--- Day 1: The Tyranny of the Rocket Equation ---

Santa has become stranded at the edge of the Solar System while delivering presents to other planets! To accurately calculate his position in space, safely align his warp drive, and return to Earth in time to save Christmas, he needs you to bring him measurements from *fifty stars*.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the Advent calendar; the second puzzle is unlocked when you complete the first. Each puzzle grants *one star*. Good luck!

The Elves quickly load you into a spacecraft and prepare to launch.

At the first Go / No Go poll, every Elf is Go until the Fuel Counter-Upper. They haven't determined the amount of fuel required yet.

Fuel required to launch a given *module* is based on its *mass*. Specifically, to find the fuel required for a module, take its mass, divide by three, round down, and subtract 2.

For example:

- For a mass of `12`, divide by 3 and round down to get `4`, then subtract 2 to get `2`.
- For a mass of `14`, dividing by 3 and rounding down still yields `4`, so the fuel required is also `2`.
- For a mass of `1969`, the fuel required is `654`.
- For a mass of `100756`, the fuel required is `33583`.

The Fuel Counter-Upper needs to know the total fuel requirement. To find it, individually calculate the fuel needed for the mass of each module (your puzzle input), then add together all the fuel values.

What is the sum of the fuel requirements for all of the modules on your spacecraft?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 2: 1202 Program Alarm ---

On the way to your gravity assist around the Moon, your ship computer beeps angrily about a "1202 program alarm". On the radio, an Elf is already explaining how to handle the situation: "Don't worry, that's perfectly norma--" The ship computer bursts into flames.

You notify the Elves that the computer's magic smoke seems to have escaped. "That computer ran *Intcode* programs like the gravity assist program it was working on; surely there are enough spare parts up there to build a new Intcode computer!"

An Intcode program is a list of integers separated by commas (like `1,0,0,3,99`). To run one, start by looking at the first integer (called position `0`). Here, you will find an *opcode* - either `1`, `2`, or `99`. The opcode indicates what to do; for example, `99` means that the program is finished and should immediately halt. Encountering an unknown opcode means something went wrong.

Opcode `1` *adds* together numbers read from two positions and stores the result in a third position. The three integers *immediately after* the opcode tell you these three positions - the first two indicate the *positions* from which you should read the input values, and the third indicates the *position* at which the output should be stored.

For example, if your Intcode computer encounters `1,10,20,30`, it should read the values at positions `10` and `20`, add those values, and then overwrite the value at position `30` with their sum.

Opcode `2` works exactly like opcode `1`, except it *multiplies* the two inputs instead of adding them. Again, the three integers after the opcode indicate *where* the inputs and outputs are, not their values.

Once you're done processing an opcode, *move to the next one* by stepping forward `4` positions.

For example, suppose you have the following program:

```
1,9,10,3,2,3,11,0,99,30,40,50
```

For the purposes of illustration, here is the same program split into multiple lines:

```
1,9,10,3,
2,3,11,0,
99,
30,40,50
```

The first four integers, `1,9,10,3`, are at positions `0`, `1`, `2`, and `3`. Together, they represent the first opcode (`1`, addition), the positions of the two inputs (`9` and `10`), and the position of the output (`3`). To handle this opcode, you first need to get the values at the input positions: position `9` contains `30`, and position `10` contains `40`. *Add* these numbers together to get `70`. Then, store this value at the output position; here, the output position (`3`) is *at* position `3`, so it overwrites itself. Afterward, the program looks like this:

```
1,9,10,70,
2,3,11,0,
99,
30,40,50
```

Step forward `4` positions to reach the next opcode, `2`. This opcode works just like the previous, but it multiplies instead of adding. The inputs are at positions

3 and 11; these positions contain 70 and 50 respectively. Multiplying these produces 3500; this is stored at position 0:

```
3500,9,10,70,
2,3,11,0,
99,
30,40,50
```

Stepping forward 4 more positions arrives at opcode 99, halting the program.

Here are the initial and final states of a few more small programs:

- 1,0,0,0,99 becomes 2,0,0,0,99 (1 + 1 = 2).
- 2,3,0,3,99 becomes 2,3,0,6,99 (3 * 2 = 6).
- 2,4,4,5,99,0 becomes 2,4,4,5,99,9801 (99 * 99 = 9801).
- 1,1,1,4,99,5,6,0,99 becomes 30,1,1,4,2,5,6,0,99.

Once you have a working computer, the first step is to restore the gravity assist program (your puzzle input) to the "1202 program alarm" state it had just before the last computer caught fire. To do this, *before running the program*, replace position 1 with the value 12 and replace position 2 with the value 2. *What value is left at position 0* after the program halts?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 3: Crossed Wires ---

The gravity assist was successful, and you're well on your way to the Venus refuelling station. During the rush back on Earth, the fuel management system wasn't completely installed, so that's next on the priority list.

Opening the front panel reveals a jumble of wires. Specifically, *two wires* are connected to a central port and extend outward on a grid. You trace the path each wire takes as it leaves the central port, one wire per line of text (your puzzle input).

The wires twist and turn, but the two wires occasionally cross paths. To fix the circuit, you need to *find the intersection point closest to the central port*. Because the wires are on a grid, use the Manhattan distance for this measurement. While the wires do technically cross right at the central port where they both start, this point does not count, nor does a wire count as crossing with itself.

For example, if the first wire's path is R8,U5,L5,D3, then starting from the central port (o), it goes right 8, up 5, left 5, and finally down 3:

```
...........
...........
...........
....+----+.
```

```
....|....|.
....|....|.
....|....|.
.........|.
.o-------+.
...........
```

Then, if the second wire's path is `U7,R6,D4,L4`, it goes up `7`, right `6`, down `4`, and left `4`:

```
...........
.+-----+...
.|.....|...
.|..+--X-+.
.|..|..|.|.
.|.-X--+.|.
.|..|....|.
.|.......|.
.o-------+.
...........
```

These wires cross at two locations (marked `X`), but the lower-left one is closer to the central port: its distance is `3 + 3 = 6`.

Here are a few more examples:

- `R75,D30,R83,U83,L12,D49,R71,U7,L72U62,R66,U55,R34,D71,R55,D58,R83` = distance `159`
- `R98,U47,R26,D63,R33,U87,L62,D20,R33,U53,R51U98,R91,D20,R16,D67,R40,U7,R15,U6,R7` = distance `135`

What is the Manhattan distance from the central port to the closest intersection?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 4: Secure Container ---

You arrive at the Venus fuel depot only to discover it's protected by a password. The Elves had written the password on a sticky note, but someone threw it out.

However, they do remember a few key facts about the password:

- It is a six-digit number.
- The value is within the range given in your puzzle input.
- Two adjacent digits are the same (like `22` in `122345`).
- Going from left to right, the digits *never decrease*; they only ever increase or stay the same (like `111123` or `135679`).

Other than the range rule, the following are true:

- `111111` meets these criteria (double `11`, never decreases).
- `223450` does not meet these criteria (decreasing pair of digits `50`).
- `123789` does not meet these criteria (no double).

How many different passwords within the range given in your puzzle input meet these criteria?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 5: Sunny with a Chance of Asteroids ---

You're starting to sweat as the ship makes its way toward Mercury. The Elves suggest that you get the air conditioner working by upgrading your ship computer to support the Thermal Environment Supervision Terminal.

The Thermal Environment Supervision Terminal (TEST) starts by running a *diagnostic program* (your puzzle input). The TEST diagnostic program will run on your existing Intcode computer after a few modifications:

First, you'll need to add *two new instructions*:

- Opcode `3` takes a single integer as *input* and saves it to the position given by its only parameter. For example, the instruction `3,50` would take an input value and store it at address `50`.
- Opcode `4` *outputs* the value of its only parameter. For example, the instruction `4,50` would output the value at address `50`.

Programs that use these instructions will come with documentation that explains what should be connected to the input and output. The program `3,0,4,0,99` outputs whatever it gets as input, then halts.

Second, you'll need to add support for *parameter modes*:

Each parameter of an instruction is handled based on its parameter mode. Right now, your ship computer already understands parameter mode `0`, *position mode*, which causes the parameter to be interpreted as a *position* - if the parameter is `50`, its value is *the value stored at address `50` in memory*. Until now, all parameters have been in position mode.

Now, your ship computer will also need to handle parameters in mode `1`, *immediate mode*. In immediate mode, a parameter is interpreted as a *value* - if the parameter is `50`, its value is simply *`50`*.

Parameter modes are stored in the same value as the instruction's opcode. The opcode is a two-digit number based only on the ones and tens digit of the value, that is, the opcode is the rightmost two digits of the first value in an instruction. Parameter modes are single digits, one per parameter, read right-to-left from the opcode: the first parameter's mode is in the hundreds digit, the second

parameter's mode is in the thousands digit, the third parameter's mode is in the ten-thousands digit, and so on. Any missing modes are `0`.

For example, consider the program `1002,4,3,4,33`.

The first instruction, `1002,4,3,4`, is a *multiply* instruction - the rightmost two digits of the first value, `02`, indicate opcode `2`, multiplication. Then, going right to left, the parameter modes are `0` (hundreds digit), `1` (thousands digit), and `0` (ten-thousands digit, not present and therefore zero):

```
ABCDE
 1002

DE - two-digit opcode,      02 == opcode 2
 C - mode of 1st parameter,  0 == position mode
 B - mode of 2nd parameter,  1 == immediate mode
 A - mode of 3rd parameter,  0 == position mode,
                                  omitted due to being a leading zero
```

This instruction multiplies its first two parameters. The first parameter, `4` in position mode, works like it did before - its value is the value stored at address `4` (`33`). The second parameter, `3` in immediate mode, simply has value `3`. The result of this operation, `33 * 3 = 99`, is written according to the third parameter, `4` in position mode, which also works like it did before - `99` is written to address `4`.

Parameters that an instruction writes to will *never be in immediate mode*.

Finally, some notes:

- It is important to remember that the instruction pointer should increase by *the number of values in the instruction* after the instruction finishes. Because of the new instructions, this amount is no longer always `4`.
- Integers can be negative: `1101,100,-1,4,0` is a valid program (find `100 + -1`, store the result in position `4`).

The TEST diagnostic program will start by requesting from the user the ID of the system to test by running an *input* instruction - provide it `1`, the ID for the ship's air conditioner unit.

It will then perform a series of diagnostic tests confirming that various parts of the Intcode computer, like parameter modes, function correctly. For each test, it will run an *output* instruction indicating how far the result of the test was from the expected value, where `0` means the test was successful. Non-zero outputs mean that a function is not working correctly; check the instructions that were run before the output instruction to see which one failed.

Finally, the program will output a *diagnostic code* and immediately halt. This final output isn't an error; an output followed immediately by a halt means the program finished. If all outputs were zero except the diagnostic code, the diagnostic program ran successfully.

After providing `1` to the only input instruction and passing all the tests, *what diagnostic code does the program produce?*

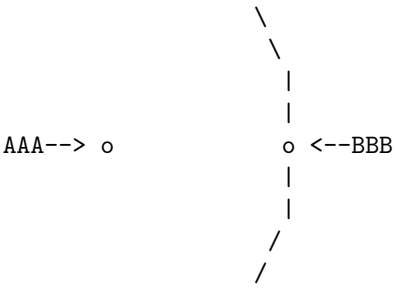
To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 6: Universal Orbit Map ---

You've landed at the Universal Orbit Map facility on Mercury. Because navigation in space often involves transferring between orbits, the orbit maps here are useful for finding efficient routes between, for example, you and Santa. You download a map of the local orbits (your puzzle input).

Except for the universal Center of Mass (`COM`), every object in space is in orbit around exactly one other object. An orbit looks roughly like this:



In this diagram, the object `BBB` is in orbit around `AAA`. The path that `BBB` takes around `AAA` (drawn with lines) is only partly shown. In the map data, this orbital relationship is written `AAA)BBB`, which means "`BBB` is in orbit around `AAA`".

Before you use your map data to plot a course, you need to make sure it wasn't corrupted during the download. To verify maps, the Universal Orbit Map facility uses *orbit count checksums* - the total number of *direct orbits* (like the one shown above) and *indirect orbits*.

Whenever `A` orbits `B` and `B` orbits `C`, then `A` *indirectly orbits* `C`. This chain can be any number of objects long: if `A` orbits `B`, `B` orbits `C`, and `C` orbits `D`, then `A` indirectly orbits `D`.

For example, suppose you have the following map:

```
COM)B
B)C
C)D
D)E
E)F
B)G
G)H
D)I
```

```
E)J
J)K
K)L
```

Visually, the above map of orbits looks like this:

```
        G - H       J - K - L
       /           /
COM - B - C - D - E - F
               \
                I
```

In this visual representation, when two objects are connected by a line, the one on the right directly orbits the one on the left.

Here, we can count the total number of orbits as follows:

- `D` directly orbits `C` and indirectly orbits `B` and `COM`, a total of `3` orbits.
- `L` directly orbits `K` and indirectly orbits `J`, `E`, `D`, `C`, `B`, and `COM`, a total of `7` orbits.
- `COM` orbits nothing.

The total number of direct and indirect orbits in this example is `42`.

What is the total number of direct and indirect orbits in your map data?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 7: Amplification Circuit ---

Based on the navigational maps, you're going to need to send more power to your ship's thrusters to reach Santa in time. To do this, you'll need to configure a series of amplifiers already installed on the ship.

There are five amplifiers connected in series; each one receives an input signal and produces an output signal. They are connected such that the first amplifier's output leads to the second amplifier's input, the second amplifier's output leads to the third amplifier's input, and so on. The first amplifier's input value is `0`, and the last amplifier's output leads to your ship's thrusters.

```
    O-------O  O-------O  O-------O  O-------O  O-------O
0 ->| Amp A |->| Amp B |->| Amp C |->| Amp D |->| Amp E |-> (to thrusters)
    O-------O  O-------O  O-------O  O-------O  O-------O
```

The Elves have sent you some *Amplifier Controller Software* (your puzzle input), a program that should run on your existing Intcode computer. Each amplifier will need to run a copy of the program.

When a copy of the program starts running on an amplifier, it will first use an input instruction to ask the amplifier for its current *phase setting* (an integer

from 0 to 4). Each phase setting is used *exactly once*, but the Elves can't remember which amplifier needs which phase setting.

The program will then call another input instruction to get the amplifier's input signal, compute the correct output signal, and supply it back to the amplifier with an output instruction. (If the amplifier has not yet received an input signal, it waits until one arrives.)

Your job is to *find the largest output signal that can be sent to the thrusters* by trying every possible combination of phase settings on the amplifiers. Make sure that memory is not shared or reused between copies of the program.

For example, suppose you want to try the phase setting sequence 3,1,2,4,0, which would mean setting amplifier A to phase setting 3, amplifier B to setting 1, C to 2, D to 4, and E to 0. Then, you could determine the output signal that gets sent from amplifier E to the thrusters with the following steps:

- Start the copy of the amplifier controller software that will run on amplifier A. At its first input instruction, provide it the amplifier's phase setting, 3. At its second input instruction, provide it the input signal, 0. After some calculations, it will use an output instruction to indicate the amplifier's output signal.
- Start the software for amplifier B. Provide it the phase setting (1) and then whatever output signal was produced from amplifier A. It will then produce a new output signal destined for amplifier C.
- Start the software for amplifier C, provide the phase setting (2) and the value from amplifier B, then collect its output signal.
- Run amplifier D's software, provide the phase setting (4) and input value, and collect its output signal.
- Run amplifier E's software, provide the phase setting (0) and input value, and collect its output signal.

The final output signal from amplifier E would be sent to the thrusters. However, this phase setting sequence may not have been the best one; another sequence might have sent a higher signal to the thrusters.

Here are some example programs:

- Max thruster signal *43210* (from phase setting sequence 4,3,2,1,0):

  ```
  3,15,3,16,1002,16,10,16,1,16,15,15,4,15,99,0,0
  ```

- Max thruster signal *54321* (from phase setting sequence 0,1,2,3,4):

  ```
  3,23,3,24,1002,24,10,24,1002,23,-1,23,
  101,5,23,23,1,24,23,23,4,23,99,0,0
  ```

- Max thruster signal *65210* (from phase setting sequence 1,0,4,3,2):

  ```
  3,31,3,32,1002,32,10,32,1001,31,-2,31,1007,31,0,33,
  1002,33,7,33,1,33,31,31,1,32,31,31,4,31,99,0,0,0
  ```

Try every combination of phase settings on the amplifiers. *What is the highest signal that can be sent to the thrusters?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 8: Space Image Format ---

The Elves' spirits are lifted when they realize you have an opportunity to reboot one of their Mars rovers, and so they are curious if you would spend a brief sojourn on Mars. You land your ship near the rover.

When you reach the rover, you discover that it's already in the process of rebooting! It's just waiting for someone to enter a BIOS password. The Elf responsible for the rover takes a picture of the password (your puzzle input) and sends it to you via the Digital Sending Network.

Unfortunately, images sent via the Digital Sending Network aren't encoded with any normal encoding; instead, they're encoded in a special Space Image Format. None of the Elves seem to remember why this is the case. They send you the instructions to decode it.

Images are sent as a series of digits that each represent the color of a single pixel. The digits fill each row of the image left-to-right, then move downward to the next row, filling rows top-to-bottom until every pixel of the image is filled.

Each image actually consists of a series of identically-sized *layers* that are filled in this way. So, the first digit corresponds to the top-left pixel of the first layer, the second digit corresponds to the pixel to the right of that on the same layer, and so on until the last digit, which corresponds to the bottom-right pixel of the last layer.

For example, given an image `3` pixels wide and `2` pixels tall, the image data `123456789012` corresponds to the following image layers:

```
Layer 1: 123
         456

Layer 2: 789
         012
```

The image you received is *`25` pixels wide and `6` pixels tall.*

To make sure the image wasn't corrupted during transmission, the Elves would like you to find the layer that contains the *fewest `0` digits*. On that layer, what is *the number of `1` digits multiplied by the number of `2` digits?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 9: Sensor Boost ---

You've just said goodbye to the rebooted rover and left Mars when you receive a faint distress signal coming from the asteroid belt. It must be the Ceres monitoring station!

In order to lock on to the signal, you'll need to boost your sensors. The Elves send up the latest *BOOST* program - Basic Operation Of System Test.

While BOOST (your puzzle input) is capable of boosting your sensors, for tenuous safety reasons, it refuses to do so until the computer it runs on passes some checks to demonstrate it is a *complete Intcode computer*.

Your existing Intcode computer is missing one key feature: it needs support for parameters in *relative mode*.

Parameters in mode `2`, *relative mode*, behave very similarly to parameters in *position mode*: the parameter is interpreted as a position. Like position mode, parameters in relative mode can be read from or written to.

The important difference is that relative mode parameters don't count from address `0`. Instead, they count from a value called the *relative base*. The *relative base* starts at `0`.

The address a relative mode parameter refers to is itself *plus* the current *relative base*. When the relative base is `0`, relative mode parameters and position mode parameters with the same value refer to the same address.

For example, given a relative base of `50`, a relative mode parameter of `-7` refers to memory address `50 + -7 = 43`.

The relative base is modified with the *relative base offset* instruction:

- Opcode `9` *adjusts the relative base* by the value of its only parameter. The relative base increases (or decreases, if the value is negative) by the value of the parameter.

For example, if the relative base is `2000`, then after the instruction `109,19`, the relative base would be `2019`. If the next instruction were `204,-34`, then the value at address `1985` would be output.

Your Intcode computer will also need a few other capabilities:

- The computer's available memory should be much larger than the initial program. Memory beyond the initial program starts with the value `0` and can be read or written like any other memory. (It is invalid to try to access memory at a negative address, though.)
- The computer should have support for large numbers. Some instructions near the beginning of the BOOST program will verify this capability.

Here are some example programs that use these features:

- `109,1,204,-1,1001,100,1,100,1008,100,16,101,1006,101,0,99` takes no input and produces a copy of itself as output.
- `1102,34915192,34915192,7,4,7,99,0` should output a 16-digit number.
- `104,1125899906842624,99` should output the large number in the middle.

The BOOST program will ask for a single input; run it in test mode by providing it the value `1`. It will perform a series of checks on each opcode, output any opcodes (and the associated parameter modes) that seem to be functioning incorrectly, and finally output a BOOST keycode.

Once your Intcode computer is fully functional, the BOOST program should report no malfunctioning opcodes when run in test mode; it should only output a single value, the BOOST keycode. *What BOOST keycode does it produce?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 10: Monitoring Station ---

You fly into the asteroid belt and reach the Ceres monitoring station. The Elves here have an emergency: they're having trouble tracking all of the asteroids and can't be sure they're safe.

The Elves would like to build a new monitoring station in a nearby area of space; they hand you a map of all of the asteroids in that region (your puzzle input).

The map indicates whether each position is empty (`.`) or contains an asteroid (`#`). The asteroids are much smaller than they appear on the map, and every asteroid is exactly in the center of its marked position. The asteroids can be described with `X,Y` coordinates where `X` is the distance from the left edge and `Y` is the distance from the top edge (so the top-left corner is `0,0` and the position immediately to its right is `1,0`).

Your job is to figure out which asteroid would be the best place to build a *new monitoring station*. A monitoring station can *detect* any asteroid to which it has *direct line of sight* - that is, there cannot be another asteroid *exactly* between them. This line of sight can be at any angle, not just lines aligned to the grid or diagonally. The *best* location is the asteroid that can *detect* the largest number of other asteroids.

For example, consider the following map:

```
.#..#
.....
#####
....#
...##
```

The best location for a new monitoring station on this map is the highlighted asteroid at `3,4` because it can detect `8` asteroids, more than any other location.

(The only asteroid it cannot detect is the one at `1,0`; its view of this asteroid is blocked by the asteroid at `2,2`.) All other asteroids are worse locations; they can detect `7` or fewer other asteroids. Here is the number of other asteroids a monitoring station on each asteroid could detect:

```
.7..7
.....
67775
....7
...87
```

Here is an asteroid (`#`) and some examples of the ways its line of sight might be blocked. If there were another asteroid at the location of a capital letter, the locations marked with the corresponding lowercase letter would be blocked and could not be detected:

```
#.........
...A......
...B..a...
.EDCG....a
..F.c.b...
.....c....
..efd.c.gb
.......c..
....f...c.
...e..d..c
```

Here are some larger examples:

- Best is `5,8` with `33` other asteroids detected:

  ```
  ......#.#.
  #..#.#....
  ..#######.
  .#.#.###..
  .#..#.....
  ..#....#.#
  #..#....#.
  .##.#..###
  ##...#..#.
  .#....####
  ```

- Best is `1,2` with `35` other asteroids detected:

  ```
  #.#...#.#.
  .###....#.
  .#....#...
  ##.#.#.#.#
  ....#.#.#.
  .##..###.#
  ```

```
..#...##..
..##....##
......#...
.####.###.
```

- Best is `6,3` with `41` other asteroids detected:

```
.#..#..###
####.###.#
....###.#.
..###.##.#
##.##.#.#.
....###..#
..#.#..#.#
#..#.#.###
.##...##.#
.....#.#..
```

- Best is `11,13` with `210` other asteroids detected:

```
.#..##.###...#######
##.############..##.
.#.######.########.#
.###.#######.####.#.
#####.##.#.##.###.##
..#####..#.#########
####################
#.####....###.#.#.##
##.##################
#####.##.###..####..
..#######..##.#######
####.##.####...##..#
.#####..#.######.###
##...#.###########...
#.###########.#######
.####.#.###.###.#.##
....##.##.###..#####
.#.#.############.###
#.#.#.#####.####.###
###.##.####.##.#..##
```

Find the best location for a new monitoring station. *How many other asteroids can be detected from that location?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 11: Space Police ---

On the way to Jupiter, you're pulled over by the *Space Police*.

"Attention, unmarked spacecraft! You are in violation of Space Law! All spacecraft must have a clearly visible *registration identifier*! You have 24 hours to comply or be sent to Space Jail!"

Not wanting to be sent to Space Jail, you radio back to the Elves on Earth for help. Although it takes almost three hours for their reply signal to reach you, they send instructions for how to power up the *emergency hull painting robot* and even provide a small Intcode program (your puzzle input) that will cause it to paint your ship appropriately.

There's just one problem: you don't have an emergency hull painting robot.

You'll need to build a new emergency hull painting robot. The robot needs to be able to move around on the grid of square panels on the side of your ship, detect the color of its current panel, and paint its current panel *black* or *white*. (All of the panels are currently *black*.)

The Intcode program will serve as the brain of the robot. The program uses input instructions to access the robot's camera: provide `0` if the robot is over a *black* panel or `1` if the robot is over a *white* panel. Then, the program will output two values:

- First, it will output a value indicating the *color to paint the panel* the robot is over: `0` means to paint the panel *black*, and `1` means to paint the panel *white*.
- Second, it will output a value indicating the *direction the robot should turn*: `0` means it should turn *left 90 degrees*, and `1` means it should turn *right 90 degrees*.

After the robot turns, it should always move *forward exactly one panel*. The robot starts facing *up*.

The robot will continue running for a while like this and halt when it is finished drawing. Do not restart the Intcode computer inside the robot during this process.

For example, suppose the robot is about to start running. Drawing black panels as `.`, white panels as `#`, and the robot pointing the direction it is facing (`< ^ > v`), the initial state and region near the robot looks like this:

```
.....
.....
..^..
.....
.....
```

The panel under the robot (not visible here because a `^` is shown instead) is also

black, and so any input instructions at this point should be provided `0`. Suppose the robot eventually outputs `1` (paint white) and then `0` (turn left). After taking these actions and moving forward one panel, the region now looks like this:

```
.....
.....
.<#..
.....
.....
```

Input instructions should still be provided `0`. Next, the robot might output `0` (paint black) and then `0` (turn left):

```
.....
.....
..#..
.v...
.....
```

After more outputs (`1,0`, `1,0`):

```
.....
.....
..^..
.##..
.....
```

The robot is now back where it started, but because it is now on a white panel, input instructions should be provided `1`. After several more outputs (`0,1`, `1,0`, `1,0`), the area looks like this:

```
.....
..<#.
...#.
.##..
.....
```

Before you deploy the robot, you should probably have an estimate of the area it will cover: specifically, you need to know the *number of panels it paints at least once*, regardless of color. In the example above, the robot painted ***6*** *panels* at least once. (It painted its starting panel twice, but that panel is still only counted once; it also never painted the panel it ended on.)

Build a new emergency hull painting robot and run the Intcode program on it. *How many panels does it paint at least once?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 12: The N-Body Problem ---

The space near Jupiter is not a very safe place; you need to be careful of a big distracting red spot, extreme radiation, and a whole lot of moons swirling around. You decide to start by tracking the four largest moons: *Io*, *Europa*, *Ganymede*, and *Callisto*.

After a brief scan, you calculate the *position of each moon* (your puzzle input). You just need to *simulate their motion* so you can avoid them.

Each moon has a 3-dimensional position (`x`, `y`, and `z`) and a 3-dimensional velocity. The position of each moon is given in your scan; the `x`, `y`, and `z` velocity of each moon starts at `0`.

Simulate the motion of the moons in *time steps*. Within each time step, first update the velocity of every moon by applying *gravity*. Then, once all moons' velocities have been updated, update the position of every moon by applying *velocity*. Time progresses by one step once all of the positions are updated.

To apply *gravity*, consider every *pair* of moons. On each axis (`x`, `y`, and `z`), the velocity of each moon changes by *exactly +1 or -1* to pull the moons together. For example, if Ganymede has an `x` position of `3`, and Callisto has a `x` position of `5`, then Ganymede's `x` velocity *changes by +1* (because `5 > 3`) and Callisto's `x` velocity *changes by -1* (because `3 < 5`). However, if the positions on a given axis are the same, the velocity on that axis *does not change* for that pair of moons.

Once all gravity has been applied, apply *velocity*: simply add the velocity of each moon to its own position. For example, if Europa has a position of `x=1, y=2, z=3` and a velocity of `x=-2, y=0,z=3`, then its new position would be `x=-1, y=2, z=6`. This process does not modify the velocity of any moon.

For example, suppose your scan reveals the following positions:

```
<x=-1, y=0, z=2>
<x=2, y=-10, z=-7>
<x=4, y=-8, z=8>
<x=3, y=5, z=-1>
```

Simulating the motion of these moons would produce the following:

```
After 0 steps:
pos=<x=-1, y=  0, z= 2>, vel=<x= 0, y= 0, z= 0>
pos=<x= 2, y=-10, z=-7>, vel=<x= 0, y= 0, z= 0>
pos=<x= 4, y= -8, z= 8>, vel=<x= 0, y= 0, z= 0>
pos=<x= 3, y=  5, z=-1>, vel=<x= 0, y= 0, z= 0>

After 1 step:
pos=<x= 2, y=-1, z= 1>, vel=<x= 3, y=-1, z=-1>
pos=<x= 3, y=-7, z=-4>, vel=<x= 1, y= 3, z= 3>
```

```
pos=<x= 1, y=-7, z= 5>, vel=<x=-3, y= 1, z=-3>
pos=<x= 2, y= 2, z= 0>, vel=<x=-1, y=-3, z= 1>

After 2 steps:
pos=<x= 5, y=-3, z=-1>, vel=<x= 3, y=-2, z=-2>
pos=<x= 1, y=-2, z= 2>, vel=<x=-2, y= 5, z= 6>
pos=<x= 1, y=-4, z=-1>, vel=<x= 0, y= 3, z=-6>
pos=<x= 1, y=-4, z= 2>, vel=<x=-1, y=-6, z= 2>

After 3 steps:
pos=<x= 5, y=-6, z=-1>, vel=<x= 0, y=-3, z= 0>
pos=<x= 0, y= 0, z= 6>, vel=<x=-1, y= 2, z= 4>
pos=<x= 2, y= 1, z=-5>, vel=<x= 1, y= 5, z=-4>
pos=<x= 1, y=-8, z= 2>, vel=<x= 0, y=-4, z= 0>

After 4 steps:
pos=<x= 2, y=-8, z= 0>, vel=<x=-3, y=-2, z= 1>
pos=<x= 2, y= 1, z= 7>, vel=<x= 2, y= 1, z= 1>
pos=<x= 2, y= 3, z=-6>, vel=<x= 0, y= 2, z=-1>
pos=<x= 2, y=-9, z= 1>, vel=<x= 1, y=-1, z=-1>

After 5 steps:
pos=<x=-1, y=-9, z= 2>, vel=<x=-3, y=-1, z= 2>
pos=<x= 4, y= 1, z= 5>, vel=<x= 2, y= 0, z=-2>
pos=<x= 2, y= 2, z=-4>, vel=<x= 0, y=-1, z= 2>
pos=<x= 3, y=-7, z=-1>, vel=<x= 1, y= 2, z=-2>

After 6 steps:
pos=<x=-1, y=-7, z= 3>, vel=<x= 0, y= 2, z= 1>
pos=<x= 3, y= 0, z= 0>, vel=<x=-1, y=-1, z=-5>
pos=<x= 3, y=-2, z= 1>, vel=<x= 1, y=-4, z= 5>
pos=<x= 3, y=-4, z=-2>, vel=<x= 0, y= 3, z=-1>

After 7 steps:
pos=<x= 2, y=-2, z= 1>, vel=<x= 3, y= 5, z=-2>
pos=<x= 1, y=-4, z=-4>, vel=<x=-2, y=-4, z=-4>
pos=<x= 3, y=-7, z= 5>, vel=<x= 0, y=-5, z= 4>
pos=<x= 2, y= 0, z= 0>, vel=<x=-1, y= 4, z= 2>

After 8 steps:
pos=<x= 5, y= 2, z=-2>, vel=<x= 3, y= 4, z=-3>
pos=<x= 2, y=-7, z=-5>, vel=<x= 1, y=-3, z=-1>
pos=<x= 0, y=-9, z= 6>, vel=<x=-3, y=-2, z= 1>
pos=<x= 1, y= 1, z= 3>, vel=<x=-1, y= 1, z= 3>

After 9 steps:
```

```
pos=<x= 5, y= 3, z=-4>, vel=<x= 0, y= 1, z=-2>
pos=<x= 2, y=-9, z=-3>, vel=<x= 0, y=-2, z= 2>
pos=<x= 0, y=-8, z= 4>, vel=<x= 0, y= 1, z=-2>
pos=<x= 1, y= 1, z= 5>, vel=<x= 0, y= 0, z= 2>

After 10 steps:
pos=<x= 2, y= 1, z=-3>, vel=<x=-3, y=-2, z= 1>
pos=<x= 1, y=-8, z= 0>, vel=<x=-1, y= 1, z= 3>
pos=<x= 3, y=-6, z= 1>, vel=<x= 3, y= 2, z=-3>
pos=<x= 2, y= 0, z= 4>, vel=<x= 1, y=-1, z=-1>
```

Then, it might help to calculate the *total energy in the system*. The total energy for a single moon is its *potential energy* multiplied by its *kinetic energy*. A moon's *potential energy* is the sum of the absolute values of its `x`, `y`, and `z` position coordinates. A moon's *kinetic energy* is the sum of the absolute values of its velocity coordinates. Below, each line shows the calculations for a moon's potential energy (`pot`), kinetic energy (`kin`), and total energy:

```
Energy after 10 steps:
pot: 2 + 1 + 3 =  6;   kin: 3 + 2 + 1 = 6;   total:  6 * 6 = 36
pot: 1 + 8 + 0 =  9;   kin: 1 + 1 + 3 = 5;   total:  9 * 5 = 45
pot: 3 + 6 + 1 = 10;   kin: 3 + 2 + 3 = 8;   total: 10 * 8 = 80
pot: 2 + 0 + 4 =  6;   kin: 1 + 1 + 1 = 3;   total:  6 * 3 = 18
Sum of total energy: 36 + 45 + 80 + 18 = 179
```

In the above example, adding together the total energy for all moons after 10 steps produces the total energy in the system, `179`.

Here's a second example:

```
<x=-8, y=-10, z=0>
<x=5, y=5, z=10>
<x=2, y=-7, z=3>
<x=9, y=-8, z=-3>
```

Every ten steps of simulation for 100 steps produces:

```
After 0 steps:
pos=<x= -8, y=-10, z=  0>, vel=<x=  0, y=  0, z=  0>
pos=<x=  5, y=  5, z= 10>, vel=<x=  0, y=  0, z=  0>
pos=<x=  2, y= -7, z=  3>, vel=<x=  0, y=  0, z=  0>
pos=<x=  9, y= -8, z= -3>, vel=<x=  0, y=  0, z=  0>

After 10 steps:
pos=<x= -9, y=-10, z=  1>, vel=<x= -2, y= -2, z= -1>
pos=<x=  4, y= 10, z=  9>, vel=<x= -3, y=  7, z= -2>
pos=<x=  8, y=-10, z= -3>, vel=<x=  5, y= -1, z= -2>
pos=<x=  5, y=-10, z=  3>, vel=<x=  0, y= -4, z=  5>
```

```
After 20 steps:
pos=<x=-10, y=  3, z= -4>, vel=<x= -5, y=  2, z=  0>
pos=<x=  5, y=-25, z=  6>, vel=<x=  1, y=  1, z= -4>
pos=<x= 13, y=  1, z=  1>, vel=<x=  5, y= -2, z=  2>
pos=<x=  0, y=  1, z=  7>, vel=<x= -1, y= -1, z=  2>

After 30 steps:
pos=<x= 15, y= -6, z= -9>, vel=<x= -5, y=  4, z=  0>
pos=<x= -4, y=-11, z=  3>, vel=<x= -3, y=-10, z=  0>
pos=<x=  0, y= -1, z= 11>, vel=<x=  7, y=  4, z=  3>
pos=<x= -3, y= -2, z=  5>, vel=<x=  1, y=  2, z= -3>

After 40 steps:
pos=<x= 14, y=-12, z= -4>, vel=<x= 11, y=  3, z=  0>
pos=<x= -1, y= 18, z=  8>, vel=<x= -5, y=  2, z=  3>
pos=<x= -5, y=-14, z=  8>, vel=<x=  1, y= -2, z=  0>
pos=<x=  0, y=-12, z= -2>, vel=<x= -7, y= -3, z= -3>

After 50 steps:
pos=<x=-23, y=  4, z=  1>, vel=<x= -7, y= -1, z=  2>
pos=<x= 20, y=-31, z= 13>, vel=<x=  5, y=  3, z=  4>
pos=<x= -4, y=  6, z=  1>, vel=<x= -1, y=  1, z= -3>
pos=<x= 15, y=  1, z= -5>, vel=<x=  3, y= -3, z= -3>

After 60 steps:
pos=<x= 36, y=-10, z=  6>, vel=<x=  5, y=  0, z=  3>
pos=<x=-18, y= 10, z=  9>, vel=<x= -3, y= -7, z=  5>
pos=<x=  8, y=-12, z= -3>, vel=<x= -2, y=  1, z= -7>
pos=<x=-18, y= -8, z= -2>, vel=<x=  0, y=  6, z= -1>

After 70 steps:
pos=<x=-33, y= -6, z=  5>, vel=<x= -5, y= -4, z=  7>
pos=<x= 13, y= -9, z=  2>, vel=<x= -2, y= 11, z=  3>
pos=<x= 11, y= -8, z=  2>, vel=<x=  8, y= -6, z= -7>
pos=<x= 17, y=  3, z=  1>, vel=<x= -1, y= -1, z= -3>

After 80 steps:
pos=<x= 30, y= -8, z=  3>, vel=<x=  3, y=  3, z=  0>
pos=<x= -2, y= -4, z=  0>, vel=<x=  4, y=-13, z=  2>
pos=<x=-18, y= -7, z= 15>, vel=<x= -8, y=  2, z= -2>
pos=<x= -2, y= -1, z= -8>, vel=<x=  1, y=  8, z=  0>

After 90 steps:
pos=<x=-25, y= -1, z=  4>, vel=<x=  1, y= -3, z=  4>
pos=<x=  2, y= -9, z=  0>, vel=<x= -3, y= 13, z= -1>
pos=<x= 32, y= -8, z= 14>, vel=<x=  5, y= -4, z=  6>
```

```
pos=<x= -1, y= -2, z= -8>, vel=<x= -3, y= -6, z= -9>

After 100 steps:
pos=<x=  8, y=-12, z= -9>, vel=<x= -7, y=  3, z=  0>
pos=<x= 13, y= 16, z= -3>, vel=<x=  3, y=-11, z= -5>
pos=<x=-29, y=-11, z= -1>, vel=<x= -3, y=  7, z=  4>
pos=<x= 16, y=-13, z= 23>, vel=<x=  7, y=  1, z=  1>

Energy after 100 steps:
pot:  8 + 12 +  9 = 29;   kin: 7 +  3 + 0 = 10;   total: 29 * 10 = 290
pot: 13 + 16 +  3 = 32;   kin: 3 + 11 + 5 = 19;   total: 32 * 19 = 608
pot: 29 + 11 +  1 = 41;   kin: 3 +  7 + 4 = 14;   total: 41 * 14 = 574
pot: 16 + 13 + 23 = 52;   kin: 7 +  1 + 1 =  9;   total: 52 *  9 = 468
Sum of total energy: 290 + 608 + 574 + 468 = 1940
```

What is the total energy in the system after simulating the moons given in your scan for `1000` steps?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 13: Care Package ---

As you ponder the solitude of space and the ever-increasing three-hour roundtrip for messages between you and Earth, you notice that the Space Mail Indicator Light is blinking. To help keep you sane, the Elves have sent you a care package.

It's a new game for the ship's arcade cabinet! Unfortunately, the arcade is *all the way* on the other end of the ship. Surely, it won't be hard to build your own - the care package even comes with schematics.

The arcade cabinet runs Intcode software like the game the Elves sent (your puzzle input). It has a primitive screen capable of drawing square *tiles* on a grid. The software draws tiles to the screen with output instructions: every three output instructions specify the `x` position (distance from the left), `y` position (distance from the top), and `tile id`. The `tile id` is interpreted as follows:

- `0` is an *empty* tile. No game object appears in this tile.
- `1` is a *wall* tile. Walls are indestructible barriers.
- `2` is a *block* tile. Blocks can be broken by the ball.
- `3` is a *horizontal paddle* tile. The paddle is indestructible.
- `4` is a *ball* tile. The ball moves diagonally and bounces off objects.

For example, a sequence of output values like `1,2,3,6,5,4` would draw a *horizontal paddle* tile (`1` tile from the left and `2` tiles from the top) and a *ball* tile (`6` tiles from the left and `5` tiles from the top).

Start the game. *How many block tiles are on the screen when the game exits?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 14: Space Stoichiometry ---

As you approach the rings of Saturn, your ship's *low fuel* indicator turns on. There isn't any fuel here, but the rings have plenty of raw material. Perhaps your ship's Inter-Stellar Refinery Union brand *nanofactory* can turn these raw materials into fuel.

You ask the nanofactory to produce a list of the *reactions* it can perform that are relevant to this process (your puzzle input). Every reaction turns some quantities of specific *input chemicals* into some quantity of an *output chemical*. Almost every *chemical* is produced by exactly one reaction; the only exception, `ORE`, is the raw material input to the entire process and is not produced by a reaction.

You just need to know how much `ORE` you'll need to collect before you can produce one unit of `FUEL`.

Each reaction gives specific quantities for its inputs and output; reactions cannot be partially run, so only whole integer multiples of these quantities can be used. (It's okay to have leftover chemicals when you're done, though.) For example, the reaction `1 A, 2 B, 3 C => 2 D` means that exactly 2 units of chemical `D` can be produced by consuming exactly 1 `A`, 2 `B` and 3 `C`. You can run the full reaction as many times as necessary; for example, you could produce 10 `D` by consuming 5 `A`, 10 `B`, and 15 `C`.

Suppose your nanofactory produces the following list of reactions:

```
10 ORE => 10 A
1 ORE => 1 B
7 A, 1 B => 1 C
7 A, 1 C => 1 D
7 A, 1 D => 1 E
7 A, 1 E => 1 FUEL
```

The first two reactions use only `ORE` as inputs; they indicate that you can produce as much of chemical `A` as you want (in increments of 10 units, each 10 costing 10 `ORE`) and as much of chemical `B` as you want (each costing 1 `ORE`). To produce 1 `FUEL`, a total of *31* `ORE` is required: 1 `ORE` to produce 1 `B`, then 30 more `ORE` to produce the $7 + 7 + 7 + 7 = 28$ `A` (with 2 extra `A` wasted) required in the reactions to convert the `B` into `C`, `C` into `D`, `D` into `E`, and finally `E` into `FUEL`. (30 `A` is produced because its reaction requires that it is created in increments of 10.)

Or, suppose you have the following list of reactions:

```
9 ORE => 2 A
8 ORE => 3 B
```

```
7 ORE => 5 C
3 A, 4 B => 1 AB
5 B, 7 C => 1 BC
4 C, 1 A => 1 CA
2 AB, 3 BC, 4 CA => 1 FUEL
```

The above list of reactions requires *165* `ORE` to produce 1 `FUEL`:

- Consume 45 `ORE` to produce 10 `A`.
- Consume 64 `ORE` to produce 24 `B`.
- Consume 56 `ORE` to produce 40 `C`.
- Consume 6 `A`, 8 `B` to produce 2 `AB`.
- Consume 15 `B`, 21 `C` to produce 3 `BC`.
- Consume 16 `C`, 4 `A` to produce 4 `CA`.
- Consume 2 `AB`, 3 `BC`, 4 `CA` to produce 1 `FUEL`.

Here are some larger examples:

- *13312* `ORE` for 1 `FUEL`:

  ```
  157 ORE => 5 NZVS
  165 ORE => 6 DCFZ
  44 XJWVT, 5 KHKGT, 1 QDVJ, 29 NZVS, 9 GPVTF, 48 HKGWZ => 1 FUEL
  12 HKGWZ, 1 GPVTF, 8 PSHF => 9 QDVJ
  179 ORE => 7 PSHF
  177 ORE => 5 HKGWZ
  7 DCFZ, 7 PSHF => 2 XJWVT
  165 ORE => 2 GPVTF
  3 DCFZ, 7 NZVS, 5 HKGWZ, 10 PSHF => 8 KHKGT
  ```

- *180697* `ORE` for 1 `FUEL`:

  ```
  2 VPVL, 7 FWMGM, 2 CXFTF, 11 MNCFX => 1 STKFG
  17 NVRVD, 3 JNWZP => 8 VPVL
  53 STKFG, 6 MNCFX, 46 VJHF, 81 HVMC, 68 CXFTF, 25 GNMV => 1 FUEL
  22 VJHF, 37 MNCFX => 5 FWMGM
  139 ORE => 4 NVRVD
  144 ORE => 7 JNWZP
  5 MNCFX, 7 RFSQX, 2 FWMGM, 2 VPVL, 19 CXFTF => 3 HVMC
  5 VJHF, 7 MNCFX, 9 VPVL, 37 CXFTF => 6 GNMV
  145 ORE => 6 MNCFX
  1 NVRVD => 8 CXFTF
  1 VJHF, 6 MNCFX => 4 RFSQX
  176 ORE => 6 VJHF
  ```

- *2210736* `ORE` for 1 `FUEL`:

  ```
  171 ORE => 8 CNZTR
  7 ZLQW, 3 BMBT, 9 XCVML, 26 XMNCP, 1 WPTQ, 2 MZWV, 1 RJRHP => 4 PLWSL
  114 ORE => 4 BHXH
  ```

```
14 VRPVC => 6 BMBT
6 BHXH, 18 KTJDG, 12 WPTQ, 7 PLWSL, 31 FHTLT, 37 ZDVW => 1 FUEL
6 WPTQ, 2 BMBT, 8 ZLQW, 18 KTJDG, 1 XMNCP, 6 MZWV, 1 RJRHP => 6 FHTLT
15 XDBXC, 2 LTCX, 1 VRPVC => 6 ZLQW
13 WPTQ, 10 LTCX, 3 RJRHP, 14 XMNCP, 2 MZWV, 1 ZLQW => 1 ZDVW
5 BMBT => 4 WPTQ
189 ORE => 9 KTJDG
1 MZWV, 17 XDBXC, 3 XCVML => 2 XMNCP
12 VRPVC, 27 CNZTR => 2 XDBXC
15 KTJDG, 12 BHXH => 5 XCVML
3 BHXH, 2 VRPVC => 7 MZWV
121 ORE => 7 VRPVC
7 XCVML => 6 RJRHP
5 BHXH, 4 VRPVC => 5 LTCX
```

Given the list of reactions in your puzzle input, *what is the minimum amount of `ORE` required to produce exactly 1 `FUEL`?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 15: Oxygen System ---

Out here in deep space, many things can go wrong. Fortunately, many of those things have indicator lights. Unfortunately, one of those lights is lit: the oxygen system for part of the ship has failed!

According to the readouts, the oxygen system must have failed days ago after a rupture in oxygen tank two; that section of the ship was automatically sealed once oxygen levels went dangerously low. A single remotely-operated *repair droid* is your only option for fixing the oxygen system.

The Elves' care package included an Intcode program (your puzzle input) that you can use to remotely control the repair droid. By running that program, you can direct the repair droid to the oxygen system and fix the problem.

The remote control program executes the following steps in a loop forever:

- Accept a *movement command* via an input instruction.
- Send the movement command to the repair droid.
- Wait for the repair droid to finish the movement operation.
- Report on the *status* of the repair droid via an output instruction.

Only four *movement commands* are understood: north (`1`), south (`2`), west (`3`), and east (`4`). Any other command is invalid. The movements differ in direction, but not in distance: in a long enough east-west hallway, a series of commands like `4,4,4,4,3,3,3,3` would leave the repair droid back where it started.

The repair droid can reply with any of the following *status* codes:

- `0`: The repair droid hit a wall. Its position has not changed.
- `1`: The repair droid has moved one step in the requested direction.
- `2`: The repair droid has moved one step in the requested direction; its new position is the location of the oxygen system.

You don't know anything about the area around the repair droid, but you can figure it out by watching the status codes.

For example, we can draw the area using `D` for the droid, `#` for walls, `.` for locations the droid can traverse, and empty space for unexplored locations. Then, the initial state looks like this:

```
D
```

To make the droid go north, send it `1`. If it replies with `0`, you know that location is a wall and that the droid didn't move:

```
#
D
```

To move east, send `4`; a reply of `1` means the movement was successful:

```
#
.D
```

Then, perhaps attempts to move north (`1`), south (`2`), and east (`4`) are all met with replies of `0`:

```
##
.D#
 #
```

Now, you know the repair droid is in a dead end. Backtrack with `3` (which you already know will get a reply of `1` because you already know that location is open):

```
##
D.#
```

```
 #
```

Then, perhaps west (`3`) gets a reply of `0`, south (`2`) gets a reply of `1`, south again (`2`) gets a reply of `0`, and then west (`3`) gets a reply of `2`:

```
  ##
 #..#
 D.#
  #
```

Now, because of the reply of `2`, you know you've found the *oxygen system*! In this example, it was only `2` moves away from the repair droid's starting position.

What is the fewest number of movement commands required to move the repair droid from its starting position to the location of the oxygen system?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 16: Flawed Frequency Transmission ---

You're 3/4ths of the way through the gas giants. Not only do roundtrip signals to Earth take five hours, but the signal quality is quite bad as well. You can clean up the signal with the Flawed Frequency Transmission algorithm, or *FFT*.

As input, FFT takes a list of numbers. In the signal you received (your puzzle input), each number is a single digit: data like `15243` represents the sequence `1`, `5`, `2`, `4`, `3`.

FFT operates in repeated *phases*. In each phase, a new list is constructed with the same length as the input list. This new list is also used as the input for the next phase.

Each element in the new list is built by multiplying every value in the input list by a value in a repeating *pattern* and then adding up the results. So, if the input list were `9, 8, 7, 6, 5` and the pattern for a given element were `1, 2, 3`, the result would be `9*1 + 8*2 + 7*3 + 6*1 + 5*2` (with each input element on the left and each value in the repeating pattern on the right of each multiplication). Then, only the ones digit is kept: `38` becomes `8`, `-17` becomes `7`, and so on.

While each element in the output array uses all of the same input array elements, the actual repeating pattern to use depends on *which output element* is being calculated. The base pattern is `0, 1, 0, -1`. Then, repeat each value in the pattern a number of times equal to the *position in the output list* being considered. Repeat once for the first element, twice for the second element, three times for the third element, and so on. So, if the third element of the output list is being

calculated, repeating the values would produce: `0, 0, 0, 1, 1, 1, 0, 0, 0, -1, -1, -1`.

When applying the pattern, skip the very first value exactly once. (In other words, offset the whole pattern left by one.) So, for the second element of the output list, the actual pattern used would be: `0, 1, 1, 0, 0, -1, -1, 0, 0, 1, 1, 0, 0, -1, -1, ...`.

After using this process to calculate each element of the output list, the phase is complete, and the output list of this phase is used as the new input list for the next phase, if any.

Given the input signal `12345678`, below are four phases of FFT. Within each phase, each output digit is calculated on a single line with the result at the far right; each multiplication operation shows the input digit on the left and the pattern value on the right:

```
Input signal: 12345678

1*1  + 2*0  + 3*-1 + 4*0  + 5*1  + 6*0  + 7*-1 + 8*0  = 4
1*0  + 2*1  + 3*1  + 4*0  + 5*0  + 6*-1 + 7*-1 + 8*0  = 8
1*0  + 2*0  + 3*1  + 4*1  + 5*1  + 6*0  + 7*0  + 8*0  = 2
1*0  + 2*0  + 3*0  + 4*1  + 5*1  + 6*1  + 7*1  + 8*0  = 2
1*0  + 2*0  + 3*0  + 4*0  + 5*1  + 6*1  + 7*1  + 8*1  = 6
1*0  + 2*0  + 3*0  + 4*0  + 5*0  + 6*1  + 7*1  + 8*1  = 1
1*0  + 2*0  + 3*0  + 4*0  + 5*0  + 6*0  + 7*1  + 8*1  = 5
1*0  + 2*0  + 3*0  + 4*0  + 5*0  + 6*0  + 7*0  + 8*1  = 8

After 1 phase: 48226158

4*1  + 8*0  + 2*-1 + 2*0  + 6*1  + 1*0  + 5*-1 + 8*0  = 3
4*0  + 8*1  + 2*1  + 2*0  + 6*0  + 1*-1 + 5*-1 + 8*0  = 4
4*0  + 8*0  + 2*1  + 2*1  + 6*1  + 1*0  + 5*0  + 8*0  = 0
4*0  + 8*0  + 2*0  + 2*1  + 6*1  + 1*1  + 5*1  + 8*0  = 4
4*0  + 8*0  + 2*0  + 2*0  + 6*1  + 1*1  + 5*1  + 8*1  = 0
4*0  + 8*0  + 2*0  + 2*0  + 6*0  + 1*1  + 5*1  + 8*1  = 4
4*0  + 8*0  + 2*0  + 2*0  + 6*0  + 1*0  + 5*1  + 8*1  = 3
4*0  + 8*0  + 2*0  + 2*0  + 6*0  + 1*0  + 5*0  + 8*1  = 8

After 2 phases: 34040438

3*1  + 4*0  + 0*-1 + 4*0  + 0*1  + 4*0  + 3*-1 + 8*0  = 0
3*0  + 4*1  + 0*1  + 4*0  + 0*0  + 4*-1 + 3*-1 + 8*0  = 3
3*0  + 4*0  + 0*1  + 4*1  + 0*1  + 4*0  + 3*0  + 8*0  = 4
3*0  + 4*0  + 0*0  + 4*1  + 0*1  + 4*1  + 3*1  + 8*0  = 1
3*0  + 4*0  + 0*0  + 4*0  + 0*1  + 4*1  + 3*1  + 8*1  = 5
3*0  + 4*0  + 0*0  + 4*0  + 0*0  + 4*1  + 3*1  + 8*1  = 5
3*0  + 4*0  + 0*0  + 4*0  + 0*0  + 4*0  + 3*1  + 8*1  = 1
```

```
3*0  + 4*0  + 0*0  + 4*0  + 0*0  + 4*0  + 3*0  + 8*1  = 8

After 3 phases: 03415518

0*1  + 3*0  + 4*-1 + 1*0  + 5*1  + 5*0  + 1*-1 + 8*0  = 0
0*0  + 3*1  + 4*1  + 1*0  + 5*0  + 5*-1 + 1*-1 + 8*0  = 1
0*0  + 3*0  + 4*1  + 1*1  + 5*1  + 5*0  + 1*0  + 8*0  = 0
0*0  + 3*0  + 4*0  + 1*1  + 5*1  + 5*1  + 1*1  + 8*0  = 2
0*0  + 3*0  + 4*0  + 1*0  + 5*1  + 5*1  + 1*1  + 8*1  = 9
0*0  + 3*0  + 4*0  + 1*0  + 5*0  + 5*1  + 1*1  + 8*1  = 4
0*0  + 3*0  + 4*0  + 1*0  + 5*0  + 5*0  + 1*1  + 8*1  = 9
0*0  + 3*0  + 4*0  + 1*0  + 5*0  + 5*0  + 1*0  + 8*1  = 8

After 4 phases: 01029498
```

Here are the first eight digits of the final output list after 100 phases for some larger inputs:

- `80871224585914546619083218645595` becomes `24176176`.
- `19617804207202209144916044189917` becomes `73745418`.
- `69317163492948606335995924319873` becomes `52432133`.

After *100* phases of FFT, *what are the first eight digits in the final output list?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 17: Set and Forget ---

An early warning system detects an incoming solar flare and automatically activates the ship's electromagnetic shield. Unfortunately, this has cut off the Wi-Fi for many small robots that, unaware of the impending danger, are now trapped on exterior scaffolding on the unsafe side of the shield. To rescue them, you'll have to act quickly!

The only tools at your disposal are some wired cameras and a small vacuum robot currently asleep at its charging station. The video quality is poor, but the vacuum robot has a needlessly bright LED that makes it easy to spot no matter where it is.

An Intcode program, the *Aft Scaffolding Control and Information Interface* (ASCII, your puzzle input), provides access to the cameras and the vacuum robot. Currently, because the vacuum robot is asleep, you can only access the cameras.

Running the ASCII program on your Intcode computer will provide the current view of the scaffolds. This is output, purely coincidentally, as ASCII code: `35` means `#`, `46` means `.`, `10` starts a new line of output below the current one, and so on. (Within a line, characters are drawn left-to-right.)

In the camera output, `#` represents a scaffold and `.` represents open space. The vacuum robot is visible as `^`, `v`, `<`, or `>` depending on whether it is facing up, down, left, or right respectively. When drawn like this, the vacuum robot is *always on a scaffold*; if the vacuum robot ever walks off of a scaffold and begins *tumbling through space uncontrollably*, it will instead be visible as `X`.

In general, the scaffold forms a path, but it sometimes loops back onto itself. For example, suppose you can see the following view from the cameras:

```
..#..........
..#..........
#######...###
#.#...#...#.#
#############
..#...#...#..
..#####...^..
```

Here, the vacuum robot, `^` is facing up and sitting at one end of the scaffold near the bottom-right of the image. The scaffold continues up, loops across itself several times, and ends at the top-left of the image.

The first step is to calibrate the cameras by getting the *alignment parameters* of some well-defined points. Locate all *scaffold intersections*; for each, its alignment parameter is the distance between its left edge and the left edge of the view multiplied by the distance between its top edge and the top edge of the view. Here, the intersections from the above image are marked `O`:

```
..#..........
..#..........
##O####...###
#.#...#...#.#
##O###O###O##
..#...#...#..
..#####...^..
```

For these intersections:

- The top-left intersection is `2` units from the left of the image and `2` units from the top of the image, so its alignment parameter is `2 * 2 = 4`.
- The bottom-left intersection is `2` units from the left and `4` units from the top, so its alignment parameter is `2 * 4 = 8`.
- The bottom-middle intersection is `6` from the left and `4` from the top, so its alignment parameter is `24`.
- The bottom-right intersection's alignment parameter is `40`.

To calibrate the cameras, you need the *sum of the alignment parameters*. In the above example, this is `76`.

Run your ASCII program. *What is the sum of the alignment parameters* for the scaffold intersections?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 18: Many-Worlds Interpretation ---

As you approach Neptune, a planetary security system detects you and activates a giant tractor beam on Triton! You have no choice but to land.

A scan of the local area reveals only one interesting feature: a massive underground vault. You generate a map of the tunnels (your puzzle input). The tunnels are too narrow to move diagonally.

Only one *entrance* (marked `@`) is present among the *open passages* (marked `.`) and *stone walls* (`#`), but you also detect an assortment of *keys* (shown as lowercase letters) and *doors* (shown as uppercase letters). Keys of a given letter open the door of the same letter: `a` opens `A`, `b` opens `B`, and so on. You aren't sure which key you need to disable the tractor beam, so you'll need to *collect all of them*.

For example, suppose you have the following map:

```
#########
#b.A.@.a#
#########
```

Starting from the entrance (`@`), you can only access a large door (`A`) and a key (`a`). Moving toward the door doesn't help you, but you can move `2` steps to collect the key, unlocking `A` in the process:

```
#########
#b.....@#
#########
```

Then, you can move `6` steps to collect the only other key, `b`:

```
#########
#@......#
#########
```

So, collecting every key took a total of `8` steps.

Here is a larger example:

```
########################
#f.D.E.e.C.b.A.@.a.B.c.#
######################.#
#d.....................#
########################
```

The only reasonable move is to take key `a` and unlock door `A`:

```
########################
#f.D.E.e.C.b.....@.B.c.#
######################.#
#d.....................#
########################
```

Then, do the same with key `b`:

```
########################
#f.D.E.e.C.@.........c.#
######################.#
#d.....................#
########################
```

...and the same with key `c`:

```
########################
#f.D.E.e.............@.#
######################.#
#d.....................#
########################
```

Now, you have a choice between keys `d` and `e`. While key `e` is closer, collecting it now would be slower in the long run than collecting key `d` first, so that's the best choice:

```
########################
#f...E.e...............#
######################.#
#@.....................#
########################
```

Finally, collect key `e` to unlock door `E`, then collect key `f`, taking a grand total of `86` steps.

Here are a few more examples:

- ```
  ########################
  #..............b.C.D.f#
  #.######################
  #.....@.a.B.c.d.A.e.F.g#
  ########################
  ```

  Shortest path is `132` steps: `b`, `a`, `c`, `d`, `f`, `e`, `g`

- ```
  ##################
  #i.G..c...e..H.p#
  ########.########
  #j.A..b...f..D.o#
  ########@########
  #k.E..a...g..B.n#
  ```

```
########.########
#l.F..d...h..C.m#
##################
```

Shortest paths are 136 steps;
one is: a, f, b, j, g, n, h, d, l, o, e, p, c, i, k, m

- ```
  ########################
  #@.............ac.GI.b#
  ###d#e#f################
  ###A#B#C################
  ###g#h#i################
  ########################
  ```

  Shortest paths are 81 steps; one is: a, c, f, i, d, g, b, e, h

How many steps is the shortest path that collects all of the keys?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 19: Tractor Beam ---

Unsure of the state of Santa's ship, you borrowed the tractor beam technology from Triton. Time to test it out.

When you're safely away from anything else, you activate the tractor beam, but nothing happens. It's hard to tell whether it's working if there's nothing to use it on. Fortunately, your ship's drone system can be configured to deploy a drone to specific coordinates and then check whether it's being pulled. There's even an Intcode program (your puzzle input) that gives you access to the drone system.

The program uses two input instructions to request the *X and Y position* to which the drone should be deployed. Negative numbers are invalid and will confuse the drone; all numbers should be *zero or positive*.

Then, the program will output whether the drone is *stationary* (0) or *being pulled by something* (1). For example, the coordinate X=0, Y=0 is directly in front of the tractor beam emitter, so the drone control program will always report 1 at that location.

To better understand the tractor beam, it is important to *get a good picture* of the beam itself. For example, suppose you scan the 10x10 grid of points closest to the emitter:

```
      X
  0->      9
 0#.........
 |.#........
 v..##......
```
```

```
  ...###....
  ....###...
Y .....####.
  ......####
  ......####
  .......###
 9........##
```

In this example, the *number of points affected by the tractor beam* in the 10x10 area closest to the emitter is `27`.

However, you'll need to scan a larger area to *understand the shape* of the beam. *How many points are affected by the tractor beam in the 50x50 area closest to the emitter?* (For each of X and Y, this will be `0` through `49`.)

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 20: Donut Maze ---

You notice a strange pattern on the surface of Pluto and land nearby to get a closer look. Upon closer inspection, you realize you've come across one of the famous space-warping mazes of the long-lost Pluto civilization!

Because there isn't much space on Pluto, the civilization that used to live here thrived by inventing a method for folding spacetime. Although the technology is no longer understood, mazes like this one provide a small glimpse into the daily life of an ancient Pluto citizen.

This maze is shaped like a donut. Portals along the inner and outer edge of the donut can instantly teleport you from one side to the other. For example:

```
         A
         A
  #######.#########
  #######.........#
  #######.#######.#
  #######.#######.#
  #######.#######.#
  #####  B    ###.#
BC...##  C    ###.#
  ##.##       ###.#
  ##...DE  F  ###.#
  #####    G  ###.#
  #########.#####.#
DE..#######...###.#
  #.#########.###.#
FG..#########.....#
```

```
  ###########.#####
             Z
             Z
```

This map of the maze shows solid walls (`#`) and open passages (`.`). Every maze on Pluto has a start (the open tile next to `AA`) and an end (the open tile next to `ZZ`). Mazes on Pluto also have portals; this maze has three pairs of portals: `BC`, `DE`, and `FG`. When on an open tile next to one of these labels, a single step can take you to the other tile with the same label. (You can only walk on `.` tiles; labels and empty space are not traversable.)

One path through the maze doesn't require any portals. Starting at `AA`, you could go down 1, right 8, down 12, left 4, and down 1 to reach `ZZ`, a total of 26 steps.

However, there is a shorter path: You could walk from `AA` to the inner `BC` portal (4 steps), warp to the outer `BC` portal (1 step), walk to the inner `DE` (6 steps), warp to the outer `DE` (1 step), walk to the outer `FG` (4 steps), warp to the inner `FG` (1 step), and finally walk to `ZZ` (6 steps). In total, this is only *23* steps.

Here is a larger example:

```
                   A
                   A
  #################.#############
  #.#...#.....................#.#.#
  #.#.#.###.###.###.#########.#.#
  #.#.#.......#...#.....#.#.#...#
  #.#########.###.#####.#.#.###.#
  #.............#.#.....#.......#
  ###.############.###.#####.#.#.#
  #.....#        A   C    #.#.#.#
  #######        S   P    #####.#
  #.#...#                 #......VT
  #.#.#.#                 #.#####
  #...#.#               YN....#.#
  #.###.#                 #####.#
DI....#.#                 #.....#
  #####.#                 #.###.#
ZZ......#               QG....#..AS
  ###.###                 #######
JO..#.#.#                 #.....#
  #.#.#.#                 ###.#.#
  #...#..DI             BU....#..LF
  #####.#                 #.#####
YN......#               VT..#....QG
  #.###.#                 #.###.#
  #.#...#                 #.....#
```

```
###.###    J L     J    #.#.###
#.....#    O F     P    #.#...#
#.###.#####.#.#####.#####.###.#
#...#.#.#...#.....#.....#.#...#
#.#####.###.###.#.#.#########.#
#...#.#.....#...#.#.#.#.....#.#
#.###.#####.###.###.#.#.#######
#.#.........#...#.............#
#########.###.###.##############
         B   J   C
         U   P   P
```

Here, `AA` has no direct path to `ZZ`, but it does connect to `AS` and `CP`. By passing through `AS`, `QG`, `BU`, and `JO`, you can reach `ZZ` in *58* steps.

In your maze, *how many steps does it take to get from the open tile marked `AA` to the open tile marked `ZZ`?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 21: Springdroid Adventure ---

You lift off from Pluto and start flying in the direction of Santa.

While experimenting further with the tractor beam, you accidentally pull an asteroid directly into your ship! It deals significant damage to your hull and causes your ship to begin tumbling violently.

You can send a droid out to investigate, but the tumbling is causing enough artificial gravity that one wrong step could send the droid through a hole in the hull and flying out into space.

The clear choice for this mission is a droid that can *jump* over the holes in the hull - a *springdroid*.

You can use an Intcode program (your puzzle input) running on an ASCII-capable computer to program the springdroid. However, springdroids don't run Intcode; instead, they run a simplified assembly language called *springscript*.

While a springdroid is certainly capable of navigating the artificial gravity and giant holes, it has one downside: it can only remember at most *15* springscript instructions.

The springdroid will move forward automatically, constantly thinking about *whether to jump*. The springscript program defines the logic for this decision.

Springscript programs only use Boolean values, not numbers or strings. Two registers are available: `T`, the *temporary value* register, and `J`, the *jump* register.

If the jump register is *true* at the end of the springscript program, the springdroid will try to jump. Both of these registers start with the value *false*.

Springdroids have a sensor that can detect *whether there is ground* at various distances in the direction it is facing; these values are provided in *read-only registers*. Your springdroid can detect ground at four distances: one tile away (`A`), two tiles away (`B`), three tiles away (`C`), and four tiles away (`D`). If there is ground at the given distance, the register will be *true*; if there is a hole, the register will be *false*.

There are only *three instructions* available in springscript:

- `AND X Y` sets `Y` to *true* if both `X` and `Y` are *true*; otherwise, it sets `Y` to *false*.
- `OR X Y` sets `Y` to *true* if at least one of `X` or `Y` is *true*; otherwise, it sets `Y` to *false*.
- `NOT X Y` sets `Y` to *true* if `X` is *false*; otherwise, it sets `Y` to *false*.

In all three instructions, the second argument (`Y`) needs to be a *writable register* (either `T` or `J`). The first argument (`X`) can be *any register* (including `A`, `B`, `C`, or `D`).

For example, the one-instruction program `NOT A J` means "if the tile immediately in front of me is not ground, jump".

Or, here is a program that jumps if a three-tile-wide hole (with ground on the other side of the hole) is detected:

```
NOT A J
NOT B T
AND T J
NOT C T
AND T J
AND D J
```

The Intcode program expects ASCII inputs and outputs. It will begin by displaying a prompt; then, input the desired instructions one per line. End each line with a newline (ASCII code `10`). *When you have finished entering your program*, provide the command `WALK` followed by a newline to instruct the springdroid to begin surveying the hull.

If the springdroid *falls into space*, an ASCII rendering of the last moments of its life will be produced. In these, `@` is the springdroid, `#` is hull, and `.` is empty space. For example, suppose you program the springdroid like this:

```
NOT D J
WALK
```

This one-instruction program sets `J` to *true* if and only if there is no ground four tiles away. In other words, it attempts to jump into any hole it finds:

```
.................
.................
```

```
@................
#####.############

.................
.................
.@...............
#####.############

.................
..@..............
.................
#####.############

...@.............
.................
.................
#####.############

.................
....@............
.................
#####.############

.................
.................
.....@...........
#####.############

.................
.................
.................
#####@############
```

However, if the springdroid successfully makes it across, it will use an output instruction to indicate the *amount of damage to the hull* as a single giant integer outside the normal ASCII range.

Program the springdroid with logic that allows it to survey the hull without falling into space. *What amount of hull damage does it report?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 22: Slam Shuffle ---

There isn't much to do while you wait for the droids to repair your ship. At least you're drifting in the right direction. You decide to practice a new card

shuffle you've been working on.

Digging through the ship's storage, you find a deck of *space cards*! Just like any deck of space cards, there are 10007 cards in the deck numbered `0` through `10006`. The deck must be new - they're still in *factory order*, with `0` on the top, then `1`, then `2`, and so on, all the way through to `10006` on the bottom.

You've been practicing three different *techniques* that you use while shuffling. Suppose you have a deck of only 10 cards (numbered `0` through `9`):

To `deal into new stack`, create a new stack of cards by dealing the top card of the deck onto the top of the new stack repeatedly until you run out of cards:

```
Top          Bottom
0 1 2 3 4 5 6 7 8 9   Your deck
                      New stack

  1 2 3 4 5 6 7 8 9   Your deck
                  0   New stack

    2 3 4 5 6 7 8 9   Your deck
                1 0   New stack

      3 4 5 6 7 8 9   Your deck
                2 1 0   New stack

Several steps later...

                    9   Your deck
  8 7 6 5 4 3 2 1 0   New stack

                      Your deck
9 8 7 6 5 4 3 2 1 0   New stack
```

Finally, pick up the new stack you've just created and use it as the deck for the next technique.

To `cut N` cards, take the top `N` cards off the top of the deck and move them as a single unit to the bottom of the deck, retaining their order. For example, to `cut 3`:

```
Top          Bottom
0 1 2 3 4 5 6 7 8 9   Your deck

      3 4 5 6 7 8 9   Your deck
0 1 2                 Cut cards

3 4 5 6 7 8 9         Your deck
              0 1 2   Cut cards
```

```
3 4 5 6 7 8 9 0 1 2   Your deck
```

You've also been getting pretty good at a version of this technique where `N` is negative! In that case, cut (the absolute value of) `N` cards from the bottom of the deck onto the top. For example, to `cut -4`:

```
Top          Bottom
0 1 2 3 4 5 6 7 8 9   Your deck

0 1 2 3 4 5           Your deck
            6 7 8 9   Cut cards

        0 1 2 3 4 5   Your deck
6 7 8 9               Cut cards

6 7 8 9 0 1 2 3 4 5   Your deck
```

To `deal with increment N`, start by clearing enough space on your table to lay out all of the cards individually in a long line. Deal the top card into the leftmost position. Then, move `N` positions to the right and deal the next card there. If you would move into a position past the end of the space on your table, wrap around and keep counting from the leftmost card again. Continue this process until you run out of cards.

For example, to `deal with increment 3`:

```
0 1 2 3 4 5 6 7 8 9   Your deck
. . . . . . . . . .   Space on table
^                     Current position

Deal the top card to the current position:

  1 2 3 4 5 6 7 8 9   Your deck
0 . . . . . . . . .   Space on table
^                     Current position

Move the current position right 3:

  1 2 3 4 5 6 7 8 9   Your deck
0 . . . . . . . . .   Space on table
      ^               Current position

Deal the top card:

    2 3 4 5 6 7 8 9   Your deck
0 . . 1 . . . . . .   Space on table
      ^               Current position
```

```
Move right 3 and deal:

      3 4 5 6 7 8 9   Your deck
0 . . 1 . . 2 . . .   Space on table
            ^         Current position

Move right 3 and deal:

        4 5 6 7 8 9   Your deck
0 . . 1 . . 2 . . 3   Space on table
                ^     Current position

Move right 3, wrapping around, and deal:

          5 6 7 8 9   Your deck
0 . 4 1 . . 2 . . 3   Space on table
    ^                 Current position

And so on:

0 7 4 1 8 5 2 9 6 3   Space on table
```

Positions on the table which already contain cards are still counted; they're not skipped. Of course, this technique is carefully designed so it will never put two cards in the same position or leave a position empty.

Finally, collect the cards on the table so that the leftmost card ends up at the top of your deck, the card to its right ends up just below the top card, and so on, until the rightmost card ends up at the bottom of the deck.

The complete shuffle process (your puzzle input) consists of applying many of these techniques. Here are some examples that combine techniques; they all start with a *factory order* deck of 10 cards:

```
deal with increment 7
deal into new stack
deal into new stack
Result: 0 3 6 9 2 5 8 1 4 7

cut 6
deal with increment 7
deal into new stack
Result: 3 0 7 4 1 8 5 2 9 6

deal with increment 7
deal with increment 9
cut -2
Result: 6 3 0 7 4 1 8 5 2 9
```

```
deal into new stack
cut -2
deal with increment 7
cut 8
cut -4
deal with increment 7
cut 3
deal with increment 9
deal with increment 3
cut -1
Result: 9 2 5 8 1 4 7 0 3 6
```

Positions within the deck count from `0` at the top, then `1` for the card immediately below the top card, and so on to the bottom. (That is, cards start in the position matching their number.)

After shuffling your *factory order* deck of 10007 cards, *what is the position of card `2019`?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 23: Category Six ---

The droids have finished repairing as much of the ship as they can. Their report indicates that this was a *Category 6* disaster - not because it was that bad, but because it destroyed the stockpile of Category 6 network cables as well as most of the ship's network infrastructure.

You'll need to *rebuild the network from scratch.*

The computers on the network are standard Intcode computers that communicate by sending *packets* to each other. There are `50` of them in total, each running a copy of the same *Network Interface Controller* (NIC) software (your puzzle input). The computers have *network addresses* `0` through `49`; when each computer boots up, it will request its network address via a single input instruction. Be sure to give each computer a unique network address.

Once a computer has received its network address, it will begin doing work and communicating over the network by sending and receiving *packets*. All packets contain *two values* named `X` and `Y`. Packets sent to a computer are queued by the recipient and read in the order they are received.

To *send* a packet to another computer, the NIC will use *three output instructions* that provide the *destination address* of the packet followed by its `X` and `Y` values. For example, three output instructions that provide the values `10`, `20`, `30` would send a packet with `X=20` and `Y=30` to the computer with address `10`.

To *receive* a packet from another computer, the NIC will use an *input instruction*.

If the incoming packet queue is *empty*, provide `-1`. Otherwise, provide the `X` value of the next packet; the computer will then use a second input instruction to receive the `Y` value for the same packet. Once both values of the packet are read in this way, the packet is removed from the queue.

Note that these input and output instructions never block. Specifically, output instructions do not wait for the sent packet to be received - the computer might send multiple packets before receiving any. Similarly, input instructions do not wait for a packet to arrive - if no packet is waiting, input instructions should receive `-1`.

Boot up all `50` computers and attach them to your network. *What is the `Y` value of the first packet sent to address `255`?*

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 24: Planet of Discord ---

You land on Eris, your last stop before reaching Santa. As soon as you do, your sensors start picking up strange life forms moving around: Eris is infested with bugs! With an over 24-hour roundtrip for messages between you and Earth, you'll have to deal with this problem on your own.

Eris isn't a very large place; a scan of the entire area fits into a 5x5 grid (your puzzle input). The scan shows *bugs* (`#`) and *empty spaces* (`.`).

Each *minute*, The bugs live and die based on the number of bugs in the *four adjacent tiles*:

- A bug *dies* (becoming an empty space) unless there is *exactly one* bug adjacent to it.
- An empty space *becomes infested* with a bug if *exactly one or two* bugs are adjacent to it.

Otherwise, a bug or empty space remains the same. (Tiles on the edges of the grid have fewer than four adjacent tiles; the missing tiles count as empty space.) This process happens in every location *simultaneously*; that is, within the same minute, the number of adjacent bugs is counted for every tile first, and then the tiles are updated.

Here are the first few minutes of an example scenario:

```
Initial state:
....#
#..#.
#..##
..#..
#....
```

```
After 1 minute:
#..#.
####.
###.#
##.##
.##..

After 2 minutes:
#####
....#
....#
...#.
#.###

After 3 minutes:
#....
####.
...##
#.##.
.##.#

After 4 minutes:
####.
....#
##..#
.....
##...
```

To understand the nature of the bugs, watch for the first time a layout of bugs and empty spaces *matches any previous layout*. In the example above, the first layout to appear twice is:

```
.....
.....
.....
#....
.#...
```

To calculate the *biodiversity rating* for this layout, consider each tile left-to-right in the top row, then left-to-right in the second row, and so on. Each of these tiles is worth biodiversity points equal to *increasing powers of two*: 1, 2, 4, 8, 16, 32, and so on. Add up the biodiversity points for tiles with bugs; in this example, the 16th tile (`32768` points) and 22nd tile (`2097152` points) have bugs, a total biodiversity rating of `2129920`.

What is the biodiversity rating for the first layout that appears twice?

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]

--- Day 25: Cryostasis ---

As you approach Santa's ship, your sensors report two important details:

First, that you might be too late: the internal temperature is `-40` degrees.

Second, that one faint life signature is somewhere on the ship.

The airlock door is locked with a code; your best option is to send in a small droid to investigate the situation. You attach your ship to Santa's, break a small hole in the hull, and let the droid run in before you seal it up again. Before your ship starts freezing, you detach your ship and set it to automatically stay within range of Santa's ship.

This droid can follow basic instructions and report on its surroundings; you can communicate with it through an Intcode program (your puzzle input) running on an ASCII-capable computer.

As the droid moves through its environment, it will describe what it encounters. When it says `Command?`, you can give it a single instruction terminated with a newline (ASCII code `10`). Possible instructions are:

- *Movement* via `north`, `south`, `east`, or `west`.
- To *take* an item the droid sees in the environment, use the command `take <name of item>`. For example, if the droid reports seeing a `red ball`, you can pick it up with `take red ball`.
- To *drop* an item the droid is carrying, use the command `drop <name of item>`. For example, if the droid is carrying a `green ball`, you can drop it with `drop green ball`.
- To get a *list of all of the items* the droid is currently carrying, use the command `inv` (for "inventory").

Extra spaces or other characters aren't allowed - instructions must be provided precisely.

Santa's ship is a *Reindeer-class starship*; these ships use pressure-sensitive floors to determine the identity of droids and crew members. The standard configuration for these starships is for all droids to weigh exactly the same amount to make them easier to detect. If you need to get past such a sensor, you might be able to reach the correct weight by carrying items from the environment.

Look around the ship and see if you can find the *password for the main airlock*.

To play, please identify yourself via one of these services:

[GitHub] [Google] [Twitter] [Reddit] - [How Does Auth Work?]