## --- Day 18: Duet ---

You discover a tablet containing some strange assembly code labeled simply "Duet". Rather than bother the sound card with it, you decide to run the code yourself. Unfortunately, you don't see any documentation, so you're left to figure out what the instructions mean on your own.

It seems like the assembly is meant to operate on a set of registers that are each named with a single letter and that can each hold a single integer. You suppose each register should start with a value of `0`.

There aren't that many instructions, so it shouldn't be hard to figure out what they do. Here's what you determine:

- `snd X` plays a sound with a frequency equal to the value of `X`.
- `set X Y` sets register `X` to the value of `Y`.
- `add X Y` increases register `X` by the value of `Y`.
- `mul X Y` sets register `X` to the result of multiplying the value contained in register `X` by the value of `Y`.
- `mod X Y` sets register `X` to the remainder of dividing the value contained in register `X` by the value of `Y` (that is, it sets `X` to the result of `X` modulo `Y`).
- `rcv X` recovers the frequency of the last sound played, but only when the value of `X` is not zero. (If it is zero, the command does nothing.)
- `jgz X Y` jumps with an offset of the value of `Y`, but only if the value of `X` is greater than zero. (An offset of `2` skips the next instruction, an offset of `-1` jumps to the previous instruction, and so on.)

Many of the instructions can take either a register (a single letter) or a number. The value of a register is the integer it contains; the value of a number is that number.

After each jump instruction, the program continues with the instruction to which the jump jumped. After any other instruction, the program continues with the next instruction. Continuing (or jumping) off either end of the program terminates it.

For example:

```
set a 1
add a 2
mul a a
mod a 5
snd a
set a 0
rcv a
jgz a -1
set a 1
jgz a -2
```

- The first four instructions set `a` to `1`, add `2` to it, square it, and then set it to itself modulo `5`, resulting in a value of `4`.
- Then, a sound with frequency `4` (the value of `a`) is played.
- After that, `a` is set to `0`, causing the subsequent `rcv` and `jgz` instructions to both be skipped (`rcv` because `a` is `0`, and `jgz` because `a` is not greater than `0`).
- Finally, `a` is set to `1`, causing the next `jgz` instruction to activate, jumping back two instructions to another jump, which jumps again to the `rcv`, which ultimately triggers the recover operation.

At the time the recover operation is executed, the frequency of the last sound played is `4`.

What is the value of the recovered frequency (the value of the most recently played sound) the first time a `rcv` instruction is executed with a non-zero value?

Our sponsors he
make Advent of
Code possible:

Cheppers -
xor(Pz0pQUI7Ch
cmER8YDAEYAh4L
GwEP,
↑↑↓↓←→←→BA)

Your puzzle answer was `3188`.

## --- Part Two ---

As you congratulate yourself for a job well done, you notice that the documentation has been on the back of the tablet this entire time. While you actually got most of the instructions correct, there are a few key differences. This assembly code isn't about sound at all - it's meant to be run twice at the same time.

Each running copy of the program has its own set of registers and follows the code independently - in fact, the programs don't even necessarily run at the same speed. To coordinate, they use the send (`snd`) and receive (`rcv`) instructions:

- `snd X` sends the value of `X` to the other program. These values wait in a queue until that program is ready to receive them. Each program has its own message queue, so a program can never receive a message it sent.
- `rcv X` receives the next value and stores it in register `X`. If no values are in the queue, the program waits for a value to be sent to it. Programs do not continue to the next instruction until they have received a value. Values are received in the order they are sent.

Each program also has its own program ID (one `0` and the other `1`); the register `p` should begin with this value.

For example:

```
snd 1
snd 2
snd p
rcv a
rcv b
rcv c
rcv d
```

Both programs begin by sending three values to the other. Program `0` sends `1, 2, 0`; program `1` sends `1, 2, 1`. Then, each program receives a value (both `1`) and stores it in `a`, receives another value (both `2`) and stores it in `b`, and then each receives the program ID of the other program (program `0` receives `1`; program `1` receives `0`) and stores it in `c`. Each program now sees a different value in its own copy of register `c`.

Finally, both programs try to `rcv` a fourth time, but no data is waiting for either of them, and they reach a deadlock. When this happens, both programs terminate.

It should be noted that it would be equally valid for the programs to run at different speeds; for example, program `0` might have sent all three values and then stopped at the first `rcv` before program `1` executed even its first instruction.

Once both of your programs have terminated (regardless of what caused them to do so), how many times did program `1` send a value?

Your puzzle answer was `7112`.

Both parts of this puzzle are complete! They provide two gold stars: **

At this point, you should return to your advent calendar and try another puzzle.

If you still want to see it, you can get your puzzle input.

You can also [Share] this puzzle.