--- Day 1: Sonar Sweep ---

You're minding your own business on a ship at sea when the overboard alarm goes off! You rush to see if you can help. Apparently, one of the Elves tripped and accidentally sent the sleigh keys flying into the ocean!

Before you know it, you're inside a submarine the Elves keep ready for situations like this. It's covered in Christmas lights (because of course it is), and it even has an experimental antenna that should be able to track the keys if you can boost its signal strength high enough; there's a little meter that indicates the antenna's signal strength by displaying 0-50 *stars*.

Your instincts tell you that in order to save Christmas, you'll need to get all *fifty stars* by December 25th.

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!

As the submarine drops below the surface of the ocean, it automatically performs a sonar sweep of the nearby sea floor. On a small screen, the sonar sweep report (your puzzle input) appears: each line is a measurement of the sea floor depth as the sweep looks further and further away from the submarine.

For example, suppose you had the following report:

```
199
200
208
210
200
207
240
269
260
263
```

This report indicates that, scanning outward from the submarine, the sonar sweep found depths of `199`, `200`, `208`, `210`, and so on.

The first order of business is to figure out how quickly the depth increases, just so you know what you're dealing with - you never know if the keys will get carried into deeper water by an ocean current or a fish or something.

To do this, count *the number of times a depth measurement increases* from the previous measurement. (There is no measurement before the first measurement.) In the example above, the changes are as follows:

```
199 (N/A - no previous measurement)
200 (increased)
208 (increased)
```

```
210 (increased)
200 (decreased)
207 (increased)
240 (increased)
269 (increased)
260 (decreased)
263 (increased)
```

In this example, there are `7` measurements that are larger than the previous measurement.

How many measurements are larger than the previous measurement?

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

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

--- Day 2: Dive! ---

Now, you need to figure out how to pilot this thing.

It seems like the submarine can take a series of commands like `forward 1`, `down 2`, or `up 3`:

- `forward X` increases the horizontal position by `X` units.
- `down X` *increases* the depth by `X` units.
- `up X` *decreases* the depth by `X` units.

Note that since you're on a submarine, `down` and `up` affect your *depth*, and so they have the opposite result of what you might expect.

The submarine seems to already have a planned course (your puzzle input). You should probably figure out where it's going. For example:

```
forward 5
down 5
forward 8
up 3
down 8
forward 2
```

Your horizontal position and depth both start at `0`. The steps above would then modify them as follows:

- `forward 5` adds `5` to your horizontal position, a total of `5`.
- `down 5` adds `5` to your depth, resulting in a value of `5`.
- `forward 8` adds `8` to your horizontal position, a total of `13`.
- `up 3` decreases your depth by `3`, resulting in a value of `2`.
- `down 8` adds `8` to your depth, resulting in a value of `10`.
- `forward 2` adds `2` to your horizontal position, a total of `15`.

After following these instructions, you would have a horizontal position of `15` and a depth of `10`. (Multiplying these together produces `150`.)

Calculate the horizontal position and depth you would have after following the planned course. *What do you get if you multiply your final horizontal position by your final depth?*

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

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

--- Day 3: Binary Diagnostic ---

The submarine has been making some odd creaking noises, so you ask it to produce a diagnostic report just in case.

The diagnostic report (your puzzle input) consists of a list of binary numbers which, when decoded properly, can tell you many useful things about the conditions of the submarine. The first parameter to check is the *power consumption*.

You need to use the binary numbers in the diagnostic report to generate two new binary numbers (called the *gamma rate* and the *epsilon rate*). The power consumption can then be found by multiplying the gamma rate by the epsilon rate.

Each bit in the gamma rate can be determined by finding the *most common bit in the corresponding position* of all numbers in the diagnostic report. For example, given the following diagnostic report:

```
00100
11110
10110
10111
10101
01111
00111
11100
10000
11001
00010
01010
```

Considering only the first bit of each number, there are five `0` bits and seven `1` bits. Since the most common bit is `1`, the first bit of the gamma rate is `1`.

The most common second bit of the numbers in the diagnostic report is `0`, so the second bit of the gamma rate is `0`.

The most common value of the third, fourth, and fifth bits are `1`, `1`, and `0`, respectively, and so the final three bits of the gamma rate are `110`.

So, the gamma rate is the binary number `10110`, or `22` in decimal.

The epsilon rate is calculated in a similar way; rather than use the most common bit, the least common bit from each position is used. So, the epsilon rate is `01001`, or `9` in decimal. Multiplying the gamma rate (`22`) by the epsilon rate (`9`) produces the power consumption, `198`.

Use the binary numbers in your diagnostic report to calculate the gamma rate and epsilon rate, then multiply them together. *What is the power consumption of the submarine?* (Be sure to represent your answer in decimal, not binary.)

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

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

--- Day 4: Giant Squid ---

You're already almost 1.5km (almost a mile) below the surface of the ocean, already so deep that you can't see any sunlight. What you *can* see, however, is a giant squid that has attached itself to the outside of your submarine.

Maybe it wants to play bingo?

Bingo is played on a set of boards each consisting of a 5x5 grid of numbers. Numbers are chosen at random, and the chosen number is *marked* on all boards on which it appears. (Numbers may not appear on all boards.) If all numbers in any row or any column of a board are marked, that board *wins*. (Diagonals don't count.)

The submarine has a *bingo subsystem* to help passengers (currently, you and the giant squid) pass the time. It automatically generates a random order in which to draw numbers and a random set of boards (your puzzle input). For example:

```
7,4,9,5,11,17,23,2,0,14,21,24,10,16,13,6,15,25,12,22,18,20,8,19,3,26,1

22 13 17 11  0
 8  2 23  4 24
21  9 14 16  7
 6 10  3 18  5
 1 12 20 15 19

 3 15  0  2 22
 9 18 13 17  5
19  8  7 25 23
20 11 10 24  4
14 21 16 12  6

14 21 17 24  4
10 16 15  9 19
18  8 23 26 20
```

```
22 11 13  6  5
 2  0 12  3  7
```

After the first five numbers are drawn (`7`, `4`, `9`, `5`, and `11`), there are no winners, but the boards are marked as follows (shown here adjacent to each other to save space):

```
22 13 17 11  0         3 15  0  2 22        14 21 17 24  4
 8  2 23  4 24         9 18 13 17  5        10 16 15  9 19
21  9 14 16  7        19  8  7 25 23        18  8 23 26 20
 6 10  3 18  5        20 11 10 24  4        22 11 13  6  5
 1 12 20 15 19        14 21 16 12  6         2  0 12  3  7
```

After the next six numbers are drawn (`17`, `23`, `2`, `0`, `14`, and `21`), there are still no winners:

```
22 13 17 11  0         3 15  0  2 22        14 21 17 24  4
 8  2 23  4 24         9 18 13 17  5        10 16 15  9 19
21  9 14 16  7        19  8  7 25 23        18  8 23 26 20
 6 10  3 18  5        20 11 10 24  4        22 11 13  6  5
 1 12 20 15 19        14 21 16 12  6         2  0 12  3  7
```

Finally, `24` is drawn:

```
22 13 17 11  0         3 15  0  2 22        14 21 17 24  4
 8  2 23  4 24         9 18 13 17  5        10 16 15  9 19
21  9 14 16  7        19  8  7 25 23        18  8 23 26 20
 6 10  3 18  5        20 11 10 24  4        22 11 13  6  5
 1 12 20 15 19        14 21 16 12  6         2  0 12  3  7
```

At this point, the third board *wins* because it has at least one complete row or column of marked numbers (in this case, the entire top row is marked: `14 21 17 24  4`).

The *score* of the winning board can now be calculated. Start by finding the *sum of all unmarked numbers* on that board; in this case, the sum is `188`. Then, multiply that sum by *the number that was just called* when the board won, `24`, to get the final score, `188 * 24 = 4512`.

To guarantee victory against the giant squid, figure out which board will win first. *What will your final score be if you choose that board?*

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

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

--- Day 5: Hydrothermal Venture ---

You come across a field of hydrothermal vents on the ocean floor! These vents constantly produce large, opaque clouds, so it would be best to avoid them if possible.

They tend to form in *lines*; the submarine helpfully produces a list of nearby lines of vents (your puzzle input) for you to review. For example:

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

Each line of vents is given as a line segment in the format `x1,y1 -> x2,y2` where `x1`,`y1` are the coordinates of one end the line segment and `x2`,`y2` are the coordinates of the other end. These line segments include the points at both ends. In other words:

- An entry like `1,1 -> 1,3` covers points `1,1`, `1,2`, and `1,3`.
- An entry like `9,7 -> 7,7` covers points `9,7`, `8,7`, and `7,7`.

For now, *only consider horizontal and vertical lines*: lines where either `x1 = x2` or `y1 = y2`.

So, the horizontal and vertical lines from the above list would produce the following diagram:

```
.......1..
..1....1..
..1....1..
.......1..
.112111211
..........
..........
..........
..........
222111....
```

In this diagram, the top left corner is `0,0` and the bottom right corner is `9,9`. Each position is shown as *the number of lines which cover that point* or `.` if no line covers that point. The top-left pair of `1`s, for example, comes from `2,2 -> 2,1`; the very bottom row is formed by the overlapping lines `0,9 -> 5,9` and `0,9 -> 2,9`.

To avoid the most dangerous areas, you need to determine *the number of points where at least two lines overlap*. In the above example, this is anywhere in the diagram with a `2` or larger - a total of `5` points.

Consider only horizontal and vertical lines. *At how many points do at least two*

lines overlap?

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

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

--- Day 6: Lanternfish ---

The sea floor is getting steeper. Maybe the sleigh keys got carried this way?

A massive school of glowing lanternfish swims past. They must spawn quickly to reach such large numbers - maybe *exponentially* quickly? You should model their growth rate to be sure.

Although you know nothing about this specific species of lanternfish, you make some guesses about their attributes. Surely, each lanternfish creates a new lanternfish once every 7 days.

However, this process isn't necessarily synchronized between every lanternfish - one lanternfish might have 2 days left until it creates another lanternfish, while another might have 4. So, you can model each fish as a single number that represents *the number of days until it creates a new lanternfish.*

Furthermore, you reason, a *new* lanternfish would surely need slightly longer before it's capable of producing more lanternfish: two more days for its first cycle.

So, suppose you have a lanternfish with an internal timer value of `3`:

- After one day, its internal timer would become `2`.
- After another day, its internal timer would become `1`.
- After another day, its internal timer would become `0`.
- After another day, its internal timer would reset to `6`, and it would create a *new* lanternfish with an internal timer of `8`.
- After another day, the first lanternfish would have an internal timer of `5`, and the second lanternfish would have an internal timer of `7`.

A lanternfish that creates a new fish resets its timer to `6`, *not* `7` (because `0` is included as a valid timer value). The new lanternfish starts with an internal timer of `8` and does not start counting down until the next day.

Realizing what you're trying to do, the submarine automatically produces a list of the ages of several hundred nearby lanternfish (your puzzle input). For example, suppose you were given the following list:

```
3,4,3,1,2
```

This list means that the first fish has an internal timer of `3`, the second fish has an internal timer of `4`, and so on until the fifth fish, which has an internal timer of `2`. Simulating these fish over several days would proceed as follows:

```
Initial state: 3,4,3,1,2
```

```
After  1 day:  2,3,2,0,1
After  2 days: 1,2,1,6,0,8
After  3 days: 0,1,0,5,6,7,8
After  4 days: 6,0,6,4,5,6,7,8,8
After  5 days: 5,6,5,3,4,5,6,7,7,8
After  6 days: 4,5,4,2,3,4,5,6,6,7
After  7 days: 3,4,3,1,2,3,4,5,5,6
After  8 days: 2,3,2,0,1,2,3,4,4,5
After  9 days: 1,2,1,6,0,1,2,3,3,4,8
After 10 days: 0,1,0,5,6,0,1,2,2,3,7,8
After 11 days: 6,0,6,4,5,6,0,1,1,2,6,7,8,8,8
After 12 days: 5,6,5,3,4,5,6,0,0,1,5,6,7,7,7,8,8
After 13 days: 4,5,4,2,3,4,5,6,6,0,4,5,6,6,6,7,7,8,8
After 14 days: 3,4,3,1,2,3,4,5,5,6,3,4,5,5,5,6,6,7,7,8
After 15 days: 2,3,2,0,1,2,3,4,4,5,2,3,4,4,4,5,5,6,6,7
After 16 days: 1,2,1,6,0,1,2,3,3,4,1,2,3,3,3,4,4,5,5,6,8
After 17 days: 0,1,0,5,6,0,1,2,2,3,0,1,2,2,2,3,3,4,4,5,7,8
After 18 days: 6,0,6,4,5,6,0,1,1,2,6,0,1,1,1,2,2,3,3,4,6,7,8,8,8,8
```

Each day, a `0` becomes a `6` and adds a new `8` to the end of the list, while each other number decreases by 1 if it was present at the start of the day.

In this example, after 18 days, there are a total of `26` fish. After 80 days, there would be a total of `5934`.

Find a way to simulate lanternfish. *How many lanternfish would there be after 80 days?*

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

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

--- Day 7: The Treachery of Whales ---

A giant whale has decided your submarine is its next meal, and it's much faster than you are. There's nowhere to run!

Suddenly, a swarm of crabs (each in its own tiny submarine - it's too deep for them otherwise) zooms in to rescue you! They seem to be preparing to blast a hole in the ocean floor; sensors indicate a *massive underground cave system* just beyond where they're aiming!

The crab submarines all need to be aligned before they'll have enough power to blast a large enough hole for your submarine to get through. However, it doesn't look like they'll be aligned before the whale catches you! Maybe you can help?

There's one major catch - crab submarines can only move horizontally.

You quickly make a list of *the horizontal position of each crab* (your puzzle input). Crab submarines have limited fuel, so you need to find a way to make

all of their horizontal positions match while requiring them to spend as little fuel as possible.

For example, consider the following horizontal positions:

```
16,1,2,0,4,2,7,1,2,14
```

This means there's a crab with horizontal position `16`, a crab with horizontal position `1`, and so on.

Each change of 1 step in horizontal position of a single crab costs 1 fuel. You could choose any horizontal position to align them all on, but the one that costs the least fuel is horizontal position `2`:

- Move from `16` to `2`: `14` fuel
- Move from `1` to `2`: `1` fuel
- Move from `2` to `2`: `0` fuel
- Move from `0` to `2`: `2` fuel
- Move from `4` to `2`: `2` fuel
- Move from `2` to `2`: `0` fuel
- Move from `7` to `2`: `5` fuel
- Move from `1` to `2`: `1` fuel
- Move from `2` to `2`: `0` fuel
- Move from `14` to `2`: `12` fuel

This costs a total of `37` fuel. This is the cheapest possible outcome; more expensive outcomes include aligning at position `1` (`41` fuel), position `3` (`39` fuel), or position `10` (`71` fuel).

Determine the horizontal position that the crabs can align to using the least fuel possible. *How much fuel must they spend to align to that position?*

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

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

--- Day 8: Seven Segment Search ---

You barely reach the safety of the cave when the whale smashes into the cave mouth, collapsing it. Sensors indicate another exit to this cave at a much greater depth, so you have no choice but to press on.

As your submarine slowly makes its way through the cave system, you notice that the four-digit seven-segment displays in your submarine are malfunctioning; they must have been damaged during the escape. You'll be in a lot of trouble without them, so you'd better figure out what's wrong.

Each digit of a seven-segment display is rendered by turning on or off any of seven segments named `a` through `g`:

```
  0:      1:      2:      3:      4:
 aaaa    ....    aaaa    aaaa    ....
```

```
b    c  .    c  .    c  .    c  b    c
b    c  .    c  .    c  .    c  b    c
 ....    ....    dddd    dddd    dddd
e    f  .    f  e    .  .    f  .    f
e    f  .    f  e    .  .    f  .    f
 gggg    ....    gggg    gggg    ....

  5:      6:      7:      8:      9:
 aaaa    aaaa    aaaa    aaaa    aaaa
b    .  b    .  .    c  b    c  b    c
b    .  b    .  .    c  b    c  b    c
 dddd    dddd    ....    dddd    dddd
.    f  e    f  .    f  e    f  .    f
.    f  e    f  .    f  e    f  .    f
 gggg    gggg    ....    gggg    gggg
```

So, to render a `1`, only segments `c` and `f` would be turned on; the rest would be off. To render a `7`, only segments `a`, `c`, and `f` would be turned on.

The problem is that the signals which control the segments have been mixed up on each display. The submarine is still trying to display numbers by producing output on signal wires `a` through `g`, but those wires are connected to segments *randomly*. Worse, the wire/segment connections are mixed up separately for each four-digit display! (All of the digits *within* a display use the same connections, though.)

So, you might know that only signal wires `b` and `g` are turned on, but that doesn't mean *segments* `b` and `g` are turned on: the only digit that uses two segments is `1`, so it must mean segments `c` and `f` are meant to be on. With just that information, you still can't tell which wire (`b`/`g`) goes to which segment (`c`/`f`). For that, you'll need to collect more information.

For each display, you watch the changing signals for a while, make a note of *all ten unique signal patterns* you see, and then write down a single *four digit output value* (your puzzle input). Using the signal patterns, you should be able to work out which pattern corresponds to which digit.

For example, here is what you might see in a single entry in your notes:

```
acedgfb cdfbe gcdfa fbcad dab cefabd cdfgeb eafb cagedb ab |
cdfeb fcadb cdfeb cdbaf
```

(The entry is wrapped here to two lines so it fits; in your notes, it will all be on a single line.)

Each entry consists of ten *unique signal patterns*, a `|` delimiter, and finally the *four digit output value*. Within an entry, the same wire/segment connections are used (but you don't know what the connections actually are). The unique signal patterns correspond to the ten different ways the submarine tries to render a digit using the current wire/segment connections. Because `7` is the only digit

that uses three segments, `dab` in the above example means that to render a `7`, signal lines `d`, `a`, and `b` are on. Because `4` is the only digit that uses four segments, `eafb` means that to render a `4`, signal lines `e`, `a`, `f`, and `b` are on.

Using this information, you should be able to work out which combination of signal wires corresponds to each of the ten digits. Then, you can decode the four digit output value. Unfortunately, in the above example, all of the digits in the output value (`cdfeb fcadb cdfeb cdbaf`) use five segments and are more difficult to deduce.

For now, *focus on the easy digits*. Consider this larger example:

```
be cfbegad cbdgef fgaecd cgeb fdcge agebfd fecdb fabcd edb |
fdgacbe cefdb cefbgd gcbe
edbfga begcd cbg gc gcadebf fbgde acbgfd abcde gfcbed gfec |
fcgedb cgb dgebacf gc
fgaebd cg bdaec gdafb agbcfd gdcbef bgcad gfac gcb cdgabef |
cg cg fdcagb cbg
fbegcd cbd adcefb dageb afcb bc aefdc ecdab fgdeca fcdbega |
efabcd cedba gadfec cb
aecbfdg fbg gf bafeg dbefa fcge gcbea fcaegb dgceab fcbdga |
gecf egdcabf bgf bfgea
fgeab ca afcebg bdacfeg cfaedg gcfdb baec bfadeg bafgc acf |
gebdcfa ecba ca fadegcb
dbcfg fgd bdegcaf fgec aegbdf ecdfab fbedc dacgb gdcebf gf |
cefg dcbef fcge gbcadfe
bdfegc cbegaf gecbf dfcage bdacg ed bedf ced adcbefg gebcd |
ed bcgafe cdgba cbgef
egadfb cdbfeg cegd fecab cgb gbdefca cg fgcdab egfdb bfceg |
gbdfcae bgc cg cgb
gcafb gcf dcaebfg ecagb gf abcdeg gaef cafbge fdbac fegbdc |
fgae cfgab fg bagce
```

Because the digits `1`, `4`, `7`, and `8` each use a unique number of segments, you should be able to tell which combinations of signals correspond to those digits. Counting *only digits in the output values* (the part after `|` on each line), in the above example, there are `26` instances of digits that use a unique number of segments (highlighted above).

In the output values, how many times do digits `1`, `4`, `7`, or `8` appear?

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

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

--- Day 9: Smoke Basin ---

These caves seem to be lava tubes. Parts are even still volcanically active; small hydrothermal vents release smoke into the caves that slowly settles like rain.

If you can model how the smoke flows through the caves, you might be able to avoid it and be that much safer. The submarine generates a heightmap of the floor of the nearby caves for you (your puzzle input).

Smoke flows to the lowest point of the area it's in. For example, consider the following heightmap:

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

Each number corresponds to the height of a particular location, where `9` is the highest and `0` is the lowest a location can be.

Your first goal is to find the *low points* - the locations that are lower than any of its adjacent locations. Most locations have four adjacent locations (up, down, left, and right); locations on the edge or corner of the map have three or two adjacent locations, respectively. (Diagonal locations do not count as adjacent.)

In the above example, there are *four* low points, all highlighted: two are in the first row (a `1` and a `0`), one is in the third row (a `5`), and one is in the bottom row (also a `5`). All other locations on the heightmap have some lower adjacent location, and so are not low points.

The *risk level* of a low point is *1 plus its height*. In the above example, the risk levels of the low points are `2`, `1`, `6`, and `6`. The sum of the risk levels of all low points in the heightmap is therefore `15`.

Find all of the low points on your heightmap. *What is the sum of the risk levels of all low points on your heightmap?*

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

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

--- Day 10: Syntax Scoring ---

You ask the submarine to determine the best route out of the deep-sea cave, but it only replies:

```
Syntax error in navigation subsystem on line: all of them
```

All of them?! The damage is worse than you thought. You bring up a copy of the navigation subsystem (your puzzle input).

The navigation subsystem syntax is made of several lines containing *chunks*. There are one or more chunks on each line, and chunks contain zero or more other chunks. Adjacent chunks are not separated by any delimiter; if one chunk stops, the next chunk (if any) can immediately start. Every chunk must *open* and *close* with one of four legal pairs of matching characters:

- If a chunk opens with `(`, it must close with `)`.
- If a chunk opens with `[`, it must close with `]`.
- If a chunk opens with `{`, it must close with `}`.
- If a chunk opens with `<`, it must close with `>`.

So, `()` is a legal chunk that contains no other chunks, as is `[]`. More complex but valid chunks include `([])`, `{()()()}`, `<([{}])>`, `[<>({}){}[([])<>]]`, and even `((((((((((()))))))))))`.

Some lines are *incomplete*, but others are *corrupted*. Find and discard the corrupted lines first.

A corrupted line is one where a chunk *closes with the wrong character* - that is, where the characters it opens and closes with do not form one of the four legal pairs listed above.

Examples of corrupted chunks include `(]`, `{()()()>`, `(((()))}`, and `<([]){()}[{}])`. Such a chunk can appear anywhere within a line, and its presence causes the whole line to be considered corrupted.

For example, consider the following navigation subsystem:

```
[({(<(())[]>[[{[]{<()<>>
[()[<>])](({[<{<<[]>>(
{([(<{}[<>[]}>{[]{[(<()>
((((({<>}<{<{<>}{[]{[]{}
[[<[([]))<([[{}[[()]]]
[{[{({}]{}}([{[{{{}}([]
{<[[]]>}<{[{{{[]{()[[[]
[<(<(<(<{}))><([]([]()
<{([([[(<>()){}]>(<<{{
<{([{{}}[<[[[<>{}]]]>[]]
```

Some of the lines aren't corrupted, just incomplete; you can ignore these lines for now. The remaining five lines are corrupted:

- `{([(<{}[<>[]}>{[]{[(<()>` - Expected `]`, but found `}` instead.
- `[[<[([]))<([[{}[[()]]]` - Expected `]`, but found `)` instead.
- `[{[{({}]{}}([{[{{{}}([]` - Expected `)`, but found `]` instead.
- `[<(<(<(<{}))><([]([]()` - Expected `>`, but found `)` instead.
- `<{([([[(<>()){}]>(<<{{` - Expected `]`, but found `>` instead.

Stop at the first incorrect closing character on each corrupted line.

Did you know that syntax checkers actually have contests to see who can get the high score for syntax errors in a file? It's true! To calculate the syntax error score for a line, take the *first illegal character* on the line and look it up in the following table:

- `)`: `3` points.
- `]`: `57` points.

- `}`: `1197` points.
- `>`: `25137` points.

In the above example, an illegal `)` was found twice (`2*3 = 6` points), an illegal `]` was found once (`57` points), an illegal `}` was found once (`1197` points), and an illegal `>` was found once (`25137` points). So, the total syntax error score for this file is `6+57+1197+25137 = 26397` points!

Find the first illegal character in each corrupted line of the navigation subsystem. *What is the total syntax error score for those errors?*

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

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

--- Day 11: Dumbo Octopus ---

You enter a large cavern full of rare bioluminescent dumbo octopuses! They seem to not like the Christmas lights on your submarine, so you turn them off for now.

There are 100 octopuses arranged neatly in a 10 by 10 grid. Each octopus slowly gains *energy* over time and *flashes* brightly for a moment when its energy is full. Although your lights are off, maybe you could navigate through the cave without disturbing the octopuses if you could predict when the flashes of light will happen.

Each octopus has an *energy level* - your submarine can remotely measure the energy level of each octopus (your puzzle input). For example:

```
5483143223
2745854711
5264556173
6141336146
6357385478
4167524645
2176841721
6882881134
4846848554
5283751526
```

The energy level of each octopus is a value between `0` and `9`. Here, the top-left octopus has an energy level of `5`, the bottom-right one has an energy level of `6`, and so on.

You can model the energy levels and flashes of light in *steps*. During a single step, the following occurs:

- First, the energy level of each octopus increases by `1`.
- Then, any octopus with an energy level greater than `9` *flashes*. This increases the energy level of all adjacent octopuses by `1`, including octopuses

that are diagonally adjacent. If this causes an octopus to have an energy level greater than `9`, it *also flashes*. This process continues as long as new octopuses keep having their energy level increased beyond `9`. (An octopus can only flash *at most once per step*.)
- Finally, any octopus that flashed during this step has its energy level set to `0`, as it used all of its energy to flash.

Adjacent flashes can cause an octopus to flash on a step even if it begins that step with very little energy. Consider the middle octopus with `1` energy in this situation:

```
Before any steps:
11111
19991
19191
19991
11111

After step 1:
34543
40004
50005
40004
34543

After step 2:
45654
51115
61116
51115
45654
```

An octopus is *highlighted* when it flashed during the given step.

Here is how the larger example above progresses:

```
Before any steps:
5483143223
2745854711
5264556173
6141336146
6357385478
4167524645
2176841721
6882881134
4846848554
5283751526
```

```
After step 1:
6594254334
3856965822
6375667284
7252447257
7468496589
5278635756
3287952832
7993992245
5957959665
6394862637

After step 2:
8807476555
5089087054
8597889608
8485769600
8700908800
6600088989
6800005943
0000007456
9000000876
8700006848

After step 3:
0050900866
8500800575
9900000039
9700000041
9935080063
7712300000
7911250009
2211130000
0421125000
0021119000

After step 4:
2263031977
0923031697
0032221150
0041111163
0076191174
0053411122
0042361120
5532241122
1532247211
```

```
1132230211

After step 5:
4484144000
2044144000
2253333493
1152333274
1187303285
1164633233
1153472231
6643352233
2643358322
2243341322

After step 6:
5595255111
3155255222
3364444605
2263444496
2298414396
2275744344
2264583342
7754463344
3754469433
3354452433

After step 7:
6707366222
4377366333
4475555827
3496655709
3500625609
3509955566
3486694453
8865585555
4865580644
4465574644

After step 8:
7818477333
5488477444
5697666949
4608766830
4734946730
4740097688
6900007564
```

```
0000009666
8000004755
6800007755

After step 9:
9060000644
7800000976
6900000080
5840000082
5858000093
6962400000
8021250009
2221130009
9111128097
7911119976

After step 10:
0481112976
0031112009
0041112504
0081111406
0099111306
0093511233
0442361130
5532252350
0532250600
0032240000
```

After step 10, there have been a total of 204 flashes. Fast forwarding, here is the same configuration every 10 steps:

```
After step 20:
3936556452
5686556806
4496555690
4448655580
4456865570
5680086577
7000009896
0000000344
6000000364
4600009543

After step 30:
0643334118
4253334611
3374333458
```

```
2225333337
2229333338
2276733333
2754574565
5544458511
9444447111
7944446119

After step 40:
6211111981
0421111119
0042111115
0003111115
0003111116
0065611111
0532351111
3322234597
2222222976
2222222762

After step 50:
9655556447
4865556805
4486555690
4458655580
4574865570
5700086566
6000009887
8000000533
6800000633
5680000538

After step 60:
2533334200
2743334640
2264333458
2225333337
2225333338
2287833333
3854573455
1854458611
1175447111
1115446111

After step 70:
8211111164
```

```
0421111166
0042111114
0004211115
0000211116
0065611111
0532351111
7322235117
5722223475
4572222754

After step 80:
1755555697
5965555609
4486555680
4458655580
4570865570
5700086566
7000008666
0000000990
0000000800
0000000000

After step 90:
7433333522
2643333522
2264333458
2226433337
2222433338
2287833333
2854573333
4854458333
3387779333
3333333333

After step 100:
0397666866
0749766918
0053976933
0004297822
0004229892
0053222877
0532222966
9322228966
7922286866
6789998766
```

After 100 steps, there have been a total of `1656` flashes.

Given the starting energy levels of the dumbo octopuses in your cavern, simulate 100 steps. *How many total flashes are there after 100 steps?*

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

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

--- Day 12: Passage Pathing ---

With your submarine's subterranean subsystems subsisting suboptimally, the only way you're getting out of this cave anytime soon is by finding a path yourself. Not just *a* path - the only way to know if you've found the *best* path is to find *all* of them.

Fortunately, the sensors are still mostly working, and so you build a rough map of the remaining caves (your puzzle input). For example:

```
start-A
start-b
A-c
A-b
b-d
A-end
b-end
```

This is a list of how all of the caves are connected. You start in the cave named `start`, and your destination is the cave named `end`. An entry like `b-d` means that cave `b` is connected to cave `d` - that is, you can move between them.

So, the above cave system looks roughly like this:

```
    start
    /   \
c--A-----b--d
    \   /
     end
```

Your goal is to find the number of distinct *paths* that start at `start`, end at `end`, and don't visit small caves more than once. There are two types of caves: *big* caves (written in uppercase, like `A`) and *small* caves (written in lowercase, like `b`). It would be a waste of time to visit any small cave more than once, but big caves are large enough that it might be worth visiting them multiple times. So, all paths you find should *visit small caves at most once*, and can *visit big caves any number of times.*

Given these rules, there are `10` paths through this example cave system:

```
start,A,b,A,c,A,end
start,A,b,A,end
```

```
start,A,b,end
start,A,c,A,b,A,end
start,A,c,A,b,end
start,A,c,A,end
start,A,end
start,b,A,c,A,end
start,b,A,end
start,b,end
```

(Each line in the above list corresponds to a single path; the caves visited by that path are listed in the order they are visited and separated by commas.)

Note that in this cave system, cave `d` is never visited by any path: to do so, cave `b` would need to be visited twice (once on the way to cave `d` and a second time when returning from cave `d`), and since cave `b` is small, this is not allowed.

Here is a slightly larger example:

```
dc-end
HN-start
start-kj
dc-start
dc-HN
LN-dc
HN-end
kj-sa
kj-HN
kj-dc
```

The `19` paths through it are as follows:

```
start,HN,dc,HN,end
start,HN,dc,HN,kj,HN,end
start,HN,dc,end
start,HN,dc,kj,HN,end
start,HN,end
start,HN,kj,HN,dc,HN,end
start,HN,kj,HN,dc,end
start,HN,kj,HN,end
start,HN,kj,dc,HN,end
start,HN,kj,dc,end
start,dc,HN,end
start,dc,HN,kj,HN,end
start,dc,end
start,dc,kj,HN,end
start,kj,HN,dc,HN,end
start,kj,HN,dc,end
start,kj,HN,end
start,kj,dc,HN,end
```

```
start,kj,dc,end
```

Finally, this even larger example has 226 paths through it:

```
fs-end
he-DX
fs-he
start-DX
pj-DX
end-zg
zg-sl
zg-pj
pj-he
RW-he
fs-DX
pj-RW
zg-RW
start-pj
he-WI
zg-he
pj-fs
start-RW
```

How many paths through this cave system are there that visit small caves at most once?

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

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

--- Day 13: Transparent Origami ---

You reach another volcanically active part of the cave. It would be nice if you could do some kind of thermal imaging so you could tell ahead of time which caves are too hot to safely enter.

Fortunately, the submarine seems to be equipped with a thermal camera! When you activate it, you are greeted with:

```
Congratulations on your purchase! To activate this infrared thermal imaging
camera system, please enter the code found on page 1 of the manual.
```

Apparently, the Elves have never used this feature. To your surprise, you manage to find the manual; as you go to open it, page 1 falls out. It's a large sheet of transparent paper! The transparent paper is marked with random dots and includes instructions on how to fold it up (your puzzle input). For example:

```
6,10
0,14
9,10
```

```
0,3
10,4
4,11
6,0
6,12
4,1
0,13
10,12
3,4
3,0
8,4
1,10
2,14
8,10
9,0

fold along y=7
fold along x=5
```

The first section is a list of dots on the transparent paper. `0,0` represents the top-left coordinate. The first value, `x`, increases to the right. The second value, `y`, increases downward. So, the coordinate `3,0` is to the right of `0,0`, and the coordinate `0,7` is below `0,0`. The coordinates in this example form the following pattern, where `#` is a dot on the paper and `.` is an empty, unmarked position:

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

Then, there is a list of *fold instructions*. Each instruction indicates a line on the transparent paper and wants you to fold the paper *up* (for horizontal `y=...` lines) or *left* (for vertical `x=...` lines). In this example, the first fold instruction is `fold along y=7`, which designates the line formed by all of the positions where `y` is `7` (marked here with `-`):

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

```
....#.....
..........
#.........
...#....#.#
..........
..........
----------
..........
..........
.#....#.##.
....#.....
......#...#
#.........
#.#.......
```

Because this is a horizontal line, fold the bottom half *up*. Some of the dots might end up overlapping after the fold is complete, but dots will never appear exactly on a fold line. The result of doing this fold looks like this:

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

Now, only `17` dots are visible.

Notice, for example, the two dots in the bottom left corner before the transparent paper is folded; after the fold is complete, those dots appear in the top left corner (at `0,0` and `0,1`). Because the paper is transparent, the dot just below them in the result (at `0,3`) remains visible, as it can be seen through the transparent paper.

Also notice that some dots can end up *overlapping*; in this case, the dots merge together and become a single dot.

The second fold instruction is `fold along x=5`, which indicates this line:

```
#.##.|#..#.
#...#|.....
.....|#...#
#...#|.....
.#.#.|#.###
.....|.....
.....|.....
```

Because this is a vertical line, fold *left*:

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

The instructions made a square!

The transparent paper is pretty big, so for now, focus on just completing the first fold. After the first fold in the example above, `17` dots are visible - dots that end up overlapping after the fold is completed count as a single dot.

How many dots are visible after completing just the first fold instruction on your transparent paper?

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

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

--- Day 14: Extended Polymerization ---

The incredible pressures at this depth are starting to put a strain on your submarine. The submarine has polymerization equipment that would produce suitable materials to reinforce the submarine, and the nearby volcanically-active caves should even have the necessary input elements in sufficient quantities.

The submarine manual contains instructions for finding the optimal polymer formula; specifically, it offers a *polymer template* and a list of *pair insertion* rules (your puzzle input). You just need to work out what polymer would result after repeating the pair insertion process a few times.

For example:

```
NNCB

CH -> B
HH -> N
CB -> H
NH -> C
HB -> C
HC -> B
HN -> C
NN -> C
BH -> H
NC -> B
NB -> B
BN -> B
BB -> N
```

```
BC -> B
CC -> N
CN -> C
```

The first line is the *polymer template* - this is the starting point of the process.

The following section defines the *pair insertion* rules. A rule like `AB -> C` means that when elements `A` and `B` are immediately adjacent, element `C` should be inserted between them. These insertions all happen simultaneously.

So, starting with the polymer template `NNCB`, the first step simultaneously considers all three pairs:

- The first pair (`NN`) matches the rule `NN -> C`, so element `C` is inserted between the first `N` and the second `N`.
- The second pair (`NC`) matches the rule `NC -> B`, so element `B` is inserted between the `N` and the `C`.
- The third pair (`CB`) matches the rule `CB -> H`, so element `H` is inserted between the `C` and the `B`.

Note that these pairs overlap: the second element of one pair is the first element of the next pair. Also, because all pairs are considered simultaneously, inserted elements are not considered to be part of a pair until the next step.

After the first step of this process, the polymer becomes `NCNBCHB`.

Here are the results of a few steps using the above rules:

```
Template:     NNCB
After step 1: NCNBCHB
After step 2: NBCCNBBBCBHCB
After step 3: NBBBCNCCNBBNBNBBCHBHHBCHB
After step 4: NBBNBNBBCCNBCNCCNBBNBBNBBBNBBNBBCBHCBHHNHCBBCBHCB
```

This polymer grows quickly. After step 5, it has length 97; After step 10, it has length 3073. After step 10, `B` occurs 1749 times, `C` occurs 298 times, `H` occurs 161 times, and `N` occurs 865 times; taking the quantity of the most common element (`B`, 1749) and subtracting the quantity of the least common element (`H`, 161) produces `1749 - 161 = 1588`.

Apply 10 steps of pair insertion to the polymer template and find the most and least common elements in the result. *What do you get if you take the quantity of the most common element and subtract the quantity of the least common element?*

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

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

--- Day 15: Chiton ---

You've almost reached the exit of the cave, but the walls are getting closer together. Your submarine can barely still fit, though; the main problem is that the walls of the cave are covered in chitons, and it would be best not to bump any of them.

The cavern is large, but has a very low ceiling, restricting your motion to two dimensions. The shape of the cavern resembles a square; a quick scan of chiton density produces a map of *risk level* throughout the cave (your puzzle input). For example:

```
1163751742
1381373672
2136511328
3694931569
7463417111
1319128137
1359912421
3125421639
1293138521
2311944581
```

You start in the top left position, your destination is the bottom right position, and you cannot move diagonally. The number at each position is its *risk level*; to determine the total risk of an entire path, add up the risk levels of each position you *enter* (that is, don't count the risk level of your starting position unless you enter it; leaving it adds no risk to your total).

Your goal is to find a path with the *lowest total risk*. In this example, a path with the lowest total risk is highlighted here:

```
1163751742
1381373672
2136511328
3694931569
7463417111
1319128137
1359912421
3125421639
1293138521
2311944581
```

The total risk of this path is `40` (the starting position is never entered, so its risk is not counted).

What is the lowest total risk of any path from the top left to the bottom right?

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

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

--- Day 16: Packet Decoder ---

As you leave the cave and reach open waters, you receive a transmission from the Elves back on the ship.

The transmission was sent using the Buoyancy Interchange Transmission System (BITS), a method of packing numeric expressions into a binary sequence. Your submarine's computer has saved the transmission in hexadecimal (your puzzle input).

The first step of decoding the message is to convert the hexadecimal representation into binary. Each character of hexadecimal corresponds to four bits of binary data:

```
0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111
```

The BITS transmission contains a single *packet* at its outermost layer which itself contains many other packets. The hexadecimal representation of this packet might encode a few extra `0` bits at the end; these are not part of the transmission and should be ignored.

Every packet begins with a standard header: the first three bits encode the packet *version*, and the next three bits encode the packet *type ID*. These two values are numbers; all numbers encoded in any packet are represented as binary with the most significant bit first. For example, a version encoded as the binary sequence `100` represents the number `4`.

Packets with type ID `4` represent a *literal value*. Literal value packets encode a single binary number. To do this, the binary number is padded with leading zeroes until its length is a multiple of four bits, and then it is broken into groups of four bits. Each group is prefixed by a `1` bit except the last group, which is

prefixed by a `0` bit. These groups of five bits immediately follow the packet header. For example, the hexadecimal string `D2FE28` becomes:

```
110100101111111000101000
VVVTTTAAAAABBBBBCCCCC
```

Below each bit is a label indicating its purpose:

- The three bits labeled `V` (`110`) are the packet version, `6`.
- The three bits labeled `T` (`100`) are the packet type ID, `4`, which means the packet is a literal value.
- The five bits labeled `A` (`10111`) start with a `1` (not the last group, keep reading) and contain the first four bits of the number, `0111`.
- The five bits labeled `B` (`11110`) start with a `1` (not the last group, keep reading) and contain four more bits of the number, `1110`.
- The five bits labeled `C` (`00101`) start with a `0` (last group, end of packet) and contain the last four bits of the number, `0101`.
- The three unlabeled `0` bits at the end are extra due to the hexadecimal representation and should be ignored.

So, this packet represents a literal value with binary representation `011111100101`, which is `2021` in decimal.

Every other type of packet (any packet with a type ID other than `4`) represent an *operator* that performs some calculation on one or more sub-packets contained within. Right now, the specific operations aren't important; focus on parsing the hierarchy of sub-packets.

An operator packet contains one or more packets. To indicate which subsequent binary data represents its sub-packets, an operator packet can use one of two modes indicated by the bit immediately after the packet header; this is called the *length type ID*:

- If the length type ID is `0`, then the next *15* bits are a number that represents the *total length in bits* of the sub-packets contained by this packet.
- If the length type ID is `1`, then the next *11* bits are a number that represents the *number of sub-packets immediately contained* by this packet.

Finally, after the length type ID bit and the 15-bit or 11-bit field, the sub-packets appear.

For example, here is an operator packet (hexadecimal string `38006F45291200`) with length type ID `0` that contains two sub-packets:

```
00111000000000000110111101000101001010010001001000000000
VVVTTTILLLLLLLLLLLLLLLAAAAAAAAAAABBBBBBBBBBBBBBBB
```

- The three bits labeled `V` (`001`) are the packet version, `1`.
- The three bits labeled `T` (`110`) are the packet type ID, `6`, which means the packet is an operator.

- The bit labeled `I` (`0`) is the length type ID, which indicates that the length is a 15-bit number representing the number of bits in the sub-packets.
- The 15 bits labeled `L` (`000000000011011`) contain the length of the sub-packets in bits, `27`.
- The 11 bits labeled `A` contain the first sub-packet, a literal value representing the number `10`.
- The 16 bits labeled `B` contain the second sub-packet, a literal value representing the number `20`.

After reading 11 and 16 bits of sub-packet data, the total length indicated in `L` (27) is reached, and so parsing of this packet stops.

As another example, here is an operator packet (hexadecimal string `EE00D40C823060`) with length type ID `1` that contains three sub-packets:

```
11101110000000001101010000001100100000100011000001100000
VVVTTTILLLLLLLLLLLAAAAAAAAAAABBBBBBBBBBBCCCCCCCCCCC
```

- The three bits labeled `V` (`111`) are the packet version, `7`.
- The three bits labeled `T` (`011`) are the packet type ID, `3`, which means the packet is an operator.
- The bit labeled `I` (`1`) is the length type ID, which indicates that the length is a 11-bit number representing the number of sub-packets.
- The 11 bits labeled `L` (`00000000011`) contain the number of sub-packets, `3`.
- The 11 bits labeled `A` contain the first sub-packet, a literal value representing the number `1`.
- The 11 bits labeled `B` contain the second sub-packet, a literal value representing the number `2`.
- The 11 bits labeled `C` contain the third sub-packet, a literal value representing the number `3`.

After reading 3 complete sub-packets, the number of sub-packets indicated in `L` (3) is reached, and so parsing of this packet stops.

For now, parse the hierarchy of the packets throughout the transmission and *add up all of the version numbers.*

Here are a few more examples of hexadecimal-encoded transmissions:

- `8A004A801A8002F478` represents an operator packet (version 4) which contains an operator packet (version 1) which contains an operator packet (version 5) which contains a literal value (version 6); this packet has a version sum of `16`.
- `620080001611562C8802118E34` represents an operator packet (version 3) which contains two sub-packets; each sub-packet is an operator packet that contains two literal values. This packet has a version sum of `12`.
- `C0015000016115A2E0802F182340` has the same structure as the previous example, but the outermost packet uses a different length type ID. This packet has a version sum of `23`.

- `A0016C880162017C3686B18A3D4780` is an operator packet that contains an operator packet that contains an operator packet that contains five literal values; it has a version sum of `31`.

Decode the structure of your hexadecimal-encoded BITS transmission; *what do you get if you add up the version numbers in all packets?*

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

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

--- Day 17: Trick Shot ---

You finally decode the Elves' message. `HI`, the message says. You continue searching for the sleigh keys.

Ahead of you is what appears to be a large ocean trench. Could the keys have fallen into it? You'd better send a probe to investigate.

The probe launcher on your submarine can fire the probe with any integer velocity in the `x` (forward) and `y` (upward, or downward if negative) directions. For example, an initial `x,y` velocity like `0,10` would fire the probe straight up, while an initial velocity like `10,-1` would fire the probe forward at a slight downward angle.

The probe's `x,y` position starts at `0,0`. Then, it will follow some trajectory by moving in *steps*. On each step, these changes occur in the following order:

- The probe's `x` position increases by its `x` velocity.
- The probe's `y` position increases by its `y` velocity.
- Due to drag, the probe's `x` velocity changes by `1` toward the value `0`; that is, it decreases by `1` if it is greater than `0`, increases by `1` if it is less than `0`, or does not change if it is already `0`.
- Due to gravity, the probe's `y` velocity decreases by `1`.

For the probe to successfully make it into the trench, the probe must be on some trajectory that causes it to be within a *target area* after any step. The submarine computer has already calculated this target area (your puzzle input). For example:

```
target area: x=20..30, y=-10..-5
```

This target area means that you need to find initial `x,y` velocity values such that after any step, the probe's `x` position is at least `20` and at most `30`, *and* the probe's `y` position is at least `-10` and at most `-5`.

Given this target area, one initial velocity that causes the probe to be within the target area after any step is `7,2`:

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

```
S........................#.....
...............................
...............................
............................#...
...............................
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTT#TT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
```

In this diagram, `S` is the probe's initial position, `0,0`. The `x` coordinate increases to the right, and the `y` coordinate increases upward. In the bottom right, positions that are within the target area are shown as `T`. After each step (until the target area is reached), the position of the probe is marked with `#`. (The bottom-right `#` is both a position the probe reaches and a position in the target area.)

Another initial velocity that causes the probe to be within the target area after any step is `6,3`:

```
...............#..#............
............#........#..........
...............................
......#..............#.........
...............................
...............................
S......................#.........
...............................
...............................
...............................
.......................#.........
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................T#TTTTTTTTT
.....................TTTTTTTTTTT
```

Another one is `9,0`:

```
S........#.....................
..................#............
...............................
........................#......
...............................
.....................TTTTTTTTTTT
```

```
.....................TTTTTTTTTT#
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
.....................TTTTTTTTTTT
```

One initial velocity that *doesn't* cause the probe to be within the target area after any step is `17,-4`:

```
S....................................................................
....................................................................
....................................................................
....................................................................
..................#.................................................
.....................TTTTTTTTTTT....................................
.....................TTTTTTTTTTT....................................
.....................TTTTTTTTTTT....................................
.....................TTTTTTTTTTT....................................
.....................TTTTTTTTTTT..#.................................
.....................TTTTTTTTTTT....................................
....................................................................
....................................................................
....................................................................
....................................................................
.............................................#......................
....................................................................
....................................................................
....................................................................
....................................................................
....................................................................
....................................................................
....................................................................
........................................................................#
```

The probe appears to pass through the target area, but is never within it after any step. Instead, it continues down and to the right - only the first few steps are shown.

If you're going to fire a highly scientific probe out of a super cool probe launcher, you might as well do it with *style*. How high can you make the probe go while still reaching the target area?

In the above example, using an initial velocity of `6,9` is the best you can do, causing the probe to reach a maximum `y` position of `45`. (Any higher initial `y` velocity causes the probe to overshoot the target area entirely.)

Find the initial velocity that causes the probe to reach the highest `y` position and still eventually be within the target area after any step. *What is the highest `y` position it reaches on this trajectory?*

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

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

--- Day 18: Snailfish ---

You descend into the ocean trench and encounter some snailfish. They say they saw the sleigh keys! They'll even tell you which direction the keys went if you help one of the smaller snailfish with his *math homework*.

Snailfish numbers aren't like regular numbers. Instead, every snailfish number is a *pair* - an ordered list of two elements. Each element of the pair can be either a regular number or another pair.

Pairs are written as `[x,y]`, where `x` and `y` are the elements within the pair. Here are some example snailfish numbers, one snailfish number per line:

```
[1,2]
[[1,2],3]
[9,[8,7]]
[[1,9],[8,5]]
[[[[1,2],[3,4]],[[5,6],[7,8]]],9]
[[[9,[3,8]],[[0,9],6]],[[[3,7],[4,9]],3]]
[[[[1,3],[5,3]],[[1,3],[8,7]]],[[[4,9],[6,9]],[[8,2],[7,3]]]]
```

This snailfish homework is about *addition*. To add two snailfish numbers, form a pair from the left and right parameters of the addition operator. For example, `[1,2]` + `[[3,4],5]` becomes `[[1,2],[[3,4],5]]`.

There's only one problem: *snailfish numbers must always be reduced*, and the process of adding two snailfish numbers can result in snailfish numbers that need to be reduced.

To *reduce a snailfish number*, you must repeatedly do the first action in this list that applies to the snailfish number:

- If any pair is *nested inside four pairs*, the leftmost such pair *explodes*.
- If any regular number is *10 or greater*, the leftmost such regular number *splits*.

Once no action in the above list applies, the snailfish number is reduced.

During reduction, at most one action applies, after which the process returns to the top of the list of actions. For example, if *split* produces a pair that meets the *explode* criteria, that pair *explodes* before other *splits* occur.

To *explode* a pair, the pair's left value is added to the first regular number to the left of the exploding pair (if any), and the pair's right value is added to the first regular number to the right of the exploding pair (if any). Exploding pairs will always consist of two regular numbers. Then, the entire exploding pair is replaced with the regular number `0`.

Here are some examples of a single explode action:

- `[[[[[9,8],1],2],3],4]` becomes `[[[[0,9],2],3],4]` (the `9` has no regular number to its left, so it is not added to any regular number).
- `[7,[6,[5,[4,[3,2]]]]]` becomes `[7,[6,[5,[7,0]]]]` (the `2` has no regular number to its right, and so it is not added to any regular number).
- `[[6,[5,[4,[3,2]]]],1]` becomes `[[6,[5,[7,0]]],3]`.
- `[[3,[2,[1,[7,3]]]],[6,[5,[4,[3,2]]]]]` becomes `[[3,[2,[8,0]]],[9,[5,[4,[3,2]]]]]` (the pair `[3,2]` is unaffected because the pair `[7,3]` is further to the left; `[3,2]` would explode on the next action).
- `[[3,[2,[8,0]]],[9,[5,[4,[3,2]]]]]` becomes `[[3,[2,[8,0]]],[9,[5,[7,0]]]]`.

To *split* a regular number, replace it with a pair; the left element of the pair should be the regular number divided by two and rounded *down*, while the right element of the pair should be the regular number divided by two and rounded *up*. For example, `10` becomes `[5,5]`, `11` becomes `[5,6]`, `12` becomes `[6,6]`, and so on.

Here is the process of finding the reduced result of `[[[[4,3],4],4],[7,[[8,4],9]]]` + `[1,1]`:

```
after addition: [[[[[4,3],4],4],[7,[[8,4],9]]],[1,1]]
after explode:  [[[[0,7],4],[7,[[8,4],9]]],[1,1]]
after explode:  [[[[0,7],4],[15,[0,13]]],[1,1]]
after split:    [[[[0,7],4],[[7,8],[0,13]]],[1,1]]
after split:    [[[[0,7],4],[[7,8],[0,[6,7]]]],[1,1]]
after explode:  [[[[0,7],4],[[7,8],[6,0]]],[8,1]]
```

Once no reduce actions apply, the snailfish number that remains is the actual result of the addition operation: `[[[[0,7],4],[[7,8],[6,0]]],[8,1]]`.

The homework assignment involves adding up a *list of snailfish numbers* (your puzzle input). The snailfish numbers are each listed on a separate line. Add the first snailfish number and the second, then add that result and the third, then add that result and the fourth, and so on until all numbers in the list have been used once.

For example, the final sum of this list is `[[[[1,1],[2,2]],[3,3]],[4,4]]`:

```
[1,1]
[2,2]
[3,3]
[4,4]
```

The final sum of this list is `[[[[3,0],[5,3]],[4,4]],[5,5]]`:

```
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
```

The final sum of this list is `[[[[5,0],[7,4]],[5,5]],[6,6]]`:

```
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
[6,6]
```

Here's a slightly larger example:

```
[[[0,[4,5]],[0,0]],[[[4,5],[2,6]],[9,5]]]
[7,[[[3,7],[4,3]],[[6,3],[8,8]]]]
[[2,[[0,8],[3,4]]],[[[6,7],1],[7,[1,6]]]]
[[[[2,4],7],[6,[0,5]]],[[[6,8],[2,8]],[[2,1],[4,5]]]]
[7,[5,[[3,8],[1,4]]]]
[[2,[2,2]],[8,[8,1]]]
[2,9]
[1,[[[9,3],9],[[9,0],[0,7]]]]
[[[5,[7,4]],7],1]
[[[[4,2],2],6],[8,7]]
```

The final sum `[[[[8,7],[7,7]],[[8,6],[7,7]]],[[[0,7],[6,6]],[8,7]]]` is found after adding up the above snailfish numbers:

```
  [[[0,[4,5]],[0,0]],[[[4,5],[2,6]],[9,5]]]
+ [7,[[[3,7],[4,3]],[[6,3],[8,8]]]]
= [[[[4,0],[5,4]],[[7,7],[6,0]]],[[8,[7,7]],[[7,9],[5,0]]]]

  [[[[4,0],[5,4]],[[7,7],[6,0]]],[[8,[7,7]],[[7,9],[5,0]]]]
+ [[2,[[0,8],[3,4]]],[[[6,7],1],[7,[1,6]]]]
= [[[[6,7],[6,7]],[[7,7],[0,7]]],[[[8,7],[7,7]],[[8,8],[8,0]]]]

  [[[[6,7],[6,7]],[[7,7],[0,7]]],[[[8,7],[7,7]],[[8,8],[8,0]]]]
+ [[[[2,4],7],[6,[0,5]]],[[[6,8],[2,8]],[[2,1],[4,5]]]]
= [[[[7,0],[7,7]],[[7,7],[7,8]]],[[[7,7],[8,8]],[[7,7],[8,7]]]]

  [[[[7,0],[7,7]],[[7,7],[7,8]]],[[[7,7],[8,8]],[[7,7],[8,7]]]]
+ [7,[5,[[3,8],[1,4]]]]
= [[[[7,7],[7,8]],[[9,5],[8,7]]],[[[6,8],[0,8]],[[9,9],[9,0]]]]

  [[[[7,7],[7,8]],[[9,5],[8,7]]],[[[6,8],[0,8]],[[9,9],[9,0]]]]
+ [[2,[2,2]],[8,[8,1]]]
= [[[[6,6],[6,6]],[[6,0],[6,7]]],[[[7,7],[8,9]],[8,[8,1]]]]

  [[[[6,6],[6,6]],[[6,0],[6,7]]],[[[7,7],[8,9]],[8,[8,1]]]]
+ [2,9]
= [[[[6,6],[7,7]],[[0,7],[7,7]]],[[[5,5],[5,6]],9]]
```

```
  [[[[6,6],[7,7]],[[0,7],[7,7]]],[[[5,5],[5,6]],9]]
+ [1,[[[9,3],9],[[9,0],[0,7]]]]
= [[[[7,8],[6,7]],[[6,8],[0,8]]],[[[7,7],[5,0]],[[5,5],[5,6]]]]

  [[[[7,8],[6,7]],[[6,8],[0,8]]],[[[7,7],[5,0]],[[5,5],[5,6]]]]
+ [[[5,[7,4]],7],1]
= [[[[7,7],[7,7]],[[8,7],[8,7]]],[[[7,0],[7,7]],9]]

  [[[[7,7],[7,7]],[[8,7],[8,7]]],[[[7,0],[7,7]],9]]
+ [[[[4,2],2],6],[8,7]]
= [[[[8,7],[7,7]],[[8,6],[7,7]]],[[[0,7],[6,6]],[8,7]]]
```

To check whether it's the right answer, the snailfish teacher only checks the *magnitude* of the final sum. The magnitude of a pair is 3 times the magnitude of its left element plus 2 times the magnitude of its right element. The magnitude of a regular number is just that number.

For example, the magnitude of `[9,1]` is `3*9 + 2*1 = 29`; the magnitude of `[1,9]` is `3*1 + 2*9 = 21`. Magnitude calculations are recursive: the magnitude of `[[9,1],[1,9]]` is `3*29 + 2*21 = 129`.

Here are a few more magnitude examples:

- `[[1,2],[[3,4],5]]` becomes `143`.
- `[[[[0,7],4],[[7,8],[6,0]]],[8,1]]` becomes `1384`.
- `[[[[1,1],[2,2]],[3,3]],[4,4]]` becomes `445`.
- `[[[[3,0],[5,3]],[4,4]],[5,5]]` becomes `791`.
- `[[[[5,0],[7,4]],[5,5]],[6,6]]` becomes `1137`.
- `[[[[8,7],[7,7]],[[8,6],[7,7]]],[[[0,7],[6,6]],[8,7]]]` becomes `3488`.

So, given this example homework assignment:

```
[[[0,[5,8]],[[1,7],[9,6]]],[[4,[1,2]],[[1,4],2]]]
[[[5,[2,8]],4],[5,[[9,9],0]]]
[6,[[[6,2],[5,6]],[[7,6],[4,7]]]]
[[[6,[0,7]],[0,9]],[4,[9,[9,0]]]]
[[[7,[6,4]],[3,[1,3]]],[[[5,5],1],9]]
[[6,[[7,3],[3,2]]],[[[3,8],[5,7]],4]]
[[[[5,4],[7,7]],8],[[8,3],8]]
[[9,3],[[9,9],[6,[4,9]]]]
[[2,[[7,7],7]],[[5,8],[[9,3],[0,2]]]]
[[[[5,2],5],[8,[3,7]]],[[5,[7,5]],[4,4]]]
```

The final sum is:

```
[[[[6,6],[7,6]],[[7,7],[7,0]]],[[[7,7],[7,7]],[[7,8],[9,9]]]]
```

The magnitude of this final sum is `4140`.

Add up all of the snailfish numbers from the homework assignment in the order they appear. *What is the magnitude of the final sum?*

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

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

--- Day 19: Beacon Scanner ---

As your probe drifted down through this area, it released an assortment of *beacons* and *scanners* into the water. It's difficult to navigate in the pitch black open waters of the ocean trench, but if you can build a map of the trench using data from the scanners, you should be able to safely reach the bottom.

The beacons and scanners float motionless in the water; they're designed to maintain the same position for long periods of time. Each scanner is capable of detecting all beacons in a large cube centered on the scanner; beacons that are at most 1000 units away from the scanner in each of the three axes (`x`, `y`, and `z`) have their precise position determined relative to the scanner. However, scanners cannot detect other scanners. The submarine has automatically summarized the relative positions of beacons detected by each scanner (your puzzle input).

For example, if a scanner is at `x,y,z` coordinates `500,0,-500` and there are beacons at `-500,1000,-1500` and `1501,0,-500`, the scanner could report that the first beacon is at `-1000,1000,-1000` (relative to the scanner) but would not detect the second beacon at all.

Unfortunately, while each scanner can report the positions of all detected beacons relative to itself, *the scanners do not know their own position*. You'll need to determine the positions of the beacons and scanners yourself.

The scanners and beacons map a single contiguous 3d region. This region can be reconstructed by finding pairs of scanners that have overlapping detection regions such that there are *at least 12 beacons* that both scanners detect within the overlap. By establishing 12 common beacons, you can precisely determine where the scanners are relative to each other, allowing you to reconstruct the beacon map one scanner at a time.

For a moment, consider only two dimensions. Suppose you have the following scanner reports:

```
--- scanner 0 ---
0,2
4,1
3,3

--- scanner 1 ---
-1,-1
-5,0
-2,1
```

Drawing `x` increasing rightward, `y` increasing upward, scanners as `S`, and beacons as `B`, scanner `0` detects this:

```
...B.
B....
....B
S....
```

Scanner `1` detects this:

```
...B..
B....S
....B.
```

For this example, assume scanners only need 3 overlapping beacons. Then, the beacons visible to both scanners overlap to produce the following complete map:

```
...B..
B....S
....B.
S.....
```

Unfortunately, there's a second problem: the scanners also don't know their *rotation or facing direction*. Due to magnetic alignment, each scanner is rotated some integer number of 90-degree turns around all of the `x`, `y`, and `z` axes. That is, one scanner might call a direction positive `x`, while another scanner might call that direction negative `y`. Or, two scanners might agree on which direction is positive `x`, but one scanner might be upside-down from the perspective of the other scanner. In total, each scanner could be in any of 24 different orientations: facing positive or negative `x`, `y`, or `z`, and considering any of four directions "up" from that facing.

For example, here is an arrangement of beacons as seen from a scanner in the same position but in different orientations:

```
--- scanner 0 ---
-1,-1,1
-2,-2,2
-3,-3,3
-2,-3,1
5,6,-4
8,0,7

--- scanner 0 ---
1,-1,1
2,-2,2
3,-3,3
2,-1,3
-5,4,-6
-8,-7,0
```

```
--- scanner 0 ---
-1,-1,-1
-2,-2,-2
-3,-3,-3
-1,-3,-2
4,6,5
-7,0,8

--- scanner 0 ---
1,1,-1
2,2,-2
3,3,-3
1,3,-2
-4,-6,5
7,0,8

--- scanner 0 ---
1,1,1
2,2,2
3,3,3
3,1,2
-6,-4,-5
0,7,-8
```

By finding pairs of scanners that both see at least 12 of the same beacons, you can assemble the entire map. For example, consider the following report:

```
--- scanner 0 ---
404,-588,-901
528,-643,409
-838,591,734
390,-675,-793
-537,-823,-458
-485,-357,347
-345,-311,381
-661,-816,-575
-876,649,763
-618,-824,-621
553,345,-567
474,580,667
-447,-329,318
-584,868,-557
544,-627,-890
564,392,-477
455,729,728
-892,524,684
```

```
-689,845,-530
423,-701,434
7,-33,-71
630,319,-379
443,580,662
-789,900,-551
459,-707,401

--- scanner 1 ---
686,422,578
605,423,415
515,917,-361
-336,658,858
95,138,22
-476,619,847
-340,-569,-846
567,-361,727
-460,603,-452
669,-402,600
729,430,532
-500,-761,534
-322,571,750
-466,-666,-811
-429,-592,574
-355,545,-477
703,-491,-529
-328,-685,520
413,935,-424
-391,539,-444
586,-435,557
-364,-763,-893
807,-499,-711
755,-354,-619
553,889,-390

--- scanner 2 ---
649,640,665
682,-795,504
-784,533,-524
-644,584,-595
-588,-843,648
-30,6,44
-674,560,763
500,723,-460
609,671,-379
-555,-800,653
```

```
-675,-892,-343
697,-426,-610
578,704,681
493,664,-388
-671,-858,530
-667,343,800
571,-461,-707
-138,-166,112
-889,563,-600
646,-828,498
640,759,510
-630,509,768
-681,-892,-333
673,-379,-804
-742,-814,-386
577,-820,562

--- scanner 3 ---
-589,542,597
605,-692,669
-500,565,-823
-660,373,557
-458,-679,-417
-488,449,543
-626,468,-788
338,-750,-386
528,-832,-391
562,-778,733
-938,-730,414
543,643,-506
-524,371,-870
407,773,750
-104,29,83
378,-903,-323
-778,-728,485
426,699,580
-438,-605,-362
-469,-447,-387
509,732,623
647,635,-688
-868,-804,481
614,-800,639
595,780,-596

--- scanner 4 ---
727,592,562
```

```
-293,-554,779
441,611,-461
-714,465,-776
-743,427,-804
-660,-479,-426
832,-632,460
927,-485,-438
408,393,-506
466,436,-512
110,16,151
-258,-428,682
-393,719,612
-211,-452,876
808,-476,-593
-575,615,604
-485,667,467
-680,325,-822
-627,-443,-432
872,-547,-609
833,512,582
807,604,487
839,-516,451
891,-625,532
-652,-548,-490
30,-46,-14
```

Because all coordinates are relative, in this example, all "absolute" positions will be expressed relative to scanner `0` (using the orientation of scanner `0` and as if scanner `0` is at coordinates `0,0,0`).

Scanners `0` and `1` have overlapping detection cubes; the 12 beacons they both detect (relative to scanner `0`) are at the following coordinates:

```
-618,-824,-621
-537,-823,-458
-447,-329,318
404,-588,-901
544,-627,-890
528,-643,409
-661,-816,-575
390,-675,-793
423,-701,434
-345,-311,381
459,-707,401
-485,-357,347
```

These same 12 beacons (in the same order) but from the perspective of scanner `1` are:

```
686,422,578
605,423,415
515,917,-361
-336,658,858
-476,619,847
-460,603,-452
729,430,532
-322,571,750
-355,545,-477
413,935,-424
-391,539,-444
553,889,-390
```

Because of this, scanner `1` must be at `68,-1246,-43` (relative to scanner `0`).

Scanner `4` overlaps with scanner `1`; the 12 beacons they both detect (relative to scanner `0`) are:

```
459,-707,401
-739,-1745,668
-485,-357,347
432,-2009,850
528,-643,409
423,-701,434
-345,-311,381
408,-1815,803
534,-1912,768
-687,-1600,576
-447,-329,318
-635,-1737,486
```

So, scanner `4` is at `-20,-1133,1061` (relative to scanner `0`).

Following this process, scanner `2` must be at `1105,-1205,1229` (relative to scanner `0`) and scanner `3` must be at `-92,-2380,-20` (relative to scanner `0`).

The full list of beacons (relative to scanner `0`) is:

```
-892,524,684
-876,649,763
-838,591,734
-789,900,-551
-739,-1745,668
-706,-3180,-659
-697,-3072,-689
-689,845,-530
-687,-1600,576
-661,-816,-575
-654,-3158,-753
```

```
-635,-1737,486
-631,-672,1502
-624,-1620,1868
-620,-3212,371
-618,-824,-621
-612,-1695,1788
-601,-1648,-643
-584,868,-557
-537,-823,-458
-532,-1715,1894
-518,-1681,-600
-499,-1607,-770
-485,-357,347
-470,-3283,303
-456,-621,1527
-447,-329,318
-430,-3130,366
-413,-627,1469
-345,-311,381
-36,-1284,1171
-27,-1108,-65
7,-33,-71
12,-2351,-103
26,-1119,1091
346,-2985,342
366,-3059,397
377,-2827,367
390,-675,-793
396,-1931,-563
404,-588,-901
408,-1815,803
423,-701,434
432,-2009,850
443,580,662
455,729,728
456,-540,1869
459,-707,401
465,-695,1988
474,580,667
496,-1584,1900
497,-1838,-617
527,-524,1933
528,-643,409
534,-1912,768
544,-627,-890
553,345,-567
```

```
564,392,-477
568,-2007,-577
605,-1665,1952
612,-1593,1893
630,319,-379
686,-3108,-505
776,-3184,-501
846,-3110,-434
1135,-1161,1235
1243,-1093,1063
1660,-552,429
1693,-557,386
1735,-437,1738
1749,-1800,1813
1772,-405,1572
1776,-675,371
1779,-442,1789
1780,-1548,337
1786,-1538,337
1847,-1591,415
1889,-1729,1762
1994,-1805,1792
```

In total, there are `79` beacons.

Assemble the full map of beacons. *How many beacons are there?*

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

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

--- Day 20: Trench Map ---

With the scanners fully deployed, you turn their attention to mapping the floor of the ocean trench.

When you get back the image from the scanners, it seems to just be random noise. Perhaps you can combine an image enhancement algorithm and the input image (your puzzle input) to clean it up a little.

For example:

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

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

The first section is the *image enhancement algorithm*. It is normally given on a single line, but it has been wrapped to multiple lines in this example for legibility. The second section is the *input image*, a two-dimensional grid of *light pixels* (`#`) and *dark pixels* (`.`).

The image enhancement algorithm describes how to enhance an image by *simultaneously* converting all pixels in the input image into an output image. Each pixel of the output image is determined by looking at a 3x3 square of pixels centered on the corresponding input image pixel. So, to determine the value of the pixel at (5,10) in the output image, nine pixels from the input image need to be considered: (4,9), (4,10), (4,11), (5,9), (5,10), (5,11), (6,9), (6,10), and (6,11). These nine input pixels are combined into a single binary number that is used as an index in the *image enhancement algorithm* string.

For example, to determine the output pixel that corresponds to the very middle pixel of the input image, the nine pixels marked by `[...]` would need to be considered:

```
# . . # .
#[. . .].
#[# . .]#
.[. # .].
. . # # #
```

Starting from the top-left and reading across each row, these pixels are `...`, then `#..`, then `.#.`; combining these forms `...#...#..` By turning dark pixels (`.`) into `0` and light pixels (`#`) into `1`, the binary number `000100010` can be formed, which is `34` in decimal.

The image enhancement algorithm string is exactly 512 characters long, enough to match every possible 9-bit binary number. The first few characters of the string (numbered starting from zero) are as follows:

```
0         10         20         30  34    40         50         60         70
|         |          |          |   |     |          |          |          |
..#.#..#####.#.#.#.###.##.....###.##.#..###.####..#####..#....#..#..##..##
```

In the middle of this first group of characters, the character at index 34 can be found: `#`. So, the output pixel in the center of the output image should be `#`, a *light pixel*.

This process can then be repeated to calculate every pixel of the output image.

Through advances in imaging technology, the images being operated on here are *infinite* in size. *Every* pixel of the infinite output image needs to be calculated exactly based on the relevant pixels of the input image. The small input image you have is only a small region of the actual infinite input image; the rest of the input image consists of dark pixels (`.`). For the purposes of the example, to save on space, only a portion of the infinite-sized input and output images will be shown.

The starting input image, therefore, looks something like this, with more dark pixels (`.`) extending forever in every direction not shown here:

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

By applying the image enhancement algorithm to every pixel simultaneously, the following output image can be obtained:

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

Through further advances in imaging technology, the above output image can also be used as an input image! This allows it to be enhanced *a second time*:

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

Truly incredible - now the small details are really starting to come through. After enhancing the original input image twice, `35` pixels are lit.

Start with the original input image and apply the image enhancement algorithm twice, being careful to account for the infinite size of the images. *How many pixels are lit in the resulting image?*

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

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

--- Day 21: Dirac Dice ---

There's not much to do as you slowly descend to the bottom of the ocean. The submarine computer challenges you to a nice game of *Dirac Dice*.

This game consists of a single die, two pawns, and a game board with a circular track containing ten spaces marked `1` through `10` clockwise. Each player's *starting space* is chosen randomly (your puzzle input). Player 1 goes first.

Players take turns moving. On each player's turn, the player rolls the die *three times* and adds up the results. Then, the player moves their pawn that many times *forward* around the track (that is, moving clockwise on spaces in order of increasing value, wrapping back around to `1` after `10`). So, if a player is on space `7` and they roll `2`, `2`, and `1`, they would move forward 5 times, to spaces `8`, `9`, `10`, `1`, and finally stopping on `2`.

After each player moves, they increase their *score* by the value of the space their pawn stopped on. Players' scores start at `0`. So, if the first player starts on space `7` and rolls a total of `5`, they would stop on space `2` and add `2` to their score (for a total score of `2`). The game immediately ends as a win for any player whose score reaches *at least `1000`*.

Since the first game is a practice game, the submarine opens a compartment labeled *deterministic dice* and a 100-sided die falls out. This die always rolls `1` first, then `2`, then `3`, and so on up to `100`, after which it starts over at `1` again. Play using this die.

For example, given these starting positions:

```
Player 1 starting position: 4
Player 2 starting position: 8
```

This is how the game would go:

- Player 1 rolls `1+2+3` and moves to space `10` for a total score of `10`.
- Player 2 rolls `4+5+6` and moves to space `3` for a total score of `3`.
- Player 1 rolls `7+8+9` and moves to space `4` for a total score of `14`.
- Player 2 rolls `10+11+12` and moves to space `6` for a total score of `9`.
- Player 1 rolls `13+14+15` and moves to space `6` for a total score of `20`.
- Player 2 rolls `16+17+18` and moves to space `7` for a total score of `16`.
- Player 1 rolls `19+20+21` and moves to space `6` for a total score of `26`.
- Player 2 rolls `22+23+24` and moves to space `6` for a total score of `22`.

...after many turns...

- Player 2 rolls `82+83+84` and moves to space `6` for a total score of `742`.
- Player 1 rolls `85+86+87` and moves to space `4` for a total score of `990`.
- Player 2 rolls `88+89+90` and moves to space `3` for a total score of `745`.
- Player 1 rolls `91+92+93` and moves to space `10` for a final score, `1000`.

Since player 1 has at least `1000` points, player 1 wins and the game ends. At this point, the losing player had `745` points and the die had been rolled a total of `993` times; `745 * 993 = 739785`.

Play a practice game using the deterministic 100-sided die. The moment either player wins, *what do you get if you multiply the score of the losing player by the number of times the die was rolled during the game?*

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

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

--- Day 22: Reactor Reboot ---

Operating at these extreme ocean depths has overloaded the submarine's reactor; it needs to be rebooted.

The reactor core is made up of a large 3-dimensional grid made up entirely of cubes, one cube per integer 3-dimensional coordinate (`x,y,z`). Each cube can be either *on* or *off*; at the start of the reboot process, they are all *off*. (Could it be an old model of a reactor you've seen before?)

To reboot the reactor, you just need to set all of the cubes to either *on* or *off* by following a list of *reboot steps* (your puzzle input). Each step specifies a cuboid

(the set of all cubes that have coordinates which fall within ranges for `x`, `y`, and `z`) and whether to turn all of the cubes in that cuboid *on* or *off*.

For example, given these reboot steps:

```
on x=10..12,y=10..12,z=10..12
on x=11..13,y=11..13,z=11..13
off x=9..11,y=9..11,z=9..11
on x=10..10,y=10..10,z=10..10
```

The first step (`on x=10..12,y=10..12,z=10..12`) turns *on* a 3x3x3 cuboid consisting of 27 cubes:

- `10,10,10`
- `10,10,11`
- `10,10,12`
- `10,11,10`
- `10,11,11`
- `10,11,12`
- `10,12,10`
- `10,12,11`
- `10,12,12`
- `11,10,10`
- `11,10,11`
- `11,10,12`
- `11,11,10`
- `11,11,11`
- `11,11,12`
- `11,12,10`
- `11,12,11`
- `11,12,12`
- `12,10,10`
- `12,10,11`
- `12,10,12`
- `12,11,10`
- `12,11,11`
- `12,11,12`
- `12,12,10`
- `12,12,11`
- `12,12,12`

The second step (`on x=11..13,y=11..13,z=11..13`) turns *on* a 3x3x3 cuboid that overlaps with the first. As a result, only 19 additional cubes turn on; the rest are already on from the previous step:

- `11,11,13`
- `11,12,13`
- `11,13,11`
- `11,13,12`

- 11,13,13
- 12,11,13
- 12,12,13
- 12,13,11
- 12,13,12
- 12,13,13
- 13,11,11
- 13,11,12
- 13,11,13
- 13,12,11
- 13,12,12
- 13,12,13
- 13,13,11
- 13,13,12
- 13,13,13

The third step (`off x=9..11,y=9..11,z=9..11`) turns *off* a 3x3x3 cuboid that overlaps partially with some cubes that are on, ultimately turning off 8 cubes:

- 10,10,10
- 10,10,11
- 10,11,10
- 10,11,11
- 11,10,10
- 11,10,11
- 11,11,10
- 11,11,11

The final step (`on x=10..10,y=10..10,z=10..10`) turns *on* a single cube, `10,10,10`. After this last step, `39` cubes are *on*.

The initialization procedure only uses cubes that have `x`, `y`, and `z` positions of at least `-50` and at most `50`. For now, ignore cubes outside this region.

Here is a larger example:

```
on x=-20..26,y=-36..17,z=-47..7
on x=-20..33,y=-21..23,z=-26..28
on x=-22..28,y=-29..23,z=-38..16
on x=-46..7,y=-6..46,z=-50..-1
on x=-49..1,y=-3..46,z=-24..28
on x=2..47,y=-22..22,z=-23..27
on x=-27..23,y=-28..26,z=-21..29
on x=-39..5,y=-6..47,z=-3..44
on x=-30..21,y=-8..43,z=-13..34
on x=-22..26,y=-27..20,z=-29..19
off x=-48..-32,y=26..41,z=-47..-37
on x=-12..35,y=6..50,z=-50..-2
off x=-48..-32,y=-32..-16,z=-15..-5
```

```
on x=-18..26,y=-33..15,z=-7..46
off x=-40..-22,y=-38..-28,z=23..41
on x=-16..35,y=-41..10,z=-47..6
off x=-32..-23,y=11..30,z=-14..3
on x=-49..-5,y=-3..45,z=-29..18
off x=18..30,y=-20..-8,z=-3..13
on x=-41..9,y=-7..43,z=-33..15
on x=-54112..-39298,y=-85059..-49293,z=-27449..7877
on x=967..23432,y=45373..81175,z=27513..53682
```

The last two steps are fully outside the initialization procedure area; all other steps are fully within it. After executing these steps in the initialization procedure region, `590784` cubes are *on*.

Execute the reboot steps. Afterward, considering only cubes in the region `x=-50..50,y=-50..50,z=-50..50`, *how many cubes are on?*

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

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

--- Day 23: Amphipod ---

A group of amphipods notice your fancy submarine and flag you down. "With such an impressive shell," one amphipod says, "surely you can help us with a question that has stumped our best scientists."

They go on to explain that a group of timid, stubborn amphipods live in a nearby burrow. Four types of amphipods live there: *Amber* (`A`), *Bronze* (`B`), *Copper* (`C`), and *Desert* (`D`). They live in a burrow that consists of a *hallway* and four *side rooms*. The side rooms are initially full of amphipods, and the hallway is initially empty.

They give you a *diagram of the situation* (your puzzle input), including locations of each amphipod (`A`, `B`, `C`, or `D`, each of which is occupying an otherwise open space), walls (`#`), and open space (`.`).

For example:

```
#############
#...........#
###B#C#B#D###
  #A#D#C#A#
  #########
```

The amphipods would like a method to organize every amphipod into side rooms so that each side room contains one type of amphipod and the types are sorted `A-D` going left to right, like this:

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

```
###A#B#C#D###
  #A#B#C#D#
  #########
```

Amphipods can move up, down, left, or right so long as they are moving into an unoccupied open space. Each type of amphipod requires a different amount of *energy* to move one step: Amber amphipods require `1` energy per step, Bronze amphipods require `10` energy, Copper amphipods require `100`, and Desert ones require `1000`. The amphipods would like you to find a way to organize the amphipods that requires the *least total energy*.

However, because they are timid and stubborn, the amphipods have some extra rules:

- Amphipods will never *stop on the space immediately outside any room*. They can move into that space so long as they immediately continue moving. (Specifically, this refers to the four open spaces in the hallway that are directly above an amphipod starting position.)
- Amphipods will never *move from the hallway into a room* unless that room is their destination room *and* that room contains no amphipods which do not also have that room as their own destination. If an amphipod's starting room is not its destination room, it can stay in that room until it leaves the room. (For example, an Amber amphipod will not move from the hallway into the right three rooms, and will only move into the leftmost room if that room is empty or if it only contains other Amber amphipods.)
- Once an amphipod stops moving in the hallway, *it will stay in that spot until it can move into a room*. (That is, once any amphipod starts moving, any other amphipods currently in the hallway are locked in place and will not move again until they can move fully into a room.)

In the above example, the amphipods can be organized using a minimum of `12521` energy. One way to do this is shown below.

Starting configuration:

```
#############
#...........#
###B#C#B#D###
  #A#D#C#A#
  #########
```

One Bronze amphipod moves into the hallway, taking 4 steps and using `40` energy:

```
#############
#...B.......#
###B#C#.#D###
  #A#D#C#A#
  #########
```

The only Copper amphipod not in its side room moves there, taking 4 steps and using 400 energy:

```
#############
#...B.......#
###B#.#C#D###
  #A#D#C#A#
  #########
```

A Desert amphipod moves out of the way, taking 3 steps and using 3000 energy, and then the Bronze amphipod takes its place, taking 3 steps and using 30 energy:

```
#############
#.....D.....#
###B#.#C#D###
  #A#B#C#A#
  #########
```

The leftmost Bronze amphipod moves to its room using 40 energy:

```
#############
#.....D.....#
###.#B#C#D###
  #A#B#C#A#
  #########
```

Both amphipods in the rightmost room move into the hallway, using 2003 energy in total:

```
#############
#.....D.D.A.#
###.#B#C#.###
  #A#B#C#.#
  #########
```

Both Desert amphipods move into the rightmost room using 7000 energy:

```
#############
#.........A.#
###.#B#C#D###
  #A#B#C#D#
  #########
```

Finally, the last Amber amphipod moves into its room, using 8 energy:

```
#############
#...........#
###A#B#C#D###
  #A#B#C#D#
  #########
```

What is the least energy required to organize the amphipods?

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

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

--- Day 24: Arithmetic Logic Unit ---

Magic smoke starts leaking from the submarine's arithmetic logic unit (ALU). Without the ability to perform basic arithmetic and logic functions, the submarine can't produce cool patterns with its Christmas lights!

It also can't navigate. Or run the oxygen system.

Don't worry, though - you *probably* have enough oxygen left to give you enough time to build a new ALU.

The ALU is a four-dimensional processing unit: it has integer variables `w`, `x`, `y`, and `z`. These variables all start with the value `0`. The ALU also supports *six instructions*:

- `inp a` - Read an input value and write it to variable `a`.
- `add a b` - Add the value of `a` to the value of `b`, then store the result in variable `a`.
- `mul a b` - Multiply the value of `a` by the value of `b`, then store the result in variable `a`.
- `div a b` - Divide the value of `a` by the value of `b`, truncate the result to an integer, then store the result in variable `a`. (Here, "truncate" means to round the value toward zero.)
- `mod a b` - Divide the value of `a` by the value of `b`, then store the *remainder* in variable `a`. (This is also called the modulo operation.)
- `eql a b` - If the value of `a` and `b` are equal, then store the value `1` in variable `a`. Otherwise, store the value `0` in variable `a`.

In all of these instructions, `a` and `b` are placeholders; `a` will always be the variable where the result of the operation is stored (one of `w`, `x`, `y`, or `z`), while `b` can be either a variable or a number. Numbers can be positive or negative, but will always be integers.

The ALU has no *jump* instructions; in an ALU program, every instruction is run exactly once in order from top to bottom. The program halts after the last instruction has finished executing.

(Program authors should be especially cautious; attempting to execute `div` with `b=0` or attempting to execute `mod` with `a<0` or `b<=0` will cause the program to crash and might even damage the ALU. These operations are never intended in any serious ALU program.)

For example, here is an ALU program which takes an input number, negates it, and stores it in `x`:

```
inp x
mul x -1
```

Here is an ALU program which takes two input numbers, then sets `z` to `1` if the second input number is three times larger than the first input number, or sets `z` to `0` otherwise:

```
inp z
inp x
mul z 3
eql z x
```

Here is an ALU program which takes a non-negative integer as input, converts it into binary, and stores the lowest (1's) bit in `z`, the second-lowest (2's) bit in `y`, the third-lowest (4's) bit in `x`, and the fourth-lowest (8's) bit in `w`:

```
inp w
add z w
mod z 2
div w 2
add y w
mod y 2
div w 2
add x w
mod x 2
div w 2
mod w 2
```

Once you have built a replacement ALU, you can install it in the submarine, which will immediately resume what it was doing when the ALU failed: validating the submarine's *model number*. To do this, the ALU will run the MOdel Number Automatic Detector program (MONAD, your puzzle input).

Submarine model numbers are always *fourteen-digit numbers* consisting only of digits `1` through `9`. The digit `0` *cannot* appear in a model number.

When MONAD checks a hypothetical fourteen-digit model number, it uses fourteen separate `inp` instructions, each expecting a *single digit* of the model number in order of most to least significant. (So, to check the model number `13579246899999`, you would give `1` to the first `inp` instruction, `3` to the second `inp` instruction, `5` to the third `inp` instruction, and so on.) This means that when operating MONAD, each input instruction should only ever be given an integer value of at least `1` and at most `9`.

Then, after MONAD has finished running all of its instructions, it will indicate that the model number was *valid* by leaving a `0` in variable `z`. However, if the model number was *invalid*, it will leave some other non-zero value in `z`.

MONAD imposes additional, mysterious restrictions on model numbers, and legend says the last copy of the MONAD documentation was eaten by a tanuki.

You'll need to *figure out what MONAD does* some other way.

To enable as many submarine features as possible, find the largest valid fourteen-digit model number that contains no `0` digits. *What is the largest model number accepted by MONAD?*

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

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

--- Day 25: Sea Cucumber ---

This is it: the bottom of the ocean trench, the last place the sleigh keys could be. Your submarine's experimental antenna *still isn't boosted enough* to detect the keys, but they *must* be here. All you need to do is *reach the seafloor* and find them.

At least, you'd touch down on the seafloor if you could; unfortunately, it's completely covered by two large herds of sea cucumbers, and there isn't an open space large enough for your submarine.

You suspect that the Elves must have done this before, because just then you discover the phone number of a deep-sea marine biologist on a handwritten note taped to the wall of the submarine's cockpit.

"Sea cucumbers? Yeah, they're probably hunting for food. But don't worry, they're predictable critters: they move in perfectly straight lines, only moving forward when there's space to do so. They're actually quite polite!"

You explain that you'd like to predict when you could land your submarine.

"Oh that's easy, they'll eventually pile up and leave enough space for-- wait, did you say submarine? And the only place with that many sea cucumbers would be at the very bottom of the Mariana--" You hang up the phone.

There are two herds of sea cucumbers sharing the same region; one always moves *east* (`>`), while the other always moves *south* (`v`). Each location can contain at most one sea cucumber; the remaining locations are *empty* (`.`). The submarine helpfully generates a map of the situation (your puzzle input). For example:

```
v...>>.vv>
.vv>>.vv..
>>.>v>...v
>>v>>.>.v.
v>v.vv.v..
>.>>..v...
.vv..>.>v.
v.v..>>v.v
....v..v.>
```

Every *step*, the sea cucumbers in the east-facing herd attempt to move forward one location, then the sea cucumbers in the south-facing herd attempt to move forward one location. When a herd moves forward, every sea cucumber in the herd first simultaneously considers whether there is a sea cucumber in the adjacent location it's facing (even another sea cucumber facing the same direction), and then every sea cucumber facing an empty location simultaneously moves into that location.

So, in a situation like this:

```
...>>>>>...
```

After one step, only the rightmost sea cucumber would have moved:

```
...>>>>.>..
```

After the next step, two sea cucumbers move:

```
...>>>.>.>.
```

During a single step, the east-facing herd moves first, then the south-facing herd moves. So, given this situation:

```
..........
.>v....v..
.......>..
..........
```

After a single step, of the sea cucumbers on the left, only the south-facing sea cucumber has moved (as it wasn't out of the way in time for the east-facing cucumber on the left to move), but both sea cucumbers on the right have moved (as the east-facing sea cucumber moved out of the way of the south-facing sea cucumber):

```
..........
.>........
..v....v>.
..........
```

Due to *strong water currents* in the area, sea cucumbers that move off the right edge of the map appear on the left edge, and sea cucumbers that move off the bottom edge of the map appear on the top edge. Sea cucumbers always check whether their destination location is empty before moving, even if that destination is on the opposite side of the map:

```
Initial state:
...>...
.......
......>
v.....>
......>
.......
```

```
..vvv..

After 1 step:
..vv>..
.......
>......
v.....>
>......
.......
....v..

After 2 steps:
....v>.
..vv...
.>.....
......>
v>.....
.......
.......

After 3 steps:
......>
..v.v..
..>v...
>......
..>....
v......
.......

After 4 steps:
>......
..v....
..>.v..
.>.v...
...>...
.......
v......
```

To find a safe place to land your submarine, the sea cucumbers need to stop moving. Again consider the first example:

```
Initial state:
v...>>.vv>
.vv>>.vv..
>>.>v>...v
>>v>>.>.v.
v>v.vv.v..
```

```
>.>>..v...
.vv..>.>v.
v.v..>>v.v
....v..v.>

After 1 step:
....>.>v.>
v.v>.>v.v.
>v>>..>v..
>>v>v>.>.v
.>v.v...v.
v>>.>vvv..
..v...>>..
vv...>>vv.
>.v.v..v.v

After 2 steps:
>.v.v>>..v
v.v.>>vv..
>v>.>.>.v.
>>v>v.>v>.
.>..v....v
.>v>>.v.v.
v....v>v>.
.vv..>>v..
v>.....vv.

After 3 steps:
v>v.v>.>v.
v...>>.v.v
>vv>.>v>..
>>v>v.>.v>
..>....v..
.>.>v>v..v
..v..v>vv>
v.v..>>v..
.v>....v..

After 4 steps:
v>..v.>>..
v.v.>.>.v.
>vv.>>.v>v
>>.>..v>.>
..v>v...v.
..>>.>vv..
>.v.vv>v.v
```

```
.....>>vv.
vvv>...v..

After 5 steps:
vv>...>v>.
v.v.v>.>v.
>.v.>.>.>v
>v>.>..v>>
..v>v.v...
..>.>>vvv.
.>...v>v..
..v.v>>v.v
v.v.>...v.

...

After 10 steps:
..>..>>vv.
v.....>>.v
..v.v>>>v>
v>.>v.>>>.
..v>v.vv.v
.v.>>>.v..
v.v..>v>..
..v...>v.>
.vv..v>vv.

...

After 20 steps:
v>.....>>.
>vv>.....v
.>v>v.vv>>
v>>>v.>v.>
....vv>v..
.v.>>>vvv.
..v..>>vv.
v.v...>>.v
..v.....v>

...

After 30 steps:
.vv.v..>>>
v>...v...>
>.v>.>vv.>
```

```
>v>.>.>v.>
.>..v.vv..
..v>..>>v.
....v>..>v
v.v...>vv>
v.v...>vvv

...

After 40 steps:
>>v>v..v..
..>>v..vv.
..>>>v.>.v
..>>>>vvv>
v.....>...
v.v...>v>>
>vv.....v>
.>v...v.>v
vvv.v..v.>

...

After 50 steps:
..>>v>vv.v
..v.>>vv..
v.>>v>>v..
..>>>>>vv.
vvv....>vv
..v....>>>
v>.......>
.vv>....v>
.>v.vv.v..

...

After 55 steps:
..>>v>vv..
..v.>>vv..
..>>v>>vv.
..>>>>>vv.
v......>vv
v>v....>>v
vvv...>..>
>vv.....>.
.>v.vv.v..
```

```
After 56 steps:
..>>v>vv..
..v.>>vv..
..>>v>>vv.
..>>>>>vv.
v......>vv
v>v....>>v
vvv....>.>
>vv......>
.>v.vv.v..

After 57 steps:
..>>v>vv..
..v.>>vv..
..>>v>>vv.
..>>>>>vv.
v......>vv
v>v....>>v
vvv.....>>
>vv......>
.>v.vv.v..

After 58 steps:
..>>v>vv..
..v.>>vv..
..>>v>>vv.
..>>>>>vv.
v......>vv
v>v....>>v
vvv.....>>
>vv......>
.>v.vv.v..
```

In this example, the sea cucumbers stop moving after `58` steps.

Find somewhere safe to land your submarine. *What is the first step on which no sea cucumbers move?*

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

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