Advent of fennel

Nazar Stasiv

2024

Contents

2023 [18/50]

DONE Day 1.1

You try to ask why they can't just use a weather machine (”not powerful enough”) and where they're even sending you (”the sky”) and why your map looks mostly blank (”you sure ask a lot of questions”) and hang on did you just say the sky (”of course, where do you think snow comes from”) when you realize that the Elves are already loading you into a trebuchet (”please hold still, we need to strap you in”).

As they're making the final adjustments, they discover that their calibration document (your puzzle input) has been amended by a very young Elf who was apparently just excited to show off her art skills. Consequently, the Elves are having trouble reading the values on the document.

The newly-improved calibration document consists of lines of text; each line originally contained a specific calibration value that the Elves now need to recover. On each line, the calibration value can be found by combining the first digit and the last digit (in that order) to form a single two-digit number.

For example:

```
1abc2
pqr3stu8vwx
a1b2c3d4e5f
treb7uchet
```

In this example, the calibration values of these four lines are 12, 38, 15, and 77. Adding these together produces 142.

Consider your entire calibration document. What is the sum of all of the calibration values?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [lines]
  (let [numbers []]
    (each [_ line (ipairs lines)]
      (let [fd (string.match line "[0-9]")
            ld (string.match (string.reverse line) "[0-9]")]
        (table.insert numbers (tonumber (.. fd ld)))))
    (aoc.fold numbers)))

(fn test1 [expected lines]
  (assert (= expected (solve lines))))

```

```
15 (test1 142 ["1abc2" "pqr3stu8vwx"
    ↪  "a1b2c3d4e5f" "treb7uchet"])
16
17 (solve (aoc.string-from "2023/01.inp"))
```

```
56506
```

DONE Day 1.2

Your calculation isn't quite right. It looks like some of the digits are actually spelled out with letters: one, two, three, four, five, six, seven, eight, and nine also count as valid ”digits”.

Equipped with this new information, you now need to find the real first and last digit on each line. For example:

```
two1nine
eightwothree
abcone2threexyz
xtwone3four
4nineeightseven2
zoneight234
7pqrstsixteen
```

In this example, the calibration values are 29, 83, 13, 24, 42, 14, and 76. Adding these together produces 281.

What is the sum of all of the calibration values?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn starts-at-index [s idx pref]
  (= (string.sub s idx (- (+ idx (length
 ↪  pref)) 1)) pref))

(fn replace-by-index [line index]
  (let [new-line
        (if (starts-at-index line index "one")
            (string.gsub line "one" "o1e" 1)
            (starts-at-index line index "two")
            (string.gsub line "two" "t2o" 1)
            (starts-at-index line index
 ↪  "three")
            (string.gsub line "three" "th3ee"
 ↪  1)
            (starts-at-index line index
 ↪  "four")
            (string.gsub line "four" "fo4r" 1)
            (starts-at-index line index
 ↪  "five")
            (string.gsub line "five" "fi5e" 1)
            (starts-at-index line index "six")
```

```
20              (string.gsub line "six" "s6x" 1)
21              (starts-at-index line index
    ↪  "seven")
22              (string.gsub line "seven" "se7en"
    ↪  1)
23              (starts-at-index line index
    ↪  "eight")
24              (string.gsub line "eight" "ei8ht"
    ↪  1)
25              (starts-at-index line index
    ↪  "nine")
26              (string.gsub line "nine" "n9ne" 1)
27              line)]
28      (if (< (+ 1 index) (length new-line))
29          (replace-by-index new-line (+ 1
    ↪  index))
30          new-line)))
31
32  (fn replace-literal-numbers [lines]
33    (let [new-lines []]
34      (each [_ line (ipairs lines)]
35        (let [new-line (replace-by-index line
    ↪  1)]
36          (table.insert new-lines new-line)))
37      new-lines))
```

```
38
39 (fn solve2 [input]
40   (let [lines (replace-literal-numbers input)]
41     (solve lines)))
42
43 (fn test2 [expected lines]
44   (assert (= expected (solve2 lines))))
45
46 (test2 281
47        ["two1nine"
48         "eightwothree"
49         "abcone2threexyz"
50         "xtwone3four"
51         "4nineeightseven2"
52         "zoneight234"
53         "7pqrstsixteen"])
54
55 (solve2 (aoc.string-from "2023/01.inp"))
```

```
56017
```

DONE Day 2.1

You're launched high into the atmosphere! The apex of your trajectory just barely reaches the surface of a large island float-

ing in the sky. You gently land in a fluffy pile of leaves. It's quite cold, but you don't see much snow. An Elf runs over to greet you.

The Elf explains that you've arrived at Snow Island and apologizes for the lack of snow. He'll be happy to explain the situation, but it's a bit of a walk, so you have some time. They don't get many visitors up here; would you like to play a game in the meantime?

As you walk, the Elf shows you a small bag and some cubes which are either red, green, or blue. Each time you play this game, he will hide a secret number of cubes of each color in the bag, and your goal is to figure out information about the number of cubes.

To get information, once a bag has been loaded with cubes, the Elf will reach into the bag, grab a handful of random cubes, show them to you, and then put them back in the bag. He'll do this a few times per game.

You play several games and record the information from each game (your puzzle input). Each game is listed with its ID number (like the 11 in Game 11: …) followed by a semicolon-separated list of subsets of cubes that were revealed from the bag (like 3

red, 5 green, 4 blue).

For example, the record of a few games might look like this:

```
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue;
 ↪  2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1
 ↪  red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4
 ↪  red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6
 ↪  red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red,
 ↪  2 green
```

In game 1, three sets of cubes are revealed from the bag (and then put back again). The first set is 3 blue cubes and 4 red cubes; the second set is 1 red cube, 2 green cubes, and 6 blue cubes; the third set is only 2 green cubes.

The Elf would first like to know which games would have been possible if the bag contained only 12 red cubes, 13 green cubes, and 14 blue cubes?

In the example above, games 1, 2, and 5 would have been possible if the bag had been loaded with that configuration. However, game 3 would have been impossible because at one point

the Elf showed you 20 red cubes at once; similarly, game 4 would also have been impossible because the Elf showed you 15 blue cubes at once. If you add up the IDs of the games that would have been possible, you get 8.

Determine which games would have been possible if the bag had been loaded with only 12 red cubes, 13 green cubes, and 14 blue cubes. What is the sum of the IDs of those games?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn parse-take [take]
  {:r (tonumber (string.match take "([0-9]*) red"))
   :g (tonumber (string.match take "([0-9]*) green"))
   :b (tonumber (string.match take "([0-9]*) blue"))})

(fn read-game [game]
  {:id (tonumber (string.match game "Game ([0-9]*):"))
   :takes (lume.map (aoc.string-split game ";") parse-take)})

```

```
(fn possible-take? [take]
  (let [max-red 12
        red (or (. take :r) 0)
        max-green 13
        green (or (. take :g) 0)
        max-blue 14
        blue (or (. take :b) 0)]
    (not (or (> red max-red)
             (> green max-green)
             (> blue max-blue)))))

(fn possible-game? [game]
  (= nil (lume.find (lume.map (. game :takes)
 ↪  possible-take?) false)))

(fn sum-game-ids [games]
  (accumulate [sum 0 _ game (ipairs games)]
    (+ sum (. game :id))))

(fn solve [lines]
  (let [games []]
    (each [_ line (ipairs lines)]
      (let [game (read-game line)]
        (table.insert games game)))
    (sum-game-ids (lume.filter games
 ↪  possible-game?))))
```

```
37
38 (fn test1 [expected lines]
39   (assert (= expected (solve lines))))
40
41 (local test-input
42        ["Game 1: 3 blue, 4 red; 1 red, 2
    ↪  green, 6 blue; 2 green"
43         "Game 2: 1 blue, 2 green; 3 green, 4
    ↪  blue, 1 red; 1 green, 1 blue"
44         "Game 3: 8 green, 6 blue, 20 red; 5
    ↪  blue, 4 red, 13 green; 5 green, 1 red"
45         "Game 4: 1 green, 3 red, 6 blue; 3
    ↪  green, 6 red; 3 green, 15 blue, 14 red"
46         "Game 5: 6 red, 1 blue, 3 green; 2
    ↪  blue, 1 red, 2 green"])
47
48 (test1 8 test-input)
49
50 (solve (aoc.string-from "2023/02.inp"))
```

```
2449
```

DONE Day 2.2

The Elf says they've stopped producing snow because they aren't getting any water! He isn't sure why the water stopped; however, he can show you how to get to the water source to check it out for yourself. It's just up ahead!

As you continue your walk, the Elf poses a second question: in each game you played, what is the fewest number of cubes of each color that could have been in the bag to make the game possible?

Again consider the example games from earlier:

```
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue;
 ↪  2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1
 ↪  red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4
 ↪  red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6
 ↪  red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red,
 ↪  2 green
```

- In game 1, the game could have been played with as few as 4 red, 2 green, and 6 blue cubes. If any color had even

one fewer cube, the game would have been impossible.
- Game 2 could have been played with a minimum of 1 red, 3 green, and 4 blue cubes.
- Game 3 must have been played with at least 20 red, 13 green, and 6 blue cubes.
- Game 4 required at least 14 red, 3 green, and 15 blue cubes.
- Game 5 needed no fewer than 6 red, 3 green, and 2 blue cubes in the bag.

The power of a set of cubes is equal to the numbers of red, green, and blue cubes multiplied together. The power of the minimum set of cubes in game 1 is 48. In games 2-5 it was 12, 1560, 630, and 36, respectively. Adding up these five powers produces the sum 2286.

For each game, find the minimum set of cubes that must have been present. What is the sum of the power of these sets?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn find-max-red-take [game]
  (aoc.table-max
   (lume.map (. game :takes)
```

```
7               (fn [take] (or (. take :r) 0)))))
8
9   (fn find-max-green-take [game]
10    (aoc.table-max
11     (lume.map (. game :takes)
12               (fn [take] (or (. take :g) 0)))))
13
14  (fn find-max-blue-take [game]
15    (aoc.table-max
16     (lume.map (. game :takes)
17               (fn [take] (or (. take :b) 0)))))
18
19  (fn find-game-power-cube [game]
20    (*
21     (find-max-red-take game)
22     (find-max-green-take game)
23     (find-max-blue-take game)))
24
25  (fn solve [lines]
26    (let [power-cubes []]
27      (each [_ line (ipairs lines)]
28        (let [game (read-game line)]
29          (table.insert power-cubes
    ↪  (find-game-power-cube game))))
30      (aoc.table-sum power-cubes)))
```

```
31
32 (fn test2 [expected lines]
33   (assert (= expected (solve lines))))
34
35 (test2 2286 test-input)
36
37 (solve (aoc.string-from "2023/02.inp"))
```

```
63981
```

DONE Day 4.1

The gondola takes you up. Strangely, though, the ground doesn't seem to be coming with you; you're not climbing a mountain. As the circle of Snow Island recedes below you, an entire new landmass suddenly appears above you! The gondola carries you to the surface of the new island and lurches into the station.

As you exit the gondola, the first thing you notice is that the air here is much warmer than it was on Snow Island. It's also quite humid. Is this where the water source is?

The next thing you notice is an Elf sitting on the floor across the station in what seems to be a pile of colorful square cards.

”Oh! Hello!” The Elf excitedly runs over to you. ”How may I be of service?” You ask about water sources.

”I'm not sure; I just operate the gondola lift. That does sound like something we'd have, though - this is Island Island, after all! I bet the gardener would know. He's on a different island, though - er, the small kind surrounded by water, not the floating kind. We really need to come up with a better naming scheme. Tell you what: if you can help me with something quick, I'll let you borrow my boat and you can go visit the gardener. I got all these scratchcards as a gift, but I can't figure out what I've won.”

The Elf leads you over to the pile of colorful cards. There, you discover dozens of scratchcards, all with their opaque covering already scratched off. Picking one up, it looks like each card has two lists of numbers separated by a vertical bar (|): a list of winning numbers and then a list of numbers you have. You organize the information into a table (your puzzle input).

As far as the Elf has been able to figure out, you have to figure out which of the numbers you have appear in the list of winning numbers. The first match makes the card worth one point and each match after the first doubles the point value of that card.

For example:

```
Card 1: 41 48 83 86 17 | 83 86  6 31 17  9 48
 ↪  53
Card 2: 13 32 20 16 61 | 61 30 68 82 17 32 24
 ↪  19
Card 3:  1 21 53 59 44 | 69 82 63 72 16 21 14
 ↪  1
Card 4: 41 92 73 84 69 | 59 84 76 51 58  5 54
 ↪  83
Card 5: 87 83 26 28 32 | 88 30 70 12 93 22 82
 ↪  36
Card 6: 31 18 13 56 72 | 74 77 10 23 35 67 36
 ↪  11
```

In the above example, card 1 has five winning numbers (41, 48, 83, 86, and 17) and eight numbers you have (83, 86, 6, 31, 17, 9, 48, and 53). Of the numbers you have, four of them (48, 83, 17, and 86) are winning numbers! That means card 1 is worth 8 points (1 for the first match, then doubled three times for each of the three matches after the first).

- Card 2 has two winning numbers (32 and 61), so it is worth 2 points.
- Card 3 has two winning numbers (1 and 21), so it is worth 2 points.

- Card 4 has one winning number (84), so it is worth 1 point.
- Card 5 has no winning numbers, so it is worth no points.
- Card 6 has no winning numbers, so it is worth no points.

So, in this example, the Elf's pile of scratchcards is worth 13 points. Take a seat in the large pile of colorful cards. How many points are they worth in total?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-game [s]
  (let [id (tonumber (string.match s "Card *([0-9]*):"))
        lottery (aoc.string-split (string.match s ":([0-9 ]*)|") " ")
        ticket (aoc.string-split (string.match s "|([0-9 ]*)$") " ")]
    {:id id :lottery (lume.map lottery tonumber) :ticket (lume.map ticket tonumber)}))

(fn read-games [lines]
  (let [result []]
    (each [_ line (ipairs lines)]
      (let [game (read-game line)]
```

```
14          (table.insert result game)))
15      result))
16
17  (fn wins-to-points [n]
18    (if (> n 0)
19        (aoc.math-pow 2 (- n 1))
20        0))
21
22  (fn count-points [games]
23    (let [result []]
24      (each [_ game (ipairs games)]
25        (let [wins (aoc.table-intersect (. game
    ↪  :lottery) (. game :ticket))
26              points (wins-to-points (length
    ↪  wins))]
27          (table.insert result points)))
28      result))
29
30  (fn solve [lines]
31    (let [games (read-games lines)]
32      (aoc.table-sum (count-points games))))
33
34  (fn test [expected input]
35    (assert (= expected (solve input))))
36
```

```
(local test-input
       ["Card 1: 41 48 83 86 17 | 83 86  6 31
 ↪  17  9 48 53"
        "Card 2: 13 32 20 16 61 | 61 30 68 82
 ↪  17 32 24 19"
        "Card 3:  1 21 53 59 44 | 69 82 63 72
 ↪  16 21 14  1"
        "Card 4: 41 92 73 84 69 | 59 84 76 51
 ↪  58  5 54 83"
        "Card 5: 87 83 26 28 32 | 88 30 70 12
 ↪  93 22 82 36"
        "Card 6: 31 18 13 56 72 | 74 77 10 23
 ↪  35 67 36 11"])

(test 13 test-input)

(solve (aoc.string-from "2023/04.inp"))
```

```
18653
```

DONE Day 4.2

Just as you're about to report your findings to the Elf, one of you realizes that the rules have actually been printed on the back of every card this whole time.

There's no such thing as ”points”. Instead, scratchcards only cause you to win more scratchcards equal to the number of winning numbers you have.

Specifically, you win copies of the scratchcards below the winning card equal to the number of matches. So, if card 10 were to have 5 matching numbers, you would win one copy each of cards 11, 12, 13, 14, and 15.

Copies of scratchcards are scored like normal scratchcards and have the same card number as the card they copied. So, if you win a copy of card 10 and it has 5 matching numbers, it would then win a copy of the same cards that the original card 10 won: cards 11, 12, 13, 14, and 15. This process repeats until none of the copies cause you to win any more cards. (Cards will never make you copy a card past the end of the table.)

This time, the above example goes differently:

```
Card 1: 41 48 83 86 17 | 83 86  6 31 17  9 48
 ↪  53
Card 2: 13 32 20 16 61 | 61 30 68 82 17 32 24
 ↪  19
Card 3:  1 21 53 59 44 | 69 82 63 72 16 21 14
 ↪  1
```

```
Card 4: 41 92 73 84 69 | 59 84 76 51 58  5 54
 ↪  83
Card 5: 87 83 26 28 32 | 88 30 70 12 93 22 82
 ↪  36
Card 6: 31 18 13 56 72 | 74 77 10 23 35 67 36
 ↪  11
```

- Card 1 has four matching numbers, so you win one copy each of the next four cards: cards 2, 3, 4, and 5.
- Your original card 2 has two matching numbers, so you win one copy each of cards 3 and 4.
- Your copy of card 2 also wins one copy each of cards 3 and 4.
- Your four instances of card 3 (one original and three copies) have two matching numbers, so you win four copies each of cards 4 and 5.
- Your eight instances of card 4 (one original and seven copies) have one matching number, so you win eight copies of card 5.
- Your fourteen instances of card 5 (one original and thirteen copies) have no matching numbers and win no more cards.
- Your one instance of card 6 (one original) has no matching numbers and wins no more cards.

Once all of the originals and copies have been processed, you end up with 1 instance of card 1, 2 instances of card 2, 4 instances of card 3, 8 instances of card 4, 14 instances of card 5, and 1 instance of card 6. In total, this example pile of scratchcards causes you to ultimately have 30 scratchcards!

Process all of the original and copied scratchcards until no more scratchcards are won. Including the original set of scratchcards, how many total scratchcards do you end up with?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn table.increment [t i v]
  (let [o (. t i)]
    (table.remove t i)
    (table.insert t i (+ o v)))
  t)

(fn collect-cards [games]
  (let [result []]
    (each [_ game (ipairs games)]
      (let [count (length (aoc.table-intersect
 ↪  (. game :lottery) (. game :ticket)))
            id (. game :id)
            cards []]
```

```
16          (when (< 0 count)
17            (fcollect [i (+ id 1) (+ count id)
    ↪  1]
18              (table.insert cards i)))
19          (tset result id cards)))
20      result))
21
22  (fn count-cards [games]
23    (let [cards (collect-cards games)
24          result (fcollect [i 1 (length cards)
    ↪  1] 1)]
25      (each [i v (ipairs cards)]
26        (each [j w (ipairs v)]
27          (table.increment result w (. result
    ↪  i))))
28      result))
29
30  (fn solve2 [input]
31    (let [games (read-games input)]
32      (aoc.table-sum (count-cards games))))
33
34  (fn test2 [expected input]
35    (assert (= expected (solve2 input))))
36
37  (test2 30 test-input)
```

```
38
39 (solve2 (aoc.string-from "2023/04.inp"))
```

```
5921508
```

DONE Day 5.1

You take the boat and find the gardener right where you were told he would be: managing a giant "garden" that looks more to you like a farm.

"A water source? Island Island is the water source!" You point out that Snow Island isn't receiving any water.

"Oh, we had to stop the water because we ran out of sand to filter it with! Can't make snow with dirty water. Don't worry, I'm sure we'll get more sand soon; we only turned off the water a few days... weeks... oh no." His face sinks into a look of horrified realization.

"I've been so busy making sure everyone here has food that I completely forgot to check why we stopped getting more sand! There's a ferry leaving soon that is headed over in that direction - it's much faster than your boat. Could you please go check it out?"

You barely have time to agree to this request when he brings up another. ”While you wait for the ferry, maybe you can help us with our food production problem. The latest Island Island Almanac just arrived and we're having trouble making sense of it.”

The almanac (your puzzle input) lists all of the seeds that need to be planted. It also lists what type of soil to use with each kind of seed, what type of fertilizer to use with each kind of soil, what type of water to use with each kind of fertilizer, and so on. Every type of seed, soil, fertilizer and so on is identified with a number, but numbers are reused by each category - that is, soil 123 and fertilizer 123 aren't necessarily related to each other.

For example:

```
seeds: 79 14 55 13

seed-to-soil map:
50 98 2
52 50 48

soil-to-fertilizer map:
0 15 37
37 52 2
39 0 15
```

```
fertilizer-to-water map:
49 53 8
0 11 42
42 0 7
57 7 4

water-to-light map:
88 18 7
18 25 70

light-to-temperature map:
45 77 23
81 45 19
68 64 13

temperature-to-humidity map:
0 69 1
1 0 69

humidity-to-location map:
60 56 37
56 93 4
```

The almanac starts by listing which seeds need to be planted: seeds 79, 14, 55, and 13.

The rest of the almanac contains a list of maps which describe how to convert numbers from a source category into numbers in a destination category. That is, the section that starts with seed-to-soil map: describes how to convert a seed number (the source) to a soil number (the destination). This lets the gardener and his team know which soil to use with which seeds, which water to use with which fertilizer, and so on.

Rather than list every source number and its corresponding destination number one by one, the maps describe entire ranges of numbers that can be converted. Each line within a map contains three numbers: the destination range start, the source range start, and the range length.

Consider again the example seed-to-soil map:

```
50 98 2
52 50 48
```

The first line has a destination range start of 50, a source range start of 98, and a range length of 2. This line means that the source range starts at 98 and contains two values: 98 and 99. The destination range is the same length, but it starts at 50, so its two values are 50 and 51. With this information, you know that seed number 98 corresponds to soil number 50 and that

seed number 99 corresponds to soil number 51.

The second line means that the source range starts at 50 and contains 48 values: 50, 51, …, 96, 97. This corresponds to a destination range starting at 52 and also containing 48 values: 52, 53, …, 98, 99. So, seed number 53 corresponds to soil number 55.

Any source numbers that aren't mapped correspond to the same destination number. So, seed number 10 corresponds to soil number 10.

So, the entire list of seed numbers and their corresponding soil numbers looks like this:

```
seed  soil
0     0
1     1
...   ...
48    48
49    49
50    52
51    53
...   ...
96    98
97    99
```

```
98 50
99 51
```

With this map, you can look up the soil number required for each initial seed number:

- Seed number 79 corresponds to soil number 81.
- Seed number 14 corresponds to soil number 14.
- Seed number 55 corresponds to soil number 57.
- Seed number 13 corresponds to soil number 13.

The gardener and his team want to get started as soon as possible, so they'd like to know the closest location that needs a seed. Using these maps, find the lowest location number that corresponds to any of the initial seeds. To do this, you'll need to convert each seed number through other categories until you can find its corresponding location number. In this example, the corresponding types are:

- Seed 79, soil 81, fertilizer 81, water 81, light 74, temperature 78, humidity 78, location 82.
- Seed 14, soil 14, fertilizer 53, water 49, light 42, temperature 42, humidity 43, location 43.
- Seed 55, soil 57, fertilizer 57, water 53, light 46, temperature 82, humidity 82, location 86.

- Seed 13, soil 13, fertilizer 52, water 41, light 34, temperature 34, humidity 35, location 35.

So, the lowest location number in this example is 35.

What is the lowest location number that corresponds to any of the initial seed numbers?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn vec2tree2 [node t ?f]
  (let [len (length t)]
    (if (= 0 len) nil
        (= 1 len) (tset node :val (. t 1))
        (do
          (table.sort t ?f)
          (let [mid (math.ceil (/ len 2))
                left (aoc.table-range t 1 (-
  ↪ mid 1))
                right (aoc.table-range t (+
  ↪ mid 1) len)]
            (tset node :val (. t mid))
            (when (not (aoc.empty? left))
  ↪ (tset node :left (vec2tree2 {} left ?f)))
```

```
15              (when (not (aoc.empty? right))
    ↪  (tset node :right (vec2tree2 {} right
    ↪  ?f))))))
16      node))
17
18  (fn comp [a b]
19    (< (. a 2) (. b 2)))
20
21  (local seed2soil
22         (vec2tree2 {}
23                    [[50 98 2]
24                     [52 50 48]]
25                    comp))
26
27  (local soil2fertilizer
28         (vec2tree2 {}
29                    [[0 15 37]
30                     [37 52 2]
31                     [39 0 15]]
32                    comp))
33
34  (local fertilizer2water
35         (vec2tree2 {}
36                    [[49 53 8]
37                     [0 11 42]
```

```
                   [42 0 7]
                   [57 7 4]]
                  comp))


(local water2light
       (vec2tree2 {}
                  [[88 18 7]
                   [18 25 70]]
                  comp))

(local light2temperature
       (vec2tree2 {}
                  [[45 77 23]
                   [81 45 19]
                   [68 64 13]]
                  comp))

(local temperature2humidity
       (vec2tree2 {}
                  [[0 69 1]
                   [1 0 69]]
                  comp))

(local humidity2location
```

```
63        (vec2tree2 {}
64                   [[60 56 37]
65                    [56 93 4]]
66                   comp))
67
68  (local test-seeds
69         [79 14 55 13])
70
71  (fn search-in-range [node v]
72    (if node
73        (let [from (. (. node :val) 2)
74              to (- (+ from (. (. node :val) 3))
    ↪  1)]
75          (if (< v from) (search-in-range (.
    ↪  node :left) v)
76              (> v to) (search-in-range (. node
    ↪  :right) v)
77              (and (<= from v) (<= v to)) (+ (-
    ↪  v from) (. (. node :val) 1))))
78        v))
79
80  (fn seed2soil2fert2water2light2temp2hum2loc [x
    ↪  t1 t2 t3 t4 t5 t6 t7]
81    (->> x
82         (search-in-range t1)
```

```
        (search-in-range t2)
        (search-in-range t3)
        (search-in-range t4)
        (search-in-range t5)
        (search-in-range t6)
        (search-in-range t7)))

(fn test1 [xs t1 t2 t3 t4 t5 t6 t7]
  (let [locations []]
    (each [_ seed (ipairs xs)]
      (let [location
  ↪ (seed2soil2fert2water2light2temp2hum2loc
  ↪ seed t1 t2 t3 t4 t5 t6 t7)]
          (table.insert locations location)))
    (assert (= 35 (aoc.table-min
  ↪ locations)))))

(test1 test-seeds seed2soil soil2fertilizer
  ↪ fertilizer2water water2light
  ↪ light2temperature temperature2humidity
  ↪ humidity2location)

(fn solve [xs]
  (let [locations []
```

```
101         seeds (aoc.table-unpack (lume.map
    ↪ (aoc.table-range xs 1 1)
    ↪ #(aoc.string-tonumarray $)))
102         t1 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 4 27)
    ↪ #(aoc.string-tonumarray $))  comp)
103         t2 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 30 60)
    ↪ #(aoc.string-tonumarray $)) comp)
104         t3 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 63 72)
    ↪ #(aoc.string-tonumarray $)) comp)
105         t4 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 75 101)
    ↪ #(aoc.string-tonumarray $)) comp)
106         t5 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 104 114)
    ↪ #(aoc.string-tonumarray $)) comp)
107         t6 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 117 129)
    ↪ #(aoc.string-tonumarray $)) comp)
108         t7 (vec2tree2 {} (lume.map
    ↪ (aoc.table-range xs 132 139)
    ↪ #(aoc.string-tonumarray $)) comp)]
109     (each [_ seed (ipairs seeds)]
```

```
110       (table.insert locations
      ↪  (seed2soil2fert2water2light2temp2hum2loc
      ↪  seed t1 t2 t3 t4 t5 t6 t7)))
111     (aoc.table-min locations)))
112
113 (solve (aoc.string-from "2023/05.inp"))
```

```
1181555926
```

DONE Day 6.1

The ferry quickly brings you across Island Island. After asking around, you discover that there is indeed normally a large pile of sand somewhere near here, but you don't see anything besides lots of water and the small island where the ferry has docked.

As you try to figure out what to do next, you notice a poster on a wall near the ferry dock. "Boat races! Open to the public! Grand prize is an all-expenses-paid trip to Desert Island!" That must be where the sand comes from! Best of all, the boat races are starting in just a few minutes.

You manage to sign up as a competitor in the boat races just in time. The organizer explains that it's not really a traditional

race - instead, you will get a fixed amount of time during which your boat has to travel as far as it can, and you win if your boat goes the farthest.

As part of signing up, you get a sheet of paper (your puzzle input) that lists the time allowed for each race and also the best distance ever recorded in that race. To guarantee you win the grand prize, you need to make sure you go farther in each race than the current record holder.

The organizer brings you over to the area where the boat races are held. The boats are much smaller than you expected - they're actually toy boats, each with a big button on top. Holding down the button charges the boat, and releasing the button allows the boat to move. Boats move faster if their button was held longer, but time spent holding the button counts against the total race time. You can only hold the button at the start of the race, and boats don't move until the button is released.

For example:

```
Time:      7  15   30
Distance:  9  40  200
```

This document describes three races:

- The first race lasts 7 milliseconds. The record distance in this race is 9 millimeters.
- The second race lasts 15 milliseconds. The record distance in this race is 40 millimeters.
- The third race lasts 30 milliseconds. The record distance in this race is 200 millimeters.

Your toy boat has a starting speed of zero millimeters per millisecond. For each whole millisecond you spend at the beginning of the race holding down the button, the boat's speed increases by one millimeter per millisecond.

So, because the first race lasts 7 milliseconds, you only have a few options:

- Don't hold the button at all (that is, hold it for 0 milliseconds) at the start of the race. The boat won't move; it will have traveled 0 millimeters by the end of the race.
- Hold the button for 1 millisecond at the start of the race. Then, the boat will travel at a speed of 1 millimeter per millisecond for 6 milliseconds, reaching a total distance traveled of 6 millimeters.
- Hold the button for 2 milliseconds, giving the boat a speed of 2 millimeters per millisecond. It will then get

5 milliseconds to move, reaching a total distance of 10 millimeters.
- Hold the button for 3 milliseconds. After its remaining 4 milliseconds of travel time, the boat will have gone 12 millimeters.
- Hold the button for 4 milliseconds. After its remaining 3 milliseconds of travel time, the boat will have gone 12 millimeters.
- Hold the button for 5 milliseconds, causing the boat to travel a total of 10 millimeters.
- Hold the button for 6 milliseconds, causing the boat to travel a total of 6 millimeters.
- Hold the button for 7 milliseconds. That's the entire duration of the race. You never let go of the button. The boat can't move until you let you of the button. Please make sure you let go of the button so the boat gets to move. 0 millimeters.

Since the current record for this race is 9 millimeters, there are actually 4 different ways you could win: you could hold the button for 2, 3, 4, or 5 milliseconds at the start of the race.

In the second race, you could hold the button for at least 4 milliseconds and at most 11 milliseconds and beat the record,

a total of 8 different ways to win.

In the third race, you could hold the button for at least 11 milliseconds and no more than 19 milliseconds and still beat the record, a total of 9 ways you could win.

To see how much margin of error you have, determine the number of ways you can beat the record in each race; in this example, if you multiply these values together, you get 288 (4 * 8 * 9).

Determine the number of ways you could beat the record in each race. What do you get if you multiply these numbers together?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn time2distance [time speed]
  (* time speed))

(fn race2distance [time]
  (fcollect [charge 0 time 1]
    (time2distance (- time charge) charge)))

(fn find-wins [input]
  (let [result []]
    (each [_ [time record] (ipairs input)]
```

```
14        (table.insert result
15                      (lume.reduce
16                       (race2distance time)
17                       (fn [acc x] (if (> x
    ↪  record) (+ acc 1) acc)))))
18      result))
19
20  (fn count-wins [wins]
21    (accumulate [prod 1
22                 _ win (ipairs wins)]
23      (* prod win)))
24
25  (fn lines-to-array [lines]
26    [(lume.map (aoc.string-split (. lines 1) "
    ↪  ") #(tonumber $))
27     (lume.map (aoc.string-split (. lines 2) "
    ↪  ") #(tonumber $))])
28
29  (fn solve [lines]
30    (let [[i1 i2] (lines-to-array lines)
31          input (aoc.table-zip i1 i2)]
32      (count-wins (find-wins input))))
33
34  (local test-input
35         ["Time:      7  15   30"
```

```
36          "Distance:  9  40  200"])
37
38  (fn test1 [expected input]
39    (assert (= expected (solve input))))
40
41  (test1 288 test-input)
42
43  (solve (aoc.string-from "2023/06.inp"))
```

```
4811940
```

DONE Day 6.2

As the race is about to start, you realize the piece of paper with race times and record distances you got earlier actually just has very bad kerning. There's really only one race - ignore the spaces between the numbers on each line.

So, the example from before:

```
Time:      7  15   30
Distance:  9  40  200
```

…now instead means this:

```
Time:      71530
Distance:  940200
```

Now, you have to figure out how many ways there are to win this single race. In this example, the race lasts for 71530 milliseconds and the record distance you need to beat is 940200 millimeters. You could hold the button anywhere from 14 to 71516 milliseconds and beat the record, a total of 71503 ways!

How many ways can you beat the record in this one much longer race?

```
(fn lines-to-string [lines]
  (let [time (string.gsub (string.gsub (. lines 1) " *" "") "Time:" "")
        distance (string.gsub (string.gsub (. lines 2) " *" "") "Distance:" "")]
    [(tonumber time)
     (tonumber distance)]))

(fn solve2 [lines]
  (let [input [(lines-to-string lines)]]
    (count-wins (find-wins input))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 71503 test-input)

```

```
16 (solve2 (aoc.string-from "2023/06.inp"))
```

```
30077773
```

DONE Day 7.1

Your all-expenses-paid trip turns out to be a one-way, five-minute ride in an airship. (At least it's a cool airship!) It drops you off at the edge of a vast desert and descends back to Island Island.

"Did you bring the parts?"

You turn around to see an Elf completely covered in white clothing, wearing goggles, and riding a large camel.

"Did you bring the parts?" she asks again, louder this time. You aren't sure what parts she's looking for; you're here to figure out why the sand stopped.

"The parts! For the sand, yes! Come with me; I will show you." She beckons you onto the camel.

After riding a bit across the sands of Desert Island, you can see what look like very large rocks covering half of the horizon. The Elf explains that the rocks are all along the part of Desert Island

that is directly above Island Island, making it hard to even get there. Normally, they use big machines to move the rocks and filter the sand, but the machines have broken down because Desert Island recently stopped receiving the parts they need to fix the machines.

You've already assumed it'll be your job to figure out why the parts stopped when she asks if you can help. You agree automatically.

Because the journey will take a few days, she offers to teach you the game of Camel Cards. Camel Cards is sort of similar to poker except it's designed to be easier to play while riding a camel.

In Camel Cards, you get a list of hands, and your goal is to order them based on the strength of each hand. A hand consists of five cards labeled one of A, K, Q, J, T, 9, 8, 7, 6, 5, 4, 3, or 2. The relative strength of each card follows this order, where A is the highest and 2 is the lowest.

Every hand is exactly one type. From strongest to weakest, they are:

- Five of a kind, where all five cards have the same label: AAAAA

- Four of a kind, where four cards have the same label and one card has a different label: AA8AA
- Full house, where three cards have the same label, and the remaining two cards share a different label: 23332
- Three of a kind, where three cards have the same label, and the remaining two cards are each different from any other card in the hand: TTT98
- Two pair, where two cards share one label, two other cards share a second label, and the remaining card has a third label: 23432
- One pair, where two cards share one label, and the other three cards have a different label from the pair and each other: A23A4
- High card, where all cards' labels are distinct: 23456

Hands are primarily ordered based on type; for example, every full house is stronger than any three of a kind.

If two hands have the same type, a second ordering rule takes effect. Start by comparing the first card in each hand. If these cards are different, the hand with the stronger first card is considered stronger. If the first card in each hand have the same label, however, then move on to considering the second card in each hand. If they differ, the hand with the higher second

card wins; otherwise, continue with the third card in each hand, then the fourth, then the fifth.

So, 33332 and 2AAAA are both four of a kind hands, but 33332 is stronger because its first card is stronger. Similarly, 77888 and 77788 are both a full house, but 77888 is stronger because its third card is stronger (and both hands have the same first and second card).

To play Camel Cards, you are given a list of hands and their corresponding bid (your puzzle input). For example:

```
32T3K 765
T55J5 684
KK677 28
KTJJT 220
QQQJA 483
```

This example shows five hands; each hand is followed by its bid amount. Each hand wins an amount equal to its bid multiplied by its rank, where the weakest hand gets rank 1, the second-weakest hand gets rank 2, and so on up to the strongest hand. Because there are five hands in this example, the strongest hand will have rank 5 and its bid will be multiplied by 5.

So, the first step is to put the hands in order of strength:

- 32T3K is the only one pair and the other hands are all a stronger type, so it gets rank 1.
- KK677 and KTJJT are both two pair. Their first cards both have the same label, but the second card of KK677 is stronger (K vs T), so KTJJT gets rank 2 and KK677 gets rank 3.
- T55J5 and QQQJA are both three of a kind. QQQJA has a stronger first card, so it gets rank 5 and T55J5 gets rank 4.

Now, you can determine the total winnings of this set of hands by adding up the result of multiplying each hand's bid with its rank (765 * 1 + 220 * 2 + 28 * 3 + 684 * 4 + 483 * 5). So the total winnings in this example are 6440.

Find the rank of every hand in your set. What are the total winnings?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn all-wins [hands]
  (accumulate [sum 0 rank hand (ipairs hands)]
    (+ sum (* rank (. hand :bid)))))

```

```
(fn hand2type [hand]
  (case (aoc.string-toarray hand)
    ;; five of a kind
    [a a a a a] :t7
    ;; four of a kind
    [a b b b b] :t6
    [b a b b b] :t6
    [b b a b b] :t6
    [b b b a b] :t6
    [b b b b a] :t6
    ;; full house
    [a a a b b] :t5
    [a a b a b] :t5
    [a b a a b] :t5
    [b a a a b] :t5
    [b a a b a] :t5
    [b a b a a] :t5
    [b b a a a] :t5
    [a b b a a] :t5
    [a a b b a] :t5
    [a b a b a] :t5
    ;; three of a kind
    [b c a a a] :t4
    [b a c a a] :t4
    [b a a c a] :t4
```

```
33      [b a a a c] :t4
34      [a b a a c] :t4
35      [a a b a c] :t4
36      [a a a b c] :t4
37      [a b c a a] :t4
38      [a a b c a] :t4
39      [a b a c a] :t4
40      ;; two pairs
41      [a a b b c] :t3
42      [a a b c b] :t3
43      [a a c b b] :t3
44      [a c a b b] :t3
45      [c a a b b] :t3
46      [a b a b c] :t3
47      [a b a c b] :t3
48      [a b c a b] :t3
49      [a c b a b] :t3
50      [c a b a b] :t3
51      [a b b a c] :t3
52      [a b b c a] :t3
53      [a b c b a] :t3
54      [a c b b a] :t3
55      [c a b b a] :t3
56      ;; one pair
57      [a a b c d] :t2
```

```
    [a b a c d] :t2
    [a b c a d] :t2
    [a b c d a] :t2
    [b a c d a] :t2
    [b c a d a] :t2
    [b c d a a] :t2
    [b a a c d] :t2
    [b c a a d] :t2
    [b a c a d] :t2
    ;; high card
    [a b c d e] :t1))

(fn hand2number [hand]
  (accumulate [sum 0 i x (ipairs
  ↪ (aoc.table-reverse (aoc.string-toarray
  ↪ hand)))]
    (+ sum (* (^ 100 (- i 1))
              (case x
                "A" 14
                "K" 13
                "Q" 12
                "J" 11
                "T" 10
                "9" 9
                "8" 8
```

```
                  "7" 7
                  "6" 6
                  "5" 5
                  "4" 4
                  "3" 3
                  "2" 2)))))

(fn test-hand2number []
  (assert (= 1212121114 (hand2number
  ↪  "QQQJA")))
  (assert (= 1414141414 (hand2number
  ↪  "AAAAA")))
  (assert (= 202020202 (hand2number "22222")))
  (assert (= 1313060707 (hand2number
  ↪  "KK677")))
  (assert (= 1310111110 (hand2number
  ↪  "KTJJT"))))

(test-hand2number)

(fn numeric-comp [a b]
  (let [an (hand2number (. a :hand))
        bn (hand2number (. b :hand))]
    (< an bn)))

```

```
(fn test-numeric-comp []
  (assert (numeric-comp {:hand "QQQJA"} {:hand
  ↪  "AAAAA"}))
  (assert (not (numeric-comp {:hand "AAAAA"}
  ↪  {:hand "QQQJA"})))
  (assert (numeric-comp {:hand "22222"} {:hand
  ↪  "QQQJA"}))
  (assert (not (numeric-comp {:hand "QQQJA"}
  ↪  {:hand "22222"})))
  (assert (not (numeric-comp {:hand "KK677"}
  ↪  {:hand "KTJJT"}))))

(test-numeric-comp)

(fn table.join [xs ys]
  (table.sort ys numeric-comp)
  (table.move ys 1 (length ys) (+ 1 (length
  ↪  xs)) xs))

(fn all-hands [lines]
  (let [t7 [] t6 [] t5 [] t4 [] t3 [] t2 [] t1
  ↪  [] allhands []]
    (each [_ line (ipairs lines)]
      (let [[hand bid] (aoc.string-split line
  ↪  " ")]
```

```
119         (case (hand2type hand)
120           "t7" (table.insert t7 {:hand hand
    ↪  :bid (tonumber bid)})
121           "t6" (table.insert t6 {:hand hand
    ↪  :bid (tonumber bid)})
122           "t5" (table.insert t5 {:hand hand
    ↪  :bid (tonumber bid)})
123           "t4" (table.insert t4 {:hand hand
    ↪  :bid (tonumber bid)})
124           "t3" (table.insert t3 {:hand hand
    ↪  :bid (tonumber bid)})
125           "t2" (table.insert t2 {:hand hand
    ↪  :bid (tonumber bid)})
126           "t1" (table.insert t1 {:hand hand
    ↪  :bid (tonumber bid)}))))
127     (table.join allhands t1)
128     (table.join allhands t2)
129     (table.join allhands t3)
130     (table.join allhands t4)
131     (table.join allhands t5)
132     (table.join allhands t6)
133     (table.join allhands t7)
134     allhands))
135
136 (fn solve [lines]
```

```
  (let [hands (all-hands lines)]
    (all-wins hands)))

(fn test1 [expected lines]
  (assert (= expected (solve lines))))

(test1 6440
       ["32T3K 765"
        "T55J5 684"
        "KK677 28"
        "KTJJT 220"
        "QQQJA 483"])

(test1 201
       ["AAATK 1"
        "TTTAA 100"])

(test1 10
       ["23456 1"
        "AAAKK 3"
        "AAAAA 1"])

(test1 6592
       ["2345A 1"
        "Q2KJJ 13"
```

```
        "Q2Q2Q 19"
        "T3T3J 17"
        "T3Q33 11"
        "2345J 3"
        "J345A 2"
        "32T3K 5"
        "T55J5 29"
        "KK677 7"
        "KTJJT 34"
        "QQQJA 31"
        "JJJJJ 37"
        "JAAAA 43"
        "AAAAJ 59"
        "AAAAA 61"
        "2AAAA 23"
        "2JJJJ 53"
        "JJJJ2 41"])

(solve (aoc.string-from "2023/07.inp"))
```

```
251545216
```

DONE Day 8.1

You're still riding a camel across Desert Island when you spot a sandstorm quickly approaching. When you turn to warn the Elf, she disappears before your eyes! To be fair, she had just finished warning you about ghosts a few minutes ago.

One of the camel's pouches is labeled ”maps” - sure enough, it's full of documents (your puzzle input) about how to navigate the desert. At least, you're pretty sure that's what they are; one of the documents contains a list of left/right instructions, and the rest of the documents seem to describe some kind of network of labeled nodes.

It seems like you're meant to use the left/right instructions to navigate the network. Perhaps if you have the camel follow the same instructions, you can escape the haunted wasteland!

After examining the maps for a bit, two nodes stick out: AAA and ZZZ. You feel like AAA is where you are now, and you have to follow the left/right instructions until you reach ZZZ.

This format defines each node of the network individually. For example:

```
RL
```

```
AAA = (BBB, CCC)
BBB = (DDD, EEE)
CCC = (ZZZ, GGG)
DDD = (DDD, DDD)
EEE = (EEE, EEE)
GGG = (GGG, GGG)
ZZZ = (ZZZ, ZZZ)
```

Starting with AAA, you need to look up the next element based on the next left/right instruction in your input. In this example, start with AAA and go right (R) by choosing the right element of AAA, CCC. Then, L means to choose the left element of CCC, ZZZ. By following the left/right instructions, you reach ZZZ in 2 steps.

Of course, you might not find ZZZ right away. If you run out of left/right instructions, repeat the whole sequence of instructions as necessary: RL really means RLRLRLRLRLRLRL… and so on. For example, here is a situation that takes 6 steps to reach ZZZ:

```
LLR

AAA = (BBB, BBB)
BBB = (AAA, ZZZ)
```

```
ZZZ = (ZZZ, ZZZ)
```

Starting at AAA, follow the left/right instructions. How many steps are required to reach ZZZ?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-node [line]
  [(string.sub line 1 3)
   (string.sub line 8 10)
   (string.sub line 13 15)])

(fn read-nodes [lines]
  (let [map {}]
    (each [_ line (ipairs lines)]
      (let [[start left right] (read-node line)]
        (tset map start [left right])))
    map))

(fn path-find [map start end path step]
  (if (= end start) step
      (case (string.sub path 1 1)
        "R" (path-find map (. (. map start) 2) end (aoc.string-pushback path) (+ 1 step))
```

```
20          "L" (path-find map (. (. map start) 1)
    ↪  end (aoc.string-pushback path) (+ 1
    ↪  step)))))
21
22  (fn solve [lines path]
23    (let [map (read-nodes lines)]
24      (path-find map "AAA" "ZZZ" path 0)))
25
26  (fn test1 [expected lines path]
27    (assert (= expected (solve lines path))))
28
29  (local test-input-1
30         ["AAA = (BBB, CCC)"
31          "BBB = (DDD, EEE)"
32          "CCC = (ZZZ, GGG)"
33          "DDD = (DDD, DDD)"
34          "EEE = (EEE, EEE)"
35          "GGG = (GGG, GGG)"
36          "ZZZ = (ZZZ, ZZZ)"])
37
38  (test1 2 test-input-1 "RL")
39
40  (local test-input-2
41         ["AAA = (BBB, BBB)"
42          "BBB = (AAA, ZZZ)"
```

```
        "ZZZ = (ZZZ, ZZZ)"])

(test1 6 test-input-2 "LLR")

(local path-input
       "LRLLRRRLRRLRRRLRLRRLLRRLRRLRRLRRRLLR-
 ↪  RLRRLLLRRLLRRRLRRRLRRRLRLRRRLRRRLRLRLRRL-
 ↪  RLRRRLRLRLRRRLLRRRLRLRRLLLRRRLLRRLLLRRRLR-
 ↪  RLLRLRLRRRLRRLLRRLRRRLRRRLLRRRLLRRRLRRLR-
 ↪  RLRLRRRLLLRRRLRRRLRLRRLRRLRRLRRLRRRLRRLR-
 ↪  RRLRRLLRRLRRRLRLLRLLRRRLLRRLRRRLRRRLRRRLR-
 ↪  RRLRRLRRRLLRRLRRRLRRLRRRLRRLRRLRRLRRLRRL-
 ↪  RLRRRR")

(solve (aoc.string-from "2023/08.inp")
 ↪  path-input)
```

```
20093
```

DONE Day 8.2

The sandstorm is upon you and you aren't any closer to escaping the wasteland. You had the camel follow the instructions, but you've barely left your starting position. It's going to take significantly more steps to escape!

What if the map isn't for people - what if the map is for ghosts? Are ghosts even bound by the laws of spacetime? Only one way to find out.

After examining the maps a bit longer, your attention is drawn to a curious fact: the number of nodes with names ending in A is equal to the number ending in Z! If you were a ghost, you'd probably just start at every node that ends with A and follow all of the paths at the same time until they all simultaneously end up at nodes that end with Z.

For example:

```
LR

11A = (11B, XXX)
11B = (XXX, 11Z)
11Z = (11B, XXX)
22A = (22B, XXX)
22B = (22C, 22C)
22C = (22Z, 22Z)
22Z = (22B, 22B)
XXX = (XXX, XXX)
```

Here, there are two starting nodes, 11A and 22A (because they both end with A). As you follow each left/right instruction, use

that instruction to simultaneously navigate away from both nodes you're currently on. Repeat this process until all of the nodes you're currently on end with Z. (If only some of the nodes you're on end with Z, they act like any other node and you continue as normal.) In this example, you would proceed as follows:

- Step 0: You are at 11A and 22A.
- Step 1: You choose all of the left paths, leading you to 11B and 22B.
- Step 2: You choose all of the right paths, leading you to 11Z and 22C.
- Step 3: You choose all of the left paths, leading you to 11B and 22Z.
- Step 4: You choose all of the right paths, leading you to 11Z and 22B.
- Step 5: You choose all of the left paths, leading you to 11B and 22C.
- Step 6: You choose all of the right paths, leading you to 11Z and 22Z.

So, in this example, you end up entirely on nodes that end in Z after 6 steps.

Simultaneously start on every node that ends with A. How many steps does it take before you're only on nodes that end with Z?

```
(fn path-to-any [map start end path step]
  (if (aoc.table-contains? end start) step
      (let [newstart (case (string.sub path 1 1)
                       "R" (. (. map start) 2)
                       "L" (. (. map start) 1))]
        (path-to-any map newstart end (aoc.string-pushback path) (+ 1 step)))))

(fn solve2 [lines path start end]
  (let [map (read-nodes lines)
        paths (lume.map start #(path-to-any map $ end path 0))]
    (lume.reduce paths aoc.math-lcm)))

(fn test2 [expected lines path start end]
  (assert (= expected (solve2 lines path start end))))

(local test-input-p2
```

```
17        ["11A = (11B, XXX)"
18         "11B = (XXX, 11Z)"
19         "11Z = (11B, XXX)"
20         "22A = (22B, XXX)"
21         "22B = (22C, 22C)"
22         "22C = (22Z, 22Z)"
23         "22Z = (22B, 22B)"
24         "XXX = (XXX, XXX)"])
25
26 (test2 6 test-input-p2 "LR"
27        ["11A" "22A"] ["11Z" "22Z"])
28
29 (solve2 (aoc.string-from "2023/08.inp")
    ↪  path-input
30         ["VGA" "AAA" "LHA" "RHA" "CVA" "LDA"]
31         ["BKZ" "KJZ" "XNZ" "XLZ" "PQZ" "ZZZ"])
```

```
22103062509257
```

DONE Day 9.1

You ride the camel through the sandstorm and stop where the ghost's maps told you to stop. The sandstorm subsequently subsides, somehow seeing you standing at an oasis!

The camel goes to get some water and you stretch your neck. As you look up, you discover what must be yet another giant floating island, this one made of metal! That must be where the parts to fix the sand machines come from.

There's even a hang glider partially buried in the sand here; once the sun rises and heats up the sand, you might be able to use the glider and the hot air to get all the way up to the metal island!

While you wait for the sun to rise, you admire the oasis hidden here in the middle of Desert Island. It must have a delicate ecosystem; you might as well take some ecological readings while you wait. Maybe you can report any environmental instabilities you find to someone so the oasis can be around for the next sandstorm-worn traveler.

You pull out your handy Oasis And Sand Instability Sensor and analyze your surroundings. The OASIS produces a report of many values and how they are changing over time (your puzzle input). Each line in the report contains the history of a single value. For example:

```
0 3 6 9 12 15
1 3 6 10 15 21
10 13 16 21 30 45
```

To best protect the oasis, your environmental report should include a prediction of the next value in each history. To do this, start by making a new sequence from the difference at each step of your history. If that sequence is not all zeroes, repeat this process, using the sequence you just generated as the input sequence. Once all of the values in your latest sequence are zeroes, you can extrapolate what the next value of the original history should be.

In the above dataset, the first history is 0 3 6 9 12 15. Because the values increase by 3 each step, the first sequence of differences that you generate will be 3 3 3 3 3. Note that this sequence has one fewer value than the input sequence because at each step it considers two numbers from the input. Since these values aren't all zero, repeat the process: the values differ by 0 at each step, so the next sequence is 0 0 0 0. This means you have enough information to extrapolate the history! Visually, these sequences can be arranged like this:

```
0   3   6   9  12  15
  3   3   3   3   3
    0   0   0   0
```

To extrapolate, start by adding a new zero to the end of your list of zeroes; because the zeroes represent differences between

the two values above them, this also means there is now a placeholder in every sequence above it:

```
0   3   6   9  12  15   B
  3   3   3   3   3   A
    0   0   0   0   0
```

You can then start filling in placeholders from the bottom up. A needs to be the result of increasing 3 (the value to its left) by 0 (the value below it); this means A must be 3:

```
0   3   6   9  12  15   B
  3   3   3   3   3   3
    0   0   0   0   0
```

Finally, you can fill in B, which needs to be the result of increasing 15 (the value to its left) by 3 (the value below it), or 18:

```
0   3   6   9  12  15  18
  3   3   3   3   3   3
    0   0   0   0   0
```

So, the next value of the first history is 18.

Finding all-zero differences for the second history requires an additional sequence:

```
1   3   6  10  15  21
  2   3   4   5   6
    1   1   1   1
      0   0   0
```

Then, following the same process as before, work out the next value in each sequence from the bottom up:

```
1   3   6  10  15  21  28
  2   3   4   5   6   7
    1   1   1   1   1
      0   0   0   0
```

So, the next value of the second history is 28.

The third history requires even more sequences, but its next value can be found the same way:

```
10  13  16  21  30  45  68
   3   3   5   9  15  23
     0   2   4   6   8
       2   2   2   2
         0   0   0
```

So, the next value of the third history is 68.

If you find the next value for each history in this example and add them together, you get 114.

Analyze your OASIS report and extrapolate the next value for each history. What is the sum of these extrapolated values?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn table-zip [t1 t2 f]
  (assert (= (length t1)
             (length t2)))
  (let [result []]
    (for [i 1 (length t1) 1]
      (table.insert result (f (. t1 i) (. t2 i))))
    result))

(fn table.dec [t]
  (let [t1 (aoc.table-range t 2 (length t))
        t2 (aoc.table-range t 1 (- (length t) 1))]
    (table-zip t1 t2 (fn [a b] (- a b)))))

(fn derive [t]
  (var dt t)
  (let [result [dt]]
    (while (not (aoc.table-zero? dt))
      (set dt (table.dec dt))
```

```
      (table.insert result dt))
    result))

(fn extrapolate [t]
  (let [result []]
    (each [_ ti (ipairs t)]
      (table.insert result (aoc.last ti)))
    (aoc.table-sum result)))

(fn read-input [lines]
  (let [input []]
    (each [_ line (ipairs lines)]
      (table.insert input (lume.map
 ↪  (aoc.string-split line " ") #(tonumber
 ↪  $))))
    input))

(fn solve [input]
  (let [xs (read-input input)]
    (aoc.table-sum (lume.map xs #(extrapolate
 ↪  (derive $))))))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(local test-input
       [" 0  3  6  9 12 15"
        " 1  3  6 10 15 21"
        "10 13 16 21 30 45"])

(test 114 test-input)

(solve (aoc.string-from "2023/09.inp"))
```

```
1702218515
```

DONE Day 9.2

Of course, it would be nice to have even more history included in your report. Surely it's safe to just extrapolate backwards as well, right?

For each history, repeat the process of finding differences until the sequence of differences is entirely zero. Then, rather than adding a zero to the end and filling in the next values of each previous sequence, you should instead add a zero to the beginning of your sequence of zeroes, then fill in new first values for each previous sequence.

In particular, here is what the third example history looks like

when extrapolating back in time:

```
5  10  13  16  21  30  45
  5   3   3   5   9  15
   -2   0   2   4   6
      2   2   2   2
        0   0   0
```

Adding the new values on the left side of each sequence from bottom to top eventually reveals the new left-most history value: 5.

Doing this for the remaining example data above results in previous values of -3 for the first history and 0 for the second history. Adding all three new values together produces 2.

Analyze your OASIS report again, this time extrapolating the previous value for each history. What is the sum of these extrapolated values?

```
(fn solve2 [lines]
  (let [input (read-input lines)]
    (aoc.table-sum (lume.map input
    ↪ #(extrapolate (derive (aoc.table-reverse
    ↪ $)))))))

(fn test2 [expected lines]
```

```
6     (assert (= expected (solve2 lines))))
7
8   (test2 2 test-input)
9
10  (solve2 (aoc.string-from "2023/09.inp"))
```

```
925
```

DONE Day 10.1

You use the hang glider to ride the hot air from Desert Island all the way up to the floating metal island. This island is surprisingly cold and there definitely aren't any thermals to glide on, so you leave your hang glider behind.

You wander around for a while, but you don't find any people or animals. However, you do occasionally find signposts labeled ”Hot Springs” pointing in a seemingly consistent direction; maybe you can find someone at the hot springs and ask them where the desert-machine parts are made.

The landscape here is alien; even the flowers and trees are made of metal. As you stop to admire some metal grass, you notice something metallic scurry away in your peripheral vision and jump into a big pipe! It didn't look like any animal you've

ever seen; if you want a better look, you'll need to get ahead of it.

Scanning the area, you discover that the entire field you're standing on is densely packed with pipes; it was hard to tell at first because they're the same metallic silver color as the "ground". You make a quick sketch of all of the surface pipes you can see (your puzzle input).

The pipes are arranged in a two-dimensional grid of tiles:

- | is a vertical pipe connecting north and south.
- - is a horizontal pipe connecting east and west.
- L is a 90-degree bend connecting north and east.
- J is a 90-degree bend connecting north and west.
- 7 is a 90-degree bend connecting south and west.
- F is a 90-degree bend connecting south and east.
- . is ground; there is no pipe in this tile.
- S is the starting position of the animal; there is a pipe on this tile, but your sketch doesn't show what shape the pipe has.

Based on the acoustics of the animal's scurrying, you're confident the pipe that contains the animal is one large, continuous loop.

For example, here is a square loop of pipe:

```
.....
.F-7.
.|.|.
.L-J.
.....
```

If the animal had entered this loop in the northwest corner, the sketch would instead look like this:

```
.....
.S-7.
.|.|.
.L-J.
.....
```

In the above diagram, the S tile is still a 90-degree F bend: you can tell because of how the adjacent pipes connect to it.

Unfortunately, there are also many pipes that aren't connected to the loop! This sketch shows the same loop as above:

```
-L|F7
7S-7|
L|7||
-L-J|
L|-JF
```

In the above diagram, you can still figure out which pipes form the main loop: they're the ones connected to S, pipes those pipes connect to, pipes those pipes connect to, and so on. Every pipe in the main loop connects to its two neighbors (including S, which will have exactly two pipes connecting to it, and which is assumed to connect back to those two pipes).

Here is a sketch that contains a slightly more complex main loop:

```
..F7.
.FJ|.
SJ.L7
|F--J
LJ...
```

Here's the same example sketch with the extra, non-main-loop pipe tiles also shown:

```
7-F7-
.FJ|7
SJLL7
|F--J
LJ.LJ
```

If you want to get out ahead of the animal, you should find the tile in the loop that is farthest from the starting position.

Because the animal is in the pipe, it doesn't make sense to measure this by direct distance. Instead, you need to find the tile that would take the longest number of steps along the loop to reach from the starting point - regardless of which way around the loop the animal went.

In the first example with the square loop:

```
.....
.S-7.
.|.|.
.L-J.
.....
```

You can count the distance each tile in the loop is from the starting point like this:

```
.....
.012.
.1.3.
.234.
.....
```

In this example, the farthest point from the start is 4 steps away.

Here's the more complex loop again:

```
..F7.
.FJ|.
SJ.L7
|F--J
LJ...
```

Here are the distances for each tile on that loop:

```
..45.
.236.
01.78
14567
23...
```

Find the single giant loop starting at S. How many steps along the loop does it take to get from the starting position to the point farthest from the starting position?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local S2N ["|" "F" "7" "S"])
(local N2S ["|" "L" "J" "S"])
(local E2W ["-" "L" "F" "S"])
(local W2E ["-" "7" "J" "S"])

(fn connected? [t x y dir]
```

```
10    (case dir
11      :north (aoc.table-contains? S2N (?. (?. t
      ↪ (- x 1)) y))
12      :east (aoc.table-contains? W2E (?. (?. t
      ↪ x) (+ 1 y)))
13      :south (aoc.table-contains? N2S (?. (?. t
      ↪ (+ x 1)) y))
14      :west (aoc.table-contains? E2W (?. (?. t
      ↪ x) (- y 1)))))
15
16  (fn move [t x y s from]
17    (let [pos (. (. t x) y)]
18      (if (and (< 0 s) (= "S" pos)) (aoc.int (/
      ↪ s 2))
19          (< (* (length t) (length (. t 1))) s)
      ↪ -1
20          (case pos
21            "S" (if (connected? t x y :north)
      ↪ (move t (- x 1) y (+ 1 s) :south)
22                    (connected? t x y :east)
      ↪ (move t x (+ y 1) (+ 1 s) :west)
23                    (connected? t x y :south)
      ↪ (move t (+ x 1) y (+ 1 s) :north)
24                    (connected? t x y :west)
      ↪ (move t x (- y 1) (+ 1 s) :east))
```

```
25            "-" (case from
26                  :west (if (connected? t x y
    ↪  :east) (move t x (+ y 1) (+ 1 s) :west))
27                  :east (if (connected? t x y
    ↪  :west) (move t x (- y 1) (+ 1 s) :east)))
28            "|" (case from
29                  :south (if (connected? t x y
    ↪  :north) (move t (- x 1) y (+ 1 s) :south))
30                  :north (if (connected? t x y
    ↪  :south) (move t (+ x 1) y (+ 1 s)
    ↪  :north)))
31            "F" (case from
32                  :south (if (connected? t x y
    ↪  :east) (move t x (+ y 1) (+ 1 s) :west))
33                  :east (if (connected? t x y
    ↪  :south) (move t (+ x 1) y (+ 1 s)
    ↪  :north)))
34            "L" (case from
35                  :east (if (connected? t x y
    ↪  :north) (move t (- x 1) y (+ 1 s) :south))
36                  :north (if (connected? t x y
    ↪  :east) (move t x (+ y 1) (+ 1 s) :west)))
37            "7" (case from
38                  :west (if (connected? t x y
    ↪  :south) (move t (+ x 1) y (+ 1 s) :north))
```

```
                :south (if (connected? t x y
    ↪  :west) (move t x (- y 1) (+ 1 s) :east)))
          "J" (case from
                :west (if (connected? t x y
    ↪  :north) (move t (- x 1) y (+ 1 s) :south))
                :north (if (connected? t x y
    ↪  :west) (move t x (- y 1) (+ 1 s)
    ↪  :east)))))))

(fn solve [input sx sy]
  (let [m (aoc.read-matrix input)]
    (move m sx sy 0 :start)))

(fn test [expected input sx sy]
  (assert (= expected (solve input sx sy))))

(local test-input1
       ["-L|F7"
        "7S-7|"
        "L|7||"
        "-L-J|"
        "L|-JF"])

(test 4 test-input1 2 2)

```

```
(local test-input2
       ["7-F7-"
        ".FJ|7"
        "SJLL7"
        "|F--J"
        "LJ.LJ"])

(test 8 test-input2 3 1)

(solve (aoc.string-from "2023/10.inp") 64 63)
```

```
7093
```

DONE Day 11.1

You continue following signs for ”Hot Springs” and eventually come across an observatory. The Elf within turns out to be a researcher studying cosmic expansion using the giant telescope here.

He doesn't know anything about the missing machine parts; he's only visiting for this research project. However, he confirms that the hot springs are the next-closest area likely to have people; he'll even take you straight there once he's done with today's observation analysis.

Maybe you can help him with the analysis to speed things up?

The researcher has collected a bunch of data and compiled the data into a single giant image (your puzzle input). The image includes empty space (.) and galaxies (#). For example:

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

The researcher is trying to figure out the sum of the lengths of the shortest path between every pair of galaxies. However, there's a catch: the universe expanded in the time it took the light from those galaxies to reach the observatory.

Due to something involving gravitational effects, only some space expands. In fact, the result is that any rows or columns that contain no galaxies should all actually be twice as big.

In the above example, three columns and two rows contain no

galaxies:

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

These rows and columns need to be twice as big; the result of cosmic expansion therefore looks like this:

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

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

Equipped with this expanded universe, the shortest path between every pair of galaxies can be found. It can help to assign every galaxy a unique number:

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

In these 9 galaxies, there are 36 pairs. Only count each pair once; order within the pair doesn't matter. For each pair, find any shortest path between the two galaxies using only steps that move up, down, left, or right exactly one . or # at a time. (The shortest path between two galaxies is allowed to pass

through another galaxy.)

For example, here is one of the shortest paths between galaxies 5 and 9:

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

This path has length 9 because it takes a minimum of nine steps to get from galaxy 5 to galaxy 9 (the eight locations marked # plus the step onto galaxy 9 itself). Here are some other example shortest path lengths:

- Between galaxy 1 and galaxy 7: 15
- Between galaxy 3 and galaxy 6: 17
- Between galaxy 8 and galaxy 9: 5

In this example, after expanding the universe, the sum of the shortest path between all 36 pairs of galaxies is 374.

Expand the universe, then find the length of the shortest path between every pair of galaxies. What is the sum of these lengths?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn distance [a b]
  (let [[x1 y1] a [x2 y2] b]
    (+ (math.abs (- x1 x2)) (math.abs (- y1
    y2)))))

(fn find-coords [matrix v]
  (let [result []]
    (for [i 1 (length matrix) 1]
      (for [j 1 (length (. matrix i)) 1]
        (when (= v (. (. matrix i) j))
          (table.insert result [i j]))))
    result))

(fn calculate-distances [xs]
  (let [result []]
    (for [i 1 (length xs) 1]
```

```
      (table.insert result
                    (lume.map (aoc.table-range
 ↪  xs i (length xs)) #(distance (. xs i)
 ↪  $))))
    result))

(fn find-blank-rows [matrix blank]
  (let [result []]
    (each [i row (ipairs matrix)]
      (if (lume.all row #(= $ blank))
          (table.insert result i)))
    (aoc.table-reverse result)))

(fn expand-empty-space [matrix]
  (let [rows (find-blank-rows matrix ".")]
    (each [_ row (ipairs rows)]
      (table.insert matrix row (. matrix
 ↪  row))))
  (let [result (aoc.table-transpose matrix)
        rows (find-blank-rows result ".")]
    (each [_ row (ipairs rows)]
      (table.insert result row (. result
 ↪  row)))
    (aoc.table-transpose result)))

```

```
(fn solve [input]
  (let [m (aoc.read-matrix input)
        mm (expand-empty-space m)
        coords (find-coords mm "#")
        dist (calculate-distances coords)]
    (aoc.table-sum dist)))

(fn test [expected input]
  (assert (= expected (solve input))))

(local test-input
       ["...#......"
        ".......#.."
        "#........."
        ".........."
        "......#..."
        ".#........"
        ".........#"
        ".........."
        ".......#.."
        "#...#....."])

(test 374 test-input)

(solve (aoc.string-from "2023/11.inp"))
```

9965032

DONE Day 14.1

You reach the place where all of the mirrors were pointing: a massive parabolic reflector dish attached to the side of another large mountain.

The dish is made up of many small mirrors, but while the mirrors themselves are roughly in the shape of a parabolic reflector dish, each individual mirror seems to be pointing in slightly the wrong direction. If the dish is meant to focus light, all it's doing right now is sending it in a vague direction.

This system must be what provides the energy for the lava! If you focus the reflector dish, maybe you can go where it's pointing and use the light to fix the lava production.

Upon closer inspection, the individual mirrors each appear to be connected via an elaborate system of ropes and pulleys to a large metal platform below the dish. The platform is covered in large rocks of various shapes. Depending on their position, the weight of the rocks deforms the platform, and the shape of the platform controls which ropes move and ultimately the

focus of the dish.

In short: if you move the rocks, you can focus the dish. The platform even has a control panel on the side that lets you tilt it in one of four directions! The rounded rocks (O) will roll when the platform is tilted, while the cube-shaped rocks (#) will stay in place. You note the positions of all of the empty spaces (.) and rocks (your puzzle input). For example:

```
O....#....
O.OO#....#
.....##...
OO.#O....O
.O.....O#.
O.#..O.#.#
..O..#O..O
.......O..
#....###..
#OO..#....
```

Start by tilting the lever so all of the rocks will slide north as far as they will go:

```
OOOO.#.O..
OO..#....#
OO..O##..O
```

```
O..#.OO...
........#.
..#....#.#
..O..#.O.O
..O.......
#....###..
#....#....
```

You notice that the support beams along the north side of the platform are damaged; to ensure the platform doesn't collapse, you should calculate the total load on the north support beams.

The amount of load caused by a single rounded rock (O) is equal to the number of rows from the rock to the south edge of the platform, including the row the rock is on. (Cube-shaped rocks (#) don't contribute to load.) So, the amount of load caused by each rock in each row is as follows:

```
OOOO.#.O.. 10
OO..#....#  9
OO..O##..O  8
O..#.OO...  7
........#.  6
..#....#.#  5
..O..#.O.O  4
```

```
..O.......  3
#....###..  2
#....#....  1
```

The total load is the sum of the load caused by all of the rounded rocks. In this example, the total load is 136.

Tilt the platform so that the rounded rocks all roll north. Afterward, what is the total load on the north support beams?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn table.swap [t i j direction]
  (let [ij (. (. t i) j)]
    (case direction
      :north (when (< 1 i)
               (let [old (aoc.table-replace t (- i 1) j ij)]
                 (aoc.table-replace t i j old)))
      :south (when (< i (length t))
               (let [old (aoc.table-replace t (+ i 1) j ij)]
                 (aoc.table-replace t i j old)))
      :east (when (< j (length (. t i)))
```

```
14                (let [old (aoc.table-replace t
    ↪  i (+ j 1) ij)]
15                  (aoc.table-replace t i j
    ↪  old)))
16      :west (when (< 1 j)
17                (let [old (aoc.table-replace t
    ↪  i (- j 1) ij)]
18                  (aoc.table-replace t i j
    ↪  old)))))
19    t)
20
21  (fn math.wsum [xs]
22    (let [xx (aoc.table-reverse xs)]
23      (accumulate [sum 0 i x (ipairs xx)]
24        (+ sum (* i x)))))
25
26  (fn table.tonumbers [lines]
27    (lume.map lines #(aoc.string-toarray $)))
28
29  (fn tilt-north [matrix]
30    (let [len1 (length matrix)
31          len2 (length (. matrix 1))]
32      (for [i (- len1 1) 1 -1]
33        (for [j len1 (+ i 1) -1]
34          (for [k len2 1 -1]
```

```
35            (when (and (= "O" (. (. matrix j)
    ↪  k))
36                       (= "." (. (. matrix (- j
    ↪  1)) k)))
37              (table.swap matrix j k
    ↪  :north))))))
38    matrix)
39
40  (fn tilt-south [matrix]
41    matrix)
42
43  (fn tilt-east [matrix]
44    matrix)
45
46  (fn tilt-west [matrix]
47    matrix)
48
49  (fn tilt [matrix direction]
50    (case direction
51      :north (tilt-north (tilt-north (tilt-north
    ↪  matrix)))
52      :south (tilt-south matrix)
53      :east (tilt-east matrix)
54      :west (tilt-west matrix)
55      _ matrix))
```

```
56
57 (fn weight [xs]
58   (var count 0)
59   (each [_ x (ipairs xs)]
60     (when (= "O" x)
61       (set count (+ 1 count))))
62   count)
63
64 (fn weights [xs]
65   (lume.map xs #(weight $)))
66
67 (fn solve [lines]
68   (let [input (table.tonumbers lines)
69         matrix (tilt input :north)]
70     (math.wsum (weights matrix))))
71
72 (fn test [expected lines]
73   (assert (= expected (solve lines))))
74
75 (test 136
76       ["O....#...."
77        "O.OO#....#"
78        ".....##..."
79        "OO.#O....O"
80        ".O.....O#."
```

```
        "O.#..O.#.#"
        "..O..#O..O"
        ".......O.."
        "#....###.."
        "#OO..#...."])

(solve (aoc.string-from "2023/14.inp"))
```

```
113456
```

DONE Day 15.1

The newly-focused parabolic reflector dish is sending all of the collected light to a point on the side of yet another mountain - the largest mountain on Lava Island. As you approach the mountain, you find that the light is being collected by the wall of a large facility embedded in the mountainside.

You find a door under a large sign that says "Lava Production Facility" and next to a smaller sign that says "Danger - Personal Protective Equipment required beyond this point".

As you step inside, you are immediately greeted by a somewhat panicked reindeer wearing goggles and a loose-fitting hard hat. The reindeer leads you to a shelf of goggles and hard hats (you

quickly find some that fit) and then further into the facility. At one point, you pass a button with a faint snout mark and the label "PUSH FOR HELP". No wonder you were loaded into that trebuchet so quickly!

You pass through a final set of doors surrounded with even more warning signs and into what must be the room that collects all of the light from outside. As you admire the large assortment of lenses available to further focus the light, the reindeer brings you a book titled "Initialization Manual".

"Hello!", the book cheerfully begins, apparently unaware of the concerned reindeer reading over your shoulder. "This procedure will let you bring the Lava Production Facility online - all without burning or melting anything unintended!"

"Before you begin, please be prepared to use the Holiday ASCII String Helper algorithm (appendix 1A)." You turn to appendix 1A. The reindeer leans closer with interest.

The HASH algorithm is a way to turn any string of characters into a single number in the range 0 to 255. To run the HASH algorithm on a string, start with a current value of 0. Then, for each character in the string starting from the beginning:

- Determine the ASCII code for the current character of the

string.
- Increase the current value by the ASCII code you just determined.
- Set the current value to itself multiplied by 17.
- Set the current value to the remainder of dividing itself by 256.

After following these steps for each character in the string in order, the current value is the output of the HASH algorithm.

So, to find the result of running the HASH algorithm on the string HASH:

- The current value starts at 0.
- The first character is H; its ASCII code is 72.
- The current value increases to 72.
- The current value is multiplied by 17 to become 1224.
- The current value becomes 200 (the remainder of 1224 divided by 256).
- The next character is A; its ASCII code is 65.
- The current value increases to 265.
- The current value is multiplied by 17 to become 4505.
- The current value becomes 153 (the remainder of 4505 divided by 256).

- The next character is S; its ASCII code is 83.
- The current value increases to 236.
- The current value is multiplied by 17 to become 4012.
- The current value becomes 172 (the remainder of 4012 divided by 256).
- The next character is H; its ASCII code is 72.
- The current value increases to 244.
- The current value is multiplied by 17 to become 4148.
- The current value becomes 52 (the remainder of 4148 divided by 256).

So, the result of running the HASH algorithm on the string HASH is 52.

The initialization sequence (your puzzle input) is a comma-separated list of steps to start the Lava Production Facility. Ignore newline characters when parsing the initialization sequence. To verify that your HASH algorithm is working, the book offers the sum of the result of running the HASH algorithm on each step in the initialization sequence.

For example:

```
rn=1,cm-,qp=3,cm=2,qp-,pc=4,ot=9,ab=5,pc-,pc=6,ot=7
```

This initialization sequence specifies 11 individual steps; the

result of running the HASH algorithm on each of the steps is as follows:

- rn=1 becomes 30.
- cm- becomes 253.
- qp=3 becomes 97.
- cm=2 becomes 47.
- qp- becomes 14.
- pc=4 becomes 180.
- ot=9 becomes 9.
- ab=5 becomes 197.
- pc- becomes 48.
- pc=6 becomes 214.
- ot=7 becomes 231.

In this example, the sum of these results is 1320. Unfortunately, the reindeer has stolen the page containing the expected verification number and is currently running around the facility with it excitedly.

Run the HASH algorithm on each step in the initialization sequence. What is the sum of the results? (The initialization sequence is one long line; be careful when copy-pasting it.)

```
(local lume (require :lib.lume))
```

```
(local aoc (require :lib.aoc))

(fn hash [s]
  (var result 0)
  (for [i 1 (length s) 1]
    (set result (% (* 17 (+ result
  ↪  (string.byte s i i))) 256)))
  result)

(fn solve [input]
  (-> (. input 1)
      (aoc.string-split ",")
      (lume.map #(hash $))
      (aoc.table-sum)))

(fn test [expected input]
  (assert (= expected (solve [input]))))

(test 1320
  ↪  "rn=1,cm-,qp=3,cm=2,qp-,pc=4,ot=9,ab=5,pc-,pc=6,ot=7")

(solve (aoc.string-from "2023/15.inp"))
```

```
514281
```

2022 [18/50]

DONE Day 1.1

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

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

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

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

carrying (your puzzle input).

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

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

```
1000
2000
3000

4000

5000
6000

7000
8000
9000

10000
```

This list represents the Calories of the food carried by five

Elves:

- The first Elf is carrying food with 1000, 2000, and 3000 Calories, a total of 6000 Calories.
- The second Elf is carrying one food item with 4000 Calories.
- The third Elf is carrying food with 5000 and 6000 Calories, a total of 11000 Calories.
- The fourth Elf is carrying food with 7000, 8000, and 9000 Calories, a total of 24000 Calories.
- The fifth Elf is carrying one food item with 10000 Calories.

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-input [lines]
  (let [res [[]]]
    (each [_ line (ipairs lines)]
```

```
      (let [num (tonumber line)]
        (if num
            (table.insert (. res (length res))
 ↪ num)
            (table.insert res [])))))
    res))

(fn solve [input]
  (let [xs (read-input input)]
    (aoc.table-max
     (lume.map xs aoc.table-sum))))

(local test-input
       ["1000" "2000" "3000" ""
        "4000" ""
        "5000" "6000" ""
        "7000" "8000" "9000" ""
        "10000"])

(fn test1 [expected input]
  (assert (= expected (solve input))))

(test1 24000 test-input)

(solve (aoc.string-from "2022/01.inp"))
```

```
69310
```

DONE Day 1.2

By the time you calculate the answer to the Elves' question, they've already realized that the Elf carrying the most Calories of food might eventually run out of snacks.

To avoid this unacceptable situation, the Elves would instead like to know the total Calories carried by the top three Elves carrying the most Calories. That way, even if one of those Elves runs out of snacks, they still have two backups.

In the example above, the top three Elves are the fourth Elf (with 24000 Calories), then the third Elf (with 11000 Calories), then the fifth Elf (with 10000 Calories). The sum of the Calories carried by these three elves is 45000.

Find the top three Elves carrying the most Calories. How many Calories are those Elves carrying in total?

```
(fn solve2 [input]
  (let [xs (read-input input)
        res (lume.map xs aoc.table-sum)]
    (table.sort res #(> $1 $2))
```

```
    (aoc.table-sum (aoc.take res 3))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 45000 test-input)

(solve2 (aoc.string-from "2022/01.inp"))
```

```
206104
```

DONE Day 2.1

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

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

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

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

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

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

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

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

This strategy guide predicts and recommends the following:

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-input [lines]
```

```
5     (let [res []]
6       (each [_ line (ipairs lines)]
7         (table.insert res (aoc.string-split line
    ↪  " ")))
8       res))
9
10  (fn score [[i j]]
11    (let [rock 1
12          paper 2
13          scissors 3]
14      (case [i j]
15        [:A :X] (+ rock 3)
16        [:A :Y] (+ paper 6)
17        [:A :Z] (+ scissors 0)
18        [:B :X] (+ rock 0)
19        [:B :Y] (+ paper 3)
20        [:B :Z] (+ scissors 6)
21        [:C :X] (+ rock 6)
22        [:C :Y] (+ paper 0)
23        [:C :Z] (+ scissors 3))))
24
25  (local test-input ["A Y" "B X" "C Z"])
26
27  (fn solve [input]
28    (-> input
```

```
      (read-input)
      (lume.map #(score $))
      (aoc.table-sum)))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 15 test-input)

(solve (aoc.string-from "2022/02.inp"))
```

```
15572
```

DONE Day 2.2

The Elf finishes helping with the tent and sneaks back over to you. ”Anyway, the second column says how the round needs to end: X means you need to lose, Y means you need to end the round in a draw, and Z means you need to win. Good luck!”

The total score is still calculated in the same way, but now you need to figure out what shape to choose so the round ends as indicated. The example above now goes like this:

- In the first round, your opponent will choose Rock (A),

and you need the round to end in a draw (Y), so you also choose Rock. This gives you a score of 1 + 3 = 4.

- In the second round, your opponent will choose Paper (B), and you choose Rock so you lose (X) with a score of 1 + 0 = 1.
- In the third round, you will defeat your opponent's Scissors with Rock for a score of 1 + 6 = 7.

Now that you're correctly decrypting the ultra top secret strategy guide, you would get a total score of 12.

Following the Elf's instructions for the second column, what would your total score be if everything goes exactly according to your strategy guide?

```
(fn score2 [[i j]]
  (let [rock 1 paper 2 scissors 3
        win 6 draw 3 loose 0]
    (case [i j]
      [:A :X] (+ scissors loose)
      [:A :Y] (+ rock draw)
      [:A :Z] (+ paper win)
      [:B :X] (+ rock loose)
      [:B :Y] (+ paper draw)
      [:B :Z] (+ scissors win)
      [:C :X] (+ paper loose)
```

```
      [:C :Y] (+ scissors draw)
      [:C :Z] (+ rock win))))

(local input2 ["A Y" "B X" "C Z"])

(fn solve2 [input]
  (-> input
      (read-input)
      (lume.map #(score2 $))
      (aoc.table-sum)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 12 input2)

(solve2 (aoc.string-from "2022/02.inp"))
```

```
16098
```

DONE Day 3.1

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

now need to be rearranged.

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

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

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

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

```
vJrwpWtwJgWrhcsFMMfFFhFp
jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL
PmmdzqPrVvPwwTWBwg
```

```
wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn
ttgJtRGJQctTZtZT
CrZsJsPPZsGzwwsLwLmpwMDw
```

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

To help prioritize item rearrangement, every item type can be

converted to a priority:

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn items-to-codes [line]
  (let [score {:a 1 :b 2 :c 3 :d 4 :e 5 :f 6
 ↪  :g 7 :h 8 :i 9
               :j 10 :k 11 :l 12 :m 13 :n 14
 ↪  :o 15 :p 16 :q 17
               :r 18 :s 19 :t 20 :u 21 :v 22
 ↪  :w 23 :x 24 :y 25
```

```
8                 :z 26 :A 27 :B 28 :C 29 :D 30
  ↪ :E 31 :F 32 :G 33
9                 :H 34 :I 35 :J 36 :K 37 :L 38
  ↪ :M 39 :N 40 :O 41
10                :P 42 :Q 43 :R 44 :S 45 :T 46
  ↪ :U 47 :V 48 :W 49
11                :X 50 :Y 51 :Z 52}]
12    (lume.map (aoc.string-toarray line) #(.
  ↪ score $))))
13
14 (fn priorities [line]
15   (let [in2 (items-to-codes line)
16         len (length in2)
17         in3 (aoc.table-range in2 1 (aoc.int/
  ↪ len 2))
18         in4 (aoc.table-range in2 (+ 1
  ↪ (aoc.int/ len 2)) len)]
19     (lume.unique (lume.filter in3 (fn [e]
  ↪ (aoc.table-contains? in4 e))))))
20
21 (local test-input
22        ["vJrwpWtwJgWrhcsFMMfFFhFp"
23         "jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL"
24         "PmmdzqPrVvPwwTWBwg"
25         "wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn"
```

```
        "ttgJtRGJQctTZtZT"
        "CrZsJsPPZsGzwwsLwLmpwMDw"])

(fn solve [input]
  (aoc.table-sum (lume.map input #(priorities
  ↪  $))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 157 test-input)

(solve (aoc.string-from "2022/03.inp"))
```

```
8085
```

DONE Day 3.2

As you finish identifying the misplaced items, the Elves come to you with another issue.

For safety, the Elves are divided into groups of three. Every Elf carries a badge that identifies their group. For efficiency, within each group of three Elves, the badge is the only item type carried by all three Elves. That is, if a group's badge is item

type B, then all three Elves will have item type B somewhere in their rucksack, and at most two of the Elves will be carrying any other item type.

The problem is that someone forgot to put this year's updated authenticity sticker on the badges. All of the badges need to be pulled out of the rucksacks so the new authenticity stickers can be attached.

Additionally, nobody wrote down which item type corresponds to each group's badges. The only way to tell which item type is the right one is by finding the one item type that is common between all three Elves in each group.

Every set of three lines in your list corresponds to a single group, but each group can have a different badge item type. So, in the above example, the first group's rucksacks are the first three lines:

```
vJrwpWtwJgWrhcsFMMfFFhFp
jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL
PmmdzqPrVvPwwTWBwg
```

And the second group's rucksacks are the next three lines:

```
wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn
ttgJtRGJQctTZtZT
```

```
CrZsJsPPZsGzwwsLwLmpwMDw
```

In the first group, the only item type that appears in all three rucksacks is lowercase r; this must be their badges. In the second group, their badge item type must be Z.

Priorities for these items must still be found to organize the sticker attachment efforts: here, they are 18 (r) for the first group and 52 (Z) for the second group. The sum of these is 70.

Find the item type that corresponds to the badges of each three-Elf group. What is the sum of the priorities of those item types?

```
(fn priorities2 [l1 l2 l3]
  (let [c1 (items-to-codes l1)
        c2 (items-to-codes l2)
        c3 (items-to-codes l3)
        common (lume.filter c2 (fn [e]
 ↪  (aoc.table-contains? c3 e)))]
    (lume.unique (lume.filter c1 (fn [e]
 ↪  (aoc.table-contains? common e))))))

(fn solve2 [input]
  (aoc.table-sum
```

```
10    (lume.map (aoc.table-group-by input 3)
11              #(priorities2 (aoc.table-unpack
    ↪  $)))))
12
13 (fn test2 [expected input]
14   (assert (= expected (solve2 input))))
15
16 (test2 70 test-input)
17
18 (solve2 (aoc.string-from "2022/03.inp"))
```

```
2515
```

DONE Day 4.1

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

However, as some of the Elves compare their section assignments with each other, they've noticed that many of the assignments overlap. To try to quickly find overlaps and reduce

duplicated effort, the Elves pair up and make a big list of the section assignments for each pair (your puzzle input).

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

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

For the first few pairs, this list means:

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

This example list uses single-digit section IDs to make it easier to draw; your actual list might contain larger numbers. Visually,

these pairs of section assignments look like this:

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

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

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

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

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

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

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

example, there are 2 such pairs.

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn find-subrange [line]
  (let [[elf1 elf2] (aoc.string-split line ",")
        [f1 t1] (aoc.string-split elf1 "-")
        [f2 t2] (aoc.string-split elf2 "-")]
    (or (and (<= (aoc.int f1) (aoc.int f2)) (>= (aoc.int t1) (aoc.int t2)))
        (and (>= (aoc.int f1) (aoc.int f2)) (<= (aoc.int t1) (aoc.int t2))))))

(local test-input
       ["2-4,6-8"
        "2-3,4-5"
        "5-7,7-9"
        "2-8,3-7"
        "6-6,4-6"
        "2-6,4-8"])

```

```
(fn solve [input]
  (-> input
      (lume.map find-subrange)
      (lume.count #(not= false $))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 2 test-input)

(solve (aoc.string-from "2022/04.inp"))
```

```
536
```

DONE Day 4.2

It seems like there is still quite a bit of duplicate work planned. Instead, the Elves would like to know the number of pairs that overlap at all.

In the above example, the first two pairs (2-4,6-8 and 2-3,4-5) don't overlap, while the remaining four pairs (5-7,7-9, 2-8,3-7, 6-6,4-6, and 2-6,4-8) do overlap:

- 5-7,7-9 overlaps in a single section, 7.

- 2-8,3-7 overlaps all of the sections 3 through 7.
- 6-6,4-6 overlaps in a single section, 6.
- 2-6,4-8 overlaps in sections 4, 5, and 6.

So, in this example, the number of overlapping assignment pairs is 4. In how many assignment pairs do the ranges overlap?

```
(fn overlapping-pairs [line]
  (let [[elf1 elf2] (aoc.string-split line
  ↪  ",")
        [f1 t1] (aoc.string-split elf1 "-")
        [f2 t2] (aoc.string-split elf2 "-")]
    (or (and (<= (aoc.int f1) (aoc.int f2))
  ↪  (>= (aoc.int t1) (aoc.int f2)))
        (and (<= (aoc.int f1) (aoc.int t2))
  ↪  (>= (aoc.int t1) (aoc.int t2)))
        (and (<= (aoc.int f2) (aoc.int f1))
  ↪  (>= (aoc.int t2) (aoc.int f1)))
        (and (<= (aoc.int f2) (aoc.int t1))
  ↪  (>= (aoc.int t2) (aoc.int t1))))))

(fn solve2 [input]
  (-> input
      (lume.map overlapping-pairs)
      (lume.count #(not= false $))))
```

```
14
15 (fn test2 [expected input]
16   (assert (= expected (solve2 input))))
17
18 (test2 4 test-input)
19
20 (solve2 (aoc.string-from "2022/04.inp"))
```

```
845
```

DONE Day 5.1

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

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

The Elves don't want to interrupt the crane operator during this delicate procedure, but they forgot to ask her which crate will

end up where, and they want to be ready to unload them as soon as possible so they can embark.

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

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

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

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

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

in this configuration:

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

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

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

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

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

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

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(macro times [t body1 & rest-body]
  `(fcollect [i# 1 ,t 1]
     (do ,body1 ,(unpack rest-body))))

(fn test-input []
  (let [crates [["N" "Z"] ["D" "C" "M"] ["P"]]
        moves [[1 2 1] [3 1 3] [2 2 1] [1 1
 ↪  2]]]
    (each [_ [n f t] (ipairs moves)]
```

```
      (times n (aoc.table-move 1 (. crates f)
    ↪  (. crates t))))
    (assert (= "CMZ"
               (aoc.table-tostring (.
    ↪  (aoc.table-transpose crates) 1))))))

(test-input)

(fn scan-crates [lines]
  (let [in (lume.map (aoc.table-range lines 1
    ↪  8) #(aoc.string-toarray $))
        loc [2 6 10 14 18 22 26 30 34]
        res []]
    (each [i v (ipairs loc)]
      (table.insert res i
                    (lume.filter
                     [(. (. in 1) v) (. (. in
    ↪  2) v) (. (. in 3) v)
                      (. (. in 4) v) (. (. in
    ↪  5) v) (. (. in 6) v)
                      (. (. in 7) v) (. (. in
    ↪  8) v)]
                    #(not= $ " "))))
    res))

```

```
(fn scan-moves [lines]
  (lume.map (aoc.table-range lines 11 (length
 ↪  lines))
            #(aoc.string-tonumarray $)))

(fn solve [lines]
  (let [crates (scan-crates lines)
        moves (scan-moves lines)]
    (each [_ [n f t] (ipairs moves)]
      (times n (aoc.table-move 1 (. crates f)
 ↪  (. crates t))))
    (aoc.table-tostring (.
 ↪  (aoc.table-transpose crates) 1))))

(solve (aoc.string-from "2022/05.inp"))
```

```
GFTNRBZPF
```

DONE Day 5.2

As you watch the crane operator expertly rearrange the crates, you notice the process isn't following your prediction.

Some mud was covering the writing on the side of the crane, and you quickly wipe it away. The crane isn't a CrateMover

9000 - it's a CrateMover 9001.

The CrateMover 9001 is notable for many new and exciting features: air conditioning, leather seats, an extra cup holder, and the ability to pick up and move multiple crates at once.

Again considering the example above, the crates begin in the same configuration:

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

Moving a single crate from stack 2 to stack 1 behaves the same as before:

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

However, the action of moving three crates from stack 1 to stack 3 means that those three moved crates stay in the same order, resulting in this new configuration:

```
        [D]
        [N]
```

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

Next, as both crates are moved from stack 2 to stack 1, they retain their order as well:

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

Finally, a single crate is still moved from stack 1 to stack 2, but now it's crate C that gets moved:

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

In this example, the CrateMover 9001 has put the crates in a totally different order: MCD.

Before the rearrangement process finishes, update your simulation so that the Elves know where they should stand to be ready

to unload the final supplies. After the rearrangement procedure completes, what crate ends up on top of each stack?

```
(fn test-input-p2 []
  (let [crates [["N" "Z"] ["D" "C" "M"] ["P"]]
        moves [[1 2 1] [3 1 3] [2 2 1] [1 1
↪  2]]]
    (each [_ [n f t] (ipairs moves)]
      (aoc.table-move 1 (. crates f) (. crates
↪  t) n))
    (assert (= "MCD"
               (aoc.table-tostring (.
↪  (aoc.table-transpose crates) 1))))))

(test-input-p2)

(fn solve2 [lines]
  (let [crates (scan-crates lines)
        moves (scan-moves lines)]
    (each [_ [n f t] (ipairs moves)]
      (aoc.table-move 1 (. crates f) (. crates
↪  t) n))
    (aoc.table-tostring (.
↪  (aoc.table-transpose crates) 1))))

(solve2 (aoc.string-from "2022/05.inp"))
```

VRQWPDSGP

DONE Day 6.1

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

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

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

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

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

To fix the communication system, you need to add a subroutine to the device that detects a start-of-packet marker in the datastream. In the protocol being used by the Elves, the start of a packet is indicated by a sequence of four characters that are all different.

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

For example, suppose you receive the following datastream buffer:

```
mjqjpqmgbljsphdztnvjfqwrcgsmlb
```

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

The first time a marker appears is after the seventh character arrives. Once it does, the last four characters received are jpqm,

which are all different. In this case, your subroutine should report the value 7, because the first start-of-packet marker is complete after 7 characters have been processed.

Here are a few more examples:

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn find-packet-marker [pos stream]
  (case (aoc.table-range stream (- pos 3) pos)
    (where [a b c d] (and (not= a b) (not= a
    ↪  c) (not= a d)
```

```
                            (not= b c) (not= b
    ↪ d) (not= c d))) pos
      [_a _b _c _d] (find-packet-marker (+ 1
    ↪ pos) stream)))

(fn solve [input]
  (find-packet-marker 4 (aoc.string-toarray (.
    ↪ input 1))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 7 ["mjqjpqmgbljsphdztnvjfqwrcgsmlb"])
(test 5 ["bvwbjplbgvbhsrlpgdmjqwftvncz"])
(test 6 ["nppdvjthqldpwncqszvftbrmjlhg"])
(test 10
    ↪ ["nznrnfrfntjfmvfwmzdfjlvtqnbhcprsg"])
(test 11 ["zcfzfwzzqfrljwzlrfnpqdbhtmscgvjw"])

(solve (aoc.string-from "2022/06.inp"))
```

```
1538
```

DONE Day 6.2

Your device's communication system is correctly detecting packets, but still isn't working. It looks like it also needs to look for messages.

A start-of-message marker is just like a start-of-packet marker, except it consists of 14 distinct characters rather than 4.

Here are the first positions of start-of-message markers for all of the above examples:

- mjqjpqmgbljsphdztnvjfqwrcgsmlb: first marker after character 19
- bvwbjplbgvbhsrlpgdmjqwftvncz: first marker after character 23
- nppdvjthqldpwncqszvftbrmjlhg: first marker after character 23
- nznrnfrfntjfmvfwmzdfjlvtqnbhcprsg: first marker after character 29
- zcfzfwzzqfrljwzlrfnpqdbhtmscgvjw: first marker after character 26

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

```
(fn find-message-marker [pos stream]
  (let [start-message (aoc.table-range stream
  ↪  (- pos 13) pos)]
    (if (= (length start-message)
           (length (lume.unique
  ↪  start-message))) pos
           (find-message-marker (+ 1 pos)
  ↪  stream))))

(fn solve2 [input]
  (find-message-marker 14 (aoc.string-toarray
  ↪  (. input 1))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 19 ["mjqjpqmgbljsphdztnvjfqwrcgsmlb"])
(test2 23 ["bvwbjplbgvbhsrlpgdmjqwftvncz"])
(test2 23 ["nppdvjthqldpwncqszvftbrmjlhg"])
(test2 29
  ↪  ["nznrnfrfntjfmvfwmzdfjlvtqnbhcprsg"])
(test2 26
  ↪  ["zcfzfwzzqfrljwzlrfnpqdbhtmscgvjw"])

(solve2 (aoc.string-from "2022/06.inp"))
```

```
2315
```

DONE Day 7.1

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

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

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

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

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

```
$ cd /
$ ls
dir a
```

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

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

contents of the directory you're currently in.

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

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

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

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

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

Since the disk is full, your first step should probably be to find directories that are good candidates for deletion. To do this, you need to determine the total size of each directory. The total size of a directory is the sum of the sizes of the files it contains,

directly or indirectly. (Directories themselves do not count as having any intrinsic size.)

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

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
```

```
3
4  (fn push [xs x]
5    (table.insert xs x)
6    xs)
7
8  (fn pop [xs]
9    (table.remove xs (length xs))
10   xs)
11
12 (fn ncdu [fs pwd s]
13   (let [name (table.concat pwd)
14         size (tonumber s)]
15     (tset fs name
16           (+ (or (. fs name) 0) size)))
17   (while (not= 0 (length (do (table.remove
     ↪  pwd) pwd)))
18     (ncdu fs pwd s)))
19
20 (fn read [lines]
21   (let [fs {}
22         pwd []]
23     (each [_ line (ipairs lines)]
24       (let [tokens (aoc.string-split line "
     ↪  ")]
25         (case tokens
```

```
            ["$" "cd" ".."] (pop pwd)
            ["$" "cd" "/"] (push pwd "/")
            ["$" "cd" x] (push pwd (.. x "/"))
            ["$" "ls"] nil
            ["dir" d] nil
            [s n] (ncdu fs (aoc.table-clone pwd)
    ↪  s))))
      fs))

(local test-input
       ["$ cd /"
        "$ ls"
        "dir a"
        "14848514 b.txt"
        "8504156 c.dat"
        "dir d"
        "$ cd a"
        "$ ls"
        "dir e"
        "29116 f"
        "2557 g"
        "62596 h.lst"
        "$ cd e"
        "$ ls"
        "584 i"
```

```
50          "$ cd .."
51          "$ cd .."
52          "$ cd d"
53          "$ ls"
54          "4060174 j"
55          "8033020 d.log"
56          "5626152 d.ext"
57          "7214296 k"])
58
59  (fn size [fs s]
60    (lume.reduce
61     (lume.filter fs
62                  (fn [x] (<= x s)))
63     (fn [a x] (+ a x))))
64
65  (fn path [fs p]
66    (let [keys (lume.filter (lume.keys fs)
67                            (fn [k]
    ↪  (aoc.string-starts-with k p)))]
68      (aoc.fold (lume.map keys #(. fs $)))))
69
70  (fn solve [input]
71    (let [fs (read input)]
72      (size fs 100000)))
73
```

```
(fn test [expected input]
  (assert (= expected (solve input))))

(test 95437 test-input)

(solve (aoc.string-from "2022/07.inp"))
```

```
1501149
```

DONE Day 7.2

Now, you're ready to choose a directory to delete.

The total disk space available to the filesystem is 70000000. To run the update, you need unused space of at least 30000000. You need to find a directory you can delete that will free up enough space to run the update.

In the example above, the total size of the outermost directory (and thus the total amount of used space) is 48381165; this means that the size of the unused space must currently be 21618835, which isn't quite the 30000000 required by the update. Therefore, the update still requires a directory with total size of at least 8381165 to be deleted before it can run.

To achieve this, you have the following options:

- Delete directory e, which would increase unused space by 584.
- Delete directory a, which would increase unused space by 94853.
- Delete directory d, which would increase unused space by 24933642.
- Delete directory /, which would increase unused space by 48381165.

Directories e and a are both too small; deleting them would not free up enough space. However, directories d and / are both big enough! Between these, choose the smallest: d, increasing unused space by 24933642.

Find the smallest directory that, if deleted, would free up enough space on the filesystem to run the update. What is the total size of that directory?

```
(fn solve2 [input]
  (let [fs (read input)
        required 30000000
        total 70000000
        available (- total (. fs "/"))
        minimum (- required available)]
    (aoc.table-min
```

```
      (lume.map
       (lume.filter
        (lume.keys fs)
        (fn [x] (>= (. fs x) minimum)))
       (fn [x] (. fs x))))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 24933642 test-input)

(solve2 (aoc.string-from "2022/07.inp"))
```

```
10096985
```

DONE Day 8.1

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

First, determine whether there is enough tree cover here to keep a tree house hidden. To do this, you need to count the

number of trees that are visible from outside the grid when looking directly along a row or column.

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

```
30373
25512
65332
33549
35390
```

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

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

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

- The top-left 5 is visible from the left and top. (It isn't

visible from the right or bottom since other trees of height 5 are in the way.)

- The top-middle 5 is visible from the top and right.
- The top-right 1 is not visible from any direction; for it to be visible, there would need to only be trees of height 0 between it and an edge.
- The left-middle 5 is visible, but only from the right.
- The center 3 is not visible from any direction; for it to be visible, there would need to be only trees of at most height 2 between it and an edge.
- The right-middle 3 is visible from the right.
- In the bottom row, the middle 5 is visible, but the 3 and 4 are not.

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn find-visible-trees [m]
  (let [res []
```

```
6           M (aoc.table-transpose m)
7           leni (length m)]
8       (for [i 1 leni]
9         (let [xi (. m i)
10              lenj (length xi)]
11          (for [j 1 lenj]
12            (let [xj (. xi j)
13                  XI (. M j)
14                  ltxj (fn [e] (< e xj))]
15              (when (or (= 1 i) (= 1 j) (= leni
    ↪  i) (= lenj j)
16                        (lume.all
    ↪  (aoc.table-range xi 1 (- j 1)) ltxj)
17                        (lume.all
    ↪  (aoc.table-range xi (+ 1 j) lenj) ltxj)
18                        (lume.all
    ↪  (aoc.table-range XI 1 (- i 1)) ltxj)
19                        (lume.all
    ↪  (aoc.table-range XI (+ 1 i) leni) ltxj))
20                (table.insert res (..  i ","
    ↪  j))))))))
21      res))
22
23  (fn solve [input]
24    (let [matrix (aoc.read-matrix input true)
```

```
        res (find-visible-trees matrix)]
    (length res)))

(fn test [expected input]
  (assert (= expected (solve input))))

(local test-input
       ["30373"
        "25512"
        "65332"
        "33549"
        "35390"])

(test 21 test-input)

(solve (aoc.string-from "2022/08.inp"))
```

```
1835
```

DONE Day 8.2

Content with the amount of tree cover available, the Elves just need to know the best spot to build their tree house: they would like to be able to see a lot of trees.

To measure the viewing distance from a given tree, look up, down, left, and right from that tree; stop if you reach an edge or at the first tree that is the same height or taller than the tree under consideration. (If a tree is right on the edge, at least one of its viewing distances will be zero.)

The Elves don't care about distant trees taller than those found by the rules above; the proposed tree house has large eaves to keep it dry, so they wouldn't be able to see higher than the tree house anyway.

In the example above, consider the middle 5 in the second row:

```
30373
25512
65332
33549
35390
```

- Looking up, its view is not blocked; it can see 1 tree (of height 3).
- Looking left, its view is blocked immediately; it can see only 1 tree (of height 5, right next to it).
- Looking right, its view is not blocked; it can see 2 trees.

- Looking down, its view is blocked eventually; it can see 2 trees (one of height 3, then the tree of height 5 that blocks its view).

A tree's scenic score is found by multiplying together its viewing distance in each of the four directions. For this tree, this is 4 (found by multiplying 1 * 1 * 2 * 2).

However, you can do even better: consider the tree of height 5 in the middle of the fourth row:

```
30373
25512
65332
33549
35390
```

- Looking up, its view is blocked at 2 trees (by another tree with a height of 5).
- Looking left, its view is not blocked; it can see 2 trees.
- Looking down, its view is also not blocked; it can see 1 tree.
- Looking right, its view is blocked at 2 trees (by a massive tree of height 9).

This tree's scenic score is 8 (2 * 2 * 1 * 2); this is the ideal spot

for the tree house.

Consider each tree on your map. What is the highest scenic score possible for any tree?

```
(fn count-trees [xs e]
  (var res 0)
  (for [i 1 (length xs) &until (<= e (. xs
  ↪  i))]
    (set res i))
  (if (< res (length xs)) (+ 1 res)
      res))

(fn find-scenic-score [m]
  (let [res []
        M (aoc.table-transpose m)
        leni (length m)]
    (for [i 1 leni]
      (let [xi (. m i)
            lenj (length xi)]
        (for [j 1 lenj]
          (let [xj (. xi j)
                XI (. M j)
                down (aoc.table-range XI (+ i
  ↪  1) leni)
```

```
19                  up (aoc.table-reverse
    ↪  (aoc.table-range XI 1 (- i 1)))
20                  left (aoc.table-reverse
    ↪  (aoc.table-range xi 1 (- j 1)))
21                  right (aoc.table-range xi (+ j
    ↪  1) lenj)]
22              (let [score (* (count-trees down
    ↪  xj)
23                             (count-trees up xj)
24                             (count-trees left
    ↪  xj)
25                             (count-trees right
    ↪  xj))]
26                (when (< 0 score)
27                  (table.insert res score)))))))
28      (aoc.table-max res)))
29
30  (fn solve2 [input]
31    (let [matrix (aoc.read-matrix input true)]
32      (find-scenic-score matrix)))
33
34  (fn test2 [expected input]
35    (assert (= expected (solve2 input))))
36
37  (test2 8 test-input)
```

```
38
39 (solve2 (aoc.string-from "2022/08.inp"))
```

```
263670
```

DONE Day 9.1

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

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

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

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

Due to nebulous reasoning involving Planck lengths, you should be able to model the positions of the knots on a two-dimensional grid. Then, by following a hypothetical series

of motions (your puzzle input) for the head, you can determine how the tail will move.

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

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

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

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

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

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

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

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

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

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

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

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

the same position, overlapping.

For example:

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

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

```
== Initial State ==

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

```
== R 4 ==

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

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

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

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

```
s..TH.

== U 4 ==

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

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

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

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

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

== L 3 ==

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

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

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

== D 1 ==
```

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

== R 4 ==

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

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

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

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

== D 1 ==

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

== L 5 ==

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

......
....T.
```

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

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

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

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

== R 2 ==

......
```

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

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

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

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(macro times [t body1 & rest-body]
  `(fcollect [i# 1 ,t 1]
     (do ,body1 ,(unpack rest-body))))

(fn move [{:x Sx :y Sy} [Hx Hy]]
  [(+ Sx Hx) (+ Sy Hy)])

(fn make-move [lines]
  (let [cur {:x 0 :y 0}
        res [[0 0]]]
    (each [_ line (ipairs lines)]
      (let [pos (move cur line)]
        (table.insert res pos)
        (tset cur :x (. pos 1))
        (tset cur :y (. pos 2))))
    res))

(fn read-input [lines]
  (let [path []]
    (each [_ line (ipairs lines)]
      (case (aoc.string-split line " ")
        ["R" dx] (let [ndx (tonumber dx)]
```

```
26                      (times ndx (table.insert
    ↪  path [1 0])))
27          ["U" dy] (let [ndy (tonumber dy)]
28                      (times ndy (table.insert
    ↪  path [0 1])))
29          ["D" Dy] (let [nDy (tonumber Dy)]
30                      (times nDy (table.insert
    ↪  path [0 -1])))
31          ["L" Dx] (let [nDx (tonumber Dx)]
32                      (times nDx (table.insert
    ↪  path [-1 0])))))
33      path))
34
35  (fn match-move [moves]
36    (let [t {:x 0 :y 0}
37          res []]
38      (each [_ h (ipairs moves)]
39        (when (< 2 (aoc.dist2rd h t))
40          (do
41            (table.insert res [(. t :x) (. t
    ↪  :y)])
42            (tset t :x (+ (. t :x) (math.max -1
    ↪  (math.min 1 (- (. h 1) (. t :x))))))
43            (tset t :y (+ (. t :y) (math.max -1
    ↪  (math.min 1 (- (. h 2) (. t :y)))))))))))
```

```
    (table.insert res [(. t :x) (. t :y)])
    res))

(fn solve [input]
  (-> input
      (read-input)
      (make-move)
      (match-move)
      (aoc.table-unique)
      (length)))

(fn test [expected input]
  (assert (= expected (solve input))))

(local test-input
       ["R 4"
        "U 4"
        "L 3"
        "D 1"
        "R 4"
        "D 1"
        "L 5"
        "R 2"])

(test 13 test-input)
```

```
69
70 (solve (aoc.string-from "2022/09.inp"))
```

```
6367
```

DONE Day 9.2

A rope snaps! Suddenly, the river is getting a lot closer than you remember. The bridge is still there, but some of the ropes that broke are now whipping toward you as you fall through the air!

The ropes are moving too quickly to grab; you only have a few seconds to choose how to arch your body to avoid being hit. Fortunately, your simulation can be extended to support longer ropes.

Rather than two knots, you now must simulate a rope consisting of ten knots. One knot is still the head of the rope and moves according to the series of motions. Each knot further down the rope follows the knot in front of it using the same rules as before.

Using the same series of motions as the above example, but with the knots marked H, 1, 2, …, 9, the motions now occur as

follows:

```
== Initial State ==

......
......
......
......
H.....  (H covers 1, 2, 3, 4, 5, 6, 7, 8, 9,
 ↪  s)

== R 4 ==

......
......
......
......
1H....  (1 covers 2, 3, 4, 5, 6, 7, 8, 9, s)

......
......
......
......
21H...  (2 covers 3, 4, 5, 6, 7, 8, 9, s)

......
```

```
......
......
......
321H..  (3 covers 4, 5, 6, 7, 8, 9, s)

......
......
......
......
4321H.  (4 covers 5, 6, 7, 8, 9, s)

== U 4 ==

......
......
......
....H.
4321..  (4 covers 5, 6, 7, 8, 9, s)

......
......
....H.
.4321.
5.....  (5 covers 6, 7, 8, 9, s)
```

```
......
....H.
....1.
.432..
5.....  (5 covers 6, 7, 8, 9, s)

....H.
....1.
..432.
.5....
6.....  (6 covers 7, 8, 9, s)

== L 3 ==

...H..
....1.
..432.
.5....
6.....  (6 covers 7, 8, 9, s)

..H1..
...2..
..43..
.5....
6.....  (6 covers 7, 8, 9, s)
```

```
.H1...
...2..
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

== D 1 ==

..1...
.H.2..
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

== R 4 ==

..1...
..H2..
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

..1...
...H..  (H covers 2)
```

```
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

......
...1H.  (1 covers 2)
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

......
...21H
..43..
.5....
6.....  (6 covers 7, 8, 9, s)

== D 1 ==

......
...21.
..43.H
.5....
6.....  (6 covers 7, 8, 9, s)

== L 5 ==
```

```
......
...21.
..43H.
.5....
6.....  (6 covers 7, 8, 9, s)

......
...21.
..4H..  (H covers 3)
.5....
6.....  (6 covers 7, 8, 9, s)

......
...2..
..H1..  (H covers 4; 1 covers 3)
.5....
6.....  (6 covers 7, 8, 9, s)

......
...2..
.H13..  (1 covers 4)
.5....
6.....  (6 covers 7, 8, 9, s)
```

```
......
......
H123..  (2 covers 4)
.5....
6.....  (6 covers 7, 8, 9, s)

== R 2 ==

......
......
.H23..  (H covers 1; 2 covers 4)
.5....
6.....  (6 covers 7, 8, 9, s)

......
......
.1H3..  (H covers 2, 4)
.5....
6.....  (6 covers 7, 8, 9, s)
```

Now, you need to keep track of the positions the new tail, 9, visits. In this example, the tail never moves, and so it only visits 1 position. However, be careful: more types of motion are possible than before, so you might want to visually compare your simulated rope to the one above.

Here's a larger example:

```
R 5
U 8
L 8
D 3
R 17
D 10
L 25
U 20
```

These motions occur as follows (individual steps are not shown):

```
== Initial State ==

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

```
.........................
.........................
.........................
.........................
..........H..............  (H covers 1, 2, 3,
↪  4, 5, 6, 7, 8, 9, s)
.........................
.........................
.........................
.........................
.........................

== R 5 ==

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

```
........................
........................
........................
........................
..........54321H.........  (5 covers 6, 7, 8,
↪  9, s)
........................
........................
........................
........................
........................

== U 8 ==

........................
........................
........................
........................
........................
........................
........................
................H.......
................1.......
................2.......
................3.......
```

```
..............54.........
..............6...........
.............7............
............8.............
...........9.............  (9 covers s)
........................
........................
........................
........................
........................

== L 8 ==

........................
........................
........................
........................
........................
........................
........................
........H1234.............
............5.............
............6.............
............7.............
............8.............
```

```
............9.............
........................
........................
...........s............
........................
........................
........................
........................
........................
```

```
== D 3 ==
```

```
........................
........................
........................
........................
........................
........................
........................
........................
.........2345...........
........1...6...........
........H...7...........
............8...........
............9...........
```

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

== R 17 ==

.........................
.........................
.........................
.........................
.........................
.........................
.........................
.........................
.........................
.........................
.............987654321H
.........................
.........................
.........................
```

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

```
== D 10 ==
```

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

```
..........s.........98765
.........................4
.........................3
.........................2
.........................1
.........................H

== L 25 ==

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

```
.........................
.........................
.........................
.........................
H123456789................

== U 20 ==

H........................
1........................
2........................
3........................
4........................
5........................
6........................
7........................
8........................
9........................
.........................
.........................
.........................
.........................
.........................
..........s..............
.........................
```

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

Now, the tail (9) visits 36 positions (including s) at least once:

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

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

Simulate your complete series of motions on a larger rope with ten knots. How many positions does the tail of the rope visit at least once?

```
(fn solve2 [input]
  (let [moves (read-input input)
        head (make-move moves)
        T1 (match-move head)
        T2 (match-move T1)
        T3 (match-move T2)
        T4 (match-move T3)
        T5 (match-move T4)
        T6 (match-move T5)
        T7 (match-move T6)
        T8 (match-move T7)]
    (length (aoc.table-unique (match-move T8)))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(local test2-input
       ["R 5"
```

```
        "U 8"
        "L 8"
        "D 3"
        "R 17"
        "D 10"
        "L 25"
        "U 20"])

(test2 36 test2-input)

(solve2 (aoc.string-from "2022/09.inp"))
```

```
2536
```

2021 [18/50]

DONE Day 1.1

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

Before you know it, you're inside a submarine the Elves keep

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

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

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

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

For example, suppose you had the following report:

```
199
200
208
210
```

```
200
207
240
269
260
263
```

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

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

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

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

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

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

How many measurements are larger than the previous measurement?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [lines]
  (let [t (aoc.table-zip
            (aoc.table-range lines 1 (- (length lines) 1))
            (aoc.table-range lines 2 (length lines)))]
    (length (lume.filter t (fn [[f t]] (< (tonumber f) (tonumber t)))))))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(local test-input ["199"
                   "200"
                   "208"
                   "210"
                   "200"
                   "207"
                   "240"
                   "269"
                   "260"
                   "263"])

(test 7 test-input)

(solve (aoc.string-from "2021/01.inp"))
```

```
1400
```

DONE Day 1.2

Considering every single measurement isn't as useful as you expected: there's just too much noise in the data.

Instead, consider sums of a three-measurement sliding window. Again considering the above example:

```
199  A
```

```
200  A B
208  A B C
210    B C D
200  E   C D
207  E F   D
240  E F G
269    F G H
260      G H
263        H
```

Start by comparing the first and second three-measurement windows. The measurements in the first window are marked A (199, 200, 208); their sum is 199 + 200 + 208 = 607. The second window is marked B (200, 208, 210); its sum is 618. The sum of measurements in the second window is larger than the sum of the first, so this first comparison increased.

Your goal now is to count the number of times the sum of measurements in this sliding window increases from the previous sum. So, compare A with B, then compare B with C, then C with D, and so on. Stop when there aren't enough measurements left to create a new three-measurement sum.

In the above example, the sum of each three-measurement window is as follows:

```
A: 607 (N/A - no previous sum)
B: 618 (increased)
C: 618 (no change)
D: 617 (decreased)
E: 647 (increased)
F: 716 (increased)
G: 769 (increased)
H: 792 (increased)
```

In this example, there are 5 sums that are larger than the previous sum.

Consider sums of a three-measurement sliding window. How many sums are larger than the previous sum?

```
(fn solve2 [lines]
  (let [res []]
    (for [i 3 (length lines)]
      (table.insert res (+ (. lines (- i 2))
                           (. lines (- i 1))
                           (. lines i))))
    (solve res)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 5 test-input)
```

```
13
14 (solve2 (aoc.string-from "2021/01.inp"))
```

```
1429
```

DONE Day 2.1

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

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

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

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

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

```
forward 5
down 5
```

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

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

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

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

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

```
(local lume (require :lib.lume))
```

```
(local aoc (require :lib.aoc))

(fn solve [lines]
  (let [s {:x 0 :y 0}]
    (each [_ line (ipairs lines)]
      (case (aoc.string-split line " ")
        ["forward" f] (tset s :x (+ f (. s :x)))
        ["down" d] (tset s :y (+ d (. s :y)))
        ["up" u] (tset s :y (- (. s :y) u))))
    (* (. s :x) (. s :y)))))

(fn test [expected input]
  (assert (= expected (solve input))))

(local test-input
       ["forward 5"
        "down 5"
        "forward 8"
        "up 3"
        "down 8"
        "forward 2"])

(test 150 test-input)

```

```
(solve (aoc.string-from "2021/02.inp"))
```

```
1561344
```

DONE Day 2.2

Based on your calculations, the planned course doesn't seem to make any sense. You find the submarine manual and discover that the process is actually slightly more complicated.

In addition to horizontal position and depth, you'll also need to track a third value, aim, which also starts at `0`. The commands also mean something entirely different than you first thought:

- down X increases your aim by X units.
- up X decreases your aim by X units.
- forward X does two things:
  - It increases your horizontal position by X units.
  - It increases your depth by your aim multiplied by X.

Again note that since you're on a submarine, down and up do the opposite of what you might expect: ”down” means aiming in the positive direction.

Now, the above example does something different:

- forward 5 adds 5 to your horizontal position, a total of 5. Because your aim is 0, your depth does not change.
- down 5 adds 5 to your aim, resulting in a value of 5.
- forward 8 adds 8 to your horizontal position, a total of 13. Because your aim is 5, your depth increases by 8*5=40.
- up 3 decreases your aim by 3, resulting in a value of 2.
- down 8 adds 8 to your aim, resulting in a value of 10.
- forward 2 adds 2 to your horizontal position, a total of 15. Because your aim is 10, your depth increases by 2*10=20 to a total of 60.

After following these new instructions, you would have a horizontal position of 15 and a depth of 60. (Multiplying these produces 900.)

Using this new interpretation of the commands, calculate the horizontal position and depth you would have after following the planned course. What do you get if you multiply your final horizontal position by your final depth?

```
(fn solve2 [lines]
  (let [s {:x 0 :y 0 :z 0}]
    (each [_ line (ipairs lines)]
```

```
      (case (aoc.string-split line " ")
        ["forward" f] (do
                        (tset s :y (+ (. s :y)
  ↪ (* (. s :z) f)))
                        (tset s :x (+ f (. s
  ↪ :x))))
        ["down" d] (tset s :z (+ d (. s :z)))
        ["up" u] (tset s :z (- (. s :z) u))))
    (* (. s :x) (. s :y))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 900 test-input)

(solve2 (aoc.string-from "2021/02.inp"))
```

```
1848454425
```

DONE Day 3.1

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

The diagnostic report (your puzzle input) consists of a list of binary numbers which, when decoded properly, can tell you

many useful things about the conditions of the submarine. The first parameter to check is the power consumption.

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

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

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

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

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

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

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
```

```
3
4  (fn read-input [input]
5    (aoc.table-transpose
6     (lume.map input #(aoc.string-toarray $))))
7
8  (fn solve [lines]
9    (let [xs (read-input lines)
10         gamma []
11         epsilon []]
12     (each [_ t (ipairs xs)]
13       (case (< (aoc.table-count t 1)
    ↪  (aoc.table-count t 0))
14         true (do
15                (table.insert gamma 0)
16                (table.insert epsilon 1))
17         false (do
18                 (table.insert gamma 1)
19                 (table.insert epsilon 0))))
20     (* (aoc.todecimal gamma)
21        (aoc.todecimal epsilon))))
22
23 (fn test [expected input]
24   (assert (= expected (solve input))))
25
26 (local test-input ["00100"
```

```
                    "11110"
                    "10110"
                    "10111"
                    "10101"
                    "01111"
                    "00111"
                    "11100"
                    "10000"
                    "11001"
                    "00010"
                    "01010"])

(test 198 test-input)

(solve (aoc.string-from "2021/03.inp"))
```

```
3009600
```

DONE Day 3.2

Next, you should verify the life support rating, which can be determined by multiplying the oxygen generator rating by the CO2 scrubber rating.

Both the oxygen generator rating and the CO2 scrubber rating

are values that can be found in your diagnostic report - finding them is the tricky part. Both values are located using a similar process that involves filtering out values until only one remains. Before searching for either rating value, start with the full list of binary numbers from your diagnostic report and consider just the first bit of those numbers. Then:

- Keep only numbers selected by the bit criteria for the type of rating value for which you are searching. Discard numbers which do not match the bit criteria.
- If you only have one number left, stop; this is the rating value for which you are searching.
- Otherwise, repeat the process, considering the next bit to the right.

The bit criteria depends on which type of rating value you want to find:

- To find oxygen generator rating, determine the most common value (0 or 1) in the current bit position, and keep only numbers with that bit in that position. If 0 and 1 are equally common, keep values with a 1 in the position being considered.
- To find CO2 scrubber rating, determine the least common

value (0 or 1) in the current bit position, and keep only numbers with that bit in that position. If 0 and 1 are equally common, keep values with a 0 in the position being considered.

For example, to determine the oxygen generator rating value using the same example diagnostic report from above:

- Start with all 12 numbers and consider only the first bit of each number. There are more 1 bits (7) than 0 bits (5), so keep only the 7 numbers with a 1 in the first position: 11110, 10110, 10111, 10101, 11100, 10000, and 11001.
- Then, consider the second bit of the 7 remaining numbers: there are more 0 bits (4) than 1 bits (3), so keep only the 4 numbers with a 0 in the second position: 10110, 10111, 10101, and 10000.
- In the third position, three of the four numbers have a 1, so keep those three: 10110, 10111, and 10101.
- In the fourth position, two of the three numbers have a 1, so keep those two: 10110 and 10111.
- In the fifth position, there are an equal number of 0 bits and 1 bits (one each). So, to find the oxygen generator rating, keep the number with a 1 in that position: 10111.
- As there is only one number left, stop; the oxygen gener-

ator rating is 10111, or 23 in decimal.

Then, to determine the CO2 scrubber rating value from the same example above:

- Start again with all 12 numbers and consider only the first bit of each number. There are fewer 0 bits (5) than 1 bits (7), so keep only the 5 numbers with a 0 in the first position: 00100, 01111, 00111, 00010, and 01010.
- Then, consider the second bit of the 5 remaining numbers: there are fewer 1 bits (2) than 0 bits (3), so keep only the 2 numbers with a 1 in the second position: 01111 and 01010.
- In the third position, there are an equal number of 0 bits and 1 bits (one each). So, to find the CO2 scrubber rating, keep the number with a 0 in that position: 01010.
- As there is only one number left, stop; the CO2 scrubber rating is 01010, or 10 in decimal.

Finally, to find the life support rating, multiply the oxygen generator rating (23) by the CO2 scrubber rating (10) to get 230.

Use the binary numbers in your diagnostic report to calculate the oxygen generator rating and CO2 scrubber rating, then multiply them together. What is the life support rating of the sub-

marine? (Be sure to represent your answer in decimal, not binary.)

```
(fn rate-generator [xs pos]
  (if (= 1 (length xs)) (aoc.todecimal (. xs 1))
      (let [bits (. (aoc.table-transpose xs) pos)
            ones (aoc.table-count bits 1)
            zeroes (aoc.table-count bits 0)]
        (var pred "")
        (if (<= zeroes ones)
            (set pred "1")
            (set pred "0"))
        (let [xss (lume.filter xs #(= pred (. $ pos)))]
          (rate-generator xss (+ 1 pos))))))

(fn rate-scrubber [xs pos]
  (if (= 1 (length xs)) (aoc.todecimal (. xs 1))
      (let [bits (. (aoc.table-transpose xs) pos)
            ones (aoc.table-count bits 1)
            zeroes (aoc.table-count bits 0)]
        (var pred "")
```

```
        (if (<= zeroes ones)
            (set pred "0")
            (set pred "1"))
        (let [xss (lume.filter xs #(= pred (. $ pos)))]
          (rate-scrubber xss (+ 1 pos))))))

(fn solve2 [lines]
  (let [xs (lume.map lines #(aoc.string-toarray $))
        g (rate-generator xs 1)
        s (rate-scrubber xs 1)]
    (* g s)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 230 test-input)

(solve2 (aoc.string-from "2021/03.inp"))
```

```
6940518
```

DONE Day 4.1

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

Maybe it wants to play bingo?

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

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

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

22 13 17 11  0
 8  2 23  4 24
21  9 14 16  7
```

```
 6 10  3 18  5
 1 12 20 15 19

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

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

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

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

```
 6 10  3 18  5        20 11 10 24  4        22
 ↪  11 13  6  5
 1 12 20 15 19        14 21 16 12  6         2
 ↪  0 12  3  7
```

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

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

Finally, 24 is drawn:

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

```
 6 10  3 18  5         20 11 10 24  4         22
↪  11 13  6  5
 1 12 20 15 19         14 21 16 12  6          2
↪  0 12  3  7
```

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input

 ↪  ["7,4,9,5,11,17,23,2,0,14,21,24,10,16,13,6,15,25,12,22
      ""
      "22 13 17 11  0"
```

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

(fn read-input [lines]
  (let [res []
        board []
        deal []]
    (each [num line (ipairs lines)]
      (let [digits (aoc.string-tonumarray line)]
        (if (= 1 num) (aoc.table-move 1 digits deal (length digits))
```

```
31              (= "" line) (let [new []]
32                            (when (= 5 (length
    ↪  board))
33                              (aoc.table-move 1
    ↪  board new 5)
34                              (table.insert res
    ↪  new)))
35              (table.insert board
36
    ↪  (aoc.string-tonumarray line)))))
37      (let [new []]
38        (aoc.table-move 1 board new 5)
39        (table.insert res new))
40      [deal res]))
41
42  (fn scan-board [board deal]
43    (var bingo false)
44    (for [i 1 (length board) &until bingo]
45      (when (lume.all (. board i) (fn [e]
    ↪  (aoc.table-contains? deal e)))
46        (set bingo i)))
47    (if bingo (. board bingo)
48        (let [nb (aoc.table-transpose board)]
49          (for [i 1 (length nb) &until bingo]
50            (when (lume.all (. nb i) (fn [e]
    ↪  (aoc.table-contains? deal e)))
```

```
51              (set bingo i)))
52          (or (?. nb bingo) [])))))
53
54  (fn score [deal winner]
55    (*
56     (. deal (length deal))
57     (aoc.table-sum-if winner
58                       #(not (aoc.table-contains?
    ↪  deal $)))))
59
60  (fn find-winner [deal boards]
61    (var winner 0)
62    (var numbers [])
63    (for [i 1 (length deal) &until (not= 0
    ↪  winner)]
64      (each [j board (ipairs boards) &until
    ↪  (not= 0 winner)]
65        (let [draw (aoc.table-range deal 1 i)
66              bingo (scan-board board draw)]
67          (when (not= 0 (length bingo))
68            (set winner j)
69            (set numbers draw)))))
70    [winner numbers])
71
72  (fn solve [input]
```

```
73   (let [[deals boards] (read-input input)
74         [winner numbers] (find-winner deals
     ↪  boards)]
75     (score numbers (. boards winner))))
76
77 (fn test [expected input]
78   (assert (= expected (solve input))))
79
80 (test 4512 test-input)
81
82 (solve (aoc.string-from "2021/04.inp"))
```

```
82440
```

DONE Day 4.2

On the other hand, it might be wise to try a different strategy: let the giant squid win.

You aren't sure how many bingo boards a giant squid could play at once, so rather than waste time counting its arms, the safe thing to do is to figure out which board will win last and choose that one. That way, no matter which boards it picks, it will win for sure.

In the above example, the second board is the last to win, which happens after 13 is eventually called and its middle column is completely marked. If you were to keep playing until this point, the second board would have a sum of unmarked numbers equal to 148 for a final score of 148 * 13 = 1924.

Figure out which board will win last. Once it wins, what would its final score be?

```
(fn find-score [deal boards]
  (let [[winner numbers] (find-winner deal
  ↪ boards)]
    (if (= 1 (length boards))
      (score numbers (. boards 1))
      (do
        (table.remove boards winner)
        (find-score deal boards)))))

(fn solve2 [input]
  (let [[deals boards] (read-input input)]
    (find-score deals boards)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 1924 test-input)
```

```
17
18 (solve2 (aoc.string-from "2021/04.inp"))
```

```
20774
```

DONE Day 5.1

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

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

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

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

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

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

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

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

In this diagram, the top left corner is 0,0 and the bottom right corner is 9,9. Each position is shown as the number of lines

which cover that point or . if no line covers that point. The top-left pair of 1s, for example, comes from 2,2 -> 2,1; the very bottom row is formed by the overlapping lines 0,9 -> 5,9 and 0,9 -> 2,9.

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["0,9 -> 5,9"
                   "8,0 -> 0,8"
                   "9,4 -> 3,4"
                   "2,2 -> 2,1"
                   "7,0 -> 7,4"
                   "6,4 -> 2,0"
                   "0,9 -> 2,9"
                   "3,4 -> 1,4"
                   "0,0 -> 8,8"
                   "5,5 -> 8,2"])

```

```
14 (fn create-matrix [n]
15   (let [res []]
16     (for [i 1 n]
17       (table.insert res (aoc.range-of 0 n)))
18     res))
19
20 (fn table-inc [xs]
21   (lume.map xs #(+ 1 $)))
22
23 (fn diagonal [x1 y1 x2 y2]
24   (let [xs (aoc.range-to x1 x2)
25         ys (aoc.range-to y1 y2)]
26     (aoc.table-zip
27      (if (> x1 x2)
28          (aoc.table-reverse xs)
29          xs)
30      (if (> y1 y2)
31          (aoc.table-reverse ys)
32          ys))))
33
34 (fn read-lines [lines diagonals]
35   (let [res (create-matrix 1000)]
36     (each [_ line (ipairs lines)]
37       (let [[x1 y1 x2 y2] (table-inc
    ↪  (aoc.string-tonumarray line))]
```

```
38          (if (= x1 x2)
39              (for [y (math.min y1 y2) (math.max
    ↪  y1 y2)]
40                (let [v (+ 1 (. (. res y) x1))]
41                  (aoc.table-replace res y x1
    ↪  v)))
42              (= y1 y2)
43              (for [x (math.min x1 x2) (math.max
    ↪  x1 x2)]
44                (let [v (+ 1 (. (. res y1) x))]
45                  (aoc.table-replace res y1 x
    ↪  v)))
46              (when diagonals
47                (each [_ [x y] (ipairs (diagonal
    ↪  x1 y1 x2 y2))]
48                  (aoc.table-replace res y x (+
    ↪  1 (. (. res y) x))))))))
49      res))
50
51  (fn solve [lines]
52    (let [xs (read-lines lines false)]
53      (accumulate [sum 0 _ x (ipairs xs)]
54        (+ sum (length (lume.filter x #(< 1
    ↪  $)))))))
55
```

```
(fn test [expected input]
  (assert (= expected (solve input))))

(test 5 test-input)

(solve (aoc.string-from "2021/05.inp"))
```

```
8622
```

DONE Day 5.2

Unfortunately, considering only horizontal and vertical lines doesn't give you the full picture; you need to also consider diagonal lines.

Because of the limits of the hydrothermal vent mapping system, the lines in your list will only ever be horizontal, vertical, or a diagonal line at exactly 45 degrees. In other words:

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

Considering all lines from the above example would now produce the following diagram:

```
1.1....11.
.111...2..
..2.1.111.
...1.2.2..
.112313211
...1.2....
..1...1...
.1.....1..
1.......1.
222111....
```

You still need to determine the number of points where at least two lines overlap. In the above example, this is still anywhere in the diagram with a 2 or larger - now a total of 12 points.

Consider all of the lines. At how many points do at least two lines overlap?

```
(fn solve2 [lines]
  (let [xs (read-lines lines true)]
    (accumulate [sum 0 _ x (ipairs xs)]
      (+ sum (length (lume.filter x #(< 1
 ↪  $)))))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))
```

```
8
9  (test2 12 test-input)
10
11 (solve2 (aoc.string-from "2021/05.inp"))
```

```
22037
```

DONE Day 6.1

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

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

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

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

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

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

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

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

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

following list:

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

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

```
Initial state: 3,4,3,1,2
After  1 day:  2,3,2,0,1
After  2 days: 1,2,1,6,0,8
After  3 days: 0,1,0,5,6,7,8
After  4 days: 6,0,6,4,5,6,7,8,8
After  5 days: 5,6,5,3,4,5,6,7,7,8
After  6 days: 4,5,4,2,3,4,5,6,6,7
After  7 days: 3,4,3,1,2,3,4,5,5,6
After  8 days: 2,3,2,0,1,2,3,4,4,5
After  9 days: 1,2,1,6,0,1,2,3,3,4,8
After 10 days: 0,1,0,5,6,0,1,2,2,3,7,8
After 11 days: 6,0,6,4,5,6,0,1,1,2,6,7,8,8,8
After 12 days:
 ↪  5,6,5,3,4,5,6,0,0,1,5,6,7,7,7,8,8
After 13 days:
 ↪  4,5,4,2,3,4,5,6,6,0,4,5,6,6,6,7,7,8,8
After 14 days:
 ↪  3,4,3,1,2,3,4,5,5,6,3,4,5,5,5,6,6,7,7,8
```

```
After 15 days:
 ↪  2,3,2,0,1,2,3,4,4,5,2,3,4,4,4,5,5,6,6,7
After 16 days:
 ↪  1,2,1,6,0,1,2,3,3,4,1,2,3,3,3,4,4,5,5,6,8
After 17 days:
 ↪  0,1,0,5,6,0,1,2,2,3,0,1,2,2,2,3,3,4,4,5,7,8
After 18 days:
 ↪  6,0,6,4,5,6,0,1,1,2,6,0,1,1,1,2,2,3,3,4,6,7,8,8,8,8
```

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input "3,4,3,1,2")

(fn life-cycle [xs]
  (let [result []]
    (each [_ x (ipairs xs)]
      (case x
```

```
        0 (do
            (table.insert result 6)
            (table.insert result 8))
        _ (table.insert result (- x 1))))
    result))

(fn solve [input]
  (var res (aoc.string-tonumarray (. input
  ↪ 1)))
  (for [i 1 80]
    (set res (life-cycle res)))
  (length res))

(fn test [expected input]
  (assert (= expected (solve [input]))))

(test 5934 test-input)

(solve (aoc.string-from "2021/06.inp"))
```

```
354564
```

DONE Day 6.2

Suppose the lanternfish live forever and have unlimited food and space. Would they take over the entire ocean?

After 256 days in the example above, there would be a total of 26984457539 lanternfish! How many lanternfish would there be after 256 days?

```
(fn read-input [input]
  (let [population [0 0 0 0 0 0 0 0 0]]
    (icollect [_ x (ipairs
    (aoc.string-tonumarray input))]
      (tset population (+ 1 x) (+ 1 (.
    population (+ 1 x)))))
    population))

(fn life-cycle2 [xs]
  [(. xs 2)
   (. xs 3)
   (. xs 4)
   (. xs 5)
   (. xs 6)
   (. xs 7)
   (+ (. xs 8) (. xs 1))
   (. xs 9)
```

```
16    (. xs 1)])
17
18  (fn run-cycle [input cycles]
19    (if (= 0 cycles) input
20        (run-cycle (life-cycle2 input) (- cycles
    ↪  1))))
21
22  (fn solve2 [input]
23    (aoc.table-sum (run-cycle (read-input (.
    ↪  input 1)) 256)))
24
25  (fn test2 [expected input]
26    (assert (= expected (solve2 [input]))))
27
28  (test2 26984457539 test-input)
29
30  (solve2 (aoc.string-from "2021/06.inp"))
```

```
1609058859115
```

DONE Day 7.1

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

Suddenly, a swarm of crabs (each in its own tiny submarine -

it's too deep for them otherwise) zooms in to rescue you! They seem to be preparing to blast a hole in the ocean floor; sensors indicate a massive underground cave system just beyond where they're aiming!

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

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

You quickly make a list of the horizontal position of each crab (your puzzle input). Crab submarines have limited fuel, so you need to find a way to make all of their horizontal positions match while requiring them to spend as little fuel as possible.

For example, consider the following horizontal positions:

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

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

Each change of 1 step in horizontal position of a single crab costs 1 fuel. You could choose any horizontal position to align

them all on, but the one that costs the least fuel is horizontal position 2:

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input "16,1,2,0,4,2,7,1,2,14")
```

```
4
5  (fn even? [x]
6    (= 0 (% x 2)))
7
8  (fn median [xs]
9    (table.sort xs)
10   (let [len (length xs)
11         mid (aoc.int/ len 2)]
12     (if (even? len)
13         (/ (+ (. xs mid) (. xs (aoc.inc mid)))
     ↪  2)
14         (. xs (+ 1 mid)))))
15
16 (fn solve [input]
17   (let [in (aoc.string-tonumarray (. input 1))
18         med (median in)]
19     (aoc.table-sum
20      (icollect [_ v (ipairs in)]
21        (math.abs (- med v))))))
22
23 (fn test [expected input]
24   (assert (= expected (solve [input]))))
25
26 (test 37 test-input)
27
```

```
28 (solve (aoc.string-from "2021/07.inp"))
```

```
336131
```

DONE Day 7.2

The crabs don't seem interested in your proposed solution. Perhaps you misunderstand crab engineering?

As it turns out, crab submarine engines don't burn fuel at a constant rate. Instead, each change of 1 step in horizontal position costs 1 more unit of fuel than the last: the first step costs 1, the second step costs 2, the third step costs 3, and so on.

As each crab moves, moving further becomes more expensive. This changes the best horizontal position to align them all on; in the example above, this becomes 5:

- Move from 16 to 5: 66 fuel
- Move from 1 to 5: 10 fuel
- Move from 2 to 5: 6 fuel
- Move from 0 to 5: 15 fuel
- Move from 4 to 5: 1 fuel
- Move from 2 to 5: 6 fuel
- Move from 7 to 5: 3 fuel

- Move from 1 to 5: 10 fuel
- Move from 2 to 5: 6 fuel
- Move from 14 to 5: 45 fuel

This costs a total of 170 fuel. This is the new cheapest possible outcome; the old alignment position (2) now costs 206 fuel instead.

Determine the horizontal position that the crabs can align to using the least fuel possible so they can make you an escape route! How much fuel must they spend to align to that position?

```
(fn mean [xs]
  (let [sum (aoc.table-sum xs)
        cnt (length xs)]
    (math.floor (/ sum cnt))))

(fn sum [n]
  (/ (* n (+ 1 n)) 2))

(fn fuel [f t]
  (sum (math.abs (- f t))))

(fn solve2 [input]
  (let [in (aoc.string-tonumarray (. input 1))
```

```
        med (mean in)]
    (aoc.table-sum
     (icollect [_ v (ipairs in)]
       (fuel med v)))))

(fn test2 [expected input]
  (assert (= expected (solve2 [input]))))

(test2 170 test-input)

(solve2 (aoc.string-from "2021/07.inp"))
```

```
92676646
```

DONE Day 9.1

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

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

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

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

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

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

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

The risk level of a low point is 1 plus its height. In the above

example, the risk levels of the low points are 2, 1, 6, and 6. The sum of the risk levels of all low points in the heightmap is therefore 15.

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["2199943210"
                   "3987894921"
                   "9856789892"
                   "8767896789"
                   "9899965678"])

(fn is-low? [xs i j]
  (let [ij (. (. xs i) j)
        lf (or (?. (?. xs i) (- j 1)) 10)
        rg (or (?. (?. xs i) (+ j 1)) 10)
        up (or (?. (?. xs (- i 1)) j) 10)
        dn (or (?. (?. xs (+ i 1)) j) 10)]
    (and
     (< ij up)
     (< ij dn)
     (< ij lf)
     (< ij rg))))
```

```
20
21 (fn solve [lines]
22   (var res 0)
23   (let [input (aoc.read-matrix lines true)]
24     (for [i 1 (length input)]
25       (for [j 1 (length (. input i))]
26         (when (is-low? input i j)
27           (set res (+ res (+ 1 (. (. input i)
    ↪  j))))))))
28   res)
29
30 (fn test [expected input]
31   (assert (= expected (solve input))))
32
33 (test 15 test-input)
34
35 (solve (aoc.string-from "2021/09.inp"))
```

```
522
```

DONE Day 9.2

Next, you need to find the largest basins so you know what areas are most important to avoid.

A basin is all locations that eventually flow downward to a single

low point. Therefore, every low point has a basin, although some basins are very small. Locations of height 9 do not count as being in any basin, and all other locations will always be part of exactly one basin.

The size of a basin is the number of locations within the basin, including the low point. The example above has four basins.

The top-left basin, size 3:

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

The top-right basin, size 9:

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

The middle basin, size 14:

```
2199943210
3987894921
```

```
9856789892
8767896789
9899965678
```

The bottom-right basin, size 9:

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

Find the three largest basins and multiply their sizes together. In the above example, this is 9 * 14 * 9 = 1134.

What do you get if you multiply together the sizes of the three largest basins?

```
(fn table-contains? [xs e]
  (var contains false)
  (each [i x (ipairs xs) &until contains]
    (when (and (= (. e 1) (. x 1))
               (= (. e 2) (. x 2)))
      (set contains true)))
  contains)

(fn size-basin [xs basin low]
```

```
10    (let [[i j] low
11          ij (or (?. (?. xs i) j) 10)
12          lf (or (?. (?. xs i) (- j 1)) 10)
13          rg (or (?. (?. xs i) (+ j 1)) 10)
14          up (or (?. (?. xs (- i 1)) j) 10)
15          dn (or (?. (?. xs (+ i 1)) j) 10)]
16      (when (and (< ij 9)
17                 (not (table-contains? basin
    ↪  low)))
18        (table.insert basin low)
19        (when (and (<= ij up) (< up 9))
20          (size-basin xs basin [(- i 1) j]))
21        (when (and (<= ij dn) (< dn 9))
22          (size-basin xs basin [(+ i 1) j]))
23        (when (and (<= ij lf) (< lf 9))
24          (size-basin xs basin [i (- j 1)]))
25        (when (and (<= ij rg) (< rg 9))
26          (size-basin xs basin [i (+ j 1)])))))
27    (length basin))
28
29  (fn solve2 [lines]
30    (let [input (aoc.read-matrix lines true)
31          res []]
32      (for [i 1 (length input)]
33        (for [j 1 (length (. input i))]
```

```
34          (when (is-low? input i j)
35            (table.insert res (size-basin input
    ↪  [] [i j])))))
36      (table.sort res)
37      (* (table.remove res)
38         (table.remove res)
39         (table.remove res))))
40
41  (fn test2 [expected input]
42    (assert (= expected (solve2 input))))
43
44  (test2 1134 test-input)
45
46  (solve2 (aoc.string-from "2021/09.inp"))
```

```
916688
```

DONE Day 10.1

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

Syntax error in navigation subsystem on line: all of them

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

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

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

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

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

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

Examples of corrupted chunks include (], {()()()>, (((()))}, and <([]){()}[{}]). Such a chunk can appear anywhere within a line,

and its presence causes the whole line to be considered corrupted.

For example, consider the following navigation subsystem:

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

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

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

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

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

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

-): 3 points.
-]: 57 points.
- }: 1197 points.
- >: 25137 points.

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

Find the first illegal character in each corrupted line of the navi-

gation subsystem. What is the total syntax error score for those errors?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["[({(<(())[]>[[{[]{<()<>>"
                   "[(()[<>])]({[<{<<[]>>("
                   "{([(<{}[<>[]}>{[]{[(<()>"
                   "(((({<>}<{<{<>}{[]{[]{}"
                   "[[<[([]))<([[{}[[()]]]"
                   "[{[{({}]{}}([{[{{{}}([]"
                   "{<[[]]>}<{[{[{[]{()[[[]"
                   "[<(<(<(<{}))><([]([]()"
                   "<{([([[(<>()){}]>(<<{{"

 ↪  "<{([{{}}[<[[[<>{}]]]>[]]"])

(fn solve [lines]
  (let [result []]
    (each [_ line (ipairs lines)]
      (let [stack []
            input (aoc.string-toarray line)]
        (var stop false)
        (each [_ chunk (ipairs input) &until
 ↪  stop]
          (case chunk
```

```
            "<" (table.insert stack -25137)
            "{" (table.insert stack -1197)
            "[" (table.insert stack -57)
            "(" (table.insert stack -3)
            ">" (when (not= 0 (+ 25137
  ↪ (table.remove stack)))
                  (table.insert result 25137)
                  (set stop true))
            "}" (when (not= 0 (+ 1197
  ↪ (table.remove stack)))
                  (table.insert result 1197)
                  (set stop true))
            "]" (when (not= 0 (+ 57
  ↪ (table.remove stack)))
                  (table.insert result 57)
                  (set stop true))
            ")" (when (not= 0 (+ 3
  ↪ (table.remove stack)))
                  (table.insert result 3)
                  (set stop true))))))
    (aoc.table-sum result)))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(test 26397 test-input)

(solve (aoc.string-from "2021/10.inp"))
```

```
399153
```

DONE Day 10.2

Now, discard the corrupted lines. The remaining lines are incomplete.

Incomplete lines don't have any incorrect characters - instead, they're missing some closing characters at the end of the line. To repair the navigation subsystem, you just need to figure out the sequence of closing characters that complete all open chunks in the line.

You can only use closing characters (),], }, or >), and you must add them in the correct order so that only legal pairs are formed and all chunks end up closed.

In the example above, there are five incomplete lines:

```
[({(<(())[]>[[{[]{<()<>> - Complete by adding
 ↪  }}]])})].
[(()[<>])]({[<{<<[]>>( - Complete by adding
 ↪  )}>]}).
```

```
((((({<>}<{<{<>}{[]{[]{} - Complete by adding
 ↪  }}>}>)))).
{<[[]]>}<{[{[{[]{()[[[] - Complete by adding
 ↪  ]]}}]}]}>.
<{([{{}}[<[[[<>{}]]]>[]] - Complete by adding
 ↪  ])}>.
```

Did you know that autocomplete tools also have contests? It's true! The score is determined by considering the completion string character-by-character. Start with a total score of 0. Then, for each character, multiply the total score by 5 and then increase the total score by the point value given for the character in the following table:

-): 1 point.
-]: 2 points.
- }: 3 points.
- >: 4 points.

So, the last completion string above -])}> - would be scored as follows:

- Start with a total score of 0.
- Multiply the total score by 5 to get 0, then add the value of] (2) to get a new total score of 2.

- Multiply the total score by 5 to get 10, then add the value of) (1) to get a new total score of 11.
- Multiply the total score by 5 to get 55, then add the value of } (3) to get a new total score of 58.
- Multiply the total score by 5 to get 290, then add the value of > (4) to get a new total score of 294.

The five lines' completion strings have total scores as follows:

```
}}]])})] - 288957 total points.
)}>]}) - 5566 total points.
}}>}>)))) - 1480781 total points.
]]}}]]}> - 995444 total points.
])}> - 294 total points.
```

Autocomplete tools are an odd bunch: the winner is found by sorting all of the scores and then taking the middle score. (There will always be an odd number of scores to consider.) In this example, the middle score is 288957 because there are the same number of scores smaller and larger than it.

Find the completion string for each incomplete line, score the completion strings, and sort the scores. What is the middle score?

```
(fn complete [xs]
```

```
  (var score 0)
  (each [_ v (ipairs xs)]
    (case v
      -3 (set score (+ (* 5 score) 1))
      -57 (set score (+ (* 5 score) 2))
      -1197 (set score (+ (* 5 score) 3))
      -25137 (set score (+ (* 5 score) 4))))
  score)

(fn solve2 [lines]
  (let [result []]
    (each [_ line (ipairs lines)]
      (let [stack []
            input (aoc.string-toarray line)]
        (var stop false)
        (each [_ chunk (ipairs input) &until stop]
          (case chunk
            "<" (table.insert stack -25137)
            "{" (table.insert stack -1197)
            "[" (table.insert stack -57)
            "(" (table.insert stack -3)
            ">" (when (not= 0 (+ 25137 (table.remove stack)))
                  (set stop true))
```

```
25              "}" (when (not= 0 (+ 1197
    ↪  (table.remove stack)))
26                    (set stop true))
27              "]" (when (not= 0 (+ 57
    ↪  (table.remove stack)))
28                    (set stop true))
29              ")" (when (not= 0 (+ 3
    ↪  (table.remove stack)))
30                    (set stop true))))
31          (when (not stop)
32            (table.insert result (complete
    ↪  (aoc.table-reverse stack))))))
33      (table.sort result)
34      (let [mid (+ 1 (aoc.int/ (length result)
    ↪  2))]
35        (. result mid))))
36
37  (fn test2 [expected input]
38    (assert (= expected (solve2 input))))
39
40  (test2 288957 test-input)
41
42  (solve2 (aoc.string-from "2021/10.inp"))
```

```
2995077699
```

2020 [18/50]

DONE Day 1.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local test-input ["1721" "979" "366" "299"
 ↪  "675" "1456"])
```

```
5
6  (fn solve [input]
7    (let [xx (tonumber (aoc.first input))
8          xs (aoc.rest input)
9          res (lume.filter xs (fn [a] (= 2020 (+
    ↪  (tonumber a) xx))))]
10     (if (aoc.empty? res)
11         (solve xs)
12         (* xx (tonumber (. res 1))))))
13
14 (fn test [expected input]
15   (assert (= expected (solve input))))
16
17 (test 514579 test-input)
18
19 (solve (aoc.string-from "2020/01.inp"))
```

```
63616
```

DONE Day 1.2

The Elves in accounting are thankful for your help; one of them even offers you a starfish coin they had left over from a past vacation. They offer you a second one if you can find three numbers in your expense report that meet the same criteria.

Using the above example again, the three entries that sum to 2020 are 979, 366, and 675. Multiplying them together produces the answer, 241861950.

In your expense report, what is the product of the three entries that sum to 2020?

```
(fn solve2 [input]
  (var res nil)
  (for [i 1 (length input) &until res]
    (for [j i (length input) &until res]
      (for [k j (length input) &until res]
        (let [ii (tonumber (. input i))
              jj (tonumber (. input j))
              kk (tonumber (. input k))]
          (when (= 2020 (+ ii jj kk))
            (set res (* ii jj kk)))))))
  res)

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 241861950 test-input)

(solve2 (aoc.string-from "2020/01.inp"))
```

```
67877784
```

DONE Day 2.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local test-input ["1-3 a: abcde"
                   "1-3 b: cdefg"
                   "2-9 c: ccccccccc"])

(fn read-policy [line]
  (let [tokens (aoc.string-split line " ")
        [min max] (aoc.string-split (. tokens 1) "-")
        val (string.sub (. tokens 2) 1 -2)
        pass (. tokens 3)]
    {:min (tonumber min) :max (tonumber max) :val val :pass pass}))
```

```
14
15 (fn validate-policy [p]
16   (var res 0)
17   (each [_ v (ipairs (aoc.string-toarray (. p
    ↪  :pass)))]
18     (when (= v (. p :val))
19         (set res (+ 1 res))))
20   (if (and (<= (. p :min) res)
21        (<= res (. p :max)))
22       1
23       0))
24
25 (fn solve [input]
26   (var count 0)
27   (each [_ line (ipairs input)]
28     (set count (+ count
29                   (validate-policy
    ↪  (read-policy line)))))
30   count)
31
32 (fn test [expected input]
33   (assert (= expected (solve input))))
34
35 (test 2 test-input)
36
```

```
37 (solve (aoc.string-from "2020/02.inp"))
```

```
600
```

DONE Day 2.2

While it appears you validated the passwords correctly, they don't seem to be what the Official Toboggan Corporate Authentication System is expecting.

The shopkeeper suddenly realizes that he just accidentally explained the password policy rules from his old job at the sled rental place down the street! The Official Toboggan Corporate Policy actually works a little differently.

Each policy actually describes two positions in the password, where 1 means the first character, 2 means the second character, and so on. (Be careful; Toboggan Corporate Policies have no concept of "index zero"!) Exactly one of these positions must contain the given letter. Other occurrences of the letter are irrelevant for the purposes of policy enforcement.

Given the same example list from above:

```
1-3 a: abcde is valid: position 1 contains a
 ↪  and position 3 does not.
```

```
1-3 b: cdefg is invalid: neither position 1
 ↪  nor position 3 contains b.
2-9 c: ccccccccc is invalid: both position 2
 ↪  and position 9 contain c.
```

How many passwords are valid according to the new interpretation of the policies?

```
(fn validate-policy2 [p]
  (let [val (. p :val)
        arr (aoc.string-toarray (. p :pass))]
    (if (aoc.xor
         (= val (. arr (. p :min)))
         (= val (. arr (. p :max))))
        1
        0)))

(fn solve2 [input]
  (var count 0)
  (each [_ line (ipairs input)]
    (set count (+ count
                  (validate-policy2
 ↪  (read-policy line)))))
  count)

(fn test2 [expected input]
```

```
18    (assert (= expected (solve2 input))))
19
20  (test2 1 test-input)
21
22  (solve2 (aoc.string-from "2020/02.inp"))
```

```
245
```

DONE Day 3.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["..##......."
                   "#...#...#.."
                   ".#....#..#."
                   "..#.#...#.#"
                   ".#...##..#."
                   "..#.##....."
                   ".#.#.#....#"
                   ".#........#"
                   "#.##...#..."
                   "#...##....#"
                   ".#..#...#.#"])

```

```
(fn solve [dx dy lines]
  (var posx 1)
  (var posy 1)
  (let [field (aoc.read-matrix lines)
        lenx (length (. field 1))
        leny (length field)
        res []]
    (while (<= posy leny)
      (case (. (. field posy ) posx)
        "." (table.insert res 0)
        "#" (table.insert res 1))
      (set posx (if (>= lenx (+ dx posx))
                    (+ dx posx)
                    (- (+ dx posx) lenx)))
      (set posy (+ dy posy)))
    (aoc.table-sum res)))

(fn test [expected dx dy input]
  (assert (= expected (solve dx dy input))))

(test 7 3 1 test-input)

(solve 3 1 (aoc.string-from "2020/03.inp"))
```

```
178
```

DONE Day 3.2

Time to check the rest of the slopes - you need to minimize the probability of a sudden arboreal stop, after all.

Determine the number of trees you would encounter if, for each of the following slopes, you start at the top-left corner and traverse the map all the way to the bottom:

- Right 1, down 1.
- Right 3, down 1. (This is the slope you already checked.)
- Right 5, down 1.
- Right 7, down 1.
- Right 1, down 2.

In the above example, these slopes would find 2, 7, 3, 4, and 2 tree(s) respectively; multiplied together, these produce the answer 336.

What do you get if you multiply together the number of trees encountered on each of the listed slopes?

```
(fn solve2 [lines]
  (let [slopes [[1 1]
                [3 1]
                [5 1]
```

```
                [7 1]
                [1 2]]
        res []]
    (each [_ [dx dy] (ipairs slopes)]
      (table.insert res (solve dx dy lines)))
    (aoc.table-prod res)))

(fn test2 [expected lines]
  (assert (= expected (solve2 lines))))

(test2 336 test-input)

(solve2 (aoc.string-from "2020/03.inp"))
```

```
3492520200
```

DONE Day 4.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["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"])
```

```
17
18 (fn valid? [record]
19   (if (and (. record :byr) (. record :iyr) (.
   ↪  record :eyr)
20            (. record :hgt) (. record :hcl) (.
   ↪  record :ecl)
21            (. record :pid))
22       record
23       nil))
24
25 (fn lines-torecords [lines]
26   (when (not (= "" (. lines (length lines))))
27     (table.insert lines "")) ;; need separator
   ↪  for last iteration
28   (let [records []
29         record []]
30     (each [_ line (ipairs lines)]
31       (if (= "" line)
32           (do
33             (table.insert records
   ↪  (aoc.string-totable (table.concat record "
   ↪  ")))
34             (aoc.table-reset record))
35           (table.insert record line)))
36     records))
```

```
37
38 (fn solve [lines]
39   (let [records (lines-torecords lines)]
40     (length (lume.filter records (fn [x]
    ↪  (valid? x))))))
41
42 (fn test [expected input]
43   (assert (= expected (solve input))))
44
45 (test 2 test-input)
46
47 (solve (aoc.string-from "2020/04.inp"))
```

```
202
```

DONE Day 4.2

The line is moving more quickly now, but you overhear airport security talking about how passports with invalid data are getting through. Better add some data validation, quick!

You can continue to ignore the cid field, but each other field has strict rules about what values are valid for automatic validation:

- byr (Birth Year) - four digits; at least 1920 and at most 2002.
- iyr (Issue Year) - four digits; at least 2010 and at most 2020.
- eyr (Expiration Year) - four digits; at least 2020 and at most 2030.
- hgt (Height) - a number followed by either cm or in:
  - If cm, the number must be at least 150 and at most 193.
  - If in, the number must be at least 59 and at most 76.
- hcl (Hair Color) - a # followed by exactly six characters 0-9 or a-f.
- ecl (Eye Color) - exactly one of: amb blu brn gry grn hzl oth.
- pid (Passport ID) - a nine-digit number, including leading zeroes.
- cid (Country ID) - ignored, missing or not.

Your job is to count the passports where all required fields are both present and valid according to the above rules. Here are some example values:

```
byr valid:   2002
```

```
byr invalid: 2003

hgt valid:   60in
hgt valid:   190cm
hgt invalid: 190in
hgt invalid: 190

hcl valid:   #123abc
hcl invalid: #123abz
hcl invalid: 123abc

ecl valid:   brn
ecl invalid: wat

pid valid:   000000001
pid invalid: 0123456789
```

Here are some invalid passports:

```
eyr:1972 cid:100
hcl:#18171d ecl:amb hgt:170 pid:186cm iyr:2018
 ↪  byr:1926

iyr:2019
hcl:#602927 eyr:1967 hgt:170cm
ecl:grn pid:012533040 byr:1946
```

```
hcl:dab227 iyr:2012
ecl:brn hgt:182cm pid:021572410 eyr:2020
 ↪  byr:1992 cid:277

hgt:59cm ecl:zzz
eyr:2038 hcl:74454a iyr:2023
pid:3556412378 byr:2007
```

Here are some valid passports:

```
pid:087499704 hgt:74in ecl:grn iyr:2012
 ↪  eyr:2030 byr:1980
hcl:#623a2f

eyr:2029 ecl:blu cid:129 byr:1989
iyr:2014 pid:896056539 hcl:#a97842 hgt:165cm

hcl:#888785
hgt:164cm byr:2001 iyr:2015 cid:88
pid:545766238 ecl:hzl
eyr:2022

iyr:2010 hgt:158cm hcl:#b6652a ecl:blu
 ↪  byr:1944 eyr:2021 pid:093154719
```

Count the number of valid passports - those that have all re-

quired fields and valid values. Continue to treat cid as optional. In your batch file, how many passports are valid?

```
(local test-input2
       ["pid:087499704 hgt:74in ecl:grn
 ↪ iyr:2012 eyr:2030 byr:1980"
        "hcl:#623a2f"
        ""
        "eyr:2029 ecl:blu cid:129 byr:1989"
        "iyr:2014 pid:896056539 hcl:#a97842
 ↪ hgt:165cm"
        ""
        "hcl:#888785"
        "hgt:164cm byr:2001 iyr:2015 cid:88"
        "pid:545766238 ecl:hzl"
        "eyr:2022"
        ""
        "iyr:2010 hgt:158cm hcl:#b6652a
 ↪ ecl:blu byr:1944 eyr:2021 pid:093154719"
        ""
        "eyr:1972 cid:100"
        "hcl:#18171d ecl:amb hgt:170 pid:186cm
 ↪ iyr:2018 byr:1926"
        ""
        "iyr:2019"
        "hcl:#602927 eyr:1967 hgt:170cm"
```

```
20          "ecl:grn pid:012533040 byr:1946"
21          ""
22          "hcl:dab227 iyr:2012"
23          "ecl:brn hgt:182cm pid:021572410
    ↪  eyr:2020 byr:1992 cid:277"
24          ""
25          "hgt:59cm ecl:zzz"
26          "eyr:2038 hcl:74454a iyr:2023"
27          "pid:3556412378 byr:2007"])
28
29  (fn valid-byr? [record]
30    "valid if four digits at least 1920 and at
    ↪  most 2002"
31    (match (tonumber (. record :byr))
32      (where byr (and (<= 1920 byr) (<= byr
    ↪  2002))) true
33      _ false))
34
35  (fn valid-iyr? [record]
36    "valid if four digits at least 2010 and at
    ↪  most 2020"
37    (match (tonumber (. record :iyr))
38      (where iyr (and (<= 2010 iyr) (<= iyr
    ↪  2020))) true
39      _ false))
```

```
40
41 (fn valid-eyr? [record]
42   "valid if four digits at least 2020 and at
   ↪  most 2030"
43   (match (tonumber (. record :eyr))
44     (where eyr (and (<= 2020 eyr) (<= eyr
   ↪  2030))) true
45     _ false))
46
47 (fn valid-hgt? [record]
48   "valid if number between 150cm and 193cm or
   ↪  59in and 76in"
49   (match (. record :hgt)
50     (where hgt (or
51                  (and (= "cm" (string.sub hgt
   ↪  -2))
52                       (let [h (tonumber
   ↪  (string.sub hgt 1 -3))]
53                         (and (<= 150 h) (<= h
   ↪  193))))
54                  (and (= "in" (string.sub hgt
   ↪  -2))
55                       (let [h (tonumber
   ↪  (string.sub hgt 1 -3))]
56                         (and (<= 59 h) (<= h
   ↪  76)))))) true
```

```
57      _ false))
58
59  (fn valid-hcl? [record]
60    "valid if a # followed by exactly six
    ↪  characters 0-9 or a-f"
61    (match (. record :hcl)
62      (where hcl (and (= 7 (length hcl))
63                      (= "#" (string.sub hcl 1
    ↪  1))
64                      (let [(b e) (string.find
    ↪  hcl "%x*" 2)]
65                        (and (= b 2) (= e 7)))))
    ↪  true
66      _ false))
67
68  (fn valid-ecl? [record]
69    "valid if exactly one of: amb blu brn gry
    ↪  grn hzl oth"
70    (case (. record :ecl)
71      "amb" true
72      "blu" true
73      "brn" true
74      "gry" true
75      "grn" true
76      "hzl" true
```

```
77      "oth" true
78      _ false))
79
80  (fn valid-pid? [record]
81    "valid if a nine-digit number, including
    ↪  leading zeroes"
82    (match (. record :pid)
83      (where pid (and (string.find pid "%d") (=
    ↪  9 (length pid)))) true
84      _ false))
85
86  (fn valid2? [record]
87    (and (valid-byr? record)
88         (valid-iyr? record)
89         (valid-eyr? record)
90         (valid-hgt? record)
91         (valid-hcl? record)
92         (valid-ecl? record)
93         (valid-pid? record)))
94
95  (fn solve2 [lines]
96    (let [records (lines-torecords lines)]
97      (length (lume.filter records (fn [x]
    ↪  (valid2? x))))))
98
```

```
(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 4 test-input2)

(solve2 (aoc.string-from "2020/04.inp"))
```

```
137
```

DONE Day 5.1

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 FBFBBF-FRLR:

- 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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["FBFBBFFRLR"
                   "BFFFBBFRRR"
                   "FFFBBBFRRR"
                   "BBFFBBFRLL"])

(fn find-row [input range]
  (let [c (aoc.first input)
        e (aoc.rest input)]
    (case c
      "F" (find-row e (aoc.table-range range 1 (/ (length range) 2)))
      "B" (find-row e (aoc.table-range range (+ 1 (/ (length range) 2)) (length range)))
      _ (. range 1))))

(fn find-col [input range]
```

```
17    (let [c (aoc.first input)
18          e (aoc.rest input)]
19      (case c
20        "L" (find-col e (aoc.table-range range 1
    ↪  (/ (length range) 2)))
21        "R" (find-col e (aoc.table-range range
    ↪  (+ 1 (/ (length range) 2)) (length
    ↪  range)))
22        _ (. range 1))))
23
24  (fn find-seat-num [line]
25    (let [rows (aoc.range 0 127)
26          cols (aoc.range 0 7)
27          input (aoc.string-toarray line)
28          input-row (aoc.table-range input  1 7)
29          input-col (aoc.table-range input  8
    ↪  10)
30          row (find-row input-row rows)
31          col (find-col input-col cols)]
32      (+ (* row 8) col)))
33
34  (fn find-seat-nums [lines]
35    (let [res []]
36      (each [_ line (ipairs lines)]
37        (table.insert res (find-seat-num line)))
```

```
    res))

(fn solve [lines]
  (aoc.table-max (find-seat-nums lines)))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 820 test-input)

(solve (aoc.string-from "2020/05.inp"))
```

```
890
```

DONE Day 5.2

Ding! The ”fasten seat belt” signs have turned on. Time to find your seat.

It's a completely full flight, so your seat should be the only missing boarding pass in your list. However, there's a catch: some of the seats at the very front and back of the plane don't exist on this aircraft, so they'll be missing from your list as well.

Your seat wasn't at the very front or back, though; the seats with IDs +1 and -1 from yours will be in your list.

What is the ID of your seat?

```
(fn solve2 [lines]
  (let [all-seats (aoc.range-to 85 890)
        taken-seats (find-seat-nums lines)
        free-seats (lume.filter all-seats (fn
 ↪  [x] (not (lume.find taken-seats x))))]
    (. free-seats 1)))

(solve2 (aoc.string-from "2020/05.inp"))
```

```
651
```

DONE Day 6.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["abc" ""
                   "a" "b" "c" ""
                   "ab" "ac" ""
                   "a" "a" "a" "a" ""
                   "b"])

(fn any-answers [input]
  (when (not= "" (. input (length input)))
    (table.insert input ""))
  (let [res []
        group []]
    (each [_ line (ipairs input)]
      (if (= "" line)
          (do (table.insert res (length
 ↪  (aoc.table-union group)))
              (aoc.table-reset group))
          (table.insert group
 ↪  (aoc.string-toarray line))))
    res))

(fn solve [input]
  (let [answers (any-answers input)]
    (accumulate [sum 0 _ answer (ipairs
 ↪  answers)]
```

```
      (+ sum answer))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 11 test-input)

(solve (aoc.string-from "2020/06.inp"))
```

```
7120
```

DONE Day 6.2

As you finish the last group's customs declaration, you notice that you misread one word in the instructions:

You don't need to identify the questions to which anyone answered ”yes”; you need to identify the questions to which everyone answered ”yes”!

Using the same example as above:

```
abc

a
b
```

```
c

ab
ac

a
a
a
a

b
```

This list represents answers from five groups:

- In the first group, everyone (all 1 person) answered ”yes” to 3 questions: a, b, and c.
- In the second group, there is no question to which everyone answered ”yes”.
- In the third group, everyone answered yes to only 1 question, a. Since some people did not answer ”yes” to b or c, they don't count.
- In the fourth group, everyone answered yes to only 1 question, a.
- In the fifth group, everyone (all 1 person) answered ”yes” to 1 question, b.

In this example, the sum of these counts is `3 + 0 + 1 + 1 + 1 = 6`.

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

```
(fn every-answers [input]
  (when (not= "" (. input (length input)))
    (table.insert input ""))
  (let [res []
        group []]
    (each [_ line (ipairs input)]
      (if (= "" line)
          (do (table.insert res (length
 ↪  (aoc.table-disjunc group)))
              (aoc.table-reset group))
          (table.insert group
 ↪  (aoc.string-toarray line))))
    res))

(fn solve2 [input]
  (let [answers (every-answers input)]
    (accumulate [sum 0 _ answer (ipairs
 ↪  answers)]
      (+ sum answer))))

```

```
18 (fn test2 [expected input]
19   (assert (= expected (solve2 input))))
20
21 (test2 6 test-input)
22
23 (solve2 (aoc.string-from "2020/06.inp"))
```

```
3570
```

DONE Day 7.1

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.)

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["light red bags contain 1 bright white
 ↪  bag, 2 muted yellow bags."
```

```
5           "dark orange bags contain 3 bright
    ↪  white bags, 4 muted yellow bags."
6           "bright white bags contain 1 shiny
    ↪  gold bag."
7           "muted yellow bags contain 2 shiny
    ↪  gold bags, 9 faded blue bags."
8           "shiny gold bags contain 1 dark olive
    ↪  bag, 2 vibrant plum bags."
9           "dark olive bags contain 3 faded blue
    ↪  bags, 4 dotted black bags."
10          "vibrant plum bags contain 5 faded
    ↪  blue bags, 6 dotted black bags."
11          "faded blue bags contain no other
    ↪  bags."
12          "dotted black bags contain no other
    ↪  bags."])
13
14  (fn read-line [line]
15    (case (aoc.string-split line " ")
16      [t11 t12 "bags" "contain" "no" "other"
    ↪  "bags."]
17      [(.. t11 t12) {}]
18      [t11 t12 "bags" "contain" n2 t21 t22
    ↪  "bags."]
19      [(.. t11 t12) {(.. t21 t22) (tonumber
    ↪  n2)}]
```

```
20      [t11 t12 "bags" "contain" n2 t21 t22
    ↪   "bags," n3 t31 t32 "bags."]
21      [(.. t11 t12) {(.. t21 t22) (tonumber n2)
    ↪   (.. t31 t32) (tonumber n3)}]
22      [t11 t12 "bags" "contain" n2 t21 t22
    ↪   "bags," n3 t31 t32 "bags," n4 t41 t42
    ↪   "bags."]
23      [(.. t11 t12) {(.. t21 t22) (tonumber n2)
    ↪   (.. t31 t32) (tonumber n3) (.. t41 t42)
    ↪   (tonumber n4)}]
24      [t11 t12 "bags" "contain" n2 t21 t22
    ↪   "bags," n3 t31 t32 "bags," n4 t41 t42
    ↪   "bags," n5 t51 t52 "bags."]
25      [(.. t11 t12) {(.. t21 t22) (tonumber n2)
    ↪   (.. t31 t32) (tonumber n3) (.. t41 t42)
    ↪   (tonumber n4) (.. t51 t52) (tonumber n5)}]
26      _ (do (print (..  "W:" line)) nil)))
27
28  (fn bag2bags [line]
29    (string.gsub (string.gsub line "bag,"
    ↪   "bags,") "bag%." "bags%."))
30
31  (fn read-lines [lines]
32    (let [res {}]
33      (each [_ line (ipairs lines)]
```

```
      (let [rule (read-line (bag2bags line))]
        (tset res (. rule 1) (. rule 2))))
    res))

(fn search-symbol [tree symbol]
  (let [res []]
    (each [k v (pairs tree)]
      (when (. v symbol)
        (table.insert res k)))
    res))

(fn solve [lines]
  (let [tree (read-lines lines)
        search [:shinygold]
        found []]
    (while (< 0 (length search))
      (let [child (table.remove search 1)
            parents (search-symbol tree
 ↪  child)]
        (when (< 0 (length parents))
          (lume.map parents
                    (fn [p]
                      (when (not (lume.find
 ↪  found p))
                        (table.insert search
 ↪  p)
```

```
57                          (table.insert found
    ↪ p)))))))
58     (length found)))
59
60 (fn test [expected input]
61   (assert (= expected (solve input))))
62
63 (test 4 test-input)
64
65 (solve (aoc.string-from "2020/07.inp"))
```

```
128
```

DONE Day 7.2

It's getting pretty expensive to fly these days - not because of ticket prices, but because of the ridiculous number of bags you need to buy!

Consider again your `shiny gold` bag and the rules from the above example:

- `faded blue` bags contain 0 other bags.
- `dotted black` bags contain 0 other bags.

- `vibrant plum` bags contain 11 other bags: 5 `faded blue` bags and 6 `dotted black` bags.
- `dark olive` bags contain 7 other bags: 3 `faded blue` bags and 4 `dotted black` bags.

So, a single `shiny gold` bag must contain 1 `dark olive` bag (and the 7 bags within it) plus 2 `vibrant plum` bags (and the 11 bags within each of those): `1 + 1*7 + 2 + 2*11 = 32` bags!

Of course, the actual rules have a small chance of going several levels deeper than this example; be sure to count all of the bags, even if the nesting becomes topologically impractical!

Here's another example:

```
shiny gold bags contain 2 dark red bags.
dark red bags contain 2 dark orange bags.
dark orange bags contain 2 dark yellow bags.
dark yellow bags contain 2 dark green bags.
dark green bags contain 2 dark blue bags.
dark blue bags contain 2 dark violet bags.
dark violet bags contain no other bags.
```

In this example, a single `shiny gold` bag must contain 126 other bags.

How many individual bags are required inside your single `shiny gold` bag?

```
(local test2-input
       ["shiny gold bags contain 2 dark red bags."
        "dark red bags contain 2 dark orange bags."
        "dark orange bags contain 2 dark yellow bags."
        "dark yellow bags contain 2 dark green bags."
        "dark green bags contain 2 dark blue bags."
        "dark blue bags contain 2 dark violet bags."
        "dark violet bags contain no other bags."])

(macro times [t body1 & rest-body]
  `(fcollect [i# 1 ,t 1]
     (do ,body1 ,(unpack rest-body))))

(fn solve2 [lines]
  (let [tree (read-lines lines)
        search [:shinygold]
```

```
        found []]
    (while (< 0 (length search))
      (let [root (table.remove search 1)]
        (each [k v (pairs (. tree root))]
          (times v (table.insert search k))
          (times v (table.insert found k)))))
    (length found)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 32 test-input)

(test2 126 test2-input)

(solve2 (aoc.string-from "2020/07.inp"))
```

```
20189
```

DONE Day 8.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["nop +0"
        "acc +1"
        "jmp +4"
        "acc +3"
        "jmp -3"
```

```
        "acc -99"
        "acc +1"
        "jmp -4"
        "acc +6"])

(fn solve [input]
  (var pos 1)
  (var res 0)
  (let [torun (aoc.range-to 1 (length input))]
    (while (lume.find torun pos)
      (aoc.table-remove torun pos)
      (case (aoc.string-split (. input pos) "
↪  ")
        ["nop" a] (set pos (+ pos 1))
        ["acc" b] (do
                    (set pos (+ pos 1))
                    (set res (+ res (tonumber
↪  b))))
        ["jmp" c] (set pos (+ pos (tonumber
↪  c))))))
  res)

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(test 5 test-input)

(solve (aoc.string-from "2020/08.inp"))
```

```
1614
```

DONE Day 8.2

After some careful analysis, you believe that exactly one instruction is corrupted.

Somewhere in the program, either a jmp is supposed to be a nop, or a nop is supposed to be a jmp. (No acc instructions were harmed in the corruption of this boot code.)

The program is supposed to terminate by attempting to execute an instruction immediately after the last instruction in the file. By changing exactly one jmp or nop, you can repair the boot code and make it terminate correctly.

For example, consider the same program from above:

```
nop +0
acc +1
jmp +4
acc +3
```

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

If you change the first instruction from nop +0 to jmp +0, it would create a single-instruction infinite loop, never leaving that instruction. If you change almost any of the jmp instructions, the program will still eventually find another jmp instruction and loop forever.

However, if you change the second-to-last instruction (from jmp -4 to nop -4), the program terminates! The instructions are visited in this order:

```
nop +0  | 1
acc +1  | 2
jmp +4  | 3
acc +3  |
jmp -3  |
acc -99 |
acc +1  | 4
nop -4  | 5
acc +6  | 6
```

After the last instruction (acc +6), the program terminates by attempting to run the instruction below the last instruction in the file. With this change, after the program terminates, the accumulator contains the value 8 (acc +1, acc +1, acc +6).

Fix the program so that it terminates normally by changing exactly one jmp (to nop) or nop (to jmp). What is the value of the accumulator after the program terminates?

```
(fn run [input]
  (local len (length input))
  (var pos 1)
  (var res 0)
  (let [torun (aoc.range-to 1 len)]
    (while (lume.find torun pos)
      (aoc.table-remove torun pos)
      (case (aoc.string-split (. input pos) "
↪ ")
        ["nop" a] (set pos (+ pos 1))
        ["NOP" A] (set pos (+ pos (tonumber
↪ A)))
        ["acc" b] (do
                    (set pos (+ pos 1))
                    (set res (+ res (tonumber
↪ b))))
```

```
14          ["jmp" c] (set pos (+ pos (tonumber
    ↪  c)))
15          ["JMP" C] (set pos (+ pos 1)))))
16    (if (> pos len)
17        res
18        -1))
19
20  (fn fix-first [input]
21    (var done false)
22    (each [i v (ipairs input) &until done]
23      (case (string.sub v 1 3)
24        "jmp" (do (set done true)
25                  (aoc.table-replace-row input i
    ↪  (.. "JMP" (string.sub v 4))))
26        "nop" (do (set done true)
27                  (aoc.table-replace-row input i
    ↪  (.. "NOP" (string.sub v 4))))))
28    input)
29
30  (fn fix [input]
31    (var found false)
32    (var replaced false)
33    (each [i v (ipairs input) &until (and found
    ↪  replaced)]
34      (case (string.sub v 1 3)
```

```
35        "JMP" (do (set found true)
36                  (aoc.table-replace-row input i
    ↪  (.. "jmp" (string.sub v 4))))
37        "NOP" (do (set found true)
38                  (aoc.table-replace-row input i
    ↪  (.. "nop" (string.sub v 4))))
39        "jmp" (when found
40                (set replaced true)
41                (aoc.table-replace-row input i
    ↪  (.. "JMP" (string.sub v 4))))
42        "nop" (when found
43                (set replaced true)
44                (aoc.table-replace-row input i
    ↪  (.. "NOP" (string.sub v 4))))))
45    input)
46
47  (fn solve2 [input]
48    (var fixed (fix-first input))
49    (var acc (run fixed))
50    (while (< acc 0)
51      (set fixed (fix fixed))
52      (set acc (run fixed)))
53    acc)
54
55  (fn test2 [expected input]
```

```
56    (assert (= expected (solve2 input))))
57
58  (test2 8 test-input)
59
60  (solve2 (aoc.string-from "2020/08.inp"))
```

```
1260
```

DONE Day 9.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["35"
        "20"
        "15"
        "25"
        "47"
        "40"
        "62"
        "55"
        "65"
        "95"
        "102"
```

```
        "117"
        "150"
        "182"
        "127"
        "219"
        "299"
        "277"
        "309"
        "576"])

(fn any-two-sum [xs x]
  (if (= 0 (length xs))
      0
      (lume.any (aoc.table-range xs 2 (length xs))
                (fn [e] (= x (+ e (. xs 1)))))
      x
      (any-two-sum (aoc.table-range xs 2 (length xs)) x)))

(fn solve [input preamble]
  (var done false)
  (let [numbers (lume.map input tonumber)]
    (for [i (+ 1 preamble) (length numbers) &until done]
```

```
37      (let [xs (aoc.table-range numbers (- i
    ↪  preamble ) (- i 1))
38            sum (any-two-sum xs (. numbers
    ↪  i))]
39        (when (= 0 sum)
40          (set done (. numbers i))))))
41  done)
42
43 (fn test [expected input]
44   (assert (= expected (solve input 5))))
45
46 (test 127 test-input)
47
48 (solve (aoc.string-from "2020/09.inp") 25)
```

```
18272118
```

DONE Day 9.2

The final step in breaking the XMAS encryption relies on the invalid number you just found: you must find a contiguous set of at least two numbers in your list which sum to the invalid number from step 1.

Again consider the above example:

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

In this list, adding up all of the numbers from 15 through 40 produces the invalid number from step 1, 127. (Of course, the contiguous set of numbers in your actual list might be much longer.)

To find the encryption weakness, add together the smallest and largest number in this contiguous range; in this example, these are 15 and 47, producing 62.

What is the encryption weakness in your XMAS-encrypted list of numbers?

```
(fn solve2 [input x]
  (var over false)
  (var res 0)
  (let [xs (lume.map input tonumber)]
    (for [i 1 (- (length xs) 1) &until (< 0 res)]
      (set over false)
      (for [j (+ 1 i) (length xs) &until over]
        (let [sum (aoc.table-sum (aoc.table-range xs i j))]
          (if (< x sum)
              (set over true)
              (= x sum)
              (do (set over true)
                  (set res (+
                            (let [xij (aoc.table-sort (aoc.table-range xs i j))]
                              (+ (. xij 1) (. xij (length xij))))))))))))
```

```
16    res)
17
18  (fn test2 [expected input sum]
19    (assert (= expected (solve2 input sum))))
20
21  (test2 62 test-input 127)
22
23  (solve2 (aoc.string-from "2020/09.inp")
    ↪  18272118)
```

```
2186361
```

2019 [14/50]

DONE Day 1.1

Santa has become stranded at the edge of the Solar System while delivering presents to other planets! To accurately calculate his position in space, safely align his warp drive, and return to Earth in time to save Christmas, he needs you to bring him measurements from fifty stars.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the Advent calendar; the second puzzle is

unlocked when you complete the first. Each puzzle grants one star. Good luck!

The Elves quickly load you into a spacecraft and prepare to launch.

At the first Go / No Go poll, every Elf is Go until the Fuel Counter-Upper. They haven't determined the amount of fuel required yet.

Fuel required to launch a given module is based on its mass. Specifically, to find the fuel required for a module, take its mass, divide by three, round down, and subtract 2.

For example:

- For a mass of 12, divide by 3 and round down to get 4, then subtract 2 to get 2.
- For a mass of 14, dividing by 3 and rounding down still yields 4, so the fuel required is also 2.
- For a mass of 1969, the fuel required is 654.
- For a mass of 100756, the fuel required is 33583.

The Fuel Counter-Upper needs to know the total fuel requirement. To find it, individually calculate the fuel needed for the mass of each module (your puzzle input), then add together all

the fuel values.

What is the sum of the fuel requirements for all of the modules on your spacecraft?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["12" "14" "1969" "100756"])

(fn fuel [mass]
  (- (aoc.int/ mass 3) 2))

(fn solve [input]
  (let [xs (lume.map input tonumber)]
    (accumulate [sum 0 _ x (ipairs xs)]
      (+ sum (fuel x)))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test (+ 2 2 654 33583) test-input)

(solve (aoc.string-from "2019/01.inp"))
```

```
3560353
```

DONE Day 1.2

During the second Go / No Go poll, the Elf in charge of the Rocket Equation Double-Checker stops the launch sequence. Apparently, you forgot to include additional fuel for the fuel you just added.

Fuel itself requires fuel just like a module - take its mass, divide by three, round down, and subtract 2. However, that fuel also requires fuel, and that fuel requires fuel, and so on. Any mass that would require negative fuel should instead be treated as if it requires zero fuel; the remaining mass, if any, is instead handled by wishing really hard, which has no mass and is outside the scope of this calculation.

So, for each module mass, calculate its fuel and add it to the total. Then, treat the fuel amount you just calculated as the input mass and repeat the process, continuing until a fuel requirement is zero or negative. For example:

- A module of mass 14 requires 2 fuel. This fuel requires no further fuel (2 divided by 3 and rounded down is 0, which would call for a negative fuel), so the total fuel required is still just 2.
- At first, a module of mass 1969 requires 654 fuel. Then,

this fuel requires 216 more fuel (654 / 3 - 2). 216 then requires 70 more fuel, which requires 21 fuel, which requires 5 fuel, which requires no further fuel. So, the total fuel required for a module of mass 1969 is 654 + 216 + 70 + 21 + 5 = 966.

- The fuel required by a module of mass 100756 and its fuel is: 33583 + 11192 + 3728 + 1240 + 411 + 135 + 43 + 12 + 2 = 50346.

What is the sum of the fuel requirements for all of the modules on your spacecraft when also taking into account the mass of the added fuel? (Calculate the fuel requirements for each module separately, then add them all up at the end.)

```
(fn added-fuel [mass]
  (let [res (aoc.lazy-seq [mass] fuel)]
    (aoc.table-sum (aoc.table-range res 2
 ↪  (length res)))))

(fn solve2 [input]
  (let [xs (lume.map input tonumber)]
    (accumulate [sum 0 _ x (ipairs xs)]
      (+ sum (added-fuel x)))))

(fn test2 [expected input]
```

```
11    (assert (= expected (solve2 input))))
12
13  (test2 (+ 2 2 966 50346) test-input)
14
15  (solve2 (aoc.string-from "2019/01.inp"))
```

```
5337642
```

DONE Day 2.1

On the way to your gravity assist around the Moon, your ship computer beeps angrily about a ”1202 program alarm”. On the radio, an Elf is already explaining how to handle the situation: ”Don't worry, that's perfectly norma–” The ship computer bursts into flames.

You notify the Elves that the computer's magic smoke seems to have escaped. ”That computer ran Intcode programs like the gravity assist program it was working on; surely there are enough spare parts up there to build a new Intcode computer!”

An Intcode program is a list of integers separated by commas (like 1,0,0,3,99). To run one, start by looking at the first integer (called position 0). Here, you will find an opcode - either

1, 2, or 99. The opcode indicates what to do; for example, 99 means that the program is finished and should immediately halt. Encountering an unknown opcode means something went wrong.

Opcode 1 adds together numbers read from two positions and stores the result in a third position. The three integers immediately after the opcode tell you these three positions - the first two indicate the positions from which you should read the input values, and the third indicates the position at which the output should be stored.

For example, if your Intcode computer encounters 1,10,20,30, it should read the values at positions 10 and 20, add those values, and then overwrite the value at position 30 with their sum.

Opcode 2 works exactly like opcode 1, except it multiplies the two inputs instead of adding them. Again, the three integers after the opcode indicate where the inputs and outputs are, not their values.

Once you're done processing an opcode, move to the next one by stepping forward 4 positions.

For example, suppose you have the following program:

```
1,9,10,3,2,3,11,0,99,30,40,50
```

For the purposes of illustration, here is the same program split into multiple lines:

```
1,9,10,3,
2,3,11,0,
99,
30,40,50
```

The first four integers, 1,9,10,3, are at positions 0, 1, 2, and 3. Together, they represent the first opcode (1, addition), the positions of the two inputs (9 and 10), and the position of the output (3). To handle this opcode, you first need to get the values at the input positions: position 9 contains 30, and position 10 contains 40. Add these numbers together to get 70. Then, store this value at the output position; here, the output position (3) is at position 3, so it overwrites itself. Afterward, the program looks like this:

```
1,9,10,70,
2,3,11,0,
99,
30,40,50
```

Step forward 4 positions to reach the next opcode, 2. This opcode works just like the previous, but it multiplies instead of adding. The inputs are at positions 3 and 11; these positions

contain 70 and 50 respectively. Multiplying these produces 3500; this is stored at position 0:

```
3500,9,10,70,
2,3,11,0,
99,
30,40,50
```

Stepping forward 4 more positions arrives at opcode 99, halting the program.

Here are the initial and final states of a few more small programs:

- 1,0,0,0,99 becomes 2,0,0,0,99 (1 + 1 = 2).
- 2,3,0,3,99 becomes 2,3,0,6,99 (3 * 2 = 6).
- 2,4,4,5,99,0 becomes 2,4,4,5,99,9801 (99 * 99 = 9801).
- 1,1,1,4,99,5,6,0,99 becomes 30,1,1,4,2,5,6,0,99.

Once you have a working computer, the first step is to restore the gravity assist program (your puzzle input) to the ”1202 program alarm” state it had just before the last computer caught fire. To do this, before running the program, replace position 1 with the value 12 and replace position 2 with the value 2. What value is left at position 0 after the program halts?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn return-code [xs]
  (var pos 1)
  (var done false)
  (while (not done)
    (if (= 99 (. xs pos))
        (set done true)
        (let [res (+ 1 (. xs (+ pos 3)))
              lar (. xs (+ 1 (. xs (+ pos 1))))
              rar (. xs (+ 1 (. xs (+ pos 2))))]
          (case (. xs pos)
            1  (aoc.table-swap xs res (+ lar rar))
            2  (aoc.table-swap xs res (* lar rar)))))
    (when (not done)
      (set pos (+ 4 pos))))
  (. xs 1))

(fn restore-gravity-assist [input]
  (let [xs (aoc.string-tonumarray input)]
```

```
    (aoc.table-swap xs 2 12)
    (aoc.table-swap xs 3 2)
    (return-code xs)))

(fn solve [input]
  (restore-gravity-assist (. input 1)))

(fn test [expected input]
  (let [xs (aoc.string-tonumarray input)]
    (assert (= expected (return-code xs)))))

(test 3500 "1,9,10,3,2,3,11,0,99,30,40,50")
(test 2 "1,0,0,0,99")
(test 2 "2,3,0,3,99")
(test 2 "2,4,4,5,99,0")
(test 30 "1,1,1,4,99,5,6,0,99")

(solve (aoc.string-from "2019/02.inp"))
```

```
7594646
```

DONE Day 2.2

”Good, the new computer seems to be working correctly! Keep it nearby during this mission - you'll probably use it again. Real

Intcode computers support many more features than your new one, but we'll let you know what they are as you need them."

"However, your current priority should be to complete your gravity assist around the Moon. For this mission to succeed, we should settle on some terminology for the parts you've already built."

Intcode programs are given as a list of integers; these values are used as the initial state for the computer's `memory`. When you run an Intcode program, make sure to start by initializing memory to the program's values. A position in memory is called an `address` (for example, the first value in memory is at "address 0").

Opcodes (like 1, 2, or 99) mark the beginning of an instruction. The values used immediately after an opcode, if any, are called the instruction's `parameters`. For example, in the instruction 1,2,3,4, 1 is the opcode; 2, 3, and 4 are the parameters. The instruction 99 contains only an opcode and has no parameters.

The address of the current instruction is called the instruction pointer; it starts at 0. After an instruction finishes, the instruction pointer increases by the `number of values in`

`the instruction`; until you add more instructions to the computer, this is always 4 (1 opcode + 3 parameters) for the add and multiply instructions. (The halt instruction would increase the instruction pointer by 1, but it halts the program instead.)

"With terminology out of the way, we're ready to proceed. To complete the gravity assist, you need to determine `what pair of inputs produces the output` 19690720."

The inputs should still be provided to the program by replacing the values at addresses 1 and 2, just like before. In this program, the value placed in address 1 is called the noun, and the value placed in address 2 is called the `verb`. Each of the two input values will be between 0 and 99, inclusive.

Once the program has halted, its output is available at address 0, also just like before. Each time you try a pair of inputs, make sure you first `reset the computer's memory to the values in the program` (your puzzle input) - in other words, don't reuse memory from a previous attempt.

Find the input noun and verb that cause the program to produce the output 19690720. `What is 100 * noun + verb?` (For example, if noun=12 and verb=2, the answer

would be 1202.)

```
(fn complete-gravity-assist [input]
  (var done false)
  (for [i 0 99 &until done]
    (for [j 0 99 &until done]
      (let [xs (aoc.string-tonumarray input)]
        (aoc.table-swap xs 2 i)
        (aoc.table-swap xs 3 j)
        (when (= 19690720 (return-code xs))
          (set done (+ (* 100 i) j))))))
  done)

(fn solve2 [input]
  (complete-gravity-assist (. input 1)))

(solve2 (aoc.string-from "2019/02.inp"))
```

```
3376
```

DONE Day 3.1

The gravity assist was successful, and you're well on your way to the Venus refuelling station. During the rush back on Earth, the fuel management system wasn't completely installed, so

that's next on the priority list.

Opening the front panel reveals a jumble of wires. Specifically, two wires are connected to a central port and extend outward on a grid. You trace the path each wire takes as it leaves the central port, one wire per line of text (your puzzle input).

The wires twist and turn, but the two wires occasionally cross paths. To fix the circuit, you need to find the intersection point closest to the central port. Because the wires are on a grid, use the Manhattan distance for this measurement. While the wires do technically cross right at the central port where they both start, this point does not count, nor does a wire count as crossing with itself.

For example, if the first wire's path is R8,U5,L5,D3, then starting from the central port (o), it goes right 8, up 5, left 5, and finally down 3:

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

```
.........|.
.o-------+.
...........
```

Then, if the second wire's path is U7,R6,D4,L4, it goes up 7, right 6, down 4, and left 4:

```
...........
.+-----+...
.|.....|...
.|..+--X-+.
.|..|..|.|.
.|.-X--+.|.
.|..|....|.
.|.......|.
.o-------+.
...........
```

These wires cross at two locations (marked X), but the lower-left one is closer to the central port: its distance is 3 + 3 = 6.

Here are a few more examples:

- R75,D30,R83,U83,L12,D49,R71,U7,L72
- U62,R66,U55,R34,D71,R55,D58,R83 = distance 159
- R98,U47,R26,D63,R33,U87,L62,D20,R33,U53,R51
- U98,R91,D20,R16,D67,R40,U7,R15,U6,R7 = distance 135

What is the Manhattan distance from the central port to the closest intersection?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test1-input
       ["R8,U5,L5,D3"
        "U7,R6,D4,L4"])
(local test2-input
       ["R75,D30,R83,U83,L12,D49,R71,U7,L72"
        "U62,R66,U55,R34,D71,R55,D58,R83"])
(local test3-input
  ↪ ["R98,U47,R26,D63,R33,U87,L62,D20,R33,U53,R51"
  ↪ "U98,R91,D20,R16,D67,R40,U7,R15,U6,R7"])

(fn find-cross [l1 l2]
  (let [res []]
    (for [i 3 (length l1)]
      (for [j 3 (length l2)]
        (let [s1 [(. l1 (- i 1)) (. l1 i)]
              s2 [(. l2 (- j 1)) (. l2 j)]
              found (aoc.intersection s1 s2)]
          (when found
            (table.insert res found)))))
```

```
    res))

(fn solve [input]
  (let [line1 (aoc.decartian (. input 1))
        line2 (aoc.decartian (. input 2))
        res (find-cross line1 line2)]
    (aoc.math-min (lume.map res (fn [e]
    ↪  (aoc.manhattan-dist [0 0] e))))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 6 test1-input)
(test 159 test2-input)
(test 135 test3-input)
(solve (aoc.string-from "2019/03.inp"))
```

```
209
```

DONE Day 3.2

It turns out that this circuit is very timing-sensitive; you actually need to `minimize the signal delay`.

To do this, calculate the number of steps each wire takes to reach each intersection; choose the intersection where the

sum of both wires' steps is lowest. If a wire visits a position on the grid multiple times, use the steps value from the first time it visits that position when calculating the total value of a specific intersection.

The number of steps a wire takes is the total number of grid squares the wire has entered to get to that location, including the intersection being considered. Again consider the example from above:

```
...........
.+-----+...
.|.....|...
.|..+--X-+.
.|..|..|.|.
.|.-X--+.|.
.|..|....|.
.|.......|.
.o-------+.
...........
```

In the above example, the intersection closest to the central port is reached after 8+5+5+2 = 20 steps by the first wire and 7+6+4+3 = 20 steps by the second wire for a total of 20+20 = 40 steps.

However, the top-right intersection is better: the first wire takes only 8+5+2 = 15 and the second wire takes only 7+6+2 = 15, a total of 15+15 = 30 steps.

Here are the best steps for the extra examples from above:

- R75,D30,R83,U83,L12,D49,R71,U7,L72
- U62,R66,U55,R34,D71,R55,D58,R83 = 610 steps
- R98,U47,R26,D63,R33,U87,L62,D20,R33,U53,R51
- U98,R91,D20,R16,D67,R40,U7,R15,U6,R7 = 410 steps

What is the fewest combined steps the wires must take to reach an intersection?

```
(fn linear-dist [p s]
  (var done false)
  (let [res []]
    (for [i 2 (length s) &until done]
      (let [s0 (. s (- i 1))
            s1 (. s i)]
        (set done (aoc.in-segment? p [s0 s1]))
        (if done
            (table.insert res
  ↪ (aoc.manhattan-dist s0 p))
            (table.insert res
  ↪ (aoc.manhattan-dist s0 s1)))))
```

```
    (aoc.table-sum res)))

(fn solve2 [input]
  (let [line1 (aoc.decartian (. input 1))
        line2 (aoc.decartian (. input 2))
        res (find-cross line1 line2)]
    (aoc.math-min
     (lume.map res (fn [e] (+ (linear-dist e line1)
                              (linear-dist e line2)))))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 30 test1-input)
(test2 610 test2-input)
(test2 410 test3-input)
(solve2 (aoc.string-from "2019/03.inp"))
```

```
43258
```

DONE Day 4.1

You arrive at the Venus fuel depot only to discover it's protected by a password. The Elves had written the password on a sticky note, but someone threw it out.

However, they do remember a few key facts about the password:

- It is a six-digit number.
- The value is within the range given in your puzzle input.
- Two adjacent digits are the same (like 22 in 122345).
- Going from left to right, the digits never decrease; they only ever increase or stay the same (like 111123 or 135679).

Other than the range rule, the following are true:

- 111111 meets these criteria (double 11, never decreases).
- 223450 does not meet these criteria (decreasing pair of digits 50).
- 123789 does not meet these criteria (no double).

How many different passwords within 134792-675810 range meet these criteria?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn identity [a b]
  (= a b))

(fn adjacent? [xs]
  (let [bins (aoc.partition-by xs identity)]
    (lume.any bins #(< 1 (length $)))))

(fn never-decrease? [xs]
  (var res true)
  (for [i 2 (length xs) &until (not res)]
    (when (< (. xs i)
             (. xs (- i 1)))
      (set res false)))
  res)

(fn solve [f t]
  (var res 0)
  (for [i f t]
    (let [xs (aoc.toarray i)]
      (when (and (adjacent? xs)
                 (never-decrease? xs))
        (set res (+ 1 res))))))
```

```
  res)

(solve 134792 675810)
```

```
1955
```

DONE Day 4.2

An Elf just remembered one more important detail: the two adjacent matching digits are not part of a larger group of matching digits.

Given this additional criterion, but still ignoring the range rule, the following are now true:

- 112233 meets these criteria because the digits never decrease and all repeated digits are exactly two digits long.
- 123444 no longer meets the criteria (the repeated 44 is part of a larger group of 444).
- 111122 meets the criteria (even though 1 is repeated more than twice, it still contains a double 22).

How many different passwords within the range given in your puzzle input meet all of the criteria?

```
(fn two-adjacent? [xs]
  (let [bins (aoc.partition-by xs identity)]
    (lume.any bins #(= 2 (length $)))))

(fn solve2 [f t]
  (var res 0)
  (for [i f t]
    (let [xs (aoc.toarray i)]
      (when (and (two-adjacent? xs)
                 (never-decrease? xs))
        (set res (+ 1 res)))))
  res)

(solve2 134792 675810)
```

```
1319
```

DONE Day 5.1

You're starting to sweat as the ship makes its way toward Mercury. The Elves suggest that you get the air conditioner working by upgrading your ship computer to support the Thermal Environment Supervision Terminal.

The Thermal Environment Supervision Terminal (TEST) starts

by running a diagnostic program (your puzzle input). The TEST diagnostic program will run on your existing Intcode computer after a few modifications:

First, you'll need to add two new instructions:

- Opcode 3 takes a single integer as input and saves it to the position given by its only parameter. For example, the instruction 3,50 would take an input value and store it at address 50.
- Opcode 4 outputs the value of its only parameter. For example, the instruction 4,50 would output the value at address 50.

Programs that use these instructions will come with documentation that explains what should be connected to the input and output. The program 3,0,4,0,99 outputs whatever it gets as input, then halts.

Second, you'll need to add support for parameter modes:

Each parameter of an instruction is handled based on its parameter mode. Right now, your ship computer already understands parameter mode 0, position mode, which causes the parameter to be interpreted as a position - if the parameter is 50, its

value is the value stored at address 50 in memory. Until now, all parameters have been in position mode.

Now, your ship computer will also need to handle parameters in mode 1, immediate mode. In immediate mode, a parameter is interpreted as a value - if the parameter is 50, its value is simply 50.

Parameter modes are stored in the same value as the instruction's opcode. The opcode is a two-digit number based only on the ones and tens digit of the value, that is, the opcode is the rightmost two digits of the first value in an instruction. Parameter modes are single digits, one per parameter, read right-to-left from the opcode: the first parameter's mode is in the hundreds digit, the second parameter's mode is in the thousands digit, the third parameter's mode is in the ten-thousands digit, and so on. Any missing modes are 0.

For example, consider the program 1002,4,3,4,33.

The first instruction, 1002,4,3,4, is a multiply instruction - the rightmost two digits of the first value, 02, indicate opcode 2, multiplication. Then, going right to left, the parameter modes are 0 (hundreds digit), 1 (thousands digit), and 0 (ten-thousands digit, not present and therefore zero):

```
ABCDE
 1002

DE - two-digit opcode,      02 == opcode 2
 C - mode of 1st parameter,  0 == position
 ↪  mode
 B - mode of 2nd parameter,  1 == immediate
 ↪  mode
 A - mode of 3rd parameter,  0 == position
 ↪  mode,
                                  omitted due
 ↪  to being a leading zero
```

This instruction multiplies its first two parameters. The first parameter, 4 in position mode, works like it did before - its value is the value stored at address 4 (33). The second parameter, 3 in immediate mode, simply has value 3. The result of this operation, 33 * 3 = 99, is written according to the third parameter, 4 in position mode, which also works like it did before - 99 is written to address 4.

Parameters that an instruction writes to will never be in immediate mode.

Finally, some notes:

- It is important to remember that the instruction pointer

should increase by the number of values in the instruction after the instruction finishes. Because of the new instructions, this amount is no longer always 4.
- Integers can be negative: 1101,100,-1,4,0 is a valid program (find 100 + -1, store the result in position 4).

The TEST diagnostic program will start by requesting from the user the ID of the system to test by running an input instruction - provide it 1, the ID for the ship's air conditioner unit.

It will then perform a series of diagnostic tests confirming that various parts of the Intcode computer, like parameter modes, function correctly. For each test, it will run an output instruction indicating how far the result of the test was from the expected value, where 0 means the test was successful. Non-zero outputs mean that a function is not working correctly; check the instructions that were run before the output instruction to see which one failed.

Finally, the program will output a diagnostic code and immediately halt. This final output isn't an error; an output followed immediately by a halt means the program finished. If all outputs were zero except the diagnostic code, the diagnostic program ran successfully.

After providing 1 to the only input instruction and passing all the tests, what diagnostic code does the program produce?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn intcode [xs input]
  (var pos 1)
  (var done false)
  (var output nil)
  (while (not done)
    (let [code (aoc.table-range xs pos (+ pos 3))]
      (case code
        [99]
        (set done true)
        [1 la ra re]
        (aoc.table-swap xs (+ 1 re) (+ (. xs (+ 1 la)) (. xs (+ 1 ra))))
        [1001 la ra re]
        (aoc.table-swap xs (+ 1 re) (+ (. xs (+ 1 la)) ra))
        [101 la ra re]
        (aoc.table-swap xs (+ 1 re) (+ la (. xs (+ 1 ra))))
        [1101 la ra re]
```

```
20          (aoc.table-swap xs (+ 1 re) (+ la ra))
21          [2 la ra re]
22          (aoc.table-swap xs (+ 1 re) (* (. xs
    ↪  (+ 1 la)) (. xs (+ 1 ra))))
23          [1002 la ra re]
24          (aoc.table-swap xs (+ 1 re) (* (. xs
    ↪  (+ 1 la)) ra))
25          [102 la ra re]
26          (aoc.table-swap xs (+ 1 re) (* la (.
    ↪  xs (+ 1 ra))))
27          [1102 la ra re]
28          (aoc.table-swap xs (+ 1 re) (* la ra))
29          [3 re _ _]
30          (aoc.table-swap xs (+ 1 re) input)
31          [4 re _ _]
32          (set output (. xs (+ 1 re)))
33          [104 re _ _]
34          (set output re)
35          _
36          (do (set done true)
37              (print (.. "W: no match at " pos
    ↪  ": " (. xs pos)))))
38        (when (not done)
39          (if (lume.any [3 4 104] #(= $ (. code
    ↪  1)))
```

```
            (set pos (+ 2 pos))
            (set pos (+ 4 pos))))))
  output)

(fn solve [input]
  (let [xs (aoc.string-tonumarray (. input
   ↪  1))]
    (intcode xs 1)))

(solve (aoc.string-from "2019/05.inp"))
```

```
6069343
```

DONE Day 5.2

The air conditioner comes online! Its cold air feels good for a while, but then the TEST alarms start to go off. Since the air conditioner can't vent its heat anywhere but back into the spacecraft, it's actually making the air inside the ship `warmer`.

Instead, you'll need to use the TEST to extend the thermal radiators. Fortunately, the diagnostic program (your puzzle input) is already equipped for this. Unfortunately, your Intcode computer is not.

Your computer is only missing a few opcodes:

- Opcode 5 is `jump-if-true`: if the first parameter is `non-zero`, it sets the instruction pointer to the value from the second parameter. Otherwise, it does nothing.
- Opcode 6 is `jump-if-false`: if the first parameter `is zero`, it sets the instruction pointer to the value from the second parameter. Otherwise, it does nothing.
- Opcode 7 is `less than`: if the first parameter `is less` than the second parameter, it stores 1 in the position given by the third parameter. Otherwise, it stores 0.
- Opcode 8 is `equals`: if the first parameter is `equal to` the second parameter, it stores 1 in the position given by the third parameter. Otherwise, it stores 0.

Like all instructions, these instructions need to support `parameter modes` as described above.

Normally, after an instruction is finished, the instruction pointer increases by the number of values in that instruction. `However`, if the instruction modifies the instruction pointer, that value is used and the instruction pointer is `not automatically increased`.

For example, here are several programs that take one input,

compare it to the value 8, and then produce one output:

- 3,9,8,9,10,9,4,9,99,-1,8 - Using `position mode`, consider whether the input is `equal to` 8; output 1 (if it is) or 0 (if it is not).
- 3,9,7,9,10,9,4,9,99,-1,8 - Using `position mode`, consider whether the input is `less than` 8; output 1 (if it is) or 0 (if it is not).
- 3,3,1108,-1,8,3,4,3,99 - Using `immediate mode`, consider whether the input is `equal to` 8; output 1 (if it is) or 0 (if it is not).
- 3,3,1107,-1,8,3,4,3,99 - Using `immediate mode`, consider whether the input is `less than` 8; output 1 (if it is) or 0 (if it is not).

Here are some jump tests that take an input, then output 0 if the input was zero or 1 if the input was non-zero:

- 3,12,6,12,15,1,13,14,13,4,13,99,-1,0,1,9 (using `position mode`)
- 3,3,1105,-1,9,1101,0,0,12,4,12,99,1 (using `immediate mode`)

Here's a larger example:

```
3,21,1008,21,8,20,1005,20,22,107,8,21,20,1006,20,31,1106,0
```

The above example program uses an input instruction to ask for a single number. The program will then output 999 if the input value is below 8, output 1000 if the input value is equal to 8, or output 1001 if the input value is greater than 8.

This time, when the TEST diagnostic program runs its input instruction to get the ID of the system to test, `provide it 5`, the ID for the ship's thermal radiator controller. This diagnostic test suite only outputs one number, the `diagnostic code`.

What is the diagnostic code for system ID 5?

```
(fn intcode-v2 [xs input]
  (var pos 1)
  (var done false)
  (var output nil)
  (while (not done)
    (let [code (aoc.table-range xs pos (+ pos
    3))]
      (case code
        [99]
        (set done true)
        [1 la ra re]
```

```
11          (aoc.table-swap xs (+ 1 re) (+ (. xs
    ↪  (+ 1 la)) (. xs (+ 1 ra))))
12          [101 la ra re]
13          (aoc.table-swap xs (+ 1 re) (+ la (.
    ↪  xs (+ 1 ra))))
14          [1001 la ra re]
15          (aoc.table-swap xs (+ 1 re) (+ (. xs
    ↪  (+ 1 la)) ra))
16          [1101 la ra re]
17          (aoc.table-swap xs (+ 1 re) (+ la ra))
18          [2 la ra re]
19          (aoc.table-swap xs (+ 1 re) (* (. xs
    ↪  (+ 1 la)) (. xs (+ 1 ra))))
20          [102 la ra re]
21          (aoc.table-swap xs (+ 1 re) (* la (.
    ↪  xs (+ 1 ra))))
22          [1002 la ra re]
23          (aoc.table-swap xs (+ 1 re) (* (. xs
    ↪  (+ 1 la)) ra))
24          [1102 la ra re]
25          (aoc.table-swap xs (+ 1 re) (* la ra))
26          [3 re _ _]
27          (aoc.table-swap xs (+ 1 re) input)
28          [4 re _ _]
29          (set output (. xs (+ 1 re)))
```

```
        [104 re _ _]
        (set output re)
        [5 la ra _]
        (when (not= 0 (. xs (+ 1 la)))
          (set pos (+ 1 (. xs (+ 1 ra))))
          (set pos (+ 3 pos)))
        [105 la ra _]
        (if (not= 0 la)
          (set pos (+ 1 (. xs (+ 1 ra))))
          (set pos (+ 3 pos)))
        [1005 la ra _]
        (if (not= 0 (. xs (+ 1 la)))
          (set pos (+ 1 ra))
          (set pos (+ 3 pos)))
        [1105 la ra _]
        (if (not= 0 la)
          (set pos (+ 1 ra))
          (set pos (+ 3 pos)))
        [6 la ra _]
        (if (= 0 (. xs (+ 1 la)))
          (set pos (+ 1 (. xs (+ 1 ra))))
          (set pos (+ 3 pos)))
        [106 la ra _]
        (if (= 0 la)
          (set pos (+ 1 (. xs (+ 1 ra)))))
```

```
          (set pos (+ 3 pos)))
        [1006 la ra _]
        (if (= 0 (. xs (+ 1 la)))
            (set pos (+ 1 ra))
            (set pos (+ 3 pos)))
        [1106 la ra _]
        (if (= 0 la)
            (set pos (+ 1 ra))
            (set pos (+ 3 pos)))
        [7 la ra re]
        (if (< (. xs (+ 1 la)) (. xs (+ 1
↪  ra)))
            (aoc.table-swap xs (+ 1 re) 1)
            (aoc.table-swap xs (+ 1 re) 0))
        [107 la ra re]
        (if (< la (. xs (+ 1 ra)))
            (aoc.table-swap xs (+ 1 re) 1)
            (aoc.table-swap xs (+ 1 re) 0))
        [1007 la ra re]
        (if (< (. xs (+ 1 la)) ra)
            (aoc.table-swap xs (+ 1 re) 1)
            (aoc.table-swap xs (+ 1 re) 0))
        [1107 la ra re]
        (if (< la ra)
            (aoc.table-swap xs (+ 1 re) 1)
```

```
79              (aoc.table-swap xs (+ 1 re) 0))
80          [8 la ra re]
81          (if (= (. xs (+ 1 la)) (. xs (+ 1
    ↪  ra)))
82              (aoc.table-swap xs (+ 1 re) 1)
83              (aoc.table-swap xs (+ 1 re) 0))
84          [108 la ra re]
85          (if (= la (. xs (+ 1 ra)))
86              (aoc.table-swap xs (+ 1 re) 1)
87              (aoc.table-swap xs (+ 1 re) 0))
88          [1008 la ra re]
89          (if (= (. xs (+ 1 la)) ra)
90              (aoc.table-swap xs (+ 1 re) 1)
91              (aoc.table-swap xs (+ 1 re) 0))
92          [1108 la ra re]
93          (if (= la ra)
94              (aoc.table-swap xs (+ 1 re) 1)
95              (aoc.table-swap xs (+ 1 re) 0))
96          _
97          (do (set done true)
98              (print (.. "W: no match at " pos
    ↪  ": " (. xs pos)))))
99        (when (not done)
100         (if (lume.any [3 4 104] #(= $ (. code
    ↪  1)))
```

```
            (set pos (+ 2 pos))
            (lume.any [1005 1105 105 5 1006
 ↪  1106 106 6]
                      #(= $ (. code 1)))
            nil
            (set pos (+ 4 pos))))))
  output)

(fn solve2 [lines input]
  (let [xs (aoc.string-tonumarray (. lines
 ↪  1))]
    (intcode-v2 xs input)))

(fn test [expected lines input]
  (assert (= expected (solve2 lines input))))

(local test-input
 ↪  ["3,21,1008,21,8,20,1005,20,22,107,8,21,20,1006,20,31,1
(test 999 test-input 7)
(test 1000 test-input 8)
(test 1001 test-input 9)

(local test2-input
 ↪  ["3,9,8,9,10,9,4,9,99,-1,8"])
(test 1 test2-input 8)
```

```
(test 0 test2-input 7)

(local test3-input
 ↪  ["3,9,7,9,10,9,4,9,99,-1,8"])
(test 0 test3-input 8)
(test 1 test3-input 7)

(local test4-input ["3,3,1108,-1,8,3,4,3,99"])
(test 1 test4-input 8)
(test 0 test4-input 7)

(local test5-input ["3,3,1107,-1,8,3,4,3,99"])
(test 0 test5-input 8)
(test 1 test5-input 7)

(local test6-input
 ↪  ["3,12,6,12,15,1,13,14,13,4,13,99,-1,0,1,9"])
(test 0 test6-input 0)
(test 1 test6-input 1)

(local test7-input
 ↪  ["3,3,1105,-1,9,1101,0,0,12,4,12,99,1"])
(test 0 test7-input 0)
(test 1 test7-input 1)

```

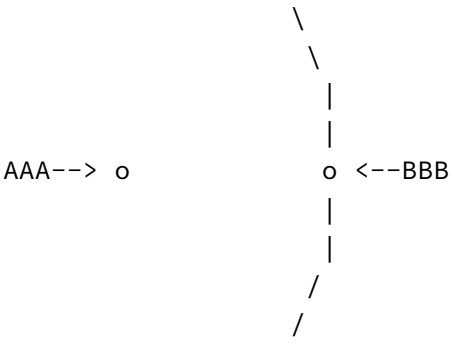
```
144 (solve2 (aoc.string-from "2019/05.inp") 5)
```

```
3188550
```

DONE Day 6.1

You've landed at the Universal Orbit Map facility on Mercury. Because navigation in space often involves transferring between orbits, the orbit maps here are useful for finding efficient routes between, for example, you and Santa. You download a map of the local orbits (your puzzle input).

Except for the universal Center of Mass (COM), every object in space is in orbit around exactly one other object. An orbit looks roughly like this:



In this diagram, the object BBB is in orbit around AAA. The path that BBB takes around AAA (drawn with lines) is only partly shown. In the map data, this orbital relationship is written AAA)BBB, which means ”BBB is in orbit around AAA”.

Before you use your map data to plot a course, you need to make sure it wasn't corrupted during the download. To verify maps, the Universal Orbit Map facility uses orbit count checksums - the total number of direct orbits (like the one shown above) and indirect orbits.

Whenever A orbits B and B orbits C, then A indirectly orbits C. This chain can be any number of objects long: if A orbits B, B orbits C, and C orbits D, then A indirectly orbits D.

For example, suppose you have the following map:

```
COM)B
B)C
C)D
D)E
E)F
B)G
G)H
D)I
E)J
```

```
J)K
K)L
```

Visually, the above map of orbits looks like this:

```
        G - H       J - K - L
       /           /
COM - B - C - D - E - F
               \
                I
```

In this visual representation, when two objects are connected by a line, the one on the right directly orbits the one on the left.

Here, we can count the total number of orbits as follows:

- D directly orbits C and indirectly orbits B and COM, a total of 3 orbits.
- L directly orbits K and indirectly orbits J, E, D, C, B, and COM, a total of 7 orbits.
- COM orbits nothing.

The total number of direct and indirect orbits in this example is 42.

What is the total number of direct and indirect orbits in your

map data?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["COM)B"
                   "B)C"
                   "C)D"
                   "D)E"
                   "E)F"
                   "B)G"
                   "G)H"
                   "D)I"
                   "E)J"
                   "J)K"
                   "K)L"])

(fn paths [xs]
  (let [res {}]
    (each [_ [k v] (ipairs xs)]
      (tset res v k))
    res))

(fn solve [input]
  (let [xs (paths (lume.map input
  ↪  #(aoc.string-split $ ")")))
```

```
23          ys (lume.map (aoc.keys xs) #(aoc.rank
    ↪  xs $))]
24      (accumulate [sum 0 _ y (ipairs ys)]
25        (+ sum y))))
26
27  (fn test [expected input]
28    (assert (= expected (solve input))))
29
30  (test 42 test-input)
31
32  (solve (aoc.string-from "2019/06.inp"))
```

```
151345
```

DONE Day 6.2

Now, you just need to figure out how many orbital transfers you (YOU) need to take to get to Santa (SAN).

You start at the object YOU are orbiting; your destination is the object SAN is orbiting. An orbital transfer lets you move from any object to an object orbiting or orbited by that object.

For example, suppose you have the following map:

```
COM)B
```

```
B)C
C)D
D)E
E)F
B)G
G)H
D)I
E)J
J)K
K)L
K)YOU
I)SAN
```

Visually, the above map of orbits looks like this:

```
                          YOU
                         /
        G - H       J - K - L
       /           /
COM - B - C - D - E - F
               \
                I - SAN
```

In this example, YOU are in orbit around K, and SAN is in orbit around I. To move from K to I, a minimum of 4 orbital transfers are required:

- K to J
- J to E
- E to D
- D to I

Afterward, the map of orbits looks like this:

```
         G - H       J - K - L
        /           /
COM - B - C - D - E - F
                \
                 I - SAN
                  \
                   YOU
```

What is the minimum number of orbital transfers required to move from the object YOU are orbiting to the object SAN is orbiting? (Between the objects they are orbiting - not between YOU and SAN.)

```
(local test2-input
       ["COM)B"
        "B)C"
        "C)D"
        "D)E"
        "E)F"
```

```
        "B)G"
        "G)H"
        "D)I"
        "E)J"
        "J)K"
        "K)L"
        "K)YOU"
        "I)SAN"])

(fn partial-paths [xs x y]
  (let [x1 (. xs x)
        y1 (. xs y)]
    (if (= x y) x
        (= x1 y1) x1
        (<= (aoc.rank xs x1) (aoc.rank xs y1))
        (partial-paths xs x1 (. xs y1))
        (partial-paths xs (. xs x1) y1))))

(fn solve2 [input]
  (let [xs (paths (lume.map input
 ↪  #(aoc.string-split $ ")")))
        x (partial-paths xs :YOU :SAN)]
    (- (+ (aoc.rank xs (. xs :SAN))
          (aoc.rank xs (. xs :YOU)))
       (* 2 (aoc.rank xs x)))))
```

```
31
32 (fn test2 [expected input]
33   (assert (= expected (solve2 input))))
34
35 (test2 4 test2-input)
36
37 (solve2 (aoc.string-from "2019/06.inp"))
```

```
391
```

DONE Day 8.1

The Elves' spirits are lifted when they realize you have an opportunity to reboot one of their Mars rovers, and so they are curious if you would spend a brief sojourn on Mars. You land your ship near the rover.

When you reach the rover, you discover that it's already in the process of rebooting! It's just waiting for someone to enter a BIOS password. The Elf responsible for the rover takes a picture of the password (your puzzle input) and sends it to you via the Digital Sending Network.

Unfortunately, images sent via the Digital Sending Network aren't encoded with any normal encoding; instead, they're

encoded in a special Space Image Format. None of the Elves seem to remember why this is the case. They send you the instructions to decode it.

Images are sent as a series of digits that each represent the color of a single pixel. The digits fill each row of the image left-to-right, then move downward to the next row, filling rows top-to-bottom until every pixel of the image is filled.

Each image actually consists of a series of identically-sized layers that are filled in this way. So, the first digit corresponds to the top-left pixel of the first layer, the second digit corresponds to the pixel to the right of that on the same layer, and so on until the last digit, which corresponds to the bottom-right pixel of the last layer.

For example, given an image 3 pixels wide and 2 pixels tall, the image data 123456789012 corresponds to the following image layers:

```
Layer 1: 123
         456

Layer 2: 789
         012
```

The image you received is 25 pixels wide and 6 pixels tall.

To make sure the image wasn't corrupted during transmission, the Elves would like you to find the layer that contains the fewest 0 digits. On that layer, what is the number of 1 digits multiplied by the number of 2 digits?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [input w h]
  (let [res {:k 100000000 :v 0}
        layers (aoc.table-group-by
 ↪  (aoc.string-toarray input) (* w h))]
    (each [_ layer (ipairs layers)]
      (let [zeros (length (lume.filter layer
 ↪  #(= "0" $)))
            ones (length (lume.filter layer
 ↪  #(= "1" $)))
            twos (length (lume.filter layer
 ↪  #(= "2" $)))]
        (when (< zeros (?. res :k))
            (tset res :k zeros)
            (tset res :v (* ones twos)))))
    (. res :v)))

```

```
(fn test [expected input w h]
  (assert (= expected (solve input w h))))

(test 1 "123456789012" 3 2)

(solve (. (aoc.string-from "2019/08.inp") 1)
 ↪  25 6)
```

```
1584
```

DONE Day 8.2

Now you're ready to decode the image. The image is rendered by stacking the layers and aligning the pixels with the same positions in each layer. The digits indicate the color of the corresponding pixel: 0 is black, 1 is white, and 2 is transparent.

The layers are rendered with the first layer in front and the last layer in back. So, if a given position has a transparent pixel in the first and second layers, a black pixel in the third layer, and a white pixel in the fourth layer, the final image would have a `black` pixel at that position.

For example, given an image 2 pixels wide and 2 pixels tall, the image data `022211222120000` corresponds to the follow-

ing image layers:

```
Layer 1:>02
         22

Layer 2: 11<
         22

Layer 3: 22
        >12

Layer 4: 00
         00<
```

Then, the full image can be found by determining the top visible pixel in each position:

- The top-left pixel is `black` because the top layer is 0.
- The top-right pixel is `white` because the top layer is 2 (transparent), but the second layer is 1.
- The bottom-left pixel is `white` because the top two layers are 2, but the third layer is 1.
- The bottom-right pixel is `black` because the only visible pixel in that position is 0 (from layer 4).

So, the final image looks like this:

```
01
10
```

What message is produced after decoding your image?

```
(fn color [xs]
  (var res false)
  (each [_ col (ipairs xs) &until res]
    (case col
      "1" (set res "1")
      "2" nil
      "0" (set res "0")))
  res)

(fn process [input w h]
  (let [res []
        layers (aoc.table-group-by
                (aoc.table-group-by
                 (aoc.string-toarray input) w)
 ↪  h)
        rows (lume.map
              (aoc.table-transpose layers)
              #(aoc.table-transpose $))]
    (each [_ columns (ipairs rows)]
      (each [_ pixels (ipairs columns)]
        (let [col (color pixels)]
```

```
          (table.insert res col))))
    (aoc.table-group-by res w)))

(fn test2 [expected input w h]
  (let [res (process input w h)
        str (aoc.table-tostring (lume.map res
 ↪  #(aoc.table-tostring $)))]
    (assert (= expected str))))

(test2 "0110" "0222112222120000" 2 2)

(fn solve2 [input w h]
  (let [inp (. input 1)
        res (process inp w h)]
    (aoc.matrix-print res)))

(solve2 (aoc.string-from "2019/08.inp") 25 6)
```

```
==    ==    ====      ====    ========    ====
 ↪
==  ==    ==    ==  ==    ==  ==        ==
 ↪  ==
====      ==        ==        ======    ==
 ↪
==  ==    ==        ==  ====  ==        ==
 ↪
```

```
==   ==     ==     ==   ==     ==   ==          ==
  ↪  ==
==     ==     ====       ======   ========     ====
  ↪
```

2018 [13/50]

DONE Day 1.1

”We've detected some temporal anomalies,” one of Santa's Elves at the Temporal Anomaly Research and Detection Instrument Station tells you. She sounded pretty worried when she called you down here. ”At 500-year intervals into the past, someone has been changing Santa's history!”

”The good news is that the changes won't propagate to our time stream for another 25 days, and we have a device” - she attaches something to your wrist - ”that will let you fix the changes with no such propagation delay. It's configured to send you 500 years further into the past every few days; that was the best we could do on such short notice.”

”The bad news is that we are detecting roughly fifty anomalies throughout time; the device will indicate fixed anomalies with

stars. The other bad news is that we only have one device and you're the best person for the job! Good lu–" She taps a button on the device and you suddenly feel like you're falling. To save Christmas, 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!

After feeling like you've been falling for a few minutes, you look at the device's tiny screen. "Error: Device must be calibrated before first use. Frequency drift detected. Cannot maintain destination lock." Below the message, the device shows a sequence of changes in frequency (your puzzle input). A value like +6 means the current frequency increases by 6; a value like -3 means the current frequency decreases by 3.

For example, if the device displays frequency changes of +1, -2, +3, +1, then starting from a frequency of zero, the following changes would occur:

- Current frequency 0, change of +1; resulting frequency 1.
- Current frequency 1, change of -2; resulting frequency -1.
- Current frequency -1, change of +3; resulting frequency

2.

- Current frequency 2, change of +1; resulting frequency 3.

In this example, the resulting frequency is 3.

Here are other example situations:

```
+1, +1, +1 results in  3
+1, +1, -2 results in  0
-1, -2, -3 results in -6
```

Starting with a frequency of zero, what is the resulting frequency after all of the changes in frequency have been applied?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["+1" "-2" "+3" "+1"])

(fn solve [input]
  (let [freq (lume.map input #(tonumber $))]
    (accumulate [sum 0 _ f (ipairs freq)]
      (+ sum f))))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(test 3 test-input)

(solve (aoc.string-from "2018/01.inp"))
```

```
574
```

DONE Day 1.2

You notice that the device repeats the same frequency change list over and over. To calibrate the device, you need to find the first frequency it reaches twice.

For example, using the same list of changes above, the device would loop as follows:

- Current frequency 0, change of +1; resulting frequency 1.
- Current frequency 1, change of -2; resulting frequency -1.
- Current frequency -1, change of +3; resulting frequency 2.
- Current frequency 2, change of +1; resulting frequency 3.
- (At this point, the device continues from the start of the list.)
- Current frequency 3, change of +1; resulting frequency 4.
- Current frequency 4, change of -2; resulting frequency 2, which has already been seen.

In this example, the first frequency reached twice is 2. Note that your device might need to repeat its list of frequency changes many times before a duplicate frequency is found, and that duplicates might be found while in the middle of processing the list.

Here are other examples:

- +1, -1 first reaches 0 twice.
- +3, +3, +4, -2, -4 first reaches 10 twice.
- -6, +3, +8, +5, -6 first reaches 5 twice.
- +7, +7, -2, -7, -4 first reaches 14 twice.

What is the first frequency your device reaches twice?

```
(fn solve2 [input]
  (let [xs (lume.map input #(tonumber $))]
    (var pos 2)
    (var res [(. xs 1)])
    (var xi (+ (. xs pos) (. res (length
 ↪  res))))
    (while (= nil (lume.find res xi))
      (table.insert res xi)
      (set pos (aoc.modulo+ 1 pos (length
 ↪  xs)))
```

```
      (set xi (+ (. xs pos) (. res (length
  ↪  res)))))
    xi))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 2 test-input)
(test2 10 ["+3" "+3" "+4" "-2" "-4"])
(test2 5 ["-6" "+3" "+8" "+5" "-6"])
(test2 14 ["+7" "+7" "-2" "-7" "-4"])

(solve2 (aoc.string-from "2018/01.inp"))
```

```
452
```

DONE Day 2.1

You stop falling through time, catch your breath, and check the screen on the device. "Destination reached. Current Year: 1518. Current Location: North Pole Utility Closet 83N10." You made it! Now, to find those anomalies.

Outside the utility closet, you hear footsteps and a voice. "…I'm not sure either. But now that so many people have chimneys,

maybe he could sneak in that way?" Another voice responds, "Actually, we've been working on a new kind of suit that would let him fit through tight spaces like that. But, I heard that a few days ago, they lost the prototype fabric, the design plans, everything! Nobody on the team can even seem to remember important details of the project!"

"Wouldn't they have had enough fabric to fill several boxes in the warehouse? They'd be stored together, so the box IDs should be similar. Too bad it would take forever to search the warehouse for two similar box IDs…" They walk too far away to hear any more.

Late at night, you sneak to the warehouse - who knows what kinds of paradoxes you could cause if you were discovered - and use your fancy wrist device to quickly scan every box and produce a list of the likely candidates (your puzzle input).

To make sure you didn't miss any, you scan the likely candidate boxes again, counting the number that have an ID containing exactly two of any letter and then separately counting those with exactly three of any letter. You can multiply those two counts together to get a rudimentary checksum and compare it to what your device predicts.

For example, if you see the following box IDs:

- abcdef contains no letters that appear exactly two or three times.
- bababc contains two a and three b, so it counts for both.
- abbcde contains two b, but no letter appears exactly three times.
- abcccd contains three c, but no letter appears exactly two times.
- aabcdd contains two a and two d, but it only counts once.
- abcdee contains two e.
- ababab contains three a and three b, but it only counts once.

Of these box IDs, four of them contain a letter which appears exactly twice, and three of them contain a letter which appears exactly three times. Multiplying these together produces a checksum of 4 * 3 = 12.

What is the checksum for your list of box IDs?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["abcdef"
```

```
5           "bababc"
6           "abbcde"
7           "abcccd"
8           "aabcdd"
9           "abcdee"
10          "ababab"])
11
12  (fn solve [input]
13    (var doubles 0)
14    (var triples 0)
15    (each [_ line (ipairs input)]
16      (let [xs (aoc.table-sort
    ↪  (aoc.string-toarray line))
17            bins (aoc.partition-by xs #(= $1
    ↪  $2))]
18        (when (< 0 (length (lume.filter bins #(=
    ↪  2 (length $)))))
19          (set doubles (+ 1 doubles)))
20        (when (< 0 (length (lume.filter bins #(=
    ↪  3 (length $)))))
21          (set triples (+ 1 triples)))))
22    (* doubles triples))
23
24  (fn test [expected input]
25    (assert (= expected (solve input))))
```

```
26
27 (test 12 test-input)
28
29 (solve (aoc.string-from "2018/02.inp"))
```

```
3952
```

DONE Day 2.2

Confident that your list of box IDs is complete, you're ready to find the boxes full of prototype fabric.

The boxes will have IDs which differ by exactly one character at the same position in both strings. For example, given the following box IDs:

```
abcde
fghij
klmno
pqrst
fguij
axcye
wvxyz
```

The IDs abcde and axcye are close, but they differ by two characters (the second and fourth). However, the IDs fghij and fguij

differ by exactly one character, the third (h and u). Those must be the correct boxes.

What letters are common between the two correct box IDs? (In the example above, this is found by removing the differing character from either ID, producing fgij.)

```
(local test2-input
       ["abcde"
        "fghij"
        "klmno"
        "pqrst"
        "fguij"
        "axcye"
        "wvxyz"])

(fn solve2 [input]
  (var done false)
  (for [i 1 (length input) &until done]
    (for [j i (length input) &until done]
      (when (= 1 (aoc.hamming-dist (. input i)
 ↪  (. input j)))
        (set done [(. input i) (. input
 ↪  j)])))))
  (aoc.table-tostring
   (lume.filter
```

```
    (aoc.string-toarray (. done 1))
    #(lume.find (aoc.string-toarray (. done
    2)) $))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 "fgij" test2-input)

(solve2 (aoc.string-from "2018/02.inp"))
```

```
vtnikorkulbfejvyznqgdxpaw
```

DONE Day 3.1

The Elves managed to locate the chimney-squeeze prototype fabric for Santa's suit (thanks to someone who helpfully wrote its box IDs on the wall of the warehouse in the middle of the night). Unfortunately, anomalies are still affecting them - nobody can even agree on how to cut the fabric.

The whole piece of fabric they're working on is a very large square - at least 1000 inches on each side.

Each Elf has made a claim about which area of fabric would be ideal for Santa's suit. All claims have an ID and consist of a

single rectangle with edges parallel to the edges of the fabric. Each claim's rectangle is defined as follows:

- The number of inches between the left edge of the fabric and the left edge of the rectangle.
- The number of inches between the top edge of the fabric and the top edge of the rectangle.
- The width of the rectangle in inches.
- The height of the rectangle in inches.

A claim like #123 @ 3,2: 5x4 means that claim ID 123 specifies a rectangle 3 inches from the left edge, 2 inches from the top edge, 5 inches wide, and 4 inches tall. Visually, it claims the square inches of fabric represented by # (and ignores the square inches of fabric represented by .) in the diagram below:

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

The problem is that many of the claims overlap, causing two or more claims to cover part of the same areas. For example, consider the following claims:

```
#1 @ 1,3: 4x4
#2 @ 3,1: 4x4
#3 @ 5,5: 2x2
```

Visually, these claim the following areas:

```
........
...2222.
...2222.
.11XX22.
.11XX22.
.111133.
.111133.
........
```

The four square inches marked with X are claimed by both 1 and 2. (Claim 3, while adjacent to the others, does not overlap either of them.)

If the Elves all proceed with their own plans, none of them will have enough fabric. How many square inches of fabric are within two or more claims?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["#1 @ 1,3: 4x4"
        "#2 @ 3,1: 4x4"
        "#3 @ 5,5: 2x2"])

(fn read-lines [lines]
  (let [res []]
    (each [_ line (ipairs lines)]
      (let [[_ _ xy wh] (aoc.string-split line
  ↪  " ")
            [x y] (aoc.string-split
  ↪  (string.sub xy 1 (- (string.len xy) 1))
  ↪  ",")
            [w h] (aoc.string-split wh "x")]
        (table.insert res [(tonumber x)
  ↪  (tonumber y) (+ w x -1) (+ h y -1)])))
    res))

(fn make-fabric [n]
  (let [rows []]
    (for [i 1 n]
      (let [cols []]
        (for [j 1 n]
```

```
          (table.insert cols 0))
        (table.insert rows cols)))
    rows))

(fn mark [fabric claim]
  (for [y (. claim 2) (. claim 4)]
    (for [x (. claim 1) (. claim 3)]
      (let [z (. (. fabric (+ 1 y)) (+ 1 x))]
        (aoc.table-replace fabric (+ 1 y) (+ 1
 ↪  x) (+ 1 z))))))

(fn mark-fabric [claims]
  (let [fabric (make-fabric 1000)]
    (each [_ claim (ipairs claims)]
      (mark fabric claim))
    fabric))

(fn solve [input]
  (let [claims (read-lines input)
        fabric (mark-fabric claims)]
    (length (lume.filter (aoc.table-flatten
 ↪  fabric) #(< 1 $)))))

(fn test [expected input]
  (assert (= expected (solve input))))
```

```
45
46 (test 4 test-input)
47
48 (solve (aoc.string-from "2018/03.inp"))
```

```
104241
```

DONE Day 3.2

Amidst the chaos, you notice that exactly one claim doesn't overlap by even a single square inch of fabric with any other claim. If you can somehow draw attention to it, maybe the Elves will be able to make Santa's suit after all!

For example, in the claims above, only claim 3 is intact after all claims are made.

What is the ID of the only claim that doesn't overlap?

```
(fn solve2 [input]
  (var done false)
  (let [claims (read-lines input)
        fabric (mark-fabric claims)]
    (each [i c (ipairs claims) &until done]
      (set done i)
      (for [y (. c 2) (. c 4)]
```

```
8           (for [x (. c 1) (. c 3)]
9             (when (not= 1 (. (. fabric (+ y 1))
    ↪  (+ 1 x)))
10              (set done false))))))
11    done)
12
13  (solve2 (aoc.string-from "2018/03.inp"))
```

```
806
```

DONE Day 4.1

You've sneaked into another supply closet - this time, it's across from the prototype suit manufacturing lab. You need to sneak inside and fix the issues with the suit, but there's a guard stationed outside the lab, so this is as close as you can safely get.

As you search the closet for anything that might help, you discover that you're not the first person to want to sneak in. Covering the walls, someone has spent an hour starting every midnight for the past few months secretly observing this guard post! They've been writing down the ID of the one guard on duty that night - the Elves seem to have decided that one guard was enough for the overnight shift - as well as when they fall

asleep or wake up while at their post (your puzzle input).

For example, consider the following records, which have already been organized into chronological order:

```
[1518-11-01 00:00] Guard #10 begins shift
[1518-11-01 00:05] falls asleep
[1518-11-01 00:25] wakes up
[1518-11-01 00:30] falls asleep
[1518-11-01 00:55] wakes up
[1518-11-01 23:58] Guard #99 begins shift
[1518-11-02 00:40] falls asleep
[1518-11-02 00:50] wakes up
[1518-11-03 00:05] Guard #10 begins shift
[1518-11-03 00:24] falls asleep
[1518-11-03 00:29] wakes up
[1518-11-04 00:02] Guard #99 begins shift
[1518-11-04 00:36] falls asleep
[1518-11-04 00:46] wakes up
[1518-11-05 00:03] Guard #99 begins shift
[1518-11-05 00:45] falls asleep
[1518-11-05 00:55] wakes up
```

Timestamps are written using year-month-day hour:minute format. The guard falling asleep or waking up is always the one whose shift most recently started. Because all asleep/awake

times are during the midnight hour (00:00 - 00:59), only the minute portion (00 - 59) is relevant for those events.

Visually, these records show that the guards are asleep at these times:

```
Date   ID   Minute

 ↪  000000000011111111112222222222333333333344444444445555

 ↪  012345678901234567890123456789012345678901234567890123
11-01  #10
 ↪  .....####################.....#########################
11-02  #99
 ↪  ........................................##########....
11-03  #10
 ↪  ........................#####.............................
11-04  #99
 ↪  ....................................##########........
11-05  #99
 ↪  .............................................#########
```

The columns are Date, which shows the month-day portion of the relevant day; ID, which shows the guard on duty that day; and Minute, which shows the minutes during which the guard was asleep within the midnight hour. (The Minute column's

header shows the minute's ten's digit in the first row and the one's digit in the second row.) Awake is shown as ., and asleep is shown as #.

Note that guards count as asleep on the minute they fall asleep, and they count as awake on the minute they wake up. For example, because Guard #10 wakes up at 00:25 on 1518-11-01, minute 25 is marked as awake.

If you can figure out the guard most likely to be asleep at a specific time, you might be able to trick that guard into working tonight so you can have the best chance of sneaking in. You have two strategies for choosing the best guard/minute combination.

Strategy 1: Find the guard that has the most minutes asleep. What minute does that guard spend asleep the most?

In the example above, Guard #10 spent the most minutes asleep, a total of 50 minutes (20+25+5), while Guard #99 only slept for a total of 30 minutes (10+10+10). Guard #10 was asleep most during minute 24 (on two days, whereas any other minute the guard was asleep was only seen on one day).

While this example listed the entries in chronological order, your entries are in the order you found them. You'll need to

organize them before they can be analyzed.

What is the ID of the guard you chose multiplied by the minute you chose? (In the above example, the answer would be 10 * 24 = 240.)

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["[1518-11-01 00:00] Guard #10 begins shift"
        "[1518-11-01 00:05] falls asleep"
        "[1518-11-01 00:25] wakes up"
        "[1518-11-01 00:30] falls asleep"
        "[1518-11-01 00:55] wakes up"
        "[1518-11-01 23:58] Guard #99 begins shift"
        "[1518-11-02 00:40] falls asleep"
        "[1518-11-02 00:50] wakes up"
        "[1518-11-03 00:05] Guard #10 begins shift"
        "[1518-11-03 00:24] falls asleep"
        "[1518-11-03 00:29] wakes up"
        "[1518-11-04 00:02] Guard #99 begins shift"
        "[1518-11-04 00:36] falls asleep"
```

```
17          "[1518-11-04 00:46] wakes up"
18          "[1518-11-05 00:03] Guard #99 begins
    ↪  shift"
19          "[1518-11-05 00:45] falls asleep"
20          "[1518-11-05 00:55] wakes up"])
21
22  (fn read-log [lines]
23    (let [res []]
24      (each [_ line (ipairs lines)]
25        (match (aoc.string-split line " ")
26          [d t "Guard" g "begins" "shift"]
27          (do (when (< 0 (# res)) (table.insert
    ↪  res -1))
28              (table.insert res (tonumber
    ↪  (string.sub g 2))))
29          [d t "falls" "asleep"]
30          (table.insert res (tonumber
    ↪  (string.match t ":(%d%d)%]")))
31          [d t "wakes" "up"]
32          (table.insert res (tonumber
    ↪  (string.match t ":(%d%d)%]")))))
33      (aoc.partition-at res -1)))
34
35  (fn process-log [logs]
36    (let [res {}]
```

```
    (each [_ log (ipairs logs)]
      (let [cur (or (. res (. log 1)) [])]
        (for [i 2 (length log) 2]
          (lume.map (aoc.range-to (. log i) (-
↪ (. log (+ 1 i)) 1))
                    #(table.insert cur $)))
        (tset res (. log 1) cur)))
    res))

(fn max-frequency [xs]
  (let [ids (aoc.keys xs)
        res []]
    (each [_ id (ipairs ids)]
      (let [ys (aoc.frequency (. xs id))]
        (table.insert res [id (length (. xs
↪ id)) (. ys 1) (length ys)])))
    (table.sort res #(< (. $1 2) (. $2 2)))
    (* (. (. res (# res)) 1)
       (. (. res (# res)) 3))))

(fn solve [input]
  (table.sort input)
  (-> input
      (read-log)
      (process-log)
```

```
      (max-frequency)))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 240 test-input)

(solve (aoc.string-from "2018/04.inp"))
```

```
84636
```

DONE Day 4.2

Strategy 2: Of all guards, which guard is most frequently asleep on the same minute?

In the example above, Guard #99 spent minute 45 asleep more than any other guard or minute - three times in total. (In all other cases, any guard spent any minute asleep at most twice.)

What is the ID of the guard you chose multiplied by the minute you chose? (In the above example, the answer would be 99 * 45 = 4455.)

```
(fn max-frequency2 [xs]
```

```
  (let [ids (aoc.keys xs)
        res []]
    (each [_ id (ipairs ids)]
      (let [ys (aoc.frequency (. xs id))]
        (table.insert res [id ys])))
    (table.sort res #(< (# (. $1 2)) (# (. $2
↪ 2))))
    (* (. (. res (# res)) 1)
       (. (. (. res (# res)) 2) 1))))

(fn solve2 [input]
  (table.sort input)
  (-> input
      (read-log)
      (process-log)
      (max-frequency2)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 4455 test-input)

(solve2 (aoc.string-from "2018/04.inp"))
```

```
91679
```

DONE Day 5.1

You've managed to sneak in to the prototype suit manufacturing lab. The Elves are making decent progress, but are still struggling with the suit's size reduction capabilities.

While the very latest in 1518 alchemical technology might have solved their problem eventually, you can do better. You scan the chemical composition of the suit's material and discover that it is formed by extremely long polymers (one of which is available as your puzzle input).

The polymer is formed by smaller units which, when triggered, react with each other such that two adjacent units of the same type and opposite polarity are destroyed. Units' types are represented by letters; units' polarity is represented by capitalization. For instance, r and R are units with the same type but opposite polarity, whereas r and s are entirely different types and do not react.

For example:

- In aA, a and A react, leaving nothing behind.
- In abBA, bB destroys itself, leaving aA. As above, this then destroys itself, leaving nothing.

- In abAB, no two adjacent units are of the same type, and so nothing happens.
- In aabAAB, even though aa and AA are of the same type, their polarities match, and so nothing happens.

Now, consider a larger example, dabAcCaCBAcCcaDA:

```
dabAcCaCBAcCcaDA  The first 'cC' is removed.
dabAaCBAcCcaDA    This creates 'Aa', which is
 ↪  removed.
dabCBAcCcaDA      Either 'cC' or 'Cc' are
 ↪  removed (the result is the same).
dabCBAcaDA        No further actions can be
 ↪  taken.
```

After all possible reactions, the resulting polymer contains 10 units.

How many units remain after fully reacting the polymer you scanned? (Note: in this puzzle and others, the input is large; if you copy/paste your input, make sure you get the whole thing.)

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["dabAcCaCBAcCcaDA"])

```

```
(fn react [xs j]
  (var done false)
  (if (= j (length xs)) xs
      (do (for [i j (# xs) &until done]
            (when (and (not= (. xs i) (. xs (- i 1)))
                       (or (= (string.upper (. xs i)) (. xs (- i 1)))
                           (= (string.lower (. xs i)) (. xs (- i 1)))))
              (table.remove xs i)
              (table.remove xs (- i 1))
              (set done i)))
          (react xs (if done (- done 1) (# xs))))))

(fn solve [input]
  (-> (. input 1)
      (aoc.string-toarray)
      (react 2)
      (length)))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(test 10 test-input)

(solve (aoc.string-from "2018/05.inp"))
```

```
11476
```

DONE Day 5.2

Time to improve the polymer.

One of the unit types is causing problems; it's preventing the polymer from collapsing as much as it should. Your goal is to figure out which unit type is causing the most problems, remove all instances of it (regardless of polarity), fully react the remaining polymer, and measure its length.

For example, again using the polymer dabAcCaCBAcCcaDA from above:

- Removing all A/a units produces dbcCCBcCcD. Fully reacting this polymer produces dbCBcD, which has length 6.
- Removing all B/b units produces daAcCaCAcCcaDA. Fully reacting this polymer produces daCAcaDA, which has length 8.

- Removing all C/c units produces dabAaBAaDA. Fully reacting this polymer produces daDA, which has length 4.
- Removing all D/d units produces abAcCaCBAcCcaA. Fully reacting this polymer produces abCBAc, which has length 6.

In this example, removing all C/c units was best, producing the answer 4.

What is the length of the shortest polymer you can produce by removing all units of exactly one type and fully reacting the result?

```
(fn solve2 [input]
  (let [line (. input 1)
        xs (aoc.string-toarray
 ↪  "abcdefghijklmnopqrstuvwxyz")]
    (aoc.math-min
     (icollect [_ x (ipairs xs)]
       (let [ys (string.gsub (string.gsub line
 ↪  x "") (string.upper x) "")]
         (length (react (aoc.string-toarray
 ↪  ys) 2)))))))

(fn test2 [expected input]
```

```
10    (assert (= expected (solve2 input))))
11
12  (test2 4 test-input)
13
14  (solve2 (aoc.string-from "2018/05.inp"))
```

```
5446
```

DONE Day 6.1

The device on your wrist beeps several times, and once again you feel like you're falling.

”Situation critical,” the device announces. ”Destination indeterminate. Chronal interference detected. Please specify new target coordinates.”

The device then produces a list of coordinates (your puzzle input). Are they places it thinks are safe or dangerous? It recommends you check manual page 729. The Elves did not give you a manual.

If they're dangerous, maybe you can minimize the danger by finding the coordinate that gives the largest distance from the other points.

Using only the Manhattan distance, determine the area around each coordinate by counting the number of integer X,Y locations that are closest to that coordinate (and aren't tied in distance to any other coordinate).

Your goal is to find the size of the largest area that isn't infinite. For example, consider the following list of coordinates:

```
1, 1
1, 6
8, 3
3, 4
5, 5
8, 9
```

If we name these coordinates A through F, we can draw them on a grid, putting 0,0 at the top left:

```
..........
.A........
..........
........C.
...D......
.....E....
.B........
..........
```

```
..........
........F.
```

This view is partial - the actual grid extends infinitely in all directions. Using the Manhattan distance, each location's closest coordinate can be determined, shown here in lowercase:

```
aaaaa.cccc
aAaaa.cccc
aaaddecccc
aadddeccCc
..dDdeeccc
bb.deEeecc
bBb.eeee..
bbb.eeefff
bbb.eefffF
bbb.ffffFf
```

Locations shown as `.` are equally far from two or more coordinates, and so they don't count as being closest to any.

In this example, the areas of coordinates A, B, C, and F are infinite - while not shown here, their areas extend forever outside the visible grid. However, the areas of coordinates D and E are finite: D is closest to 9 locations, and E is closest to 17 (both including the coordinate's location itself). Therefore, in this

example, the size of the largest area is 17.

What is the size of the largest area that isn't infinite?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["1, 1" "1, 6" "8, 3" "3, 4"
 ↪  "5, 5" "8, 9"])

(fn min-uniq-index [xs]
  (let [res (aoc.min-index xs)]
    (if (< 1 (lume.count xs #(= (. xs res)
 ↪  $))) 0 res)))

(fn populate [plane points]
  (for [y 1 (# plane)]
    (for [x 1 (# (. plane y))]
      (let [distances (icollect [i j (ipairs
 ↪  points)]
                        (aoc.manhattan-dist [x
 ↪  y] j))]
        (aoc.table-replace plane y x
 ↪  (min-uniq-index distances)))))
  plane)

(fn area [plane index]
```

```
18    (var res 0)
19    (let [yx (aoc.table-transpose plane)
20          x-edge (aoc.table-join (. plane 1) (.
      ↪  plane (# plane)))
21          y-edge (aoc.table-join (. yx 1) (. yx
      ↪  (# yx)))
22          edges (aoc.table-join x-edge y-edge)]
23      (when (not (lume.find edges index))
24        (for [y 1 (# plane)]
25          (for [x 1 (# (. plane y))]
26            (when (= index (. (. plane y) x))
27              (set res (+ 1 res)))))))
28    res)
29
30  (fn solve [input]
31    (let [points (lume.map input
      ↪  #(aoc.string-tonumarray $))
32          plane (populate (aoc.new-matrix 399
      ↪  399 0) points)]
33      (aoc.math-max (icollect [k v (ipairs
      ↪  points)] (area plane k)))))
34
35  (fn test [expected input]
36    (assert (= expected (solve input))))
37
```

```
(test 17 test-input)

(solve (aoc.string-from "2018/06.inp"))
```

```
3933
```

DONE Day 6.2

On the other hand, `if the coordinates are safe`, maybe the best you can do is try to find a `region` near as many coordinates as possible.

For example, suppose you want the sum of the Manhattan distance to all of the coordinates to be `less than 32`. For each location, add up the distances to all of the given coordinates; if the total of those distances is less than 32, that location is within the desired region. Using the same coordinates as above, the resulting region looks like this:

```
..........
.A........
..........
...###..C.
..#D###...
..###E#...
```

```
.B.###....
..........
..........
........F.
```

In particular, consider the highlighted location 4,3 located at the top middle of the region. Its calculation is as follows, where abs() is the absolute value function:

- Distance to coordinate A: abs(4-1) + abs(3-1) = 5
- Distance to coordinate B: abs(4-1) + abs(3-6) = 6
- Distance to coordinate C: abs(4-8) + abs(3-3) = 4
- Distance to coordinate D: abs(4-3) + abs(3-4) = 2
- Distance to coordinate E: abs(4-5) + abs(3-5) = 3
- Distance to coordinate F: abs(4-8) + abs(3-9) = 10
- Total distance: 5 + 6 + 4 + 2 + 3 + 10 = 30

Because the total distance to all coordinates (30) is less than 32, the location is `within` the region.

This region, which also includes coordinates D and E, has a total size of 16.

Your actual region will need to be much larger than this example, though, instead including all locations with a total distance of less than 10000.

What is the size of the region containing all locations which have a total distance to all given coordinates of less than 10000?

```
(fn populate2 [plane points]
  (for [y 1 (# plane)]
    (for [x 1 (# (. plane y))]
      (let [dist (icollect [i j (ipairs points)]
                   (aoc.manhattan-dist [x y] j))
            dist-sum (accumulate [s 0 _ dist (ipairs dist)]
                       (+ s dist))]
        (aoc.table-replace plane y x dist-sum))))
  plane)

(fn area2 [plane max-dist]
  (var res 0)
  (for [y 1 (# plane)]
    (for [x 1 (# (. plane y))]
      (when (< (. (. plane y) x) max-dist)
        (set res (+ 1 res)))))
  res)

(fn solve2 [input max]
```

```
20    (let [points (lume.map input
      ↪  #(aoc.string-tonumarray $))
21          plane (populate2 (aoc.new-matrix 399
      ↪  399 0) points)]
22      (area2 plane max)))
23
24  (fn test2 [expected input max]
25    (assert (= expected (solve2 input max))))
26
27  (test2 16 test-input 32)
28
29  (solve2 (aoc.string-from "2018/06.inp") 10000)
```

```
41145
```

DONE Day 7.1

You find yourself standing on a snow-covered coastline; apparently, you landed a little off course. The region is too hilly to see the North Pole from here, but you do spot some Elves that seem to be trying to unpack something that washed ashore. It's quite cold out, so you decide to risk creating a paradox by asking them for directions.

”Oh, are you the search party?” Somehow, you can understand

whatever Elves from the year 1018 speak; you assume it's Ancient Nordic Elvish. Could the device on your wrist also be a translator? ”Those clothes don't look very warm; take this.” They hand you a heavy coat.

”We do need to find our way back to the North Pole, but we have higher priorities at the moment. You see, believe it or not, this box contains something that will solve all of Santa's transportation problems - at least, that's what it looks like from the pictures in the instructions.” It doesn't seem like they can read whatever language it's in, but you can: ”Sleigh kit. Some assembly required.”

”'Sleigh'? What a wonderful name! You must help us assemble this 'sleigh' at once!” They start excitedly pulling more parts out of the box.

The instructions specify a series of steps and requirements about which steps must be finished before others can begin (your puzzle input). Each step is designated by a single letter. For example, suppose you have the following instructions:

```
Step C must be finished before step A can
 ↪  begin.
Step C must be finished before step F can
 ↪  begin.
```

```
Step A must be finished before step B can
 ↪  begin.
Step A must be finished before step D can
 ↪  begin.
Step B must be finished before step E can
 ↪  begin.
Step D must be finished before step E can
 ↪  begin.
Step F must be finished before step E can
 ↪  begin.
```

Visually, these requirements look like this:

```
  -->A--->B--
 /    \      \
C      -->D----->E
 \           /
  ---->F-----
```

Your first goal is to determine the order in which the steps should be completed. If more than one step is ready, choose the step which is first alphabetically. In this example, the steps would be completed as follows:

- Only C is available, and so it is done first.
- Next, both A and F are available. A is first alphabetically, so it is done next.

- Then, even though F was available earlier, steps B and D are now also available, and B is the first alphabetically of the three.
- After that, only D and F are available. E is not available because only some of its prerequisites are complete. Therefore, D is completed next.
- F is the only choice, so it is done next.
- Finally, E is completed.

So, in this example, the correct order is CABDFE.

In what order should the steps in your instructions be completed?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["Step C must be finished before step A can begin."
        "Step C must be finished before step F can begin."
        "Step A must be finished before step B can begin."
        "Step A must be finished before step D can begin."
```

```
        "Step B must be finished before step E
    ↪ can begin."
        "Step D must be finished before step E
    ↪ can begin."
        "Step F must be finished before step E
    ↪ can begin."])

(fn read-lines [lines]
  (let [res []]
    (each [_ line (ipairs lines)]
      (let [[_ f _ _ _ _ _ t _ _]
    ↪ (aoc.string-split line " ")]
        (table.insert res [f t])))
    res))

(fn topo-sort [edges]
  (let [dag (aoc.adjacency-list edges)
        len (# (aoc.keys dag))
        ind {}
        queue []
        res []]
    (each [_ [f t] (ipairs edges)]
      (if (. ind f) nil (tset ind f 0))
      (if (. ind t)
          (tset ind t (+ 1 (. ind t)))
```

```
29            (tset ind t 1)))
30      (lume.each (aoc.keys (lume.filter ind #(=
    ↪  0 $) true))
31                 (fn [e] (aoc.qpush queue e)))
32      (while (aoc.not-empty? queue)
33        (table.sort queue)
34        (let [w (aoc.qpop queue)
35              adj (. dag w)]
36          (table.insert res w)
37          (each [_ v (ipairs adj)]
38            (let [decr (- (. ind v) 1)]
39              (when (= 0 decr) (aoc.qpush queue
    ↪  v))
40              (tset ind v decr)))))
41      (if (not= (# res) len) nil res)))
42
43  (fn solve [input]
44    (-> input
45        (read-lines)
46        (topo-sort)
47        (aoc.table-tostring)))
48
49  (fn test [expected input]
50    (assert (= expected (solve input))))
51
```

```
(test "CABDFE" test-input)

(solve (aoc.string-from "2018/07.inp"))
```

```
ACHOQRXSEKUGMYIWDZLNBFTJVP
```

2017 [10/50]

DONE Day 1.1

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

When your eyes can focus again, everything seems a lot more pixelated than before. She must have sent you inside the computer! You check the system clock: 25 milliseconds until mid-

night. With that much time, you should be able to collect all fifty stars by December 25th.

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

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

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

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

For example:

- 1122 produces a sum of 3 (1 + 2) because the first digit (1)

matches the second digit and the third digit (2) matches the fourth digit.
- 1111 produces 4 because each digit (all 1) matches the next.
- 1234 produces 0 because no digit matches the next.
- 91212129 produces 9 because the only digit that matches the next one is the last digit, 9.

What is the solution to your captcha?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [input]
  (-> (. input 1)
      (.. (string.sub (. input 1) 1 1))
      (aoc.string-toarray)
      (lume.map (fn [e] (tonumber e)))
      (aoc.partition1)
      (lume.map (fn [[e1 e2]] (if (= e1 e2) e1 0)))
      (aoc.table-sum)))

(fn test [expected input]
  (assert (= expected (solve [input]))))

```

```
(test 3 "1122")
(test 4 "1111")
(test 0 "1234")
(test 9 "91212129")

(solve (aoc.string-from "2017/01.inp"))
```

```
995
```

DONE Day 1.2

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

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

For example:

- 1212 produces 6: the list contains 4 items, and all four digits match the digit 2 items ahead.

- 1221 produces 0, because every comparison is between a 1 and a 2.
- 123425 produces 4, because both 2s match each other, but no other digit has a match.
- 123123 produces 12.
- 12131415 produces 4.

What is the solution to your new captcha?

```
(fn solve2 [input]
  (let [xs (lume.map (aoc.string-toarray (. input 1)) #(tonumber $))]
    (aoc.table-sum
     (icollect [i v (ipairs xs)]
       (if (= v (. xs (aoc.modulo+ i (/ (# xs) 2) (# xs)))) v 0)))))

(fn test2 [expected input]
  (assert (= expected (solve2 [input]))))

(test2 6 "1212")
(test2 0 "1221")
(test2 4 "123425")
(test2 12 "123123")
(test2 4 "12131415")

```

```
16 (solve2 (aoc.string-from "2017/01.inp"))
```

```
1130
```

DONE Day 2.1

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

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

For example, given the following spreadsheet:

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

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["5\t1\t9\t5"
        "7\t5\t3"
        "2\t4\t6\t8"])

(fn solve [input]
  (let [xs (lume.map input
  ↪  #(aoc.string-tonumarray $))
        ys (icollect [k v (ipairs xs)]
             (- (aoc.max v) (aoc.min v)))]
    (accumulate [sum 0 k v (ipairs ys)]
      (+ sum v))))
```

```
14
15 (fn test [expected input]
16   (assert (= expected (solve input))))
17
18 (test 18 test-input)
19
20 (solve (aoc.string-from "2017/02.inp"))
```

```
42299
```

DONE Day 2.2

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

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

It sounds like the goal is to find the only two numbers in each row where one evenly divides the other - that is, where the result of the division operation is a whole number. They would

like you to find those numbers on each line, divide them, and add up each line's result.

For example, given the following spreadsheet:

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

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

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

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

```
(local test2-input
       ["5\t9\t2\t8"
        "9\t4\t7\t3"
        "3\t8\t6\t5"])

(fn even-division [xs]
  (let [res []]
```

```
8       (for [i 1 (- (# xs) 1)]
9         (for [j (+ i 1) (# xs)]
10          (table.insert res
11                        (if (= 0 (% (. xs i) (.
    ↪ xs j)))
12                            (/ (. xs i) (. xs
    ↪ j))
13                            (= 0 (% (. xs j) (.
    ↪ xs i)))
14                            (/ (. xs j) (. xs
    ↪ i))
15                            0))))
16      res))
17
18  (fn solve2 [input]
19    (-> input
20        (lume.map #(aoc.string-tonumarray $))
21        (lume.map #(even-division $))
22        (aoc.table-sum)))
23
24  (fn test2 [expected input]
25    (assert (= expected (solve2 input))))
26
27  (test2 9 test2-input)
28
```

```
(solve2 (aoc.string-from "2017/02.inp"))
```

```
277
```

DONE Day 4.1

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

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

For example:

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["aa bb cc dd ee"
        "aa bb cc dd aa"
        "aa bb cc dd aaa"])

(fn solve [input]
  (var res 0)
  (let [lines (lume.map input
   ↪  #(aoc.string-split $ " "))]
    (each [_ line (ipairs lines)]
      (when (aoc.table-no-dups? line)
        (set res (+ 1 res)))))
  res)

(fn test [expected input]
  (assert (= expected (solve input))))

(test 2 test-input)

(solve (aoc.string-from "2017/04.inp"))
```

```
455
```

DONE Day 4.2

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

For example:

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

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

```
(local test2-input
       ["abcde fghij"
        "abcde xyz ecdab"
```

```
        "a ab abc abd abf abj"
        "iiii oiii ooii oooi oooo"
        "oiii ioii iioi iiio"])

(fn solve2 [input]
  (var res 0)
  (let [lines (lume.map input (fn [line]
    (aoc.string-split line " ")))
        lines2 (lume.map lines
                         (fn [line] (lume.map
  line
  (fn [word]
  (let [t (aoc.string-toarray word)]
  (table.sort t)
  (table.concat t "")))))))]
    (each [_ line (ipairs lines2)]
      (when (aoc.table-no-dups? line)
        (set res (+ 1 res)))))
  res)

(fn test2 [expected input]
```

```
23    (assert (= expected (solve2 input))))
24
25  (test2 3 test2-input)
26
27  (solve2 (aoc.string-from "2017/04.inp"))
```

```
186
```

DONE Day 5.1

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

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

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

For example, consider the following list of jump offsets:

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

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

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

In this example, the exit is reached in 5 steps. How many steps does it take to reach the exit?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["0"
        "3"
        "0"
        "1"
        "-3"])

(fn table-inc [t p]
  (let [old (. t (+ 1 p))]
    (aoc.table-swap t (+ 1 p) (+ 1 old))
    old))

(fn solve [input]
  (let [xs (lume.map input #(tonumber $))
        len (# xs)]
    (var p 0)
    (var c 0)
    (while (<= p (- len 1))
      (set p (+ p (table-inc xs p)))
      (set c (+ 1 c)))
    c))

(fn test [expected input]
```

```
  (assert (= expected (solve input))))

(test 5 test-input)

(solve (aoc.string-from "2017/05.inp"))
```

```
326618
```

DONE Day 5.2

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

Using this rule with the above example, the process now takes 10 steps, and the offset values after finding the exit are left as 2 3 2 3 -1. How many steps does it now take to reach the exit?

```
(fn table-inc2 [t p]
  (let [old (. t (+ 1 p))]
    (if (<= 3 old)
        (aoc.table-swap t (+ 1 p) (- old 1))
        (aoc.table-swap t (+ 1 p) (+ old 1)))
    old))

(fn solve2 [input]
```

```
  (let [xs (lume.map input #(tonumber $))
        len (# xs)]
    (var p 0)
    (var c 0)
    (while (<= p (- len 1))
      (set p (+ p (table-inc2 xs p)))
      (set c (+ 1 c)))
    c))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 10 test-input)

(solve2 (aoc.string-from "2017/05.inp"))
```

```
21841249
```

DONE Day 6.1

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

In this area, there are sixteen memory banks; each memory

bank can hold any number of blocks. The goal of the reallocation routine is to balance the blocks between the memory banks.

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

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

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

- The banks start with 0, 2, 7, and 0 blocks. The third bank has the most blocks, so it is chosen for redistribution.
- Starting with the next bank (the fourth bank) and then continuing to the first bank, the second bank, and so on, the 7 blocks are spread out over the memory banks. The

fourth, first, and second banks get two blocks each, and the third bank gets one back. The final result looks like this: 2 4 1 2.

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

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

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

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["0\t2\t7\t0"])

(fn table-inc [xs i]
  (var v (. xs i))
  (var j (if (< i (# xs)) (+ 1 i) 1))
  (aoc.table-swap xs i 0)
  (while (< 0 v)
    (aoc.table-swap xs j (+ 1 (. xs j)))
    (set v (- v 1))
    (set j (if (< j (# xs)) (+ 1 j) 1)))
  xs)

(fn find-cycle [input]
  (let [xs (aoc.string-tonumarray (. input 1))
        res [(aoc.table-clone xs)]]
    (table-inc xs (aoc.max-index xs))
    (while (not (aoc.matrix-contains? res xs))
      (table.insert res (aoc.table-clone xs))
      (table-inc xs (aoc.max-index xs)))
    res))

(fn solve [input]
  (length (find-cycle input)))
```

```
26
27 (fn test [expected input]
28   (assert (= expected (solve input))))
29
30 (test 5 test-input)
31
32 (solve (aoc.string-from "2017/06.inp"))
```

```
4074
```

DONE Day 6.2

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

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

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

```
1 (fn solve2 [input]
2   (let [res (find-cycle input)
3         xs0 (. res (# res))
```

```
        xs (table-inc xs0 (aoc.max-index xs0))
        (_ cycle-start) (aoc.matrix-contains?
↪  res xs)]
    (- (+ 1 (# res)) cycle-start)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 4 test-input)

(solve2 (aoc.string-from "2017/06.inp"))
```

```
2793
```

2016 [12/50]

DONE Day 1.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [input]
  (var x 0)
  (var y 0)
  (let [steps (aoc.string-split (. input 1) ",
  ↪  ")]
    (each [_ step (ipairs steps)]
      (let [D (string.sub step 1 1)
            d (string.sub step 2)]
        (if (= "R" D)
            (let [y1 (- y)]
              (set y x)
              (set x y1))
            (let [x1 (- x)]
              (set x y)
              (set y x1)))
        (set x (+ (tonumber d) x)))))
  (aoc.manhattan-dist [x y] [0 0]))
```

```
(fn test [expected input]
  (assert (= expected (solve [input]))))

(test 5 "R2, L3")
(test 2 "R2, R2, R2")
(test 12 "R5, L5, R5, R3")

(solve (aoc.string-from "2016/01.inp"))
```

```
161
```

DONE Day 1.2

Then, you notice the instructions continue on the back of the Recruiting Document. Easter Bunny HQ is actually at the first location you visit twice.

For example, if your instructions are R8, R4, R4, R8, the first location you visit twice is 4 blocks away, due East.

How many blocks away is the first location you visit twice?

```
(fn solve2 [input]
  (var dir (/ math.pi 2))
  (var x 0)
```

```
  (var y 0)
  (var done false)
  (let [steps (aoc.string-split (. input 1) ",
↪  ")
        res [[0 0]]]
    (each [_ step (ipairs steps) &until done]
      (let [D (string.sub step 1 1)
            d (string.sub step 2)]
        (if (= "R" D)
            (set dir (- dir (/ math.pi 2)))
            (set dir (+ dir (/ math.pi 2))))
        (set x (+ x (* d (lume.round (math.cos
↪  dir)))))
        (set y (+ y (* d (lume.round (math.sin
↪  dir))))))
      (let [[fx fy] (table.remove res)
            [tx ty] [x y]]
        (if (= fx tx)
            (for [i fy ty (if (< ty fy) -1 1)
↪  &until done]
              (if (aoc.matrix-contains? res [x
↪  i])
                  (do
                    (set done true)
                    (set y i))
```

```
24                    (table.insert res [x i])))
25              (for [i fx tx (if (< tx fx) -1 1)
    ↪  &until done]
26                (if (aoc.matrix-contains? res [i
    ↪  y])
27                    (do
28                      (set done true)
29                      (set x i))
30                    (table.insert res [i
    ↪  y])))))))
31      (+ (math.abs x) (math.abs y))))
32
33  (fn test2 [expected input]
34    (assert (= expected (solve2 [input]))))
35
36  (test2 4 "R8, R4, R4, R8")
37
38  (solve2 (aoc.string-from "2016/01.inp"))
```

```
110
```

DONE Day 2.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

```

```
(local test-input
       ["ULL"
        "RRDDD"
        "LURDL"
        "UUUUD"])

(fn solve [input]
  (let [pos {:x 2 :y 2}
        xs (lume.map input
 ↪  #(aoc.string-toarray $))
        keypad [[1 2 3]
                [4 5 6]
                [7 8 9]]
        res []]
    (each [_ ys (ipairs xs)]
      (each [_ x (ipairs ys)]
        (case x
          "U" (tset pos :y (math.max 1 (- (.
 ↪  pos :y) 1)))
          "D" (tset pos :y (math.min 3 (+ 1 (.
 ↪  pos :y))))
          "R" (tset pos :x (math.min 3 (+ 1 (.
 ↪  pos :x))))
          "L" (tset pos :x (math.max 1 (- (.
 ↪  pos :x) 1)))))))
```

```
      (table.insert res (. (. keypad (. pos
 ↪  :y)) (. pos :x))))
    (table.concat res "")))

(fn test [expected input]
  (assert (= expected (solve input))))

(test "1985" test-input)

(solve (aoc.string-from "2016/02.inp"))
```

```
35749
```

DONE Day 2.2

You finally arrive at the bathroom (it's a several minute walk from the lobby so visitors can behold the many fancy conference rooms and water coolers on this floor) and go to punch in the code. Much to your bladder's dismay, the keypad is not at all like you imagined it. Instead, you are confronted with the result of hundreds of man-hours of bathroom-keypad-design meetings:

```
    1
  2 3 4
```

```
5 6 7 8 9
  A B C
    D
```

You still start at ”5” and stop when you're at an edge, but given the same instructions as above, the outcome is very different:

- You start at ”5” and don't move at all (up and left are both edges), ending at 5.
- Continuing from ”5”, you move right twice and down three times (through ”6”, ”7”, ”B”, ”D”, ”D”), ending at D.
- Then, from ”D”, you move five more times (through ”D”, ”B”, ”C”, ”C”, ”B”), ending at B.
- Finally, after five more moves, you end at 3.

So, given the actual keypad layout, the code would be 5DB3. Using the same instructions in your puzzle input, what is the correct bathroom code?

```
(fn solve2 [input]
  (let [pos {:x 1 :y 3}
        xs (lume.map input
 ↪  #(aoc.string-toarray $))
        keypad [["0" "0" "1" "0" "0"]
                ["0" "2" "3" "4" "0"]
```

```
                ["5" "6" "7" "8" "9"]
                ["0" "A" "B" "C" "0"]
                ["0" "0" "D" "0" "0"]]
        res []]
    (each [_ ys (ipairs xs)]
      (each [_ x (ipairs ys)]
        (case x
          "U" (let [newy (math.max 1 (- (. pos
  ↪ :y) 1))]
                (when (not= 0 (tonumber (. (.
  ↪ keypad newy) (. pos :x))))
                  (tset pos :y newy)))
          "D" (let [newy (math.min 5 (+ 1 (.
  ↪ pos :y)))]
                (when (not= 0 (tonumber (. (.
  ↪ keypad newy) (. pos :x))))
                  (tset pos :y newy)))
          "R" (let [newx (math.min 5 (+ 1 (.
  ↪ pos :x)))]
                (when (not= 0 (tonumber (. (.
  ↪ keypad (. pos :y)) newx)))
                  (tset pos :x newx)))
          "L" (let [newx (math.max 1 (- (. pos
  ↪ :x) 1))]
                (when (not= 0 (tonumber (. (.
  ↪ keypad (. pos :y)) newx)))
```

```
                    (tset pos :x newx)))))
      (table.insert res (. (. keypad (. pos
 ↪  :y)) (. pos :x))))
    (table.concat res "")))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 "5DB3" test-input)

(solve2 (aoc.string-from "2016/02.inp"))
```

```
9365C
```

DONE Day 3.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["  5  10  25"])

(fn triangle? [a b c]
  (and
   (< a (+ b c))
   (< b (+ a c))
   (< c (+ a b))))

(fn count-triangles [xs]
  (var res 0)
  (each [_ [a b c] (ipairs xs)]
    (when (triangle? a b c)
      (set res (+ 1 res))))
  res)
```

```
17
18 (fn solve [input]
19   (-> input
20       (lume.map #(aoc.string-tonumarray $))
21       (count-triangles)))
22
23 (fn test [expected input]
24   (assert (= expected (solve input))))
25
26 (test 0 test-input)
27
28 (solve (aoc.string-from "2016/03.inp"))
```

```
1050
```

DONE Day 3.2

Now that you've helpfully marked up their design documents, it occurs to you that triangles are specified in groups of three vertically. Each set of three numbers in a column specifies a triangle. Rows are unrelated.

For example, given the following specification, numbers with the same hundreds digit would be part of the same triangle:

```
101 301 501
102 302 502
103 303 503
201 401 601
202 402 602
203 403 603
```

In your puzzle input, and instead reading by columns, how many of the listed triangles are possible?

```
(fn solve2 [input]
  (-> input
      (lume.map #(aoc.string-tonumarray $))
      (aoc.table-transpose)
      (lume.map #(aoc.partition3 $))
      (aoc.table-flatten)
      (count-triangles)))

(solve2 (aoc.string-from "2016/03.inp"))
```

```
1921
```

DONE Day 4.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
```

```
(local test-input
       ["aaaaa-bbb-z-y-x-123[abxyz]"
        "a-b-c-d-e-f-g-h-987[abcde]"
        "not-a-real-room-404[oarel]"
        "totally-real-room-200[decoy]"])

(fn parse-rooms [s]
  (let [len (string.len s)]
    [(string.sub s 1 (- len 11))
     (tonumber (string.sub s (- len 9) (- len 7)))
     (string.sub s (- len 5) (- len 1))]))

(fn checksum [room]
  (-> room
      (string.gsub "%-" "")
      (aoc.string-toarray)
      (aoc.table-sort)
      (aoc.partition-by #(= $1 $2))
      (aoc.table-sort (fn [x y] (if (= (# x) (# y))
                                    (< (. x 1) (. y 1))
                                    (>= (# x) (# y)))))))
```

```
      (lume.map #(. $ 1))
      (table.concat "")
      (string.sub 1 5)))

(fn solve [input]
  (-> input
      (lume.map #(parse-rooms $))
      (lume.filter #(= (checksum (. $ 1)) (. $
 ↪  3)))
      (lume.map #(. $ 2))
      (lume.reduce #(+ $1 $2))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 1514 test-input)

(solve (aoc.string-from "2016/04.inp"))
```

```
361724
```

DONE Day 4.2

With all the decoy data out of the way, it's time to decrypt this list and get moving.

The room names are encrypted by a state-of-the-art shift cipher, which is nearly unbreakable without the right software. However, the information kiosk designers at Easter Bunny HQ were not expecting to deal with a master cryptographer like yourself.

To decrypt a room name, rotate each letter forward through the alphabet a number of times equal to the room's sector ID. A becomes B, B becomes C, Z becomes A, and so on. Dashes become spaces.

For example, the real name for `qzmt-zixmtkozy-ivhz-343` is `very encrypted name`.

What is the sector ID of the room where North Pole objects are stored?

```
(local test2-input
  ↪  ["qzmt-zixmtkozy-ivhz-343[zimth]"])

(fn ceasar-decrypt [[room id sum]]
  (let [xs (aoc.string-split room "-")]
    (lume.map xs #(aoc.string-shift $ id))))

(fn solve2 [input target]
  (-> input
```

```
      (lume.map #(parse-rooms $))
      (lume.filter #(= (checksum (. $ 1)) (. $
↪  3)))
      (lume.map (fn [e] [(table.concat
↪  (ceasar-decrypt e) " ")
                         (. e 2)]))
      (lume.filter (fn [e] (= target (. e
↪  1))))
      (aoc.first)
      (. 2)))

(fn test2 [expected input target]
  (assert (= expected (solve2 input target))))

(test2 343 test2-input "very encrypted name")

(solve2 (aoc.string-from "2016/04.inp")
↪  "northpole object storage")
```

```
482
```

DONE Day 5.1

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?

```
(local md5 (require :lib.md5))
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn solve [input]
  (var salt 0)
  (let [res []]
    (while (< (# res) 8)
      (let [hash (md5.sumhexa (.. input salt))
            prefix (string.sub hash 1 5)]
        (when (= "00000" prefix)
          (table.insert res (string.sub hash 6
 ↪  6)))
        (set salt (+ 1 salt))))
    (table.concat res "")))

(solve "reyedfim")
```

```
f97c354d
```

DONE Day 5.2

As the door slides open, you are presented with a second door that uses a slightly more inspired security mechanism. Clearly

unimpressed by the last version (in what movie is the password decrypted in order?!), the Easter Bunny engineers have worked out a better solution.

Instead of simply filling in the password from left to right, the hash now also indicates the position within the password to fill. You still look for hashes that begin with five zeroes; however, now, the sixth character represents the position (0-7), and the seventh character is the character to put in that position.

A hash result of 000001f means that f is the second character in the password. Use only the first result for each position, and ignore invalid positions.

For example, if the Door ID is abc:

- The first interesting hash is from abc3231929, which produces 0000015…; so, 5 goes in position 1: 5_____.
- In the previous method, 5017308 produced an interesting hash; however, it is ignored, because it specifies an invalid position (8).
- The second interesting hash is at index 5357525, which produces 000004e…; so, e goes in position 4: 5_e__.

You almost choke on your popcorn as the final character falls into place, producing the password 05ace8e3.

Given the actual Door ID and this new method, what is the password?

```
(fn solve2 [input]
  (var salt 0)
  (let [res ["" "" "" "" "" "" "" ""]]
    (while (lume.find res "")
      (let [hash (md5.sumhexa (.. input salt))
            prefix (string.sub hash 1 5)]
        (when (= "00000" prefix)
          (let [pos (tonumber (string.sub hash
 ↪  6 6))]
            (when (and pos (<= 0 pos 7) (= ""
 ↪  (. res (+ 1 pos))))
              (aoc.table-swap res (+ 1 pos)
 ↪  (string.sub hash 7 7)))))
        (set salt (+ 1 salt))))
    (table.concat res "")))

(solve2 "reyedfim")
```

```
863dde27
```

DONE Day 6.1

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?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local test-input
       ["eedadn"
        "drvtee"
        "eandsr"
        "raavrd"
        "atevrs"
        "tsrnev"
        "sdttsa"
        "rasrtv"
```

```
13          "nssdts"
14          "ntnada"
15          "svetve"
16          "tesnvt"
17          "vntsnd"
18          "vrdear"
19          "dvrsen"
20          "enarar"])
21
22  (fn solve [input]
23    (-> input
24        (aoc.read-matrix)
25        (aoc.table-transpose)
26        (lume.map aoc.table-sort)
27        (lume.map #(aoc.partition-by $ (fn [e1
    ↪  e2] (= e1 e2))))
28        (lume.map #(aoc.table-sort $ (fn [e1 e2]
    ↪  (> (length e1) (length e2)))))
29        (lume.map #(. (. $ 1) 1))
30        (lume.reduce #(.. $1 $2))))
31
32  (fn test [expected input]
33    (assert (= expected (solve input))))
34
35  (test "easter" test-input)
```

```
36
37 (solve (aoc.string-from "2016/06.inp"))
```

```
qtbjqiuq
```

DONE Day 6.2

Of course, that would be the message - if you hadn't agreed to use a modified repetition code instead.

In this modified code, the sender instead transmits what looks like random data, but for each character, the character they actually want to send is slightly less likely than the others. Even after signal-jamming noise, you can look at the letter distributions in each column and choose the least common letter to reconstruct the original message.

In the above example, the least common character in the first column is a; in the second, d, and so on. Repeating this process for the remaining characters produces the original message, advent.

Given the recording in your puzzle input and this new decoding methodology, what is the original message that Santa is trying to send?

```
(fn solve2 [input]
  (-> input
      (aoc.read-matrix)
      (aoc.table-transpose)
      (lume.map aoc.table-sort)
      (lume.map #(aoc.partition-by $ (fn [e1 e2] (= e1 e2))))
      (lume.map #(aoc.table-sort $ (fn [e1 e2] (< (length e1) (length e2)))))
      (lume.map #(. (. $ 1) 1))
      (lume.reduce #(.. $1 $2))))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 "advent" test-input)

(solve2 (aoc.string-from "2016/06.inp"))
```

akothqli

2015 [44/50]

DONE Day 1.1

Santa was hoping for a white Christmas, but his weather machine's ”snow” function is powered by stars, and he's fresh out! To save Christmas, he needs you to collect fifty stars by December 25th.

Collect stars by helping Santa solve 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!

Here's an easy puzzle to warm you up.

Santa is trying to deliver presents in a large apartment building, but he can't find the right floor - the directions he got are a little confusing. He starts on the ground floor (floor 0) and then follows the instructions one character at a time.

An opening parenthesis, (, means he should go up one floor, and a closing parenthesis,), means he should go down one floor.

The apartment building is very tall, and the basement is very

deep; he will never find the top or bottom floors.

For example:

- (()) and ()() both result in floor 0.
- (((and (()(()(both result in floor 3.
-))(((((also results in floor 3.
- ()) and))(both result in floor -1 (the first basement level).
-))) and)())()) both result in floor -3.

To what floor do the instructions take Santa?

```
(local aoc (require :lib.aoc))

(fn solve [input]
  (var res 0)
  (let [xs (aoc.string-toarray (. input 1))]
    (each [_ x (ipairs xs)]
      (if (= ")" x)
          (set res (- res 1))
          (set res (+ 1 res)))))
  res)

(solve (aoc.string-from "2015/01.inp"))
```

```
74
```

DONE Day 1.2

Now, given the same instructions, find the position of the first character that causes him to enter the basement (floor -1). The first character in the instructions has position 1, the second character has position 2, and so on.

For example:

-) causes him to enter the basement at character position 1.
- ()()) causes him to enter the basement at character position 5.

What is the position of the character that causes Santa to first enter the basement?

```
(fn solve2 [input]
  (var res 0)
  (var done false)
  (let [xs (aoc.string-toarray (. input 1))]
    (each [i x (ipairs xs) &until done]
      (set res (+ res (if (= ")" x) -1 1)))
      (when (< res 0) (set done i))))
  done)

```

```
10 (solve2 (aoc.string-from "2015/01.inp"))
```

```
1795
```

DONE Day 2.1

The elves are running low on wrapping paper, and so they need to submit an order for more. They have a list of the dimensions (length l, width w, and height h) of each present, and only want to order exactly as much as they need.

Fortunately, every present is a box (a perfect right rectangular prism), which makes calculating the required wrapping paper for each gift a little easier: find the surface area of the box, which is 2*l*w + 2*w*h + 2*h*l. The elves also need a little extra paper for each present: the area of the smallest side.

For example:

- A present with dimensions 2x3x4 requires 2*6 + 2*12 + 2*8 = 52 square feet of wrapping paper plus 6 square feet of slack, for a total of 58 square feet.
- A present with dimensions 1x1x10 requires 2*1 + 2*10 + 2*10 = 42 square feet of wrapping paper plus 1 square foot of slack, for a total of 43 square feet.

All numbers in the elves' list are in feet. How many total square feet of wrapping paper should they order?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local test-input
       ["2x3x4"
        "1x1x10"])

(fn area [w h l]
  (let [a1 (* l w) a2 (* w h) a3 (* h l)]
    (+ (* 2 a1) (* 2 a2) (* 2 a3) (math.min a1 a2 a3))))

(fn solve [input]
  (var res 0)
  (each [_ dim (ipairs input)]
    (case (aoc.string-split dim "x")
      [x y z] (set res (+ res (area x y z)))
      _ (print (.. "No match found for " dim))))
  res)

(fn test [expected input]
  (assert (= expected (solve input))))
```

```
22
23 (test 101 test-input)
24
25 (solve (aoc.string-from "2015/02.inp"))
```

```
1588178
```

DONE Day 2.2

The elves are also running low on ribbon. Ribbon is all the same width, so they only have to worry about the length they need to order, which they would again like to be exact.

The ribbon required to wrap a present is the shortest distance around its sides, or the smallest perimeter of any one face. Each present also requires a bow made out of ribbon as well; the feet of ribbon required for the perfect bow is equal to the cubic feet of volume of the present. Don't ask how they tie the bow, though; they'll never tell.

For example:

- A present with dimensions 2x3x4 requires 2+2+3+3 = 10 feet of ribbon to wrap the present plus 2*3*4 = 24 feet of ribbon for the bow, for a total of 34 feet.

- A present with dimensions 1x1x10 requires 1+1+1+1 = 4 feet of ribbon to wrap the present plus 1*1*10 = 10 feet of ribbon for the bow, for a total of 14 feet.

How many total feet of ribbon should they order?

```
(fn volume [w h l]
  (* w h l))

(fn perimeter [a b]
  (+ (* 2 a) (* 2 b)))

(fn solve2 [input]
  (var res 0)
  (each [_ dim (ipairs input)]
    (case (aoc.string-split dim "x")
      [x y z] (set res (+ res
                          (math.min (perimeter x y)
                                    (perimeter y z)
                                    (perimeter z x))
                          (volume x y z)))
      _ (print (.. "No match found for " dim))))
```

```
  res)

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 48 test-input)

(solve2 (aoc.string-from "2015/02.inp"))
```

```
3783758
```

DONE Day 3.1

Santa is delivering presents to an infinite two-dimensional grid of houses.

He begins by delivering a present to the house at his starting location, and then an elf at the North Pole calls him via radio and tells him where to move next. Moves are always exactly one house to the north (^), south (v), east (>), or west (<). After each move, he delivers another present to the house at his new location.

However, the elf back at the north pole has had a little too much eggnog, and so his directions are a little off, and Santa ends

up visiting some houses more than once. How many houses receive at least one present?

For example:

- > delivers presents to 2 houses: one at the starting location, and one to the east.
- ^>v< delivers presents to 4 houses in a square, including twice to the house at his starting/ending location.
- ^v^v^v^v^v delivers a bunch of presents to some very lucky children at only 2 houses.

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn visit-houses [dir res pos]
  (case dir
    "^" (tset pos :y (+ (. pos :y) 1))
    "v" (tset pos :y (+ (. pos :y) -1 ))
    ">" (tset pos :x (+ (. pos :x) 1 ))
    "<" (tset pos :x (+ (. pos :x) -1 ))
    _ (print (.. "Too much eggnog. Instruction unclear: " dir)))
  (let [cur [(. pos :x) (. pos :y)]]
    (when (not (aoc.matrix-contains? res cur))
```

```
      (table.insert res cur))))

(fn solve [input]
  (let [pos {:x 0 :y 0}
        res [[(. pos :x) (. pos :y)]]
        xs (aoc.string-toarray (. input 1))]
    (lume.map xs #(visit-houses $ res pos))
    (length res)))

(fn test [expected input]
  (assert (= expected (solve [input]))))

(test 2 ">")
(test 4 "^>v<")
(test 2 "^v^v^v^v^v")

(solve (aoc.string-from "2015/03.inp"))
```

```
2081
```

DONE Day 3.2

The next year, to speed up the process, Santa creates a robot version of himself, Robo-Santa, to deliver presents with him.

Santa and Robo-Santa start at the same location (delivering two

presents to the same starting house), then take turns moving based on instructions from the elf, who is eggnoggedly reading from the same script as the previous year.

This year, how many houses receive at least one present?

For example:

- `v` delivers presents to 3 houses, because Santa goes north, and then Robo-Santa goes south.
- `>v<` now delivers presents to 3 houses, and Santa and Robo-Santa end up back where they started.
- `vvvvv` now delivers presents to 11 houses, with Santa going one direction and Robo-Santa going the other.

```
(fn solve2 [input]
  (let [pos-odd {:x 0 :y 0}
        pos-even {:x 0 :y 0}
        res [[(. pos-odd :x) (. pos-odd :y)]]
        xs (aoc.string-toarray (. input 1))]
    (lume.map (aoc.table-odd xs)
    ↪ #(visit-houses $ res pos-odd))
    (lume.map (aoc.table-even xs)
    ↪ #(visit-houses $ res pos-even))
    (length res)))

```

```
(fn test2 [expected input]
  (assert (= expected (solve2 [input]))))

(test2 3 "^v")
(test2 3 "^>v<")
(test2 11 "^v^v^v^v^v")

(solve2 (aoc.string-from "2015/03.inp"))
```

```
2341
```

DONE Day 4.1

Santa needs help mining some AdventCoins (very similar to bitcoins) to use as gifts for all the economically forward-thinking little girls and boys.

To do this, he needs to find MD5 hashes which, in hexadecimal, start with at least five zeroes. The input to the MD5 hash is some secret key (your puzzle input, given below) followed by a number in decimal. To mine AdventCoins, you must find Santa the lowest positive number (no leading zeroes: 1, 2, 3, …) that produces such a hash.

For example:

- If your secret key is abcdef, the answer is 609043, because the MD5 hash of abcdef609043 starts with five zeroes (000001dbbfa…), and it is the lowest such number to do so.
- If your secret key is pqrstuv, the lowest number it combines with to make an MD5 hash starting with five zeroes is 1048970; that is, the MD5 hash of pqrstuv1048970 looks like 000006136ef….

```
(local md5 (require :lib.md5))

(fn solve [seed]
  (var res false)
  (var salt 0)
  (while (not res)
    (let [hash (md5.sumhexa (.. seed salt))
          prefix (string.sub hash 1 5)]
      (if (= "00000" prefix)
          (set res salt)
          (set salt (+ 1 salt)))))
  res)

(solve "yzbqklnj")
```

```
282749
```

DONE Day 4.2

Now find one that starts with six zeroes.

```
(fn solve2 [seed]
  (var res false)
  (var salt 0)
  (while (not res)
    (let [hash (md5.sumhexa (.. seed salt))
          prefix (string.sub hash 1 6)]
      (if (= "000000" prefix)
          (set res salt)
          (set salt (+ 1 salt)))))
  res)

(solve2 "yzbqklnj")
```

```
9962624
```

DONE Day 5.1

Santa needs help figuring out which strings in his text file are naughty or nice.

A nice string is one with all of the following properties:

- It contains at least three vowels (aeiou only), like aei, xazegov, or aeiouaeiouaeiou.
- It contains at least one letter that appears twice in a row, like xx, abcdde (dd), or aabbccdd (aa, bb, cc, or dd).
- It does not contain the strings ab, cd, pq, or xy, even if they are part of one of the other requirements.

For example:

- ugknbfddgicrmopn is nice because it has at least three vowels (u…i…o…), a double letter (…dd…), and none of the disallowed substrings.
- aaa is nice because it has at least three vowels and a double letter, even though the letters used by different rules overlap.
- jchzalrnumimnmhp is naughty because it has no double letter.
- haegwjzuvuyypxyu is naughty because it contains the string xy.
- dvszwmarrgswjxmb is naughty because it contains only one vowel.

How many strings are nice?

```
(local lume (require :lib.lume))
```

```
(local aoc (require :lib.aoc))
(local test-input
       ["ugknbfddgicrmopn"
        "aaa"
        "jchzalrnumimnmhp"
        "haegwjzuvuyypxyu"
        "dvszwmarrgswjxmb"])

(fn none-of-ab-p [x]
  (not (string.find x "ab" 1 true)))

(fn none-of-cd-p [x]
  (not (string.find x "cd" 1 true)))

(fn none-of-pq-p [x]
  (not (string.find x "pq" 1 true)))

(fn none-of-xy-p [x]
  (not (string.find x "xy" 1 true)))

(fn one-double-p [x]
  (let [ys (aoc.partition1 (aoc.string-toarray
  ↪  x))]
    (lume.any ys (fn [[c1 c2]] (= c1 c2)))))

```

```
(fn three-vowels-p [x]
  (var count 0)
  (each [_ c (ipairs (aoc.string-toarray x))]
    (case c
      "a" (set count (+ 1 count))
      "e" (set count (+ 1 count))
      "i" (set count (+ 1 count))
      "o" (set count (+ 1 count))
      "u" (set count (+ 1 count))
      _ nil))
  (<= 3 count))

(fn solve [input]
  (-> input
      (lume.filter #(none-of-ab-p $))
      (lume.filter #(none-of-cd-p $))
      (lume.filter #(none-of-pq-p $))
      (lume.filter #(none-of-xy-p $))
      (lume.filter #(one-double-p $))
      (lume.filter #(three-vowels-p $))
      (length)))

(fn test [expected input]
  (assert (= expected (solve input))))

```

```
(test 2 test-input)

(solve (aoc.string-from "2015/05.inp"))
```

```
236
```

DONE Day 5.2

Realizing the error of his ways, Santa has switched to a better model of determining whether a string is naughty or nice. None of the old rules apply, as they are all clearly ridiculous.

Now, a nice string is one with all of the following properties:

- It contains a pair of any two letters that appears at least twice in the string without overlapping, like xyxy (xy) or aabcdefgaa (aa), but not like aaa (aa, but it overlaps).
- It contains at least one letter which repeats with exactly one letter between them, like xyx, abcdefeghi (efe), or even aaa.

For example:

- qjhvhtzxzqqjkmpb is nice because is has a pair that appears twice (qj) and a letter that repeats with exactly one letter between them (zxz).

- xxyxx is nice because it has a pair that appears twice and a letter that repeats with one between, even though the letters used by each rule overlap.
- uurcxstgmygtbstg is naughty because it has a pair (tg) but no repeat with a single letter between them.
- ieodomkazucvgmuy is naughty because it has a repeating letter with one between (odo), but no pair that appears twice.

How many strings are nice under these new rules?

```
(local test2-input
       ["qjhvhtzxzqqjkmpb"
        "xxyxx"
        "uurcxstgmygtbstg"
        "ieodomkazucvgmuy"])

(fn two-pairs-p [x]
  (var done false)
  (let [xs (aoc.string-toarray x)]
    (for [i 1 (- (# xs) 1) &until done]
      (let [token (.. (. xs i) (. xs (+ 1
 ↪  i)))]
        (when (string.find x token (+ 2 i)
 ↪  true)
```

```
          (set done true)))))
  done)

(fn triple-p [x]
  (-> x
      (aoc.string-toarray)
      (aoc.partition3step1)
      (lume.any (fn [[a b c]] (= a c)))))

(fn solve2 [input]
  (-> input
      (lume.filter #(two-pairs-p $))
      (lume.filter #(triple-p $))
      (length)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 2 test2-input)

(solve2 (aoc.string-from "2015/05.inp"))
```

```
51
```

DONE Day 6.1

Because your neighbors keep defeating you in the holiday house decorating contest year after year, you've decided to deploy one million lights in a 1000x1000 grid.

Furthermore, because you've been especially nice this year, Santa has mailed you instructions on how to display the ideal lighting configuration.

Lights in your grid are numbered from 0 to 999 in each direction; the lights at each corner are at 0,0, 0,999, 999,999, and 999,0. The instructions include whether to turn on, turn off, or toggle various inclusive ranges given as coordinate pairs. Each coordinate pair represents opposite corners of a rectangle, inclusive; a coordinate pair like 0,0 through 2,2 therefore refers to 9 lights in a 3x3 square. The lights all start turned off.

To defeat your neighbors this year, all you have to do is set up your lights by doing the instructions Santa sent you in order.

For example:

- turn on 0,0 through 999,999 would turn on (or leave on) every light.
- toggle 0,0 through 999,0 would toggle the first line of 1000

lights, turning off the ones that were on, and turning on the ones that were off.
- turn off 499,499 through 500,500 would turn off (or leave off) the middle four lights.

After following the instructions, how many lights are lit?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-line [line]
  (case (aoc.string-split line " ")
    ["turn" "off" from "through" to]
    [0 (aoc.string-split from ",")
    ↪ (aoc.string-split to ",")]
    ["toggle" from "through" to]
    [-1 (aoc.string-split from ",")
    ↪ (aoc.string-split to ",")]
    ["turn" "on" from "through" to]
    [1 (aoc.string-split from ",")
    ↪ (aoc.string-split to ",")]
    _ (print (.. "No match for " line))))

(fn solve [input]
  (let [m (aoc.matrix 1000 1000 0)
```

```
        xs (lume.map input (fn [l] (read-line
↪ l)))]
    (each [_ [op [a1 a2] [b1 b2]] (ipairs xs)]
      (let [x1 (+ 1 (tonumber a1))
            y1 (+ 1 (tonumber a2))
            x2 (+ 1 (tonumber b1))
            y2 (+ 1 (tonumber b2))]
        (case op
          -1 (aoc.matrix-toggle m x1 y1 x2 y2)
          0 (aoc.matrix-set m x1 y1 x2 y2 0)
          1 (aoc.matrix-set m x1 y1 x2 y2 1)
          _ (print (.. "No match for " op)))))
    (aoc.table-sum m)))

(solve (aoc.string-from "2015/06.inp"))
```

```
569999
```

DONE Day 6.2

You just finish implementing your winning light pattern when you realize you mistranslated Santa's message from Ancient Nordic Elvish.

The light grid you bought actually has individual brightness

controls; each light can have a brightness of zero or more. The lights all start at zero.

The phrase turn on actually means that you should increase the brightness of those lights by 1.

The phrase turn off actually means that you should decrease the brightness of those lights by 1, to a minimum of zero.

The phrase toggle actually means that you should increase the brightness of those lights by 2.

What is the total brightness of all lights combined after following Santa's instructions?

For example:

- turn on 0,0 through 0,0 would increase the total brightness by 1.
- toggle 0,0 through 999,999 would increase the total brightness by 2000000.

```
(fn solve2 [input]
  (let [m (aoc.matrix 1000 1000 0)
        xs (lume.map input (fn [l] (read-line
↪ l)))]
    (each [_ [op [a1 a2] [b1 b2]] (ipairs xs)]
      (let [x1 (+ 1 (tonumber a1))
```

```
            y1 (+ 1 (tonumber a2))
            x2 (+ 1 (tonumber b1))
            y2 (+ 1 (tonumber b2))]
        (case op
          -1 (aoc.matrix-apply m x1 y1 x2 y2
↪  (fn [v] (+ 2 v)))
          0 (aoc.matrix-apply m x1 y1 x2 y2
↪  (fn [v] (math.max 0 (- v 1))))
          1 (aoc.matrix-apply m x1 y1 x2 y2
↪  (fn [v] (+ 1 v)))
          _ (print (.. "No match for " op)))))
    (aoc.table-sum m)))

(solve2 (aoc.string-from "2015/06.inp"))
```

```
17836115
```

DONE Day 7.1

This year, Santa brought little Bobby Tables a set of wires and bitwise logic gates! Unfortunately, little Bobby is a little under the recommended age range, and he needs help assembling the circuit.

Each wire has an identifier (some lowercase letters) and can

carry a 16-bit signal (a number from 0 to 65535). A signal is provided to each wire by a gate, another wire, or some specific value. Each wire can only get a signal from one source, but can provide its signal to multiple destinations. A gate provides no signal until all of its inputs have a signal.

The included instructions booklet describes how to connect the parts together: x AND y -> z means to connect wires x and y to an AND gate, and then connect its output to wire z.

For example:

- 123 -> x means that the signal 123 is provided to wire x.
- x AND y -> z means that the bitwise AND of wire x and wire y is provided to wire z.
- p LSHIFT 2 -> q means that the value from wire p is left-shifted by 2 and then provided to wire q.
- NOT e -> f means that the bitwise complement of the value from wire e is provided to wire f.

Other possible gates include OR (bitwise OR) and RSHIFT (right-shift). If, for some reason, you'd like to emulate the circuit instead, almost all programming languages (for example, C, JavaScript, or Python) provide operators for these gates.

For example, here is a simple circuit:

```
123 -> x
456 -> y
x AND y -> d
x OR y -> e
x LSHIFT 2 -> f
y RSHIFT 2 -> g
NOT x -> h
NOT y -> i
```

After it is run, these are the signals on the wires:

```
d: 72
e: 507
f: 492
g: 114
h: 65412
i: 65079
x: 123
y: 456
```

In little Bobby's kit's instructions booklet (provided as your puzzle input), what signal is ultimately provided to wire a?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local bit (require :bit))

```

```
(fn eval [xs k]
  (or (tonumber k)
      (let [v (?. xs k)]
        (match (aoc.string-split v " ")
          ["NOT" f] (tset xs k (bit.bnot (eval xs f)))
          [f1 "AND" f2] (tset xs k (bit.band (eval xs f1) (eval xs f2)))
          [f1 "OR" f2] (tset xs k (bit.bor (eval xs f1) (eval xs f2)))
          [f "LSHIFT" n] (tset xs k (bit.lshift (eval xs f) (tonumber n)))
          [f "RSHIFT" n] (tset xs k (bit.rshift (eval xs f) (tonumber n)))
          [f] (tset xs k (eval xs f))
          _ (print (.. "No match for " k)))
        (. xs k))))

(fn init [xs line]
  (let [[op res] (aoc.string-split line "->")]
    (tset xs (aoc.string-trim2 res) (aoc.string-trim2 op))))

(fn solve [lines k]
  (let [xs {}]
```

```
    (lume.map lines #(init xs $))
    (eval xs k)))

(solve (aoc.string-from "2015/07.inp") :a)
```

```
956
```

DONE Day 7.2

Now, take the signal you got on wire a, override wire b to that signal, and reset the other wires (including wire a). What new signal is ultimately provided to wire a?

```
(fn solve2 [lines k]
  (let [xs {}]
    (lume.map lines #(init xs $))
    (tset xs :b 956)
    (eval xs k)))

(solve2 (aoc.string-from "2015/07.inp") :a)
```

```
40149
```

DONE Day 8.1

Space on the sleigh is limited this year, and so Santa will be bringing his list as a digital copy. He needs to know how much space it will take up when stored.

It is common in many programming languages to provide a way to escape special characters in strings. For example, C, JavaScript, Perl, Python, and even PHP handle special characters in very similar ways.

However, it is important to realize the difference between the number of characters in the code representation of the string literal and the number of characters in the in-memory string itself.

For example:

- "" is 2 characters of code (the two double quotes), but the string contains zero characters.
- "abc" is 5 characters of code, but 3 characters in the string data.
- "aaa\"aaa" is 10 characters of code, but the string itself contains six "a" characters and a single, escaped quote character, for a total of 7 characters in the string data.

- `"\x27"` is 6 characters of code, but the string itself contains just one - an apostrophe (`'`), escaped using hexadecimal notation.

Santa's list is a file that contains many double-quoted string literals, one on each line. The only escape sequences used are \\ (which represents a single backslash), \" (which represents a lone double-quote character), and \x plus two hexadecimal characters (which represents a single character with that ASCII code).

Disregarding the whitespace in the file, what is the number of characters of code for string literals minus the number of characters in memory for the values of the strings in total for the entire file?

For example, given the four strings above, the total number of characters of string code (2 + 5 + 10 + 6 = 23) minus the total number of characters in memory for string values (0 + 3 + 7 + 1 = 11) is 23 - 11 = 12.

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn char-len [s]
  (var len 0)
```

```
6     (var i 1)
7     (let [xs (aoc.string-toarray s)]
8       (while (<= i (# xs))
9         (case (. xs i)
10          "\\" (set i (+ i (if (= "x" (. xs (+ 1
    ↪  i))) 4 2)))
11          _ (set i (+ i 1)))
12        (set len (+ 1 len))))
13    len)
14
15  (fn raw-len [s]
16    (+
17     (string.len "\"")
18     (string.len "\"")
19     (string.len s)))
20
21  (fn solve [lines]
22    (var len1 0)
23    (var len2 0)
24    (each [_ line (ipairs lines)]
25      (set len1 (+ len1 (raw-len line)))
26      (set len2 (+ len2 (char-len line))))
27    (- len1 len2))
28
29  (solve (aoc.string-from "2015/08.inp"))
```

1371

DONE Day 8.2

Now, let's go the other way. In addition to finding the number of characters of code, you should now encode each code representation as a new string and find the number of characters of the new encoded representation, including the surrounding double quotes.

For example:

- "" encodes to "\"\"", an increase from 2 characters to 6.
- "abc" encodes to "\"abc\"", an increase from 5 characters to 9.
- "aaa\"aaa" encodes to "\"aaa\\\"aaa\"", an increase from 10 characters to 16.
- `"\x27"` encodes to `"\"\\x27\""`, an increase from 6 characters to 11.

Your task is to find the total number of characters to represent the newly encoded strings minus the number of characters of code in each original string literal. For example, for the strings above, the total encoded length (6 + 9 + 16 + 11 = 42) minus the

characters in the original code representation (23, just like in the first part of this puzzle) is 42 - 23 = 19.

```
(fn solve2 [lines]
  (-> lines
      (lume.map (fn [s] (- (string.len
 ↪  (aoc.string-escape s))
                           (string.len s))))
      (lume.reduce (fn [s x] (+ s x)))))

(solve2 (aoc.string-from "2015/08.inp"))
```

```
2117
```

DONE Day 9.1

Every year, Santa manages to deliver all of his presents in a single night.

This year, however, he has some new locations to visit; his elves have provided him the distances between every pair of locations. He can start and end at any two (different) locations he wants, but he must visit each location exactly once. What is the shortest distance he can travel to achieve this?

For example, given the following distances:

```
London to Dublin = 464
London to Belfast = 518
Dublin to Belfast = 141
```

The possible routes are therefore:

```
Dublin -> London -> Belfast = 982
London -> Dublin -> Belfast = 605
London -> Belfast -> Dublin = 659
Dublin -> Belfast -> London = 659
Belfast -> Dublin -> London = 605
Belfast -> London -> Dublin = 982
```

The shortest of these is London -> Dublin -> Belfast = 605, and so the answer is 605 in this example.

What is the distance of the shortest route?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["London to Dublin = 464"
        "London to Belfast = 518"
        "Dublin to Belfast = 141"])

(fn read-lines [lines]
  (let [res {}]
```

```
10      (each [_ line (ipairs lines)]
11        (case (aoc.string-split line " ")
12          [dest1 "to" dest2 "=" dist]
13          (do
14            (aoc.table-update res dest1 dest2
    ↪  dist)
15            (aoc.table-update res dest2 dest1
    ↪  dist))
16          _ (print "No match found for line")))
17      res))
18
19  (fn distance [itin t]
20    (accumulate [sum 0 _ [b e] (ipairs
    ↪  (aoc.partition1 itin))]
21      (+ sum (tonumber (. (. t b) e)))))
22
23  (fn find-distances [input]
24    (let [t (read-lines input)
25          d (aoc.keys t)
26          itiner []]
27      (aoc.permutation d (# d) itiner)
28      (lume.map itiner #(distance $ t))))
29
30  (fn solve [input]
31    (-> input
```

```
      (find-distances)
      (aoc.math-min)))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 605 test-input)

(solve (aoc.string-from "2015/09.inp"))
```

```
251
```

DONE Day 9.2

The next year, just to show off, Santa decides to take the route with the longest distance instead.

He can still start and end at any two (different) locations he wants, and he still must visit each location exactly once.

For example, given the distances above, the longest route would be 982 via (for example) Dublin -> London -> Belfast.

What is the distance of the longest route?

```
(fn solve2 [input]
  (-> input
```

```
      (find-distances)
      (aoc.math-max)))

(fn test2 [expected input]
  (assert (= expected (solve2 input))))

(test2 982 test-input)

(solve2 (aoc.string-from "2015/09.inp"))
```

```
898
```

DONE Day 10.1

Today, the Elves are playing a game called look-and-say. They take turns making sequences by reading aloud the previous sequence and using that reading as the next sequence. For example, 211 is read as ”one two, two ones”, which becomes 1221 (1 2, 2 1s).

Look-and-say sequences are generated iteratively, using the previous value as input for the next step. For each step, take the previous value, and replace each run of digits (like 111) with the number of digits (3) followed by the digit itself (1).

For example:

- 1 becomes 11 (1 copy of digit 1).
- 11 becomes 21 (2 copies of digit 1).
- 21 becomes 1211 (one 2 followed by one 1).
- 1211 becomes 111221 (one 1, one 2, and two 1s).
- 111221 becomes 312211 (three 1s, two 2s, and one 1).

Starting with the digits in your puzzle input, apply this process 40 times. What is the length of the result?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn soundas [xs]
  (let [ys (aoc.partition-by xs #(= $1 $2))
        zs (lume.map ys #(.. "" (length $) (. $ 1)))]
    (table.concat zs "")))

(fn solve [input cycles]
  (var res input)
  (for [i 1 cycles]
    (let [xs (aoc.string-toarray res)]
      (set res (soundas xs))))
  (length res))
```

```
15
16 (fn test [expected input cycles]
17   (assert (= expected (solve input cycles))))
18
19 (test 6 "1" 5)
20
21 (solve "1321131112" 40)
```

```
492982
```

DONE Day 10.2

Neat, right? You might also enjoy hearing John Conway talking about this sequence (that's Conway of Conway's Game of Life fame).

Now, starting again with the digits in your puzzle input, apply this process 50 times. What is the length of the new result?

```
1 (solve "1321131112" 50)
```

```
6989950
```

DONE Day 11.1

Santa's previous password expired, and he needs help choosing a new one.

To help him remember his new password after the old one expires, Santa has devised a method of coming up with a password based on the previous one. Corporate policy dictates that passwords must be exactly eight lowercase letters (for security reasons), so he finds his new password by incrementing his old password string repeatedly until it is valid.

Incrementing is just like counting with numbers: xx, xy, xz, ya, yb, and so on. Increase the rightmost letter one step; if it was z, it wraps around to a, and repeat with the next letter to the left until one doesn't wrap around.

Unfortunately for Santa, a new Security-Elf recently started, and he has imposed some additional password requirements:

- Passwords must include one increasing straight of at least three letters, like abc, bcd, cde, and so on, up to xyz. They cannot skip letters; abd doesn't count.
- Passwords may not contain the letters i, o, or l, as these

letters can be mistaken for other characters and are therefore confusing.
- Passwords must contain at least two different, non-overlapping pairs of letters, like aa, bb, or zz.

For example:

- hijklmmn meets the first requirement (because it contains the straight hij) but fails the second requirement (because it contains i and l).
- abbceffg meets the third requirement (because it repeats bb and ff) but fails the first requirement.
- abbcegjk fails the third requirement, because it only has one double letter (bb).
- The next password after abcdefgh is abcdffaa.
- The next password after ghijklmn is ghjaabcc, because you eventually skip all the passwords that start with ghi…, since i is not allowed.

Given Santa's current password (your puzzle input), what should his next password be?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

```

```
(fn incchar [xs ?pos]
  (let [pos (or ?pos (# xs))]
    (if (not (<= 1 pos (# xs)))
        xs
        (let [xn (- (string.byte (. xs pos)) 96)
              xo (aoc.modulo+ 1 xn 26)]
          (table.remove xs pos)
          (table.insert xs pos (string.char (+ 96 xo)))
          (if (= 1 xo)
              (incchar xs (- pos 1))
              xs)))))

(fn valid? [xs]
  (and
   (lume.all xs
             #(and (not= $ "i") (not= $ "o") (not= $ "l")))
   (lume.any (aoc.partition3step1 xs)
             #(and (= (+ 2 (string.byte (. $ 1)))
                      (+ 1 (string.byte (. $ 2)))
                      (string.byte (. $ 3))))))
```

```
    (<= 2 (length (lume.filter
 ↪  (aoc.partition-by xs #(= $1 $2))
                                #(<= 2 (length
 ↪  $)))))))

(fn pwdgen [xs]
  (var ys (incchar xs))
  (while (not (valid? ys))
    (set ys (incchar ys)))
  ys)

(fn solve [input]
  (-> input
      (aoc.string-toarray)
      (pwdgen)
      (table.concat "")))

(fn test [expected input]
  (assert (= expected (solve input))))

(test "abcdffaa" "abcdefgh")
(test "ghjaabcc" "ghijklmn")
(solve "vzbxkghb")
```

```
vzbxxyzz
```

DONE Day 11.2

Santa's password expired again. What's the next one?

```
(solve "vzbxxyzz")
```

```
vzcaabcc
```

DONE Day 12.1

Santa's Accounting-Elves need help balancing the books after a recent order. Unfortunately, their accounting software uses a peculiar storage format. That's where you come in.

They have a JSON document which contains a variety of things: arrays ([1,2,3]), objects ({”a”:1, ”b”:2}), numbers, and strings. Your first job is to simply find all of the numbers throughout the document and add them together.

For example:

- [1,2,3] and {”a”:2,”b”:4} both have a sum of 6.
- [[[3]]] and {”a”:{”b”:4},”c”:-1} both have a sum of 3.
- {”a”:[-1,1]} and [-1,{”a”:1}] both have a sum of 0.
- [] and {} both have a sum of 0.

You will not encounter any strings containing numbers.

What is the sum of all numbers in the document?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local json (require :lib.dkjson))

(fn sum-numbers [x]
  (case (type x)
    :string 0
    :number x
    :table
    (if (= 0 (length x))
        (accumulate [sum 0 k v (pairs x)]
          (+ sum (sum-numbers v)))
        (accumulate [sum 0 k v (ipairs x)]
          (+ sum (sum-numbers v))))
    _ (print (.. "No match found for " x))))

(fn solve [input]
  (-> (. input 1)
      (json.decode)
      (sum-numbers)))

(fn test [expected input]
```

```
23    (assert (= expected (solve [input]))))
24
25  (test 6 "[1,2,3]")
26  (test 6 "{\"a\":2,\"b\":4}")
27  (test 3 "[[[3] ] ]")
28  (test 3 "{\"a\":{\"b\":4},\"c\":-1}")
29  (test 0 "{\"a\":[-1,1]}")
30  (test 0 "[-1,{\"a\":1}]")
31  (test 0 "[]")
32  (test 0 "{}")
33
34  (solve (aoc.string-from "2015/12.inp"))
```

```
191164
```

DONE Day 12.2

Uh oh - the Accounting-Elves have realized that they double-counted everything red.

Ignore any object (and all of its children) which has any property with the value ”red”. Do this only for objects ({…}), not arrays ([…]).

- [1,2,3] still has a sum of 6.

- [1,{”c”:”red”,”b”:2},3] now has a sum of 4, because the middle object is ignored.
- {”d”:”red”,”e”:[1,2,3,4],”f”:5} now has a sum of 0, because the entire structure is ignored.
- [1,”red”,5] has a sum of 6, because ”red” in an array has no effect.

```
(fn sum-numbers2 [x]
  (case (type x)
    :string 0
    :number x
    :table
    (if (= 0 (length x))
        (if (lume.any x #(= "red" $))
            0
            (accumulate [sum 0 k v (pairs x)]
              (+ sum (sum-numbers2 v))))
        (accumulate [sum 0 k v (ipairs x)]
          (+ sum (sum-numbers2 v))))
    _ (print (.. "No match found for " x))))

(fn solve2 [input]
  (-> (. input 1)
      (json.decode)
      (sum-numbers2)))
```

```
19
20 (solve2 (aoc.string-from "2015/12.inp"))
```

```
87842
```

DONE Day 13.1

In years past, the holiday feast with your family hasn't gone so well. Not everyone gets along! This year, you resolve, will be different. You're going to find the optimal seating arrangement and avoid all those awkward conversations.

You start by writing up a list of everyone invited and the amount their happiness would increase or decrease if they were to find themselves sitting next to each other person. You have a circular table that will be just big enough to fit everyone comfortably, and so each person will have exactly two neighbors.

For example, suppose you have only four attendees planned, and you calculate their potential happiness as follows:

```
Alice would gain 54 happiness units by sitting
 ↪  next to Bob.
Alice would lose 79 happiness units by sitting
 ↪  next to Carol.
```

```
Alice would lose 2 happiness units by sitting
 ↪  next to David.
Bob would gain 83 happiness units by sitting
 ↪  next to Alice.
Bob would lose 7 happiness units by sitting
 ↪  next to Carol.
Bob would lose 63 happiness units by sitting
 ↪  next to David.
Carol would lose 62 happiness units by sitting
 ↪  next to Alice.
Carol would gain 60 happiness units by sitting
 ↪  next to Bob.
Carol would gain 55 happiness units by sitting
 ↪  next to David.
David would gain 46 happiness units by sitting
 ↪  next to Alice.
David would lose 7 happiness units by sitting
 ↪  next to Bob.
David would gain 41 happiness units by sitting
 ↪  next to Carol.
```

Then, if you seat Alice next to David, Alice would lose 2 happiness units (because David talks so much), but David would gain 46 happiness units (because Alice is such a good listener), for a total change of 44.

If you continue around the table, you could then seat Bob next to Alice (Bob gains 83, Alice gains 54). Finally, seat Carol, who sits next to Bob (Carol gains 60, Bob loses 7) and David (Carol gains 55, David gains 41). The arrangement looks like this:

```
     +41 +46
+55   David    -2
Carol       Alice
+60    Bob    +54
     -7  +83
```

After trying every other seating arrangement in this hypothetical scenario, you find that this one is the most optimal, with a total change in happiness of 330.

What is the total change in happiness for the optimal seating arrangement of the actual guest list?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input
       ["Alice would gain 54 happiness units by sitting next to Bob."
        "Alice would lose 79 happiness units by sitting next to Carol."
        "Alice would lose 2 happiness units by sitting next to David."
```

```
7           "Bob would gain 83 happiness units by
    ↪  sitting next to Alice."
8           "Bob would lose 7 happiness units by
    ↪  sitting next to Carol."
9           "Bob would lose 63 happiness units by
    ↪  sitting next to David."
10          "Carol would lose 62 happiness units
    ↪  by sitting next to Alice."
11          "Carol would gain 60 happiness units
    ↪  by sitting next to Bob."
12          "Carol would gain 55 happiness units
    ↪  by sitting next to David."
13          "David would gain 46 happiness units
    ↪  by sitting next to Alice."
14          "David would lose 7 happiness units by
    ↪  sitting next to Bob."
15          "David would gain 41 happiness units
    ↪  by sitting next to Carol."])
16
17  (fn read-line [line xs]
18    (case (aoc.string-split line " ")
19      [me "would" "gain" x "happiness" "units"
    ↪  "by" "sitting" "next" "to" who]
20      (aoc.table-update xs me (string.sub who 1
    ↪  (- (# who) 1)) (tonumber x))
```

```
    [me "would" "lose" x "happiness" "units"
    "by" "sitting" "next" "to" who]
    (aoc.table-update xs me (string.sub who 1
    (- (# who) 1)) (- (tonumber x)))
    _ (print (.. "No match found for "
    line))))

(fn read-lines [lines]
  (let [scores {}]
    (each [_ line (ipairs lines)]
      (read-line line scores))
    scores))

(fn score-pairs [xs scores]
  (let [fst (. xs 1)
        lst (. xs (# xs))
        ys (aoc.partition1 xs)
        res [(+ (. (. scores fst) lst) (. (.
    scores lst) fst))]]
    (each [_ [p1 p2] (ipairs ys)]
      (table.insert res (+ (. (. scores p1)
    p2) (. (. scores p2) p1))))
    res))

(fn score-arrangements [lines]
```

```
  (let [scores (read-lines lines)
        guests (aoc.keys scores)
        arrangement []]
    (aoc.permutation guests (# guests)
    ↪ arrangement)
    (lume.map arrangement #(score-pairs $
    ↪ scores))))

(fn solve [lines]
  (let [xs (score-arrangements lines)]
    (aoc.math-max (lume.map xs #(aoc.table-sum
    ↪ $)))))

(fn test [expected input]
  (assert (= expected (solve input))))

(test 330 test-input)

(solve (aoc.string-from "2015/13.inp"))
```

```
733
```

DONE Day 13.2

In all the commotion, you realize that you forgot to seat yourself. At this point, you're pretty apathetic toward the whole thing, and your happiness wouldn't really go up or down regardless of who you sit next to. You assume everyone else would be just as ambivalent about sitting next to you, too.

So, add yourself to the list, and give all happiness relationships that involve you a score of 0.

What is the total change in happiness for the optimal seating arrangement that actually includes yourself?

```
(fn solve2 [lines]
  (let [xs (score-arrangements lines)]
    (each [_ x (ipairs xs)]
        (table.sort x)
        ;; maximize happiness sitting at its
 ↪  min
        (aoc.table-swap x 1 0))
    (aoc.math-max (lume.map xs #(aoc.table-sum
 ↪  $)))))

(solve2 (aoc.string-from "2015/13.inp"))
```

```
725
```

DONE Day 14.1

This year is the Reindeer Olympics! Reindeer can fly at high speeds, but must rest occasionally to recover their energy. Santa would like to know which of his reindeer is fastest, and so he has them race.

Reindeer can only either be flying (always at their top speed) or resting (not moving at all), and always spend whole seconds in either state.

For example, suppose you have the following Reindeer:

- Comet can fly 14 km/s for 10 seconds, but then must rest for 127 seconds.
- Dancer can fly 16 km/s for 11 seconds, but then must rest for 162 seconds.

After one second, Comet has gone 14 km, while Dancer has gone 16 km. After ten seconds, Comet has gone 140 km, while Dancer has gone 160 km. On the eleventh second, Comet begins resting (staying at 140 km), and Dancer continues on for a total distance of 176 km. On the 12th second, both reindeer are resting. They continue to rest until the 138th second, when Comet flies for another ten seconds. On the 174th second, Dancer flies for

another 11 seconds.

In this example, after the 1000th second, both reindeer are resting, and Comet is in the lead at 1120 km (poor Dancer has only gotten 1056 km by that point). So, in this situation, Comet would win (if the race ended at 1000 seconds).

Given the descriptions of each reindeer (in your puzzle input), after exactly 2503 seconds, what distance has the winning reindeer traveled?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(local test-input
       ["Comet can fly 14 km/s for 10 seconds, but then must rest for 127 seconds."
        "Dancer can fly 16 km/s for 11 seconds, but then must rest for 162 seconds."])

(fn read-line [line]
  (case (aoc.string-split line " ")
    [n "can" "fly" km "km/s" "for" m "seconds," "but" "then" "must" "rest" "for" s "seconds."]
```

```
    {:n n :km (tonumber km) :m (tonumber m) :s
↪   (tonumber s)}
    _ (print (.. "No match for " line))))

(fn time-to-dist [xs t]
  (let [n (. xs :n)
        km (. xs :km)
        m (. xs :m)
        s (. xs :s)
        cycles (math.floor (/ t (+ m s)))
        reminder (- t (* cycles (+ m s)))]
    {:n n :d (+ (* km m cycles)
                (if (< reminder m)
                    (* km reminder)
                    (* km m)))}))

(fn solve [input maxtime]
  (-> input
      (lume.map read-line)
      (lume.map #(time-to-dist $ maxtime))
      (lume.map #(. $ :d))
      (aoc.table-max)))

(fn test [expected input]
  (assert (= expected (solve input 1000))))
```

```
35
36 (test 1120 test-input)
37
38 (solve (aoc.string-from "2015/14.inp") 2503)
```

```
2696
```

DONE Day 14.2

Seeing how reindeer move in bursts, Santa decides he's not pleased with the old scoring system.

Instead, at the end of each second, he awards one point to the reindeer currently in the lead. (If there are multiple reindeer tied for the lead, they each get one point.) He keeps the traditional 2503 second time limit, of course, as doing otherwise would be entirely ridiculous.

Given the example reindeer from above, after the first second, Dancer is in the lead and gets one point. He stays in the lead until several seconds into Comet's second burst: after the 140th second, Comet pulls into the lead and gets his first point. Of course, since Dancer had been in the lead for the 139 seconds before that, he has accumulated 139 points by the 140th second.

After the 1000th second, Dancer has accumulated 689 points, while poor Comet, our old champion, only has 312. So, with the new scoring system, Dancer would win (if the race ended at 1000 seconds).

Again given the descriptions of each reindeer (in your puzzle input), after exactly 2503 seconds, how many points does the winning reindeer have?

```
(fn in-motion? [t0 t1 t2]
  (<= 1 (% t0 (+ t1 t2)) t1))

(fn update-distance [x time]
  (when (in-motion? time (. x :m) (. x :s))
    (tset x :d (+ (. x :km) (or (. x :d)
 ↪  0)))))

(fn award-points [x dist]
  (when (= dist (. x :d))
    (tset x :p (+ 1 (or (. x :p) 0)))))

(fn solve2 [input maxtime]
  (let [xs (lume.map input read-line)]
    (for [i 1 maxtime]
      (lume.each xs #(update-distance $ i))
```

```
      (let [dist (aoc.table-max (lume.map xs
↪  #(. $ :d)))]
        (lume.each xs #(award-points $
↪  dist))))
    (aoc.table-max (lume.map xs #(. $ :p)))))

(fn test2 [expected input]
  (assert (= expected (solve2 input 1000))))

(test2 689 test-input)

(solve2 (aoc.string-from "2015/14.inp") 2503)
```

```
1084
```

DONE Day 15.1

Today, you set out on the task of perfecting your milk-dunking cookie recipe. All you have to do is find the right balance of ingredients.

Your recipe leaves room for exactly 100 teaspoons of ingredients. You make a list of the remaining ingredients you could use to finish the recipe (your puzzle input) and their properties per teaspoon:

- capacity (how well it helps the cookie absorb milk)
- durability (how well it keeps the cookie intact when full of milk)
- flavor (how tasty it makes the cookie)
- texture (how it improves the feel of the cookie)
- calories (how many calories it adds to the cookie)

You can only measure ingredients in whole-teaspoon amounts accurately, and you have to be accurate so you can reproduce your results in the future. The total score of a cookie can be found by adding up each of the properties (negative totals become 0) and then multiplying together everything except calories.

For instance, suppose you have these two ingredients:

```
Butterscotch: capacity -1, durability -2,
 ↪  flavor 6, texture 3, calories 8
Cinnamon: capacity 2, durability 3, flavor -2,
 ↪  texture -1, calories 3
```

Then, choosing to use 44 teaspoons of butterscotch and 56 teaspoons of cinnamon (because the amounts of each ingredient must add up to 100) would result in a cookie with the following properties:

- A capacity of 44*-1 + 56*2 = 68
- A durability of 44*-2 + 56*3 = 80
- A flavor of 44*6 + 56*-2 = 152
- A texture of 44*3 + 56*-1 = 76

Multiplying these together (68 * 80 * 152 * 76, ignoring calories for now) results in a total score of 62842880, which happens to be the best score possible given these ingredients. If any properties had produced a negative total, it would have instead become zero, causing the whole score to multiply to zero.

Given the ingredients in your kitchen and their properties, what is the total score of the highest-scoring cookie you can make?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-lines [lines]
  (let [res []]
    (each [_ line (ipairs lines)]
      (case (aoc.string-split line " ")
        [i "capacity" c "durability" d
 ↪  "flavor" f "texture" t "calories" cl]
        (table.insert res
                      {:ing (string.sub i 1
 ↪  -2)
```

```
                        :cap (tonumber
    ↪ (string.sub c 1 -2))
                        :dur (tonumber
    ↪ (string.sub d 1 -2))
                        :fla (tonumber
    ↪ (string.sub f 1 -2))
                        :tex (tonumber
    ↪ (string.sub t 1 -2))
                        :cal (tonumber cl)})
        _ (print (.. "No match found for "
    ↪ line))))
    res))

(fn score-ingredient [n x]
  [(* n (. x :cap))
   (* n (. x :dur))
   (* n (. x :fla))
   (* n (. x :tex))
   (* n (. x :cal))])

(fn score-recipe [xs ?cals]
  (let [ys (lume.map xs #(score-ingredient (.
    ↪ $ 1) (. $ 2)))
        zs (aoc.table-column-sum ys)
        cal (table.remove zs)]
```

```
    (if ?cals
        (if (= ?cals cal)
            (-> zs
                (lume.map #(if (< 0 $) $ 0))
                (lume.reduce #(* $1 $2)))
            0)
        (-> zs
            (lume.map #(if (< 0 $) $ 0))
            (lume.reduce #(* $1 $2))))))

(fn solve [input ?cal]
  (let [xs (read-lines input)
        res []]
    (for [i 1 100]
      (for [j 1 (- 100 i)]
        (for [k 1 (- 100 i j)]
          (table.insert res
           (score-recipe [[i (. xs 1)] [j (. xs 2)] [k (. xs 3)]
                          [(- 100 i j k) (. xs 4)]] ?cal)))))
    (aoc.table-max (lume.filter res #(< 0 $)))))

(solve (aoc.string-from "2015/15.inp"))
```

```
21367368
```

DONE Day 15.2

Your cookie recipe becomes wildly popular! Someone asks if you can make another recipe that has exactly 500 calories per cookie (so they can use it as a meal replacement). Keep the rest of your award-winning process the same (100 teaspoons, same ingredients, same scoring system).

For example, given the ingredients above, if you had instead selected 40 teaspoons of butterscotch and 60 teaspoons of cinnamon (which still adds to 100), the total calorie count would be 40*8 + 60*3 = 500. The total score would go down, though: only 57600000, the best you can do in such trying circumstances.

Given the ingredients in your kitchen and their properties, what is the total score of the highest-scoring cookie you can make with a calorie total of 500?

```
(solve (aoc.string-from "2015/15.inp") 500)
```

```
1766400
```

DONE Day 16.1

Your Aunt Sue has given you a wonderful gift, and you'd like to send her a thank you card. However, there's a small problem: she signed it ”From, Aunt Sue”.

You have 500 Aunts named ”Sue”.

So, to avoid sending the card to the wrong person, you need to figure out which Aunt Sue (which you conveniently number 1 to 500, for sanity) gave you the gift. You open the present and, as luck would have it, good ol' Aunt Sue got you a My First Crime Scene Analysis Machine! Just what you wanted. Or needed, as the case may be.

The My First Crime Scene Analysis Machine (MFCSAM for short) can detect a few specific compounds in a given sample, as well as how many distinct kinds of those compounds there are. According to the instructions, these are what the MFCSAM can detect:

- children, by human DNA age analysis.
- cats. It doesn't differentiate individual breeds.
- Several seemingly random breeds of dog: samoyeds, pomeranians, akitas, and vizslas.

- goldfish. No other kinds of fish.
- trees, all in one group.
- cars, presumably by exhaust or gasoline or something.
- perfumes, which is handy, since many of your Aunts Sue wear a few kinds.

In fact, many of your Aunts Sue have many of these. You put the wrapping from the gift into the MFCSAM. It beeps inquisitively at you a few times and then prints out a message on ticker tape:

```
children: 3
cats: 7
samoyeds: 2
pomeranians: 3
akitas: 0
vizslas: 0
goldfish: 5
trees: 3
cars: 2
perfumes: 1
```

You make a list of the things you can remember about each Aunt Sue. Things missing from your list aren't zero - you simply don't remember the value.

What is the number of the Sue that got you the gift?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-input [input]
  (let [res []]
    (each [_ line (ipairs input)]
      (case (aoc.string-split line " ")
       ["Sue" n p1 v1 p2 v2 p3 v3]
       (table.insert res
        {:n (tonumber (string.sub n 1 -2))
         (string.sub p1 1 -2) (tonumber
 ↪  (string.sub v1 1 -2))
         (string.sub p2 1 -2) (tonumber
 ↪  (string.sub v2 1 -2))
         (string.sub p3 1 -2) (tonumber v3)})
       _ (print (.. "No match for " line))))
    res))

(fn pred [xs k v]
  (let [res (. xs k)]
    (if (= nil res) true
        (= v res) true
        false)))

```

```
(fn solve [input]
  (-> input
      (read-input)
      (lume.filter #(pred $ :children 3))
      (lume.filter #(pred $ :cats 7))
      (lume.filter #(pred $ :samoyeds 2))
      (lume.filter #(pred $ :pomeranians 3))
      (lume.filter #(pred $ :akitas 0))
      (lume.filter #(pred $ :vizslas 0))
      (lume.filter #(pred $ :goldfish 5))
      (lume.filter #(pred $ :trees 3))
      (lume.filter #(pred $ :cars 2))
      (lume.filter #(pred $ :perfumes 1))
      (. 1)
      (. :n)))

(solve (aoc.string-from "2015/16.inp"))
```

```
373
```

DONE Day 16.2

As you're about to send the thank you note, something in the MFCSAM's instructions catches your eye. Apparently, it has an outdated retroencabulator, and so the output from the ma-

chine isn't exact values - some of them indicate ranges.

In particular, the `cats` and `trees` readings indicates that there are greater than that many (due to the unpredictable nuclear decay of cat dander and tree pollen), while the `pomeranians` and `goldfish` readings indicate that there are fewer than that many (due to the modial interaction of magnetoreluctance).

What is the number of the real Aunt Sue?

```
(fn pred2 [xs k v]
  (let [res (. xs k)]
    (if (= nil res) true
        (< v res) true
        false)))

(fn pred3 [xs k v]
  (let [res (. xs k)]
    (if (= nil res) true
        (> v res) true
        false)))

(fn solve2 [input]
  (-> input
      (read-input)
```

```
      (lume.filter #(pred $ :children 3))
      (lume.filter #(pred2 $ :cats 7))
      (lume.filter #(pred $ :samoyeds 2))
      (lume.filter #(pred3 $ :pomeranians 3))
      (lume.filter #(pred $ :akitas 0))
      (lume.filter #(pred $ :vizslas 0))
      (lume.filter #(pred3 $ :goldfish 5))
      (lume.filter #(pred2 $ :trees 3))
      (lume.filter #(pred $ :cars 2))
      (lume.filter #(pred $ :perfumes 1))
      (. 1)
      (. :n)))

(solve2 (aoc.string-from "2015/16.inp"))
```

```
260
```

DONE Day 17.1

The elves bought too much eggnog again - 150 liters this time. To fit it all into your refrigerator, you'll need to move it into smaller containers. You take an inventory of the capacities of the available containers.

For example, suppose you have containers of size 20, 15, 10, 5,

and 5 liters. If you need to store 25 liters, there are four ways to do it:

- 15 and 10
- 20 and 5 (the first 5)
- 20 and 5 (the second 5)
- 15, 5, and 5

Filling all containers entirely, how many different combinations of containers can exactly fit all 150 liters of eggnog?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
(local test-input ["20" "15" "10" "5" "5"])

(fn solve [input total]
  (-> input
      (lume.map tonumber)
      (aoc.powerset)
      (lume.filter #(= total (aoc.table-sum $)))
      (length)))

(fn test [expected input total]
  (assert (= expected (solve input total))))

```

```
(test 4 test-input 25)

(solve (aoc.string-from "2015/17.inp") 150)
```

```
1304
```

DONE Day 17.2

While playing with all the containers in the kitchen, another load of eggnog arrives! The shipping and receiving department is requesting as many containers as you can spare.

Find the minimum number of containers that can exactly fit all 150 liters of eggnog. How many different ways can you fill that number of containers and still hold exactly 150 litres?

In the example above, the minimum number of containers was two. There were three ways to use that many containers, and so the answer there would be 3.

```
(fn solve2 [input total]
  (let [xs (lume.map input #(tonumber $))
        ys (aoc.powerset xs)
        zs (lume.filter ys #(= total
 ↪  (aoc.table-sum $)))
        len (lume.map zs (fn [z] (length z)))]
```

```
    (let [min (aoc.table-min len)]
      (lume.count zs (fn [z] (= min (#
↪ z)))))))

(fn test2 [expected input max]
  (assert (= expected (solve2 input max))))

(test2 3 test-input 25)

(solve2 (aoc.string-from "2015/17.inp") 150)
```

```
18
```

DONE Day 18.1

After the million lights incident, the fire code has gotten stricter: now, at most ten thousand lights are allowed. You arrange them in a 100x100 grid.

Never one to let you down, Santa again mails you instructions on the ideal lighting configuration. With so few lights, he says, you'll have to resort to animation.

Start by setting your lights to the included initial configuration (your puzzle input). A # means ”on”, and a . means ”off”.

Then, animate your grid in steps, where each step decides the next configuration based on the current one. Each light's next state (either on or off) depends on its current state and the current states of the eight lights adjacent to it (including diagonals). Lights on the edge of the grid might have fewer than eight neighbors; the missing ones always count as "off".

For example, in a simplified 6x6 grid, the light marked A has the neighbors numbered 1 through 8, and the light marked B, which is on an edge, only has the neighbors marked 1 through 5:

```
1B5...
234...
......
..123.
..8A4.
..765.
```

The state a light should have next is based on its current state (on or off) plus the number of neighbors that are on:

- A light which is on stays on when 2 or 3 neighbors are on, and turns off otherwise.
- A light which is off turns on if exactly 3 neighbors are on, and stays off otherwise.

All of the lights update simultaneously; they all consider the same current state before moving to the next.

Here's a few steps from an example configuration of another 6x6 grid:

```
Initial state:
.#.#.#
...##.
#....#
..#...
#.#..#
####..

After 1 step:
..##..
..##.#
...##.
......
#.....
#.##..

After 2 steps:
..###.
......
..###.
```

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

After 3 steps:
...#..
......
...#..
..##..
......
......

After 4 steps:
......
......
..##..
..##..
......
......
```

After 4 steps, this example has four lights on.

In your grid of 100x100 lights, given your initial configuration, how many lights are on after 100 steps?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))
```

```
3
4  (local test-input
5         [".#.#.#"
6          "...##."
7          "#....#"
8          "..#..."
9          "#.#..#"
10         "####.."])
11
12 (fn dothash2num [s]
13   (let [xs (aoc.string-toarray s)]
14     (lume.map xs (fn [x] (if (= "#" x) 1
    ↪  0)))))
15
16 (fn animate [xs]
17   (let [res (aoc.matrix-clone xs)]
18     (for [i 1 (# xs)]
19       (for [j 1 (# (. xs i))]
20         (let [ns (aoc.matrix-adjvals xs i j)
21               nsum (aoc.table-sum ns)]
22           (aoc.matrix-swap
23            res j i
24            (case (. (. xs i) j)
25              0 (if (= 3 nsum) 1 0)
26              1 (if (or (= 2 nsum)
```

```
                          (= 3 nsum)) 1 0))))))
    res))

(fn solve [input steps]
  (var xs (lume.map input dothash2num))
  (for [i 1 steps]
    (set xs (animate xs)))
  (aoc.table-sum xs))

(fn test [expected input steps]
  (assert (= expected (solve input steps))))

(test 4 test-input 4)

(solve (aoc.string-from "2015/18.inp") 100)
```

```
814
```

DONE Day 18.2

You flip the instructions over; Santa goes on to point out that this is all just an implementation of Conway's Game of Life. At least, it was, until you notice that something's wrong with the grid of lights you bought: four lights, one in each corner, are

`stuck` on and can't be turned off. The example above will actually run like this:

```
Initial state:
##.#.#
...##.
#....#
..#...
#.#..#
####.#

After 1 step:
#.##.#
####.#
...##.
......
#...#.
#.####

After 2 steps:
#..#.#
#....#
.#.##.
...##.
.#..##
##.###
```

```
After 3 steps:
#...##
####.#
..##.#
......
##....
####.#

After 4 steps:
#.####
#....#
...#..
.##...
#.....
#.#..#

After 5 steps:
##.###
.##..#
.##...
.##...
#.#...
##...#
```

After 5 steps, this example now has 17 lights on.

In your grid of 100x100 lights, given your initial configuration, but with the four corners always in the on state, how many lights are on after 100 steps?

```
(fn animate2 [xs]
  (let [res (aoc.matrix-clone xs)]
    (for [i 1 (# xs)]
      (for [j 1 (# (. xs i))]
        (let [ns (aoc.matrix-adjvals xs i j)
              nsum (aoc.table-sum ns)]
          (aoc.matrix-swap
           res j i
           (if (= 3 (# ns)) 1 ;; corners have 3 neighbours
               (case (. (. xs i) j)
                 0 (if (= 3 nsum) 1 0)
                 1 (if (or (= 2 nsum)
                           (= 3 nsum)) 1 0)))))))
    res))

(fn reset-corners [xs v]
  (aoc.matrix-swap xs 1 1 v)
  (aoc.matrix-swap xs (# (. xs 1)) 1 v)
  (aoc.matrix-swap xs 1 (# xs) v)
  (aoc.matrix-swap xs (# (. xs 1)) (# xs) v)
```

```
  xs)

(fn solve2 [input steps]
  (var xs (reset-corners (lume.map input
 ↪  dothash2num) 1))
  (for [i 1 steps]
    (set xs (animate2 xs)))
  (aoc.table-sum xs))

(fn test2 [expected input steps]
  (assert (= expected (solve2 input steps))))

(test2 17 test-input 5)

(solve2 (aoc.string-from "2015/18.inp") 100)
```

```
924
```

DONE Day 19.1

Rudolph the Red-Nosed Reindeer is sick! His nose isn't shining very brightly, and he needs medicine.

Red-Nosed Reindeer biology isn't similar to regular reindeer biology; Rudolph is going to need custom-made medicine. Unfor-

tunately, Red-Nosed Reindeer chemistry isn't similar to regular reindeer chemistry, either.

The North Pole is equipped with a Red-Nosed Reindeer nuclear fusion/fission plant, capable of constructing any Red-Nosed Reindeer molecule you need. It works by starting with some input molecule and then doing a series of replacements, one per step, until it has the right molecule.

However, the machine has to be calibrated before it can be used. Calibration involves determining the number of molecules that can be generated in one step from a given starting point.

For example, imagine a simpler machine that supports only the following replacements:

```
H => HO
H => OH
O => HH
```

Given the replacements above and starting with HOH, the following molecules could be generated:

- HOOH (via H => HO on the first H).
- HOHO (via H => HO on the second H).
- OHOH (via H => OH on the first H).

- HOOH (via H => OH on the second H).
- HHHH (via O => HH).

So, in the example above, there are 4 distinct molecules (not five, because HOOH appears twice) after one replacement from HOH. Santa's favorite molecule, HOHOHO, can become 7 distinct molecules (over nine replacements: six from H, and three from O).

The machine replaces without regard for the surrounding characters. For example, given the string H2O, the transition H => OO would result in OO2O.

Your puzzle input describes all of the possible replacements and, at the bottom, the medicine molecule for which you need to calibrate the machine. How many distinct molecules can be created after all the different ways you can do one replacement on the medicine molecule?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn read-input [lines]
  (let [rules {}]
    (var mol "")
    (each [_ line (ipairs lines)]
```

```
      (case (aoc.string-split line " ")
        [m1 "=>" m2]
        (let [v (?. rules m1)]
          (if (= nil v)
              (tset rules m1 [m2])
              (table.insert v m2)))
        [molecule] (set mol molecule)
        "" nil))
    {:mol mol :rules rules}))

(fn string-gsubs [a b c r ?j]
  (let [(i res) (aoc.string-subs a b c ?j)]
    (if (= 0 i) r
        (do
          (table.insert r res)
          (string-gsubs a b c r i)))))

(fn transform [{: mol : rules}]
  (let [res {}]
    (each [_ k (ipairs (aoc.keys rules))]
      (each [_ v (ipairs (. rules k))]
        (lume.map (string-gsubs mol k v [])
                  #(tset res $ true))))
    (aoc.keys res)))

```

```
(fn solve [input]
  (-> input
      read-input
      transform
      length))

(solve (aoc.string-from "2015/19.inp"))
```

```
518
```

DONE Day 20.1

To keep the Elves busy, Santa has them deliver some presents by hand, door-to-door. He sends them down a street with infinite houses numbered sequentially: 1, 2, 3, 4, 5, and so on.

Each Elf is assigned a number, too, and delivers presents to houses based on that number:

- The first Elf (number 1) delivers presents to every house: 1, 2, 3, 4, 5, ….
- The second Elf (number 2) delivers presents to every second house: 2, 4, 6, 8, 10, ….
- Elf number 3 delivers presents to every third house: 3, 6, 9, 12, 15, ….

There are infinitely many Elves, numbered starting with 1. Each Elf delivers presents equal to ten times his or her number at each house.

So, the first nine houses on the street end up like this:

```
House 1 got 10 presents.
House 2 got 30 presents.
House 3 got 40 presents.
House 4 got 70 presents.
House 5 got 60 presents.
House 6 got 120 presents.
House 7 got 80 presents.
House 8 got 150 presents.
House 9 got 130 presents.
```

The first house gets 10 presents: it is visited only by Elf 1, which delivers 1 * 10 = 10 presents. The fourth house gets 70 presents, because it is visited by Elves 1, 2, and 4, for a total of 10 + 20 + 40 = 70 presents.

What is the lowest house number of the house to get at least `29000000` presents?

```
(fn count-presents [house]
  (var sum 0)
  (for [elf 1 house]
```

```
    (when (= 0 (% house elf))
      (set sum (+ sum (* 10 elf)))))
  sum)

(fn solve [input]
  (var res false)
  (var house 1)
  (while (not res)
    (if (<= input (count-presents house))
        (set res house)
        (set house (+ 1 house))))
  res)

(solve 29000000)
```

```
665280
```

DONE Day 20.2

The Elves decide they don't want to visit an infinite number of houses. Instead, each Elf will stop after delivering presents to 50 houses. To make up for it, they decide to deliver presents equal to eleven times their number at each house.

With these changes, what is the new lowest house number of

the house to get at least as many presents as the number in your puzzle input?

```
(fn count-presents2 [house]
  (var sum 0)
  (for [elf 1 house]
    (when (and (= 0 (% house elf))
               (<= (/ house elf) 50))
      (set sum (+ sum (* 11 elf)))))
  sum)

(fn solve2 [input]
  (var res false)
  (var house 1)
  (while (not res)
    (if (<= input (count-presents2 house))
        (set res house)
        (set house (+ 1 house))))
  res)

(solve2 29000000)
```

```
705600
```

DONE Day 21.1

Little Henry Case got a new video game for Christmas. It's an RPG, and he's stuck on a boss. He needs to know what equipment to buy at the shop. He hands you the controller.

In this game, the player (you) and the enemy (the boss) take turns attacking. The player always goes first. Each attack reduces the opponent's hit points by at least 1. The first character at or below 0 hit points loses.

Damage dealt by an attacker each turn is equal to the attacker's damage score minus the defender's armor score. An attacker always does at least 1 damage. So, if the attacker has a damage score of 8, and the defender has an armor score of 3, the defender loses 5 hit points. If the defender had an armor score of 300, the defender would still lose 1 hit point.

Your damage score and armor score both start at zero. They can be increased by buying items in exchange for gold. You start with no items and have as much gold as you need. Your total damage or armor is equal to the sum of those stats from all of your items. You have 100 hit points.

Here is what the item shop is selling:

```
Weapons:    Cost  Damage  Armor
Dagger        8     4       0
Shortsword   10     5       0
Warhammer    25     6       0
Longsword    40     7       0
Greataxe     74     8       0

Armor:      Cost  Damage  Armor
Leather      13     0       1
Chainmail    31     0       2
Splintmail   53     0       3
Bandedmail   75     0       4
Platemail   102     0       5

Rings:      Cost  Damage  Armor
Damage +1    25     1       0
Damage +2    50     2       0
Damage +3   100     3       0
Defense +1   20     0       1
Defense +2   40     0       2
Defense +3   80     0       3
```

You must buy exactly one weapon; no dual-wielding. Armor is optional, but you can't use more than one. You can buy 0-2 rings (at most one for each hand). You must use any items you

buy. The shop only has one of each item, so you can't buy, for example, two rings of Damage +3.

For example, suppose you have 8 hit points, 5 damage, and 5 armor, and that the boss has 12 hit points, 7 damage, and 2 armor:

- The player deals 5-2 = 3 damage; the boss goes down to 9 hit points.
- The boss deals 7-5 = 2 damage; the player goes down to 6 hit points.
- The player deals 5-2 = 3 damage; the boss goes down to 6 hit points.
- The boss deals 7-5 = 2 damage; the player goes down to 4 hit points.
- The player deals 5-2 = 3 damage; the boss goes down to 3 hit points.
- The boss deals 7-5 = 2 damage; the player goes down to 2 hit points.
- The player deals 5-2 = 3 damage; the boss goes down to 0 hit points.

In this scenario, the player wins! (Barely.)

You have 100 hit points. The boss's actual stats are in your

puzzle input. What is the least amount of gold you can spend and still win the fight?

```
(local lume (require :lib.lume))
(local aoc (require :lib.aoc))

(fn play2win [attacker defender]
  (if (<= (. attacker :hits) 0)
      defender
      (let [damage (- (. attacker :damage)
                      (. defender :armor))]
        (tset defender :hits (- (. defender :hits)
                                (if (<= damage 0) 1 damage)))
        (play2win defender attacker))))

(fn game [boss inventory score]
  (let [weap (. inventory :weapons)
        arm (. inventory :armor)
        rin (. inventory :rings)
        res []]
    (for [i 1 (# weap)]
      (for [j 1 (# arm)]
        (for [k 0 (# rin)]
          (for [l 0 (# rin)]
```

```
22              (when (or (not= k l) (= 0 k l))
23                (let [armor (+ (. (. arm j)
    ↪ :armor)
24                               (or (?. (?. rin
    ↪ k) :armor) 0)
25                               (or (?. (?. rin
    ↪ l) :armor) 0))
26                      damage (+ (. (. weap i)
    ↪ :damage)
27                                (or (?. (?. rin
    ↪ k) :damage) 0)
28                                (or (?. (?. rin
    ↪ l) :damage) 0))
29                      cost (+ (. (. weap i)
    ↪ :cost)
30                              (. (. arm j)
    ↪ :cost)
31                              (or (?. (?. rin k)
    ↪ :cost) 0)
32                              (or (?. (?. rin l)
    ↪ :cost) 0))
33                      newboss {:hits (. boss
    ↪ :hits)
34                               :damage (. boss
    ↪ :damage)
```

```
                                :armor (. boss
    ↪ :armor)
                                :cost 0}]
                  (table.insert
                   res
                   (score {:hits 100 :armor
    ↪ armor :damage damage :cost cost}
    ↪ newboss))))))))
    res))

(fn solve [boss inventory]
  (-> (game boss inventory play2win)
      (lume.filter #(< 0 (. $ :cost)))
      (lume.map #(. $ :cost))
      (aoc.table-min)))

(local weapons
       [{:name "dagger" :cost 8 :damage 4
    ↪ :armor 0}
        {:name "shortsword" :cost 10 :damage 5
    ↪ :armor 0}
        {:name "warhammer" :cost 25 :damage 6
    ↪ :armor 0}
        {:name "longsword" :cost 40 :damage 7
    ↪ :armor 0}
```

```
53          {:name "greataxe" :cost 74 :damage 8
    ↪  :armor 0}])
54
55  (local armor
56         [{:name "leather" :cost 13 :damage 0
    ↪  :armor 1}
57          {:name "chainmail" :cost 31 :damage 0
    ↪  :armor 2}
58          {:name "splintmail" :cost 53 :damage 0
    ↪  :armor 3}
59          {:name "bandedmail" :cost 75 :damage 0
    ↪  :armor 4}
60          {:name "platemail" :cost 102 :damage 0
    ↪  :armor 5}])
61
62  (local rings
63         [{:name "Damage +1" :cost 25 :damage 1
    ↪  :armor 0}
64          {:name "Damage +2" :cost 50 :damage 2
    ↪  :armor 0}
65          {:name "Damage +3" :cost 100 :damage 3
    ↪  :armor 0}
66          {:name "Defense +1" :cost 20 :damage 0
    ↪  :armor 1}
67          {:name "Defense +2" :cost 40 :damage 0
    ↪  :armor 2}
```

```
68          {:name "Defense +3" :cost 80 :damage 0
    ↪  :armor 3}])
69
70  (solve {:hits 103 :damage 9 :armor 2 :cost 0}
71         {:weapons weapons :armor armor :rings
    ↪  rings})
```

```
121
```

DONE Day 21.2

Turns out the shopkeeper is working with the boss, and can persuade you to buy whatever items he wants. The other rules still apply, and he still only has one of each item.

What is the most amount of gold you can spend and still lose the fight?

```
1  (fn play2loose [attacker defender]
2    (if (<= (. attacker :hits) 0)
3        attacker
4        (let [damage (- (. attacker :damage)
5                        (. defender :armor))]
6          (tset defender :hits (- (. defender
    ↪  :hits)
```

```
                                    (if (<= damage
 ↪  0) 1 damage)))
        (play2loose defender attacker))))

(fn solve2 [boss inventory]
  (-> (game boss inventory play2loose)
      (lume.filter #(< 0 (. $ :cost)))
      (lume.map #(. $ :cost))
      (aoc.table-max)))

(solve2 {:hits 103 :damage 9 :armor 2 :cost 0}
        {:weapons weapons :armor armor :rings
 ↪  rings})
```

```
201
```

DONE Day 23.1

Little Jane Marie just got her very first computer for Christmas from some unknown benefactor. It comes with instructions and an example program, but the computer itself seems to be malfunctioning. She's curious what the program does, and would like you to help her run it.

The manual explains that the computer supports two registers

and six instructions (truly, it goes on to remind the reader, a state-of-the-art technology). The registers are named a and b, can hold any non-negative integer, and begin with a value of 0. The instructions are as follows:

- hlf r sets register r to half its current value, then continues with the next instruction.
- tpl r sets register r to triple its current value, then continues with the next instruction.
- inc r increments register r, adding 1 to it, then continues with the next instruction.
- jmp offset is a jump; it continues with the instruction offset away relative to itself.
- jie r, offset is like jmp, but only jumps if register r is even ("jump if even").
- jio r, offset is like jmp, but only jumps if register r is 1 ("jump if one", not odd).

All three jump instructions work with an offset relative to that instruction. The offset is always written with a prefix + or - to indicate the direction of the jump (forward or backward, respectively). For example, jmp +1 would simply continue with the next instruction, while jmp +0 would continuously jump back to itself forever.

The program exits when it tries to run an instruction beyond the ones defined.

For example, this program sets a to 2, because the jio instruction causes it to skip the tpl instruction:

```
inc a
jio a, +2
tpl a
inc a
```

What is the value in register b when the program in your puzzle input is finished executing?

```
(local aoc (require :lib.aoc))

(fn hlf [xs x]
  (tset xs x (math.floor (/ (. xs x) 2)))
  1)

(fn tpl [xs x]
  (tset xs x (* 3 (. xs x)))
  1)

(fn inc [xs x]
  (tset xs x (+ 1 (. xs x)))
  1)
```

```
14
15 (fn jmp [offset]
16   (tonumber offset))
17
18 (fn jie [xs x offset]
19   (if (= 0 (% (. xs x) 2))
20     (jmp offset)
21     1))
22
23 (fn jio [xs x offset]
24   (if (= 1 (. xs x))
25     (jmp offset)
26     1))
27
28 (fn eval [line xs]
29   (case (aoc.string-split line " ")
30     ["hlf" r] (hlf xs r)
31     ["inc" r] (inc xs r)
32     ["jie" re of] (jie xs (string.sub re 1 -2)
    ↪  of)
33     ["jio" re of] (jio xs (string.sub re 1 -2)
    ↪  of)
34     ["jmp" of] (jmp of)
35     ["tpl" r] (tpl xs r)
36     _ (print (.. "No match for " line))))
```

```
37
38 (fn solve [input res reg]
39   (let [count (# input)]
40     (var cur 1)
41     (while (<= cur count)
42       (set cur (+ cur (eval (. input cur)
   ↪  res))))
43     (. res reg)))
44
45 (fn test [expected input]
46   (assert (= expected (solve input {:a 0 :b 0}
   ↪  :a))))
47
48 (test 2 ["inc a" "jio a, +2" "tpl a" "inc a"])
49
50 (solve (aoc.string-from "2015/23.inp") {:a 0
   ↪  :b 0} :b)
```

```
184
```

DONE Day 23.2

The unknown benefactor is very thankful for releasi– er, helping little Jane Marie with her computer. Definitely not to distract you, what is the value in register b after the program is finished

executing if register a starts as 1 instead?

```
(solve (aoc.string-from "2015/23.inp") {:a 1
 ↪  :b 0} :b)
```

```
231
```

DONE Day 25.1

Merry Christmas! Santa is booting up his weather machine; looks like you might get a white Christmas after all.

The weather machine beeps! On the console of the machine is a copy protection message asking you to enter a code from the instruction manual. Apparently, it refuses to run unless you give it that code. No problem; you'll just look up the code in the–

"Ho ho ho", Santa ponders aloud. "I can't seem to find the manual."

You look up the support number for the manufacturer and give them a call. Good thing, too - that 49th star wasn't going to earn itself.

"Oh, that machine is quite old!", they tell you. "That model went out of support six minutes ago, and we just finished shredding

all of the manuals. I bet we can find you the code generation algorithm, though."

After putting you on hold for twenty minutes (your call is very important to them, it reminded you repeatedly), they finally find an engineer that remembers how the code system works.

The codes are printed on an infinite sheet of paper, starting in the top-left corner. The codes are filled in by diagonals: starting with the first row with an empty first box, the codes are filled in diagonally up and to the right. This process repeats until the infinite paper is covered. So, the first few codes are filled in in this order:

```
   | 1   2   3   4   5   6
---+---+---+---+---+---+---+
 1 |  1   3   6  10  15  21
 2 |  2   5   9  14  20
 3 |  4   8  13  19
 4 |  7  12  18
 5 | 11  17
 6 | 16
```

For example, the 12th code would be written to row 4, column 2; the 15th code would be written to row 1, column 5.

The voice on the other end of the phone continues with how the codes are actually generated. The first code is 20151125. After that, each code is generated by taking the previous one, multiplying it by 252533, and then keeping the remainder from dividing that value by 33554393.

So, to find the second code (which ends up in row 2, column 1), start with the previous value, 20151125. Multiply it by 252533 to get 5088824049625. Then, divide that by 33554393, which leaves a remainder of 31916031. That remainder is the second code.

"Oh!", says the voice. "It looks like we missed a scrap from one of the manuals. Let me read it to you." You write down his numbers:

```
   |    1         2         3         4
↪  5         6
---+---------+---------+---------+---------+---------+---------+-----
 1 | 20151125  18749137  17289845  30943339
↪  10071777  33511524
 2 | 31916031  21629792  16929656   7726640
↪  15514188   4041754
 3 | 16080970   8057251   1601130   7981243
↪  11661866  16474243
```

```
 4 | 24592653  32451966  21345942   9380097
↪  10600672  31527494
 5 |    77061  17552253  28094349   6899651
↪  9250759  31663883
 6 | 33071741   6796745  25397450  24659492
↪  1534922  27995004
```

”Now remember”, the voice continues, ”that's not even all of the first few numbers; for example, you're missing the one at 7,1 that would come before 6,2. But, it should be enough to let your– oh, it's time for lunch! Bye!” The call disconnects.

Santa looks nervous. Your puzzle input contains the message on the machine's console. What code do you give the machine?

```
(fn nth-code [n]
  (var res 20151125)
  (for [i 2 n]
    (set res (% (* 252533 res) 33554393)))
  res)

(fn ind2ord [x y]
  (let [xy (+ x y -1)]
    (+ y (math.floor (/ (* xy (- xy 1)) 2)))))

```

```
(fn xy-code [x y]
  (let [xy (ind2ord x y)]
    (nth-code xy)))

(fn solve [x y]
  (xy-code x y))

(solve 2981 3075)
```

```
9132360
```