2^n Dynamic Programming and Masking

Samuel Hsiang

December 4, 2015

Today we will discuss a rather standard USACO Gold topic. Every year one or two of these problems show up.

1 Dynamic Programming over Subsets

We've already covered how dynamic programming can turn exponential solutions into polynomial solutions, but it can also help turn factorial solutions into exponential. Problems where the bound on n is 20, for example, signal that an exponential solution is the one required. Consider the following problem:

(USACO December 2014, guard) 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 ($1 \le H \le 1,000,000,000$), but there are N cows on Bessie's team gathered around Mark ($2 \le N \le 20$). 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.

We can try the $O(N!)$ brute force, trying every permutation of cows possible. However, this is far too slow. $N \le 20$ hints at an exponential solution, so we think of somehow iterating over every possible subset of the cows. Given a subset S of cows, the height reached is the same, so perhaps we sort the subset by strength, and put the strongest cow on the bottom. We see that this greedy approach fails: suppose that the first cow has weight 1 and strength 3 and the second cow has weight 4 and strength 2. Greedy would tell us to put the first cow on the bottom, but this fails, while putting the second cow on the bottom succeeds.

When greedy fails, the next strategy we look at is dynamic programming. To decide whether S is stable, we have to find whether there exists a cow j in S that can support the weight of all the other cows in S. But how do we know whether the set $S \setminus \{j\}$ is stable? This is where dynamic programming comes in.

This leads to a $O(N2^N)$ solution. This seems like a pain to code iteratively, but there is a nice fact about subsets: there is a cute bijection from the subsets of $\{0, 1, 2, \ldots, N-1\}$ to the integers from 0 to $2^N - 1$. That is, the subset $\{0, 2, 5, 7\}$ maps to $2^0 + 2^2 + 2^5 + 2^7 = 165$ in the bijection. We call this technique *masking*. We require all the subsets of S to be processed before S is processed, but that property is also handled by our bijection, since subtracting a power of 2 from a number decreases it. With a little knowledge of bit operators, this can be handled easily.

```
for $i \leftarrow 0, 2^N - 1$ do                              ▷ $i$ represents the subset $S$
    $dp(i) \leftarrow -1$
    for all $j \in S$ do                                      ▷ $j \in S$ satisfy i & (1 << j) != 0
        $alt \leftarrow \min(dp(i - 2^j), strength(j) - \sum_{k \in S \setminus \{j\}} weight(k))$
        if $dp(i) < alt$ then
            $dp(i) \leftarrow alt$
```

`&` is the bitwise and function, while `<<` is the left shift operator.

2 Bit Operators

Pay special attention to `unsigned` vs. `signed`, etc. In general this shouldn't matter too much since we don't really care about negative numbers.

Operator	Name	Usage	Note
`<<`	leftshift	`5 << 3 == 40`	shift and append 0; in particular, `1 << k` is 2^k
`>>`	rightshift	`-6 >> 2 == -2`	shift and prepend either 0 or 1 depending on first bit
`>>>`	unsigned rightshift	`-6 >>> 2 == -2`	always prepend 0; Java only (C, C++ use `unsigned`)
`&`	bitwise AND	`5 & 3 == 1`	not the same as `&&`
`|`	bitwise OR	`5 | 3 == 7`	not the same as `||`
`~`	bitwise NOT	`~5 == -6`	not the same as `!`
`^`	bitwise XOR	`5 ^ 3 == 2`	

These make your lives a lot easier.

3 Example Code

Not every bit operator is represented, but the operators needed for masking all are.

```
#include <iostream>
#include <fstream>
#include <algorithm>
using namespace std;
typedef long long ll;

ll N;
ll H;
ll th[1 << 20];
ll tw[1 << 20];
ll msf[1 << 20];
ll h[20];
ll w[20];
ll s[20];

int main() {
    ifstream fin("guard.in");
    ofstream fout("guard.out");
    fin >> N >> H;
    for(int k = 0; k < N; ++k) {
        fin >> h[k] >> w[k] >> s[k];
    }
    int cap = 1 << N;
    ll ans = -1;
    msf[0] = (1<<31)-1;
    for(int index = 1; index < cap; ++index) {
        for(int j = 0; j < N; ++j) {
            if(index & (1 << j)) {
                int prev = index - (1 << j);
                tw[index] = max(tw[index], w[j] + tw[prev]);
                if(s[j] >= tw[prev]) {
                    th[index] = max(th[index], h[j] + th[prev]);
                    msf[index] = max(msf[index], min(s[j] - tw[prev], msf[
   prev]));
                    if(th[index] >= H) {
                        if(msf[index] > ans) {
                            ans = msf[index];
                        }
                    }
                }
            }
        }
    }
    if(ans == -1) {
        fout << "Mark is too tall\n";
    } else {
        fout << ans << '\n';
    }
    fin.close();
    fout.close();
    return 0;
}
```

4 Problems

1. USACO 2015 January Contest, Gold Problem 2. Moovie Mooving
2. USACO 2014 February Contest, Gold Problem 2. Cow Decathlon
3. USACO 2013 January Contest, Gold Problem 2. Island Travels