--- Day 1: Report Repair ---

After saving Christmas five years in a row, you've decided to take a vacation at a nice resort on a tropical island. Surely, Christmas will go on without you.

The tropical island has its own currency and is entirely cash-only. The gold coins used there have a little picture of a starfish; the locals just call them *stars*. None of the currency exchanges seem to have heard of them, but somehow, you'll need to find fifty of these coins by the time you arrive so you can pay the deposit on your room.

To save your vacation, you need to get all *fifty stars* by December 25th.

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

Before you leave, the Elves in accounting just need you to fix your *expense report* (your puzzle input); apparently, something isn't quite adding up.

Specifically, they need you to *find the two entries that sum to `2020`* and then multiply those two numbers together.

For example, suppose your expense report contained the following:

```
1721
979
366
299
675
1456
```

In this list, the two entries that sum to `2020` are `1721` and `299`. Multiplying them together produces `1721 * 299 = 514579`, so the correct answer is `514579`.

Of course, your expense report is much larger. *Find the two entries that sum to `2020`; what do you get if you multiply them together?*

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

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

--- Day 2: Password Philosophy ---

Your flight departs in a few days from the coastal airport; the easiest way down to the coast from here is via toboggan.

The shopkeeper at the North Pole Toboggan Rental Shop is having a bad day. "Something's wrong with our computers; we can't log in!" You ask if you can take a look.

Their password database seems to be a little corrupted: some of the passwords wouldn't have been allowed by the Official Toboggan Corporate Policy that was in effect when they were chosen.

To try to debug the problem, they have created a list (your puzzle input) of *passwords* (according to the corrupted database) and *the corporate policy when that password was set*.

For example, suppose you have the following list:

```
1-3 a: abcde
1-3 b: cdefg
2-9 c: ccccccccc
```

Each line gives the password policy and then the password. The password policy indicates the lowest and highest number of times a given letter must appear for the password to be valid. For example, `1-3 a` means that the password must contain `a` at least `1` time and at most `3` times.

In the above example, `2` passwords are valid. The middle password, `cdefg`, is not; it contains no instances of `b`, but needs at least `1`. The first and third passwords are valid: they contain one `a` or nine `c`, both within the limits of their respective policies.

How many passwords are valid according to their policies?

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

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

--- Day 3: Toboggan Trajectory ---

With the toboggan login problems resolved, you set off toward the airport. While travel by toboggan might be easy, it's certainly not safe: there's very minimal steering and the area is covered in trees. You'll need to see which angles will take you near the fewest trees.

Due to the local geology, trees in this area only grow on exact integer coordinates in a grid. You make a map (your puzzle input) of the open squares (`.`) and trees (`#`) you can see. For example:

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

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

These aren't the only trees, though; due to something you read about once involving arboreal genetics and biome stability, the same pattern repeats to the right many times:

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

You start on the open square (`.`) in the top-left corner and need to reach the bottom (below the bottom-most row on your map).

The toboggan can only follow a few specific slopes (you opted for a cheaper model that prefers rational numbers); start by *counting all the trees* you would encounter for the slope *right 3, down 1*:

From your starting position at the top-left, check the position that is right 3 and down 1. Then, check the position that is right 3 and down 1 from there, and so on until you go past the bottom of the map.

The locations you'd check in the above example are marked here with `O` where there was an open square and `X` where there was a tree:

```
..##.........##.........##.........##.........##.........##.......  --->
#..O#...#..#...#...#..#...#...#..#...#...#..#...#...#..#...#...#..
.#....X..#..#....#..#..#....#..#..#....#..#..#....#..#..#....#..#.
..#.#...#O#..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#
.#...##..#..X...##..#..#...##..#..#...##..#..#...##..#..#...##..#.
..#.##.......#.X#.......#.##.......#.##.......#.##.......#.##.....  --->
.#.#.#....#.#.#.#.O..#.#.#.#....#.#.#.#....#.#.#.#....#.#.#.#....#
.#........#.#........X.#........#.#........#.#........#.#........#
#.##...#...#.##...#...#.X#...#...#.##...#...#.##...#...#.##...#...
#...##....##...##....##...#X....##...##....##...##....##...##....#
.#..#...#.#.#..#...#.#.#..#...X.#.#..#...#.#.#..#...#.#.#..#...#.#  --->
```

In this example, traversing the map using this slope would cause you to encounter 7 trees.

Starting at the top-left corner of your map and following a slope of right 3 and down 1, *how many trees would you encounter?*

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

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

--- Day 4: Passport Processing ---

You arrive at the airport only to realize that you grabbed your North Pole Credentials instead of your passport. While these documents are extremely similar, North Pole Credentials aren't issued by a country and therefore aren't actually valid documentation for travel in most of the world.

It seems like you're not the only one having problems, though; a very long line has formed for the automatic passport scanners, and the delay could upset your travel itinerary.

Due to some questionable network security, you realize you might be able to solve both of these problems at the same time.

The automatic passport scanners are slow because they're having trouble *detecting which passports have all required fields*. The expected fields are as follows:

- `byr` (Birth Year)
- `iyr` (Issue Year)
- `eyr` (Expiration Year)
- `hgt` (Height)
- `hcl` (Hair Color)
- `ecl` (Eye Color)
- `pid` (Passport ID)
- `cid` (Country ID)

Passport data is validated in batch files (your puzzle input). Each passport is represented as a sequence of `key:value` pairs separated by spaces or newlines. Passports are separated by blank lines.

Here is an example batch file containing four passports:

```
ecl:gry pid:860033327 eyr:2020 hcl:#fffffd
byr:1937 iyr:2017 cid:147 hgt:183cm

iyr:2013 ecl:amb cid:350 eyr:2023 pid:028048884
hcl:#cfa07d byr:1929

hcl:#ae17e1 iyr:2013
eyr:2024
ecl:brn pid:760753108 byr:1931
hgt:179cm

hcl:#cfa07d eyr:2025 pid:166559648
iyr:2011 ecl:brn hgt:59in
```

The first passport is *valid* - all eight fields are present. The second passport is *invalid* - it is missing `hgt` (the Height field).

The third passport is interesting; the *only missing field* is `cid`, so it looks like data from North Pole Credentials, not a passport at all! Surely, nobody would mind if you made the system temporarily ignore missing `cid` fields. Treat this "passport" as *valid*.

The fourth passport is missing two fields, `cid` and `byr`. Missing `cid` is fine, but missing any other field is not, so this passport is *invalid*.

According to the above rules, your improved system would report `2` valid passports.

Count the number of *valid* passports - those that have all required fields. Treat `cid` as optional. *In your batch file, how many passports are valid?*

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

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

--- Day 5: Binary Boarding ---

You board your plane only to discover a new problem: you dropped your boarding pass! You aren't sure which seat is yours, and all of the flight attendants are busy with the flood of people that suddenly made it through passport control.

You write a quick program to use your phone's camera to scan all of the nearby boarding passes (your puzzle input); perhaps you can find your seat through process of elimination.

Instead of zones or groups, this airline uses *binary space partitioning* to seat people. A seat might be specified like `FBFBBFFRLR`, where `F` means "front", `B` means "back", `L` means "left", and `R` means "right".

The first `7` characters will either be `F` or `B`; these specify exactly one of the *128 rows* on the plane (numbered `0` through `127`). Each letter tells you which half of a region the given seat is in. Start with the whole list of rows; the first letter indicates whether the seat is in the *front* (`0` through `63`) or the *back* (`64` through `127`). The next letter indicates which half of that region the seat is in, and so on until you're left with exactly one row.

For example, consider just the first seven characters of `FBFBBFFRLR`:

- Start by considering the whole range, rows `0` through `127`.
- `F` means to take the *lower half*, keeping rows `0` through `63`.
- `B` means to take the *upper half*, keeping rows `32` through `63`.
- `F` means to take the *lower half*, keeping rows `32` through `47`.
- `B` means to take the *upper half*, keeping rows `40` through `47`.
- `B` keeps rows `44` through `47`.
- `F` keeps rows `44` through `45`.
- The final `F` keeps the lower of the two, *row 44*.

The last three characters will be either `L` or `R`; these specify exactly one of the *8 columns* of seats on the plane (numbered `0` through `7`). The same process as above proceeds again, this time with only three steps. `L` means to keep the *lower half*, while `R` means to keep the *upper half*.

For example, consider just the last 3 characters of `FBFBBFFRLR`:

- Start by considering the whole range, columns `0` through `7`.
- `R` means to take the *upper half*, keeping columns `4` through `7`.
- `L` means to take the *lower half*, keeping columns `4` through `5`.
- The final `R` keeps the upper of the two, *column `5`*.

So, decoding `FBFBBFFRLR` reveals that it is the seat at *row `44`, column `5`*.

Every seat also has a unique *seat ID*: multiply the row by 8, then add the column. In this example, the seat has ID `44 * 8 + 5 = 357`.

Here are some other boarding passes:

- `BFFFBBFRRR`: row `70`, column `7`, seat ID `567`.
- `FFFBBBFRRR`: row `14`, column `7`, seat ID `119`.
- `BBFFBBFRLL`: row `102`, column `4`, seat ID `820`.

As a sanity check, look through your list of boarding passes. *What is the highest seat ID on a boarding pass?*

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

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

--- Day 6: Custom Customs ---

As your flight approaches the regional airport where you'll switch to a much larger plane, customs declaration forms are distributed to the passengers.

The form asks a series of 26 yes-or-no questions marked `a` through `z`. All you need to do is identify the questions for which *anyone in your group* answers "yes". Since your group is just you, this doesn't take very long.

However, the person sitting next to you seems to be experiencing a language barrier and asks if you can help. For each of the people in their group, you write down the questions for which they answer "yes", one per line. For example:

```
abcx
abcy
abcz
```

In this group, there are *`6`* questions to which anyone answered "yes": `a`, `b`, `c`, `x`, `y`, and `z`. (Duplicate answers to the same question don't count extra; each question counts at most once.)

Another group asks for your help, then another, and eventually you've collected answers from every group on the plane (your puzzle input). Each group's answers are separated by a blank line, and within each group, each person's answers are on a single line. For example:

```
abc

a
b
c

ab
ac

a
a
a
a

b
```

This list represents answers from five groups:

- The first group contains one person who answered "yes" to *3* questions: `a`, `b`, and `c`.
- The second group contains three people; combined, they answered "yes" to *3* questions: `a`, `b`, and `c`.
- The third group contains two people; combined, they answered "yes" to *3* questions: `a`, `b`, and `c`.
- The fourth group contains four people; combined, they answered "yes" to only *1* question, `a`.
- The last group contains one person who answered "yes" to only *1* question, `b`.

In this example, the sum of these counts is `3 + 3 + 3 + 1 + 1` = *11*.

For each group, count the number of questions to which anyone answered "yes". *What is the sum of those counts?*

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

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

--- Day 7: Handy Haversacks ---

You land at the regional airport in time for your next flight. In fact, it looks like you'll even have time to grab some food: all flights are currently delayed due to *issues in luggage processing.*

Due to recent aviation regulations, many rules (your puzzle input) are being enforced about bags and their contents; bags must be color-coded and must contain specific quantities of other color-coded bags. Apparently, nobody responsible for these regulations considered how long they would take to enforce!

For example, consider the following rules:

```
light red bags contain 1 bright white bag, 2 muted yellow bags.
dark orange bags contain 3 bright white bags, 4 muted yellow bags.
bright white bags contain 1 shiny gold bag.
muted yellow bags contain 2 shiny gold bags, 9 faded blue bags.
shiny gold bags contain 1 dark olive bag, 2 vibrant plum bags.
dark olive bags contain 3 faded blue bags, 4 dotted black bags.
vibrant plum bags contain 5 faded blue bags, 6 dotted black bags.
faded blue bags contain no other bags.
dotted black bags contain no other bags.
```

These rules specify the required contents for 9 bag types. In this example, every `faded blue` bag is empty, every `vibrant plum` bag contains 11 bags (5 `faded blue` and 6 `dotted black`), and so on.

You have a `shiny gold` bag. If you wanted to carry it in at least one other bag, how many different bag colors would be valid for the outermost bag? (In other words: how many colors can, eventually, contain at least one `shiny gold` bag?)

In the above rules, the following options would be available to you:

- A `bright white` bag, which can hold your `shiny gold` bag directly.
- A `muted yellow` bag, which can hold your `shiny gold` bag directly, plus some other bags.
- A `dark orange` bag, which can hold `bright white` and `muted yellow` bags, either of which could then hold your `shiny gold` bag.
- A `light red` bag, which can hold `bright white` and `muted yellow` bags, either of which could then hold your `shiny gold` bag.

So, in this example, the number of bag colors that can eventually contain at least one `shiny gold` bag is `4`.

How many bag colors can eventually contain at least one `shiny gold` bag? (The list of rules is quite long; make sure you get all of it.)

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

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

--- Day 8: Handheld Halting ---

Your flight to the major airline hub reaches cruising altitude without incident. While you consider checking the in-flight menu for one of those drinks that come with a little umbrella, you are interrupted by the kid sitting next to you.

Their handheld game console won't turn on! They ask if you can take a look.

You narrow the problem down to a strange *infinite loop* in the boot code (your puzzle input) of the device. You should be able to fix it, but first you need to be able to run the code in isolation.

The boot code is represented as a text file with one *instruction* per line of text. Each instruction consists of an *operation* (`acc`, `jmp`, or `nop`) and an *argument* (a signed number like `+4` or `-20`).

- `acc` increases or decreases a single global value called the *accumulator* by the value given in the argument. For example, `acc +7` would increase the accumulator by 7. The accumulator starts at `0`. After an `acc` instruction, the instruction immediately below it is executed next.
- `jmp` *jumps* to a new instruction relative to itself. The next instruction to execute is found using the argument as an *offset* from the `jmp` instruction; for example, `jmp +2` would skip the next instruction, `jmp +1` would continue to the instruction immediately below it, and `jmp -20` would cause the instruction 20 lines above to be executed next.
- `nop` stands for *No OPeration* - it does nothing. The instruction immediately below it is executed next.

For example, consider the following program:

```
nop +0
acc +1
jmp +4
acc +3
jmp -3
acc -99
acc +1
jmp -4
acc +6
```

These instructions are visited in this order:

```
nop +0  | 1
acc +1  | 2, 8(!)
jmp +4  | 3
acc +3  | 6
jmp -3  | 7
acc -99 |
acc +1  | 4
jmp -4  | 5
acc +6  |
```

First, the `nop +0` does nothing. Then, the accumulator is increased from 0 to 1 (`acc +1`) and `jmp +4` sets the next instruction to the other `acc +1` near the bottom. After it increases the accumulator from 1 to 2, `jmp -4` executes, setting the next instruction to the only `acc +3`. It sets the accumulator to 5, and `jmp -3` causes the program to continue back at the first `acc +1`.

This is an *infinite loop*: with this sequence of jumps, the program will run forever. The moment the program tries to run any instruction a second time, you know it will never terminate.

Immediately *before* the program would run an instruction a second time, the value in the accumulator is *`5`*.

Run your copy of the boot code. Immediately before any instruction is executed a second time, *what value is in the accumulator?*

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

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

--- Day 9: Encoding Error ---

With your neighbor happily enjoying their video game, you turn your attention to an open data port on the little screen in the seat in front of you.

Though the port is non-standard, you manage to connect it to your computer through the clever use of several paperclips. Upon connection, the port outputs a series of numbers (your puzzle input).

The data appears to be encrypted with the eXchange-Masking Addition System (XMAS) which, conveniently for you, is an old cypher with an important weakness.

XMAS starts by transmitting a *preamble* of 25 numbers. After that, each number you receive should be the sum of any two of the 25 immediately previous numbers. The two numbers will have different values, and there might be more than one such pair.

For example, suppose your preamble consists of the numbers `1` through `25` in a random order. To be valid, the next number must be the sum of two of those numbers:

- `26` would be a *valid* next number, as it could be `1` plus `25` (or many other pairs, like `2` and `24`).
- `49` would be a *valid* next number, as it is the sum of `24` and `25`.
- `100` would *not* be valid; no two of the previous 25 numbers sum to `100`.
- `50` would also *not* be valid; although `25` appears in the previous 25 numbers, the two numbers in the pair must be different.

Suppose the 26th number is `45`, and the first number (no longer an option, as it is more than 25 numbers ago) was `20`. Now, for the next number to be valid, there needs to be some pair of numbers among `1-19`, `21-25`, or `45` that add up to it:

- `26` would still be a *valid* next number, as `1` and `25` are still within the previous 25 numbers.
- `65` would *not* be valid, as no two of the available numbers sum to it.

- `64` and `66` would both be *valid*, as they are the result of `19+45` and `21+45` respectively.

Here is a larger example which only considers the previous *5* numbers (and has a preamble of length 5):

```
35
20
15
25
47
40
62
55
65
95
102
117
150
182
127
219
299
277
309
576
```

In this example, after the 5-number preamble, almost every number is the sum of two of the previous 5 numbers; the only number that does not follow this rule is *`127`*.

The first step of attacking the weakness in the XMAS data is to find the first number in the list (after the preamble) which is *not* the sum of two of the 25 numbers before it. *What is the first number that does not have this property?*

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

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

--- Day 10: Adapter Array ---

Patched into the aircraft's data port, you discover weather forecasts of a massive tropical storm. Before you can figure out whether it will impact your vacation plans, however, your device suddenly turns off!

Its battery is dead.

You'll need to plug it in. There's only one problem: the charging outlet near your seat produces the wrong number of *jolts*. Always prepared, you make a list of all of the joltage adapters in your bag.

Each of your joltage adapters is rated for a specific *output joltage* (your puzzle input). Any given adapter can take an input `1`, `2`, or `3` jolts *lower* than its rating and still produce its rated output joltage.

In addition, your device has a built-in joltage adapter rated for *`3` jolts higher* than the highest-rated adapter in your bag. (If your adapter list were `3`, `9`, and `6`, your device's built-in adapter would be rated for `12` jolts.)

Treat the charging outlet near your seat as having an effective joltage rating of `0`.

Since you have some time to kill, you might as well test all of your adapters. Wouldn't want to get to your resort and realize you can't even charge your device!

If you *use every adapter in your bag* at once, what is the distribution of joltage differences between the charging outlet, the adapters, and your device?

For example, suppose that in your bag, you have adapters with the following joltage ratings:

```
16
10
15
5
1
11
7
19
6
12
4
```

With these adapters, your device's built-in joltage adapter would be rated for `19 + 3 = 22` jolts, 3 higher than the highest-rated adapter.

Because adapters can only connect to a source 1-3 jolts lower than its rating, in order to use every adapter, you'd need to choose them like this:

- The charging outlet has an effective rating of `0` jolts, so the only adapters that could connect to it directly would need to have a joltage rating of `1`, `2`, or `3` jolts. Of these, only one you have is an adapter rated `1` jolt (difference of *`1`*).
- From your `1`-jolt rated adapter, the only choice is your `4`-jolt rated adapter (difference of *`3`*).
- From the `4`-jolt rated adapter, the adapters rated `5`, `6`, or `7` are valid choices. However, in order to not skip any adapters, you have to pick the adapter rated `5` jolts (difference of *`1`*).
- Similarly, the next choices would need to be the adapter rated `6` and then the adapter rated `7` (with difference of *`1`* and *`1`*).

- The only adapter that works with the `7`-jolt rated adapter is the one rated `10` jolts (difference of *`3`*).
- From `10`, the choices are `11` or `12`; choose `11` (difference of *`1`*) and then `12` (difference of *`1`*).
- After `12`, only valid adapter has a rating of `15` (difference of *`3`*), then `16` (difference of *`1`*), then `19` (difference of *`3`*).
- Finally, your device's built-in adapter is always 3 higher than the highest adapter, so its rating is `22` jolts (always a difference of *`3`*).

In this example, when using every adapter, there are *`7`* differences of 1 jolt and *`5`* differences of 3 jolts.

Here is a larger example:

```
28
33
18
42
31
14
46
20
48
47
24
23
49
45
19
38
39
11
1
32
25
35
8
17
7
9
4
2
34
10
3
```

In this larger example, in a chain that uses all of the adapters, there are *`22`* differences of 1 jolt and *`10`* differences of 3 jolts.

Find a chain that uses all of your adapters to connect the charging outlet to your device's built-in adapter and count the joltage differences between the charging outlet, the adapters, and your device. *What is the number of 1-jolt differences multiplied by the number of 3-jolt differences?*

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

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

--- Day 11: Seating System ---

Your plane lands with plenty of time to spare. The final leg of your journey is a ferry that goes directly to the tropical island where you can finally start your vacation. As you reach the waiting area to board the ferry, you realize you're so early, nobody else has even arrived yet!

By modeling the process people use to choose (or abandon) their seat in the waiting area, you're pretty sure you can predict the best place to sit. You make a quick map of the seat layout (your puzzle input).

The seat layout fits neatly on a grid. Each position is either floor (`.`), an empty seat (`L`), or an occupied seat (`#`). For example, the initial seat layout might look like this:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.LL.LL
L.LL.LL.LL
L.LLLLL.LL
..L.L.....
LLLLLLLLLL
L.LLLLLL.L
L.LLLLL.LL
```

Now, you just need to model the people who will be arriving shortly. Fortunately, people are entirely predictable and always follow a simple set of rules. All decisions are based on the *number of occupied seats* adjacent to a given seat (one of the eight positions immediately up, down, left, right, or diagonal from the seat). The following rules are applied to every seat simultaneously:

- If a seat is *empty* (`L`) and there are *no* occupied seats adjacent to it, the seat becomes *occupied*.
- If a seat is *occupied* (`#`) and *four or more* seats adjacent to it are also occupied, the seat becomes *empty*.
- Otherwise, the seat's state does not change.

Floor (`.`) never changes; seats don't move, and nobody sits on the floor.

After one round of these rules, every seat in the example layout becomes occupied:

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

After a second round, the seats with four or more occupied adjacent seats become empty again:

```
#.LL.L#.##
#LLLLLL.L#
L.L.L..L..
#LLL.LL.L#
#.LL.LL.LL
#.LLLL#.##
..L.L.....
#LLLLLLLL#
#.LLLLLL.L
#.#LLLL.##
```

This process continues for three more rounds:

```
#.##.L#.##
#L###LL.L#
L.#.#..#..
#L##.##.L#
#.##.LL.LL
#.###L#.##
..#.#.....
#L######L#
#.LL###L.L
#.#L###.##
```

```
#.#L.L#.##
#LLL#LL.L#
L.L.L..#..
#LLL.##.L#
#.LL.LL.LL
#.LL#L#.##
..L.L.....
#L#LLLL#L#
#.LLLLLL.L
#.#L#L#.##
```

```
#.#L.L#.##
#LLL#LL.L#
L.#.L..#..
#L##.##.L#
#.#L.LL.LL
#.#L#L#.##
..L.L.....
#L#L##L#L#
#.LLLLLL.L
#.#L#L#.##
```

At this point, something interesting happens: the chaos stabilizes and further applications of these rules cause no seats to change state! Once people stop moving around, you count *37* occupied seats.

Simulate your seating area by applying the seating rules repeatedly until no seats change state. *How many seats end up occupied?*

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

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

--- Day 12: Rain Risk ---

Your ferry made decent progress toward the island, but the storm came in faster than anyone expected. The ferry needs to take *evasive actions*!

Unfortunately, the ship's navigation computer seems to be malfunctioning; rather than giving a route directly to safety, it produced extremely circuitous instructions. When the captain uses the PA system to ask if anyone can help, you quickly volunteer.

The navigation instructions (your puzzle input) consists of a sequence of single-character *actions* paired with integer input *values*. After staring at them for a few minutes, you work out what they probably mean:

- Action *`N`* means to move *north* by the given value.
- Action *`S`* means to move *south* by the given value.
- Action *`E`* means to move *east* by the given value.
- Action *`W`* means to move *west* by the given value.
- Action *`L`* means to turn *left* the given number of degrees.
- Action *`R`* means to turn *right* the given number of degrees.
- Action *`F`* means to move *forward* by the given value in the direction the ship is currently facing.

The ship starts by facing *east*. Only the `L` and `R` actions change the direction the ship is facing. (That is, if the ship is facing east and the next instruction is `N10`, the ship would move north 10 units, but would still move east if the following action were `F`.)

For example:

```
F10
N3
F7
R90
F11
```

These instructions would be handled as follows:

- `F10` would move the ship 10 units east (because the ship starts by facing east) to *east 10, north 0*.
- `N3` would move the ship 3 units north to *east 10, north 3*.
- `F7` would move the ship another 7 units east (because the ship is still facing east) to *east 17, north 3*.
- `R90` would cause the ship to turn right by 90 degrees and face *south*; it remains at *east 17, north 3*.
- `F11` would move the ship 11 units south to *east 17, south 8*.

At the end of these instructions, the ship's Manhattan distance (sum of the absolute values of its east/west position and its north/south position) from its starting position is `17 + 8` = *25*.

Figure out where the navigation instructions lead. *What is the Manhattan distance between that location and the ship's starting position?*

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

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

--- Day 13: Shuttle Search ---

Your ferry can make it safely to a nearby port, but it won't get much further. When you call to book another ship, you discover that no ships embark from that port to your vacation island. You'll need to get from the port to the nearest airport.

Fortunately, a shuttle bus service is available to bring you from the sea port to the airport! Each bus has an ID number that also indicates *how often the bus leaves for the airport*.

Bus schedules are defined based on a *timestamp* that measures the *number of minutes* since some fixed reference point in the past. At timestamp `0`, every bus simultaneously departed from the sea port. After that, each bus travels to the airport, then various other locations, and finally returns to the sea port to repeat its journey forever.

The time this loop takes a particular bus is also its ID number: the bus with ID `5` departs from the sea port at timestamps `0`, `5`, `10`, `15`, and so on. The bus with ID `11` departs at `0`, `11`, `22`, `33`, and so on. If you are there when the bus departs, you can ride that bus to the airport!

Your notes (your puzzle input) consist of two lines. The first line is your estimate of the *earliest timestamp you could depart on a bus.* The second line lists the bus IDs that are in service according to the shuttle company; entries that show `x` must be out of service, so you decide to ignore them.

To save time once you arrive, your goal is to figure out *the earliest bus you can take to the airport.* (There will be exactly one such bus.)

For example, suppose you have the following notes:

```
939
7,13,x,x,59,x,31,19
```

Here, the earliest timestamp you could depart is `939`, and the bus IDs in service are `7`, `13`, `59`, `31`, and `19`. Near timestamp `939`, these bus IDs depart at the times marked `D`:

```
time   bus 7   bus 13  bus 59  bus 31  bus 19
929      .       .       .       .       .
930      .       .       .       D       .
931      D       .       .       .       D
932      .       .       .       .       .
933      .       .       .       .       .
934      .       .       .       .       .
935      .       .       .       .       .
936      .       D       .       .       .
937      .       .       .       .       .
938      D       .       .       .       .
939      .       .       .       .       .
940      .       .       .       .       .
941      .       .       .       .       .
942      .       .       .       .       .
943      .       .       .       .       .
944      .       .       D       .       .
945      D       .       .       .       .
946      .       .       .       .       .
947      .       .       .       .       .
948      .       .       .       .       .
949      .       D       .       .       .
```

The earliest bus you could take is bus ID `59`. It doesn't depart until timestamp `944`, so you would need to wait `944 - 939 = 5` minutes before it departs. Multiplying the bus ID by the number of minutes you'd need to wait gives ***`295`***.

What is the ID of the earliest bus you can take to the airport multiplied by the number of minutes you'll need to wait for that bus?

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

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

--- Day 14: Docking Data ---

As your ferry approaches the sea port, the captain asks for your help again. The computer system that runs this port isn't compatible with the docking program on the ferry, so the docking parameters aren't being correctly initialized in the docking program's memory.

After a brief inspection, you discover that the sea port's computer system uses a strange bitmask system in its initialization program. Although you don't have the correct decoder chip handy, you can emulate it in software!

The initialization program (your puzzle input) can either update the bitmask or write a value to memory. Values and memory addresses are both 36-bit unsigned integers. For example, ignoring bitmasks for a moment, a line like `mem[8] = 11` would write the value `11` to memory address `8`.

The bitmask is always given as a string of 36 bits, written with the most significant bit (representing `2^35`) on the left and the least significant bit (`2^0`, that is, the `1`s bit) on the right. The current bitmask is applied to values immediately before they are written to memory: a `0` or `1` overwrites the corresponding bit in the value, while an `X` leaves the bit in the value unchanged.

For example, consider the following program:

```
mask = XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
mem[8] = 11
mem[7] = 101
mem[8] = 0
```

This program starts by specifying a bitmask (`mask = ....`). The mask it specifies will overwrite two bits in every written value: the `2`s bit is overwritten with `0`, and the `64`s bit is overwritten with `1`.

The program then attempts to write the value `11` to memory address `8`. By expanding everything out to individual bits, the mask is applied as follows:

```
value:  000000000000000000000000000000001011  (decimal 11)
mask:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
result: 000000000000000000000000000001001001  (decimal 73)
```

So, because of the mask, the value `73` is written to memory address `8` instead. Then, the program tries to write `101` to address `7`:

```
value:  000000000000000000000000000001100101  (decimal 101)
mask:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
result: 000000000000000000000000000001100101  (decimal 101)
```

This time, the mask has no effect, as the bits it overwrote were already the values the mask tried to set. Finally, the program tries to write `0` to address `8`:

```
value:  000000000000000000000000000000000000  (decimal 0)
mask:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
```

```
result: 000000000000000000000000000001000000  (decimal 64)
```

`64` is written to address `8` instead, overwriting the value that was there previously.

To initialize your ferry's docking program, you need the sum of all values left in memory after the initialization program completes. (The entire 36-bit address space begins initialized to the value `0` at every address.) In the above example, only two values in memory are not zero - `101` (at address `7`) and `64` (at address `8`) - producing a sum of *`165`*.

Execute the initialization program. *What is the sum of all values left in memory after it completes?* (Do not truncate the sum to 36 bits.)

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

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

--- Day 15: Rambunctious Recitation ---

You catch the airport shuttle and try to book a new flight to your vacation island. Due to the storm, all direct flights have been cancelled, but a route is available to get around the storm. You take it.

While you wait for your flight, you decide to check in with the Elves back at the North Pole. They're playing a *memory game* and are ever so excited to explain the rules!

In this game, the players take turns saying *numbers*. They begin by taking turns reading from a list of *starting numbers* (your puzzle input). Then, each turn consists of considering the *most recently spoken number*:

- If that was the *first* time the number has been spoken, the current player says *`0`*.
- Otherwise, the number had been spoken before; the current player announces *how many turns apart* the number is from when it was previously spoken.

So, after the starting numbers, each turn results in that player speaking aloud either *`0`* (if the last number is new) or an *age* (if the last number is a repeat).

For example, suppose the starting numbers are `0,3,6`:

- *Turn 1*: The `1`st number spoken is a starting number, *`0`*.
- *Turn 2*: The `2`nd number spoken is a starting number, *`3`*.
- *Turn 3*: The `3`rd number spoken is a starting number, *`6`*.
- *Turn 4*: Now, consider the last number spoken, `6`. Since that was the first time the number had been spoken, the `4`th number spoken is *`0`*.
- *Turn 5*: Next, again consider the last number spoken, `0`. Since it *had* been spoken before, the next number to speak is the difference between the turn number when it was last spoken (the previous turn, `4`) and the

turn number of the time it was most recently spoken before then (turn `1`). Thus, the `5`th number spoken is `4 - 1`, *`3`*.
- *Turn 6*: The last number spoken, `3` had also been spoken before, most recently on turns `5` and `2`. So, the `6`th number spoken is `5 - 2`, *`3`*.
- *Turn 7*: Since `3` was just spoken twice in a row, and the last two turns are `1` turn apart, the `7`th number spoken is *`1`*.
- *Turn 8*: Since `1` is new, the `8`th number spoken is *`0`*.
- *Turn 9*: `0` was last spoken on turns `8` and `4`, so the `9`th number spoken is the difference between them, *`4`*.
- *Turn 10*: `4` is new, so the `10`th number spoken is *`0`*.

(The game ends when the Elves get sick of playing or dinner is ready, whichever comes first.)

Their question for you is: what will be the *`2020`th* number spoken? In the example above, the `2020`th number spoken will be `436`.

Here are a few more examples:

- Given the starting numbers `1,3,2`, the `2020`th number spoken is `1`.
- Given the starting numbers `2,1,3`, the `2020`th number spoken is `10`.
- Given the starting numbers `1,2,3`, the `2020`th number spoken is `27`.
- Given the starting numbers `2,3,1`, the `2020`th number spoken is `78`.
- Given the starting numbers `3,2,1`, the `2020`th number spoken is `438`.
- Given the starting numbers `3,1,2`, the `2020`th number spoken is `1836`.

Given your starting numbers, *what will be the `2020`th number spoken?*

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

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

--- Day 16: Ticket Translation ---

As you're walking to yet another connecting flight, you realize that one of the legs of your re-routed trip coming up is on a high-speed train. However, the train ticket you were given is in a language you don't understand. You should probably figure out what it says before you get to the train station after the next flight.

Unfortunately, you can't actually *read* the words on the ticket. You can, however, read the numbers, and so you figure out *the fields these tickets must have* and *the valid ranges* for values in those fields.

You collect the *rules for ticket fields*, the *numbers on your ticket*, and the *numbers on other nearby tickets* for the same train service (via the airport security cameras) together into a single document you can reference (your puzzle input).

The *rules for ticket fields* specify a list of fields that exist *somewhere* on the ticket and the *valid ranges of values* for each field. For example, a rule like `class: 1-3 or 5-7` means that one of the fields in every ticket is named `class` and can be any value in the ranges `1-3` or `5-7` (inclusive, such that `3` and `5` are both valid in this field, but `4` is not).

Each ticket is represented by a single line of comma-separated values. The values are the numbers on the ticket in the order they appear; every ticket has the same format. For example, consider this ticket:

```
.--------------------------------------------------------.
| ????: 101    ?????: 102   ??????????: 103     ???: 104 |
|                                                        |
| ??: 301  ??: 302             ???????: 303      ??????? |
| ??: 401  ??: 402           ???? ????: 403    ????????? |
'--------------------------------------------------------'
```

Here, `?` represents text in a language you don't understand. This ticket might be represented as `101,102,103,104,301,302,303,401,402,403`; of course, the actual train tickets you're looking at are *much* more complicated. In any case, you've extracted just the numbers in such a way that the first number is always the same specific field, the second number is always a different specific field, and so on - you just don't know what each position actually means!

Start by determining which tickets are *completely invalid*; these are tickets that contain values which *aren't valid for any field*. Ignore *your ticket* for now.

For example, suppose you have the following notes:

```
class: 1-3 or 5-7
row: 6-11 or 33-44
seat: 13-40 or 45-50

your ticket:
7,1,14

nearby tickets:
7,3,47
40,4,50
55,2,20
38,6,12
```

It doesn't matter which position corresponds to which field; you can identify invalid *nearby tickets* by considering only whether tickets contain *values that are not valid for any field*. In this example, the values on the first *nearby ticket* are all valid for at least one field. This is not true of the other three *nearby tickets*: the values `4`, `55`, and `12` are are not valid for any field. Adding together all of the invalid values produces your *ticket scanning error rate*: `4 + 55 + 12` = *`71`*.

Consider the validity of the *nearby tickets* you scanned. *What is your ticket scanning error rate?*

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

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

--- Day 17: Conway Cubes ---

As your flight slowly drifts through the sky, the Elves at the Mythical Information Bureau at the North Pole contact you. They'd like some help debugging a malfunctioning experimental energy source aboard one of their super-secret imaging satellites.

The experimental energy source is based on cutting-edge technology: a set of Conway Cubes contained in a pocket dimension! When you hear it's having problems, you can't help but agree to take a look.

The pocket dimension contains an infinite 3-dimensional grid. At every integer 3-dimensional coordinate (`x,y,z`), there exists a single cube which is either *active* or *inactive*.

In the initial state of the pocket dimension, almost all cubes start *inactive*. The only exception to this is a small flat region of cubes (your puzzle input); the cubes in this region start in the specified *active* (`#`) or *inactive* (`.`) state.

The energy source then proceeds to boot up by executing six *cycles*.

Each cube only ever considers its *neighbors*: any of the 26 other cubes where any of their coordinates differ by at most `1`. For example, given the cube at `x=1,y=2,z=3`, its neighbors include the cube at `x=2,y=2,z=2`, the cube at `x=0,y=2,z=3`, and so on.

During a cycle, *all* cubes *simultaneously* change their state according to the following rules:

- If a cube is *active* and *exactly `2` or `3`* of its neighbors are also active, the cube remains *active*. Otherwise, the cube becomes *inactive*.
- If a cube is *inactive* but *exactly `3`* of its neighbors are active, the cube becomes *active*. Otherwise, the cube remains *inactive*.

The engineers responsible for this experimental energy source would like you to simulate the pocket dimension and determine what the configuration of cubes should be at the end of the six-cycle boot process.

For example, consider the following initial state:

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

Even though the pocket dimension is 3-dimensional, this initial state represents a small 2-dimensional slice of it. (In particular, this initial state defines a 3x3x1 region of the 3-dimensional space.)

Simulating a few cycles from this initial state produces the following configurations, where the result of each cycle is shown layer-by-layer at each given `z` coordinate (and the frame of view follows the active cells in each cycle):

```
Before any cycles:

z=0
.#.
..#
###


After 1 cycle:

z=-1
#..
..#
.#.

z=0
#.#
.##
.#.

z=1
#..
..#
.#.


After 2 cycles:

z=-2
.....
.....
..#..
.....
.....

z=-1
..#..
.#..#
....#
.#...
.....
```

```
z=0
##...
##...
#....
....#
.###.

z=1
..#..
.#..#
....#
.#...
.....

z=2
.....
.....
..#..
.....
.....


After 3 cycles:

z=-2
.......
.......
..##...
..###..
.......
.......
.......

z=-1
..#....
...#...
#......
.....##
.#...#.
..#.#..
...#...

z=0
...#...
.......
#......
```

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

z=1
..#....
...#...
#......
.....##
.#...#.
..#.#..
...#...

z=2
.......
.......
..##...
..###..
.......
.......
.......
```

After the full six-cycle boot process completes, *`112`* cubes are left in the *active* state.

Starting with your given initial configuration, simulate six cycles. *How many cubes are left in the active state after the sixth cycle?*

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

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

--- Day 18: Operation Order ---

As you look out the window and notice a heavily-forested continent slowly appear over the horizon, you are interrupted by the child sitting next to you. They're curious if you could help them with their math homework.

Unfortunately, it seems like this "math" follows different rules than you remember.

The homework (your puzzle input) consists of a series of expressions that consist of addition (`+`), multiplication (`*`), and parentheses (`(...)`). Just like normal math, parentheses indicate that the expression inside must be evaluated before it can be used by the surrounding expression. Addition still finds the sum of the numbers on both sides of the operator, and multiplication still finds the product.

However, the rules of *operator precedence* have changed. Rather than evaluating multiplication before addition, the operators have the *same precedence*, and are evaluated left-to-right regardless of the order in which they appear.

For example, the steps to evaluate the expression `1 + 2 * 3 + 4 * 5 + 6` are as follows:

```
1 + 2 * 3 + 4 * 5 + 6
  3   * 3 + 4 * 5 + 6
      9   + 4 * 5 + 6
         13   * 5 + 6
             65   + 6
                 71
```

Parentheses can override this order; for example, here is what happens if parentheses are added to form `1 + (2 * 3) + (4 * (5 + 6))`:

```
1 + (2 * 3) + (4 * (5 + 6))
1 +    6    + (4 * (5 + 6))
     7      + (4 * (5 + 6))
     7      + (4 *   11   )
     7      +     44
            51
```

Here are a few more examples:

- `2 * 3 + (4 * 5)` becomes *`26`*.
- `5 + (8 * 3 + 9 + 3 * 4 * 3)` becomes *`437`*.
- `5 * 9 * (7 * 3 * 3 + 9 * 3 + (8 + 6 * 4))` becomes *`12240`*.
- `((2 + 4 * 9) * (6 + 9 * 8 + 6) + 6) + 2 + 4 * 2` becomes *`13632`*.

Before you can help with the homework, you need to understand it yourself. *Evaluate the expression on each line of the homework; what is the sum of the resulting values?*

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

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

--- Day 19: Monster Messages ---

You land in an airport surrounded by dense forest. As you walk to your high-speed train, the Elves at the Mythical Information Bureau contact you again. They think their satellite has collected an image of a *sea monster*! Unfortunately, the connection to the satellite is having problems, and many of the messages sent back from the satellite have been corrupted.

They sent you a list of *the rules valid messages should obey* and a list of *received messages* they've collected so far (your puzzle input).

The *rules for valid messages* (the top part of your puzzle input) are numbered and build upon each other. For example:

```
0: 1 2
1: "a"
2: 1 3 | 3 1
3: "b"
```

Some rules, like `3: "b"`, simply match a single character (in this case, `b`).

The remaining rules list the sub-rules that must be followed; for example, the rule `0: 1 2` means that to match rule `0`, the text being checked must match rule `1`, and the text after the part that matched rule `1` must then match rule `2`.

Some of the rules have multiple lists of sub-rules separated by a pipe (`|`). This means that *at least one* list of sub-rules must match. (The ones that match might be different each time the rule is encountered.) For example, the rule `2: 1 3 | 3 1` means that to match rule `2`, the text being checked must match rule `1` followed by rule `3` *or* it must match rule `3` followed by rule `1`.

Fortunately, there are no loops in the rules, so the list of possible matches will be finite. Since rule `1` matches `a` and rule `3` matches `b`, rule `2` matches either `ab` or `ba`. Therefore, rule `0` matches `aab` or `aba`.

Here's a more interesting example:

```
0: 4 1 5
1: 2 3 | 3 2
2: 4 4 | 5 5
3: 4 5 | 5 4
4: "a"
5: "b"
```

Here, because rule `4` matches `a` and rule `5` matches `b`, rule `2` matches two letters that are the same (`aa` or `bb`), and rule `3` matches two letters that are different (`ab` or `ba`).

Since rule `1` matches rules `2` and `3` once each in either order, it must match two pairs of letters, one pair with matching letters and one pair with different letters. This leaves eight possibilities: `aaab`, `aaba`, `bbab`, `bbba`, `abaa`, `abbb`, `baaa`, or `babb`.

Rule `0`, therefore, matches `a` (rule `4`), then any of the eight options from rule `1`, then `b` (rule `5`): `aaaabb`, `aaabab`, `abbabb`, `abbbab`, `aabaab`, `aabbbb`, `abaaab`, or `ababbb`.

The *received messages* (the bottom part of your puzzle input) need to be checked against the rules so you can determine which are valid and which are corrupted. Including the rules and the messages together, this might look like:

```
0: 4 1 5
1: 2 3 | 3 2
2: 4 4 | 5 5
3: 4 5 | 5 4
```

```
4: "a"
5: "b"

ababbb
bababa
abbbab
aaabbb
aaaabbb
```

Your goal is to determine *the number of messages that completely match rule 0*. In the above example, `ababbb` and `abbbab` match, but `bababa`, `aaabbb`, and `aaaabbb` do not, producing the answer *2*. The whole message must match all of rule `0`; there can't be extra unmatched characters in the message. (For example, `aaaabbb` might appear to match rule `0` above, but it has an extra unmatched `b` on the end.)

How many messages completely match rule 0?

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

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

--- Day 20: Jurassic Jigsaw ---

The high-speed train leaves the forest and quickly carries you south. You can even see a desert in the distance! Since you have some spare time, you might as well see if there was anything interesting in the image the Mythical Information Bureau satellite captured.

After decoding the satellite messages, you discover that the data actually contains many small images created by the satellite's *camera array*. The camera array consists of many cameras; rather than produce a single square image, they produce many smaller square image *tiles* that need to be *reassembled back into a single image*.

Each camera in the camera array returns a single monochrome *image tile* with a random unique *ID number*. The tiles (your puzzle input) arrived in a random order.

Worse yet, the camera array appears to be malfunctioning: each image tile has been *rotated and flipped to a random orientation*. Your first task is to reassemble the original image by orienting the tiles so they fit together.

To show how the tiles should be reassembled, each tile's image data includes a border that should line up exactly with its adjacent tiles. All tiles have this border, and the border lines up exactly when the tiles are both oriented correctly. Tiles at the edge of the image also have this border, but the outermost edges won't line up with any other tiles.

For example, suppose you have the following nine tiles:

```
Tile 2311:
..##.#..#.
##..#.....
#...##..#.
####.#...#
##.##.###.
##...#.###
.#.#.#..##
..#....#..
###...#.#.
..###..###

Tile 1951:
#.##...##.
#.####...#
.....#..##
#...######
.##.#....#
.###.#####
###.##.##.
.###....#.
..#.#..#.#
#...##.#..

Tile 1171:
####...##.
#..##.#..#
##.#..#.#.
.###.####.
..###.####
.##....##.
.#...####.
#.##.####.
####..#...
.....##...

Tile 1427:
###.##.#..
.#..#.##..
.#.##.#..#
#.#.#.##.#
....#...##
...##..##.
...#.#####
.#.####.#.
..#..###.#
```

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

Tile 1489:
##.#.#....
..##...#..
.##..##...
..#...#...
#####...#.
#..#.#.#.#
...#.#.#..
##.#...##.
..##.##.##
###.##.#..

Tile 2473:
#....####.
#..#.##...
#.##..#...
######.#.#
.#...#.#.#
.#########
.###.#..#.
########.#
##...##.#.
..###.#.#.

Tile 2971:
..#.#....#
#...###...
#.#.###...
##.##..#..
.#####..##
.#..####.#
#..#.#..#.
..####.###
..#.#.###.
...#.#.#.#

Tile 2729:
...#.#.#.#
####.#....
..#.#.....
....#..#.#
.##..##.#.
.#.####...
####.#.#..
```

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

Tile 3079:
#.#.#####.
.#..######
..#.......
######....
####.#..#.
.#...#.##.
#.#####.##
..#.###...
..#.......
..#.###...
```

By rotating, flipping, and rearranging them, you can find a square arrangement that causes all adjacent borders to line up:

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

#.##...##. ..##.#..#. ..#.###...
##..#.##.. ..#..###.# ##.##....#
##.####... .#.####.#. ..#.###..#
####.#.#.. ...#.##### ###.#..###
.#.####... ...##..##. .######.##
.##..##.#. ....#...## #.#.#.#...
....#..#.# #.#.#.##.# #.###.###.
..#.#..... .#.##.#..# #.###.##..
####.#.... .#..#.##.. .######...
...#.#.#.# ###.##.#.. .##...####

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

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

For reference, the IDs of the above tiles are:

```
1951    2311    3079
2729    1427    2473
2971    1489    1171
```

To check that you've assembled the image correctly, multiply the IDs of the four corner tiles together. If you do this with the assembled tiles from the example above, you get `1951 * 3079 * 2971 * 1171` = *20899048083289*.

Assemble the tiles into an image. *What do you get if you multiply together the IDs of the four corner tiles?*

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

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

--- Day 21: Allergen Assessment ---

You reach the train's last stop and the closest you can get to your vacation island without getting wet. There aren't even any boats here, but nothing can stop you now: you build a raft. You just need a few days' worth of food for your journey.

You don't speak the local language, so you can't read any ingredients lists. However, sometimes, allergens are listed in a language you *do* understand. You should be able to use this information to determine which ingredient contains which allergen and work out which foods are safe to take with you on your trip.

You start by compiling a list of foods (your puzzle input), one food per line. Each line includes that food's *ingredients list* followed by some or all of the allergens the food contains.

Each allergen is found in exactly one ingredient. Each ingredient contains zero or one allergen. *Allergens aren't always marked*; when they're listed (as in `(contains nuts, shellfish)` after an ingredients list), the ingredient that contains each listed allergen will be *somewhere in the corresponding ingredients list*. However, even if an allergen isn't listed, the ingredient that contains that allergen could still be present: maybe they forgot to label it, or maybe it was labeled in a language you don't know.

For example, consider the following list of foods:

```
mxmxvkd kfcds sqjhc nhms (contains dairy, fish)
trh fvjkl sbzzf mxmxvkd (contains dairy)
sqjhc fvjkl (contains soy)
sqjhc mxmxvkd sbzzf (contains fish)
```

The first food in the list has four ingredients (written in a language you don't understand): `mxmxvkd`, `kfcds`, `sqjhc`, and `nhms`. While the food might contain other allergens, a few allergens the food definitely contains are listed afterward: `dairy` and `fish`.

The first step is to determine which ingredients *can't possibly* contain any of the allergens in any food in your list. In the above example, none of the ingredients `kfcds`, `nhms`, `sbzzf`, or `trh` can contain an allergen. Counting the number of times any of these ingredients appear in any ingredients list produces *`5`*: they all appear once each except `sbzzf`, which appears twice.

Determine which ingredients cannot possibly contain any of the allergens in your list. *How many times do any of those ingredients appear?*

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

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

--- Day 22: Crab Combat ---

It only takes a few hours of sailing the ocean on a raft for boredom to sink in. Fortunately, you brought a small deck of space cards! You'd like to play a game of *Combat*, and there's even an opponent available: a small crab that climbed aboard your raft before you left.

Fortunately, it doesn't take long to teach the crab the rules.

Before the game starts, split the cards so each player has their own deck (your puzzle input). Then, the game consists of a series of *rounds*: both players draw their top card, and the player with the higher-valued card wins the round. The winner keeps both cards, placing them on the bottom of their own deck so that the winner's card is above the other card. If this causes a player to have all of the cards, they win, and the game ends.

For example, consider the following starting decks:

```
Player 1:
9
2
6
3
1

Player 2:
5
8
4
7
10
```

This arrangement means that player 1's deck contains 5 cards, with `9` on top and `1` on the bottom; player 2's deck also contains 5 cards, with `5` on top and `10` on the bottom.

The first round begins with both players drawing the top card of their decks: `9` and `5`. Player 1 has the higher card, so both cards move to the bottom of player 1's deck such that `9` is above `5`. In total, it takes 29 rounds before a player has all of the cards:

```
-- Round 1 --
Player 1's deck: 9, 2, 6, 3, 1
Player 2's deck: 5, 8, 4, 7, 10
Player 1 plays: 9
Player 2 plays: 5
Player 1 wins the round!

-- Round 2 --
Player 1's deck: 2, 6, 3, 1, 9, 5
Player 2's deck: 8, 4, 7, 10
Player 1 plays: 2
Player 2 plays: 8
Player 2 wins the round!

-- Round 3 --
Player 1's deck: 6, 3, 1, 9, 5
Player 2's deck: 4, 7, 10, 8, 2
Player 1 plays: 6
Player 2 plays: 4
Player 1 wins the round!

-- Round 4 --
Player 1's deck: 3, 1, 9, 5, 6, 4
Player 2's deck: 7, 10, 8, 2
Player 1 plays: 3
Player 2 plays: 7
Player 2 wins the round!

-- Round 5 --
Player 1's deck: 1, 9, 5, 6, 4
Player 2's deck: 10, 8, 2, 7, 3
Player 1 plays: 1
Player 2 plays: 10
Player 2 wins the round!

...several more rounds pass...

-- Round 27 --
```

```
Player 1's deck: 5, 4, 1
Player 2's deck: 8, 9, 7, 3, 2, 10, 6
Player 1 plays: 5
Player 2 plays: 8
Player 2 wins the round!

-- Round 28 --
Player 1's deck: 4, 1
Player 2's deck: 9, 7, 3, 2, 10, 6, 8, 5
Player 1 plays: 4
Player 2 plays: 9
Player 2 wins the round!

-- Round 29 --
Player 1's deck: 1
Player 2's deck: 7, 3, 2, 10, 6, 8, 5, 9, 4
Player 1 plays: 1
Player 2 plays: 7
Player 2 wins the round!


== Post-game results ==
Player 1's deck:
Player 2's deck: 3, 2, 10, 6, 8, 5, 9, 4, 7, 1
```

Once the game ends, you can calculate the winning player's *score*. The bottom card in their deck is worth the value of the card multiplied by 1, the second-from-the-bottom card is worth the value of the card multiplied by 2, and so on. With 10 cards, the top card is worth the value on the card multiplied by 10. In this example, the winning player's score is:

```
   3 * 10
+  2 *  9
+ 10 *  8
+  6 *  7
+  8 *  6
+  5 *  5
+  9 *  4
+  4 *  3
+  7 *  2
+  1 *  1
= 306
```

So, once the game ends, the winning player's score is *`306`*.

Play the small crab in a game of Combat using the two decks you just dealt. *What is the winning player's score?*

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

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

--- Day 23: Crab Cups ---

The small crab challenges *you* to a game! The crab is going to mix up some cups, and you have to predict where they'll end up.

The cups will be arranged in a circle and labeled *clockwise* (your puzzle input). For example, if your labeling were `32415`, there would be five cups in the circle; going clockwise around the circle from the first cup, the cups would be labeled `3`, `2`, `4`, `1`, `5`, and then back to `3` again.

Before the crab starts, it will designate the first cup in your list as the *current cup*. The crab is then going to do *100 moves*.

Each *move*, the crab does the following actions:

- The crab picks up the *three cups* that are immediately *clockwise* of the *current cup*. They are removed from the circle; cup spacing is adjusted as necessary to maintain the circle.
- The crab selects a *destination cup*: the cup with a *label* equal to the *current cup's* label minus one. If this would select one of the cups that was just picked up, the crab will keep subtracting one until it finds a cup that wasn't just picked up. If at any point in this process the value goes below the lowest value on any cup's label, it *wraps around* to the highest value on any cup's label instead.
- The crab places the cups it just picked up so that they are *immediately clockwise* of the destination cup. They keep the same order as when they were picked up.
- The crab selects a new *current cup*: the cup which is immediately clockwise of the current cup.

For example, suppose your cup labeling were `389125467`. If the crab were to do merely 10 moves, the following changes would occur:

```
-- move 1 --
cups: (3) 8  9  1  2  5  4  6  7
pick up: 8, 9, 1
destination: 2

-- move 2 --
cups:  3 (2) 8  9  1  5  4  6  7
pick up: 8, 9, 1
destination: 7

-- move 3 --
cups:  3  2 (5) 4  6  7  8  9  1
```

```
pick up: 4, 6, 7
destination: 3

-- move 4 --
cups:  7  2  5 (8) 9  1  3  4  6
pick up: 9, 1, 3
destination: 7

-- move 5 --
cups:  3  2  5  8 (4) 6  7  9  1
pick up: 6, 7, 9
destination: 3

-- move 6 --
cups:  9  2  5  8  4 (1) 3  6  7
pick up: 3, 6, 7
destination: 9

-- move 7 --
cups:  7  2  5  8  4  1 (9) 3  6
pick up: 3, 6, 7
destination: 8

-- move 8 --
cups:  8  3  6  7  4  1  9 (2) 5
pick up: 5, 8, 3
destination: 1

-- move 9 --
cups:  7  4  1  5  8  3  9  2 (6)
pick up: 7, 4, 1
destination: 5

-- move 10 --
cups: (5) 7  4  1  8  3  9  2  6
pick up: 7, 4, 1
destination: 3

-- final --
cups:  5 (8) 3  7  4  1  9  2  6
```

In the above example, the cups' values are the labels as they appear moving clockwise around the circle; the *current cup* is marked with `( )`.

After the crab is done, what order will the cups be in? Starting *after the cup labeled 1*, collect the other cups' labels clockwise into a single string with no extra characters; each number except `1` should appear exactly once. In the above example, after 10 moves, the cups clockwise from `1` are labeled `9`, `2`, `6`, `5`, and so on, producing *`92658374`*. If the crab were to complete all 100 moves, the order after cup `1` would be *`67384529`*.

Using your labeling, simulate 100 moves. *What are the labels on the cups after cup 1?*

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

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

--- Day 24: Lobby Layout ---

Your raft makes it to the tropical island; it turns out that the small crab was an excellent navigator. You make your way to the resort.

As you enter the lobby, you discover a small problem: the floor is being renovated. You can't even reach the check-in desk until they've finished installing the *new tile floor*.

The tiles are all *hexagonal*; they need to be arranged in a hex grid with a very specific color pattern. Not in the mood to wait, you offer to help figure out the pattern.

The tiles are all *white* on one side and *black* on the other. They start with the white side facing up. The lobby is large enough to fit whatever pattern might need to appear there.

A member of the renovation crew gives you a *list of the tiles that need to be flipped over* (your puzzle input). Each line in the list identifies a single tile that needs to be flipped by giving a series of steps starting from a *reference tile* in the very center of the room. (Every line starts from the same reference tile.)

Because the tiles are hexagonal, every tile has *six neighbors*: east, southeast, southwest, west, northwest, and northeast. These directions are given in your list, respectively, as `e`, `se`, `sw`, `w`, `nw`, and `ne`. A tile is identified by a series of these directions with *no delimiters*; for example, `esenee` identifies the tile you land on if you start at the reference tile and then move one tile east, one tile southeast, one tile northeast, and one tile east.

Each time a tile is identified, it flips from white to black or from black to white. Tiles might be flipped more than once. For example, a line like `esew` flips a tile immediately adjacent to the reference tile, and a line like `nwwswee` flips the reference tile itself.

Here is a larger example:

```
sesenwnenenewseeswwswswwnenewsewsw
neeenesenwnwwswnenewnwwsewnenwseswesw
seswneswswsenwwnwse
nwnwneseeswswnenewneswwnewseswneseene
```

```
swweswneswnenwsewnwneneseenw
eesenwseswswnenwswnwnwsewwnwsene
sewnenenenesenwsewnenwwwse
wenwwweseeeweswwwnwwe
wsweesenenewnwwnwsenewsenwwsesesenwne
neeswseenwwswnwswswnw
nenwswwsewswnenenewsenwsenwnesesenew
enewnwewneswsewnwswenweswnenwsenwsw
sweneswneswneneenwnewenewwneswswnese
swwesenesewenwneswnwwneseswwne
enesenwswwswneneswsenwnewswseenwsese
wnwnesenesenenwwnenwsewesewsesesew
nenewswnwewswnenesenwnesewesw
eneswnwswnwsenenwnwnwwseeswneewsenese
neswnwewnwnwseenwseesewsenwsweewe
wseweeenwnesenwwwswnew
```

In the above example, 10 tiles are flipped once (to black), and 5 more are flipped twice (to black, then back to white). After all of these instructions have been followed, a total of *`10`* tiles are *black*.

Go through the renovation crew's list and determine which tiles they need to flip. After all of the instructions have been followed, *how many tiles are left with the black side up?*

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

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

--- Day 25: Combo Breaker ---

You finally reach the check-in desk. Unfortunately, their registration systems are currently offline, and they cannot check you in. Noticing the look on your face, they quickly add that tech support is already on the way! They even created all the room keys this morning; you can take yours now and give them your room deposit once the registration system comes back online.

The room key is a small RFID card. Your room is on the 25th floor and the elevators are also temporarily out of service, so it takes what little energy you have left to even climb the stairs and navigate the halls. You finally reach the door to your room, swipe your card, and - *beep* - the light turns red.

Examining the card more closely, you discover a phone number for tech support.

"Hello! How can we help you today?" You explain the situation.

"Well, it sounds like the card isn't sending the right command to unlock the door. If you go back to the check-in desk, surely someone there can reset it for you." Still catching your breath, you describe the status of the elevator and the exact number of stairs you just had to climb.

"I see! Well, your only other option would be to reverse-engineer the cryptographic handshake the card does with the door and then inject your own commands into the data stream, but that's definitely impossible." You thank them for their time.

Unfortunately for the door, you know a thing or two about cryptographic handshakes.

The handshake used by the card and the door involves an operation that *transforms* a *subject number*. To transform a subject number, start with the value `1`. Then, a number of times called the *loop size*, perform the following steps:

- Set the value to itself multiplied by the *subject number*.
- Set the value to the remainder after dividing the value by *`20201227`*.

The card always uses a specific, secret *loop size* when it transforms a subject number. The door always uses a different, secret loop size.

The cryptographic handshake works like this:

- The *card* transforms the subject number of *`7`* according to the *card's* secret loop size. The result is called the *card's public key*.
- The *door* transforms the subject number of *`7`* according to the *door's* secret loop size. The result is called the *door's public key*.
- The card and door use the wireless RFID signal to transmit the two public keys (your puzzle input) to the other device. Now, the *card* has the *door's* public key, and the *door* has the *card's* public key. Because you can eavesdrop on the signal, you have both public keys, but neither device's loop size.
- The *card* transforms the subject number of *the door's public key* according to the *card's* loop size. The result is the *encryption key*.
- The *door* transforms the subject number of *the card's public key* according to the *door's* loop size. The result is the same *encryption key* as the *card* calculated.

If you can use the two public keys to determine each device's loop size, you will have enough information to calculate the secret *encryption key* that the card and door use to communicate; this would let you send the `unlock` command directly to the door!

For example, suppose you know that the card's public key is `5764801`. With a little trial and error, you can work out that the card's loop size must be *`8`*, because transforming the initial subject number of `7` with a loop size of `8` produces `5764801`.

Then, suppose you know that the door's public key is `17807724`. By the same process, you can determine that the door's loop size is *`11`*, because transforming the initial subject number of `7` with a loop size of `11` produces `17807724`.

At this point, you can use either device's loop size with the other device's public key to calculate the *encryption key*. Transforming the subject number of `17807724` (the door's public key) with a loop size of `8` (the card's loop size) produces the encryption key, ***`14897079`***. (Transforming the subject number of `5764801` (the card's public key) with a loop size of `11` (the door's loop size) produces the same encryption key: ***`14897079`***.)

What encryption key is the handshake trying to establish?