Homework #7

You do not need to turn in these problems. The goal is to reinforce what we learned in class, as well as to cover material we didn't have time to cover in class. The in-class quiz that will cover the same or similar problems. Material on homework can also appear on exams.

Problem 1: Knapsack: which items to take?

In class we developed the dynamic programming solution for the knapsack problem, which allows us to compute the *value* of the optimal solution (OPT) in $\Theta(nW)$ time. In this problem you will extend this solution to compute which items the thief should take.

(a) At first glance we might think that we can compute the optimal solution at the same time as we are computing the optimal value. Consider the following naive modification (underlined code) of the iterative knapsack solution we saw from class, and explain why it does not work. Hint: why can't we start printing items to take until we compute all the way up to $M[n, W]$?

Solution

Each optimal sub-solution is not necessarily contained in the overall optimal solution. In other words, you can't decide what to pick until you've completed the entire OPT matrix.

(b) Describe how once M is computed, we can generate the list of items that the thief should take. What is your algorithm's asymptotic running time?

Solution

Keep an auxiliary matrix OPTLIST that holds the list of items that corresponds to an optimal solution. In other words, if OPT[5][10] holds the maximum value obtained from 5 items and capacity 10, OPTLIST[5][10] holds the list of items that make up that maximum value. If each list in OPTLIST has to be copied and appended to at each index, the runtime is $O(n^2Z)$. Think of how you can use pointers to reduce the runtime to $O(nZ)$.

Problem 2: Summing Integers

Suppose you are given a collection $A = \{a_1, a_2, \ldots, a_n\}$ of n positive integers that add up to $2Z$. Design an $O(nZ)$ time algorithm to decide if the set can be partitioned into two groups B and $A - B$ such that:

$$\sum_{a_i \in B} a_j = \sum_{a_i \in (A-B)} a_i = Z$$

Solution

Let $m[i, z]$ be 1 if there exists a subset of $\{a_1, a_2, \ldots, a_i\}$ that sums to z and 0 otherwise. The recursive formula for computing $m[i, z]$ is:

$$\begin{aligned} m[0,0] &= 1 \\ m[0,z] &= 0 \text{ for all } z \neq 0 \\ m[i,z] &= \max \begin{cases} m[i-1,z] \\ m[i-1,z-a_i] \end{cases} \end{aligned}$$

In the last part of the recurrence, the first choice corresponds to not including a_i in the sum, while the second choice corresponds to including a_i in the sum. In the end, the answer to the problem is "yes" if and only if $m[n, Z] = 1$. The size of the table we're filling in is $n \times Z$, and each entry can be filled in in constant time, so the overall running time is $O(nZ)$. It should be straightforward to write pseudocode that calculates the values for this table.

Problem 3: Dynamic Programming for Change

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. Give a dynamic programming algorithm that makes change for an amount of money n using as few coins as possible. What is its running time? (Recall from a past homework that the natural greedy algorithm doesn't work for all sets of coin denominations. The dynamic programming algorithm should always work.)

Solution

Let $C[p]$ be the minimum number of coins of the k denominations that sum to p cents. There must exist some "first coin" d_i, where $d_i \leq p$. The remaining coins in the optimal solution must be the optimal solution to making change for $p - d_i$ cents. Then $C[p] = 1 + C[p - d_i]$. But which coin is d_i? We don't know, so the optimal solution is the one that minimizes $C[p]$ over all choices of i such that $1 \leq i \leq k$ *and* $d_i \leq p$. Also, when $p = 0$, the optimal solution is clearly to have 0 coins:

$$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i:1 \leq i \leq k \wedge d_i \leq p}\{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

Problem 4: End of Semester Dynamic Programming

It's the end of the semester, and you're taking n courses, each with a final project. Each project will be graded on a scale of 1 to $g > 1$, where a higher number is a better grade. Your goal is to maximize your average grade on the n projects. (Note: this is equivalent to maximizing the *total* of all grades, since the difference between the total and the average is just the factor n.)

You have a finite number of hours $H > n$ to complete all of your course projects; you need to decide how to divide your time. H is a positive integer, and you can spend an integer number of hours on each project. Assume your grades are deterministically based on time spent; you have a set of functions $\{f_i : 1, 2, \ldots, n\}$ for your n courses; if you spend $h \leq H$ hours on the project for course i, you will get a grade of $f_i(h)$. The functions f_i are nondecreasing; spending more time on a course project will not *lower* your grade in the course.

To help get you started, think about the (i, h) subproblem. Let the (i, h) subproblem be the problem that maximizes your grade on the first i courses in at most h hours. Clearly, the complete solution is the solution to the (n, H) subproblem. Start by defining the values of the $(0, h)$ subproblems for all h and the $(i, 0)$ subproblems for all i (the latter corresponds to the grade you will get on a course project if you spend *no* time on it).

Give the dynamic programming equation to define the value of the optimal solution (i.e., the value of any (i, h) subproblem, where $0 \leq i \leq n$ and $0 \leq h \leq H$), describe an algorithm for iteratively computing the value of the optimal solution, and describe an algorithm for recreating the optimal solution (i.e., mapping your H hours to your n projects).

Solution

Let $Opt(i, h)$ be the maximum total grade that can be achieved for this subproblem. Then $Opt(0, h) = 0$ for all h and $Opt(i, 0) = \sum_{j=1}^{i} f_j(0)$ for all i. Now in the optimal solution to the (i, h) subproblem, one spends k hours on course i for some value $k \in [0, h]$. Thus:

$$Opt(i, h) = \max_{0 \leq k \leq h} f_i(k) + Opt(i - 1, h - k)$$

To compute the table, we first fill in the values of $Opt(0, h)$ for all h and the values of $Opt(i, 0)$ for all i. Then, starting with $Opt(1, 1)$, we fill in the values in row major order. The value in the table at location (n, H) (the bottom right corner) is the maximum possible grade.

To be able to recreate the optimal solution from the table (i.e., how many hours to spend on each course), we also record the value k that produces the maximum for each decision in the table. Then starting in entry (n, H) in the table, we can trace back through the table of optimal values to find the solution. Specifically, we look at entry (n, H) and the value of k recorded for it. This is how many hours to spend on course n. We then go to entry $(i - 1, h - k)$ for that particular k and repeat.