--- Day 19: Go With The Flow ---

With the Elves well on their way constructing the North Pole base, you turn
your attention back to understanding the inner workings of programming the
device.

You can't help but notice that the device's opcodes don't contain any flow
control like jump instructions. The device's manual goes on to explain:

"In programs where flow control is required, the instruction pointer can be
bound to a register so that it can be manipulated directly. This way, setr/
seti can function as absolute jumps, addr/addi can function as relative
jumps, and other opcodes can cause truly fascinating effects."

This mechanism is achieved through a declaration like #ip 1, which would
modify register 1 so that accesses to it let the program indirectly access
the instruction pointer itself. To compensate for this kind of binding,
there are now six registers (numbered 0 through 5); the five not bound to
the instruction pointer behave as normal. Otherwise, the same rules apply
as the last time you worked with this device.

When the instruction pointer is bound to a register, its value is written
to that register just before each instruction is executed, and the value of
that register is written back to the instruction pointer immediately after
each instruction finishes execution. Afterward, move to the next
instruction by adding one to the instruction pointer, even if the value in
the instruction pointer was just updated by an instruction. (Because of
this, instructions must effectively set the instruction pointer to the
instruction before the one they want executed next.)

The instruction pointer is 0 during the first instruction, 1 during the
second, and so on. If the instruction pointer ever causes the device to
attempt to load an instruction outside the instructions defined in the
program, the program instead immediately halts. The instruction pointer
starts at 0.

It turns out that this new information is already proving useful: the CPU
in the device is not very powerful, and a background process is occupying
most of its time. You dump the background process' declarations and
instructions to a file (your puzzle input), making sure to use the names of
the opcodes rather than the numbers.

For example, suppose you have the following program:

```
#ip 0
seti 5 0 1
seti 6 0 2
addi 0 1 0
addr 1 2 3
setr 1 0 0
seti 8 0 4
seti 9 0 5
```

When executed, the following instructions are executed. Each line contains
the value of the instruction pointer at the time the instruction started,
the values of the six registers before executing the instructions (in
square brackets), the instruction itself, and the values of the six
registers after executing the instruction (also in square brackets).

```
ip=0 [0, 0, 0, 0, 0, 0] seti 5 0 1 [0, 5, 0, 0, 0, 0]
ip=1 [1, 5, 0, 0, 0, 0] seti 6 0 2 [1, 5, 6, 0, 0, 0]
ip=2 [2, 5, 6, 0, 0, 0] addi 0 1 0 [3, 5, 6, 0, 0, 0]
ip=4 [4, 5, 6, 0, 0, 0] setr 1 0 0 [5, 5, 6, 0, 0, 0]
ip=6 [6, 5, 6, 0, 0, 0] seti 9 0 5 [6, 5, 6, 0, 0, 9]
```

In detail, when running this program, the following events occur:

  - The first line (#ip 0) indicates that the instruction pointer should
    be bound to register 0 in this program. This is not an instruction,
    and so the value of the instruction pointer does not change during the
    processing of this line.
  - The instruction pointer contains 0, and so the first instruction is
    executed (seti 5 0 1). It updates register 0 to the current
    instruction pointer value (0), sets register 1 to 5, sets the
    instruction pointer to the value of register 0 (which has no effect,
    as the instruction did not modify register 0), and then adds one to
    the instruction pointer.
  - The instruction pointer contains 1, and so the second instruction,
    seti 6 0 2, is executed. This is very similar to the instruction
    before it: 6 is stored in register 2, and the instruction pointer is
    left with the value 2.
  - The instruction pointer is 2, which points at the instruction
    addi 0 1 0. This is like a relative jump: the value of the instruction

pointer, `2`, is loaded into register `0`. Then, `addi` finds the result of adding the value in register `0` and the value `1`, storing the result, `3`, back in register `0`. Register `0` is then copied back to the instruction pointer, which will cause it to end up `1` larger than it would have otherwise and skip the next instruction (`addr 1 2 3`) entirely. Finally, `1` is added to the instruction pointer.
 - The instruction pointer is `4`, so the instruction `setr 1 0 0` is run. This is like an **absolute jump**: it copies the value contained in register `1`, `5`, into register `0`, which causes it to end up in the instruction pointer. The instruction pointer is then incremented, leaving it at `6`.
 - The instruction pointer is `6`, so the instruction `seti 9 0 5` stores `9` into register `5`. The instruction pointer is incremented, causing it to point outside the program, and so the program ends.

What value is left in register `0` when the background process halts?

To begin, get your puzzle input.

Answer: [ ] [Submit]

You can also [Share] this puzzle.