Code Jam 2012 - World Finals

Analysis: Upstairs/Downstairs

Upstairs/Downstairs was a last-minute addition to the Finals, replacing a year-old problem proposal that had appeared on a separate contest two months before. This problem loosely mirrors an experience the author had once while on vacation. He was downstairs.

Solving this problem involves two observations and an algorithm. Before making our observations, let's start by writing out the formula for the quantity we want to minimize. p_i will represent the probability that the i^{th} activity Konstantin performs will result in Ilia being awake:

$$P(\text{woken up}) =$$
$$(1-p_0) * (1 - (1-p_1)(1-p_2)*...* (1 - p_K)) +$$
$$p_0(1-p_1) * (1 - (1-p_2)(1-p_3)*...*(1 - p_K)) +$$
$$p_0p_1(1-p_2) * (1 - (1-p_3)(1-p_4)*...*(1 - p_K)) +$$
$$...$$

Observation 1: Noisy First

For our first observation, we'll look for a reason to prefer that Konstantin perform noisier or quieter activities earlier. Intuitively, we suspect that noisier activities should come first: a strategy that keeps Ilia awake and then tries to keep him asleep seems pretty reasonable.

Looking at the structure of the formula above, we can look for differences between how p_i and p_{i+1} are used. Here are the terms in which swapping them would lead to a difference in the final result:

$$p_0p_1...p_{i-1}(1-p_i) * (1 - (1-p_{i+1})(1-p_{i+2})*...*(1 - p_K)) +$$
$$p_0p_1...p_{i-1}p_i(1-p_{i+1}) * (1 - (1-p_{i+2})*...*(1 - p_K))$$

Simplifying these two terms, we have:

$$p_0p_1...p_{i-1} * [$$
$$(1-p_i) + p_i(1-p_{i+1}) -$$
$$(1-p_i)(1-p_{i+1})...(1-p_K) -$$
$$p_i(1-p_{i+1})...(1-p_K)]$$
$$= p_0p_1...p_{i-1} * [1 - p_ip_{i+1} - (1 - p_{i+1})(1 - p_{i+2})...(1 - p_K)]$$

This quantity is minimized by choosing $p_{i+1} < p_i$, which confirms the intuition that noisier activities should happen earlier. Repeatedly swapping adjacent activities like this shows that in an optimal solution, activities should be performed from noisiest to quietest.

Observation 2: Go Extreme!

Our original formula for $P(\text{woken up})$ is clearly linear in p_i for all i. This quantity therefore must be minimized by taking either the largest possible value or the smallest possible value for p_i.

Putting this observation together with the previous one, we can conclude that Konstantin should start by performing the $K-q$ noisiest activities in order from noisiest to quietest, and then should perform the q quietest activities, also in order from noisiest to quietest. We'll call the first set of activities the *prefix*, and the others the *suffix*. Note that here we're considering an activity that can be repeated c times as c different activities.

Possible Algorithms

For the Small, it's sufficient to try all values of `q`. There are `O(K)` possible values, and evaluating each one naively takes $O(\text{sum}(c_i))$ time, so the running time is $O(\text{sum}(c_i)^2)$.

There are only two numbers we really need to keep track of for each possible suffix: the probability that Ilia will end up being woken up if he starts the suffix awake, and the probability that he will end up being woken up if he starts the suffix asleep. After precomputing those 2 values for each of the K possible suffixes, which we can do in `O(K)` time, we can simulate each possible prefix, also in `O(K)` time, and do a quick lookup for the corresponding suffix for each one. A little math produces the correct answer, in $O(\text{sum}(c_i))$ time.

Another algorithm that boils down to the same thing involves treating Ilia's three states—awake, asleep, and woken up—as a 3x1 vector that can be operated on by matrices. For any activity, we can build a 3x3 transition matrix, looking like this:

```
[[p     0     0]
 [1-p   1-p   0]
 [0     p     1]]
```

Ilia's initial state is:

```
[1
 0
 0]
```

To build the transition matrix for a series of activities, we can multiply the activities' matrices together, with the noisiest activity on the right. In this way, we can compute a 3x3 matrix corresponding to each prefix of length up to K, and one for each suffix of length up to K, in `O(K)` time. Then it's an `O(1)` operation to check the result for any given prefix/suffix pair: multiplying those two matrices by each other, then by Ilia's initial state. The algorithm ends up taking $O(\text{sum}(c_i))$ time.

Note that the bolded entries of the transition matrix, above, correspond to the two numbers we cared about for each suffix in the previous algorithm.

Ternary Search Passed Test Cases, But...

Because this problem was prepared at the last minute, it didn't occur to us until during the contest to wonder whether ternary search would work for selecting `q`, the length of the suffix. We would have been happy with the answer either way, but we would have wanted test data that would break ternary search if the answer turned out to be "no".

It turned out that the answer *was* "no", and that none of our existing cases broke any ternary search that we found. Because of this, some contestants—most notably **misof**, who came third partly on the strength of the points he got from this problem—managed to submit ternary search solutions that passed our test data.

Sometimes it happens in Code Jam that contestants will come up with algorithms that don't work in general given the limits we've provided, but do work on our test data. We try to avoid situations of that sort, but they do happen. We think the consequences here weren't significant, for a few reasons: Implementing the ternary search is about as hard as implementing a correct solution, if not a little harder; the "hard part" of the problem was already done by that point; and if we'd come up with a breaking case, we'd have put it in the Small data as well as the Large. Contestants who implemented ternary search would have seen it was wrong in the Small, and

taken the time to fix it. **misof** had plenty of time separating him from the fourth-place contestant, so the top three likely would not have changed.

Why Ternary Search Fails

Ternary search seems like it should work. Indeed, the function that we're minimizing, P(woken up), is strictly *non-increasing* and then strictly *non-decreasing* in `q`. Unfortunately, that isn't quite enough: it needs to be strictly *decreasing* and then strictly *increasing*; otherwise a large constant patch will leave the ternary search unable to tell which direction it should go in.

Here's an example case that breaks ternary search in principle, and breaks a few contestants' submissions in practice:

```
2
2 200
1/2 40
1/100 400
2 200
1/2 40
99/100 400
```

The correct answer is:

```
Case #1: 0.863976521
Case #2: 0.863976521
```

In the first case, the "1/2" activities are best not performed; but because they are at the very start of the list of activities, outnumbered severely by the "1/100" activities, a standard ternary search will find no difference between the two suffix lengths it tries. It will merely consider a different number of "1/100" activities to be part of the suffix as opposed to the prefix, and leave all the "1/2" activities in place. The second case is identical, but with "99/100" activities instead of "1/100" activities, which should defeat a ternary search that happens to choose the right direction in the first case.

PaweÅ, i GaweÅ,

At dinner after the finals, several Polish contestants were shocked to discover that the problem was not based on a poem called PaweÅ, i GaweÅ,. The similarity was entirely coincidental, as the contestants should have known: GaweÅ,, who (according to Google Translate's translation of that page) "invented the wildest frolics", lived *downstairs*. Still, in retrospect, we wish we'd named the problem after that poem, and perhaps had GaweÅ, minimize the probability that PaweÅ, would go fishing.