -> int:
```



```
    result = next()
    if result != None:
        return int(result)
    raise Exception("no input")
```