EE360C: Algorithms
University of Texas at Austin — Homework #8
Dr. Christine Julien, Dr. Vallath Nandakumar — Due Date: April 5, 2018

Homework #8

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **You do not need to turn in these problems. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: 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:

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

Solution

Let $m[i, z]$ be 1 if there exists a subset $\{a_1, a_2, \ldots, a_i\}$ 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 2: Dynamic Programming

Yes, this homework problem should look really familiar to you!

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.

Given these functions, decide how many hours to spend on each project (again, in integer numbers of hours) to maximize your average grade. Your algorithm should be polynomial in n, g, and H.

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).

For full credit, argue that the problem has optimal substructure, 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:

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

Obviously, the problem has optimal substructure as the value of an optimal solution is based on the values of the optimal solutions to smaller subproblems.

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.

Problem 3: Planning a Company Party

You are consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a *conviviality* rating, which is a real number. In order to make the party the most fun for all of the attendees, the president does not want both an employee and his or her immediate supervisor to attend.

You are given the tree that describes the structure of the corporation, where each node represents an employee, and a node's children represent the employees under his or her direct supervision. Each node of the tree holds, in addition to the child pointers, the name of the employee and the employee's conviviality rating.

Give a dynamic programming algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Hints:

- Construct your solution as a *binary choice.*
- Even the president of the company may end up not invited if that turns out to be optimal.
- For full credit, argue that the problem exhibits optimal substructure, define the optimal solution via recursion, provide a description or pseudocode of the bottom-up algorithm, and analyze the running time. For partial credit, give some of these.

Solution

First, the problem has optimal substructure. That is, an optimal solution for the entire tree contains within it optimal solutions for subtrees. Consider an optimal solution that includes some node i but none of the employees under i's direct supervision (because that wouldn't be a correct solution, let alone an optimal one). We can break this solution into components, one rooted at each of i's *grand-children* (i.e., employees under the direct supervision of employees under i's direct supervision). I claim that the sub-solution created by breaking our original optimal solution in this way is an optimal solution to the subproblem rooted at this node. The proof is simple; if it wasn't optimal, I could replace it with a sub-solution that WAS optimal, combine it with the other subproblems, and end up with a MORE optimal solution than the original one. Which is a contradiction.

The binary choice here is whether or not a particular employee is invited to the party. We will create an array of size n (the number of employees in the company), where the president is entry n in the array. The first entries in the array are employees at the bottom "level" of the tree; employees with no others under their supervision are leaves. I start by making a list of the employees to give them indices into this array; I do this with a breadth first search starting at the president, then I reverse the resulting list (so that the president is at the end and the leaves are at the beginning. This took $O(n)$ time. Then I start at the beginning of this list and use the list in conjunction with the tree that I was given. I assume c_i refers to the conviviality rating of employee i in this list (i.e., c_n is the conviviality rating of the president).

Solution

The optimal solution, defined via recursion, is:

$$C[i] = \max(c_i + \sum_{g \in \text{grandchildren}} C[g], \sum_{g \in \text{children}} C[g])$$

This gives me the *value* of the optimal solution; I can also create a second array $I[i]$ (for "invite"), which contains a 1 in location i if the first choice was the max and a 0 if the second choice was the max. This is *NOT* the guest list. See below to create the guest list. The pseudocode for the bottom up algorithm is the following (the below assumes that if there are no children (or grandchildren) then the sum of their conviviality ratings is 0):

```
PARTY
1  for $i = 1$ to $n$
2        do $C[i] = \max(c_i + \sum_{g \in \text{grandchildren}} C[g], \sum_{g \in \text{children}} C[g])$
3           if $c_i + \sum_{g \in \text{grandchildren}} C[g] > \sum_{g \in \text{children}} C[g]$
4              then $I[i] = 1$
```

This gets the C and I tables filled in; the following tells us how to create the guest list:

```
INVITE($i$)
1  if $I[i] = 1$
2     then output $i$
3           for every grandchild $g$ of $i$
4                 do INVITE($g$)
5  if $I[i] = 0$
6     then for every child $c$ of $i$
7                 do INVITE($c$)
```

The party procedure fills in n entries in the C and I tables iteratively; filling in each entry takes $O(n)$ time (it just looks up entries in the data structure or tables and does simple arithmetic, but the sum may be over $O(n)$ other elements). Therefore the overall running time to compute the value of the optimal solution is $O(n^2)$. To print out the guest list, each entry in I is visited at most once, and constant work is done on each visit. This is an overall running time of $O(n)$ as well.