CSC373 Fall'19

Tutorial 3

Mon. Sept. 30, 2019

Q1 Coin change

Consider again the problem of making change when the denominations are arbitrary.
Input: Positive integer "amount" A, and positive integer "denominations" $d[1] < d[2] < ... < d[m]$.
Output: List of "coins" $c = [c[1], c[2], ..., c[n]]$ where each $c[i]$ is in d, repeated coins are allowed (possible for $c[i] = c[j]$ with $i \neq j$), $c[1] + ... + c[n] = A$, and n is minimum. If no solution is possible, output $n = 0$ and an empty list c.
Example: If we only have pennies, dimes and quarters to make change for 30c, then the input is $d = [1, 10, 25]$ and an optimum output is $c = [10, 10, 10]$. If we only have nickels, dimes and quarters to make change for 52c, then an optimum output is $c = []$ – no solution exists.

Follow the dynamic programming paradigm to solve this problem.

(a) Describe the recursive structure of sub-problems.

(b) Define an array that stores optimum values for arbitrary sub-problems.

(c) Give a recurrence relation (Bellman equation) for the array values, based on the recursive structure of sub-problems.

(d) Write a simple algorithm to compute the array values bottom-up.

(e) Use the computed array values to reconstruct an optimum solution; when necessary, define a second array to store partial information about solutions and modify the algorithm from part (d) accordingly. Then, analyze the worst-case runtime of your algorithm carefully. Does it run in polynomial time? Explain.

solution

For $i = 0, \ldots, A$ and $j = 1, \ldots, d$, let $C[i, j]$ be the minimum number of coins required to make change for amount i using denominations $d[1], \ldots, d[j]$. If it's impossible to make change in this way, let $C[i, j] = \infty$.

Clearly $C[0, j] = 0$ for all j, and $C[i, 0] = \infty$ for all $i > 0$. For $i, j > 0$ we can determine $C[i, j]$ as follows. Consider the following two cases:

1. *It's possible to make change for i using $d[1], \ldots, d[j]$, and the solution with the fewest coins involves at least one copy of $d[j]$.* Then if we remove a copy of $d[j]$ from this solution, the remaining sequence of coins must be an optimal way to make change for $A - d[j]$. Therefore $C[i, j] = 1 + C[i - d[j], j]$.

2. *Either it's impossible to make change for amount i using $d[1], \ldots, d[j]$, or it's possible but the optimal solution doesn't use any copies of $d[j]$.* Either way, $C[i, j] = C[i, j - 1]$.

If $A < d[j]$ then only the second case above is possible. Therefore, if $A < d[j]$ then $C[i,j] = C[i,j-1]$, otherwise $C[i,j] = \min(1 + C[i-d[j],j], C[i,j-1])$.

We can compute C as follows. For i from 0 to A, for j from 0 to m, compute $C[i,j]$ as described above. Each computation takes constant time using the previously computed elements of C, so the loop takes $O(Am)$ time overall.

Given C, we can reconstruct the output list of coins as follows. Let $i = A$ and $j = m$. While $i, j > 0$ do the following. If $C[i,j] = C[i,j-1]$ then let $j = j-1$; otherwise add a copy of $d[j]$ to the output list and let $i = i - d[j]$. This loop has a runtime of $O(m + A)$, so the algorithm has a runtime of $O(mA)$ overall.

This runtime is not polytime because it's linear in the value of A, which means it's exponential in the number of bits required to write down the value of A. This kind of runtime (tehcnically exponential as a function of the input size, but polynomial as a function of input values) is called "pseudo-polynomial time".

Q2 Longest Increasing Subsequence

Consider the following Longest Increasing Subsequence (LIS) problem:
Input: $I = \langle a_1, a_2, \ldots, a_n \rangle$ an ordered sequence of n integers.
Output: An ordered subsequence S of I such that each member of S is strictly larger than all the members that have come before it, and S contains as many integers as possible.
Example: For $I = \langle 4, 1, 7, 3, 10, 2, 5, 9 \rangle$, we want $S = \langle 1, 3, 5, 9 \rangle$ or $S = \langle 1, 2, 5, 9 \rangle$. $\langle 6, 7, 8 \rangle$, $\langle 1, 2, 3 \rangle$ are not subsequences (they either include integers not in I or include integers out-of-order); $\langle 4, 1, 7, 10 \rangle$ is not increasing; $\langle 1, 3, 9 \rangle$ is not as long as possible.

In other words, the ordering of S must respect the ordering of I, S's members must be strictly increasing, and S must be as long as possible.

Write an efficient algorithm to solve the longest increasing subsequence problem. Briefly justify its correctness and runtime.

solution

For $1 \leq i \leq n$ let $L[i]$ be the length of a longest increasing subsequence of I that ends with a_i. Any such subsequence can be divided into two parts: *first*, an increasing subsequence S of $\langle a_1, \ldots, a_{i-1} \rangle$ that's as long as possible subject to the constraint that the last element of S is less than a_i; and *second*, a_i itself. Therefore,

$$L[i] = 1 + \max_{1 \leq j < i \text{ and } a_j < a_i} L[j],$$

where we define the maximum over the empty set to be 0. Also let $M[i]$ be a j that achieves the maximum $L[j]$ in the above expression, and if no such j exists (i.e. the maximum is over the empty set) then $M[i] = 0$.

We can compute $(M[1], L[1]), \ldots, (M[n], L[n])$ in this order using the above recurrence. Then, to construct the longest increasing subsequence, do the following. Let i be the index of L that maximizes $L[i]$. While $i \neq 0$, append a_i to the front of the output sequence, and then let $i \leftarrow M[i]$.

Computing $(M[i], L[i])$ for any given i takes $O(n)$ time, so constructing (M, L) takes $O(n^2)$ time. Then constructing the output takes $O(n)$ time, so overall the algorithm takes $O(n^2)$ time.