--- Day 1: Calorie Counting ---

Santa's reindeer typically eat regular reindeer food, but they need a lot of magical energy to deliver presents on Christmas. For that, their favorite snack is a special type of *star* fruit that only grows deep in the jungle. The Elves have brought you on their annual expedition to the grove where the fruit grows.

To supply enough magical energy, the expedition needs to retrieve a minimum of *fifty stars* by December 25th. Although the Elves assure you that the grove has plenty of fruit, you decide to grab any fruit you see along the way, just in case.

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

The jungle must be too overgrown and difficult to navigate in vehicles or access from the air; the Elves' expedition traditionally goes on foot. As your boats approach land, the Elves begin taking inventory of their supplies. One important consideration is food - in particular, the number of *Calories* each Elf is carrying (your puzzle input).

The Elves take turns writing down the number of Calories contained by the various meals, snacks, rations, etc. that they've brought with them, one item per line. Each Elf separates their own inventory from the previous Elf's inventory (if any) by a blank line.

For example, suppose the Elves finish writing their items' Calories and end up with the following list:

```
1000
2000
3000

4000

5000
6000

7000
8000
9000

10000
```

This list represents the Calories of the food carried by five Elves:

- The first Elf is carrying food with `1000`, `2000`, and `3000` Calories, a total of `6000` Calories.
- The second Elf is carrying one food item with `4000` Calories.

- The third Elf is carrying food with `5000` and `6000` Calories, a total of `11000` Calories.
- The fourth Elf is carrying food with `7000`, `8000`, and `9000` Calories, a total of `24000` Calories.
- The fifth Elf is carrying one food item with `10000` Calories.

In case the Elves get hungry and need extra snacks, they need to know which Elf to ask: they'd like to know how many Calories are being carried by the Elf carrying the *most* Calories. In the example above, this is *`24000`* (carried by the fourth Elf).

Find the Elf carrying the most Calories. *How many total Calories is that Elf carrying?*

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

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

--- Day 2: Rock Paper Scissors ---

The Elves begin to set up camp on the beach. To decide whose tent gets to be closest to the snack storage, a giant Rock Paper Scissors tournament is already in progress.

Rock Paper Scissors is a game between two players. Each game contains many rounds; in each round, the players each simultaneously choose one of Rock, Paper, or Scissors using a hand shape. Then, a winner for that round is selected: Rock defeats Scissors, Scissors defeats Paper, and Paper defeats Rock. If both players choose the same shape, the round instead ends in a draw.

Appreciative of your help yesterday, one Elf gives you an *encrypted strategy guide* (your puzzle input) that they say will be sure to help you win. "The first column is what your opponent is going to play: `A` for Rock, `B` for Paper, and `C` for Scissors. The second column--" Suddenly, the Elf is called away to help with someone's tent.

The second column, you reason, must be what you should play in response: `X` for Rock, `Y` for Paper, and `Z` for Scissors. Winning every time would be suspicious, so the responses must have been carefully chosen.

The winner of the whole tournament is the player with the highest score. Your *total score* is the sum of your scores for each round. The score for a single round is the score for the *shape you selected* (1 for Rock, 2 for Paper, and 3 for Scissors) plus the score for the *outcome of the round* (0 if you lost, 3 if the round was a draw, and 6 if you won).

Since you can't be sure if the Elf is trying to help you or trick you, you should calculate the score you would get if you were to follow the strategy guide.

For example, suppose you were given the following strategy guide:

```
A Y
B X
C Z
```

This strategy guide predicts and recommends the following:

- In the first round, your opponent will choose Rock (`A`), and you should choose Paper (`Y`). This ends in a win for you with a score of *8* (2 because you chose Paper + 6 because you won).
- In the second round, your opponent will choose Paper (`B`), and you should choose Rock (`X`). This ends in a loss for you with a score of *1* (1 + 0).
- The third round is a draw with both players choosing Scissors, giving you a score of 3 + 3 = *6*.

In this example, if you were to follow the strategy guide, you would get a total score of `15` (8 + 1 + 6).

What would your total score be if everything goes exactly according to your strategy guide?

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

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

--- Day 3: Rucksack Reorganization ---

One Elf has the important job of loading all of the rucksacks with supplies for the jungle journey. Unfortunately, that Elf didn't quite follow the packing instructions, and so a few items now need to be rearranged.

Each rucksack has two large *compartments*. All items of a given type are meant to go into exactly one of the two compartments. The Elf that did the packing failed to follow this rule for exactly one item type per rucksack.

The Elves have made a list of all of the items currently in each rucksack (your puzzle input), but they need your help finding the errors. Every item type is identified by a single lowercase or uppercase letter (that is, `a` and `A` refer to different types of items).

The list of items for each rucksack is given as characters all on a single line. A given rucksack always has the same number of items in each of its two compartments, so the first half of the characters represent items in the first compartment, while the second half of the characters represent items in the second compartment.

For example, suppose you have the following list of contents from six rucksacks:

```
vJrwpWtwJgWrhcsFMMfFFhFp
jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL
PmmdzqPrVvPwwTWBwg
wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn
```

```
ttgJtRGJQctTZtZT
CrZsJsPPZsGzwwsLwLmpwMDw
```

- The first rucksack contains the items `vJrwpWtwJgWrhcsFMMfFFhFp`, which means its first compartment contains the items `vJrwpWtwJgWr`, while the second compartment contains the items `hcsFMMfFFhFp`. The only item type that appears in both compartments is lowercase `p`.
- The second rucksack's compartments contain `jqHRNqRjqzjGDLGL` and `rsFMfFZSrLrFZsSL`. The only item type that appears in both compartments is uppercase `L`.
- The third rucksack's compartments contain `PmmdzqPrV` and `vPwwTWBwg`; the only common item type is uppercase `P`.
- The fourth rucksack's compartments only share item type `v`.
- The fifth rucksack's compartments only share item type `t`.
- The sixth rucksack's compartments only share item type `s`.

To help prioritize item rearrangement, every item type can be converted to a *priority*:

- Lowercase item types `a` through `z` have priorities 1 through 26.
- Uppercase item types `A` through `Z` have priorities 27 through 52.

In the above example, the priority of the item type that appears in both compartments of each rucksack is 16 (`p`), 38 (`L`), 42 (`P`), 22 (`v`), 20 (`t`), and 19 (`s`); the sum of these is `157`.

Find the item type that appears in both compartments of each rucksack. *What is the sum of the priorities of those item types?*

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

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

--- Day 4: Camp Cleanup ---

Space needs to be cleared before the last supplies can be unloaded from the ships, and so several Elves have been assigned the job of cleaning up sections of the camp. Every section has a unique *ID number*, and each Elf is assigned a range of section IDs.

However, as some of the Elves compare their section assignments with each other, they've noticed that many of the assignments *overlap*. To try to quickly find overlaps and reduce duplicated effort, the Elves pair up and make a *big list of the section assignments for each pair* (your puzzle input).

For example, consider the following list of section assignment pairs:

```
2-4,6-8
2-3,4-5
5-7,7-9
2-8,3-7
```

```
6-6,4-6
2-6,4-8
```

For the first few pairs, this list means:

- Within the first pair of Elves, the first Elf was assigned sections `2-4` (sections `2`, `3`, and `4`), while the second Elf was assigned sections `6-8` (sections `6`, `7`, `8`).
- The Elves in the second pair were each assigned two sections.
- The Elves in the third pair were each assigned three sections: one got sections `5`, `6`, and `7`, while the other also got `7`, plus `8` and `9`.

This example list uses single-digit section IDs to make it easier to draw; your actual list might contain larger numbers. Visually, these pairs of section assignments look like this:

```
.234.....  2-4
.....678.  6-8

.23......  2-3
...45....  4-5

....567..  5-7
......789  7-9

.2345678.  2-8
..34567..  3-7

.....6...  6-6
...456...  4-6

.23456...  2-6
...45678.  4-8
```

Some of the pairs have noticed that one of their assignments *fully contains* the other. For example, `2-8` fully contains `3-7`, and `6-6` is fully contained by `4-6`. In pairs where one assignment fully contains the other, one Elf in the pair would be exclusively cleaning sections their partner will already be cleaning, so these seem like the most in need of reconsideration. In this example, there are `2` such pairs.

In how many assignment pairs does one range fully contain the other?

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

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

--- Day 5: Supply Stacks ---

The expedition can depart as soon as the final supplies have been unloaded from the ships. Supplies are stored in stacks of marked *crates*, but because the needed supplies are buried under many other crates, the crates need to be rearranged.

The ship has a *giant cargo crane* capable of moving crates between stacks. To ensure none of the crates get crushed or fall over, the crane operator will rearrange them in a series of carefully-planned steps. After the crates are rearranged, the desired crates will be at the top of each stack.

The Elves don't want to interrupt the crane operator during this delicate procedure, but they forgot to ask her *which* crate will end up where, and they want to be ready to unload them as soon as possible so they can embark.

They do, however, have a drawing of the starting stacks of crates *and* the rearrangement procedure (your puzzle input). For example:

```
    [D]
[N] [C]
[Z] [M] [P]
 1   2   3

move 1 from 2 to 1
move 3 from 1 to 3
move 2 from 2 to 1
move 1 from 1 to 2
```

In this example, there are three stacks of crates. Stack 1 contains two crates: crate `Z` is on the bottom, and crate `N` is on top. Stack 2 contains three crates; from bottom to top, they are crates `M`, `C`, and `D`. Finally, stack 3 contains a single crate, `P`.

Then, the rearrangement procedure is given. In each step of the procedure, a quantity of crates is moved from one stack to a different stack. In the first step of the above rearrangement procedure, one crate is moved from stack 2 to stack 1, resulting in this configuration:

```
[D]
[N] [C]
[Z] [M] [P]
 1   2   3
```

In the second step, three crates are moved from stack 1 to stack 3. Crates are moved *one at a time*, so the first crate to be moved (`D`) ends up below the second and third crates:

```
        [Z]
        [N]
    [C] [D]
```

```
    [M] [P]
 1   2   3
```

Then, both crates are moved from stack 2 to stack 1. Again, because crates are moved *one at a time*, crate `C` ends up below crate `M`:

```
        [Z]
        [N]
[M]     [D]
[C]     [P]
 1   2   3
```

Finally, one crate is moved from stack 1 to stack 2:

```
        [Z]
        [N]
        [D]
[C] [M] [P]
 1   2   3
```

The Elves just need to know *which crate will end up on top of each stack*; in this example, the top crates are `C` in stack 1, `M` in stack 2, and `Z` in stack 3, so you should combine these together and give the Elves the message `CMZ`.

After the rearrangement procedure completes, what crate ends up on top of each stack?

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

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

--- Day 6: Tuning Trouble ---

The preparations are finally complete; you and the Elves leave camp on foot and begin to make your way toward the *star* fruit grove.

As you move through the dense undergrowth, one of the Elves gives you a handheld *device*. He says that it has many fancy features, but the most important one to set up right now is the *communication system*.

However, because he's heard you have significant experience dealing with signal-based systems, he convinced the other Elves that it would be okay to give you their one malfunctioning device - surely you'll have no problem fixing it.

As if inspired by comedic timing, the device emits a few colorful sparks.

To be able to communicate with the Elves, the device needs to *lock on to their signal*. The signal is a series of seemingly-random characters that the device receives one at a time.

To fix the communication system, you need to add a subroutine to the device that detects a *start-of-packet marker* in the datastream. In the protocol being used

by the Elves, the start of a packet is indicated by a sequence of *four characters that are all different*.

The device will send your subroutine a datastream buffer (your puzzle input); your subroutine needs to identify the first position where the four most recently received characters were all different. Specifically, it needs to report the number of characters from the beginning of the buffer to the end of the first such four-character marker.

For example, suppose you receive the following datastream buffer:

```
mjqjpqmgbljsphdztnvjfqwrcgsmlb
```

After the first three characters (`mjq`) have been received, there haven't been enough characters received yet to find the marker. The first time a marker could occur is after the fourth character is received, making the most recent four characters `mjqj`. Because `j` is repeated, this isn't a marker.

The first time a marker appears is after the *seventh* character arrives. Once it does, the last four characters received are `jpqm`, which are all different. In this case, your subroutine should report the value `7`, because the first start-of-packet marker is complete after `7` characters have been processed.

Here are a few more examples:

- `bvwbjplbgvbhsrlpgdmjqwftvncz`: first marker after character `5`
- `nppdvjthqldpwncqszvftbrmjlhg`: first marker after character `6`
- `nznrnfrfntjfmvfwmzdfjlvtqnbhcprsg`: first marker after character `10`
- `zcfzfwzzqfrljwzlrfnpqdbhtmscgvjw`: first marker after character `11`

How many characters need to be processed before the first start-of-packet marker is detected?

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

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

--- Day 7: No Space Left On Device ---

You can hear birds chirping and raindrops hitting leaves as the expedition proceeds. Occasionally, you can even hear much louder sounds in the distance; how big do the animals get out here, anyway?

The device the Elves gave you has problems with more than just its communication system. You try to run a system update:

```
$ system-update --please --pretty-please-with-sugar-on-top
Error: No space left on device
```

Perhaps you can delete some files to make space for the update?

You browse around the filesystem to assess the situation and save the resulting terminal output (your puzzle input). For example:

```
$ cd /
$ ls
dir a
14848514 b.txt
8504156 c.dat
dir d
$ cd a
$ ls
dir e
29116 f
2557 g
62596 h.lst
$ cd e
$ ls
584 i
$ cd ..
$ cd ..
$ cd d
$ ls
4060174 j
8033020 d.log
5626152 d.ext
7214296 k
```

The filesystem consists of a tree of files (plain data) and directories (which can contain other directories or files). The outermost directory is called `/`. You can navigate around the filesystem, moving into or out of directories and listing the contents of the directory you're currently in.

Within the terminal output, lines that begin with `$` are *commands you executed*, very much like some modern computers:

- `cd` means *change directory*. This changes which directory is the current directory, but the specific result depends on the argument:
  - `cd x` moves *in* one level: it looks in the current directory for the directory named `x` and makes it the current directory.
  - `cd ..` moves *out* one level: it finds the directory that contains the current directory, then makes that directory the current directory.
  - `cd /` switches the current directory to the outermost directory, `/`.
- `ls` means *list*. It prints out all of the files and directories immediately contained by the current directory:
  - `123 abc` means that the current directory contains a file named `abc` with size `123`.
  - `dir xyz` means that the current directory contains a directory named `xyz`.

Given the commands and output in the example above, you can determine that the filesystem looks visually like this:

```
- / (dir)
  - a (dir)
    - e (dir)
      - i (file, size=584)
    - f (file, size=29116)
    - g (file, size=2557)
    - h.lst (file, size=62596)
  - b.txt (file, size=14848514)
  - c.dat (file, size=8504156)
  - d (dir)
    - j (file, size=4060174)
    - d.log (file, size=8033020)
    - d.ext (file, size=5626152)
    - k (file, size=7214296)
```

Here, there are four directories: `/` (the outermost directory), `a` and `d` (which are in `/`), and `e` (which is in `a`). These directories also contain files of various sizes.

Since the disk is full, your first step should probably be to find directories that are good candidates for deletion. To do this, you need to determine the *total size* of each directory. The total size of a directory is the sum of the sizes of the files it contains, directly or indirectly. (Directories themselves do not count as having any intrinsic size.)

The total sizes of the directories above can be found as follows:

- The total size of directory `e` is *584* because it contains a single file `i` of size 584 and no other directories.
- The directory `a` has total size *94853* because it contains files `f` (size 29116), `g` (size 2557), and `h.lst` (size 62596), plus file `i` indirectly (`a` contains `e` which contains `i`).
- Directory `d` has total size *24933642*.
- As the outermost directory, `/` contains every file. Its total size is *48381165*, the sum of the size of every file.

To begin, find all of the directories with a total size of *at most 100000*, then calculate the sum of their total sizes. In the example above, these directories are `a` and `e`; the sum of their total sizes is `95437` (94853 + 584). (As in this example, this process can count files more than once!)

Find all of the directories with a total size of at most 100000. *What is the sum of the total sizes of those directories?*

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

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

--- Day 8: Treetop Tree House ---

The expedition comes across a peculiar patch of tall trees all planted carefully in a grid. The Elves explain that a previous expedition planted these trees as a reforestation effort. Now, they're curious if this would be a good location for a tree house.

First, determine whether there is enough tree cover here to keep a tree house *hidden*. To do this, you need to count the number of trees that are *visible from outside the grid* when looking directly along a row or column.

The Elves have already launched a quadcopter to generate a map with the height of each tree (your puzzle input). For example:

```
30373
25512
65332
33549
35390
```

Each tree is represented as a single digit whose value is its height, where `0` is the shortest and `9` is the tallest.

A tree is *visible* if all of the other trees between it and an edge of the grid are *shorter* than it. Only consider trees in the same row or column; that is, only look up, down, left, or right from any given tree.

All of the trees around the edge of the grid are *visible* - since they are already on the edge, there are no trees to block the view. In this example, that only leaves the *interior nine trees* to consider:

- The top-left `5` is *visible* from the left and top. (It isn't visible from the right or bottom since other trees of height `5` are in the way.)
- The top-middle `5` is *visible* from the top and right.
- The top-right `1` is not visible from any direction; for it to be visible, there would need to only be trees of height *0* between it and an edge.
- The left-middle `5` is *visible*, but only from the right.
- The center `3` is not visible from any direction; for it to be visible, there would need to be only trees of at most height `2` between it and an edge.
- The right-middle `3` is *visible* from the right.
- In the bottom row, the middle `5` is *visible*, but the `3` and `4` are not.

With 16 trees visible on the edge and another 5 visible in the interior, a total of `21` trees are visible in this arrangement.

Consider your map; *how many trees are visible from outside the grid?*

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

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

--- Day 9: Rope Bridge ---

This rope bridge creaks as you walk along it. You aren't sure how old it is, or whether it can even support your weight.

It seems to support the Elves just fine, though. The bridge spans a gorge which was carved out by the massive river far below you.

You step carefully; as you do, the ropes stretch and twist. You decide to distract yourself by modeling rope physics; maybe you can even figure out where *not* to step.

Consider a rope with a knot at each end; these knots mark the *head* and the *tail* of the rope. If the head moves far enough away from the tail, the tail is pulled toward the head.

Due to nebulous reasoning involving Planck lengths, you should be able to model the positions of the knots on a two-dimensional grid. Then, by following a hypothetical *series of motions* (your puzzle input) for the head, you can determine how the tail will move.

Due to the aforementioned Planck lengths, the rope must be quite short; in fact, the head (`H`) and tail (`T`) must *always be touching* (diagonally adjacent and even overlapping both count as touching):

```
....
.TH.
....

....
.H..
..T.
....

...
.H. (H covers T)
...
```

If the head is ever two steps directly up, down, left, or right from the tail, the tail must also move one step in that direction so it remains close enough:

```
.....    .....    .....
.TH.. -> .T.H. -> ..TH.
.....    .....    .....

...    ...    ...
.T.    .T.    ...
.H. -> ... -> .T.
...    .H.    .H.
...    ...    ...
```

Otherwise, if the head and tail aren't touching and aren't in the same row or column, the tail always moves one step diagonally to keep up:

```
.....    .....    .....
.....    ..H..    ..H..
..H.. -> ..... -> ..T..
.T...    .T...    .....
.....    .....    .....

.....    .....    .....
.....    .....    .....
..H.. -> ...H. -> ..TH.
.T...    .T...    .....
.....    .....    .....
```

You just need to work out where the tail goes as the head follows a series of motions. Assume the head and the tail both start at the same position, overlapping.

For example:

```
R 4
U 4
L 3
D 1
R 4
D 1
L 5
R 2
```

This series of motions moves the head *right* four steps, then *up* four steps, then *left* three steps, then *down* one step, and so on. After each step, you'll need to update the position of the tail if the step means the head is no longer adjacent to the tail. Visually, these motions occur as follows (`s` marks the starting position as a reference point):

```
== Initial State ==

......
......
......
......
H.....  (H covers T, s)

== R 4 ==

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

```
......
TH....  (T covers s)

......
......
......
......
sTH...

......
......
......
......
s.TH..

......
......
......
......
s..TH.

== U 4 ==

......
......
......
....H.
s..T..

......
......
....H.
....T.
s.....

......
....H.
....T.
......
s.....

....H.
....T.
......
......
s.....
```

```
== L 3 ==

...H..
....T.
......
......
s.....

..HT..
......
......
......
s.....

.HT...
......
......
......
s.....

== D 1 ==

..T...
.H....
......
......
s.....

== R 4 ==

..T...
..H...
......
......
s.....

..T...
...H..
......
......
s.....

......
...TH.
......
```

```
......
s.....

......
....TH
......
......
s.....

== D 1 ==

......
....T.
.....H
......
s.....

== L 5 ==

......
....T.
....H.
......
s.....

......
....T.
...H..
......
s.....

......
......
..HT..
......
s.....

......
......
.HT...
......
s.....

......
......
HT....
```

```
......
s.....

== R 2 ==

......
......
.H....  (H covers T)
......
s.....

......
......
.TH...
......
s.....
```

After simulating the rope, you can count up all of the positions the *tail visited at least once*. In this diagram, `s` again marks the starting position (which the tail also visited) and `#` marks other positions the tail visited:

```
..##..
...##.
.####.
....#.
s###..
```

So, there are `13` positions the tail visited at least once.

Simulate your complete hypothetical series of motions. *How many positions does the tail of the rope visit at least once?*

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

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

--- Day 10: Cathode-Ray Tube ---

You avoid the ropes, plunge into the river, and swim to shore.

The Elves yell something about meeting back up with them upriver, but the river is too loud to tell exactly what they're saying. They finish crossing the bridge and disappear from view.

Situations like this must be why the Elves prioritized getting the communication system on your handheld device working. You pull it out of your pack, but the amount of water slowly draining from a big crack in its screen tells you it probably won't be of much immediate use.

Unless, that is, you can design a replacement for the device's video system! It

seems to be some kind of cathode-ray tube screen and simple CPU that are both driven by a precise *clock circuit*. The clock circuit ticks at a constant rate; each tick is called a *cycle*.

Start by figuring out the signal being sent by the CPU. The CPU has a single register, `X`, which starts with the value `1`. It supports only two instructions:

- `addx V` takes *two cycles* to complete. *After* two cycles, the `X` register is increased by the value `V`. (`V` can be negative.)
- `noop` takes *one cycle* to complete. It has no other effect.

The CPU uses these instructions in a program (your puzzle input) to, somehow, tell the screen what to draw.

Consider the following small program:

```
noop
addx 3
addx -5
```

Execution of this program proceeds as follows:

- At the start of the first cycle, the `noop` instruction begins execution. During the first cycle, `X` is `1`. After the first cycle, the `noop` instruction finishes execution, doing nothing.
- At the start of the second cycle, the `addx 3` instruction begins execution. During the second cycle, `X` is still `1`.
- During the third cycle, `X` is still `1`. After the third cycle, the `addx 3` instruction finishes execution, setting `X` to `4`.
- At the start of the fourth cycle, the `addx -5` instruction begins execution. During the fourth cycle, `X` is still `4`.
- During the fifth cycle, `X` is still `4`. After the fifth cycle, the `addx -5` instruction finishes execution, setting `X` to `-1`.

Maybe you can learn something by looking at the value of the `X` register throughout execution. For now, consider the *signal strength* (the cycle number multiplied by the value of the `X` register) *during* the 20th cycle and every 40 cycles after that (that is, during the 20th, 60th, 100th, 140th, 180th, and 220th cycles).

For example, consider this larger program:

```
addx 15
addx -11
addx 6
addx -3
addx 5
addx -1
addx -8
addx 13
addx 4
noop
```

```
addx -1
addx 5
addx -1
addx 5
addx -1
addx 5
addx -1
addx 5
addx -1
addx -35
addx 1
addx 24
addx -19
addx 1
addx 16
addx -11
noop
noop
addx 21
addx -15
noop
noop
addx -3
addx 9
addx 1
addx -3
addx 8
addx 1
addx 5
noop
noop
noop
noop
noop
addx -36
noop
addx 1
addx 7
noop
noop
noop
addx 2
addx 6
noop
noop
noop
```

```
noop
noop
addx 1
noop
noop
addx 7
addx 1
noop
addx -13
addx 13
addx 7
noop
addx 1
addx -33
noop
noop
noop
addx 2
noop
noop
noop
addx 8
noop
addx -1
addx 2
addx 1
noop
addx 17
addx -9
addx 1
addx 1
addx -3
addx 11
noop
noop
addx 1
noop
addx 1
noop
noop
addx -13
addx -19
addx 1
addx 3
addx 26
addx -30
```

```
addx 12
addx -1
addx 3
addx 1
noop
noop
noop
addx -9
addx 18
addx 1
addx 2
noop
noop
addx 9
noop
noop
noop
addx -1
addx 2
addx -37
addx 1
addx 3
noop
addx 15
addx -21
addx 22
addx -6
addx 1
noop
addx 2
addx 1
noop
addx -10
noop
noop
addx 20
addx 1
addx 2
addx 2
addx -6
addx -11
noop
noop
noop
```

The interesting signal strengths can be determined as follows:

- During the 20th cycle, register `X` has the value `21`, so the signal strength is 20 * 21 = *420*. (The 20th cycle occurs in the middle of the second `addx -1`, so the value of register `X` is the starting value, `1`, plus all of the other `addx` values up to that point: 1 + 15 - 11 + 6 - 3 + 5 - 1 - 8 + 13 + 4 = 21.)
- During the 60th cycle, register `X` has the value `19`, so the signal strength is 60 * 19 = `1140`.
- During the 100th cycle, register `X` has the value `18`, so the signal strength is 100 * 18 = `1800`.
- During the 140th cycle, register `X` has the value `21`, so the signal strength is 140 * 21 = `2940`.
- During the 180th cycle, register `X` has the value `16`, so the signal strength is 180 * 16 = `2880`.
- During the 220th cycle, register `X` has the value `18`, so the signal strength is 220 * 18 = `3960`.

The sum of these signal strengths is `13140`.

Find the signal strength during the 20th, 60th, 100th, 140th, 180th, and 220th cycles. *What is the sum of these six signal strengths?*

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

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

--- Day 11: Monkey in the Middle ---

As you finally start making your way upriver, you realize your pack is much lighter than you remember. Just then, one of the items from your pack goes flying overhead. Monkeys are playing Keep Away with your missing things!

To get your stuff back, you need to be able to predict where the monkeys will throw your items. After some careful observation, you realize the monkeys operate based on *how worried you are about each item.*

You take some notes (your puzzle input) on the items each monkey currently has, how worried you are about those items, and how the monkey makes decisions based on your worry level. For example:

```
Monkey 0:
  Starting items: 79, 98
  Operation: new = old * 19
  Test: divisible by 23
    If true: throw to monkey 2
    If false: throw to monkey 3

Monkey 1:
  Starting items: 54, 65, 75, 74
  Operation: new = old + 6
```

```
  Test: divisible by 19
    If true: throw to monkey 2
    If false: throw to monkey 0

Monkey 2:
  Starting items: 79, 60, 97
  Operation: new = old * old
  Test: divisible by 13
    If true: throw to monkey 1
    If false: throw to monkey 3

Monkey 3:
  Starting items: 74
  Operation: new = old + 3
  Test: divisible by 17
    If true: throw to monkey 0
    If false: throw to monkey 1
```

Each monkey has several attributes:

- `Starting items` lists your *worry level* for each item the monkey is currently holding in the order they will be inspected.
- `Operation` shows how your worry level changes as that monkey inspects an item. (An operation like `new = old * 5` means that your worry level after the monkey inspected the item is five times whatever your worry level was before inspection.)
- `Test` shows how the monkey uses your worry level to decide where to throw an item next.
  - `If true` shows what happens with an item if the `Test` was true.
  - `If false` shows what happens with an item if the `Test` was false.

After each monkey inspects an item but before it tests your worry level, your relief that the monkey's inspection didn't damage the item causes your worry level to be *divided by three* and rounded down to the nearest integer.

The monkeys take turns inspecting and throwing items. On a single monkey's *turn*, it inspects and throws all of the items it is holding one at a time and in the order listed. Monkey `0` goes first, then monkey `1`, and so on until each monkey has had one turn. The process of each monkey taking a single turn is called a *round*.

When a monkey throws an item to another monkey, the item goes on the *end* of the recipient monkey's list. A monkey that starts a round with no items could end up inspecting and throwing many items by the time its turn comes around. If a monkey is holding no items at the start of its turn, its turn ends.

In the above example, the first round proceeds as follows:

```
Monkey 0:
```

```
  Monkey inspects an item with a worry level of 79.
    Worry level is multiplied by 19 to 1501.
    Monkey gets bored with item. Worry level is divided by 3 to 500.
    Current worry level is not divisible by 23.
    Item with worry level 500 is thrown to monkey 3.
  Monkey inspects an item with a worry level of 98.
    Worry level is multiplied by 19 to 1862.
    Monkey gets bored with item. Worry level is divided by 3 to 620.
    Current worry level is not divisible by 23.
    Item with worry level 620 is thrown to monkey 3.
Monkey 1:
  Monkey inspects an item with a worry level of 54.
    Worry level increases by 6 to 60.
    Monkey gets bored with item. Worry level is divided by 3 to 20.
    Current worry level is not divisible by 19.
    Item with worry level 20 is thrown to monkey 0.
  Monkey inspects an item with a worry level of 65.
    Worry level increases by 6 to 71.
    Monkey gets bored with item. Worry level is divided by 3 to 23.
    Current worry level is not divisible by 19.
    Item with worry level 23 is thrown to monkey 0.
  Monkey inspects an item with a worry level of 75.
    Worry level increases by 6 to 81.
    Monkey gets bored with item. Worry level is divided by 3 to 27.
    Current worry level is not divisible by 19.
    Item with worry level 27 is thrown to monkey 0.
  Monkey inspects an item with a worry level of 74.
    Worry level increases by 6 to 80.
    Monkey gets bored with item. Worry level is divided by 3 to 26.
    Current worry level is not divisible by 19.
    Item with worry level 26 is thrown to monkey 0.
Monkey 2:
  Monkey inspects an item with a worry level of 79.
    Worry level is multiplied by itself to 6241.
    Monkey gets bored with item. Worry level is divided by 3 to 2080.
    Current worry level is divisible by 13.
    Item with worry level 2080 is thrown to monkey 1.
  Monkey inspects an item with a worry level of 60.
    Worry level is multiplied by itself to 3600.
    Monkey gets bored with item. Worry level is divided by 3 to 1200.
    Current worry level is not divisible by 13.
    Item with worry level 1200 is thrown to monkey 3.
  Monkey inspects an item with a worry level of 97.
    Worry level is multiplied by itself to 9409.
    Monkey gets bored with item. Worry level is divided by 3 to 3136.
    Current worry level is not divisible by 13.
```

```
    Item with worry level 3136 is thrown to monkey 3.
Monkey 3:
  Monkey inspects an item with a worry level of 74.
    Worry level increases by 3 to 77.
    Monkey gets bored with item. Worry level is divided by 3 to 25.
    Current worry level is not divisible by 17.
    Item with worry level 25 is thrown to monkey 1.
  Monkey inspects an item with a worry level of 500.
    Worry level increases by 3 to 503.
    Monkey gets bored with item. Worry level is divided by 3 to 167.
    Current worry level is not divisible by 17.
    Item with worry level 167 is thrown to monkey 1.
  Monkey inspects an item with a worry level of 620.
    Worry level increases by 3 to 623.
    Monkey gets bored with item. Worry level is divided by 3 to 207.
    Current worry level is not divisible by 17.
    Item with worry level 207 is thrown to monkey 1.
  Monkey inspects an item with a worry level of 1200.
    Worry level increases by 3 to 1203.
    Monkey gets bored with item. Worry level is divided by 3 to 401.
    Current worry level is not divisible by 17.
    Item with worry level 401 is thrown to monkey 1.
  Monkey inspects an item with a worry level of 3136.
    Worry level increases by 3 to 3139.
    Monkey gets bored with item. Worry level is divided by 3 to 1046.
    Current worry level is not divisible by 17.
    Item with worry level 1046 is thrown to monkey 1.
```

After round 1, the monkeys are holding items with these worry levels:

```
Monkey 0: 20, 23, 27, 26
Monkey 1: 2080, 25, 167, 207, 401, 1046
Monkey 2:
Monkey 3:
```

Monkeys 2 and 3 aren't holding any items at the end of the round; they both inspected items during the round and threw them all before the round ended.

This process continues for a few more rounds:

```
After round 2, the monkeys are holding items with these worry levels:
Monkey 0: 695, 10, 71, 135, 350
Monkey 1: 43, 49, 58, 55, 362
Monkey 2:
Monkey 3:

After round 3, the monkeys are holding items with these worry levels:
Monkey 0: 16, 18, 21, 20, 122
```

```
Monkey 1: 1468, 22, 150, 286, 739
Monkey 2:
Monkey 3:

After round 4, the monkeys are holding items with these worry levels:
Monkey 0: 491, 9, 52, 97, 248, 34
Monkey 1: 39, 45, 43, 258
Monkey 2:
Monkey 3:

After round 5, the monkeys are holding items with these worry levels:
Monkey 0: 15, 17, 16, 88, 1037
Monkey 1: 20, 110, 205, 524, 72
Monkey 2:
Monkey 3:

After round 6, the monkeys are holding items with these worry levels:
Monkey 0: 8, 70, 176, 26, 34
Monkey 1: 481, 32, 36, 186, 2190
Monkey 2:
Monkey 3:

After round 7, the monkeys are holding items with these worry levels:
Monkey 0: 162, 12, 14, 64, 732, 17
Monkey 1: 148, 372, 55, 72
Monkey 2:
Monkey 3:

After round 8, the monkeys are holding items with these worry levels:
Monkey 0: 51, 126, 20, 26, 136
Monkey 1: 343, 26, 30, 1546, 36
Monkey 2:
Monkey 3:

After round 9, the monkeys are holding items with these worry levels:
Monkey 0: 116, 10, 12, 517, 14
Monkey 1: 108, 267, 43, 55, 288
Monkey 2:
Monkey 3:

After round 10, the monkeys are holding items with these worry levels:
Monkey 0: 91, 16, 20, 98
Monkey 1: 481, 245, 22, 26, 1092, 30
Monkey 2:
Monkey 3:
```

```
...

After round 15, the monkeys are holding items with these worry levels:
Monkey 0: 83, 44, 8, 184, 9, 20, 26, 102
Monkey 1: 110, 36
Monkey 2:
Monkey 3:

...

After round 20, the monkeys are holding items with these worry levels:
Monkey 0: 10, 12, 14, 26, 34
Monkey 1: 245, 93, 53, 199, 115
Monkey 2:
Monkey 3:
```

Chasing all of the monkeys at once is impossible; you're going to have to focus on the *two most active* monkeys if you want any hope of getting your stuff back. Count the *total number of times each monkey inspects items* over 20 rounds:

```
Monkey 0 inspected items 101 times.
Monkey 1 inspected items 95 times.
Monkey 2 inspected items 7 times.
Monkey 3 inspected items 105 times.
```

In this example, the two most active monkeys inspected items 101 and 105 times. The level of *monkey business* in this situation can be found by multiplying these together: `10605`.

Figure out which monkeys to chase by counting how many items they inspect over 20 rounds. *What is the level of monkey business after 20 rounds of stuff-slinging simian shenanigans?*

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

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

--- Day 12: Hill Climbing Algorithm ---

You try contacting the Elves using your handheld device, but the river you're following must be too low to get a decent signal.

You ask the device for a heightmap of the surrounding area (your puzzle input). The heightmap shows the local area from above broken into a grid; the elevation of each square of the grid is given by a single lowercase letter, where `a` is the lowest elevation, `b` is the next-lowest, and so on up to the highest elevation, `z`.

Also included on the heightmap are marks for your current position (`S`) and the location that should get the best signal (`E`). Your current position (`S`) has elevation `a`, and the location that should get the best signal (`E`) has elevation `z`.

You'd like to reach `E`, but to save energy, you should do it in *as few steps as possible*. During each step, you can move exactly one square up, down, left, or right. To avoid needing to get out your climbing gear, the elevation of the destination square can be *at most one higher* than the elevation of your current square; that is, if your current elevation is `m`, you could step to elevation `n`, but not to elevation `o`. (This also means that the elevation of the destination square can be much lower than the elevation of your current square.)

For example:

```
Sabqponm
abcryxxl
accszExk
acctuvwj
abdefghi
```

Here, you start in the top-left corner; your goal is near the middle. You could start by moving down or right, but eventually you'll need to head toward the `e` at the bottom. From there, you can spiral around to the goal:

```
v..v<<<<
>v.vv<<^
.>vv>E^^
..v>>>^^
..>>>>>^
```

In the above diagram, the symbols indicate whether the path exits each square moving up (`^`), down (`v`), left (`<`), or right (`>`). The location that should get the best signal is still `E`, and `.` marks unvisited squares.

This path reaches the goal in `31` steps, the fewest possible.

What is the fewest steps required to move from your current position to the location that should get the best signal?

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

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

--- Day 13: Distress Signal ---

You climb the hill and again try contacting the Elves. However, you instead receive a signal you weren't expecting: a *distress signal*.

Your handheld device must still not be working properly; the packets from the distress signal got decoded *out of order*. You'll need to re-order the list of received packets (your puzzle input) to decode the message.

Your list consists of pairs of packets; pairs are separated by a blank line. You need to identify *how many pairs of packets are in the right order*.

For example:

```
[1,1,3,1,1]
[1,1,5,1,1]

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

[9]
[[8,7,6]]

[[4,4],4,4]
[[4,4],4,4,4]

[7,7,7,7]
[7,7,7]

[]
[3]

[[[]]]
[[]]

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

Packet data consists of lists and integers. Each list starts with `[`, ends with `]`, and contains zero or more comma-separated values (either integers or other lists). Each packet is always a list and appears on its own line.

When comparing two values, the first value is called *left* and the second value is called *right*. Then:

- If *both values are integers*, the *lower integer* should come first. If the left integer is lower than the right integer, the inputs are in the right order. If the left integer is higher than the right integer, the inputs are not in the right order. Otherwise, the inputs are the same integer; continue checking the next part of the input.
- If *both values are lists*, compare the first value of each list, then the second value, and so on. If the left list runs out of items first, the inputs are in the right order. If the right list runs out of items first, the inputs are not in the right order. If the lists are the same length and no comparison makes a decision about the order, continue checking the next part of the input.
- If *exactly one value is an integer*, convert the integer to a list which contains that integer as its only value, then retry the comparison. For example, if comparing `[0,0,0]` and `2`, convert the right value to `[2]` (a list containing `2`); the result is then found by instead comparing `[0,0,0]` and `[2]`.

Using these rules, you can determine which of the pairs in the example are in the right order:

```
== Pair 1 ==
- Compare [1,1,3,1,1] vs [1,1,5,1,1]
  - Compare 1 vs 1
  - Compare 1 vs 1
  - Compare 3 vs 5
    - Left side is smaller, so inputs are in the right order

== Pair 2 ==
- Compare [[1],[2,3,4]] vs [[1],4]
  - Compare [1] vs [1]
    - Compare 1 vs 1
  - Compare [2,3,4] vs 4
    - Mixed types; convert right to [4] and retry comparison
    - Compare [2,3,4] vs [4]
      - Compare 2 vs 4
        - Left side is smaller, so inputs are in the right order

== Pair 3 ==
- Compare [9] vs [[8,7,6]]
  - Compare 9 vs [8,7,6]
    - Mixed types; convert left to [9] and retry comparison
    - Compare [9] vs [8,7,6]
      - Compare 9 vs 8
        - Right side is smaller, so inputs are not in the right order

== Pair 4 ==
- Compare [[4,4],4,4] vs [[4,4],4,4,4]
  - Compare [4,4] vs [4,4]
    - Compare 4 vs 4
    - Compare 4 vs 4
  - Compare 4 vs 4
  - Compare 4 vs 4
  - Left side ran out of items, so inputs are in the right order

== Pair 5 ==
- Compare [7,7,7,7] vs [7,7,7]
  - Compare 7 vs 7
  - Compare 7 vs 7
  - Compare 7 vs 7
  - Right side ran out of items, so inputs are not in the right order

== Pair 6 ==
- Compare [] vs [3]
  - Left side ran out of items, so inputs are in the right order

== Pair 7 ==
```

```
- Compare [[[]]] vs [[]]
  - Compare [[]] vs []
    - Right side ran out of items, so inputs are not in the right order

== Pair 8 ==
- Compare [1,[2,[3,[4,[5,6,7]]]],8,9] vs [1,[2,[3,[4,[5,6,0]]]],8,9]
  - Compare 1 vs 1
  - Compare [2,[3,[4,[5,6,7]]]] vs [2,[3,[4,[5,6,0]]]]
    - Compare 2 vs 2
    - Compare [3,[4,[5,6,7]]] vs [3,[4,[5,6,0]]]
      - Compare 3 vs 3
      - Compare [4,[5,6,7]] vs [4,[5,6,0]]
        - Compare 4 vs 4
        - Compare [5,6,7] vs [5,6,0]
          - Compare 5 vs 5
          - Compare 6 vs 6
          - Compare 7 vs 0
            - Right side is smaller, so inputs are not in the right order
```

What are the indices of the pairs that are already *in the right order*? (The first pair has index 1, the second pair has index 2, and so on.) In the above example, the pairs in the right order are 1, 2, 4, and 6; the sum of these indices is `13`.

Determine which pairs of packets are already in the right order. *What is the sum of the indices of those pairs?*

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

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

--- Day 14: Regolith Reservoir ---

The distress signal leads you to a giant waterfall! Actually, hang on - the signal seems like it's coming from the waterfall itself, and that doesn't make any sense. However, you do notice a little path that leads *behind* the waterfall.

Correction: the distress signal leads you behind a giant waterfall! There seems to be a large cave system here, and the signal definitely leads further inside.

As you begin to make your way deeper underground, you feel the ground rumble for a moment. Sand begins pouring into the cave! If you don't quickly figure out where the sand is going, you could quickly become trapped!

Fortunately, your familiarity with analyzing the path of falling material will come in handy here. You scan a two-dimensional vertical slice of the cave above you (your puzzle input) and discover that it is mostly *air* with structures made of *rock*.

Your scan traces the path of each solid rock structure and reports the `x,y` coordinates that form the shape of the path, where `x` represents distance to the

right and `y` represents distance down. Each path appears as a single line of text in your scan. After the first point of each path, each point indicates the end of a straight horizontal or vertical line to be drawn from the previous point. For example:

```
498,4 -> 498,6 -> 496,6
503,4 -> 502,4 -> 502,9 -> 494,9
```

This scan means that there are two paths of rock; the first path consists of two straight lines, and the second path consists of three straight lines. (Specifically, the first path consists of a line of rock from `498,4` through `498,6` and another line of rock from `498,6` through `496,6`.)

The sand is pouring into the cave from point `500,0`.

Drawing rock as `#`, air as `.`, and the source of the sand as `+`, this becomes:

```
  4     5  5
  9     0  0
  4     0  3
0 ......+...
1 ..........
2 ..........
3 ..........
4 ....#...##
5 ....#...#.
6 ..###...#.
7 ........#.
8 ........#.
9 #########.
```

Sand is produced *one unit at a time*, and the next unit of sand is not produced until the previous unit of sand *comes to rest*. A unit of sand is large enough to fill one tile of air in your scan.

A unit of sand always falls *down one step* if possible. If the tile immediately below is blocked (by rock or sand), the unit of sand attempts to instead move diagonally *one step down and to the left*. If that tile is blocked, the unit of sand attempts to instead move diagonally *one step down and to the right*. Sand keeps moving as long as it is able to do so, at each step trying to move down, then down-left, then down-right. If all three possible destinations are blocked, the unit of sand *comes to rest* and no longer moves, at which point the next unit of sand is created back at the source.

So, drawing sand that has come to rest as `o`, the first unit of sand simply falls straight down and then stops:

```
......+...
..........
..........
```

```
..........
....#...##
....#...#.
..###...#.
........#.
......o.#.
#########.
```

The second unit of sand then falls straight down, lands on the first one, and then comes to rest to its left:

```
......+...
..........
..........
..........
....#...##
....#...#.
..###...#.
........#.
.....oo.#.
#########.
```

After a total of five units of sand have come to rest, they form this pattern:

```
......+...
..........
..........
..........
....#...##
....#...#.
..###...#.
......o.#.
....oooo#.
#########.
```

After a total of 22 units of sand:

```
......+...
..........
......o...
.....ooo..
....#ooo##
....#ooo#.
..###ooo#.
....oooo#.
...ooooo#.
#########.
```

Finally, only two more units of sand can possibly come to rest:

```
......+...
..........
......o...
.....ooo..
....#ooo##
...o#ooo#.
..###ooo#.
....oooo#.
.o.ooooo#.
#########.
```

Once all 24 units of sand shown above have come to rest, all further sand flows out the bottom, falling into the endless void. Just for fun, the path any new sand takes before falling forever is shown here with ~:

```
.......+...
.......~...
......~o...
.....~ooo..
....~#ooo##
...~o#ooo#.
..~###ooo#.
..~..oooo#.
.~o.ooooo#.
~#########.
~..........
~..........
~..........
```

Using your scan, simulate the falling sand. *How many units of sand come to rest before sand starts flowing into the abyss below?*

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

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

--- Day 15: Beacon Exclusion Zone ---

You feel the ground rumble again as the distress signal leads you to a large network of subterranean tunnels. You don't have time to search them all, but you don't need to: your pack contains a set of deployable *sensors* that you imagine were originally built to locate lost Elves.

The sensors aren't very powerful, but that's okay; your handheld device indicates that you're close enough to the source of the distress signal to use them. You pull the emergency sensor system out of your pack, hit the big button on top, and the sensors zoom off down the tunnels.

Once a sensor finds a spot it thinks will give it a good reading, it attaches itself

to a hard surface and begins monitoring for the nearest signal source *beacon*. Sensors and beacons always exist at integer coordinates. Each sensor knows its own position and can *determine the position of a beacon precisely*; however, sensors can only lock on to the one beacon *closest to the sensor* as measured by the Manhattan distance. (There is never a tie where two beacons are the same distance to a sensor.)

It doesn't take long for the sensors to report back their positions and closest beacons (your puzzle input). For example:

```
Sensor at x=2, y=18: closest beacon is at x=-2, y=15
Sensor at x=9, y=16: closest beacon is at x=10, y=16
Sensor at x=13, y=2: closest beacon is at x=15, y=3
Sensor at x=12, y=14: closest beacon is at x=10, y=16
Sensor at x=10, y=20: closest beacon is at x=10, y=16
Sensor at x=14, y=17: closest beacon is at x=10, y=16
Sensor at x=8, y=7: closest beacon is at x=2, y=10
Sensor at x=2, y=0: closest beacon is at x=2, y=10
Sensor at x=0, y=11: closest beacon is at x=2, y=10
Sensor at x=20, y=14: closest beacon is at x=25, y=17
Sensor at x=17, y=20: closest beacon is at x=21, y=22
Sensor at x=16, y=7: closest beacon is at x=15, y=3
Sensor at x=14, y=3: closest beacon is at x=15, y=3
Sensor at x=20, y=1: closest beacon is at x=15, y=3
```

So, consider the sensor at `2,18`; the closest beacon to it is at `-2,15`. For the sensor at `9,16`, the closest beacon to it is at `10,16`.

Drawing sensors as `S` and beacons as `B`, the above arrangement of sensors and beacons looks like this:

```
               1    1    2    2
     0    5    0    5    0    5
 0 ....S.......................
 1 ......................S.....
 2 ................S.............
 3 .................SB..........
 4 ............................
 5 ............................
 6 ............................
 7 ..........S.......S.........
 8 ............................
 9 ............................
10 ....B.......................
11 ..S.........................
12 ............................
13 ............................
14 ..............S.......S.....
```

```
15 B.........................
16 ...........SB..............
17 ................S.........B
18 ....S......................
19 .........................
20 ............S......S........
21 .........................
22 ......................B....
```

This isn't necessarily a comprehensive map of all beacons in the area, though. Because each sensor only identifies its closest beacon, if a sensor detects a beacon, you know there are no other beacons that close or closer to that sensor. There could still be beacons that just happen to not be the closest beacon to any sensor. Consider the sensor at `8,7`:

```
               1    1    2    2
     0    5    0    5    0    5
-2 ..........#.................
-1 .........###................
 0 ....S...#####...............
 1 .......#######........S.....
 2 ......#########S............
 3 .....###########SB..........
 4 ....#############...........
 5 ...###############..........
 6 ..#################.........
 7 .#########S#######S#........
 8 ..#################.........
 9 ...###############..........
10 ....B############...........
11 ..S..###########............
12 ......#########.............
13 .......#######..............
14 ........#####.S.......S.....
15 B........###................
16 ..........#SB...............
17 ................S..........B
18 ....S......................
19 .........................
20 ............S......S........
21 .........................
22 ......................B....
```

This sensor's closest beacon is at `2,10`, and so you know there are no beacons that close or closer (in any positions marked `#`).

None of the detected beacons seem to be producing the distress signal, so you'll need to work out where the distress beacon is by working out where it *isn't*.

For now, keep things simple by counting the positions where a beacon cannot possibly be along just a single row.

So, suppose you have an arrangement of beacons and sensors like in the example above and, just in the row where `y=10`, you'd like to count the number of positions a beacon cannot possibly exist. The coverage from all sensors near that row looks like this:

```
                 1    1    2    2
       0    5    0    5    0    5
 9 ...#########################...
10 ..####B######################..
11 .###S#############.###########.
```

In this example, in the row where `y=10`, there are `26` positions where a beacon cannot be present.

Consult the report from the sensors you just deployed. *In the row where `y=2000000`, how many positions cannot contain a beacon?*

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

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

--- Day 16: Proboscidea Volcanium ---

The sensors have led you to the origin of the distress signal: yet another handheld device, just like the one the Elves gave you. However, you don't see any Elves around; instead, the device is surrounded by elephants! They must have gotten lost in these tunnels, and one of the elephants apparently figured out how to turn on the distress signal.

The ground rumbles again, much stronger this time. What kind of cave is this, exactly? You scan the cave with your handheld device; it reports mostly igneous rock, some ash, pockets of pressurized gas, magma... this isn't just a cave, it's a volcano!

You need to get the elephants out of here, quickly. Your device estimates that you have *30 minutes* before the volcano erupts, so you don't have time to go back out the way you came in.

You scan the cave for other options and discover a network of pipes and pressure-release *valves*. You aren't sure how such a system got into a volcano, but you don't have time to complain; your device produces a report (your puzzle input) of each valve's *flow rate* if it were opened (in pressure per minute) and the tunnels you could use to move between the valves.

There's even a valve in the room you and the elephants are currently standing in labeled `AA`. You estimate it will take you one minute to open a single valve and one minute to follow any tunnel from one valve to another. What is the most pressure you could release?

For example, suppose you had the following scan output:

```
Valve AA has flow rate=0; tunnels lead to valves DD, II, BB
Valve BB has flow rate=13; tunnels lead to valves CC, AA
Valve CC has flow rate=2; tunnels lead to valves DD, BB
Valve DD has flow rate=20; tunnels lead to valves CC, AA, EE
Valve EE has flow rate=3; tunnels lead to valves FF, DD
Valve FF has flow rate=0; tunnels lead to valves EE, GG
Valve GG has flow rate=0; tunnels lead to valves FF, HH
Valve HH has flow rate=22; tunnel leads to valve GG
Valve II has flow rate=0; tunnels lead to valves AA, JJ
Valve JJ has flow rate=21; tunnel leads to valve II
```

All of the valves begin *closed*. You start at valve `AA`, but it must be damaged or jammed or something: its flow rate is `0`, so there's no point in opening it. However, you could spend one minute moving to valve `BB` and another minute opening it; doing so would release pressure during the remaining *28 minutes* at a flow rate of `13`, a total eventual pressure release of `28 * 13 = 364`. Then, you could spend your third minute moving to valve `CC` and your fourth minute opening it, providing an additional *26 minutes* of eventual pressure release at a flow rate of `2`, or `52` total pressure released by valve `CC`.

Making your way through the tunnels like this, you could probably open many or all of the valves by the time 30 minutes have elapsed. However, you need to release as much pressure as possible, so you'll need to be methodical. Instead, consider this approach:

```
== Minute 1 ==
No valves are open.
You move to valve DD.

== Minute 2 ==
No valves are open.
You open valve DD.

== Minute 3 ==
Valve DD is open, releasing 20 pressure.
You move to valve CC.

== Minute 4 ==
Valve DD is open, releasing 20 pressure.
You move to valve BB.

== Minute 5 ==
Valve DD is open, releasing 20 pressure.
You open valve BB.

== Minute 6 ==
```

```
Valves BB and DD are open, releasing 33 pressure.
You move to valve AA.

== Minute 7 ==
Valves BB and DD are open, releasing 33 pressure.
You move to valve II.

== Minute 8 ==
Valves BB and DD are open, releasing 33 pressure.
You move to valve JJ.

== Minute 9 ==
Valves BB and DD are open, releasing 33 pressure.
You open valve JJ.

== Minute 10 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve II.

== Minute 11 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve AA.

== Minute 12 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve DD.

== Minute 13 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve EE.

== Minute 14 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve FF.

== Minute 15 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve GG.

== Minute 16 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You move to valve HH.

== Minute 17 ==
Valves BB, DD, and JJ are open, releasing 54 pressure.
You open valve HH.
```

```
== Minute 18 ==
Valves BB, DD, HH, and JJ are open, releasing 76 pressure.
You move to valve GG.

== Minute 19 ==
Valves BB, DD, HH, and JJ are open, releasing 76 pressure.
You move to valve FF.

== Minute 20 ==
Valves BB, DD, HH, and JJ are open, releasing 76 pressure.
You move to valve EE.

== Minute 21 ==
Valves BB, DD, HH, and JJ are open, releasing 76 pressure.
You open valve EE.

== Minute 22 ==
Valves BB, DD, EE, HH, and JJ are open, releasing 79 pressure.
You move to valve DD.

== Minute 23 ==
Valves BB, DD, EE, HH, and JJ are open, releasing 79 pressure.
You move to valve CC.

== Minute 24 ==
Valves BB, DD, EE, HH, and JJ are open, releasing 79 pressure.
You open valve CC.

== Minute 25 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.

== Minute 26 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.

== Minute 27 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.

== Minute 28 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.

== Minute 29 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.

== Minute 30 ==
Valves BB, CC, DD, EE, HH, and JJ are open, releasing 81 pressure.
```

This approach lets you release the most pressure possible in 30 minutes with this valve layout, `1651`.

Work out the steps to release the most pressure in 30 minutes. *What is the most pressure you can release?*

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

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

--- Day 17: Pyroclastic Flow ---

Your handheld device has located an alternative exit from the cave for you and the elephants. The ground is rumbling almost continuously now, but the strange valves bought you some time. It's definitely getting warmer in here, though.

The tunnels eventually open into a very tall, narrow chamber. Large, oddly-shaped rocks are falling into the chamber from above, presumably due to all the rumbling. If you can't work out where the rocks will fall next, you might be crushed!

The five types of rocks have the following peculiar shapes, where `#` is rock and `.` is empty space:

```
####

.#.
###
.#.

..#
..#
###

#
#
#
#

##
##
```

The rocks fall in the order shown above: first the `-` shape, then the `+` shape, and so on. Once the end of the list is reached, the same order repeats: the `-` shape falls first, sixth, 11th, 16th, etc.

The rocks don't spin, but they do get pushed around by jets of hot gas coming out of the walls themselves. A quick scan reveals the effect the jets of hot gas will have on the rocks as they fall (your puzzle input).

For example, suppose this was the jet pattern in your cave:

```
>>><<><>><<<>><>>><<<>>><<<><<<>>><>><<>>
```

In jet patterns, `<` means a push to the left, while `>` means a push to the right. The pattern above means that the jets will push a falling rock right, then right, then right, then left, then left, then right, and so on. If the end of the list is reached, it repeats.

The tall, vertical chamber is exactly *seven units wide*. Each rock appears so that its left edge is two units away from the left wall and its bottom edge is three units above the highest rock in the room (or the floor, if there isn't one).

After a rock appears, it alternates between *being pushed by a jet of hot gas* one unit (in the direction indicated by the next symbol in the jet pattern) and then *falling one unit down*. If any movement would cause any part of the rock to move into the walls, floor, or a stopped rock, the movement instead does not occur. If a *downward* movement would have caused a falling rock to move into the floor or an already-fallen rock, the falling rock stops where it is (having landed on something) and a new rock immediately begins falling.

Drawing falling rocks with `@` and stopped rocks with `#`, the jet pattern in the example above manifests as follows:

```
The first rock begins falling:
|..@@@@.|
|.......|
|.......|
|.......|
+-------+

Jet of gas pushes rock right:
|...@@@@|
|.......|
|.......|
|.......|
+-------+

Rock falls 1 unit:
|...@@@@|
|.......|
|.......|
+-------+

Jet of gas pushes rock right, but nothing happens:
|...@@@@|
|.......|
|.......|
+-------+
```

```
Rock falls 1 unit:
|...@@@@|
|.......|
+-------+

Jet of gas pushes rock right, but nothing happens:
|...@@@@|
|.......|
+-------+

Rock falls 1 unit:
|...@@@@|
+-------+

Jet of gas pushes rock left:
|..@@@@.|
+-------+

Rock falls 1 unit, causing it to come to rest:
|..####.|
+-------+

A new rock begins falling:
|...@...|
|..@@@..|
|...@...|
|.......|
|.......|
|.......|
|..####.|
+-------+

Jet of gas pushes rock left:
|..@....|
|.@@@...|
|..@....|
|.......|
|.......|
|.......|
|..####.|
+-------+

Rock falls 1 unit:
|..@....|
|.@@@...|
```

```
|..@....|
|.......|
|.......|
|..####.|
+-------+

Jet of gas pushes rock right:
|...@...|
|..@@@..|
|...@...|
|.......|
|.......|
|..####.|
+-------+

Rock falls 1 unit:
|...@...|
|..@@@..|
|...@...|
|.......|
|..####.|
+-------+

Jet of gas pushes rock left:
|..@....|
|.@@@...|
|..@....|
|.......|
|..####.|
+-------+

Rock falls 1 unit:
|..@....|
|.@@@...|
|..@....|
|..####.|
+-------+

Jet of gas pushes rock right:
|...@...|
|..@@@..|
|...@...|
|..####.|
+-------+

Rock falls 1 unit, causing it to come to rest:
```

```
|...#...|
|..###..|
|...#...|
|..####.|
+-------+

A new rock begins falling:
|....@..|
|....@..|
|..@@@..|
|.......|
|.......|
|.......|
|...#...|
|..###..|
|...#...|
|..####.|
+-------+
```

The moment each of the next few rocks begins falling, you would see this:

```
|..@....|
|..@....|
|..@....|
|..@....|
|.......|
|.......|
|.......|
|..#....|
|..#....|
|####...|
|..###..|
|...#...|
|..####.|
+-------+

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

```
|..####.|
+-------+

|..@@@@.|
|.......|
|.......|
|.......|
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+

|...@...|
|..@@@..|
|...@...|
|.......|
|.......|
|.......|
|.####..|
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+

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

```
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+

|..@....|
|..@....|
|..@....|
|..@....|
|.......|
|.......|
|.......|
|.....#.|
|.....#.|
|..####.|
|.###...|
|..#....|
|.####..|
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+

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

```
|..#....|
|.####..|
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+

|..@@@@.|
|.......|
|.......|
|.......|
|....#..|
|....#..|
|....##.|
|##..##.|
|######.|
|.###...|
|..#....|
|.####..|
|....##.|
|....##.|
|....#..|
|..#.#..|
|..#.#..|
|#####..|
|..###..|
|...#...|
|..####.|
+-------+
```

To prove to the elephants your simulation is accurate, they want to know how tall the tower will get after 2022 rocks have stopped (but before the 2023rd rock begins falling). In this example, the tower of rocks will be `3068` units tall.

How many units tall will the tower of rocks be after 2022 rocks have stopped falling?

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

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

--- Day 18: Boiling Boulders ---

You and the elephants finally reach fresh air. You've emerged near the base of a large volcano that seems to be actively erupting! Fortunately, the lava seems to be flowing away from you and toward the ocean.

Bits of lava are still being ejected toward you, so you're sheltering in the cavern exit a little longer. Outside the cave, you can see the lava landing in a pond and hear it loudly hissing as it solidifies.

Depending on the specific compounds in the lava and speed at which it cools, it might be forming obsidian! The cooling rate should be based on the surface area of the lava droplets, so you take a quick scan of a droplet as it flies past you (your puzzle input).

Because of how quickly the lava is moving, the scan isn't very good; its resolution is quite low and, as a result, it approximates the shape of the lava droplet with *1x1x1 cubes on a 3D grid*, each given as its `x,y,z` position.

To approximate the surface area, count the number of sides of each cube that are not immediately connected to another cube. So, if your scan were only two adjacent cubes like `1,1,1` and `2,1,1`, each cube would have a single side covered and five sides exposed, a total surface area of `10` sides.

Here's a larger example:

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

In the above example, after counting up all the sides that aren't connected to another cube, the total surface area is `64`.

What is the surface area of your scanned lava droplet?

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

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

--- Day 19: Not Enough Minerals ---

Your scans show that the lava did indeed form obsidian!

The wind has changed direction enough to stop sending lava droplets toward you, so you and the elephants exit the cave. As you do, you notice a collection of geodes around the pond. Perhaps you could use the obsidian to create some *geode-cracking robots* and break them open?

To collect the obsidian from the bottom of the pond, you'll need waterproof *obsidian-collecting robots*. Fortunately, there is an abundant amount of clay nearby that you can use to make them waterproof.

In order to harvest the clay, you'll need special-purpose *clay-collecting robots*. To make any type of robot, you'll need *ore*, which is also plentiful but in the opposite direction from the clay.

Collecting ore requires *ore-collecting robots* with big drills. Fortunately, *you have exactly one ore-collecting robot* in your pack that you can use to kickstart the whole operation.

Each robot can collect 1 of its resource type per minute. It also takes one minute for the robot factory (also conveniently from your pack) to construct any type of robot, although it consumes the necessary resources available when construction begins.

The robot factory has many *blueprints* (your puzzle input) you can choose from, but once you've configured it with a blueprint, you can't change it. You'll need to work out which blueprint is best.

For example:

```
Blueprint 1:
  Each ore robot costs 4 ore.
  Each clay robot costs 2 ore.
  Each obsidian robot costs 3 ore and 14 clay.
  Each geode robot costs 2 ore and 7 obsidian.

Blueprint 2:
  Each ore robot costs 2 ore.
  Each clay robot costs 3 ore.
  Each obsidian robot costs 3 ore and 8 clay.
  Each geode robot costs 3 ore and 12 obsidian.
```

(Blueprints have been line-wrapped here for legibility. The robot factory's actual assortment of blueprints are provided one blueprint per line.)

The elephants are starting to look hungry, so you shouldn't take too long; you need to figure out which blueprint would maximize the number of opened geodes after *24 minutes* by figuring out which robots to build and when to build them.

Using blueprint 1 in the example above, the largest number of geodes you could open in 24 minutes is `9`. One way to achieve that is:

```
== Minute 1 ==
1 ore-collecting robot collects 1 ore; you now have 1 ore.

== Minute 2 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.

== Minute 3 ==
Spend 2 ore to start building a clay-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 1 ore.
The new clay-collecting robot is ready; you now have 1 of them.

== Minute 4 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.
1 clay-collecting robot collects 1 clay; you now have 1 clay.

== Minute 5 ==
Spend 2 ore to start building a clay-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 1 ore.
1 clay-collecting robot collects 1 clay; you now have 2 clay.
The new clay-collecting robot is ready; you now have 2 of them.

== Minute 6 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.
2 clay-collecting robots collect 2 clay; you now have 4 clay.

== Minute 7 ==
Spend 2 ore to start building a clay-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 1 ore.
2 clay-collecting robots collect 2 clay; you now have 6 clay.
The new clay-collecting robot is ready; you now have 3 of them.

== Minute 8 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.
3 clay-collecting robots collect 3 clay; you now have 9 clay.

== Minute 9 ==
1 ore-collecting robot collects 1 ore; you now have 3 ore.
3 clay-collecting robots collect 3 clay; you now have 12 clay.

== Minute 10 ==
1 ore-collecting robot collects 1 ore; you now have 4 ore.
3 clay-collecting robots collect 3 clay; you now have 15 clay.
```

```
== Minute 11 ==
Spend 3 ore and 14 clay to start building an obsidian-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 2 ore.
3 clay-collecting robots collect 3 clay; you now have 4 clay.
The new obsidian-collecting robot is ready; you now have 1 of them.

== Minute 12 ==
Spend 2 ore to start building a clay-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 1 ore.
3 clay-collecting robots collect 3 clay; you now have 7 clay.
1 obsidian-collecting robot collects 1 obsidian; you now have 1 obsidian.
The new clay-collecting robot is ready; you now have 4 of them.

== Minute 13 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.
4 clay-collecting robots collect 4 clay; you now have 11 clay.
1 obsidian-collecting robot collects 1 obsidian; you now have 2 obsidian.

== Minute 14 ==
1 ore-collecting robot collects 1 ore; you now have 3 ore.
4 clay-collecting robots collect 4 clay; you now have 15 clay.
1 obsidian-collecting robot collects 1 obsidian; you now have 3 obsidian.

== Minute 15 ==
Spend 3 ore and 14 clay to start building an obsidian-collecting robot.
1 ore-collecting robot collects 1 ore; you now have 1 ore.
4 clay-collecting robots collect 4 clay; you now have 5 clay.
1 obsidian-collecting robot collects 1 obsidian; you now have 4 obsidian.
The new obsidian-collecting robot is ready; you now have 2 of them.

== Minute 16 ==
1 ore-collecting robot collects 1 ore; you now have 2 ore.
4 clay-collecting robots collect 4 clay; you now have 9 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 6 obsidian.

== Minute 17 ==
1 ore-collecting robot collects 1 ore; you now have 3 ore.
4 clay-collecting robots collect 4 clay; you now have 13 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 8 obsidian.

== Minute 18 ==
Spend 2 ore and 7 obsidian to start building a geode-cracking robot.
1 ore-collecting robot collects 1 ore; you now have 2 ore.
4 clay-collecting robots collect 4 clay; you now have 17 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 3 obsidian.
The new geode-cracking robot is ready; you now have 1 of them.
```

```
== Minute 19 ==
1 ore-collecting robot collects 1 ore; you now have 3 ore.
4 clay-collecting robots collect 4 clay; you now have 21 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 5 obsidian.
1 geode-cracking robot cracks 1 geode; you now have 1 open geode.

== Minute 20 ==
1 ore-collecting robot collects 1 ore; you now have 4 ore.
4 clay-collecting robots collect 4 clay; you now have 25 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 7 obsidian.
1 geode-cracking robot cracks 1 geode; you now have 2 open geodes.

== Minute 21 ==
Spend 2 ore and 7 obsidian to start building a geode-cracking robot.
1 ore-collecting robot collects 1 ore; you now have 3 ore.
4 clay-collecting robots collect 4 clay; you now have 29 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 2 obsidian.
1 geode-cracking robot cracks 1 geode; you now have 3 open geodes.
The new geode-cracking robot is ready; you now have 2 of them.

== Minute 22 ==
1 ore-collecting robot collects 1 ore; you now have 4 ore.
4 clay-collecting robots collect 4 clay; you now have 33 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 4 obsidian.
2 geode-cracking robots crack 2 geodes; you now have 5 open geodes.

== Minute 23 ==
1 ore-collecting robot collects 1 ore; you now have 5 ore.
4 clay-collecting robots collect 4 clay; you now have 37 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 6 obsidian.
2 geode-cracking robots crack 2 geodes; you now have 7 open geodes.

== Minute 24 ==
1 ore-collecting robot collects 1 ore; you now have 6 ore.
4 clay-collecting robots collect 4 clay; you now have 41 clay.
2 obsidian-collecting robots collect 2 obsidian; you now have 8 obsidian.
2 geode-cracking robots crack 2 geodes; you now have 9 open geodes.
```

However, by using blueprint 2 in the example above, you could do even better: the largest number of geodes you could open in 24 minutes is 12.

Determine the *quality level* of each blueprint by *multiplying that blueprint's ID number* with the largest number of geodes that can be opened in 24 minutes using that blueprint. In this example, the first blueprint has ID 1 and can open 9 geodes, so its quality level is 9. The second blueprint has ID 2 and can open 12 geodes, so its quality level is 24. Finally, if you *add up the quality levels* of all

of the blueprints in the list, you get `33`.

Determine the quality level of each blueprint using the largest number of geodes it could produce in 24 minutes. *What do you get if you add up the quality level of all of the blueprints in your list?*

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

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

--- Day 20: Grove Positioning System ---

It's finally time to meet back up with the Elves. When you try to contact them, however, you get no reply. Perhaps you're out of range?

You know they're headed to the grove where the *star* fruit grows, so if you can figure out where that is, you should be able to meet back up with them.

Fortunately, your handheld device has a file (your puzzle input) that contains the grove's coordinates! Unfortunately, the file is *encrypted* - just in case the device were to fall into the wrong hands.

Maybe you can decrypt it?

When you were still back at the camp, you overheard some Elves talking about coordinate file encryption. The main operation involved in decrypting the file is called *mixing*.

The encrypted file is a list of numbers. To *mix* the file, move each number forward or backward in the file a number of positions equal to the value of the number being moved. The list is *circular*, so moving a number off one end of the list wraps back around to the other end as if the ends were connected.

For example, to move the `1` in a sequence like `4, 5, 6, 1, 7, 8, 9`, the `1` moves one position forward: `4, 5, 6, 7, 1, 8, 9`. To move the `-2` in a sequence like `4, -2, 5, 6, 7, 8, 9`, the `-2` moves two positions backward, wrapping around: `4, 5, 6, 7, 8, -2, 9`.

The numbers should be moved *in the order they originally appear* in the encrypted file. Numbers moving around during the mixing process do not change the order in which the numbers are moved.

Consider this encrypted file:

```
1
2
-3
3
-2
0
4
```

Mixing this file proceeds as follows:

```
Initial arrangement:
1, 2, -3, 3, -2, 0, 4

1 moves between 2 and -3:
2, 1, -3, 3, -2, 0, 4

2 moves between -3 and 3:
1, -3, 2, 3, -2, 0, 4

-3 moves between -2 and 0:
1, 2, 3, -2, -3, 0, 4

3 moves between 0 and 4:
1, 2, -2, -3, 0, 3, 4

-2 moves between 4 and 1:
1, 2, -3, 0, 3, 4, -2

0 does not move:
1, 2, -3, 0, 3, 4, -2

4 moves between -3 and 0:
1, 2, -3, 4, 0, 3, -2
```

Then, the grove coordinates can be found by looking at the 1000th, 2000th, and 3000th numbers after the value `0`, wrapping around the list as necessary. In the above example, the 1000th number after `0` is `4`, the 2000th is `-3`, and the 3000th is `2`; adding these together produces `3`.

Mix your encrypted file exactly once. *What is the sum of the three numbers that form the grove coordinates?*

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

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

--- Day 21: Monkey Math ---

The monkeys are back! You're worried they're going to try to steal your stuff again, but it seems like they're just holding their ground and making various monkey noises at you.

Eventually, one of the elephants realizes you don't speak monkey and comes over to interpret. As it turns out, they overheard you talking about trying to find the grove; they can show you a shortcut if you answer their *riddle*.

Each monkey is given a *job*: either to *yell a specific number* or to *yell the result*

of a math operation. All of the number-yelling monkeys know their number from the start; however, the math operation monkeys need to wait for two other monkeys to yell a number, and those two other monkeys might *also* be waiting on other monkeys.

Your job is to *work out the number the monkey named* `root` *will yell* before the monkeys figure it out themselves.

For example:

```
root: pppw + sjmn
dbpl: 5
cczh: sllz + lgvd
zczc: 2
ptdq: humn - dvpt
dvpt: 3
lfqf: 4
humn: 5
ljgn: 2
sjmn: drzm * dbpl
sllz: 4
pppw: cczh / lfqf
lgvd: ljgn * ptdq
drzm: hmdt - zczc
hmdt: 32
```

Each line contains the name of a monkey, a colon, and then the job of that monkey:

- A lone number means the monkey's job is simply to yell that number.
- A job like `aaaa + bbbb` means the monkey waits for monkeys `aaaa` and `bbbb` to yell each of their numbers; the monkey then yells the sum of those two numbers.
- `aaaa - bbbb` means the monkey yells `aaaa`'s number minus `bbbb`'s number.
- Job `aaaa * bbbb` will yell `aaaa`'s number multiplied by `bbbb`'s number.
- Job `aaaa / bbbb` will yell `aaaa`'s number divided by `bbbb`'s number.

So, in the above example, monkey `drzm` has to wait for monkeys `hmdt` and `zczc` to yell their numbers. Fortunately, both `hmdt` and `zczc` have jobs that involve simply yelling a single number, so they do this immediately: `32` and `2`. Monkey `drzm` can then yell its number by finding `32` minus `2`: `30`.

Then, monkey `sjmn` has one of its numbers (`30`, from monkey `drzm`), and already has its other number, `5`, from `dbpl`. This allows it to yell its own number by finding `30` multiplied by `5`: `150`.

This process continues until `root` yells a number: `152`.

However, your actual situation involves considerably more monkeys. *What number will the monkey named* `root` *yell?*

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

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

--- Day 22: Monkey Map ---

The monkeys take you on a surprisingly easy trail through the jungle. They're even going in roughly the right direction according to your handheld device's Grove Positioning System.

As you walk, the monkeys explain that the grove is protected by a *force field*. To pass through the force field, you have to enter a password; doing so involves tracing a specific *path* on a strangely-shaped board.

At least, you're pretty sure that's what you have to do; the elephants aren't exactly fluent in monkey.

The monkeys give you notes that they took when they last saw the password entered (your puzzle input).

For example:

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

10R5L5R10L4R5L5
```

The first half of the monkeys' notes is a *map of the board*. It is comprised of a set of *open tiles* (on which you can move, drawn `.`) and *solid walls* (tiles which you cannot enter, drawn `#`).

The second half is a description of *the path you must follow*. It consists of alternating numbers and letters:

- A *number* indicates the *number of tiles to move* in the direction you are facing. If you run into a wall, you stop moving forward and continue with the next instruction.
- A *letter* indicates whether to turn 90 degrees *clockwise* (`R`) or *counter-clockwise* (`L`). Turning happens in-place; it does not change your current tile.

So, a path like `10R5` means "go forward 10 tiles, then turn clockwise 90 degrees, then go forward 5 tiles".

You begin the path in the leftmost open tile of the top row of tiles. Initially, you are facing *to the right* (from the perspective of how the map is drawn).

If a movement instruction would take you off of the map, you *wrap around* to the other side of the board. In other words, if your next tile is off of the board, you should instead look in the direction opposite of your current facing as far as you can until you find the opposite edge of the board, then reappear there.

For example, if you are at `A` and facing to the right, the tile in front of you is marked `B`; if you are at `C` and facing down, the tile in front of you is marked `D`:

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

It is possible for the next tile (after wrapping around) to be a *wall*; this still counts as there being a wall in front of you, and so movement stops before you actually wrap to the other side of the board.

By drawing the *last facing you had* with an arrow on each tile you visit, the full path taken by the above example looks like this:

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

To finish providing the password to this strange input device, you need to determine numbers for your final *row*, *column*, and *facing* as your final position appears from the perspective of the original map. Rows start from `1` at the top

and count downward; columns start from `1` at the left and count rightward. (In the above example, row 1, column 1 refers to the empty space with no tile on it in the top-left corner.) Facing is `0` for right (`>`), `1` for down (`v`), `2` for left (`<`), and `3` for up (`^`). The *final password* is the sum of 1000 times the row, 4 times the column, and the facing.

In the above example, the final row is `6`, the final column is `8`, and the final facing is `0`. So, the final password is 1000 * 6 + 4 * 8 + 0: `6032`.

Follow the path given in the monkeys' notes. *What is the final password?*

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

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

--- Day 23: Unstable Diffusion ---

You enter a large crater of gray dirt where the grove is supposed to be. All around you, plants you imagine were expected to be full of fruit are instead withered and broken. A large group of Elves has formed in the middle of the grove.

"...but this volcano has been dormant for months. Without ash, the fruit can't grow!"

You look up to see a massive, snow-capped mountain towering above you.

"It's not like there are other active volcanoes here; we've looked everywhere."

"But our scanners show active magma flows; clearly it's going *somewhere*."

They finally notice you at the edge of the grove, your pack almost overflowing from the random *star* fruit you've been collecting. Behind you, elephants and monkeys explore the grove, looking concerned. Then, the Elves recognize the ash cloud slowly spreading above your recent detour.

"Why do you--" "How is--" "Did you just--"

Before any of them can form a complete question, another Elf speaks up: "Okay, new plan. We have almost enough fruit already, and ash from the plume should spread here eventually. If we quickly plant new seedlings now, we can still make it to the extraction point. Spread out!"

The Elves each reach into their pack and pull out a tiny plant. The plants rely on important nutrients from the ash, so they can't be planted too close together.

There isn't enough time to let the Elves figure out where to plant the seedlings themselves; you quickly scan the grove (your puzzle input) and note their positions.

For example:

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

The scan shows Elves `#` and empty ground `.`; outside your scan, more empty ground extends a long way in every direction. The scan is oriented so that *north is up*; orthogonal directions are written N (north), S (south), W (west), and E (east), while diagonal directions are written NE, NW, SE, SW.

The Elves follow a time-consuming process to figure out where they should each go; you can speed up this process considerably. The process consists of some number of *rounds* during which Elves alternate between considering where to move and actually moving.

During the *first half* of each round, each Elf considers the eight positions adjacent to themself. If no other Elves are in one of those eight positions, the Elf *does not do anything* during this round. Otherwise, the Elf looks in each of four directions in the following order and *proposes* moving one step in the *first valid direction*:

- If there is no Elf in the N, NE, or NW adjacent positions, the Elf proposes moving *north* one step.
- If there is no Elf in the S, SE, or SW adjacent positions, the Elf proposes moving *south* one step.
- If there is no Elf in the W, NW, or SW adjacent positions, the Elf proposes moving *west* one step.
- If there is no Elf in the E, NE, or SE adjacent positions, the Elf proposes moving *east* one step.

After each Elf has had a chance to propose a move, the *second half* of the round can begin. Simultaneously, each Elf moves to their proposed destination tile if they were the *only* Elf to propose moving to that position. If two or more Elves propose moving to the same position, *none* of those Elves move.

Finally, at the end of the round, the *first direction* the Elves considered is moved to the end of the list of directions. For example, during the second round, the Elves would try proposing a move to the south first, then west, then east, then north. On the third round, the Elves would first consider west, then east, then north, then south.

As a smaller example, consider just these five Elves:

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

```
.....
```

The northernmost two Elves and southernmost two Elves all propose moving north, while the middle Elf cannot move north and proposes moving south. The middle Elf proposes the same destination as the southwest Elf, so neither of them move, but the other three do:

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

Next, the northernmost two Elves and the southernmost Elf all propose moving south. Of the remaining middle two Elves, the west one cannot move south and proposes moving west, while the east one cannot move south *or* west and proposes moving east. All five Elves succeed in moving to their proposed positions:

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

Finally, the southernmost two Elves choose not to move at all. Of the remaining three Elves, the west one proposes moving west, the east one proposes moving east, and the middle one proposes moving north; all three succeed in moving:

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

At this point, no Elves need to move, and so the process ends.

The larger example above proceeds as follows:

```
== Initial State ==
..............
..............
.......#......
.....###.#....
...#...#.#....
....#...##....
...#.###......
...##.#.##....
```

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

== End of Round 1 ==
..............
.......#......
.....#...#....
...#..#.#.....
.......#..#...
....#.#.##....
..#..#.#......
..#.#.#.##....
..............
....#..#......
..............
..............

== End of Round 2 ==
..............
.......#......
....#.....#...
...#..#.#.....
.......#...#..
...#..#.#.....
.#...#.#.#....
..............
..#.#.#.##....
....#..#......
..............
..............

== End of Round 3 ==
..............
.......#......
.....#....#...
..#..#...#....
.......#...#..
...#..#.#.....
.#..#.....#...
.......##.....
..##.#....#...
...#..........
.......#......
..............
```

```
== End of Round 4 ==
..............
.......#......
......#....#..
..#...##......
...#.....#.#..
.........#....
.#...###..#...
..#......#....
....##....#...
....#.........
.......#......
..............

== End of Round 5 ==
.......#......
..............
..#..#.....#..
.........#....
......##...#..
.#.#.####.....
...........#..
....##..#.....
..#...........
..........#...
....#..#......
..............
```

After a few more rounds...

```
== End of Round 10 ==
.......#......
...........#..
..#.#..#......
......#.......
...#.....#..#.
.#......##....
.....##.......
..#........#..
....#.#..#....
..............
....#..#..#...
..............
```

To make sure they're on the right track, the Elves like to check after round 10 that they're making good progress toward covering enough ground. To do this,

count the number of empty ground tiles contained by the smallest rectangle that contains every Elf. (The edges of the rectangle should be aligned to the N/S/E/W directions; the Elves do not have the patience to calculate arbitrary rectangles.) In the above example, that rectangle is:

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

In this region, the number of empty ground tiles is `110`.

Simulate the Elves' process and find the smallest rectangle that contains the Elves after 10 rounds. *How many empty ground tiles does that rectangle contain?*

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

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

--- Day 24: Blizzard Basin ---

With everything replanted for next year (and with elephants and monkeys to tend the grove), you and the Elves leave for the extraction point.

Partway up the mountain that shields the grove is a flat, open area that serves as the extraction point. It's a bit of a climb, but nothing the expedition can't handle.

At least, that would normally be true; now that the mountain is covered in snow, things have become more difficult than the Elves are used to.

As the expedition reaches a valley that must be traversed to reach the extraction site, you find that strong, turbulent winds are pushing small *blizzards* of snow and sharp ice around the valley. It's a good thing everyone packed warm clothes! To make it across safely, you'll need to find a way to avoid them.

Fortunately, it's easy to see all of this from the entrance to the valley, so you make a map of the valley and the blizzards (your puzzle input). For example:

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

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

The walls of the valley are drawn as `#`; everything else is ground. Clear ground - where there is currently no blizzard - is drawn as `.`. Otherwise, blizzards are drawn with an arrow indicating their direction of motion: up (`^`), down (`v`), left (`<`), or right (`>`).

The above map includes two blizzards, one moving right (`>`) and one moving down (`v`). In one minute, each blizzard moves one position in the direction it is pointing:

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

Due to conservation of blizzard energy, as a blizzard reaches the wall of the valley, a new blizzard forms on the opposite side of the valley moving in the same direction. After another minute, the bottom downward-moving blizzard has been replaced with a new downward-moving blizzard at the top of the valley instead:

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

Because blizzards are made of tiny snowflakes, they pass right through each other. After another minute, both blizzards temporarily occupy the same position, marked `2`:

```
#.#####
#.....#
#...2.#
#.....#
#.....#
#.....#
#####.#
```

After another minute, the situation resolves itself, giving each blizzard back its personal space:

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

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

Finally, after yet another minute, the rightward-facing blizzard on the right is replaced with a new one on the left facing the same direction:

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

This process repeats at least as long as you are observing it, but probably forever.

Here is a more complex example:

```
#.######
#>>.<^<#
#.<..<<#
#>v.><>#
#<^v^^>#
######.#
```

Your expedition begins in the only non-wall position in the top row and needs to reach the only non-wall position in the bottom row. On each minute, you can *move* up, down, left, or right, or you can *wait* in place. You and the blizzards act *simultaneously*, and you cannot share a position with a blizzard.

In the above example, the fastest way to reach your goal requires `18` steps. Drawing the position of the expedition as `E`, one way to achieve this is:

```
Initial state:
#E######
#>>.<^<#
#.<..<<#
#>v.><>#
#<^v^^>#
######.#

Minute 1, move down:
#.######
#E>3.<.#
#<..<<.#
#>2.22.#
```

```
#>v..^<#
######.#

Minute 2, move down:
#.######
#.2>2..#
#E^22^<#
#.>2.^>#
#.>..<.#
######.#

Minute 3, wait:
#.######
#<^<22.#
#E2<.2.#
#><2>..#
#..><..#
######.#

Minute 4, move up:
#.######
#E<..22#
#<<.<..#
#<2.>>.#
#.^22^.#
######.#

Minute 5, move right:
#.######
#2Ev.<>#
#<.<..<#
#.^>^22#
#.2..2.#
######.#

Minute 6, move right:
#.######
#>2E<.<#
#.2v^2<#
#>..>2>#
#<....>#
######.#

Minute 7, move down:
#.######
#.22^2.#
```

```
#<vE<2.#
#>>v<>.#
#>....<#
######.#

Minute 8, move left:
#.######
#.<>2^.#
#.E<<.<#
#.22..>#
#.2v^2.#
######.#

Minute 9, move up:
#.######
#<E2>>.#
#.<<.<.#
#>2>2^.#
#.v><^.#
######.#

Minute 10, move right:
#.######
#.2E.>2#
#<2v2^.#
#<>.>2.#
#..<>..#
######.#

Minute 11, wait:
#.######
#2^E^2>#
#<v<.^<#
#..2.>2#
#.<..>.#
######.#

Minute 12, move down:
#.######
#>>.<^<#
#.<E.<<#
#>v.><>#
#<^v^^>#
######.#

Minute 13, move down:
```

```
#.######
#.>3.<.#
#<..<<.#
#>2E22.#
#>v..^<#
######.#

Minute 14, move right:
#.######
#.2>2..#
#.^22^<#
#.>2E^>#
#.>..<.#
######.#

Minute 15, move right:
#.######
#<^<22.#
#.2<.2.#
#><2>E.#
#..><..#
######.#

Minute 16, move right:
#.######
#.<..22#
#<<.<..#
#<2.>>E#
#.^22^.#
######.#

Minute 17, move down:
#.######
#2.v.<>#
#<.<..<#
#.^>^22#
#.2..2E#
######.#

Minute 18, move down:
#.######
#>2.<.<#
#.2v^2<#
#>..>2>#
#<....>#
######E#
```

What is the fewest number of minutes required to avoid the blizzards and reach the goal?

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

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

--- Day 25: Full of Hot Air ---

As the expedition finally reaches the extraction point, several large hot air balloons drift down to meet you. Crews quickly start unloading the equipment the balloons brought: many hot air balloon kits, some fuel tanks, and a *fuel heating machine*.

The fuel heating machine is a new addition to the process. When this mountain was a volcano, the ambient temperature was more reasonable; now, it's so cold that the fuel won't work at all without being warmed up first.

The Elves, seemingly in an attempt to make the new machine feel welcome, have already attached a pair of googly eyes and started calling it "Bob".

To heat the fuel, Bob needs to know the total amount of fuel that will be processed ahead of time so it can correctly calibrate heat output and flow rate. This amount is simply the *sum* of the fuel requirements of all of the hot air balloons, and those fuel requirements are even listed clearly on the side of each hot air balloon's burner.

You assume the Elves will have no trouble adding up some numbers and are about to go back to figuring out which balloon is yours when you get a tap on the shoulder. Apparently, the fuel requirements use numbers written in a format the Elves don't recognize; predictably, they'd like your help deciphering them.

You make a list of all of the fuel requirements (your puzzle input), but you don't recognize the number format either. For example:

```
1=-0-2
12111
2=0=
21
2=01
111
20012
112
1=-1=
1-12
12
1=
122
```

Fortunately, Bob is labeled with a support phone number. Not to be deterred, you call and ask for help.

"That's right, just supply the fuel amount to the-- oh, for more than one burner? No problem, you just need to add together our Special Numeral-Analogue Fuel Units. Patent pending! They're way better than normal numbers for--"

You mention that it's quite cold up here and ask if they can skip ahead.

"Okay, our Special Numeral-Analogue Fuel Units - SNAFU for short - are sort of like normal numbers. You know how starting on the right, normal numbers have a ones place, a tens place, a hundreds place, and so on, where the digit in each place tells you how many of that value you have?"

"SNAFU works the same way, except it uses powers of five instead of ten. Starting from the right, you have a ones place, a fives place, a twenty-fives place, a one-hundred-and-twenty-fives place, and so on. It's that easy!"

You ask why some of the digits look like `-` or `=` instead of "digits".

"You know, I never did ask the engineers why they did that. Instead of using digits four through zero, the digits are `2`, `1`, `0`, *minus* (written `-`), and *double-minus* (written `=`). Minus is worth -1, and double-minus is worth -2."

"So, because ten (in normal numbers) is two fives and no ones, in SNAFU it is written `20`. Since eight (in normal numbers) is two fives minus two ones, it is written `2=`."

"You can do it the other direction, too. Say you have the SNAFU number `2=-01`. That's `2` in the 625s place, `=` (double-minus) in the 125s place, `-` (minus) in the 25s place, `0` in the 5s place, and `1` in the 1s place. (2 times 625) plus (-2 times 125) plus (-1 times 25) plus (0 times 5) plus (1 times 1). That's 1250 plus -250 plus -25 plus 0 plus 1. *976*!"

"I see here that you're connected via our premium uplink service, so I'll transmit our handy SNAFU brochure to you now. Did you need anything else?"

You ask if the fuel will even work in these temperatures.

"Wait, it's *how* cold? There's no *way* the fuel - or *any* fuel - would work in those conditions! There are only a few places in the-- where did you say you are again?"

Just then, you notice one of the Elves pour a few drops from a snowflake-shaped container into one of the fuel tanks, thank the support representative for their time, and disconnect the call.

The SNAFU brochure contains a few more examples of decimal ("normal") numbers and their SNAFU counterparts:

Decimal	SNAFU
1	1
2	2

```
         3             1=
         4             1-
         5             10
         6             11
         7             12
         8             2=
         9             2-
        10             20
        15            1=0
        20            1-0
      2022         1=11-2
     12345        1-0---0
314159265  1121-1110-1=0
```

Based on this process, the SNAFU numbers in the example above can be converted to decimal numbers as follows:

```
 SNAFU  Decimal
1=-0-2     1747
 12111      906
  2=0=      198
    21       11
  2=01      201
   111       31
 20012     1257
   112       32
 1=-1=      353
  1-12      107
    12        7
    1=        3
   122       37
```

In decimal, the sum of these numbers is `4890`.

As you go to input this number on Bob's console, you discover that some buttons you expected are missing. Instead, you are met with buttons labeled `=`, `-`, `0`, `1`, and `2`. Bob needs the input value expressed as a SNAFU number, not in decimal.

Reversing the process, you can determine that for the decimal number `4890`, the SNAFU number you need to supply to Bob's console is `2=-1=0`.

The Elves are starting to get cold. *What SNAFU number do you supply to Bob's console?*

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

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