Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 11, 2024

Contents

Finding the Chief Historian! This program contains my Advent of Code solutions for 2024, which you can find on my sourcehut and GitHub. I believe GitHub doesn't show the results of code blocks, which means that viewing it there might leave you a bit confused.

In general, I've added line numbers to code blocks when that code block is part of the solution file. In some cases, I've added some code that explains, clarifies, justifies or otherifies something. Those lines aren't numbered if they aren't necessary to the final solution.

Utils

I define some common functions in `aoc.util`, mostly related to parsing the input. The input always comes in a file but also usually has an example input. The former is a file (which we read as a string with `slurp`) and the latter is just a string in the same format. Therefore it's easiest to let the days itself take care of reading the file (since they also have the example input), and just operate on strings here.

```clojure
1  (ns aoc.util
2    (:require [clojure.string :as str]))
```

Files

```clojure
3  (defn string-as-lines
4    "Outputs the string as a vector, one element per line."
5    [input]
6    (str/split input #"\n"))
```

Often the lines contain numbers:

```clojure
7  (defn string-as-numbers-per-line
8    "Assumes there is a number on each line: we parse it and return a
9    vector, one element per line. Technically each number is parsed with
10   `read-string`, so it isn't just limited to numbers, but I've only
11   tested numbers."
12   [input]
13   (->> (string-as-lines input)
14        (map read-string)))
```

or lists of numbers. This was the case in both Day 1 and Day 2, where the input had the following format:

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

Here, we want the whole file to be represented by a vector, where each element is itself a vector of the space-separated numbers on a line.

```clojure
15  (defn num-list-per-line
16    "Returns a vector of vectors, the outer vector has an element per
17    line, the inner has space-separated elements. "
```

```
18    [input]
19    (->> (string-as-lines input)
20         (map #(mapv read-string (str/split % #"\s+")))
21         vec))
```

This function, run on the example table seen above, will return the following:

```
(num-list-per-line example)
```

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

Which is fine, but sometimes (see Day 10) you need to parse numbers without any space in between them. In that case, just map over the characters and call (Character/digit c 10) on it:

```
22    (defn adjacent-num-list-per-line
23      "Returns a vector of vectors. The outer vector has an element per
24      line, the inner has an element per character, parsed as int"
25      [input]
26      (->> (string-as-lines input)
27           (map #(mapv (fn [ch] (Character/digit ch 10)) %))
28           vec))
```

```
(adjacent-num-list-per-line example)
```

Grids

First, the directions you can move in a grid:

```
29    (def cardinal-directions
30      [[-1  0]    ; Up
31       [ 1  0]    ; Down
32       [ 0 -1]    ; Left
33       [ 0  1]])  ; Right
34
35    (def diagonal-directions
36      [[-1 -1]    ; NW
37       [-1  1]    ; NR
38       [ 1 -1]    ; SW
39       [ 1  1]])  ; SE
40
41    (def all-directions
42      (concat cardinal-directions diagonal-directions))
```

And how you can use them:

```
43  (defn move [[x y] [x' y']]
44    [(+ x x') (+ y y')])

    (move [4 2] (first cardinal-directions))

    [3 2]
```

And finding characters in a grid:

```
45  (defn char-locations [grid x]
46    (mapcat (fn [row string]
47              (keep-indexed (fn [col char] (when (= char x) [row col]))
48                            string))
49            (range)
50            grid))
```

For example:

```
(def input "MMMSXXMASM
MSAMXMSMSA
AMXSXMAAMM
MSAMASMSMX
XMASAMXAMM
XXAMMXXAMA
SMSMSASXSS
SAXAMASAAA
MAMMMXMMMM
MXMXAXMASX")

(def grid (string-as-lines input))

(take 10 (char-locations grid \X))

([0 4] [0 5] [1 4] [2 2] [2 4] [3 9] [4 0] [4 6] [5 0] [5 1])
```

1 Day 1

1.1 Part 1

We need to reconcile two lists. We get them in the following form:

```
3   4
4   3
2   5
1   3
3   9
3   3
```

And our goal is to find the "distance" between the two lists.

> To find the total distance between the left list and the right list,
> add up the distances between all of the [sorted] pairs you found.

For the example above, the correct answer is **11**.

My strategy is: convert the input to pairs of numbers (`aoc.util/num-list-per-line` takes care of this), transpose them (so we have two lists), sort them, transpose them again (pairs), and take the difference and sum it. Makes sense? We need the two tiny helper functions `sum` and `transpose`:

```clojure
1  (ns aoc.1)

2  (defn sum "Finds the sum of a vector of numbers" [vec]
3    (reduce + vec))

4
5  (defn transpose "Transposes a matrix" [m]
6    (apply mapv vector m))

7
8  (defn p1 [input]
9    (let [input (aoc.util/num-list-per-line input)]
10     (->> input
11          (transpose)
12          (map sort)
13          (transpose)
14          (map #(abs (- (first %) (second %))))
15          (sum))))
```

It works for the testinput, fantastic. Now let's open the file and run it on the input. The input file for day 1 can be found in the file **inputs/1**.

```clojure
16  (assert (= 11 (p1 testinput)))
17  (def input (slurp "inputs/1"))
18  (p1 input)
```

2057374

Hurrah! We get a **Gold Star**!

1.2   Part 2

Now, we need to find a "similarity score" for the two lists:

> Calculate a total similarity score by adding up each number in the
> left list after multiplying it by the number of times that number
> appears in the right list.

A naive way to do this would be to iterate over the first list, where, for
each element, we count how many items in the second list are equal to that
element, and multiply the element with the count. However, you'd be doing
a lot of duplicate counting. A faster way to do it is to convert the second (it
doesn't really matter which one you pick) list to a map once, with `{element
frequency}`. Let's use the function `frequencies`!

```
(frequencies (last (transpose (aoc.util/num-list-per-line testinput))))
```

```
{4 1, 3 3, 5 1, 9 1}
```

Now, we can iterate over the first list (which we get by `(transpose
(numbers input))`), multiply the element itself by the count in `frequencies`,
and sum the result.

```
19  (defn p2 [input]
20    (let [input (transpose (aoc.util/num-list-per-line input))
21          one (first input)
22          freqs (frequencies (second input))]
23      (->> one
24          (map #(* % (freqs % 0)))
25          (sum))))
26
27  (assert (= 31 (p2 testinput)))
28  (p2 input)
```

23177084

2   Day 2

2.1   Part 1

Analysing some unusual data from a nuclear reactor. The data consists of *reports* separated by lines, each of which is a list of numbers (*levels*), separated by spaces.

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

We need to find out how many reports are **safe**, which is the case if all levels are gradually increasing or decreasing. This is defined as such:

> [A] report only counts as safe if both of the following are true:
>
> - The levels are either all increasing or all decreasing.
> - Any two adjacent levels differ by at least one and at most three.

In the example input, there are **2** safe reports—the first and last.

Let's convert all numbers to the difference between the previous number. Then, a report is safe is all numbers are of the same sign, and the absolute of the number is between 1 and 3.

Since we're computing the difference between each element and the element before, I want to use `partition`, which does exactly this. Then, we can use `mapv` to compute the difference. For the last element of the testinput:

```
1   (ns aoc.2)

2   (defn diffs [record]
3     (->> record
4          (partition 2 1)
5          (mapv (fn [[a b]] (- b a)))))
```

test it out:

```
(diffs (last (aoc.util/num-list-per-line testinput)))

[2 3 1 2]
```

Now just use that to determine whether a record is safe. Recall that the testinput had **2** safe records.

```
6   (defn is-safe? [record]
7     (let [differences (diffs record)]
8       (and (every? #(<= 1 (abs %) 3) differences)
9            (apply = (map pos? differences)))))
10
11  (defn p1 [input]
12    (->> (aoc.util/num-list-per-line input)
13         (filter is-safe?)
14         (count)))
15
16  (p1 testinput)

    2


17  (def input (slurp "inputs/2"))
18  (p1 input)

    242
```

2.2   Part 2

> Now, the same rules apply as before, except if removing a single
> level from an unsafe report would make it safe, the report instead
> counts as safe.

First I had a smart idea. Check out e2dcab2f0de76c21477c5e871e029f0282c8fabc.
It is much more efficient than the current solution, but much more convo-
luted and ugly to read. Right now, I just remove each level one by one and
check if the record is safe then.

```
19  (defn drop-nth [coll n]
20    (keep-indexed #(if (not= %1 n) %2) coll))
21
22  (defn dampened-is-safe? [record]
23    (some is-safe? (map #(drop-nth record %)
24                        (range (count record)))))
25
26  (defn p2 [input]
27    (->> (aoc.util/num-list-per-line input)
28         (filter dampened-is-safe?)
```

```
29          (count)))
30
31  (p2 testinput)

    4


32  (p2 input)

    311
```

2.2.1 Benchmark results

The old solution took on average `3.8` milliseconds to execute `(p2 input)`, and the new solution about `6.0`. This is worth it, IMO, since the code is *much* simpler. Next time, first do the easy thing, and then benchmark to see if it needs to be improved!

3 Day 3

3.1 Part 1

We have an input string that contains a lot of characters, for example:

```
xmul(2,4)%&mul[3,7]!@^do_not_mul(5,5)+mul(32,64]then(mul(11,8)mul(8,5))
```

The goal is to extract all substrings that are of the exact form `mul(\d+,\d+)`, and in that case multiply the two numbers together. This is straightforward, I'm not really going to create any helper functions: parse with regex, convert to int, multiply and sum.

```clojure
(ns aoc.3)

(defn p1 [input]
  (let [matches (re-seq #"mul\((\d+),(\d+)\)" input)]
    (->> matches
         (map #(list (Integer/parseInt (nth % 1)) (Integer/parseInt (nth % 2))))
         (map #(apply * %))
         (reduce +))))

(let [input (slurp "inputs/3")]
  (p1 input))

155955228
```

3.2 Part 2

We get a new example string for Part Two:

```
xmul(2,4)&mul[3,7]!^don't()_mul(5,5)+mul(32,64](mul(11,8)undo()?mul(8,5))
```

This contains the substrings `don't()` and `do()`, which disable and enable `mul()` instructions. I can do fancy clojure things, but Emacs is way too good for this, so let's do it quickly in Elisp. We want to remove everything from the input file that's in between a `don't()` and a `do()` instruction, and then call `(p1)` on this input. There are three slightly tricky things about this:

- The input file has some newlines, and in some cases a `do()` instruction is on a later line than the previous `don't()` instruction.

- You need to match non-greedy in between a `don't()` and a `do()`.

- If you call `(replace-regexp)` with just the regex and replacement string, it will move point to the last match. This is easily fixed by adding the fourth and fifth arguments to `replace-regexp`: START and END.

11

So, here's some elisp code that does that.

```elisp
(with-temp-buffer
  (insert-file-contents "inputs/3")
  (replace-regexp "\n" "" nil (point-min) (point-max))
  (replace-regexp "don't().+?do()" "" nil (point-min) (point-max))
  (write-region (point-min) (point-max) "inputs/3-enabled"))
```

And back to clojure for the now trivial second part.

```clojure
(let [fixed-input (slurp "inputs/3-enabled")]
  (p1 fixed-input))
```

100189366

4 Day 4

4.1 Part 1

We need to find all instances of `XMAS`, appearing in a text like below, either horizontally, vertically, or diagonally, including written backwards. According to these rules, the example below contains **18 `XMAS`**-es.

```
MMMSXXMASM
MSAMXMSMSA
AMXSXMAAMM
MSAMASMSMX
XMASAMXAMM
XXAMMXXAMA
SMSMSASXSS
SAXAMASAAA
MAMMMXMMMM
MXMXAXMASX
```

My idea is to search on the letter `X` and use each `X` as a starting point, where we count `XMAS` occurrences in each of the 8 different directions. Let's create a function `count-xmases-at` that counts the number of `XMAS`-es starting from a location. Then simply call that for each `X` found in the grid and sum.

We're going to need to define two helper functions:

`char-locations` Returning all the locations of given character in the grid;

`is-xmas?` A function that takes the `grid`, a `start` coordinate and a `direction`. It returns `true` if `"XMAS"` occurs in the `grid` from `start` in the given `direction`.

Since we're working in a grid, let's make `char-locations` return a 2d vector like `[0 0]` to denote the coordinates in the grid. While we're at it, let's define those directions like so:

```clojure
1  (ns aoc.4)

2  (def directions
3    [[-1  0]    ; Up
4     [ 1  0]    ; Down
5     [ 0 -1]    ; Left
6     [ 0  1]    ; Right
7     [-1 -1]    ; NW
8     [-1  1]    ; NR
9     [ 1 -1]    ; SW
10    [ 1  1]])  ; SE
```

Which is a nice format to have, because you can then do something like this:

```clojure
(defn move [start direction]
  (let [[x y] start
        [x' y'] direction]
    [(+ x x') (+ y y')]))

(let [start [4 5]
      direction [-1 0]] ;; up
  (move start direction))

[3 5]
```

Which is really nice! So, let's define the final functions necessary for Part One:

```clojure
11  (defn char-locations [grid x]
12    (mapcat
13     (fn [row string]
14       (keep-indexed (fn [col char] (when (= char x) [row col]))
15                     string))
16     (range)
17     grid))
18
19  (defn is-xmas? "Does the grid `grid` contain the string \"XMAS\",
20   starting at `start` and going in `direction`?"
21    [grid start direction]
22    (let [[startx starty] start
23          [x' y'] direction]
24      (loop [x startx
25             y starty
26             chars (seq "XMAS")]
27        (if (empty? chars)
28          true
29          (if (not= (get-in grid [x y]) (first chars))
30            false
31            (recur (+ x x')
32                   (+ y y')
33                   (rest chars)))))))
```

Now we can tie everything together. `is-xmas?` returns true if the grid contains the word `"XMAS"` in a given `direction`. After we've found all `X` characters, we can count the amount of `XMAS`-es connected to it by counting all `direction` for which `is-xmas?` returns true.

```
34  (defn count-xmases-at [grid start directions]
35    (count (filter #(is-xmas? grid start %) directions)))
36
37  (defn p1 [input]
38    (let [grid (aoc.util/string-as-lines input)
39          xs (char-locations grid \X)]
40      (->> xs
41           (map #(count-xmases-at grid % directions))
42           (reduce +))))
43
44  (assert (= 18 (p1 example)))
45  (def input (slurp "inputs/4"))
46  (p1 input)
```

    2447

4.2 Part 2

Ah, it seems the Elf thinks we're idiots because they use letters more literally.
We don't need to find the string XMAS, we need to find the string MAS in an
X, like so!

```
M.S
.A.
M.S
```

We could have reused the functionality above to search for MAS-es, and
then only count a MAS that has a nice diagonal partner sharing the A. How-
ever, I found that a bit tricky to reason about, so I've opted to search for
all of the A-s in the text, and finding MAS strings diagonally from that A. If
there are exactly two MAS-es, we know that we got an X-MAS.

Instead of is-xmas?, we now have is-mas?, checking from a middle A
instead of a starting X. Note that we're only counting **X**-MAS-es, so only use
diagonals:

```
47  (def diagonal-directions
48    [[-1 -1]   ; NW
49     [-1  1]   ; NE
50     [ 1 -1]   ; SW
51     [ 1  1]]) ; SE
```

is-mas? is now pretty trivial:

```
52  (defn is-mas? [grid middle direction]
53    (let [[x y] middle
```

```
54          [x' y'] direction]
55       (and (= \M (get-in grid [(+ x x') (+ y y')]))
56            (= \S (get-in grid [(- x x') (- y y')]))))))
```

And `count-mases-at` is virtually identical to `count-xmases-at` from Part One.

```
57  (defn count-mases-at [grid middle directions]
58    (count (filter #(is-mas? grid middle %) directions)))
59
60  (defn p2 [input]
61    (let [grid (aoc.util/string-as-lines input)
62          as (char-locations grid \A)]
63      (->> as
64           (map #(count-mases-at grid % diagonal-directions))
65           (filter #(= % 2))
66           (count))))
67
68  (assert (= 9 (p2 example)))
69  (p2 input)
```

1868

5 Day 5

5.1 Part 1

Graphs! We get an input file that looks like this:

```
47|53
97|13
97|61
97|47
75|29
61|13
75|53
29|13
97|29
53|29
61|53
97|53
61|29
47|13
75|47
97|75
47|61
75|61
47|29
75|13
53|13

75,47,61,53,29
97,61,53,29,13
75,29,13
75,97,47,61,53
61,13,29
97,13,75,29,47
```

The first part contains required orderings, where 29|13 means that 29 should always come before 13. The second part contains "updates" that might or might not be correctly sorted. In Part One, we need to take the correctly sorted updates, take the middle number, and sum those. I wonder what the second part will be? Actually, I don't wonder, I'm virtually certain of it so I'm just going to sort them already. If the update is equal to the sorted input, it's sorted and we can solve Part One.

I already alluded to graphs, that's because you can think of this as a ~~DAG~~ Directed Graph. In the case before, 29|13 will lead to a vertex from 29 to 13. My "graph" will basically be a list of dependencies, but I'll call it a

graph because that's cool and it sort of is one. Before we get into the weeds, let's zoom out and think of what we need: the sum of the middle numbers of the sorted updates.

First look at the easy functions, leaving `sort` and `build-dependency-graph` empty for the time being:

```clojure
1  (ns aoc.5
2    (:require [clojure.string :as str]))
3
4  (defn sort [dependency-graph update])
5  (defn build-dependency-graph [orderings])
6
7  (defn sorted? [dependency-graph update]
8    (= update (sort dependency-graph update)))
9
10 (defn middle-num
11   "Finds the middle string in a list of string, and parses it to a
12   number. Assumes the length of the list list is odd."
13   [update]
14   (read-string (nth update (/ (count update) 2))))
```

Now we can write `p1`. Since I expect to need the orderings, updates and dependency-graph later as well, I'll create a small function `parse-input` that extracts these from the puzzle input.

```clojure
15 (defn parse-input
16   "Parses an input string and returns three useful objects.
17   The first obj is a list of orderings, strings of type \"A|B\".
18   The second obj is a list of updates, each one a list of strings.
19   The third obj is a dependency graph, a map."
20   [input]
21   (let [[orderings updates] (str/split input #"\n\n")
22         orderings (str/split orderings #"\n")
23         updates (str/split updates #"\n")
24         updates (map #(str/split % #",") updates)
25         dependency-graph (build-dependency-graph orderings)]
26     [orderings updates dependency-graph]))
27
28 (defn p1 [input]
29   (let [[orderings updates dep-graph] (parse-input input)
30         sorted? (partial sorted? dep-graph)]
31     (->> updates
32          (filter sorted?)
33          (map middle-num)
34          (reduce +))))
```

Hmm, yes, extremely reasonable, but we haven't yet filled in `build-dependency-graph` and `sort`. `build-dependency-graph` should take as input the `orderings` (a list of strings from the input, separated by |), and return a map of the following form:

```
{"75" ["97"], "13" ["97" "61" "29" "47" "75" "53"], ...}.
```

To do so, I'll first create a hash-map of the following form:

```
{"75" ["97"], "13" ["97"], "13" ["61"], ...},
```

and then merge identical keys with `merge-with` and `into`, creating our desired dependency graph.

```
35  (defn build-dependency-graph
36    [orderings]
37    (let [order-pairs (->> orderings
38                           (map #(str/split % #"\|"))
39                           (map #(hash-map (second %), [(first %)])))]
40      (apply (partial merge-with into) order-pairs)))
```

Verifying that this next result is correct is left as an exercise for the reader, but let's test it out on the example input:

```
(let [[orderings _ _] (parse-input example)]
  (build-dependency-graph orderings))

{"61" ["97" "47" "75"],
 "47" ["97" "75"],
 "53" ["47" "75" "61" "97"],
 "13" ["97" "61" "29" "47" "75" "53"],
 "75" ["97"],
 "29" ["75" "97" "53" "61" "47"]}
```

And now, ladies and gentleman, the moment you've all been waiting for, `sort`! We need to sort an `update` based on a `dependency-graph`. You can see it below, but how it works:

1. It creates a `graph`: a subset of `dep-graph`, *limited to the items local to the current* `update`. It starts with an empty map {}, and then for each `item` in `update`, adds the elements in the `dependency-graph` that depend on `item`. `graph` ends up as a map with key a number, and value a set of the dependencies.

   Limiting the dependency graph to be local only to the current `update` gives us a tremendous advantage: we can sort the items based on the number of dependencies each item has.

19

2. Sort the items in `update` by their amount of dependencies.

```
41  (defn sort
42    "Sort a list of strings based on a dependency map.
43     The map defines which elements should come after others."
44    [dep-graph update]
45    (let [graph (reduce (fn [acc item]
46                          (assoc acc item
47                            (set (get dep-graph item []))))
48                        {} update)
49          local-deps (fn [deps] (filter #(contains? (set update) %) deps))]
50      (vec (sort-by (fn [item]
51                      (let [deps (get dep-graph item [])]
52                        (count (local-deps deps))))
53                    update)))))
```

Now we got everything, ain't we?

```
54  (assert (= 143 (p1 example)))
55  (def input (slurp "inputs/5"))
56  (p1 input)
```

```
4637
```

yes

5.2   Part 2

Surprise surprise, we need to sort the incorrect updates! We need to take the sum of the middle numbers of only the *incorrect* updates. Our prescience is immeasurable.

```
57  (defn p2 [input]
58    (let [[orderings updates deps] (parse-input input)
59          is-sorted? (partial sorted? deps)
60          sort (partial sort deps)]
61      (->> updates
62           (filter #(not (is-sorted? %)))
63           (map sort)
64           (map middle-num)
65           (reduce +))))
```

```
66  (assert (= 123 (p2 example)))
67  (p2 input)
```

```
6370
```

6 Day 6

6.1 Part 1

We get a grid again, now representing a map. It looks like this:

```
....#.....
.........#
..........
..#.......
.......#..
..........
.#..^.....
........#.
#.........
......#...
```

The `^` represents the starting location of our guard, and they start by going *up*. A `#` is an obstacle, and will force the guard to move direction, turning 90° clockwise. Our goal is to find out how many distinct places the guard has entered by the time he leaves the puzzle.

If you replace entered places by `X`, you'd get the following output, with **41** distinct places:

```
....#.....
....XXXXX#
....X...X.
..#.X...X.
..XXXXX#X.
..X.X.X.X.
.#XXXXXXX.
.XXXXXXX#.
#XXXXXXX..
......#X..
```

Turning clockwise means that we have only four directions:

```clojure
(ns aoc.6)

(def directions
  [[-1  0]    ; Up
   [ 0  1]    ; Right
   [ 1  0]    ; Down
   [ 0 -1]])  ; Left
```

Our function will simply compute the route the guard takes as a vector of coordinates, and count the distinct elements of said vector:

```
7   (defn guard-route
8     "Takes a `grid` as input returns a vector of 2d coordinates: the route
9     of the guard, starting at `start` and turning clockwise at \"#\"
10    characters. "
11    [grid start])
12
13  (defn p1 [input]
14    (let [grid (aoc.util/string-as-lines input)
15          start (first (aoc.4/char-locations grid \^))
16          route (guard-route grid start)]
17      (count (distinct route))))
```

As for `guard-route`, we loop through the grid, where each iteration of the loop is a move: go to the next location given some direction, or change direction, building a `route` along the way. We replace the ^ character with a . after determining the start so that we only have two cases to deal with, . and #. We can reuse the `char-locations` formula from Day 4 (which gives us a list of coordinates where a certain character can be found) to find our starting location.

```
18  (defn replace-char
19    [grid [x y] new-char]
20    (update grid x
21            #(str (subs % 0 y)
22                  new-char
23                  (subs % (inc y)))))
24
25  (defn guard-route [grid start]
26    (let [size (count grid)
27          grid (replace-char grid start \.)]
28      (loop [location start
29             directions (cycle directions)
30             route []]
31        (let [[x y] location
32              [x' y'] (first directions)
33              next-location [(+ x x') (+ y y')]
34              next-object (get-in grid next-location)
35              route (conj route location)]
36          (condp = next-object
37            nil route
38            \. (recur next-location
39                      directions
40                      route)
41            \# (recur location
```

22

```
42                    (next directions)
43                    route))))))
```

Perhaps this is a little too imperative, but I'm fine with it.

```
44  (assert (= 41 (p1 example)))
45  (def input (slurp "inputs/6"))
46  (p1 input)
```

5208

6.2   Part 2

It's of course possible that the guard enters a loop, but fortunately that didn't occur in the input we were given. Part Two is concerned with *creating* loops by adding obstacles. Specifically, *how many loops can we create by adding just a single obstacle?*

I'm afraid that I'll have to create a very similar function to `guard-route`, except that now we keep track of the places we've been before. If we ever enter the same location while going in the same direction, we know we've entered a loop and can exit immediately. In that case, let's return `true` and name the function `route-has-loop?`. Since we're exiting earlier and I don't want to create cycle-detection, I'm not reusing the function from Part One. In python I'd use a generator, but I haven't figured out `lazy-seq` yet in clojure.

I can't think of a way to do this intelligently, but at least one insight is that you don't have to consider *all* cases: you only have to add obstacles on parts of the original route; adding them elsewhere will have no effect.

```
47  (defn route-has-loop? [grid start])
48
49  (defn p2 [input]
50    (let [grid (aoc.util/string-as-lines input)
51          start (first (aoc.4/char-locations grid \^))
52          route (disj (set (guard-route grid start)) start)]
53      (->> route
54          (pmap (fn [new-obstacle]
55                  (route-has-loop? (replace-char grid new-obstacle \#) start)))
56          (filter true?)
57          (count))))
```

`route-has-loop?` is virtually identical to `guard-route`, except that we keep track of the `visited` set (keeping track of visited `[location direction]` pairs), and that we return `true` or `false` instead of the route.

```clojure
58  (defn route-has-loop? [grid start]
59    (let [size (count grid)
60          grid (replace-char grid start \.)]
61      (loop [location start
62             directions (cycle directions)
63             visited #{}]
64        (let [[x y] location
65              [x' y'] (first directions)
66              next-location [(+ x x') (+ y y')]
67              next-object (get-in grid next-location)
68              pair [next-location [x' y']]]
69          (if (contains? visited pair)
70            true ;; we have a loop!
71            (condp = next-object
72              nil false ;; we exited the puzzle
73              \. (recur next-location
74                        directions
75                        (conj visited pair))
76              \# (recur location
77                        (next directions)
78                        (conj visited pair)))))))))
```

On my laptop, this takes about 15 seconds to run on a single thread, but by default uses all of the threads (just by changing `map` into `pmap`, how freaking awesome is that!)

```clojure
79  (assert (= 6 (p2 example)))
80  (p2 input)

    1972
```

7 Day 7

7.1 Part 1

The elephants stole our operators! We had a list of equations, but they stole the operators between the numbers. We get an input where each line represents a single equation, which may be correct. We have to determine whether the equation can be correct, if we limit ourselves to + and *. In this example:

```
190: 10 19
3267: 81 40 27
83: 17 5
156: 15 6
7290: 6 8 6 15
161011: 16 10 13
192: 17 8 14
21037: 9 7 18 13
292: 11 6 16 20
```

only three of the equations can be made true, and their results sum up to **3749**—that is our goal.

```clojure
1  (ns aoc.7 (:require [clojure.string :as str]))
```

Again we get a familiar pattern: `map`, `filter`, `reduce`.

```clojure
2  (defn is-correct? [equation])
3
4  (defn parse-equations [input]
5    (let [lines (str/split input #"\n")
6          equations (map #(str/split % #": ") lines)]
7      (map (fn [[lhs rhs]]
8             [(read-string lhs) (vec (map read-string (str/split rhs #" ")))])
9           equations)))
10
11  (defn p1 [input]
12    (->> input
13         (parse-equations)
14         (filter is-correct?)
15         (map first)
16         (reduce +)))

(parse-equations example)
```

```
([190 [10 19]]
 [3267 [81 40 27]]
 [83 [17 5]]
 [156 [15 6]]
 [7290 [6 8 6 15]]
 [161011 [16 10 13]]
 [192 [17 8 14]]
 [21037 [9 7 18 13]]
 [292 [11 6 16 20]])
```

Now the banger `is-correct?`. There are ~800 equations, the longest one
has 12 numbers to add or multiply, so 2048 possible operations to check out.
I think brute-forcing is pretty viable.

```
17  (defn possible-ops
18    [x y]
19    [(* x y)
20     (+ x y)])
21
22  (defn equation-possibilities
23    [target nums]
24    (->> (range 1 (count nums))
25         (reduce (fn [possible-results idx]
26                    (->> possible-results
27                         (mapcat (fn [result]
28                                     (possible-ops result (nth nums idx))))))
29                 [(first nums)])))
30
31  (defn is-correct? [equation]
32    (let [[result numbers] equation
33          targets (equation-possibilities result numbers)]
34      (some #(= % result) targets)))
35  (assert (= 3749 (p1 example)))
36  (def input (slurp "inputs/7"))
37  (p1 input)

    12839601725877
```

7.2   Part 2

Now this is an elegant Part Two.

```
38  (defn possible-ops
39    [x y]
40    [(* x y)
41     (+ x y)
42     (Long/parseLong (str x y))])
43
44  (assert (= 11387 (p1 example)))
45  (p1 input)

    149956401519484
```

7.3 Optimization

There is a nice way to optimize this. Since this one actually takes quite long (Part One takes about 200ms, Part Two about 6.2s), I might end up doing this at some time, but the trick is that you don't need to multiply the last two numbers together if the equation target isn't divisible by the last number. That frees up half of the possible combinations, and you can of course do this for the second-to-last number as well, et cetera. It's probably nice to reverse the operation list for this.

For Part Two, you can optimize the || operation by skipping it if the target number doesn't have the final number as suffix.

8 Day 8

8.1 Part 1

We get a grid that looks more or less like this:

```
............
........0...
.....0......
.......0....
....0.......
......A.....
............
............
........A...
.........A..
............
............
```

And we need to find the specific *antinode*. An *antinode* is defined as

an antinode occurs at any point that is perfectly in line with two antennas of the same frequency - but only when one of the antennas is twice as far away as the other.

For the example above, there are **14** unique antinodes within the bounds of the map. The pseudo-function (correct with some good imagination) is:

$$\text{antinodes}(a_1, a_2) = \text{distance}(a_1, a_2) \pm [a_1, a_2]. \tag{1}$$

You can see the antinodes marked by **#** here:

```
......#....#
...#....0...
....#0....#.
..#....0....
....0....#..
.#....A.....
...#........
#......#....
........A...
.........A..
..........#.
..........#.
```

First, we need to identify all frequencies—all characters that aren't . or
\n. For each frequency, find all pairs of antennas that have said frequencies,
and find the antinodes. Put the locations in a set and count it.

```clojure
(ns aoc.8
  (:require [clojure.math.combinatorics :as combo]))

(defn all-antinodes-for-freq-in-grid [grid freq])

(defn p1 [input]
  (let [grid (aoc.util/string-as-lines input)
        freqs (->> input
                   set
                   (remove #{\. \newline}))]
    (->> freqs
         (mapcat #(all-antinodes-for-freq-in-grid grid %))
         distinct
         count)))
```

all-antinodes-for-freq-in-grid finds all antinodes that are valid within
the grid bounds. For that we need two small helper functions, all-antinodes-for-freq
(which finds all antinodes, possibly out of bounds), and in-grid?, whether
a coordinate is in a grid. In turn, all-antinodes-for-freq find all antin-
odes for all pairs, and uses the function antinodes-for-pair to find the
antinodes for a given pair.

```clojure
(defn- in-grid? [grid [x y]]
  (and (< -1 x (count grid))
       (< -1 y (count (first grid)))))

(defn antinodes-for-pair [[[ay ax] [by bx]]]
  (let [dx (abs (- ay by))
        dx-sign (compare by ay)
        dy (abs (- ax bx))
        dy-sign (compare bx ax)]
    [[(- ay (* dx-sign dx)) (- ax (* dy-sign dy))]
     [(+ by (* dx-sign dx)) (+ bx (* dy-sign dy))]]))
```

Okay that function was a bit ugly, but the weird behaviour with the sign
was so that you can simply add the difference to the correct parts and be
done with it. You see, if you have two locations a = [1 8] and b = [2 5],
you can compute the absolute difference ([1 3]), but you need to *subtract*
1 from ay (since a is above b), but *add* 3 to ax (since a is to the right of
b). This is the bit of imagination I requested from you above for Equation
1. Anyhow, this sign business takes care of that, such that:

```
(antinodes-for-pair [[1 8] [2 5]])

[[0 11] [3 2]]


26  (defn all-antinodes-for-freq [grid freq]
27    (let [indices (aoc.util/char-locations grid freq)
28          pairs (combo/combinations indices 2)]
29      (mapcat antinodes-for-pair pairs)))
30
31  (defn all-antinodes-for-freq-in-grid [grid freq]
32    (let [antinodes (all-antinodes-for-freq grid freq)]
33      (filter (partial in-grid? grid) antinodes)))

34  (assert (= 14 (p1 example)))
35  (def input (slurp "inputs/8"))
36  (p1 input)

371
```

8.2 Part 2

In Part Two we don't just find two antinodes (at equal distance from both points), but instead we find all antinodes that are in line with any given pair. In order to do this, we follow the same structure:

- find all antinodes

- filter those outside of the grid

- count distinct elements

But now the finding all antinodes is of course slightly different. If we redefine `all-antinodes-for-freq-in-grid` to return the new harmonic antinodes, we can keep `p1` identical.

```
37  (defn harmonic-antinodes-for-pair [grid [a b]])
38
39  (defn all-antinodes-for-freq-in-grid [grid freq]
40    (let [indices (aoc.util/char-locations grid freq)
41          pairs (combo/combinations indices 2)]
42      (->> pairs
43           (mapcat #(harmonic-antinodes-for-pair grid %))
44           (filter (partial in-grid? grid)))))
```

Alright now `harmonic-antinodes-for-pair` should find *all* harmonic antinodes for a pair, and we filter the ones that aren't in the grid. However, how should we do this? There's an infinite amount of 'em! There's surely some smart way to compute only the ones that are in the grid, but I'm doing a sort of dumb method:

- compute the difference between two pairs (for `[[1 8] [2 5]]` that's `[1 -3]`);

- subtracting that from `a (count grid)` times.

- adding that to `a (count grid)` times.

This way we know for sure that we don't miss anything, but we do know that a lot of what we calculate will fall outside of the grid.

```
45  (defn- dx-dy-pair [[ax ay] [bx by]]
46    (let [[dx dy] [(abs (- ax bx)) (abs (- ay by))]
47          [dx dy] [(* (compare bx ax) dx)
48                   (* (compare by ay) dy)]]
49      [dx dy]))
50
51  (defn harmonic-antinodes-for-pair [grid [a b]]
52    (let [[ax ay] a
53          [bx by] b
54          [dx dy] (dx-dy-pair a b)]
55      (for [n (range (- (count grid)) (count grid))]
56        [(+ ax (* n dx)) (+ ay (* n dy))])))
```

Check how it works (and how much wasted work we do) (line goes off page for dramatic effect):

```
(harmonic-antinodes-for-pair (range 12) [[1 8] [2 5]])

([-11 44] [-10 41] [-9 38] [-8 35] [-7 32] [-6 29] [-5 26] [-4 23] [-3 20] [-2 17] [-
```

But, since it's easily fast enough (runs basically instantly), I don't really care. Personal growth!

```
57  (assert (= 34 (p1 example)))
58  (p1 input)

1229
```

9 TODO Day 9

I'm doing today's in python because I failed at clojure.

A *disk map* is given like below, and we need to rearrange it to remove the empty spaces, and compute a checksum based on the new arrangement.

2333133121414131402

The digits alternate between indicating the length of a file, and the length of free space. The final goal is to move the rightmost file blocks to the leftmost empty spaces, until that's no longer possible. It's useful to keep track of the empty spaces and file blocks separately, so we build those two by looping over the file input.

```python
def parse_puzzle(puzzle):
    files = list()
    freespace = list()
    for i, elem in enumerate(puzzle.strip()):
        if i % 2 == 0:
            files.append([i // 2] * int(elem))
        if i % 2 == 1:
            freespace.append(int(elem))
    freespace.append(0)
    return [list(a) for a in zip(files, freespace)]
```

9.1 Part 1

We iterate over the input, where for each empty space we find, we move fileblock from the right to the empty space. We keep track of two pointers: where we are at the beginning (where empty spaces might be), and where we are at the end (where we move blocks forward). We do this until the pointers overlap, and the moving logic breaks down into three rules:

space size == amount of file blocks Move file blocks to empty space, move to next empty space, remove file blocks from end;

space size < amount of file blocks Move `space` amount of file blocks to empty space, move to next empty space, remove `space` amount of file blocks from end;

space size > amount of file blocks Move file blocks to empty space, keep pointer at current empty space, decrease empty space size, remove file blocks from end.

```python
def defragment(puzzle):
    diskmap = parse_puzzle(puzzle)
```

```
13
14      result = list()
15      j = len(diskmap) - 1
16      i = 0
17      while i < j:
18          group, space = diskmap[i]
19          fileblock, _ = diskmap[j]
20
21          result.extend(group)
22
23          if space == len(fileblock):
24              result.extend(fileblock)
25              i += 1
26              j -= 1
27          elif space < len(fileblock):
28              result.extend([fileblock[0]] * space)
29              diskmap[j][0] = fileblock[:len(fileblock) - space]
30              i += 1
31          elif space > len(fileblock):
32              result.extend(fileblock)
33              diskmap[i][0] = []
34              diskmap[i][1] -= len(fileblock)
35              j -= 1
36
37      result.extend(diskmap[j][0])
38
39      return result
```

Finally, we need to compute a checksum: `block_position * file_id`, where `file_id` is the index of the file blocks before moving them.

```
40  def checksum(diskmap):
41      result = 0
42      for i, elem in enumerate(diskmap):
43          elem = 0 if elem == '.' else elem
44          result += i * elem
45      return result
46
47  def p1(puzzle):
48      diskmap = defragment(puzzle)
49      return checksum(diskmap)
50
51  with open("inputs/9") as f:
52      contents = f.read()
53
```

```
54  assert p1(example) == 1928
55  print(p1(contents))
```

6283170117911

9.2   Part 2

Part Two is slightly different, and I mistakenly thought I properly understood it twice. You have to move *entire* file blocks, starting from the end of the diskmap and trying to put each fileblock at the leftmost possible space.

```
56  def defrag_2(puzzle):
57      diskmap = parse_puzzle(puzzle)
58
59      j = len(diskmap) - 1
60      while j > 1:
61          tomove, innerspace = diskmap[j]
62          for i, (group, space) in enumerate(diskmap[:j]):
63              if len(tomove) <= space:
64                  diskmap[i][1] = 0   # immediately after i
65                  diskmap[j-1][1] += len(tomove) + innerspace   # add room where j was
66                  del diskmap[j]
67                  diskmap.insert(i+1, [tomove, space - len(tomove)])
68                  break
69          j -= 1
70
71      return flatten_diskmap(diskmap)
72
73  def flatten_diskmap(diskmap):
74      result = list()
75      for (group, space) in diskmap:
76          result.extend(group)
77          result.extend(['.'] * space)
78      return result
79
80  def p2(puzzle):
81      diskmap = defrag_2(puzzle)
82      return checksum(diskmap)
83  assert p2(example) == 2858
84  print(p2(contents))
```

9813645302006

10 Day 10

10.1 Part 1

```
89010123
78121874
87430965
96549874
45678903
32019012
01329801
10456732
```

Each position in the above map is given by a number and represents the *height* of the map. A *hiking trail* is a path that starts at height 0, ends at height 9, always increases by a height of exactly 1 at each step, and don't include diagonal steps. A *trailhead* is any position that starts one or more hiking trails (so they always start at 0 here). A trailhead's *score* is the number of 9-height positions reachable from that trailhead.

The example above has 9 trailheads, which have scores of 5, 6, 5, 3, 1, 3, 5, 3 and 5, summing to **36**.

```
1   (ns aoc.10)

2   (defn trailhead-score [grid trailhead])

3
4   (defn p1 [input]
5     (let [grid (aoc.util/adjacent-num-list-per-line input)
6           trailheads (aoc.util/char-locations grid 0)]
7       (->> trailheads
8            (map (partial trailhead-routes grid))
9            (map distinct)
10           (map count)
11           (reduce +))))
```

trailhead-routes is a recursive function, where we find the 9s we can reach, duplicated if they are reachable via multiple paths (spoilers for part 2). We traverse the **position**'s neighbours and filter the ones that are 1 step higher. From these neighbours, we call the function again. When we are at height 8, we don't traverse further but instead return the amount of 9's next to us.

```
12  (defn trailhead-routes
13    [grid position]
14    (let [height (get-in grid position)
```

```
15          higher-neighbours
16          (->> aoc.util/cardinal-directions
17                (map (partial aoc.util/move position)) ;; all neighbours
18                (filter (fn [pos] (= (inc height)      ;; higher neighbours
19                                     (get-in grid pos)))))]
20      (if (= 8 height)
21        higher-neighbours
22        (mapcat #(trailhead-routes grid %) higher-neighbours))))
```

As an example, see what end spaces you can reach from the trailhead at [4 6]:

```
(let [grid (aoc.util/adjacent-num-list-per-line example)]
  (trailhead-routes grid [4 6]))

([2 5] [2 5] [4 5] [3 4])
```

Note that [2 5] is duplicated so we need to remove this, hence the distinct call.

```
23  (def input (slurp "inputs/10"))
24  (assert (= 36 (p1 example)))
25  (p1 input)

    482
```

10.2   Part 2

Foreshadowing is complete! We need to find out many distinct routes begin at a given position, and sum that for each of the starting positions. For the example, the sum is **81**. This means that we just remove the (map distinct) call and we are done.

```
26  (defn p2 [input]
27    (let [grid (aoc.util/adjacent-num-list-per-line input)
28          trailheads (aoc.util/char-locations grid 0)]
29      (->> trailheads
30           (map (partial trailhead-routes grid))
31           (map count)
32           (reduce +))))

33  (assert (= 81 (p2 example)))
34  (p2 input)

    1094
```

11 Day 11

11.1 Part 1

We get a row of stones with numbers on them that changes each time we blink:

- An engraved 0 will be converted to a 1;

- An engraged number with an even amount of digits will be split into two stones;

- Otherwise, an engraved n will be converted to n \cdot 2024.

After 25 blinks, the two numbers below will have be converted into **55312** stones. This leads itself nicely to recursion, and even though it grows exponentially that way, I'll have to look into caching/memoization for clojure.

125 17

```
1  (ns aoc.11)
```

With some knowledgable lookahead into the future, let's make p1 variable in the amount of blinks, defaulting to *25*. The input numbers are on a single line, separated by space—we parse them to an int, run num-stones on it in parallel, and sum the result.

```
2  (defn num-stones [stone n])
3
4  (defn p1
5    ([input] (p1 input 25))
6    ([input blinks]
7     (let [numbers (clojure.string/split input #"\s")]
8       (->> numbers
9            (map read-string)
10           (pmap #(num-stones % blinks))
11           (reduce +)))))
```

num-stones is a recursive function that's pretty straightforward:

```
12  (defn split-stone [stone]
13    (let [sstone (str stone)
14          mid (/ (count sstone) 2)]
15      (map Long/parseLong [(subs sstone 0 mid) (subs sstone mid)])))
16
17  (defn num-stones [stone n]
18    (if (= 0 n)
```

```
19        1
20      (cond (zero? stone) (num-stones 1 (dec n))
21            (even? (count (str stone)))
22            (let [[l r] (split-stone stone)]
23              (+ (num-stones l (dec n))
24                 (num-stones r (dec n))))
25            :default (num-stones (* 2024 stone) (dec n)))))
```

And now let's run it! This is *not* memoized, meaning that it computes everything all the time. We're in Part One, after all.

```
26   (assert (= 55312 (p1 example)))
27   (def input (slurp "inputs/11"))
28   (time (p1 input))

     "Elapsed time: 112.797349 msecs"
     200446
```

11.2 Part 2

Okay that was decent, but it grows exponentially wrt `blinks`:

```
(time (p1 input 25))
(time (p1 input 29))
(time (p1 input 33))

"Elapsed time: 126.310147 msecs"
"Elapsed time: 601.609624 msecs"
"Elapsed time: 3127.742377 msecs"
5655557
```

And now it just so happens that Part Two is Part One, except for **75 blinks**. This means that the above growth isn't all that feasible. Fortunately, we can use the clojure built-in `memoize`. Let's also rerun it on Part One to see how fast that can go.

```
29   (def num-stones (memoize num-stones))
30   (time (p1 input))
31   (time (p1 input 75))

     "Elapsed time: 5.958738 msecs"
     "Elapsed time: 191.842359 msecs"
     238317474993392
```