COSC 311: ALGORITHMS
HW3: GREEDY AND DYNAMIC PROGRAMMING
Solutions

Problem 1: Scheduling to minimize response time.
In many computer systems settings, all of the jobs that are submitted to a server are allowed to run (unlike the interval scheduling problem, in which only a subset of the jobs can run), and the goal is to ensure that all jobs complete running as quickly as possible. The *response time* problem is defined as follows:

Input: a set J consisting of n jobs, where each job j has an arrival time $a(j)$ (the earliest time at which job j is allowed to start running—job j can start at any time at or after $a(j)$) and a size $x(j)$ (the amount of time for which job j must occupy the server continuously).

Output: a schedule of start times $s(j)$ for all jobs j in J, such that:
(1) The schedule is *feasible*, meaning it satisfies the following properties:

- For all jobs j, $s(j) \geq a(j)$ (a job can only start after it has arrived).
- Let $f(j)$ be job j's finish time (the time at which it completes running). For all jobs j, we must have $f(j) = s(j)+x(j)$ (jobs cannot be interrupted; each job must run for a contiguous block of time).
- For any pair of jobs i and j, either $f(i) \leq s(j)$ or $f(j) \leq s(i)$ (jobs cannot overlap; the server can only run one job at a time).

(2) The schedule minimizes *overall response time*, T. A job j's *response time* is defined to be $t(j) = f(j) - a(j)$ (the time that passes between its arrival time and its finish time). The overall response time is $T = \sum_{j=1}^{n} t(j)$. Our goal is to find a feasible schedule that minimizes T.

(a) Assume that all jobs arrive at time 0, so $a(j) = 0$ for all j. Give a greedy algorithm that finds a schedule that minimizes the overall response time.

Solution: The optimal algorithm is Shortest Job First (SJF). This is a greedy algorithm that at all times schedules the unfinished job j with the smallest size $x(j)$. For example, if we have three jobs with $x(1) = 4$, $x(2) = 2$, and $x(3) = 7$, then we schedule the jobs in order 2, 1, 3. Job 2 finishes at time $f(2) = 2$, job 1 finishes at time $f(1) = 6$, and job 3 finishes at time $f(3) = 13$. The total response time is 21.

```
1 Schedule(J)
2     S = jobs in J, sorted by size x(j)
3     nextStart = 0
4     for each job j in S
5         set s(j) = nextStart
6         set nextStart = nextStart + x(j)
7     return S
```

Intuitively, SJF is the right thing to do because if we choose a job j to run first, *all* jobs experience the size of job j as part of their response time. Since all jobs have to incur a cost equal to $x(j)$ for whatever j we schedule first, it makes sense to choose the smallest job possible as the first job. Consider our above example. If we scheduled job 3 first, then jobs 2 and 1, we would have $f(3) = 7$, $f(2) = 9$, and $f(1) = 13$. The total time it takes to run all jobs does not change (the last job still ends at time 13), but now the total response time is 29 because jobs 2 and 1 had to wait for the much larger job 3 to finish.

(b) Here's an example input (continue to assume that $a(j) = 0$ for all jobs j):

j	1	2	3	4	5
$x(j)$	4	8	3	1	5

Show what happens when you run your algorithm on this input (i.e., what schedule does your algorithm produce, and in what order are jobs added to this schedule?)

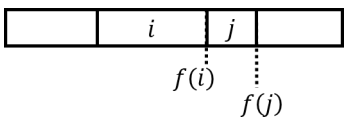
Solution: The output is: $s(4) = 0$, $s(3) = 1$, $s(1) = 4$, $s(5) = 8$, $s(2) = 13$. Jobs are added to the schedule in the order 4, 3, 1, 5, 2; that is, in increasing order of size. The overall response time in this schedule is 1 + 4 + 8 + 13 + 21 = 47.

(c) In terms of n (the number of jobs in J), what is the runtime of your algorithm?

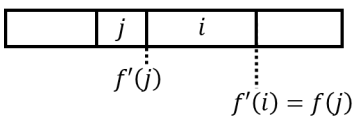
Solution: In order to schedule the jobs in SJF order, we must first sort the jobs in increasing order of $x(j)$ (line 2). This takes time $\Theta(n \lg n)$. We then make a single pass through the sorted list of jobs (lines 4-6) and assign each job a start time $s(j) = s(j-1) + x(j-1)$ (the start time of job 1 is $s(1) = 0$). This additional pass takes linear time. The overall runtime therefore is $\Theta(n \lg n)$.

(d) Explain how you know that your algorithm produces an optimal solution.

Solution: We'll prove this by contradiction. Suppose we have some schedule OPT that minimizes the total response time, but is not SJF. Then at some point, OPT schedules a job i before job j, where $x(i) > x(j)$. In particular, there must be a time when job i is scheduled *immediately* before job j. (Why? Because if there is ever a time when job a is scheduled before job b and $x(a) > x(b)$, we can imagine walking down the schedule from a to b. Since $x(b)$ is smaller than $x(a)$, there must be some first point along this walk when the job size decreases. This occurs at a pair of jobs i, j where i is scheduled immediately before j and $x(i) > x(j)$.) So we have the scenario in figure (a) below.



(a) Original schedule



(b) After swapping i and j

Consider what happens if we swap the order of jobs i and j in our schedule; now we have the

scenario in figure (b). Under OPT, the total response time for all jobs is

$$T = \sum_{k=1}^{n} f(k) = \sum_{k \neq i,j} f(k) + f(i) + f(j).$$

After making the swap, job j finishes at some earlier time $f'(j) < f(j)$. Job i finishes later than it did before, but job i's new finish time is the same as job j's old finish time: $f'(i) = f(j)$. So our new total response time is

$$T' = \sum_{k \neq i,j} f(k) + f'(j) + f(j) \leq T.$$

This contradicts our assumption that there is some optimal schedule OPT that has a pair of jobs in "inverted" order, i.e., that OPT could be something other than SJF. Hence the optimal schedule must be the schedule produced by SJF.

Other forms of reasoning are possible, including something like the intuitive argument given in part (a) (with a little more math to back it up). Another reasonable approach is to prove that (1) the greedy choice property holds (i.e., that there is some optimal solution that schedules the smallest job first) and (2) the optimal substructure property holds (i.e., that any optimal schedule involves optimally scheduling the first k jobs and optimally scheduling the remaining $n - k$ jobs, for any $k < n$).

(e) [Challenge*] Now let's relax the assumption that all jobs arrive at time 0, and say that each job j arrives at some time $a(j) \geq 0$. We only become aware of job j at its arrival time, so at all times we can only make our scheduling decisions based on our knowledge of the jobs that already have arrived. We also remove the constraint that each job must run for a contiguous block of time, so that now we are allowed to *preempt* jobs (that is, pause a job, run a different job or jobs for a while, and then restart the job that was paused). Does your algorithm from part (a) still give the optimal solution? If so, explain why. If not, give a counterexample and suggest an algorithm that might perform better.
***Note about challenge problem:** This part of the problem will be worth 1 point out of 20. This is meant to give you an incentive to think about it if you have time, but still be a small enough value that you can skip it without significant penalty if you don't.

Solution: SJF is no longer optimal when our jobs have positive arrival times and preemption is allowed. Here is a simple counterexample: suppose job 1 arrives at time $a(1) = 0$ and has size $x(1) = 5$, and job 2 arrives at time $a(2) = 4$ and has size $x(2) = 2$. Under SJF, we run job 1 from time 0 until time 4. At time 4, job 2 arrives and since $x(2) < x(1)$, SJF says that we should preempt job 1 and run job 2 instead. At time 6, job 2 finishes and we resume job 1, which finishes at time 7. Job 1 has response time $t(1) = f(1) - a(1) = 7 - 0 = 7$ and job 2 has response time $t(2) = f(2) - a(2) = 6 - 4 = 2$; the total response time is $T = 9$. A better schedule would be to run job 1 from time 0 until time 5, when it finishes, and then run job 2 from time 5 to time 7. Under this schedule, job 1 has response time $t'(1) = f'(1) - a(1) = 5 - 0 = 5$ and job 2 has response time $t'(2) = f'(2) - a(2) = 7 - 4 = 3$; the total response time is $T' = 8$.

Intuitively, the problem with SJF is that when job 2 arrives, a lot of job 1 has already completed running, so our intuition that it's best to make everyone wait for the job with smallest size no longer makes sense. If we make job 2 wait for job 1 to finish running, job 2 only has to wait time 1, not the entire $x(1) = 5$. This suggests that a more relevant factor is the *remaining time* of a job, rather than its original size. Indeed, the optimal policy in the setting with positive arrival times and preemption is to at all times schedule the job with Shortest Remaining Processing Time (SRPT). Under SPRT, every time a job j arrives to the system we compare $x(j)$ to the remaining time of the job currently in service. If the job currently in service has a remaining time that is less than $x(j)$, we continue running that job. If $x(j)$ is smaller than the current job's remaining time, we preempt the current job and run j instead. When a job finishes, we start running the job with the smallest *remaining* time. The intuition for why this works is the same as the intuition behind SJF. At all times we want to schedule the job that causes all of the other jobs to have to wait the least amount of time. In this setting, the amount of time the other jobs will have to wait is the remaining time of the job that we schedule.

2) Filling knapsacks.
Imagine you are a thief and you are trying to steal some items. Unfortunately you can't take them all because you have a knapsack that can only hold up to W pounds. There are n total items that you're interested in, and each item i has some positive integer weight w_i, in pounds. Additionally, each item has a positive integer value v_i, and you would like to steal as much value as you can, while remaining within the weight constraint of your knapsack. You are allowed to take a fractional amount p_i of an item, where p_i is between 0 and 1 (0 meaning you take none of the item, 1 meaning you take all of the item, 0.5 meaning that you take half of the item, etc.; think of the items as being something like gold powder, rather than gold ingots). The *fractional knapsack* problem is defined as follows:

Input: a set of n items, where each item i has a weight w_i and a value v_i, and a total capacity W that is the maximum weight your knapsack can hold.

Output: a fraction p_i, $0 \leq p_i \leq 1$, for each item i that indicates what fraction of the item you are stealing, such that:
(1) The total weight of the stolen items fits within the capacity of your backpack: $\sum_{i=1}^{n} p_i w_i \leq W$.
(2) The total value of the stolen items in your backpack, $V = \sum_{i=1}^{n} p_i v_i$, is maximized.

(a) Give a greedy algorithm to solve the fractional knapsack problem optimally (i.e., your algorithm should result in the most value possible in the knapsack, while staying within the weight constraint).

Solution: Our goal is to fit as much value as possible in our backpack, where we're limited by the amount of weight we can hold. Intuitively, we'll be able to do the best possible by taking as much *weight per pound* as we can. So instead of considering the absolute weight of each item, or the absolute value, we will choose which items to take and in what order based on v_i/w_i, the weight per pound of the item.

```
1 FractionalKnapsack(I,W) // I: set of items, W: weight capacity
2     for each item i in I
3         set c_i = v_i/w_i
4         set p_i = 0
5     while W > 0
6         i = item in I with maximum c_i
7         if w_i <= W, set p_i = 1
8         else set p_i = W/W_i
9         set W = W - w_i * p_i
10    return array of p_i's
```

(b) Here's a set of items:

i	1	2	3	4
w_i	15	6	9	2
v_i	15	24	27	7

Show what happens when you run your algorithm on this input with $W = 10$ (i.e., what fraction of each item does your algorithm tell you to steal, and in what order are the fractions of items determined?)

Solution: We first compute c_i for each item i: $c_1 = 1$, $c_2 = 4$, $c_3 = 3$, $c_4 = 3.5$. We initialize each p_i to be 0.

We first consider item 2, which has the highest c_i. Since $w_2 = 6 < 10$, we set $p_2 = 1$. We update $W = 10 - 1 \cdot w_2 = 4$. We next consider item 4, which has the next highest c_i. Since $w_4 = 2 < W = 4$, we set $p_4 = 1$. We update $W = 4 - 1 \cdot w_4 = 2$. We next consider item 3, which has the next highest c_i. We have $w_3 = 9 > 2 = W$, so we cannot take all of item 3. Instead we take W weight of item 3, so we want p_i such that $p_i w_i = W$. That is: $p_i = W/w_i$, and we set $p_3 = 2/9$.

Our solution is: $p_1 = 0$, $p_2 = 1$, $p_3 = 2/9$, $p_4 = 1$, and the value of this solution is $0 \cdot 15 + 1 \cdot 24 + 2/9 \cdot 27 + 1 \cdot 7 = 31$.

(c) In terms of n (the number of items), what is the runtime of your algorithm?

In lines 2-4 we make a single $\Theta(n)$ pass through the items. In lines 5-9, we could consider all n items again if they all fit in the knapsack. The key to understanding the runtime is in making a choice about how we are going to find the item with maximum c_i. One option is to sort the items by c_i before the loop in line 5; we can do this in $\Theta(n \lg n)$ time. Another option is to store the items in a heap arranged by c_i; we can heapify the elements in $\Theta(n)$ time before line 5, but then would need to take $O(\lg n)$ time to extract the maximum element on line 6 each time through the loop. Either solution requires $\Theta(n \lg n)$ time to find the maximum element on every iteration. This will dominate our runtime, so we end up with $\Theta(n \lg n)$ time overall.

(d) Explain how you know that your algorithm produces an optimal solution.

Solution: We will show that (1) the greedy choice property holds, and (2) the optimal substructure property holds.

To show that the greedy choice property holds, we need to argue that there exists some optimal solution that contains the greedy choice; that is, that takes as much as possible of the item with highest value per pound. Let i be the item with the highest value per pound. Suppose we have an optimal solution. There are two cases: either $W \geq w_i$ or $W < w_i$. If $W \geq w_i$, then the greedy algorithm sets $p_i = 1$. Suppose our optimal solution does not, and instead sets $p_i = p < 1$. Then we can take $(1-p)w_i$ units of weight out of the optimal solution (that were allocated to some item other than i) and replace it with $(1-p)w_i$ units of weight of item i. Since c_i is the highest value per pound among all items, we cannot have decreased the value of the $(1-p)w_i$ units of weight that we replaced. Hence we now have a better (or equivalently valued) solution that includes all w_i pounds of item i. If instead $W < w_i$, then the greedy algorithm sets $p_i = W/w_i$; that is, all W pounds of the knapsack are allocated to item i. Suppose our optimal solution sets $p_i = p < W/w_i$. Then again, we can swap out $(1-p)w_i$ units of weight that were allocated to some other item and include $(1-p)w_i$ pounds of item i instead. Since c_i is the highest value per pound among all items, we cannot have decreased the value of the weight we replaced. So we now have a better (or equivalently valued) solution that includes only item i. Hence there is an optimal solution that includes the greedily selected amount of item i.

To show that the optimal substructure property holds, we need to argue that any optimal solution contains within it optimal solutions to subproblems. Suppose we are given an arbitrary optimal solution to the knapsack problem. This solution specifies a fraction of each item to take, p_i for each item i. Now imagine splitting up the optimal solution as follows: piece 1 will consist of p_i fraction of item i for $1 \leq i \leq k$ for some $k < n$, and piece 2 will consist of everything else. Splitting up the optimal solution in this way imposes two subproblems. In subproblem 1, we need to fill a knapsack with capacity $W_1 = \sum_{i=1}^{k} p_i w_i$, where the items available to us are $p_i w_i$ weight worth of item i, for $1 \leq i \leq k$. In subproblem 2, we need to fill a knapsack with capacity $W_2 = W - W_1$, where the items available to us are $(1-p_i)w_i$ weight worth of item i for $1 \leq i \leq k$, and w_i weight worth of item i for $k < i \leq n$.

In subproblem 1, our knapsack can exactly fit the items we have available to us, so the best solution is to take everything available. In subproblem 2, our original optimal solution solved the problem by taking p_i fraction of item i for $k < i \leq n$. Suppose that there's some better (i.e., higher value) way to fill the knapsack with the items available to us in subproblem 2. Then we could replace our original solution to subproblem 2 with this better solution. That would give us a higher value solution to our original problem, which cannot happen because we started with an optimal solution to the original problem. Hence there cannot be a better solution to subproblem 2; our original optimal solution must contain inside of it optimal solutions to the subproblems.

Putting together the greedy choice property and the optimal substructure property, we find that our greedy algorithm must be optimal.

3) Rod cutting.
Imagine that you have decided to open a small business, and your grand entrepreneurial idea is to take steel rods of length n, cut up the rods into smaller pieces, and sell the pieces. You know something about the market for steel rods, so you know that you can sell a piece of length i for price p_i, where $p_i \leq p_j$ if $i < j$ (a longer piece is worth at least as much as a shorter piece). Your goal is to figure out how best to cut up the length-n rods in order to make the highest profit. The *rod cutting* problem is defined as follows:

Input: a length n, and, for all (positive, integer) $i \leq n$, a price p_i for which you can sell a rod of length i.

Output: a "decomposition" of the length-n rod into shorter pieces so that the total price of all of the shorter pieces is maximized.

(a) There is no greedy algorithm for this problem that is guaranteed to be optimal on all inputs. Give a dynamic programming algorithm that is guaranteed to solve the rod cutting problem optimally for all inputs.

Solution: We can think about the rod cutting problem as follows. We will, at some point, have to cut off a first piece of the rod. Once we've done that, we are left with the task of cutting up the rest of the rod. The problem is that we don't know how to choose the length of the first piece to cut off—no greedy strategy is guaranteed to always work. So instead, our algorithm will need to consider all possible ways of cutting off the first piece, and choose the best option.

Version 1 (iterative, bottom-up):

```
1 BottomUpRodCut(n, P) // n: length of the rod,
                       // P: array of prices of each piece size
2     r: a new array of length n+1
3     set r[0] = 0 // a length-0 piece is worthless
4     for i = 1 to n
5         set q = -infty
          // consider all possible ways of cutting a first piece of
          // size j and then optimally cutting up the rest of the rod
6         for j = 1 to i
7             set q = max{q, P[j] + r[i-j]}
8         r[i] = q // we found the best way to cut up a length-i rod
9     return r[n]
```

In Version 1 we work from smaller pieces to bigger pieces, building up as we go the best way to cut up longer and longer rods. At each step, once we have found the optimal solution for a length-i rod, we record it in the array `r` so that later on when we need to know how best to cut up a rod of length i, we can simply look up the answer in our array.

Version 2 (recursive, memoized):

```
1 MemoizedRodCut(n, P) // n: length of the rod,
                       // P: array of prices of each piece size
2     r: a new array of length n+1
3     initialize all elements in r to -infty
4     return MemoizedRodCutHelper(n, P, r)

5 MemoizedRodCutHelper(n, P, r)
6     if r[n] >= 0, return r[n] // if we've already solved this
                                // size subproblem, return the answer
7     if n == 0, set q = 0
8     else
9         set q = -infty
10        for j = 1 to n
11            set q = max{q, P[j] + MemoizedRodCutHelper(n-j, P, r)}
12    r[n] = q
13    return q
```

In Version 2 we solve the problem recursively, calling our `MemoizedRodCutHelper` every time we need to solve a smaller subproblem. The key here is that we *memoize* our solutions to smaller problems as we find them: we maintain (and pass around as a parameter) an array storing the optimal solutions of problems of size 0 to n. This way, whenever we need to re-solve a subproblem that we've already solved in the past, we can simply look up the answer instead of redoing all the work.

Regardless of which approach (bottom-up or recursive) we use, after filling the array of optimal solutions to subproblems we will need to retrace our steps to figure out what size pieces to cut in order to obtain the maximum profit. We do this as follows, making an initial call to `ReconstructSolution(r, n)`:

```
1 ReconstructSolution(r, P, i) // r: array of solutions to subproblems
                               // P: array of prices of each piece size
                               // i: size of rod we're cutting up
2     for j = 1 to i
3         set val = P[j] + r[i-j]
4         if val == r[i]
5             return {j} + ReconstructSolution(r, P, i-j)
```

Here we start at the end, with the best value we got from cutting up a rod of length n. We consider all possible ways we could have arrived at that value: cutting off a piece of length 1 and optimally cutting up the remaining piece of length $n-1$, cutting off a piece of length 2 ad optimally cutting up the remaining piece of length $n-2$, etc. We compute the value of each of these options by looking back into our array. One of these options must match the value in position n, because these were exactly the options we considered when filling in position n. When we find the option that matches, we know how to cut off the first piece and what subproblem we were left with. We then

recursively reconstruct the optimal way of cutting up that remaining piece.

An alternative approach to reconstructing the optimal set of pieces to cut involves being slightly more clever about what information we're storing as we compute the optimal value. In particular, we could imagine storing both the best value possible when cutting up a rod of size i, and the length of the first piece we should cut off in order to obtain this value. We could update our bottom-up solution to do this as follows:

```
1 BottomUpRodCutV2(n, P) // n: length of the rod,
                         // P: array of prices of each piece size
2     r: a new array of length n+1
3     set r[0] = 0 // a length-0 piece is worthless
4     for i = 1 to n
5         set q = -infty
6         set first_piece = 0
          // consider all possible ways of cutting a first piece of
          // size j and then optimally cutting up the rest of the rod
7         for j = 1 to i
8             if P[j] + r[i-j] > q
9                 set q = P[j] + r[i-j]
10                set first_piece = j
11        r[i] = (q,first_piece)
12    return r[n]
```

And now to reconstruct our optimal set of pieces, we do:

```
1 ReconstructSolutionV2(r, i) // r: array of solutions to subproblems
                              // i: size of rod we're cutting up
2     if i=0, return {}
3     return {r[i].first_piece} + ReconstructSolutionV2(r, i-first_piece)
```

(b) Here's an example for a rod of length 5:

i	1	2	3	4	5
p_i	\$2	\$2	\$12	\$15	\$16

Show what happens when you run your algorithm on this input (i.e., what solution does your algorithm find, and how does it go about finding this solution?)

Solution: We'll run through the bottom-up version where we're tracking the best piece to cut off as we go. We start by determining how to optimally cut a rod of length 1. There's only one possibility (we can't cut it into any smaller pieces), so we record `r[1] = (2,1)` (that is, we can get value 2, and the first piece we cut off has length 1).

Next we determine the best way to cut up a rod of length 2. There are two options: (1) cut a piece of size 1, then optimally solve the remaining (length-1) subproblem, or (2) cut a piece of size 2,

then optimally solve the remaining (length-0) problem. In the first option, we get \$2 for the size-1 piece we cut off, then \$2 for the optimal solution to the subproblem (which we look up in `r[1]`). In the second option, we just get \$2 for the size-2 piece (and \$0 for the length-0 subproblem). Cutting off a length-1 piece and optimally solving the length-1 subproblem is better, so we record `r[2] = (4,1)`.

We now move to the problem of size 3. There are three options: (1) cut a piece of size 1, then optimally solve the remaining (length-2) subproblem; (2) cut a piece of size 2, then optimally solve the remaining (length-1) subproblem; or (3) cut a piece of size 3, then optimally solve the remaining (length-0) subproblem. We find that option (1) has value \$2 + \$4 = \$6, option (2) has value \$2 + \$2 = \$4, and option (3) has value \$12 + \$0 = \$12, so we record `r[3] = (12,3)`. Continuing in this manner, we find that `r[4] = (15,4)` and `r[5] = (17,4)`.

We now go to reconstruct our solution. We see that `r[5] = (17,4)`, which means that we can get value 17 by first cutting off a piece of length 4. So we add 4 to our solution, and then recursively reconstruct the best solution for a rod of length $5 - 4 = 1$. Here we see that `r[1] = (2,1)`, meaning that we can get value 2 by first cutting off a piece of length 1. So we add 1 to our solution, and then recursively reconstruct the best solution for a rod of length $1 - 1 = 0$. Here we hit the base case, so we're done, and our solution is to cut pieces of length 4 and length 1.

(c) In terms of n (the length of the original rod), what is the runtime of your algorithm?

Solution: We'll look at the bottom-up version. We start in lines 2-3 by creating a length-n array and setting the first element to 0; this takes constant time. We then have in line 4 and in line 6 two nested for loops. Inside the inner loop we do constant work (a few array accesses, an addition, a comparison, and a variable assignment), and the inner loop runs $O(n)$ times. We do constant additional work inside the outer loop but outside the inner loop, and the outer loop runs $O(n)$ times, so in total lines 5-8 take time $O(n^2)$. Line 9 is a constant-time return statement. So the overall algorithm takes time $O(n^2)$. The reconstruction takes linear time; we simply walk backwards through the array following our "parent pointers."

(d) Explain how you know that your algorithm produces an optimal solution.

Solution: In effect, our algorithm checks all possible ways of cutting up the rod, so it must find the best one. In slightly more detail: we can first argue that the optimal substructure property holds. Suppose we have an optimal solution to our problem. We then split up the optimal solution into two pieces, with some of our partial rods in one piece and some of our partial rods in the other, and imagine joining the rods in piece 1 back together into one longer rod. Suppose we could find a better way (i.e., a higher-priced way) of cutting up the piece 1 rod than the way we originally cut it up. Then we replace piece 1 in our original solution with the better way of cutting up piece 1, and we'd have a higher-priced way of cutting up the entire rod. But this isn't possible since our original solution was optimal. So we must have started with the optimal way of cutting up piece 1.

Unfortunately the greedy choice property does not hold here for any way of defining a greedy choice, so we can't use this as a way of determining what initial choice to make and what remaining

subproblem to solve. What we do know is that we must make *some* first choice, and any first choice we can make will leave us with a subproblem. At every step of the way, our algorithm considers all possible first choices combined with the optimal way of solving the subproblem, and chooses the best first choice + optimal subproblem solution. By solving smaller subproblems first, we know that we've already found the optimal solution to a subproblem when we need that optimal solution to consider as we're solving a larger problem.