Design and Analysis of Algorithms March 6, 2015
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Erik Demaine, Srini Devadas and Nancy Lynch Recitation 5

Dynamic Programming

Rectangular blocks

Given a set of n rectangular three-dimensional blocks, where block B_i has length l_i, width w_i, and height h_i, all real numbers, find the maximum height of a tower of blocks that is as tall as possible, using any subset of the blocks, such that the tower respects the following constraints:

1. Blocks cannot be rotated: the length always refers to the east-west direction, the width is always north-south, and the height is always up-down.
2. Block B_i can be stacked on top of block B_j only if $l_i \leq l_j$ and $w_i \leq w_j$, that is, the two dimensions of the base of block B_i are no greater than those of block B_j.

Sub-problem definition

First we assume that our blocks $\{1, 2, \ldots, n\}$ are arranged in non-increasing order of length and then breadth – note that this is easy to do as a pre-processing step that takes $O(n \log n)$ time.

Given this ordering of blocks, we define $H[i]$ to be the height of the tallest tower that has block B_i at the top.

Recursive formulation

We see that $H[i]$ can be expressed as,

$$H[i] = h_i + \max_{j<i; l_j \geq l_i; w_j \geq w_i} H[j]$$

If there exists no compatible box in the above maximization, the max term equals 0.

Our base case here is $H[1] = h_1$.

In this problem, we're guessing over all possible legal blocks under the top block (block B_i in this case).

The final answer is then $\max_i H[i]$.

Runtime analysis

Total number of sub-problems here is $O(n)$, and the total time required to solve each sub-problem is $O(n)$, which means that the total running time of the Dynamic Programming part of this algorithm is $O(n^2)$.

Note that sorting the boxes in non-increasing order of lengths and then breadths takes $O(n \log n)$ time, which means the total running time of this algorithm is $O(n \log n + n^2) = O(n^2)$.

Counting Boolean Parenthesizations

Given a boolean expression consisting of a string of the symbols `True`, `False`, `AND`, `OR`, and `XOR`, count the number of ways to parenthesize the expression such that it will evaluate to `True`. For example, there are 2 ways to parenthesize `True AND False XOR True` such that it evaluates to `True`.

Sub-problem definition

Let $T[i,j]$ be the number of ways of parenthesizing the string $S[i:j]$ such that the expression between indices i and j evaluates to `True`, and $F[i,j]$ be the number of ways of parenthesizing the string $S[i:j]$ such that the expression between indices i and j evaluates to `False`. Here indices i and j are inclusive.

Recursive formulation

Then, given that $Tot[i,j] = T[i,j] + F[i,j]$, we see that $T[i,j]$ and $F[i,j]$ can be expressed by the following recursive formulations,

$$T[i,j] = \sum_{k=i}^{j-1} \begin{cases} T[i,k]\cdot T[k+1,j] & \text{if } S[k] = \texttt{AND} \\ Tot[i,k]\cdot Tot[k+1,j] - F[i,k]\cdot F[k+1,j] & \text{if } S[k] = \texttt{OR} \\ T[i,k]\cdot F[k+1,j] + F[i,k]\cdot T[k+1,j] & \text{if } S[k] = \texttt{XOR} \end{cases}$$

$$F[i,j] = \sum_{k=i}^{j-1} \begin{cases} F[i,k]\cdot F[k+1,j] & \text{if } S[k] = \texttt{OR} \\ Tot[i,k]\cdot Tot[k+1,j] - T[i,k]\cdot T[k+1,j] & \text{if } S[k] = \texttt{AND} \\ T[i,k]\cdot T[k+1,j] + F[i,k]\cdot F[k+1,j] & \text{if } S[k] = \texttt{XOR} \end{cases}$$

We have the following base cases, for all i between 1 and n,

$$\begin{aligned} T[i,i] &= 1 \quad \text{if } S[i] = \texttt{True} \\ T[i,i] &= 0 \quad \text{if } S[i] = \texttt{False} \\ F[i,i] &= 0 \quad \text{if } S[i] = \texttt{True} \\ F[i,i] &= 1 \quad \text{if } S[i] = \texttt{False} \end{aligned}$$

The final answer is $T[1,n]$ (the total number of ways of obtaining a final value of `True` given that the entire string $S[1:n]$ is used.

Here, we are making a guess over the positions of the outermost parentheses in the expression sub-stringed between i and j.

Runtime analysis

Total number of sub-problems here is $O(n^2)$, and the amount of time required to solve each sub-problem is $O(n)$, so the total runtime complexity of the algorithm is $O(n^3)$.

Make change

Given a value N, if we want to make change for N cents, and we have infinite supply of each of $S = \{S_1, S_2, .., S_m\}$ valued coins, what's the minimum number of coins needed to get to a total of N? For simplicity, assume that $S_1 > S_2 > \ldots > S_m$.

Sub-problem definition

Let $C[p]$ be the minimum number of coins need to make change for p cents using coins of denominations $S_1, S_2, \ldots, S_m$.

Recursive formulation

If $p > 0$,

$$C[p] = \min_{i:S_i \leq p} C[p - S_i] + 1$$

Our base case is $C[0] = 0$.

The final answer is $C[N]$.

In this formulation, we make a guess on which coin denomination S_i belongs to the optimal configuration.

Runtime analysis

The total number of sub-problems here is $O(N)$, and the total time required to solve each sub-problem is $O(m)$, so the total runtime complexity of this algorithm is $O(mN)$.

Note that this runtime is pseudo-polynomial (similar to Knapsack).

Other problems

- (Warmup) A robot starts from the top left corner $(1, 1)$ of a $M \times N$ grid. The goal of the robot is to reach right bottom (M, N). At each step the robot can make one of the two choices - move one cell right, move one cell bottom. Write a function which takes M and N as arguments and returns the total number of paths the robot can take to reach its destination.
- Consider an input text consisting of n words, each of lengths $l_1, l_2, \ldots, l_n$ characters. We want to print this text as neatly as possible, with the restriction that each line can hold only a maximum of M characters. The goal here is to minimize the sum over all lines except the last, of the cubes of the numbers of extra space characters at the end of each line. The number of extra characters at the end of a line that consists of words i through j is $M - j + i + \sum_{k=i}^{j} l_k$.