--- Day 1: No Time for a Taxicab ---

Santa's sleigh uses a very high-precision clock to guide its movements, and the clock's oscillator is regulated by stars. Unfortunately, the stars have been stolen... by the Easter Bunny. To save Christmas, Santa needs you to retrieve all *fifty stars* by December 25th.

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

You're airdropped near *Easter Bunny Headquarters* in a city somewhere. "Near", unfortunately, is as close as you can get - the instructions on the Easter Bunny Recruiting Document the Elves intercepted start here, and nobody had time to work them out further.

The Document indicates that you should start at the given coordinates (where you just landed) and face North. Then, follow the provided sequence: either turn left (`L`) or right (`R`) 90 degrees, then walk forward the given number of blocks, ending at a new intersection.

There's no time to follow such ridiculous instructions on foot, though, so you take a moment and work out the destination. Given that you can only walk on the street grid of the city, how far is the shortest path to the destination?

For example:

- Following `R2, L3` leaves you `2` blocks East and `3` blocks North, or `5` blocks away.
- `R2, R2, R2` leaves you `2` blocks due South of your starting position, which is `2` blocks away.
- `R5, L5, R5, R3` leaves you `12` blocks away.

How many blocks away is Easter Bunny HQ?

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

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

--- Day 2: Bathroom Security ---

You arrive at *Easter Bunny Headquarters* under cover of darkness. However, you left in such a rush that you forgot to use the bathroom! Fancy office buildings like this one usually have keypad locks on their bathrooms, so you search the front desk for the code.

"In order to improve security," the document you find says, "bathroom codes will no longer be written down. Instead, please memorize and follow the procedure below to access the bathrooms."

The document goes on to explain that each button to be pressed can be found by starting on the previous button and moving to adjacent buttons on the

keypad: `U` moves up, `D` moves down, `L` moves left, and `R` moves right. Each line of instructions corresponds to one button, starting at the previous button (or, for the first line, *the "5" button*); press whatever button you're on at the end of each line. If a move doesn't lead to a button, ignore it.

You can't hold it much longer, so you decide to figure out the code as you walk to the bathroom. You picture a keypad like this:

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

Suppose your instructions are:

```
ULL
RRDDD
LURDL
UUUUD
```

- You start at "5" and move up (to "2"), left (to "1"), and left (you can't, and stay on "1"), so the first button is `1`.
- Starting from the previous button ("1"), you move right twice (to "3") and then down three times (stopping at "9" after two moves and ignoring the third), ending up with `9`.
- Continuing from "9", you move left, up, right, down, and left, ending with `8`.
- Finally, you move up four times (stopping at "2"), then down once, ending with `5`.

So, in this example, the bathroom code is `1985`.

Your puzzle input is the instructions from the document you found at the front desk. What is the *bathroom code*?

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

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

--- Day 3: Squares With Three Sides ---

Now that you can think clearly, you move deeper into the labyrinth of hallways and office furniture that makes up this part of Easter Bunny HQ. This must be a graphic design department; the walls are covered in specifications for triangles.

Or are they?

The design document gives the side lengths of each triangle it describes, but... `5 10 25`? Some of these aren't triangles. You can't help but mark the impossible ones.

In a valid triangle, the sum of any two sides must be larger than the remaining side. For example, the "triangle" given above is impossible, because `5 + 10` is

not larger than `25`.

In your puzzle input, *how many* of the listed triangles are *possible*?

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

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

--- Day 4: Security Through Obscurity ---

Finally, you come across an information kiosk with a list of rooms. Of course, the list is encrypted and full of decoy data, but the instructions to decode the list are barely hidden nearby. Better remove the decoy data first.

Each room consists of an encrypted name (lowercase letters separated by dashes) followed by a dash, a sector ID, and a checksum in square brackets.

A room is real (not a decoy) if the checksum is the five most common letters in the encrypted name, in order, with ties broken by alphabetization. For example:

- `aaaaa-bbb-z-y-x-123[abxyz]` is a real room because the most common letters are `a` (5), `b` (3), and then a tie between `x`, `y`, and `z`, which are listed alphabetically.
- `a-b-c-d-e-f-g-h-987[abcde]` is a real room because although the letters are all tied (1 of each), the first five are listed alphabetically.
- `not-a-real-room-404[oarel]` is a real room.
- `totally-real-room-200[decoy]` is not.

Of the real rooms from the list above, the sum of their sector IDs is `1514`.

What is the *sum of the sector IDs of the real rooms*?

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

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

--- Day 5: How About a Nice Game of Chess? ---

You are faced with a security door designed by Easter Bunny engineers that seem to have acquired most of their security knowledge by watching hacking movies.

The *eight-character password* for the door is generated one character at a time by finding the MD5 hash of some Door ID (your puzzle input) and an increasing integer index (starting with `0`).

A hash indicates the *next character* in the password if its hexadecimal representation starts with *five zeroes*. If it does, the sixth character in the hash is the next character of the password.

For example, if the Door ID is `abc`:

- The first index which produces a hash that starts with five zeroes is `3231929`, which we find by hashing `abc3231929`; the sixth character of the hash, and thus the first character of the password, is `1`.
- `5017308` produces the next interesting hash, which starts with `000008f82...`, so the second character of the password is `8`.
- The third time a hash starts with five zeroes is for `abc5278568`, discovering the character `f`.

In this example, after continuing this search a total of eight times, the password is `18f47a30`.

Given the actual Door ID, *what is the password*?

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

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

--- Day 6: Signals and Noise ---

Something is jamming your communications with Santa. Fortunately, your signal is only partially jammed, and protocol in situations like this is to switch to a simple repetition code to get the message through.

In this model, the same message is sent repeatedly. You've recorded the repeating message signal (your puzzle input), but the data seems quite corrupted - almost too badly to recover. *Almost.*

All you need to do is figure out which character is most frequent for each position. For example, suppose you had recorded the following messages:

```
eedadn
drvtee
eandsr
raavrd
atevrs
tsrnev
sdttsa
rasrtv
nssdts
ntnada
svetve
tesnvt
vntsnd
vrdear
dvrsen
enarar
```

The most common character in the first column is `e`; in the second, `a`; in the third, `s`, and so on. Combining these characters returns the error-corrected message, `easter`.

Given the recording in your puzzle input, *what is the error-corrected version* of the message being sent?

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

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

--- Day 7: Internet Protocol Version 7 ---

While snooping around the local network of EBHQ, you compile a list of IP addresses (they're IPv7, of course; IPv6 is much too limited). You'd like to figure out which IPs support *TLS* (transport-layer snooping).

An IP supports TLS if it has an Autonomous Bridge Bypass Annotation, or *ABBA*. An ABBA is any four-character sequence which consists of a pair of two different characters followed by the reverse of that pair, such as `xyyx` or `abba`. However, the IP also must not have an ABBA within any hypernet sequences, which are contained by *square brackets*.

For example:

- `abba[mnop]qrst` supports TLS (`abba` outside square brackets).
- `abcd[bddb]xyyx` does *not* support TLS (`bddb` is within square brackets, even though `xyyx` is outside square brackets).
- `aaaa[qwer]tyui` does *not* support TLS (`aaaa` is invalid; the interior characters must be different).
- `ioxxoj[asdfgh]zxcvbn` supports TLS (`oxxo` is outside square brackets, even though it's within a larger string).

How many IPs in your puzzle input support TLS?

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

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

--- Day 8: Two-Factor Authentication ---

You come across a door implementing what you can only assume is an implementation of two-factor authentication after a long game of requirements telephone.

To get past the door, you first swipe a keycard (no problem; there was one on a nearby desk). Then, it displays a code on a little screen, and you type that code on a keypad. Then, presumably, the door unlocks.

Unfortunately, the screen has been smashed. After a few minutes, you've taken everything apart and figured out how it works. Now you just have to work out what the screen *would* have displayed.

The magnetic strip on the card you swiped encodes a series of instructions for the screen; these instructions are your puzzle input. The screen is *`50` pixels wide*

and `6` pixels tall, all of which start *off*, and is capable of three somewhat peculiar operations:

- `rect AxB` turns *on* all of the pixels in a rectangle at the top-left of the screen which is `A` wide and `B` tall.
- `rotate row y=A by B` shifts all of the pixels in row `A` (0 is the top row) *right* by `B` pixels. Pixels that would fall off the right end appear at the left end of the row.
- `rotate column x=A by B` shifts all of the pixels in column `A` (0 is the left column) *down* by `B` pixels. Pixels that would fall off the bottom appear at the top of the column.

For example, here is a simple sequence on a smaller screen:

- `rect 3x2` creates a small rectangle in the top-left corner:

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

- `rotate column x=1 by 1` rotates the second column down by one pixel:

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

- `rotate row y=0 by 4` rotates the top row right by four pixels:

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

- `rotate column x=1 by 1` again rotates the second column down by one pixel, causing the bottom pixel to wrap back to the top:

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

As you can see, this display technology is extremely powerful, and will soon dominate the tiny-code-displaying-screen market. That's what the advertisement on the back of the display tries to convince you, anyway.

There seems to be an intermediate check of the voltage used by the display: after you swipe your card, if the screen did work, *how many pixels should be lit?*

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

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

--- Day 9: Explosives in Cyberspace ---

Wandering around a secure area, you come across a datalink port to a new part of the network. After briefly scanning it for interesting files, you find one file in particular that catches your attention. It's compressed with an experimental format, but fortunately, the documentation for the format is nearby.

The format compresses a sequence of characters. Whitespace is ignored. To indicate that some sequence should be repeated, a marker is added to the file, like `(10x2)`. To decompress this marker, take the subsequent `10` characters and repeat them `2` times. Then, continue reading the file *after* the repeated data. The marker itself is not included in the decompressed output.

If parentheses or other characters appear within the data referenced by a marker, that's okay - treat it like normal data, not a marker, and then resume looking for markers after the decompressed section.

For example:

- `ADVENT` contains no markers and decompresses to itself with no changes, resulting in a decompressed length of `6`.
- `A(1x5)BC` repeats only the `B` a total of `5` times, becoming `ABBBBBC` for a decompressed length of `7`.
- `(3x3)XYZ` becomes `XYZXYZXYZ` for a decompressed length of `9`.
- `A(2x2)BCD(2x2)EFG` doubles the `BC` and `EF`, becoming `ABCBCDEFEFG` for a decompressed length of `11`.
- `(6x1)(1x3)A` simply becomes `(1x3)A` - the `(1x3)` looks like a marker, but because it's within a data section of another marker, it is not treated any differently from the `A` that comes after it. It has a decompressed length of `6`.
- `X(8x2)(3x3)ABCY` becomes `X(3x3)ABC(3x3)ABCY` (for a decompressed length of `18`), because the decompressed data from the `(8x2)` marker (the `(3x3)ABC`) is skipped and not processed further.

What is the *decompressed length* of the file (your puzzle input)? Don't count whitespace.

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

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

--- Day 10: Balance Bots ---

You come upon a factory in which many robots are zooming around handing small microchips to each other.

Upon closer examination, you notice that each bot only proceeds when it has *two* microchips, and once it does, it gives each one to a different bot or puts it in a marked "output" bin. Sometimes, bots take microchips from "input" bins, too.

Inspecting one of the microchips, it seems like they each contain a single number; the bots must use some logic to decide what to do with each chip. You access the local control computer and download the bots' instructions (your puzzle input).

Some of the instructions specify that a specific-valued microchip should be given to a specific bot; the rest of the instructions indicate what a given bot should do with its *lower-value* or *higher-value* chip.

For example, consider the following instructions:

```
value 5 goes to bot 2
bot 2 gives low to bot 1 and high to bot 0
value 3 goes to bot 1
bot 1 gives low to output 1 and high to bot 0
bot 0 gives low to output 2 and high to output 0
value 2 goes to bot 2
```

- Initially, bot `1` starts with a value-`3` chip, and bot `2` starts with a value-`2` chip and a value-`5` chip.
- Because bot `2` has two microchips, it gives its lower one (`2`) to bot `1` and its higher one (`5`) to bot `0`.
- Then, bot `1` has two microchips; it puts the value-`2` chip in output `1` and gives the value-`3` chip to bot `0`.
- Finally, bot `0` has two microchips; it puts the `3` in output `2` and the `5` in output `0`.

In the end, output bin `0` contains a value-`5` microchip, output bin `1` contains a value-`2` microchip, and output bin `2` contains a value-`3` microchip. In this configuration, bot number `2` is responsible for comparing value-`5` microchips with value-`2` microchips.

Based on your instructions, *what is the number of the bot* that is responsible for comparing value-`61` microchips with value-`17` microchips?

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

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

--- Day 11: Radioisotope Thermoelectric Generators ---

You come upon a column of four floors that have been entirely sealed off from the rest of the building except for a small dedicated lobby. There are some radiation warnings and a big sign which reads "Radioisotope Testing Facility".

According to the project status board, this facility is currently being used to experiment with Radioisotope Thermoelectric Generators (RTGs, or simply "generators") that are designed to be paired with specially-constructed microchips. Basically, an RTG is a highly radioactive rock that generates electricity through heat.

The experimental RTGs have poor radiation containment, so they're dangerously radioactive. The chips are prototypes and don't have normal radiation shielding, but they do have the ability to *generate an electromagnetic radiation shield when powered*. Unfortunately, they can *only* be powered by their corresponding RTG. An RTG powering a microchip is still dangerous to other microchips.

In other words, if a chip is ever left in the same area as another RTG, and it's not connected to its own RTG, the chip will be *fried*. Therefore, it is assumed that you will follow procedure and keep chips connected to their corresponding RTG when they're in the same room, and away from other RTGs otherwise.

These microchips sound very interesting and useful to your current activities, and you'd like to try to retrieve them. The fourth floor of the facility has an assembling machine which can make a self-contained, shielded computer for you to take with you - that is, if you can bring it all of the RTGs and microchips.

Within the radiation-shielded part of the facility (in which it's safe to have these pre-assembly RTGs), there is an elevator that can move between the four floors. Its capacity rating means it can carry at most yourself and two RTGs or microchips in any combination. (They're rigged to some heavy diagnostic equipment - the assembling machine will detach it for you.) As a security measure, the elevator will only function if it contains at least one RTG or microchip. The elevator always stops on each floor to recharge, and this takes long enough that the items within it and the items on that floor can irradiate each other. (You can prevent this if a Microchip and its Generator end up on the same floor in this way, as they can be connected while the elevator is recharging.)

You make some notes of the locations of each component of interest (your puzzle input). Before you don a hazmat suit and start moving things around, you'd like to have an idea of what you need to do.

When you enter the containment area, you and the elevator will start on the first floor.

For example, suppose the isolated area has the following arrangement:

```
The first floor contains a hydrogen-compatible microchip and a lithium-compatible microchip.
The second floor contains a hydrogen generator.
The third floor contains a lithium generator.
The fourth floor contains nothing relevant.
```

As a diagram (`F#` for a Floor number, `E` for Elevator, `H` for Hydrogen, `L` for Lithium, `M` for Microchip, and `G` for Generator), the initial state looks like this:

```
F4 .  .  .  .  .
F3 .  .  .  LG .
F2 .  HG .  .  .
F1 E  .  HM .  LM
```

Then, to get everything up to the assembling machine on the fourth floor, the following steps could be taken:

- Bring the Hydrogen-compatible Microchip to the second floor, which is safe because it can get power from the Hydrogen Generator:

```
F4 .  .  .  .  .
F3 .  .  .  LG .
F2 E  HG HM .  .
F1 .  .  .  .  LM
```

- Bring both Hydrogen-related items to the third floor, which is safe because the Hydrogen-compatible microchip is getting power from its generator:

```
F4 .  .  .  .  .
F3 E  HG HM LG .
F2 .  .  .  .  .
F1 .  .  .  .  LM
```

- Leave the Hydrogen Generator on floor three, but bring the Hydrogen-compatible Microchip back down with you so you can still use the elevator:

```
F4 .  .  .  .  .
F3 .  HG .  LG .
F2 E  .  HM .  .
F1 .  .  .  .  LM
```

- At the first floor, grab the Lithium-compatible Microchip, which is safe because Microchips don't affect each other:

```
F4 .  .  .  .  .
F3 .  HG .  LG .
F2 .  .  .  .  .
F1 E  .  HM .  LM
```

- Bring both Microchips up one floor, where there is nothing to fry them:

```
F4 .  .  .  .  .
F3 .  HG .  LG .
F2 E  .  HM .  LM
F1 .  .  .  .  .
```

- Bring both Microchips up again to floor three, where they can be temporarily connected to their corresponding generators while the elevator recharges, preventing either of them from being fried:

```
F4 .  .  .  .  .
F3 E  HG HM LG LM
F2 .  .  .  .  .
F1 .  .  .  .  .
```

- Bring both Microchips to the fourth floor:

```
F4 E  .  HM .  LM
F3 .  HG .  LG .
```

```
F2 .  .  .  .  .
F1 .  .  .  .  .
```

- Leave the Lithium-compatible microchip on the fourth floor, but bring the Hydrogen-compatible one so you can still use the elevator; this is safe because although the Lithium Generator is on the destination floor, you can connect Hydrogen-compatible microchip to the Hydrogen Generator there:

```
F4 .  .  .  .  LM
F3 E  HG HM LG .
F2 .  .  .  .  .
F1 .  .  .  .  .
```

- Bring both Generators up to the fourth floor, which is safe because you can connect the Lithium-compatible Microchip to the Lithium Generator upon arrival:

```
F4 E  HG .  LG LM
F3 .  .  HM .  .
F2 .  .  .  .  .
F1 .  .  .  .  .
```

- Bring the Lithium Microchip with you to the third floor so you can use the elevator:

```
F4 .  HG .  LG .
F3 E  .  HM .  LM
F2 .  .  .  .  .
F1 .  .  .  .  .
```

- Bring both Microchips to the fourth floor:

```
F4 E  HG HM LG LM
F3 .  .  .  .  .
F2 .  .  .  .  .
F1 .  .  .  .  .
```

In this arrangement, it takes `11` steps to collect all of the objects at the fourth floor for assembly. (Each elevator stop counts as one step, even if nothing is added to or removed from it.)

In your situation, what is the *minimum number of steps* required to bring all of the objects to the fourth floor?

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

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

--- Day 12: Leonardo's Monorail ---

You finally reach the top floor of this building: a garden with a slanted glass ceiling. Looks like there are no more stars to be had.

While sitting on a nearby bench amidst some tiger lilies, you manage to decrypt some of the files you extracted from the servers downstairs.

According to these documents, Easter Bunny HQ isn't just this building - it's a collection of buildings in the nearby area. They're all connected by a local monorail, and there's another building not far from here! Unfortunately, being night, the monorail is currently not operating.

You remotely connect to the monorail control systems and discover that the boot sequence expects a password. The password-checking logic (your puzzle input) is easy to extract, but the code it uses is strange: it's assembunny code designed for the new computer you just assembled. You'll have to execute the code and get the password.

The assembunny code you've extracted operates on four registers (`a`, `b`, `c`, and `d`) that start at `0` and can hold any integer. However, it seems to make use of only a few instructions:

- `cpy x y` *copies* `x` (either an integer or the *value* of a register) into register `y`.
- `inc x` *increases* the value of register `x` by one.
- `dec x` *decreases* the value of register `x` by one.
- `jnz x y` *jumps* to an instruction `y` away (positive means forward; negative means backward), but only if `x` is *not zero*.

The `jnz` instruction moves relative to itself: an offset of `-1` would continue at the previous instruction, while an offset of `2` would *skip over* the next instruction.

For example:

```
cpy 41 a
inc a
inc a
dec a
jnz a 2
dec a
```

The above code would set register `a` to `41`, increase its value by `2`, decrease its value by `1`, and then skip the last `dec a` (because `a` is not zero, so the `jnz a 2` skips it), leaving register `a` at `42`. When you move past the last instruction, the program halts.

After executing the assembunny code in your puzzle input, *what value is left in register `a`?*

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

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

--- Day 13: A Maze of Twisty Little Cubicles ---

You arrive at the first floor of this new building to discover a much less welcoming environment than the shiny atrium of the last one. Instead, you are in a maze of twisty little cubicles, all alike.

Every location in this area is addressed by a pair of non-negative integers (`x,y`). Each such coordinate is either a wall or an open space. You can't move diagonally. The cube maze starts at `0,0` and seems to extend infinitely toward *positive* `x` and `y`; negative values are *invalid*, as they represent a location outside the building. You are in a small waiting area at `1,1`.

While it seems chaotic, a nearby morale-boosting poster explains, the layout is actually quite logical. You can determine whether a given `x,y` coordinate will be a wall or an open space using a simple system:

- Find `x*x + 3*x + 2*x*y + y + y*y`.
- Add the office designer's favorite number (your puzzle input).
- Find the binary representation of that sum; count the *number* of bits that are `1`.
  - If the number of bits that are `1` is *even*, it's an *open space*.
  - If the number of bits that are `1` is *odd*, it's a *wall*.

For example, if the office designer's favorite number were `10`, drawing walls as `#` and open spaces as `.`, the corner of the building containing `0,0` would look like this:

```
  0123456789
0 .#.####.##
1 ..#..#...#
2 #....##...
3 ###.#.###.
4 .##..#..#.
5 ..##....#.
6 #...##.###
```

Now, suppose you wanted to reach `7,4`. The shortest route you could take is marked as `O`:

```
  0123456789
0 .#.####.##
1 .O#..#...#
2 #OOO.##...
3 ###O#.###.
4 .##OO#OO#.
5 ..##OOO.#.
6 #...##.###
```

Thus, reaching `7,4` would take a minimum of `11` steps (starting from your current location, `1,1`).

What is the *fewest number of steps required* for you to reach `31,39`?

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

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

--- Day 14: One-Time Pad ---

In order to communicate securely with Santa while you're on this mission, you've been using a one-time pad that you generate using a pre-agreed algorithm. Unfortunately, you've run out of keys in your one-time pad, and so you need to generate some more.

To generate keys, you first get a stream of random data by taking the MD5 of a pre-arranged salt (your puzzle input) and an increasing integer index (starting with `0`, and represented in decimal); the resulting MD5 hash should be represented as a string of *lowercase* hexadecimal digits.

However, not all of these MD5 hashes are *keys*, and you need `64` new keys for your one-time pad. A hash is a key *only if*:

- It contains *three* of the same character in a row, like `777`. Only consider the first such triplet in a hash.
- One of the next `1000` hashes in the stream contains that same character *five* times in a row, like `77777`.

Considering future hashes for five-of-a-kind sequences does not cause those hashes to be skipped; instead, regardless of whether the current hash is a key, always resume testing for keys starting with the very next hash.

For example, if the pre-arranged salt is `abc`:

- The first index which produces a triple is `18`, because the MD5 hash of `abc18` contains `...cc38887a5....` However, index `18` does not count as a key for your one-time pad, because none of the next thousand hashes (index `19` through index `1018`) contain `88888`.
- The next index which produces a triple is `39`; the hash of `abc39` contains `eee`. It is also the first key: one of the next thousand hashes (the one at index 816) contains `eeeee`.
- None of the next six triples are keys, but the one after that, at index `92`, is: it contains `999` and index `200` contains `99999`.
- Eventually, index `22728` meets all of the criteria to generate the `64`th key.

So, using our example salt of `abc`, index `22728` produces the `64`th key.

Given the actual salt in your puzzle input, *what index* produces your `64`th one-time pad key?

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

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

--- Day 15: Timing is Everything ---

The halls open into an interior plaza containing a large kinetic sculpture. The sculpture is in a sealed enclosure and seems to involve a set of identical spherical capsules that are carried to the top and allowed to bounce through the maze of spinning pieces.

Part of the sculpture is even interactive! When a button is pressed, a capsule is dropped and tries to fall through slots in a set of rotating discs to finally go through a little hole at the bottom and come out of the sculpture. If any of the slots aren't aligned with the capsule as it passes, the capsule bounces off the disc and soars away. You feel compelled to get one of those capsules.

The discs pause their motion each second and come in different sizes; they seem to each have a fixed number of positions at which they stop. You decide to call the position with the slot `0`, and count up for each position it reaches next.

Furthermore, the discs are spaced out so that after you push the button, one second elapses before the first disc is reached, and one second elapses as the capsule passes from one disc to the one below it. So, if you push the button at `time=100`, then the capsule reaches the top disc at `time=101`, the second disc at `time=102`, the third disc at `time=103`, and so on.

The button will only drop a capsule at an integer time - no fractional seconds allowed.

For example, at `time=0`, suppose you see the following arrangement:

```
Disc #1 has 5 positions; at time=0, it is at position 4.
Disc #2 has 2 positions; at time=0, it is at position 1.
```

If you press the button exactly at `time=0`, the capsule would start to fall; it would reach the first disc at `time=1`. Since the first disc was at position `4` at `time=0`, by `time=1` it has ticked one position forward. As a five-position disc, the next position is `0`, and the capsule falls through the slot.

Then, at `time=2`, the capsule reaches the second disc. The second disc has ticked forward two positions at this point: it started at position `1`, then continued to position `0`, and finally ended up at position `1` again. Because there's only a slot at position `0`, the capsule bounces away.

If, however, you wait until `time=5` to push the button, then when the capsule reaches each disc, the first disc will have ticked forward `5+1 = 6` times (to position `0`), and the second disc will have ticked forward `5+2 = 7` times (also to position `0`). In this case, the capsule would fall through the discs and come out of the machine.

However, your situation has more than two discs; you've noted their positions in your puzzle input. What is the *first time you can press the button* to get a

capsule?

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

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

--- Day 16: Dragon Checksum ---

You're done scanning this part of the network, but you've left traces of your presence. You need to overwrite some disks with random-looking data to cover your tracks and update the local security system with a new checksum for those disks.

For the data to not be suspicious, it needs to have certain properties; purely random data will be detected as tampering. To generate appropriate random data, you'll need to use a modified dragon curve.

Start with an appropriate initial state (your puzzle input). Then, so long as you don't have enough data yet to fill the disk, repeat the following steps:

- Call the data you have at this point "a".
- Make a copy of "a"; call this copy "b".
- Reverse the order of the characters in "b".
- In "b", replace all instances of `0` with `1` and all `1`s with `0`.
- The resulting data is "a", then a single `0`, then "b".

For example, after a single step of this process,

- `1` becomes `100`.
- `0` becomes `001`.
- `11111` becomes `11111000000`.
- `111100001010` becomes `1111000010100101011110000`.

Repeat these steps until you have enough data to fill the desired disk.

Once the data has been generated, you also need to create a checksum of that data. Calculate the checksum *only* for the data that fits on the disk, even if you generated more data than that in the previous step.

The checksum for some given data is created by considering each non-overlapping *pair* of characters in the input data. If the two characters match (`00` or `11`), the next checksum character is a `1`. If the characters do not match (`01` or `10`), the next checksum character is a `0`. This should produce a new string which is exactly half as long as the original. If the length of the checksum is *even*, repeat the process until you end up with a checksum with an *odd* length.

For example, suppose we want to fill a disk of length `12`, and when we finally generate a string of at least length `12`, the first `12` characters are `110010110100`. To generate its checksum:

- Consider each pair: `11`, `00`, `10`, `11`, `01`, `00`.
- These are same, same, different, same, different, same, producing `110101`.

- The resulting string has length `6`, which is *even*, so we repeat the process.
- The pairs are `11` (same), `01` (different), `01` (different).
- This produces the checksum `100`, which has an *odd* length, so we stop.

Therefore, the checksum for `110010110100` is `100`.

Combining all of these steps together, suppose you want to fill a disk of length `20` using an initial state of `10000`:

- Because `10000` is too short, we first use the modified dragon curve to make it longer.
- After one round, it becomes `10000011110` (`11` characters), still too short.
- After two rounds, it becomes `1000001111001000011110` (`23` characters), which is enough.
- Since we only need `20`, but we have `23`, we get rid of all but the first `20` characters: `10000011110010000111`.
- Next, we start calculating the checksum; after one round, we have `0111110101`, which `10` characters long (*even*), so we continue.
- After two rounds, we have `01100`, which is `5` characters long (*odd*), so we are done.

In this example, the correct checksum would therefore be `01100`.

The first disk you have to fill has length `272`. Using the initial state in your puzzle input, *what is the correct checksum*?

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

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

--- Day 17: Two Steps Forward ---

You're trying to access a secure vault protected by a `4x4` grid of small rooms connected by doors. You start in the top-left room (marked `S`), and you can access the vault (marked `V`) once you reach the bottom-right room:

```
#########
#S| | | #
#-#-#-#-#
# | | | #
#-#-#-#-#
# | | | #
#-#-#-#-#
# | | |
####### V
```

Fixed walls are marked with `#`, and doors are marked with `-` or `|`.

The doors in your *current room* are either open or closed (and locked) based on the hexadecimal MD5 hash of a passcode (your puzzle input) followed by a

sequence of uppercase characters representing the *path you have taken so far* (`U` for up, `D` for down, `L` for left, and `R` for right).

Only the first four characters of the hash are used; they represent, respectively, the doors *up, down, left, and right* from your current position. Any `b`, `c`, `d`, `e`, or `f` means that the corresponding door is *open*; any other character (any number or `a`) means that the corresponding door is *closed and locked*.

To access the vault, all you need to do is reach the bottom-right room; reaching this room opens the vault and all doors in the maze.

For example, suppose the passcode is `hijkl`. Initially, you have taken no steps, and so your path is empty: you simply find the MD5 hash of `hijkl` alone. The first four characters of this hash are `ced9`, which indicate that up is open (`c`), down is open (`e`), left is open (`d`), and right is closed and locked (`9`). Because you start in the top-left corner, there are no "up" or "left" doors to be open, so your only choice is *down*.

Next, having gone only one step (down, or `D`), you find the hash of `hijklD`. This produces `f2bc`, which indicates that you can go back up, left (but that's a wall), or right. Going right means hashing `hijklDR` to get `5745` - all doors closed and locked. However, going *up* instead is worthwhile: even though it returns you to the room you started in, your path would then be `DU`, opening a *different set of doors*.

After going `DU` (and then hashing `hijklDU` to get `528e`), only the right door is open; after going `DUR`, all doors lock. (Fortunately, your actual passcode is not `hijkl`).

Passcodes actually used by Easter Bunny Vault Security do allow access to the vault if you know the right path. For example:

- If your passcode were `ihgpwlah`, the shortest path would be `DDRRRD`.
- With `kglvqrro`, the shortest path would be `DDUDRLRRUDRD`.
- With `ulqzkmiv`, the shortest would be `DRURDRUDDLLDLUURRDULRLDUUDDDRR`.

Given your vault's passcode, *what is the shortest path* (the actual path, not just the length) to reach the vault?

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

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

--- Day 18: Like a Rogue ---

As you enter this room, you hear a loud click! Some of the tiles in the floor here seem to be pressure plates for traps, and the trap you just triggered has run out of... whatever it tried to do to you. You doubt you'll be so lucky next time.

Upon closer examination, the traps and safe tiles in this room seem to follow a pattern. The tiles are arranged into rows that are all the same width; you take

note of the safe tiles (`.`) and traps (`^`) in the first row (your puzzle input).

The type of tile (trapped or safe) in each row is based on the types of the tiles in the same position, and to either side of that position, in the previous row. (If either side is off either end of the row, it counts as "safe" because there isn't a trap embedded in the wall.)

For example, suppose you know the first row (with tiles marked by letters) and want to determine the next row (with tiles marked by numbers):

```
ABCDE
12345
```

The type of tile `2` is based on the types of tiles `A`, `B`, and `C`; the type of tile `5` is based on tiles `D`, `E`, and an imaginary "safe" tile. Let's call these three tiles from the previous row the *left*, *center*, and *right* tiles, respectively. Then, a new tile is a *trap* only in one of the following situations:

- Its *left* and *center* tiles are traps, but its *right* tile is not.
- Its *center* and *right* tiles are traps, but its *left* tile is not.
- Only its *left* tile is a trap.
- Only its *right* tile is a trap.

In any other situation, the new tile is safe.

Then, starting with the row `..^^.`, you can determine the next row by applying those rules to each new tile:

- The leftmost character on the next row considers the left (nonexistent, so we assume "safe"), center (the first `.`, which means "safe"), and right (the second `.`, also "safe") tiles on the previous row. Because all of the trap rules require a trap in at least one of the previous three tiles, the first tile on this new row is also safe, `.`.
- The second character on the next row considers its left (`.`), center (`.`), and right (`^`) tiles from the previous row. This matches the fourth rule: only the right tile is a trap. Therefore, the next tile in this new row is a trap, `^`.
- The third character considers `.^^`, which matches the second trap rule: its center and right tiles are traps, but its left tile is not. Therefore, this tile is also a trap, `^`.
- The last two characters in this new row match the first and third rules, respectively, and so they are both also traps, `^`.

After these steps, we now know the next row of tiles in the room: `.^^^^`. Then, we continue on to the next row, using the same rules, and get `^^..^`. After determining two new rows, our map looks like this:

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

Here's a larger example with ten tiles per row and ten rows:

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

In ten rows, this larger example has `38` safe tiles.

Starting with the map in your puzzle input, in a total of `40` rows (including the starting row), *how many safe tiles* are there?

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

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

--- Day 19: An Elephant Named Joseph ---

The Elves contact you over a highly secure emergency channel. Back at the North Pole, the Elves are busy misunderstanding White Elephant parties.

Each Elf brings a present. They all sit in a circle, numbered starting with position `1`. Then, starting with the first Elf, they take turns stealing all the presents from the Elf to their left. An Elf with no presents is removed from the circle and does not take turns.

For example, with five Elves (numbered `1` to `5`):

```
  1
5   2
 4 3
```

- Elf `1` takes Elf `2`'s present.
- Elf `2` has no presents and is skipped.
- Elf `3` takes Elf `4`'s present.
- Elf `4` has no presents and is also skipped.
- Elf `5` takes Elf `1`'s two presents.
- Neither Elf `1` nor Elf `2` have any presents, so both are skipped.
- Elf `3` takes Elf `5`'s three presents.

So, with *five* Elves, the Elf that sits starting in position `3` gets all the presents.

With the number of Elves given in your puzzle input, *which Elf gets all the presents?*

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

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

--- Day 20: Firewall Rules ---

You'd like to set up a small hidden computer here so you can use it to get back into the network later. However, the corporate firewall only allows communication with certain external IP addresses.

You've retrieved the list of blocked IPs from the firewall, but the list seems to be messy and poorly maintained, and it's not clear which IPs are allowed. Also, rather than being written in dot-decimal notation, they are written as plain 32-bit integers, which can have any value from `0` through `4294967295`, inclusive.

For example, suppose only the values `0` through `9` were valid, and that you retrieved the following blacklist:

```
5-8
0-2
4-7
```

The blacklist specifies ranges of IPs (inclusive of both the start and end value) that are *not* allowed. Then, the only IPs that this firewall allows are `3` and `9`, since those are the only numbers not in any range.

Given the list of blocked IPs you retrieved from the firewall (your puzzle input), *what is the lowest-valued IP* that is not blocked?

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

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

--- Day 21: Scrambled Letters and Hash ---

The computer system you're breaking into uses a weird scrambling function to store its passwords. It shouldn't be much trouble to create your own scrambled password so you can add it to the system; you just have to implement the scrambler.

The scrambling function is a series of operations (the exact list is provided in your puzzle input). Starting with the password to be scrambled, apply each operation in succession to the string. The individual operations behave as follows:

- `swap position X with position Y` means that the letters at indexes `X` and `Y` (counting from `0`) should be *swapped*.
- `swap letter X with letter Y` means that the letters `X` and `Y` should be *swapped* (regardless of where they appear in the string).
- `rotate left/right X steps` means that the whole string should be *rotated*; for example, one right rotation would turn `abcd` into `dabc`.
- `rotate based on position of letter X` means that the whole string should be *rotated to the right* based on the *index* of letter `X` (counting from `0`) as determined *before* this instruction does any rotations. Once the index is determined, rotate the string to the right one time, plus a number

of times equal to that index, plus one additional time if the index was at least `4`.
- `reverse positions X through Y` means that the span of letters at indexes `X` through `Y` (including the letters at `X` and `Y`) should be *reversed in order*.
- `move position X to position Y` means that the letter which is at index `X` should be *removed* from the string, then *inserted* such that it ends up at index `Y`.

For example, suppose you start with `abcde` and perform the following operations:

- `swap position 4 with position 0` swaps the first and last letters, producing the input for the next step, `ebcda`.
- `swap letter d with letter b` swaps the positions of `d` and `b`: `edcba`.
- `reverse positions 0 through 4` causes the entire string to be reversed, producing `abcde`.
- `rotate left 1 step` shifts all letters left one position, causing the first letter to wrap to the end of the string: `bcdea`.
- `move position 1 to position 4` removes the letter at position `1` (`c`), then inserts it at position `4` (the end of the string): `bdeac`.
- `move position 3 to position 0` removes the letter at position `3` (`a`), then inserts it at position `0` (the front of the string): `abdec`.
- `rotate based on position of letter b` finds the index of letter `b` (`1`), then rotates the string right once plus a number of times equal to that index (`2`): `ecabd`.
- `rotate based on position of letter d` finds the index of letter `d` (`4`), then rotates the string right once, plus a number of times equal to that index, plus an additional time because the index was at least `4`, for a total of `6` right rotations: `decab`.

After these steps, the resulting scrambled password is `decab`.

Now, you just need to generate a new scrambled password and you can access the system. Given the list of scrambling operations in your puzzle input, *what is the result of scrambling* ***`abcdefgh`***?

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

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

--- Day 22: Grid Computing ---

You gain access to a massive storage cluster arranged in a grid; each storage node is only connected to the four nodes directly adjacent to it (three if the node is on an edge, two if it's in a corner).

You can directly access data *only* on node `/dev/grid/node-x0-y0`, but you can perform some limited actions on the other nodes:

- You can get the disk usage of all nodes (via `df`). The result of doing this is in your puzzle input.
- You can instruct a node to *move* (not copy) *all* of its data to an adjacent node (if the destination node has enough space to receive the data). The sending node is left empty after this operation.

Nodes are named by their position: the node named `node-x10-y10` is adjacent to nodes `node-x9-y10`, `node-x11-y10`, `node-x10-y9`, and `node-x10-y11`.

Before you begin, you need to understand the arrangement of data on these nodes. Even though you can only move data between directly connected nodes, you're going to need to rearrange a lot of the data to get access to the data you need. Therefore, you need to work out how you might be able to shift data around.

To do this, you'd like to count the number of *viable pairs* of nodes. A viable pair is any two nodes (A,B), *regardless of whether they are directly connected*, such that:

- Node A is *not* empty (its `Used` is not zero).
- Nodes A and B are *not the same* node.
- The data on node A (its `Used`) *would fit* on node B (its `Avail`).

How many viable pairs of nodes are there?

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

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

--- Day 23: Safe Cracking ---

This is one of the top floors of the nicest tower in EBHQ. The Easter Bunny's private office is here, complete with a safe hidden behind a painting, and who *wouldn't* hide a star in a safe behind a painting?

The safe has a digital screen and keypad for code entry. A sticky note attached to the safe has a password hint on it: "eggs". The painting is of a large rabbit coloring some eggs. You see `7`.

When you go to type the code, though, nothing appears on the display; instead, the keypad comes apart in your hands, apparently having been smashed. Behind it is some kind of socket - one that matches a connector in your prototype computer! You pull apart the smashed keypad and extract the logic circuit, plug it into your computer, and plug your computer into the safe.

Now, you just need to figure out what output the keypad would have sent to the safe. You extract the assembunny code from the logic chip (your puzzle input).

The code looks like it uses *almost* the same architecture and instruction set that the monorail computer used! You should be able to *use the same assembunny interpreter* for this as you did there, but with one new instruction:

`tgl x` *toggles* the instruction `x` away (pointing at instructions like `jnz` does: positive means forward; negative means backward):

- For *one-argument* instructions, `inc` becomes `dec`, and all other one-argument instructions become `inc`.
- For *two-argument* instructions, `jnz` becomes `cpy`, and all other two-instructions become `jnz`.
- The arguments of a toggled instruction are *not affected*.
- If an attempt is made to toggle an instruction outside the program, *nothing happens*.
- If toggling produces an *invalid instruction* (like `cpy 1 2`) and an attempt is later made to execute that instruction, *skip it instead*.
- If `tgl` toggles *itself* (for example, if `a` is `0`, `tgl a` would target itself and become `inc a`), the resulting instruction is not executed until the next time it is reached.

For example, given this program:

```
cpy 2 a
tgl a
tgl a
tgl a
cpy 1 a
dec a
dec a
```

- `cpy 2 a` initializes register `a` to `2`.
- The first `tgl a` toggles an instruction `a` (`2`) away from it, which changes the third `tgl a` into `inc a`.
- The second `tgl a` also modifies an instruction `2` away from it, which changes the `cpy 1 a` into `jnz 1 a`.
- The fourth line, which is now `inc a`, increments `a` to `3`.
- Finally, the fifth line, which is now `jnz 1 a`, jumps `a` (`3`) instructions ahead, skipping the `dec a` instructions.

In this example, the final value in register `a` is `3`.

The rest of the electronics seem to place the keypad entry (the number of eggs, `7`) in register `a`, run the code, and then send the value left in register `a` to the safe.

What value should be sent to the safe?

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

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

--- Day 24: Air Duct Spelunking ---

You've finally met your match; the doors that provide access to the roof are locked tight, and all of the controls and related electronics are inaccessible. You simply can't reach them.

The robot that cleans the air ducts, however, *can*.

It's not a very fast little robot, but you reconfigure it to be able to interface with some of the exposed wires that have been routed through the HVAC system. If you can direct it to each of those locations, you should be able to bypass the security controls.

You extract the duct layout for this area from some blueprints you acquired and create a map with the relevant locations marked (your puzzle input). `0` is your current location, from which the cleaning robot embarks; the other numbers are (in *no particular order*) the locations the robot needs to visit at least once each. Walls are marked as `#`, and open passages are marked as `.`. Numbers behave like open passages.

For example, suppose you have a map like the following:

```
###########
#0.1.....2#
#.#######.#
#4.......3#
###########
```

To reach all of the points of interest as quickly as possible, you would have the robot take the following path:

- `0` to `4` (`2` steps)
- `4` to `1` (`4` steps; it can't move diagonally)
- `1` to `2` (`6` steps)
- `2` to `3` (`2` steps)

Since the robot isn't very fast, you need to find it the *shortest route*. This path is the fewest steps (in the above example, a total of `14`) required to start at `0` and then visit every other location at least once.

Given your actual map, and starting from location `0`, what is the *fewest number of steps* required to visit every non-`0` number marked on the map at least once?

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

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

--- Day 25: Clock Signal ---

You open the door and find yourself on the roof. The city sprawls away from you for miles and miles.

There's not much time now - it's already Christmas, but you're nowhere near the North Pole, much too far to deliver these stars to the sleigh in time.

However, maybe the *huge antenna* up here can offer a solution. After all, the sleigh doesn't need the stars, exactly; it needs the timing data they provide, and you happen to have a massive signal generator right here.

You connect the stars you have to your prototype computer, connect that to the antenna, and begin the transmission.

Nothing happens.

You call the service number printed on the side of the antenna and quickly explain the situation. "I'm not sure what kind of equipment you have connected over there," he says, "but you need a clock signal." You try to explain that this is a signal for a clock.

"No, no, a clock signal - timing information so the antenna computer knows how to read the data you're sending it. An endless, alternating pattern of `0`, `1`, `0`, `1`, `0`, `1`, `0`, `1`, `0`, `1`...." He trails off.

You ask if the antenna can handle a clock signal at the frequency you would need to use for the data from the stars. "There's *no way* it can! The only antenna we've installed capable of *that* is on top of a top-secret Easter Bunny installation, and you're *definitely* not-" You hang up the phone.

You've extracted the antenna's clock signal generation assembunny code (your puzzle input); it looks mostly compatible with code you worked on just recently.

This antenna code, being a signal generator, uses one extra instruction:

- `out x` *transmits* `x` (either an integer or the *value* of a register) as the next value for the clock signal.

The code takes a value (via register `a`) that describes the signal to generate, but you're not sure how it's used. You'll have to find the input to produce the right signal through experimentation.

What is the lowest positive integer that can be used to initialize register `a` and cause the code to output a clock signal of `0`, `1`, `0`, `1`... repeating forever?

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

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