COSC 311: ALGORITHMS
HW3: GREEDY AND DYNAMIC PROGRAMMING

Due Friday, November 8 in class

1 Submission and Collaboration

This assignment is due on Friday, November 8 at the start of class; bring a hard copy of your typed responses to submit in class. **Each problem must be on a separate sheet of paper, with your name and section number at the top of the page.** If you use more than one sheet of paper for an individual problem, please staple together those pages. There is no need for you to rewrite the problems on your submission.

In accordance with this course's intellectual responsibility policy, you are welcome to discuss the problems with other students who are currently taking the course, but you must write up your solutions individually. Please note at the top of your submission with whom you discussed each problem.

2 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.

(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?)

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

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

(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.

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.

(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.

(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?)

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

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

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.

(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?)

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

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