Code Jam 2012 - Round 1B

Analysis: Safety in Numbers

Introduction

Like some earlier problems in Code Jam 2012, this one was inspired by the author watching significant amounts of Dancing With the Stars. Unlike previous problems, this one needed no modification to be used in Code Jam (although we don't know what they do in case of a tie): this is exactly how their scoring system works.

Sample Cases

Sometimes it's useful when solving Code Jam problems to start by working to understand the provided sample cases. Let's start there.

Case #1

In the first case, if the first dancer (err, contestant) gets 1/3 of the audience votes, then his score is 30. There's only one way for the rest of the votes to be distributed, which is for 2/3 of the votes to go to the second contestant. That contestant's score would then be 30, which is a tie, and would lead to contestant #1 being safe. Any lower score would result in that contestant being eliminated. We can do similar math on the second contestant.

Case #4

Now let's think about a case where there are more than two contestants. In the fourth sample case, there are three contestants, and their judges' scores are 24, 30 and 21. The sample output claims that the first contestant is safe if she gets 34.666667% of the vote. Is that really true?

That proportion of the votes would give contestant #1 a total of 50 points. For her to be eliminated, all other contestants would need more than 50 points. We can do a little math and find out that for the second contestant to get more than 50 points, he would need more than 26.66666...% of the vote. For the third contestant to get more than 50 points, she would need more than 38.66666...% of the vote. Adding those numbers up gives us 100%.

That means the other contestants would need more votes than the 65.333333% that remain in order to all beat the first contestant, so the first contestant can't be eliminated. If we lowered the first contestant's percentage at all, then the other contestants would all be able to beat her, and she'd end up being eliminated; so 34.666667% is correct.

N Binary Searches

Considering the sample cases has led us very naturally to an algorithm. When we're trying to find the minimum score that will make a particular contestant safe, first we'll choose that contestant's audience vote percentage, and then we'll allocate the rest of the audience votes to other contestants in the worst possible way.

This lends itself very naturally to **binary search**. First we pick a percentage. Is the contestant safe at that percentage? If so, the minimum safe percentage is lower. If not, the minimum safe percentage is higher. Repeat that **N** times, and you have the answer for each contestant.

Those numbers add up to 1!

It's a bit surprising that, when you independently calculate the minimum safe percentage for each contestant, you apparently end up with percentages that add up to 1. A little thought shows us why.

If each contestant has a minimum "safe" percentage, we can say that contestant has a minimum "safe" score (50 for contestant #1 in case #4). Thinking about the last example, we might say the minimum safe score is the value `x` such that we use up 100% of the audience votes if all contestants have a score equal to `x`. The percentages that contestants need to reach that score will naturally add up to 100%.

This is almost true, but there is one exception. Consider the following case: `4 30 0 0 0`. The safe score clearly won't be more than 30, since not all contestants can get to 30, so we have to deal specially with contestants whose score is greater than the safe score. With that in mind, the safe score is the value `x` such that we use up 100% of the audience votes if all contestants have either a score equal to `x`, or 0% of the audience votes and a score greater than `x`.

A Different Binary Search

Now we know how to write a single binary search to find the "safe" score. The safe score is the score for which all contestants with a higher number of judge points get 0% of the audience votes; the other contestants get a percentage that gives them that total score; and the total audience vote percentages add up to 100%.

So to find the safe score, we can try a candidate score, and see if the total audience vote percentage adds up to more than 100%. If so, we need a lower score. If not, we need a higher score. As one of our preparers put it, "Binary search for the number such that it's impossible for all contestants to have that many points, and you're done!"

A Linear Solution

After this editorial was first published, a number of contestants wrote in to tell us that they didn't use binary search at all. One way to avoid it is to use a single loop, plus some math, to compute the "safe" score we talked about before. First, create a sorted list of judges' points. Then, for `i` from 0 to **N**-1, repeatedly determine the following number: if we allocated all the audience votes to the first `i` people to give them the same score, what would that score be? If it's less than the score for contestant `i+1`, or `i+1=`**`N`**, then that's the "safe" score.

Precision

The output of this problem had to be a floating-point number, and any number would be fine as long as it was within 1e-5 of the correct number, absolute or relative. What does that mean?

- **Absolute:** If the correct answer is `y` and you output `x`, you will be judged correct if $|y-x| < 10^{-5}$.
- **Relative:** If the correct answer is `y` and you output `x`, you will be judged correct if $|1-\min(y/x,x/y)| < 10^{-5}$.

We confused some of our contestants by outputting an inconsistent number of decimal places in our sample output. The reason we did that was to illustrate that the number of decimal places you output isn't important, as long as the answer was right; unfortunately, although a number of contestants got an impromptu education on this rule, many more were confused. We'll think about careful ways of presenting problems with floating-point output in the future.

Solving the Small

There's actually an algorithmic approach for this problem that isn't good enough to solve the Large, but does work for the Small, if you aren't up on your binary search. Because each answer you come up with only has to be within 1e-5 of the correct answer, and we know the right answer is between 0 and 100, there are only a couple million numbers you have to check for each user's minimum score that avoids elimination.

Check 0, 0.00001, 0.00002, 0.00003, ... 9.99999, 10. That's 1 million numbers. Then, because the right answer can be within 1e-5 multiplicatively, and the answers we're checking now are more than 10, you can check 10.0001, 10.0002, ..., 99.9999, 100. That's another 900,000 numbers. So by checking 1.9 million numbers, we avoid having to implement binary search; though to be honest, it's probably easier to implement binary search than do that -- and binary search only has to check $\log_2(10^5)=17$ numbers!

Also, note that this method will only work to replace the first binary search we talked about here, not the second one. It isn't enough that the safe score is within 1e-5 of the correct value: the numbers we *output* have to be that close.

Why did people get it wrong?

5608 people downloaded the Small input for this problem, but only 2687 of them got it right. That's an unusually low success rate. So what did those people do wrong?

Some of them outputted a negative percentage for users with a score above the "safe score". Presumably that would have moved the safe score for them as well. Users who did that, and were told that they'd gotten the wrong answer, could have checked their output to see that they'd printed a negative number. In Code Jam you have access to the input and output file you're being tested on; use it!

Others made assumptions that ended up not being true about certain derived values being integers. One contestant who did that got our sample cases right, but outputted `0.0 33.0 33.0 33.0` for the case `4 10 0 0 0`. Still others had problems with integer division: in many programming languages an integer divided by an integer will always return an integer. If you want to avoid that, you have to "cast" one of the integers to a floating-point number -- or just add 0.0 to it, which amounts to the same thing.

Some users might have made a mistake early, then fixed it, but submitted the output for the wrong input file. Whenever you retry a submission, it's important to run your code on the input you just downloaded, or you'll get the wrong answer! We're working on ways to make it easier for users to catch this problem.