2^n Dynamic Programming

Lawrence Wang and Kevin Geng*

2 December 2016

1 Problem statement

(USACO December 2014, Gold Problem 1: Guard Mark)

Farmer John and his herd are playing frisbee. Bessie throws the frisbee down the field, but it's going straight to Mark the field hand on the other team! Mark has height H ($1 \leq H \leq 1,000,000,000$), but there are N cows on Bessie's team gathered around Mark ($2 \leq N \leq 20$)[1]. They can only catch the frisbee if they can stack up to be at least as high as Mark. Each of the N cows has a height, weight, and strength. A cow's strength indicates the maximum amount of total weight of the cows that can be stacked above her.

Given these constraints, Bessie wants to know if it is possible for her team to build a tall enough stack to catch the frisbee, and if so, what is the maximum safety factor of such a stack. The safety factor of a stack is the amount of weight that can be added to the top of the stack without exceeding any cow's strength.

2 $N!$ solution

A naive solution would be to recursively construct all permutations of the N cows, where on each level of recursion we append a cow to the bottom of the stack. This solution has complexity $O(N!)$, which is too slow for the input size.

To see why the naive solution performs so badly, we can examine the search tree for a small case (say $N = 4$). Notice how if we select any subset of cows, there is only one optimal permutation that maximizes the height and minimizes the total weight. For example, there will be one optimal way to order cows A, B, and C. However, we notice that in the search tree, there are multiple paths originating from the root of length three that consist of cows A, B, and C.

If we want to check all solutions that begin with some permutation of A, B, and C and end with a fourth cow D, then we only need to consider the optimal ordering of A, B, and C. We can do this because we are trying to add D to the bottom, and the total weight of the three cows will not change.

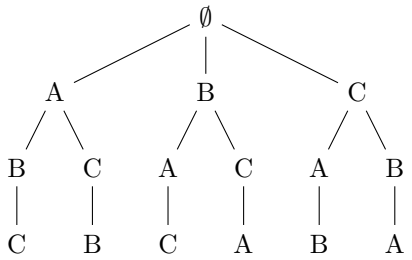

Figure 1: All permutations of A, B, and C.

*Highly inspired by Samuel Hsiang's lecture from last year.

[1]Whenever you see $N \leq 20$, an exponential-time solution is likely!

3 2^n solution

Thus, we wish to constructively generate subsets of the N cows rather than permutations, since each subset encapsulates an optimal permutation. This leads us to develop *subset DP*, which allows us to perform DP over 2^n possible subsets instead of $n!$ permutations.

The base case of our dynamic programming formulation are subsets containing one element. For our recurrence relation, we consider a subset S, which has some optimal ordering that has already been computed. In order to increase the size of this subset, we consider adding a cow that is not already in S to the bottom of S. This is possible whenever the strength of the cow to be added is greater than the total weight of S.

Adding cows to an existing subset takes $O(N)$ since in the worst case we must check every cow to add. Since there are $O(2^N)$ total subsets of the N cows (why?), the overall time complexity is $O(N2^N)$. While this may still appear to be very slow, it is sufficient for the input limits and is a significant speed-up from the naive $O(N!)$ solution.

4 Bit Masking

While the above solution theoretically solves the problem, it may at first appear too clumsy to implement. Fortunately, we can compactly and elegantly represent subsets using a bit string: there is a bijective mapping between a bit string of size N and subsets of a size N set. Specifically, we can represent each subset as a sequence of bits, where each bit corresponds with a cow. If a bit is set to one, then the corresponding cow is in the subset, while if a bit is set to zero, then the cow is not in the subset.

We can manipulate bits using bit masking operations: namely, and, or, xor, not, left shift, and right shift.[2] For the most part, however, we only need to know a few operations:

1. `num | (1 << i)` to set bit i
2. `num & ~(1 << i)` to clear bit i
3. `num ^ (1 << i)` to toggle bit i
4. `num & (1 << i)` to test bit i

Algorithm 1 2^n solution

for $i \leftarrow 0 \dots 2^n - 1$ **do** ▷ i represents the subset S
  $dp(index) \leftarrow \infty$ ▷ safety factor
  for all $j \in S$ **do** ▷ such that bit j is set in i
    $prev \leftarrow i - 2^j$ ▷ clear bit j
    $alt \leftarrow min(dp(prev), strength(j) - \sum_{k \in S \setminus \{j\}} weight(j))$ ▷ calculate new safety factor
    if $dp(i) < alt$ **then**
      $dp(i) \leftarrow alt$

This pseudocode finds the maximum safety factor for any subset. There are, however, a few more complications in the problem that need to be handled. Specifically, we need to keep track of the total weight and height of each subset to keep the runtime $O(n * 2^n)$, as well as determine whether any subset actually exceeds the requested height. Completing the implementation of these details is left as an exercise to the reader.

[2] Throwback to ACSL!

5 More problems

1. USACO 2015 January, Gold Problem 2: Movie Mooving

   Bessie is out at the movies. Being mischievous as always, she has decided to hide from Farmer John for L ($1 \leq L \leq 100,000,000$) minutes, during which time she wants to watch movies continuously. She has N ($1 \leq N \leq 20$) movies to choose from, each of which has a certain duration and a set of showtimes during the day. Bessie may enter and exit a movie at any time during one if its showtimes, but she does not want to ever visit the same movie twice, and she cannot switch to another showtime of the same movie that overlaps the current showtime.

   Help Bessie by determining if it is possible for her to achieve her goal of watching movies continuously from time 0 through time L. If it is, determine the minimum number of movies she needs to see to achieve this goal (Bessie gets confused with plot lines if she watches too many movies).

2. USACO 2014 February, Gold Problem 2: Cow Decathlon

3. USACO 2013 January, Gold Problem 2: Island Travels