COSC 311: ALGORITHMS
HW3: GREEDY AND DYNAMIC PROGRAMMING
Solutions

1 Problems

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.

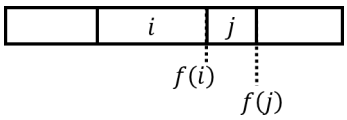
Solution: We know that $T = \sum_{j=1}^{n} t(j) = \sum_{j=1}^{n} (f(j) - a(j)) = \sum_{j=1}^{n} f(j)$. Let the index of a job denote its position in the final scheduled order, so that job 1 is the job that runs first, job 2 runs second, etc. Job j finishes after it waits for all jobs that run before it, i.e., jobs 1 through $j-1$, plus its own size, and so job j's response time is $\sum_{i=1}^{j} x(i)$ (recall that $x(i)$ denotes the size of job

i). So the total response time is:

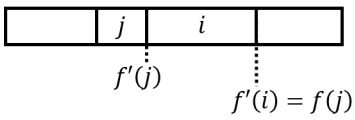
$$\begin{aligned} T &= \sum_{j=1}^{n} f(j) \\ &= \sum_{j=1}^{n} \sum_{i=1}^{j} x(i) \\ &= \sum_{j=1}^{n} (n - j + 1)x(j). \end{aligned}$$

That is, $x(1)$ gets added to the sum n times, $x(2)$ gets added $n - 1$ times, and so on, until $x(n)$ is added only once. This function will be minimized if the jobs are sorted in increasing order of size, so that $x(1)$ is the smallest and $x(n)$ is the largest.

Alternative 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) 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 efficient greedy algorithm for this problem that is guaranteed to be optimal on all inputs. Give a recursive solution to this problem. Your recursive function price(n) should determine the best sale price for a rod of length n, which may be cut up in any possible way.

Solution: We start by observing that we can phrase this problem as "cut off a piece of the rod, and then figure out how to cut up the rest of the rod." That is, we make an initial choice (what size piece to cut off initially?) and then solve the remaining subproblem (how to cut up the rest of the rod?) The optimal solution for a rod of length n must involve cutting off a first piece of *some* length from 1 to n, so we'll try them all. Our recursive algorithm is as follows:

```
1 price(n)
2     if n == 0, return 0
3     bestValue = infty
4     for i = 1 to n
5          val_i = P[i] + price(n-i)
6          if val_i < bestValue
7              bestValue = val_i
8     return bestValue
```

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

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

(i) On this example, trace enough of your recursive algorithm to show that there are repeated subproblems.
(ii) Estimate the runtime of your recursive algorithm in terms of n, the length of the original rod.

Solution: (i) We start by calling price(5). On line 5 (inside the for loop), we'll make recursive calls to price(4), price(3), price(2), price(1), and price(0). The first of these to be executed is the call to price(4). In this call, again on line 5, we make recursive calls to price(3), price(2), price (1), and price(0). Already we can see that there are overlapping subproblems: we call price(3), price(2), price(1), and price(0) from inside our price(5) call, and (separately) from inside our price(4) call.

(ii) This ends up taking time $O(2^n)$. That is exponential, and very slow.

(c) Now, design a bottom-up dynamic programming algorithm based on your recursive solution.
(i) Your algorithm should fill in a table. Describe that table: how many columns, how many rows, what is in the cells?
(ii) Describe your algorithm.
(iii) Show what happens when you run your algorithm on the input above (i.e., what solution does your algorithm find, and how does it go about finding this solution?)

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.

(i) Our table will have one row and n columns, one for each rod length. Entry i in the table will hold the answer to the question "what is the highest profit I can make from cutting up a rod of length i?"

(ii) Here's our algorithm:

```
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]
```

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.

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 and 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.
(iii) 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`.

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`.

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`.
Continuing in this manner, we find that `r[4] = (15,4)` and `r[5] = 17`.

We now go to reconstruct our solution. We see that `r[5] = 17`; working backwards, this was the result of either (1) a piece of size 1 and the optimal solution to the remaining length-4 subproblem, (2) a piece of size 2 and the optimal solution to the remaining length-3 subproblem, etc. We find that the value we stored, 17, matches the value we get from a piece of length 4 (\$15) plus the optimal solution for the length-1 subproblem (\$2, stored in `r[1]`). 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`, which, again working backwards, we see is the value obtained from cutting off an initial piece of length 1 (value \$2) plus the optimal solution to the remaining length-0 subproblem (\$0, stored in `r[0]`). 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.

Alternatively, we could have written our algorithm to store "parent pointers" as we go, and then reconstruct the solution by following the parent pointers backwards.

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

Solution: 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 is similar.

(e) 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.

3) 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. The items cannot be broken

up; you must either take the entire item or leave it. The *0-1 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 choice, $c_i = 0$ or 1 for each item i that indicates whether or not you are stealing the item (0 indicates that you are not stealing it, 1 indicates that you are), such that:
(1) The total weight of the stolen items fits within the capacity of your backpack: $\sum_{i=1}^{n} c_i w_i \leq W$.
(2) The total value of the stolen items in your backpack, $V = \sum_{i=1}^{n} c_i v_i$, is maximized.

(a) Give a dynamic programming algorithm to solve the 0-1 knapsack problem optimally (i.e., your algorithm should result in the most value possible in the knapsack, while staying within the weight constraint). You may want to follow the same approach as in problem 2, i.e., begin by coming up with a recursive solution, then translate that into a bottom-up DP algorithm.

Solution: We'll begin by rephrasing our problem as follows: we want to find the maximum value we can fit in up to W units of weight, choosing from among the first n items.

For each item, we get to choose whether or not to include it in our solution. We'll start by considering item n. If we do not include item n, then the best possible solution is the same as the best possible solution for weight W, choosing from among the first $n-1$ items. If we do include item n, then the best possible solution includes item n, plus the best solution for weight $W - w_n$ and choosing from among the first $n-1$ items. This suggests the following recurrence, where $V(w, i)$ denotes the maximum value we can steal for a backpack with weight capacity w, considering the first i items:

$$V(W, n) = \max\{V(W, n-1), v_n + V(W - w_n, n-1)\}.$$

Because our recurrence has two parameters, we will want our algorithm to fill in a 2-dimensional table. The rows represent the possible remaining weight capacities of our backpack (ranging from 0 to W), and the columns represent the set of items we're including (where column i means that we're considering items $1 \ldots i$). There are $W+1$ rows and $n+1$ columns. In order to fill the entry in row w and column i, our recurrence says that we need to have already recorded the solutions for fewer than i items (the columns to the left) and for less than w weight (the rows above). So we'll fill in the table from left to right and from top to bottom.

Our algorithm is as follows:

```
1 Knapsack(W,n)
2     V[0..W][0..n]: a new array
3     V[0][i] = 0 for all columns i
4     V[w][0] = 0 for all rows w
5     for i = 1 to n
6         for w = 1 to W
7             skip = V[w][i-1]
8             include = v_n + V[W-w_n][i-1]
```

```
9               if skip < include
10                  V[w][i] = skip
11              else V[w][i] = include
12      return V[W][n]
```

After filling in the table, we need to reconstruct our solution to determine which items to steal (the table only tells us how much value we get). We can do this as follows, making an initial call to `Reconstruct(V, W, n)`:

```
1 Reconstruct(V, w, i)
2       if w == 0 or i == 0
3           return emptyset
4           if V[w][i] == V[w][i-1]
5           return Reconstruct(V, w, i-1)
6       else return {i} + Reconstruct(V, w-w_i, i-1)
```

(b) Here's a set of items:

i	1	2	3	4
w_i	3	6	5	2
v_i	5	11	7	8

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

Here's what the table looks like at the end of the algorithm's run:

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	8
3	0	5	5	5	8
4	0	5	5	5	8
5	0	5	5	7	13
6	0	5	11	11	13
7	0	5	11	11	15
8	0	5	11	12	19
9	0	5	16	16	19
10	0	5	16	16	20

We fill this table in column by column, and within each column from top to bottom. To fill in, for example, row 8 column 4, we compare `V[8][3]`, which stores value 12, to `v_4 + V[6][3]`, which is 8 + 11 = 19. 19 is higher, so we record 19 in `V[8][4]`.

Once we have the table, we can retrace our steps to find the set of items to steal. We first see that `V[10][4] = 20 = 8 + V[8][3]`, which means that the option including item 4 was

better than the option not including item 4 (`V[10][3] = 16`). So we add item 4 to our optimal solution. Now, continuing to trace backwards, we have `V[8][3] = 12 = 7 + V[3][2]`, which means that the option including item 3 was better than the option not including item 3 (`V[8][2] = 11`). So we add item 3 to our optimal solution. Next, we see that `V[3][2] = 5 = V[3][1]`, which means that the option not including item 2 was better than the option including item 2. We do not add item 2 to our optimal solution. Finally, `V[3][1] = 5 = 5 + V[0][0]`, indicating that we include item 1 in our solution. Our answer is thus that we want to steal items 1, 3, and 4, giving us total value 20.

(c) In terms of n (the number of items) and W (the weight your knapsack can hold), what is the runtime of your algorithm?

Solution: This takes time $\Theta(nW)$. Lines 5-11 involve two nested for loops. The outer loop (line 5) runs n times, and the inner loop runs W times. All of the work inside the inner loop (lines 7-11) takes constant time.

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

Solution: We first argue that this problem has optimal substructure. Imagine we have an optimal solution to this problem (importantly, we are not thinking about a specific solution, or a specific instance of the problem). This optimal solution has some number of items in it, call that number k. Now imagine that we split up that optimal solution into two pieces, where all items up to and including item j are in the first piece and all items after item j are in the second piece, where $1 < j < n$.