Code Jam 2013 - Round 2

Analysis: Multiplayer Pong

Preliminaries

We hope you took our Fair Warning, which we fairly repeated in Fair and Square this year — we consider Big Integers to be fair game, so if your language doesn't natively support them, you'd better have a library to handle them ready.

This problem seems to be about fractions initially, since the ball can hit the walls at fractional positions. There's a way to move it to integers (man, Big Fractions would be too much). The way to go about it is to scale things. Let's scale the time, so now there are $\mathbf{V_X}$ units to a second, and scale vertical distances, so that there are $\mathbf{V_X}$ units to the old unit. This means the vertical speeds stay the same, horizontal speeds get $\mathbf{V_X}$ times smaller, horizontal distances stay the same, and vertical distances grow $\mathbf{V_X}$ times larger. In implementation terms, this means we shrink $\mathbf{V_X}$ to 1 and grow **A** times $\mathbf{V_X}$ — and now suddenly the ball moves a integral number of units upwards and one unit to the side in each unit of time (and so will hit the vertical walls in integral moments of time).

The above assumes $\mathbf{V_X}$ is positive. If it's zero, then the ball will never hit the walls, and so the game ends in a draw. If it's negative, we can flip the whole board across the **Y** axis and swap the teams to make $\mathbf{V_X}$ change signs. Similarly, we can assume $\mathbf{V_Y}$ positive — if it's zero, putting all paddles in the single impact point will guarantee a draw, and if it's negative, we can flip the board vertically.

So now the ball will hit a given vertical wall every 2**B** units of time. It's also relatively easy to figure out what is the position of the impact. If there were no vertical walls, the ball would hit at the initial hit position (**Y** + (**B** - **X**) $\mathbf{V_Y}$), and then at 2**B**$\mathbf{V_Y}$ intervals from there. To calculate the positions in real life, notice that every 2**A** upwards the ball is in the same position again (so we can take impact points module 2**A**), and if that number is larger than **A**, the ball goes in the other direction, and the impact position is 2**A** minus whatever we calculated.

Many bounces

So now we know how to calculate hit positions, so we can just simulate and see who loses first, right? Well, wrong, because the ball can bounce very many times. Even in the small dataset, the number of bounces can go up to 10^{11}, too much for simulation.

Notice that the positions of the paddles are pretty much predetermined. The paddle of a given player has to be exactly at the point of impact when it is the player's turn to bounce it, and then the only question is whether the player will have enough time to reach the next point of impact before the ball. With **N** paddles on a team, and **V** being the speed of the paddle, the player can move the paddle by 2**BVN** before the next impact. The ball, in the same time, will move by 2**B**$\mathbf{V_Y}$**N**, but while the paddle moves in absolute numbers, the ball moves "modulo 2**A** with wrapping", as described above.

If the distance the ball moves (modulo **A**) is smaller or equal to the distance the paddle moves, the paddle will always be there in time. The interesting case is when it is not so. In this case, there is still a chance for the paddle to be in the right place on time due to the "wrapping effect". For instance, if the ball moves by **A**+1 with each bounce, and the first bounce happens at **A** / 2, the next one will happen at **A** / 2 - 1, the next one at **A** / 2 + 2, and so on — so the initial

bounces will be pretty close by. We can calculate exactly the set of positions where the ball hitting the wall will allow the paddle to catch up, and it turns out that it is two intervals modulo 2**A**.

So now the question becomes "how long can the ball bounce without hitting a prohibited set of two intervals", which is easy to reduce to "without hitting the given interval". This is a pure number-theoretic question: we have an arithmetic sequence **I** + K**S**, modulo 2**A**, and we are interested in the first element of this sequence to fall into a given interval.

Euclid strikes again

There is a number of approaches one can take to solving this problem. We will take an approach similar to the Euclidean algorithm. First, we can shift everything modulo 2**A** so that **I** is zero, and we are dealing with the sequence K**S**. Also, we can shift the problem so that **S** ≤ **A** — if not, we can (once again) flip the problem vertically. Thus, the ball will bounce at least twice before wrapping around the edge of 2**A**.

We can calculate when is the first time the ball will pass the beginning of the forbidden interval (by integral division). If at this point the ball hits the forbidden interval, we are done. Otherwise, the ball will travel all the way to 2**A**, and then wrap around and hit the wall again at some position **P** smaller than **S**. Notice that in this case the interval obviously is shorter than **S**.

Now, the crucial question is what **P** is. It's relatively easy to calculate for what values of **P** will the next iteration land in the interval (if the interval is [k**S** + a, k**S** + b] for some k, a, b, then the interesting set of values of **P** is [a, b]). If **P** happens to be in this interval, we can again calculate the answer fast. If not, however, we will do another cycle, and then hit the wall (after cycling) at the point 2**P**, mod **S**. Notice that this is very similar to the original problem! We operate modulo **S**, we increment by **P** with each iteration, and we are interested in when we enter the interval [a,b].

Thus, we can apply recursion to learn after how many cycles will we finally be in a position to fall into the original interval we were interested in. So we make that many full cycles, and then we finish with a part of the last cycle to finally hit the interval.

Complexity analysis

All the numbers we will be operating on will have at most **D** = 200 digits in the large dataset, so operations on them will take at most $O(\mathbf{D}^2 \log \mathbf{D})$ time (assuming a quadratic multiplication implementation and a binary-search implementation of division). Each recursion step involves a constant number of arithmetic operations plus a recursive call, for which the number **A** (the modulo) decreases at least twice. This means we make at most O(**D**) recursion steps — so the whole algorithm will run in $O(\mathbf{D}^3 \log \mathbf{D})$ time, fast enough allowing even some room for inefficiency.