CS 577: Introduction to Algorithms Fall 2020

Homework 5 Solutions

Instructor: Dieter van Melkebeek TA: Ryan Moreno

Problem 1

You are given a knapsack with a weight limit of W and n items with non-negative 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 FillKnapsack($w_1, w_2, \dots w_n, W$)

Input: n items with non-negative weights $w_1, w_2, \dots, w_n$ 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: $K \leftarrow \varnothing$
2: $w \leftarrow 0$
3: $i \leftarrow n$
4: while $i > 0$ do
5:     if $w + w_i \leq W$ then
6:         $K \leftarrow K \cup \{i\}$
7:         $w \leftarrow w + w_i$
8:     $i \leftarrow i - 1$
9: 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 number 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 inductive 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.

Problem 3

You are going to compete in a puzzle competition. There will be n different puzzles released throughout the competition; the i-th puzzle is released at time r_i. Once a puzzle has been released, you can access it whenever you choose to, but you must complete the puzzles in order. You will have t minutes to complete a puzzle once you begin and you cannot move on to the next puzzle (or nap) until those t minutes are complete. Because the puzzles are mentally taxing, you can only do up to k puzzles in a row before requiring an m minute nap. You have just received the schedule of events, a list of puzzle titles ordered by the time they will be released. You are not only awarded points based on your solutions, but also based on how fast you finish the competition. So, you need to design an algorithm that computes your schedule for napping and problem solving such that you finish the competition at the earliest time possible. Your algorithm should take $O(n)$ time.

Let f_i^* denote the earliest time you can finish puzzle i for $i \in [n]$. We are interested in a schedule that realizes f_n^*.

Intuition and algorithm We begin by developing intuition for our greedy algorithm. First let's consider what happens when $n \leq k$. In this case we never need to nap, so there are no real decisions to be made. We start puzzle 1 at its release time r_1. Adding the time t you get to solve it yields $f_1^* = r_1 + t$. We start each puzzle $i \geq 2$ as soon as (1) puzzle i has been released and (2) we finished puzzle $i-1$, whichever comes later. Thus, $f_i^* = \max(r_i, f_{i-1}^*) + t$. For completeness and future reference, we provide pseudocode for computing f_n^* as well as the above optimal schedule in the case where $n \leq k$.

```
Algorithm 2 SimpleSchedule($r_1, r_2, \ldots, r_n; t$)
Input: $r_1, r_2, \ldots, r_n \in \mathbb{R}$; $t \in [0, \infty)$
Output: $f_n^*$ and a schedule realizing $f_n^*$ assuming $n \leq k$
1: $S \leftarrow$ empty schedule
2: $s \leftarrow -\infty$
3: for $i = 1$ to $n$ do
4:     $s \leftarrow \max(s + t, r_i)$
5:     Append "Start puzzle $i$ at time $s$" to $S$
6: return $(s + t, S)$
```

For $n > k$ puzzles, we have to nap at least once. When we nap, we'll do so immediately after a puzzle (there's no benefit to just sitting around between the puzzle and napping). Once we've decided which puzzles we will nap after, an optimal schedule based on those choices consists of applications of SimpleSchedule on the sequence of puzzles before the first nap, the sequence in between naps, and the sequence after the last nap. We formally state the optimality of using SimpleSchedule as follows:

Claim 2. *Consider a sequence of at most k puzzles which the greedy algorithm G has scheduled with no naps in between puzzles. If G's starting time for the first puzzle in this sequence is minimized, then we have $f_y = f_y^*$, where f_y is the finish time in G of the last puzzle in this sequence, puzzle y.*

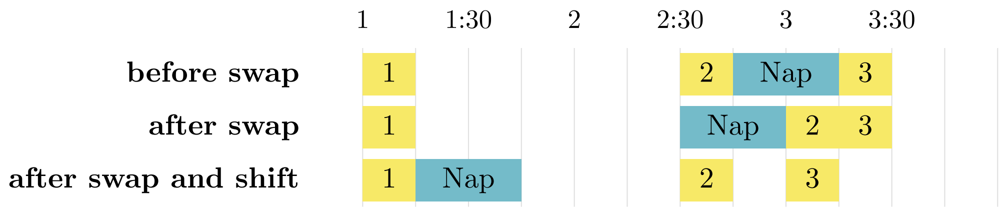
Now we need to choose which puzzles we will nap after. The crucial observation regarding the choice of nap times is the following: Suppose that the last time we nap before puzzle n is right after puzzle p. Say the schedule contains:

"Start puzzle p at time s. Take a nap starting at time $s+t$."

As long as $p > n-k$, we can swap puzzle p and the nap, i.e., we can modify that part of the schedule as follows without violating its validity and without affecting the start/finish time of puzzle n:

"Take a nap starting at time s. Start puzzle p at time $s+m$."

Moreover, once that swap is made, we may be able to shift the time s to earlier in the day, and thereby open up the possibility of shifting the rest of the schedule and puzzle n to earlier in the day. This is the case for the example from the assignment, for which the steps are illustrated in the figure below with $p = 2$.



As the sequence of puzzles from $n-k$ to n contains $k+1 > k$ puzzles, it has to be interrupted by at least one nap. Based on our crucial observation above, we will schedule this nap immediately following puzzle $n-k$. This leads us to the following algorithm:

- Minimize the start/finish time of puzzle $n-k$.
- Schedule a nap right after finishing puzzle $n-k$.
- Use SimpleSchedule to schedule puzzles $n-k+1$ to n right after this nap.
- Repeat this process considering only the puzzles up to $n-k$.

Repeated application of this strategy ultimately leads us to minimizing the start/finish time of the puzzle ℓ where $1 \le \ell \le n$ and $\ell \equiv n \bmod k$. The following pseudocode describes the complete algorithm.

Algorithm 3

```
1: $\ell \leftarrow n \bmod k$
2: **if** $\ell = 0$ **then** $\ell \leftarrow k$
3: $(f, S) \leftarrow$ SimpleSchedule$(r_1, r_2, \ldots, r_\ell; t)$
4: **for** $i = 1$ **to** $\lceil n/k \rceil - 1$ **do**
5:     **if** $i > 1$ **then**
6:         Append "Take a nap starting at time $f$" to $S$
7:         $f \leftarrow f + m$
8:     $(f, S') \leftarrow$ SimpleSchedule$(\max(f, r_{\ell+1}), r_{\ell+2}, \ldots, r_{\ell+k}; t)$
9:     Append $S'$ to $S$
10:    $\ell \leftarrow \ell + k$
11: **return** $S$
```

Correctness First we give a proof sketch of Claim 2, which can be formalized using a greedy-stays-ahead argument with the finish time of the latest puzzle as the quality measure. However, it is straightforward enough to say the following.

Proof of Claim 2. Since there are $\leq k$ puzzles, we do not need to nap at any point in this sequence. Since the starting time of the first puzzle is minimized and G schedules us to begin each subsequent puzzle as soon as it's available and we're available (and we are not allowed to skip puzzles or take less than t minutes on each), there is no way to have begun the last puzzle in the sequence any earlier, so there is no way to have finished it earlier. □

This problem lends itself to either a greedy stays ahead argument or an exchange argument. We will focus on the greedy stays ahead proof, but a brief explanation of the exchange argument is given at the end. We want to make a claim about f_n^* and we know that we can use Claim 2 to say something about the last puzzle in a sequence of uninterrupted puzzles. (Note: when we say uninterrupted puzzles we mean that there are no naps scheduled between any of the puzzles. There might be breaks due to the release times.) Our greedy-stays-ahead claim is the following:

Claim 3. *For a sequence of uninterrupted puzzles in G, the last puzzle in that group is completed no later than in any other schedule. In other words, if puzzle j is the last puzzle before a nap and f_j is the finish time of puzzle j in G, $f_j = f_j^*$.*

Proof. By induction on the number of groups.

Base Case: For the first group of uninterrupted puzzles, this follows immediately from Claim 2 since the starting time of puzzle 1 is minimized (there's nothing to wait for).

Inductive Step: Now assume that Claim 3 holds for the $(i-1)^{th}$ group of uninterrupted puzzles, and consider the i^{th} group. Denote f_j as the finish time of j in G.

Let w be the last puzzle in the i^{th} group, and v be the last puzzle in the $(i-1)^{th}$ group. If $f_w = t + r_w$, then our claim holds because there is no way to have begun puzzle w before its release. Otherwise, beginning puzzle w was delayed because we had to wait for a nap to finish or for the previous puzzle to finish. Since our greedy strategy completes k puzzles in each uninterrupted group after the first group, there are $k-1$ puzzles between puzzle v and w. Thus, any schedule S must nap between these puzzles. Additionally, by our inductive hypothesis, puzzle v cannot have been completed any earlier, i.e., $f_v = f_v^*$. So any valid schedule must have a nap scheduled before puzzle w that begins at least as late as f_v.

Note that between finishing puzzle v and beginning puzzle w, G immediately naps for t minutes and then completes $k-1$ puzzles as soon as the puzzles are available and we are available. Consider puzzle $v+1$, the first puzzle in the i^{th} group of uninterrupted puzzles. Note that it's possible that puzzle $v+1$ is released during our nap and some other solution S actually finishes it before we do. However, this is not a contradiction with our claim since puzzle $v+1$ is not the final puzzle in an uninterrupted sequence of puzzles in G.

Now consider the puzzles $v+1$ to w. Either the puzzles have been released so close together that we never complete a puzzle before the next is released or we're available at the time of release of some puzzle x. In the second case, Claim 2 implies $f_w = f_w^*$ because the sequence of puzzles x to w is uninterrupted in G and the start time of x is r_x which is optimal. In the first case, there is no downtime between puzzles or between our nap and puzzle $v+1$, so we spend exactly $m + t(k-1)$ minutes between f_v and beginning puzzle w. No schedule can do better than this since we said that any other schedule S must have a nap scheduled between puzzles v and w (and so must take at least $m + t(k-1)$ minutes between finishing puzzle v and starting w). Hence, $f_w = f^*_w$. $\square$

The optimality of the algorithm follows from Claim 3. It implies that the last puzzle cannot have been completed earlier, so we cannot have finished the competition earlier.

As exchange arguments were not the focus of this week, we will merely give the main idea and you can work out the formal proof as practice. In an exchange argument, we would consider some optimal solution S, and we would shift the last nap before puzzle n to right after puzzle $n-k$ (without increasing the time we finish working on puzzle n). We then repeat this process considering only the puzzles up to $n-k$.

Runtime Analysis Algorithm SimpleSchedule performs a constant number of operations per puzzle, and therefore runs in time $O(n)$. It follows that the final algorithm spends $O(k)$ time for each consecutive block of k puzzles starting from puzzle n, and another $O(k)$ time for the remaining puzzles. Thus, the resulting algorithm runs in time $O(n)$.

Problem 4

Given n half-closed intervals $(s_1, f_1], (s_2, f_2], \ldots (s_n, f_n]$ representing lecture start and finish times, we are looking for the smallest number of visits such that every single interval is covered by at least one visit.

(a) You decide that you should pick your first visit to be a time that overlaps with the maximum number of lectures, that is a number t that is contained in the maximum number of intervals $(s_i, f_i]$. Then you remove these intervals and pick your second visit with the same rule. You continue until no more lectures are uncovered. Construct a counter-example where this greedy strategy fails to compute an optimal solution.

(b) Design a greedy algorithm that computes the smallest number of visits to the university that cover all lectures. Your algorithm should run in time $O(n \log n)$.

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. $t_1 = 3.5$. Then it would pick the next visits $t_2 \in (4,6] \cap (4,7]$, e.g. $t_2 = 5$ and finally some $t_3 \in (0,2]$, $t_3 = 0.5$. The optimal number of visits in this example is 2, for example $t_1 = 2$ and $t_2 = 5$.

Part (b)

Intuitively, for this problem it makes sense to wait as long as possible for a visit, i.e., until we are forced to visit because we otherwise would not be able to satisfy all the requirements. This suggests the following greedy strategy.

Find the earliest finish time of an uncovered class, and schedule a visit at that time.

Correctness To prove that this greedy strategy is correct we first introduce some useful notation. Let $L = [(s_1, f_1], (s_2, f_2], \ldots, (s_n, f_n]]$ be the list of lectures (half-open intervals) sorted by increasing finish time. Let $G = (g_1, \ldots, g_k)$ be the list of visit times of the greedy solution and let $S = (t_1, \ldots, t_m)$ be another solution. We assume that both solutions are sorted in increasing order, that is $g_1 \leq g_2 \leq \ldots \leq g_k$ and $t_1 \leq t_2 \leq \ldots \leq t_m$.

For any solution S and any positive integer k, we define $t_S(k)$ as the time of the k-th visit in S (in order of non-decreasing time), or ∞ if S has less than k visits. We also define $t_S(0) = -\infty$ (this is just a technical assumption). Now we can define the notion of "ahead" for our greedy strategy, namely that for every k the k-th visit of greedy is at least as late as the k-th visit of any valid schedule S.

Claim 4. *For any valid schedule S and any positive integer k, $t_G(k) \geq t_S(k)$.*

Before we prove the claim we note that the optimality of G follows from Claim 4 as it implies that if some valid schedule S consists of less than k visits (indicated by $t_S(k) = \infty$) then so does G. Note also that G always produces a valid solution because it covers all classes (the greedy algorithm always ensures that the earliest finish time of an uncovered class gets covered).

Proof. We establish the claim by induction on k

1. Base case, $k = 0$. This trivially holds because $t_G(0) = t_S(0) = -\infty$.

2. Induction step, $k > 0$. The key observation is that for any valid schedule S, the intervals that are not yet covered by the first k visits are exactly those whose start time is at or after $t_S(k)$. This is true because we consider the two schedules G and S in increasing order. If an interval starts before $t_S(k)$ and is not covered by the first k visits, this means that this interval will remain uncovered in the end. From the inductive hypotheses we have that $t_S(k-1) \leq t_G(k-1)$. Using the above observation, we have that all intervals of G [respectively S] that remain uncovered after $k-1$ visits start at or after $t_G(k-1)$ [respectively $t_S(k-1)$]. Since $t_S(k-1) \leq t_G(k-1)$ the set of uncovered intervals of G after $k-1$ visits is a subset of the uncovered intervals of S after $k-1$ visits. Remember also that G schedules a visit for the earliest finish time of an uncovered class (call this class C). Therefore, if $t_S(k) > t_G(k)$ then S does not cover C, which is a contradiction since we assumed that S is a valid solution. This means that $t_G(k) \geq t_S(k)$.

□

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 increasing order of f_i. This step takes time $O(n \log n)$ using mergesort. We can then find the optimal number of visits by doing a linear scan of the list where we keep track of the time of the latest visit, updating our latest visit with the finish time of the next interval that remains uncovered. 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 4.

Algorithm 4

Input: A list L of n tuples $(s_i, f_i]$.
Output: The smallest number of visits to the university that covers all intervals.

```
1:  procedure InviteEveryone($(s_1, f_1], (s_2, f_2], \ldots, (s_n, f_n]$)
2:      Sort $L$ in non-decreasing order of finish time
3:      $t \leftarrow -\infty$
4:      $k \leftarrow 0$
5:      $i \leftarrow 1$
6:      while $i \leq n$ do
7:          if $s_i \geq t$ then                ▷ If $(s_i, f_i]$ is not covered by a visit at time $t$
8:              $t \leftarrow f_i$                       ▷ Then visit at time $f_i$
9:              $k \leftarrow k + 1$
10:         $i \leftarrow i + 1$
11:     return $k$
```

Alternate solution This problem is actually dual to the interval scheduling problem from class: the minimum number of visits 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 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 (moving the first leg forward, then then second etc.), and then to move the body as far forward as possible. We repeat this process 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 5. *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 between any two bubble moves, each leg is moved at most once, either all legs or no legs are moved, and if they are moved then 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 (always possible because moving the first one opens up a plank). 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 the bubbles as far right as possible, limited by the position of the last leg. Note that the inductive 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 5

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($k, l, planks[1, \ldots, n]$)
2:     $last\_leg \leftarrow 1$
3:     $body\_front \leftarrow k$
4:     $num\_planks \leftarrow$ the number of 1's in $planks[\ell + 1, ..., k]$
5:     $steps \leftarrow 0$
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:             $last\_leg \leftarrow last\_leg + 1$
10:            if $planks[last\_leg] = 1$ then
11:                $num\_planks \leftarrow num\_planks - 1$
12:        $steps \leftarrow steps + \ell$
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:            $body\_front \leftarrow body\_front + 1$
16:            if $planks[body\_front] = 1$ then
17:                $num\_planks \leftarrow num\_planks + 1$
18:            $steps \leftarrow steps + 1$
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 $steps + \ell$
25:        else
26:            return $steps$
```