Problem 2: Rock Paper Strategy!
0+10+12+14=36 Points

Problem ID: `rps`
Rank: `0+1+2+3`

Introduction

Problem Summary

Play rock paper scissors against a bot whose source code has been given to you. In each round, you output your move first before reading in the bot move, but you can assume the bot does the same and you're basically playing simultaneously.

Each round, the bot first uses logic depending on the last two moves played by both players to decide what move to play. If that fails, it falls back to letting a pseudorandom number generator decide instead.

You can find the original problem statement here:
https://calico.berkeley.edu/files/calsuco-22/contest/rps/rps.pdf

Big Ideas

While the main test set can be solved with randomization or a simple pattern, the first bonus required some logical reverse engineering, and the second bonus required some number theory to crack a linear congruential generator.

Linear congruential generators, or LCGs are a very popular type of pseudorandom number generator! They're simple, fast, and the default random algorithm used by many languages. But the quality of their randomness can sometimes be poor, which leads to them being exploited like in this problem.

If there's one thing you should take away from this problem, never use an LCG for cryptographic or security purposes.

Main Test Set

To pass this test set, we need to achieve a win rate of 30%. This can actually be done using a strategy that doesn't even consider the bot's moves!

Roll the Dice *Solutions available in Java, Python*

The game theory optimal strategy for rock paper scissors is to always play a random move with equally likely probability. This can be done by picking a random number between 0, 1, and 2 using `random.randrange` in Python, `java.util.Random::nextInt` in Java, or `std::rand()` and modding by 3 in C++. Then we decide to play `R`, `P`, or `S` depending if the value is `0`, `1`, or `2`. It's easy to see that this yields an expected $\frac{1}{3}$ win rate, and when run 10^5 times, it is almost certain to exceed 30%. Although there is no better strategy in general, it's possible to do much better if we instead exploit the bot's strategy specifically.

Finding a Pattern *Solutions available in C++, Java*

Notice that the bot's strategy will always first check if our last two moves are the same. If so, then it will play the move that beats our last move. We can exploit this logic by designing a pattern that tricks the bot so that playing the move that beats our last move is always a losing move! For example, we can play `RRSSPPRRSSPPRR...` because after seeing us play `R` twice, the bot will play `P`, only to be defeated by our `S` next move. Then, after seeing us play `S` twice, the bot will play `R`, only to be defeated by our `P` next move, and so on. This guarantees we at least win every other game, yielding a win rate of ≥50% which exceeds 30%.

There are also other simple patterns that can achieve a 30% win rate or better without even considering the bot's actual moves! We challenge you to try finding some of these on your own.

Bonus Test Set 1

To pass this test set, we need to achieve a win rate of 60%. This can be done by exploiting all aspects of the logic in the bot's main function.

Counter Logic Gaming *Solutions available in C++, Python*

Since the logic depends on our previous two moves and the bot's previous two moves, it'll be useful to keep track of them. We can do this by storing all our moves and all the bot's moves into an array, and keeping track of the current round number.

Now, for every round, we can simulate the bot's logic using the previous moves we've stored. If our last two moves are the same, we can play the move that beats the move that beats our last move. Else, if the bot's last two moves are the same, we can play the bot's last move (since the bot's last move beats the move that beats the move that beats the bot's last move). Else, we can default to playing a random move as well.

The calculation to find the expected win rate is a bit involved so we won't be covering it here. It can be solved by setting up a system of equations or using a Markov chain. However, by running it against the practice test set, we see that this achieves a win rate of around 62%, which exceeds 60%.

Forcing the Hand *Solutions available in C++, Java, Python*

Although it wasn't necessary to pass the test set, we can achieve a slightly better win rate by instead defaulting to playing the last move we've played instead of a random move. This is no better for us on that turn since the opponent's move is random, but it guarantees that the next move played by the opponent is exploitable since our last two moves are the same. This brings our win rate up to about 72%.

Bonus Test Set 2

To pass this test set, we not only need to exploit all of the bot's logic, but also crack the linear congruential generator it uses when it decides to play a "random" move.

A Trit of Number Theory *Solutions available in C++, Python*

The bot's `random_move()` function works by taking X, effectively converting it into base 3, then using the digits to play the next 19 random moves in big endian. When all 19 moves are used up, it calculates the next X value using `X = (A * X + C) % M`.

If only we can figure out the values of X, A, and C, then we would be able to predict the exact "random" move performed by the bot by simulating its source code ourselves. However, the bot retrieves X, A, and C from random.org during runtime, so we can't know beforehand.

Every time the bot plays 19 random moves, we can reverse the base 3 digits to recover a prior X value. We can tell if a move is random by tracking its logic like in the previous section.

Notice that calculating X is performed under a Galois field: the set of integers mod M, as $M = 10^9 + 7$ is prime. After finding just 3 X values using 57 random moves, we can use modular arithmetic to solve all future values of X, A, and C. Call these values X_0, X_1, and X_2. We know:

$$X_1 \equiv AX_0 + C \bmod M$$

$$X_2 \equiv AX_1 + C \bmod M$$

Subtracting these equations, we get:

$$X_2 - X_1 \equiv AX_1 + C - AX_0 - C \bmod M$$

After canceling the Cs out, we collect the As with the distributive property. We can now solve for A by taking the modular inverse of (X_1 - X_0) under M using the extended Euclidean algorithm or using binary exponentiation with Fermat's little theorem and multiplying by it on both sides:

$$X_2 - X_1 \equiv A(X_1 - X_0) \bmod M$$

$$A \equiv (X_2 - X_1)(X_1 - X_0)^{-1} \bmod M$$

Knowing A, we can substitute it into any of our original equations to solve for C. We now have all we need to predict and counter every future move made by the bot with perfect accuracy!

Although we cannot always win on the first 57 random moves, every move after that is guaranteed. Thus, our win rate over 10^5 games is >99.943%, which exceeds the required 90%.