--- Day 1: Inverse Captcha ---

The night before Christmas, one of Santa's Elves calls you in a panic. "The printer's broken! We can't print the *Naughty or Nice List*!" By the time you make it to sub-basement 17, there are only a few minutes until midnight. "We have a big problem," she says; "there must be almost *fifty* bugs in this system, but nothing else can print The List. Stand in this square, quick! There's no time to explain; if you can convince them to pay you in *stars*, you'll be able to--" She pulls a lever and the world goes blurry.

When your eyes can focus again, everything seems a lot more pixelated than before. She must have sent you inside the computer! You check the system clock: *25 milliseconds* until midnight. With that much time, you should be able to collect all *fifty stars* by December 25th.

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

You're standing in a room with "digitization quarantine" written in LEDs along one wall. The only door is locked, but it includes a small interface. "Restricted Area - Strictly No Digitized Users Allowed."

It goes on to explain that you may only leave by solving a captcha to prove you're *not* a human. Apparently, you only get one millisecond to solve the captcha: too fast for a normal human, but it feels like hours to you.

The captcha requires you to review a sequence of digits (your puzzle input) and find the *sum* of all digits that match the *next* digit in the list. The list is circular, so the digit after the last digit is the *first* digit in the list.

For example:

- `1122` produces a sum of `3` (`1` + `2`) because the first digit (`1`) matches the second digit and the third digit (`2`) matches the fourth digit.
- `1111` produces `4` because each digit (all `1`) matches the next.
- `1234` produces `0` because no digit matches the next.
- `91212129` produces `9` because the only digit that matches the next one is the last digit, `9`.

What is the solution to your captcha?

Your puzzle answer was `1158`.

--- Part Two ---

You notice a progress bar that jumps to 50% completion. Apparently, the door isn't yet satisfied, but it did emit a *star* as encouragement. The instructions change:

Now, instead of considering the *next* digit, it wants you to consider the digit *halfway around* the circular list. That is, if your list contains `10` items, only include a digit in your sum if the digit `10/2 = 5` steps forward matches it. Fortunately, your list has an even number of elements.

For example:

- `1212` produces `6`: the list contains `4` items, and all four digits match the digit `2` items ahead.
- `1221` produces `0`, because every comparison is between a `1` and a `2`.
- `123425` produces `4`, because both `2`s match each other, but no other digit has a match.
- `123123` produces `12`.
- `12131415` produces `4`.

What is the solution to your new captcha?

Your puzzle answer was `1132`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 2: Corruption Checksum ---

As you walk through the door, a glowing humanoid shape yells in your direction. "You there! Your state appears to be idle. Come help us repair the corruption in this spreadsheet - if we take another millisecond, we'll have to display an hourglass cursor!"

The spreadsheet consists of rows of apparently-random numbers. To make sure the recovery process is on the right track, they need you to calculate the spreadsheet's *checksum*. For each row, determine the difference between the largest value and the smallest value; the checksum is the sum of all of these differences.

For example, given the following spreadsheet:

```
5 1 9 5
7 5 3
2 4 6 8
```

- The first row's largest and smallest values are `9` and `1`, and their difference is `8`.
- The second row's largest and smallest values are `7` and `3`, and their difference is `4`.
- The third row's difference is `6`.

In this example, the spreadsheet's checksum would be `8 + 4 + 6 = 18`.

What is the checksum for the spreadsheet in your puzzle input?

Your puzzle answer was `30994`.

--- Part Two ---

"Great work; looks like we're on the right track after all. Here's a *star* for your effort." However, the program seems a little worried. Can programs *be* worried?

"Based on what we're seeing, it looks like all the User wanted is some information about the *evenly divisible values* in the spreadsheet. Unfortunately, none of us are equipped for that kind of calculation - most of us specialize in bitwise operations."

It sounds like the goal is to find the only two numbers in each row where one evenly divides the other - that is, where the result of the division operation is a whole number. They would like you to find those numbers on each line, divide them, and add up each line's result.

For example, given the following spreadsheet:

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

- In the first row, the only two numbers that evenly divide are `8` and `2`; the result of this division is `4`.
- In the second row, the two numbers are `9` and `3`; the result is `3`.
- In the third row, the result is `2`.

In this example, the sum of the results would be `4 + 3 + 2 = 9`.

What is the *sum of each row's result* in your puzzle input?

Your puzzle answer was `233`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 3: Spiral Memory ---

You come across an experimental new kind of memory stored on an infinite two-dimensional grid.

Each square on the grid is allocated in a spiral pattern starting at a location marked `1` and then counting up while spiraling outward. For example, the first few squares are allocated like this:

```
17  16  15  14  13
18   5   4   3  12
19   6   1   2  11
20   7   8   9  10
21  22  23---> ...
```

While this is very space-efficient (no squares are skipped), requested data must be carried back to square `1` (the location of the only access port for this memory system) by programs that can only move up, down, left, or right. They always take the shortest path: the Manhattan Distance between the location of the data and square `1`.

For example:

- Data from square `1` is carried `0` steps, since it's at the access port.
- Data from square `12` is carried `3` steps, such as: down, left, left.
- Data from square `23` is carried only `2` steps: up twice.
- Data from square `1024` must be carried `31` steps.

How many steps are required to carry the data from the square identified in your puzzle input all the way to the access port?

Your puzzle answer was `371`.

--- Part Two ---

As a stress test on the system, the programs here clear the grid and then store the value `1` in square `1`. Then, in the same allocation order as shown above, they store the sum of the values in all adjacent squares, including diagonals.

So, the first few squares' values are chosen as follows:

- Square `1` starts with the value `1`.
- Square `2` has only one adjacent filled square (with value `1`), so it also stores `1`.
- Square `3` has both of the above squares as neighbors and stores the sum of their values, `2`.
- Square `4` has all three of the aforementioned squares as neighbors and stores the sum of their values, `4`.
- Square `5` only has the first and fourth squares as neighbors, so it gets the value `5`.

Once a square is written, its value does not change. Therefore, the first few squares would receive the following values:

```
147  142  133  122   59
304    5    4    2   57
```

```
330   10    1    1   54
351   11   23   25   26
362  747  806--->   ...
```

What is the *first value written* that is *larger* than your puzzle input?

Your puzzle answer was `369601`.

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

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

Your puzzle input was `368078`.

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 4: High-Entropy Passphrases ---

A new system policy has been put in place that requires all accounts to use a *passphrase* instead of simply a pass*word*. A passphrase consists of a series of words (lowercase letters) separated by spaces.

To ensure security, a valid passphrase must contain no duplicate words.

For example:

- `aa bb cc dd ee` is valid.
- `aa bb cc dd aa` is not valid - the word `aa` appears more than once.
- `aa bb cc dd aaa` is valid - `aa` and `aaa` count as different words.

The system's full passphrase list is available as your puzzle input. *How many passphrases are valid?*

Your puzzle answer was `455`.

--- Part Two ---

For added security, yet another system policy has been put in place. Now, a valid passphrase must contain no two words that are anagrams of each other - that is, a passphrase is invalid if any word's letters can be rearranged to form any other word in the passphrase.

For example:

- `abcde fghij` is a valid passphrase.
- `abcde xyz ecdab` is not valid - the letters from the third word can be rearranged to form the first word.
- `a ab abc abd abf abj` is a valid passphrase, because *all* letters need to be used when forming another word.
- `iiii oiii ooii oooi oooo` is valid.
- `oiii ioii iioi iiio` is not valid - any of these words can be rearranged to form any other word.

Under this new system policy, *how many passphrases are valid?*

Your puzzle answer was `186`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 5: A Maze of Twisty Trampolines, All Alike ---

An urgent interrupt arrives from the CPU: it's trapped in a maze of jump instructions, and it would like assistance from any programs with spare cycles to help find the exit.

The message includes a list of the offsets for each jump. Jumps are relative: `-1` moves to the previous instruction, and `2` skips the next one. Start at the first instruction in the list. The goal is to follow the jumps until one leads *outside* the list.

In addition, these instructions are a little strange; after each jump, the offset of that instruction increases by `1`. So, if you come across an offset of `3`, you would move three instructions forward, but change it to a `4` for the next time it is encountered.

For example, consider the following list of jump offsets:

```
0
3
0
1
-3
```

Positive jumps ("forward") move downward; negative jumps move upward. For legibility in this example, these offset values will be written all on one line, with the current instruction marked in parentheses. The following steps would be taken before an exit is found:

- `(0) 3  0  1  -3 ` - *before* we have taken any steps.
- `(1) 3  0  1  -3 ` - jump with offset `0` (that is, don't jump at all). Fortunately, the instruction is then incremented to `1`.
- ` 2 (3) 0  1  -3 ` - step forward because of the instruction we just modified. The first instruction is incremented again, now to `2`.
- ` 2  4  0  1 (-3)` - jump all the way to the end; leave a `4` behind.
- ` 2 (4) 0  1  -2 ` - go back to where we just were; increment `-3` to `-2`.
- ` 2  5  0  1  -2 ` - jump `4` steps forward, escaping the maze.

In this example, the exit is reached in `5` steps.

How many steps does it take to reach the exit?

Your puzzle answer was `372671`.

--- Part Two ---

Now, the jumps are even stranger: after each jump, if the offset was *three or more*, instead *decrease* it by `1`. Otherwise, increase it by `1` as before.

Using this rule with the above example, the process now takes `10` steps, and the offset values after finding the exit are left as `2 3 2 3 -1`.

How many steps does it now take to reach the exit?

Your puzzle answer was `25608480`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 6: Memory Reallocation ---

A debugger program here is having an issue: it is trying to repair a memory reallocation routine, but it keeps getting stuck in an infinite loop.

In this area, there are sixteen memory banks; each memory bank can hold any number of *blocks*. The goal of the reallocation routine is to balance the blocks between the memory banks.

The reallocation routine operates in cycles. In each cycle, it finds the memory bank with the most blocks (ties won by the lowest-numbered memory bank) and redistributes those blocks among the banks. To do this, it removes all of the blocks from the selected bank, then moves to the next (by index) memory bank and inserts one of the blocks. It continues doing this until it runs out of blocks; if it reaches the last memory bank, it wraps around to the first one.

The debugger would like to know how many redistributions can be done before a blocks-in-banks configuration is produced that *has been seen before*.

For example, imagine a scenario with only four memory banks:

- The banks start with `0`, `2`, `7`, and `0` blocks. The third bank has the most blocks, so it is chosen for redistribution.
- Starting with the next bank (the fourth bank) and then continuing to the first bank, the second bank, and so on, the `7` blocks are spread out over the memory banks. The fourth, first, and second banks get two blocks each, and the third bank gets one back. The final result looks like this: `2 4 1 2`.

- Next, the second bank is chosen because it contains the most blocks (four). Because there are four memory banks, each gets one block. The result is: `3 1 2 3`.
- Now, there is a tie between the first and fourth memory banks, both of which have three blocks. The first bank wins the tie, and its three blocks are distributed evenly over the other three banks, leaving it with none: `0 2 3 4`.
- The fourth bank is chosen, and its four blocks are distributed such that each of the four banks receives one: `1 3 4 1`.
- The third bank is chosen, and the same thing happens: `2 4 1 2`.

At this point, we've reached a state we've seen before: `2 4 1 2` was already seen. The infinite loop is detected after the fifth block redistribution cycle, and so the answer in this example is `5`.

Given the initial block counts in your puzzle input, *how many redistribution cycles* must be completed before a configuration is produced that has been seen before?

Your puzzle answer was `5042`.

--- Part Two ---

Out of curiosity, the debugger would also like to know the size of the loop: starting from a state that has already been seen, how many block redistribution cycles must be performed before that same state is seen again?

In the example above, `2 4 1 2` is seen again after four cycles, and so the answer in that example would be `4`.

How many cycles are in the infinite loop that arises from the configuration in your puzzle input?

Your puzzle answer was `1086`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 7: Recursive Circus ---

Wandering further through the circuits of the computer, you come upon a tower of programs that have gotten themselves into a bit of trouble. A recursive algorithm has gotten out of hand, and now they're balanced precariously in a large tower.

One program at the bottom supports the entire tower. It's holding a large disc, and on the disc are balanced several more sub-towers. At the bottom of these sub-towers, standing on the bottom disc, are other programs, each holding *their* own disc, and so on. At the very tops of these sub-sub-sub-...-towers, many programs stand simply keeping the disc below them balanced but with no disc of their own.

You offer to help, but first you need to understand the structure of these towers. You ask each program to yell out their *name*, their *weight*, and (if they're holding a disc) the *names of the programs immediately above them* balancing on that disc. You write this information down (your puzzle input). Unfortunately, in their panic, they don't do this in an orderly fashion; by the time you're done, you're not sure which program gave which information.

For example, if your list is the following:

```
pbga (66)
xhth (57)
ebii (61)
havc (66)
ktlj (57)
fwft (72) -> ktlj, cntj, xhth
qoyq (66)
padx (45) -> pbga, havc, qoyq
tknk (41) -> ugml, padx, fwft
jptl (61)
ugml (68) -> gyxo, ebii, jptl
gyxo (61)
cntj (57)
```

...then you would be able to recreate the structure of the towers that looks like this:

```
                gyxo
              /
         ugml - ebii
       /      \
      |         jptl
      |
      |         pbga
     /        /
tknk --- padx - havc
     \        \
      |         qoyq
      |
      |         ktlj
       \      /
         fwft - cntj
```

```
                \
                 xhth
```

In this example, `tknk` is at the bottom of the tower (the *bottom program*), and is holding up `ugml`, `padx`, and `fwft`. Those programs are, in turn, holding up other programs; in this example, none of those programs are holding up any other programs, and are all the tops of their own towers. (The actual tower balancing in front of you is much larger.)

Before you're ready to help them, you need to make sure your information is correct. *What is the name of the bottom program?*

Your puzzle answer was `qibuqqg`.

--- Part Two ---

The programs explain the situation: they can't get down. Rather, they *could* get down, if they weren't expending all of their energy trying to keep the tower balanced. Apparently, one program has the *wrong weight*, and until it's fixed, they're stuck here.

For any program holding a disc, each program standing on that disc forms a sub-tower. Each of those sub-towers are supposed to be the same weight, or the disc itself isn't balanced. The weight of a tower is the sum of the weights of the programs in that tower.

In the example above, this means that for `ugml`'s disc to be balanced, `gyxo`, `ebii`, and `jptl` must all have the same weight, and they do: `61`.

However, for `tknk` to be balanced, each of the programs standing on its disc *and all programs above it* must each match. This means that the following sums must all be the same:

- `ugml` + (`gyxo` + `ebii` + `jptl`) = 68 + (61 + 61 + 61) = 251
- `padx` + (`pbga` + `havc` + `qoyq`) = 45 + (66 + 66 + 66) = 243
- `fwft` + (`ktlj` + `cntj` + `xhth`) = 72 + (57 + 57 + 57) = 243

As you can see, `tknk`'s disc is unbalanced: `ugml`'s stack is heavier than the other two. Even though the nodes above `ugml` are balanced, `ugml` itself is too heavy: it needs to be `8` units lighter for its stack to weigh `243` and keep the towers balanced. If this change were made, its weight would be `60`.

Given that exactly one program is the wrong weight, *what would its weight need to be* to balance the entire tower?

Your puzzle answer was `1079`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 8: I Heard You Like Registers ---

You receive a signal directly from the CPU. Because of your recent assistance with jump instructions, it would like you to compute the result of a series of unusual register instructions.

Each instruction consists of several parts: the register to modify, whether to increase or decrease that register's value, the amount by which to increase or decrease it, and a condition. If the condition fails, skip the instruction without modifying the register. The registers all start at `0`. The instructions look like this:

```
b inc 5 if a > 1
a inc 1 if b < 5
c dec -10 if a >= 1
c inc -20 if c == 10
```

These instructions would be processed as follows:

- Because `a` starts at `0`, it is not greater than `1`, and so `b` is not modified.
- `a` is increased by `1` (to `1`) because `b` is less than `5` (it is `0`).
- `c` is decreased by `-10` (to `10`) because `a` is now greater than or equal to `1` (it is `1`).
- `c` is increased by `-20` (to `-10`) because `c` is equal to `10`.

After this process, the largest value in any register is `1`.

You might also encounter `<=` (less than or equal to) or `!=` (not equal to). However, the CPU doesn't have the bandwidth to tell you what all the registers are named, and leaves that to you to determine.

What is the largest value in any register after completing the instructions in your puzzle input?

Your puzzle answer was `5946`.

--- Part Two ---

To be safe, the CPU also needs to know *the highest value held in any register during this process* so that it can decide how much memory to allocate to these operations. For example, in the above instructions, the highest value ever held was `10` (in register `c` after the third instruction was evaluated).

Your puzzle answer was `6026`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 9: Stream Processing ---

A large stream blocks your path. According to the locals, it's not safe to cross the stream at the moment because it's full of *garbage*. You look down at the stream; rather than water, you discover that it's a *stream of characters*.

You sit for a while and record part of the stream (your puzzle input). The characters represent *groups* - sequences that begin with `{` and end with `}`. Within a group, there are zero or more other things, separated by commas: either another *group* or *garbage*. Since groups can contain other groups, a `}` only closes the *most-recently-opened unclosed group* - that is, they are nestable. Your puzzle input represents a single, large group which itself contains many smaller ones.

Sometimes, instead of a group, you will find *garbage*. Garbage begins with `<` and ends with `>`. Between those angle brackets, almost any character can appear, including `{` and `}`. *Within* garbage, `<` has no special meaning.

In a futile attempt to clean up the garbage, some program has *canceled* some of the characters within it using `!`: inside garbage, *any* character that comes after `!` should be *ignored*, including `<`, `>`, and even another `!`.

You don't see any characters that deviate from these rules. Outside garbage, you only find well-formed groups, and garbage always terminates according to the rules above.

Here are some self-contained pieces of garbage:

- `<>`, empty garbage.
- `<random characters>`, garbage containing random characters.
- `<<<<>`, because the extra `<` are ignored.
- `<{!>}>`, because the first `>` is canceled.
- `<!!>`, because the second `!` is canceled, allowing the `>` to terminate the garbage.
- `<!!!>>`, because the second `!` and the first `>` are canceled.
- `<{o"i!a,<{i<a>`, which ends at the first `>`.

Here are some examples of whole streams and the number of groups they contain:

- `{}`, `1` group.
- `{{{}}}`, `3` groups.
- `{{},{}}`, also `3` groups.
- `{{{},{},{{}}}}`, `6` groups.
- `{<{},{},{{}}>}`, `1` group (which itself contains garbage).
- `{<a>,<a>,<a>,<a>}`, `1` group.
- `{{<a>},{<a>},{<a>},{<a>}}`, `5` groups.
- `{{<!>},{<!>},{<!>},{<a>}}`, `2` groups (since all but the last `>` are canceled).

Your goal is to find the total score for all groups in your input. Each group is assigned a *score* which is one more than the score of the group that immediately contains it. (The outermost group gets a score of `1`.)

- `{}`, score of `1`.
- `{{{}}}`, score of `1 + 2 + 3 = 6`.
- `{{},{}}`, score of `1 + 2 + 2 = 5`.
- `{{{},{},{{}}}}`, score of `1 + 2 + 3 + 3 + 3 + 4 = 16`.
- `{<a>,<a>,<a>,<a>}`, score of `1`.
- `{{<ab>},{<ab>},{<ab>},{<ab>}}`, score of `1 + 2 + 2 + 2 + 2 = 9`.
- `{{<!!>},{<!!>},{<!!>},{<!!>}}`, score of `1 + 2 + 2 + 2 + 2 = 9`.
- `{{<a!>},{<a!>},{<a!>},{<ab>}}`, score of `1 + 2 = 3`.

What is the total score for all groups in your input?

Your puzzle answer was `14204`.

--- Part Two ---

Now, you're ready to remove the garbage.

To prove you've removed it, you need to count all of the characters within the garbage. The leading and trailing `<` and `>` don't count, nor do any canceled characters or the `!` doing the canceling.

- `<>`, `0` characters.
- `<random characters>`, `17` characters.
- `<<<<>`, `3` characters.
- `<{!>}>`, `2` characters.
- `<!!>`, `0` characters.
- `<!!!>>`, `0` characters.
- `<{o"i!a,<{i<a>`, `10` characters.

How many non-canceled characters are within the garbage in your puzzle input?

Your puzzle answer was `6622`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 10: Knot Hash ---

You come across some programs that are trying to implement a software emulation of a hash based on knot-tying. The hash these programs are implementing isn't very strong, but you decide to help them anyway. You make a mental note to remind the Elves later not to invent their own cryptographic functions.

This hash function simulates tying a knot in a circle of string with 256 marks on it. Based on the input to be hashed, the function repeatedly selects a span of string, brings the ends together, and gives the span a half-twist to reverse the order of the marks within it. After doing this many times, the order of the marks is used to build the resulting hash.

```
  4--5   pinch   4  5           4   1
 /    \  5,0,1  / \/ \  twist  / \ / \
3      0  -->  3      0  -->  3   X   0
 \    /         \ /\ /         \ / \ /
  2--1           2  1           2   5
```

To achieve this, begin with a *list* of numbers from `0` to `255`, a *current position* which begins at `0` (the first element in the list), a *skip size* (which starts at `0`), and a sequence of *lengths* (your puzzle input). Then, for each length:

- *Reverse* the order of that *length* of elements in the *list*, starting with the element at the *current position*.
- *Move* the *current position* forward by that *length* plus the *skip size*.
- *Increase* the *skip size* by one.

The *list* is circular; if the *current position* and the *length* try to reverse elements beyond the end of the list, the operation reverses using as many extra elements as it needs from the front of the list. If the *current position* moves past the end of the list, it wraps around to the front. *Lengths* larger than the size of the *list* are invalid.

Here's an example using a smaller list:

Suppose we instead only had a circular list containing five elements, `0, 1, 2, 3, 4`, and were given input lengths of `3, 4, 1, 5`.

- The list begins as `[0] 1 2 3 4` (where square brackets indicate the *current position*).
- The first length, `3`, selects `([0] 1 2) 3 4` (where parentheses indicate the sublist to be reversed).
- After reversing that section (`0 1 2` into `2 1 0`), we get `([2] 1 0) 3 4`.
- Then, the *current position* moves forward by the *length*, `3`, plus the *skip size*, 0: `2 1 0 [3] 4`. Finally, the *skip size* increases to `1`.

- The second length, `4`, selects a section which wraps: `2 1) 0 ([3] 4`.
- The sublist `3 4 2 1` is reversed to form `1 2 4 3`: `4 3) 0 ([1] 2`.
- The *current position* moves forward by the *length* plus the *skip size*, a total of `5`, causing it not to move because it wraps around: `4 3 0 [1] 2`. The *skip size* increases to `2`.

- The third length, `1`, selects a sublist of a single element, and so reversing it has no effect.
- The *current position* moves forward by the *length* (`1`) plus the *skip size* (`2`): `4 [3] 0 1 2`. The *skip size* increases to `3`.

- The fourth length, `5`, selects every element starting with the second: `4) ([3] 0 1 2`. Reversing this sublist (`3 0 1 2 4` into `4 2 1 0 3`) produces: `3) ([4] 2 1 0`.
- Finally, the *current position* moves forward by `8`: `3 4 2 1 [0]`. The *skip size* increases to `4`.

In this example, the first two numbers in the list end up being `3` and `4`; to check the process, you can multiply them together to produce `12`.

However, you should instead use the standard list size of `256` (with values `0` to `255`) and the sequence of *lengths* in your puzzle input. Once this process is complete, *what is the result of multiplying the first two numbers in the list*?

Your puzzle answer was `1980`.

--- Part Two ---

The logic you've constructed forms a single *round* of the *Knot Hash* algorithm; running the full thing requires many of these rounds. Some input and output processing is also required.

First, from now on, your input should be taken not as a list of numbers, but as a string of bytes instead. Unless otherwise specified, convert characters to bytes using their ASCII codes. This will allow you to handle arbitrary ASCII strings, and it also ensures that your input lengths are never larger than `255`. For example, if you are given `1,2,3`, you should convert it to the ASCII codes for each character: `49,44,50,44,51`.

Once you have determined the sequence of lengths to use, add the following lengths to the end of the sequence: `17, 31, 73, 47, 23`. For example, if you are given `1,2,3`, your final sequence of lengths should be `49,44,50,44,51,17,31,73,47,23` (the ASCII codes from the input string combined with the standard length suffix values).

Second, instead of merely running one *round* like you did above, run a total of `64` rounds, using the same *length* sequence in each round. The *current position* and *skip size* should be preserved between rounds. For example, if the previous example was your first round, you would start your second round with the same *length* sequence (`3, 4, 1, 5, 17, 31, 73, 47, 23`, now assuming they came from ASCII codes and include the suffix), but start with the previous round's *current position* (`4`) and *skip size* (`4`).

Once the rounds are complete, you will be left with the numbers from `0` to `255` in some order, called the *sparse hash*. Your next task is to reduce these to a list of only `16` numbers called the *dense hash*. To do this, use numeric bitwise XOR to combine each consecutive block of `16` numbers in the sparse hash (there are `16` such blocks in a list of `256` numbers). So, the first element in the dense hash is the first sixteen elements of the sparse hash XOR'd together, the second element in the dense hash is the second sixteen elements of the sparse hash XOR'd together, etc.

For example, if the first sixteen elements of your sparse hash are as shown below, and the XOR operator is `^`, you would calculate the first output number like this:

```
65 ^ 27 ^ 9 ^ 1 ^ 4 ^ 3 ^ 40 ^ 50 ^ 91 ^ 7 ^ 6 ^ 0 ^ 2 ^ 5 ^ 68 ^ 22 = 64
```

Perform this operation on each of the sixteen blocks of sixteen numbers in your sparse hash to determine the sixteen numbers in your dense hash.

Finally, the standard way to represent a Knot Hash is as a single hexadecimal string; the final output is the dense hash in hexadecimal notation. Because each number in your dense hash will be between `0` and `255` (inclusive), always represent each number as two hexadecimal digits (including a leading zero as necessary). So, if your first three numbers are `64, 7, 255`, they correspond to the hexadecimal numbers `40, 07, ff`, and so the first six characters of the hash would be `4007ff`. Because every Knot Hash is sixteen such numbers, the hexadecimal representation is always `32` hexadecimal digits (`0`-`f`) long.

Here are some example hashes:

- The empty string becomes `a2582a3a0e66e6e86e3812dcb672a272`.
- `AoC 2017` becomes `33efeb34ea91902bb2f59c9920caa6cd`.
- `1,2,3` becomes `3efbe78a8d82f29979031a4aa0b16a9d`.
- `1,2,4` becomes `63960835bcdc130f0b66d7ff4f6a5a8e`.

Treating your puzzle input as a string of ASCII characters, *what is the Knot Hash of your puzzle input?* Ignore any leading or trailing whitespace you might encounter.

Your puzzle answer was `899124dac21012ebc32e2f4d11eaec55`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 11: Hex Ed ---

Crossing the bridge, you've barely reached the other side of the stream when a program comes up to you, clearly in distress. "It's my child process," she says, "he's gotten lost in an infinite grid!"

Fortunately for her, you have plenty of experience with infinite grids.

Unfortunately for you, it's a hex grid.

The hexagons ("hexes") in this grid are aligned such that adjacent hexes can be found to the north, northeast, southeast, south, southwest, and northwest:

```
  \ n  /
nw +--+ ne
  /    \
-+      +-
  \    /
sw +--+ se
  / s  \
```

You have the path the child process took. Starting where he started, you need to determine the fewest number of steps required to reach him. (A "step" means to move from the hex you are in to any adjacent hex.)

For example:

- `ne,ne,ne` is `3` steps away.
- `ne,ne,sw,sw` is `0` steps away (back where you started).
- `ne,ne,s,s` is `2` steps away (`se,se`).
- `se,sw,se,sw,sw` is `3` steps away (`s,s,sw`).

Your puzzle answer was `818`.

--- Part Two ---

How many steps away is the *furthest* he ever got from his starting position?

Your puzzle answer was `1596`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 12: Digital Plumber ---

Walking along the memory banks of the stream, you find a small village that is experiencing a little confusion: some programs can't communicate with each other.

Programs in this village communicate using a fixed system of *pipes*. Messages are passed between programs using these pipes, but most programs aren't connected to each other directly. Instead, programs pass messages between each other until the message reaches the intended recipient.

For some reason, though, some of these messages aren't ever reaching their intended recipient, and the programs suspect that some pipes are missing. They would like you to investigate.

You walk through the village and record the ID of each program and the IDs with which it can communicate directly (your puzzle input). Each program has one or more programs with which it can communicate, and these pipes are bidirectional; if `8` says it can communicate with `11`, then `11` will say it can communicate with `8`.

You need to figure out how many programs are in the group that contains program ID `0`.

For example, suppose you go door-to-door like a travelling salesman and record the following list:

```
0 <-> 2
1 <-> 1
2 <-> 0, 3, 4
3 <-> 2, 4
4 <-> 2, 3, 6
5 <-> 6
6 <-> 4, 5
```

In this example, the following programs are in the group that contains program ID `0`:

- Program `0` by definition.
- Program `2`, directly connected to program `0`.
- Program `3` via program `2`.
- Program `4` via program `2`.
- Program `5` via programs `6`, then `4`, then `2`.
- Program `6` via programs `4`, then `2`.

Therefore, a total of `6` programs are in this group; all but program `1`, which has a pipe that connects it to itself.

How many programs are in the group that contains program ID `0`?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 13: Packet Scanners ---

You need to cross a vast *firewall*. The firewall consists of several layers, each with a *security scanner* that moves back and forth across the layer. To succeed, you must not be detected by a scanner.

By studying the firewall briefly, you are able to record (in your puzzle input) the *depth* of each layer and the *range* of the scanning area for the scanner within it, written as **`depth: range`**. Each layer has a thickness of exactly `1`. A layer at depth `0` begins immediately inside the firewall; a layer at depth `1` would start immediately after that.

For example, suppose you've recorded the following:

```
0: 3
1: 2
4: 4
6: 4
```

This means that there is a layer immediately inside the firewall (with range `3`), a second layer immediately after that (with range `2`), a third layer which begins at depth `4` (with range `4`), and a fourth layer which begins at depth 6 (also with range `4`). Visually, it might look like this:

```
 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
                [ ]     [ ]
```

Within each layer, a security scanner moves back and forth within its range. Each security scanner starts at the top and moves down until it reaches the bottom, then moves up until it reaches the top, and repeats. A security scanner takes *one picosecond* to move one step. Drawing scanners as `S`, the first few picoseconds look like this:

```
Picosecond 0:
 0   1   2   3   4   5   6
[S] [S] ... ... [S] ... [S]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
                [ ]     [ ]

Picosecond 1:
 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]

Picosecond 2:
 0   1   2   3   4   5   6
[ ] [S] ... ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]

Picosecond 3:
 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] ... [ ]
[S] [S]         [ ]     [ ]
[ ]             [ ]     [ ]
                [S]     [S]
```

Your plan is to hitch a ride on a packet about to move through the firewall. The packet will travel along the top of each layer, and it moves at *one layer per picosecond*. Each picosecond, the packet moves one layer forward (its first move takes it into layer 0), and then the scanners move one step. If there is a scanner at the top of the layer *as your packet enters it*, you are *caught*. (If a scanner moves into the top of its layer while you are there, you are *not* caught: it doesn't have time to notice you before you leave.) If you were to do this in the configuration above, marking your current position with parentheses, your passage through the firewall would look like this:

```
Initial state:
 0   1   2   3   4   5   6
[S] [S] ... ... [S] ... [S]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
                [ ]     [ ]

Picosecond 0:
 0   1   2   3   4   5   6
(S) [S] ... ... [S] ... [S]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
( ) [ ] ... ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]


Picosecond 1:
 0   1   2   3   4   5   6
[ ] ( ) ... ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] (S) ... ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
```

```
[S]             [S]     [S]
                [ ]     [ ]


Picosecond 2:
 0   1   2   3   4   5   6
[ ] [S] (.) ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] (.) ... [ ] ... [ ]
[S] [S]         [ ]     [ ]
[ ]             [ ]     [ ]
                [S]     [S]


Picosecond 3:
 0   1   2   3   4   5   6
[ ] [ ] ... (.) [ ] ... [ ]
[S] [S]         [ ]     [ ]
[ ]             [ ]     [ ]
                [S]     [S]

 0   1   2   3   4   5   6
[S] [S] ... (.) [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[ ]             [S]     [S]
                [ ]     [ ]


Picosecond 4:
 0   1   2   3   4   5   6
[S] [S] ... ... ( ) ... [ ]
[ ] [ ]         [ ]     [ ]
[ ]             [S]     [S]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] ... ... ( ) ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]
```

```
Picosecond 5:
 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] (.) [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [S] ... ... [S] (.) [S]
[ ] [ ]         [ ]     [ ]
[S]             [ ]     [ ]
                [ ]     [ ]


Picosecond 6:
 0   1   2   3   4   5   6
[ ] [S] ... ... [S] ... (S)
[ ] [ ]         [ ]     [ ]
[S]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] ... ( )
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]
```

In this situation, you are *caught* in layers `0` and `6`, because your packet entered the layer when its scanner was at the top when you entered it. You are *not* caught in layer `1`, since the scanner moved into the top of the layer once you were already there.

The *severity* of getting caught on a layer is equal to its *depth* multiplied by its *range*. (Ignore layers in which you do not get caught.) The severity of the whole trip is the sum of these values. In the example above, the trip severity is `0*3 + 6*4 = 24`.

Given the details of the firewall you've recorded, if you leave immediately, *what is the severity of your whole trip*?

Your puzzle answer was `1728`.

--- Part Two ---

Now, you need to pass through the firewall without being caught - easier said than done.

You can't control the speed of the packet, but you can *delay* it any number of picoseconds. For each picosecond you delay the packet before beginning your trip, all security scanners move one step. You're not in the firewall during this time; you don't enter layer `0` until you stop delaying the packet.

In the example above, if you delay `10` picoseconds (picoseconds `0 - 9`), you won't get caught:

```
State after delaying:
 0   1   2   3   4   5   6
[ ] [S] ... ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]

Picosecond 10:
 0   1   2   3   4   5   6
( ) [S] ... ... [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]

 0   1   2   3   4   5   6
( ) [ ] ... ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]


Picosecond 11:
 0   1   2   3   4   5   6
[ ] ( ) ... ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
[S] (S) ... ... [S] ... [S]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
                [ ]     [ ]


Picosecond 12:
 0   1   2   3   4   5   6
[S] [S] (.) ... [S] ... [S]
[ ] [ ]         [ ]     [ ]
[ ]             [ ]     [ ]
```

```
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] (.) ... [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]


Picosecond 13:
 0   1   2   3   4   5   6
[ ] [ ] ... (.) [ ] ... [ ]
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [S] ... (.) [ ] ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]


Picosecond 14:
 0   1   2   3   4   5   6
[ ] [S] ... ... ( ) ... [ ]
[ ] [ ]         [ ]     [ ]
[S]             [S]     [S]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] ... ... ( ) ... [ ]
[S] [S]         [ ]     [ ]
[ ]             [ ]     [ ]
                [S]     [S]


Picosecond 15:
 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] (.) [ ]
[S] [S]         [ ]     [ ]
[ ]             [ ]     [ ]
                [S]     [S]

 0   1   2   3   4   5   6
[S] [S] ... ... [ ] (.) [ ]
```

```
[ ] [ ]         [ ]     [ ]
[ ]             [S]     [S]
                [ ]     [ ]


Picosecond 16:
 0   1   2   3   4   5   6
[S] [S] ... ... [ ] ... ( )
[ ] [ ]         [ ]     [ ]
[ ]             [S]     [S]
                [ ]     [ ]

 0   1   2   3   4   5   6
[ ] [ ] ... ... [ ] ... ( )
[S] [S]         [S]     [S]
[ ]             [ ]     [ ]
                [ ]     [ ]
```

Because all smaller delays would get you caught, the fewest number of picoseconds you would need to delay to get through safely is `10`.

What is the fewest number of picoseconds that you need to delay the packet to pass through the firewall without being caught?

Your puzzle answer was `3946838`.

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

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

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

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 14: Disk Defragmentation ---

Suddenly, a scheduled job activates the system's disk defragmenter. Were the situation different, you might sit and watch it for a while, but today, you just don't have that kind of time. It's soaking up valuable system resources that are needed elsewhere, and so the only option is to help it finish its task as soon as possible.

The disk in question consists of a 128x128 grid; each square of the grid is either *free* or *used*. On this disk, the state of the grid is tracked by the bits in a sequence of knot hashes.

A total of 128 knot hashes are calculated, each corresponding to a single row in the grid; each hash contains 128 bits which correspond to individual grid squares. Each bit of a hash indicates whether that square is *free* (`0`) or *used* (`1`).

The hash inputs are a key string (your puzzle input), a dash, and a number from `0` to `127` corresponding to the row. For example, if your key string were `flqrgnkx`, then the first row would be given by the bits of the knot hash of `flqrgnkx-0`, the second row from the bits of the knot hash of `flqrgnkx-1`, and so on until the last row, `flqrgnkx-127`.

The output of a knot hash is traditionally represented by 32 hexadecimal digits; each of these digits correspond to 4 bits, for a total of `4 * 32 = 128` bits. To convert to bits, turn each hexadecimal digit to its equivalent binary value, high-bit first: `0` becomes `0000`, `1` becomes `0001`, `e` becomes `1110`, `f` becomes `1111`, and so on; a hash that begins with `a0c2017...` in hexadecimal would begin with `10100000110000100000000101110000...` in binary.

Continuing this process, the *first 8 rows and columns* for key `flqrgnkx` appear as follows, using `#` to denote used squares, and `.` to denote free ones:

```
##.#.#..-->
.#.#.#.#
....#.#.
#.#.##.#
.##.#...
##..#..#
.#...#..
##.#.##.-->
|      |
V      V
```

In this example, `8108` squares are used across the entire 128x128 grid.

Given your actual key string, *how many squares are used*?

Your puzzle input is `jzgqcdpd`.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 15: Dueling Generators ---

Here, you encounter a pair of dueling generators. The generators, called *generator A* and *generator B*, are trying to agree on a sequence of numbers. However, one of them is malfunctioning, and so the sequences don't always match.

As they do this, a *judge* waits for each of them to generate its next value, compares the lowest 16 bits of both values, and keeps track of the number of times those parts of the values match.

The generators both work on the same principle. To create its next value, a generator will take the previous value it produced, multiply it by a *factor* (generator A uses `16807`; generator B uses `48271`), and then keep the remainder of dividing that resulting product by `2147483647`. That final remainder is the value it produces next.

To calculate each generator's first value, it instead uses a specific starting value as its "previous value" (as listed in your puzzle input).

For example, suppose that for starting values, generator A uses `65`, while generator B uses `8921`. Then, the first five pairs of generated values are:

```
--Gen. A--  --Gen. B--
   1092455   430625591
1181022009  1233683848
 245556042  1431495498
1744312007   137874439
1352636452   285222916
```

In binary, these pairs are (with generator A's value first in each pair):

```
00000000000100001010101101100111
00011001101010101101001100110111

01000110011001001111011100111001
01001001100010001000010110001000

00001110101000101110001101001010
01010101010100101110001101001010

01100111111110000001011011000111
00001000001101111100110000000111

01010000100111111001100000100100
00010001000000000010100000000100
```

Here, you can see that the lowest (here, rightmost) 16 bits of the third value match: `1110001101001010`. Because of this one match, after processing these five pairs, the judge would have added only `1` to its total.

To get a significant sample, the judge would like to consider *40 million* pairs. (In the example above, the judge would eventually find a total of `588` pairs that match in their lowest 16 bits.)

After 40 million pairs, *what is the judge's final count*?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 16: Permutation Promenade ---

You come upon a very unusual sight; a group of programs here appear to be dancing.

There are sixteen programs in total, named `a` through `p`. They start by standing in a line: `a` stands in position `0`, `b` stands in position `1`, and so on until `p`, which stands in position `15`.

The programs' *dance* consists of a sequence of *dance moves*:

- *Spin*, written `sX`, makes `X` programs move from the end to the front, but maintain their order otherwise. (For example, `s3` on `abcde` produces `cdeab`).
- *Exchange*, written `xA/B`, makes the programs at positions `A` and `B` swap places.
- *Partner*, written `pA/B`, makes the programs named `A` and `B` swap places.

For example, with only five programs standing in a line (`abcde`), they could do the following dance:

- `s1`, a spin of size `1`: `eabcd`.
- `x3/4`, swapping the last two programs: `eabdc`.
- `pe/b`, swapping programs `e` and `b`: `baedc`.

After finishing their dance, the programs end up in order `baedc`.

You watch the dance for a while and record their dance moves (your puzzle input). *In what order are the programs standing* after their dance?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 17: Spinlock ---

Suddenly, whirling in the distance, you notice what looks like a massive, pixelated hurricane: a deadly spinlock. This spinlock isn't just consuming computing power, but memory, too; vast, digital mountains are being ripped from the ground and consumed by the vortex.

If you don't move quickly, fixing that printer will be the least of your problems.

This spinlock's algorithm is simple but efficient, quickly consuming everything in its path. It starts with a circular buffer containing only the value `0`, which it marks as the *current position.* It then steps forward through the circular buffer some number of steps (your puzzle input) before inserting the first new value, `1`, after the value it stopped on. The inserted value becomes the *current position.* Then, it steps forward from there the same number of steps, and wherever it stops, inserts after it the second new value, `2`, and uses that as the new *current position* again.

It repeats this process of *stepping forward*, *inserting a new value*, and *using the location of the inserted value as the new current position* a total of `2017` times, inserting `2017` as its final operation, and ending with a total of `2018` values (including `0`) in the circular buffer.

For example, if the spinlock were to step `3` times per insert, the circular buffer would begin to evolve like this (using parentheses to mark the current position after each iteration of the algorithm):

- `(0)`, the initial state before any insertions.
- `0 (1)`: the spinlock steps forward three times (`0`, `0`, `0`), and then inserts the first value, `1`, after it. `1` becomes the current position.
- `0 (2) 1`: the spinlock steps forward three times (`0`, `1`, `0`), and then inserts the second value, `2`, after it. `2` becomes the current position.
- `0  2 (3) 1`: the spinlock steps forward three times (`1`, `0`, `2`), and then inserts the third value, `3`, after it. `3` becomes the current position.

And so on:

- `0  2 (4) 3  1`
- `0 (5) 2  4  3  1`
- `0  5  2  4  3 (6) 1`
- `0  5 (7) 2  4  3  6  1`
- `0  5  7  2  4  3 (8) 6  1`
- `0 (9) 5  7  2  4  3  8  6  1`

Eventually, after 2017 insertions, the section of the circular buffer near the last insertion looks like this:

```
1512  1134  151 (2017) 638  1513  851
```

Perhaps, if you can identify the value that will ultimately be *after* the last value written (2017), you can short-circuit the spinlock. In this example, that would be `638`.

What is the value after `2017` in your completed circular buffer?

Your puzzle input is `382`.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 18: Duet ---

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

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

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

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

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

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

For example:

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

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

back two instructions to another jump, which jumps again to the `rcv`, which ultimately triggers the *recover* operation.

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

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

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 19: A Series of Tubes ---

Somehow, a network packet got lost and ended up here. It's trying to follow a routing diagram (your puzzle input), but it's confused about where to go.

Its starting point is just off the top of the diagram. Lines (drawn with `|`, `-`, and `+`) show the path it needs to take, starting by going down onto the only line connected to the top of the diagram. It needs to follow this path until it reaches the end (located somewhere within the diagram) and stop there.

Sometimes, the lines cross over each other; in these cases, it needs to continue going the same direction, and only turn left or right when there's no other option. In addition, someone has left *letters* on the line; these also don't change its direction, but it can use them to keep track of where it's been. For example:

```
     |          
     |  +--+    
     A  |  C    
 F---|----E|--+ 
     |  |  |  D 
     +B-+  +--+ 
```

Given this diagram, the packet needs to take the following path:

- Starting at the only line touching the top of the diagram, it must go down, pass through `A`, and continue onward to the first `+`.
- Travel right, up, and right, passing through `B` in the process.
- Continue down (collecting `C`), right, and up (collecting `D`).
- Finally, go all the way left through `E` and stopping at `F`.

Following the path to the end, the letters it sees on its path are `ABCDEF`.

The little packet looks up at you, hoping you can help it find the way. *What letters will it see* (in the order it would see them) if it follows the path? (The routing diagram is very wide; make sure you view it without line wrapping.)

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 20: Particle Swarm ---

Suddenly, the GPU contacts you, asking for help. Someone has asked it to simulate *too many particles*, and it won't be able to finish them all in time to render the next frame at this rate.

It transmits to you a buffer (your puzzle input) listing each particle in order (starting with particle `0`, then particle `1`, particle `2`, and so on). For each particle, it provides the `X`, `Y`, and `Z` coordinates for the particle's position (`p`), velocity (`v`), and acceleration (`a`), each in the format `<X,Y,Z>`.

Each tick, all particles are updated simultaneously. A particle's properties are updated in the following order:

- Increase the `X` velocity by the `X` acceleration.
- Increase the `Y` velocity by the `Y` acceleration.
- Increase the `Z` velocity by the `Z` acceleration.
- Increase the `X` position by the `X` velocity.
- Increase the `Y` position by the `Y` velocity.
- Increase the `Z` position by the `Z` velocity.

Because of seemingly tenuous rationale involving z-buffering, the GPU would like to know which particle will stay closest to position `<0,0,0>` in the long term. Measure this using the Manhattan distance, which in this situation is simply the sum of the absolute values of a particle's `X`, `Y`, and `Z` position.

For example, suppose you are only given two particles, both of which stay entirely on the X-axis (for simplicity). Drawing the current states of particles `0` and `1` (in that order) with an adjacent a number line and diagram of current `X` positions (marked in parentheses), the following would take place:

```
p=< 3,0,0>, v=< 2,0,0>, a=<-1,0,0>    -4 -3 -2 -1  0  1  2  3  4
p=< 4,0,0>, v=< 0,0,0>, a=<-2,0,0>                         (0)(1)

p=< 4,0,0>, v=< 1,0,0>, a=<-1,0,0>    -4 -3 -2 -1  0  1  2  3  4
p=< 2,0,0>, v=<-2,0,0>, a=<-2,0,0>                      (1)   (0)

p=< 4,0,0>, v=< 0,0,0>, a=<-1,0,0>    -4 -3 -2 -1  0  1  2  3  4
p=<-2,0,0>, v=<-4,0,0>, a=<-2,0,0>          (1)               (0)

p=< 3,0,0>, v=<-1,0,0>, a=<-1,0,0>    -4 -3 -2 -1  0  1  2  3  4
p=<-8,0,0>, v=<-6,0,0>, a=<-2,0,0>                         (0)
```

At this point, particle `1` will never be closer to `<0,0,0>` than particle `0`, and so, in the long run, particle `0` will stay closest.

Which particle will stay closest to position `<0,0,0>` in the long term?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 21: Fractal Art ---

You find a program trying to generate some art. It uses a strange process that involves repeatedly enhancing the detail of an image through a set of rules.

The image consists of a two-dimensional square grid of pixels that are either on (`#`) or off (`.`). The program always begins with this pattern:

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

Because the pattern is both `3` pixels wide and `3` pixels tall, it is said to have a *size* of `3`.

Then, the program repeats the following process:

- If the size is evenly divisible by `2`, break the pixels up into `2x2` squares, and convert each `2x2` square into a `3x3` square by following the corresponding *enhancement rule*.
- Otherwise, the size is evenly divisible by `3`; break the pixels up into `3x3` squares, and convert each `3x3` square into a `4x4` square by following the corresponding *enhancement rule*.

Because each square of pixels is replaced by a larger one, the image gains pixels and so its *size* increases.

The artist's book of enhancement rules is nearby (your puzzle input); however, it seems to be missing rules. The artist explains that sometimes, one must *rotate* or *flip* the input pattern to find a match. (Never rotate or flip the output pattern, though.) Each pattern is written concisely: rows are listed as single units, ordered top-down, and separated by slashes. For example, the following rules correspond to the adjacent patterns:

```
../.#  =  ..
          .#

                .#.
.#./..#/###  =  ..#
                ###

                        #..#
#..#/..../#..#/.##.  =  ....
```

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

When searching for a rule to use, rotate and flip the pattern as necessary. For example, all of the following patterns match the same rule:

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

Suppose the book contained the following two rules:

```
../.# => ##./#../...
.#./..#/### => #..#/..../..../#..#
```

As before, the program begins with this pattern:

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

The size of the grid (`3`) is not divisible by `2`, but it is divisible by `3`. It divides evenly into a single square; the square matches the second rule, which produces:

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

The size of this enhanced grid (`4`) is evenly divisible by `2`, so that rule is used. It divides evenly into four squares:

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

Each of these squares matches the same rule (`../.# => ##./#../...`), three of which require some flipping and rotation to line up with the rule. The output for the rule is the same in all four cases:

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

Finally, the squares are joined into a new grid:

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

Thus, after `2` iterations, the grid contains `12` pixels that are *on*.

How many pixels stay on after `5` iterations?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 22: Sporifica Virus ---

Diagnostics indicate that the local *grid computing cluster* has been contaminated with the *Sporifica Virus*. The grid computing cluster is a seemingly-infinite two-dimensional grid of compute nodes. Each node is either *clean* or *infected* by the virus.

To prevent overloading the nodes (which would render them useless to the virus) or detection by system administrators, exactly one *virus carrier* moves through the network, infecting or cleaning nodes as it moves. The virus carrier is always located on a single node in the network (the *current node*) and keeps track of the *direction* it is facing.

To avoid detection, the virus carrier works in bursts; in each burst, it *wakes up*, does some *work*, and goes back to *sleep*. The following steps are all executed *in order* one time each burst:

- If the *current node* is *infected*, it turns to its *right*. Otherwise, it turns to its *left*. (Turning is done in-place; the *current node* does not change.)
- If the *current node* is *clean*, it becomes *infected*. Otherwise, it becomes *cleaned*. (This is done *after* the node is considered for the purposes of changing direction.)
- The virus carrier moves *forward* one node in the direction it is facing.

Diagnostics have also provided a *map of the node infection status* (your puzzle input). *Clean* nodes are shown as `.`; *infected* nodes are shown as `#`. This map only shows the center of the grid; there are many more nodes beyond those shown, but none of them are currently infected.

The virus carrier begins in the middle of the map facing *up*.

For example, suppose you are given a map like this:

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

...

Then, the middle of the infinite grid looks like this, with the virus carrier's position marked with `[ ]`:

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

The virus carrier is on a *clean* node, so it turns *left*, *infects* the node, and moves left:

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

The virus carrier is on an *infected* node, so it turns *right*, *cleans* the node, and moves up:

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

Four times in a row, the virus carrier finds a *clean*, *infects* it, turns *left*, and moves forward, ending in the same place and still facing up:

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

Now on the same node as before, it sees an infection, which causes it to turn *right*, *clean* the node, and move forward:

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

After the above actions, a total of `7` bursts of activity had taken place. Of them, `5` bursts of activity caused an infection.

After a total of `70`, the grid looks like this, with the virus carrier facing up:

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

By this time, `41` bursts of activity caused an infection (though most of those nodes have since been cleaned).

After a total of `10000` bursts of activity, `5587` bursts will have caused an infection.

Given your actual map, after `10000` bursts of activity, *how many bursts cause a node to become infected*? (Do not count nodes that begin infected.)

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 23: Coprocessor Conflagration ---

You decide to head directly to the CPU and fix the printer from there. As you get close, you find an *experimental coprocessor* doing so much work that the local programs are afraid it will halt and catch fire. This would cause serious issues for the rest of the computer, so you head in and see what you can do.

The code it's running seems to be a variant of the kind you saw recently on that tablet. The general functionality seems *very similar*, but some of the instructions are different:

- `set X Y` *sets* register `X` to the value of `Y`.

- `sub X Y` *decreases* register `X` by the value of `Y`.
- `mul X Y` sets register `X` to the result of *multiplying* the value contained in register `X` by the value of `Y`.
- `jnz X Y` *jumps* with an offset of the value of `Y`, but only if the value of `X` is *not zero*. (An offset of `2` skips the next instruction, an offset of `-1` jumps to the previous instruction, and so on.)

The coprocessor is currently set to some kind of *debug mode*, which allows for testing, but prevents it from doing any meaningful work.

If you run the program (your puzzle input), *how many times is the `mul` instruction invoked?*

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 24: Electromagnetic Moat ---

The CPU itself is a large, black building surrounded by a bottomless pit. Enormous metal tubes extend outward from the side of the building at regular intervals and descend down into the void. There's no way to cross, but you need to get inside.

No way, of course, other than building a *bridge* out of the magnetic components strewn about nearby.

Each component has two *ports*, one on each end. The ports come in all different types, and only matching types can be connected. You take an inventory of the components by their port types (your puzzle input). Each port is identified by the number of *pins* it uses; more pins mean a stronger connection for your bridge. A `3/7` component, for example, has a type-`3` port on one side, and a type-`7` port on the other.

Your side of the pit is metallic; a perfect surface to connect a magnetic, *zero-pin port*. Because of this, the first port you use must be of type `0`. It doesn't matter what type of port you end with; your goal is just to make the bridge as strong as possible.

The *strength* of a bridge is the sum of the port types in each component. For example, if your bridge is made of components `0/3`, `3/7`, and `7/4`, your bridge has a strength of `0+3 + 3+7 + 7+4 = 24`.

For example, suppose you had the following components:

```
0/2
2/2
2/3
3/4
```

```
3/5
0/1
10/1
9/10
```

With them, you could make the following valid bridges:

- `0/1`
- `0/1--10/1`
- `0/1--10/1--9/10`
- `0/2`
- `0/2--2/3`
- `0/2--2/3--3/4`
- `0/2--2/3--3/5`
- `0/2--2/2`
- `0/2--2/2--2/3`
- `0/2--2/2--2/3--3/4`
- `0/2--2/2--2/3--3/5`

(Note how, as shown by `10/1`, order of ports within a component doesn't matter. However, you may only use each port on a component once.)

Of these bridges, the *strongest* one is `0/1--10/1--9/10`; it has a strength of `0+1 + 1+10 + 10+9 = 31`.

What is the strength of the strongest bridge you can make with the components you have available?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.

--- Day 25: The Halting Problem ---

Following the twisty passageways deeper and deeper into the CPU, you finally reach the core of the computer. Here, in the expansive central chamber, you find a grand apparatus that fills the entire room, suspended nanometers above your head.

You had always imagined CPUs to be noisy, chaotic places, bustling with activity. Instead, the room is quiet, motionless, and dark.

Suddenly, you and the CPU's *garbage collector* startle each other. "It's not often we get many visitors here!", he says. You inquire about the stopped machinery.

"It stopped milliseconds ago; not sure why. I'm a garbage collector, not a doctor." You ask what the machine is for.

"Programs these days, don't know their origins. That's the *Turing machine*! It's what makes the whole computer work." You try to explain that Turing machines are merely models of computation, but he cuts you off. "No, see, that's just what they *want* you to think. Ultimately, inside every CPU, there's a Turing machine driving the whole thing! Too bad this one's broken. We're doomed!"

You ask how you can help. "Well, unfortunately, the only way to get the computer running again would be to create a whole new Turing machine from scratch, but there's no *way* you can-" He notices the look on your face, gives you a curious glance, shrugs, and goes back to sweeping the floor.

You find the *Turing machine blueprints* (your puzzle input) on a tablet in a nearby pile of debris. Looking back up at the broken Turing machine above, you can start to identify its parts:

- A *tape* which contains `0` repeated infinitely to the left and right.
- A *cursor*, which can move left or right along the tape and read or write values at its current position.
- A set of *states*, each containing rules about what to do based on the current value under the cursor.

Each slot on the tape has two possible values: `0` (the starting value for all slots) and `1`. Based on whether the cursor is pointing at a `0` or a `1`, the current state says *what value to write* at the current position of the cursor, whether to *move the cursor* left or right one slot, and *which state to use next.*

For example, suppose you found the following blueprint:

```
Begin in state A.
Perform a diagnostic checksum after 6 steps.

In state A:
  If the current value is 0:
    - Write the value 1.
    - Move one slot to the right.
    - Continue with state B.
  If the current value is 1:
    - Write the value 0.
    - Move one slot to the left.
    - Continue with state B.

In state B:
  If the current value is 0:
    - Write the value 1.
    - Move one slot to the left.
    - Continue with state A.
  If the current value is 1:
    - Write the value 1.
    - Move one slot to the right.
    - Continue with state A.
```

Running it until the number of steps required to take the listed *diagnostic checksum* would result in the following tape configurations (with the *cursor* marked in square brackets):

```
... 0  0  0 [0] 0  0 ... (before any steps; about to run state A)
... 0  0  0  1 [0] 0 ... (after 1 step;     about to run state B)
... 0  0  0 [1] 1  0 ... (after 2 steps;    about to run state A)
... 0  0 [0] 0  1  0 ... (after 3 steps;    about to run state B)
... 0 [0] 1  0  1  0 ... (after 4 steps;    about to run state A)
... 0  1 [1] 0  1  0 ... (after 5 steps;    about to run state B)
... 0  1  1 [0] 1  0 ... (after 6 steps;    about to run state A)
```

The CPU can confirm that the Turing machine is working by taking a *diagnostic checksum* after a specific number of steps (given in the blueprint). Once the specified number of steps have been executed, the Turing machine should pause; once it does, count the number of times `1` appears on the tape. In the above example, the *diagnostic checksum* is ***3***.

Recreate the Turing machine and save the computer! *What is the diagnostic checksum* it produces once it's working again?

To begin, get your puzzle input.

Answer:

You can also [Shareon Twitter Mastodon] this puzzle.