Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 13, 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.

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

Files

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

Often the lines contain numbers:

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

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

```
  [input]
  (->> (string-as-lines input)
       (map #(mapv read-string (str/split % #"\s+")))
       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:

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

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

Grids

First, the directions you can move in a grid:

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

(def diagonal-directions
  [[-1 -1]    ; NW
   [-1  1]    ; NR
   [ 1 -1]    ; SW
   [ 1  1]]) ; SE

(def all-directions
  (concat cardinal-directions diagonal-directions))
```

And how you can use them:

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

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

```
[3 2]
```

And finding characters in a grid:

```
(defn char-locations [grid x]
  (mapcat (fn [row string]
            (keep-indexed (fn [col char] (when (= char x) [row col]))
                          string))
          (range)
          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
(ns aoc.1)

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

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

(defn p1 [input]
  (let [input (aoc.util/num-list-per-line input)]
    (->> input
         (transpose)
         (map sort)
         (transpose)
         (map #(abs (- (first %) (second %))))
         (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
(assert (= 11 (p1 testinput)))
(def input (slurp "inputs/1"))
(time (p1 input))
```

```
"Elapsed time: 5.296958 msecs"
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.

```
(defn p2 [input]
  (let [input (transpose (aoc.util/num-list-per-line input))
        one (first input)
        freqs (frequencies (second input))]
    (->> one
         (map #(* % (freqs % 0)))
         (sum))))

(assert (= 31 (p2 testinput)))
(time (p2 input))
```

```
"Elapsed time: 4.619827 msecs"
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:

```
(ns aoc.2)

(defn diffs [record]
  (->> record
       (partition 2 1)
       (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.

```
(defn is-safe? [record]
  (let [differences (diffs record)]
    (and (every? #(<= 1 (abs %) 3) differences)
         (apply = (map pos? differences)))))

(defn p1 [input]
  (->> (aoc.util/num-list-per-line input)
       (filter is-safe?)
       (count)))
```

Recall that the testinput had **2** safe records.

```
(assert (= 2 (p1 testinput)))
(def input (slurp "inputs/2"))
(time (p1 input))
```

```
"Elapsed time: 16.115067 msecs"
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 convoluted and ugly to read. Right now, I just remove each level one by one and check if the record is safe then.

```
(defn drop-nth [coll n]
  (keep-indexed #(if (not= %1 n) %2) coll))

(defn dampened-is-safe? [record]
  (some is-safe? (map #(drop-nth record %)
                      (range (count record)))))

(defn p2 [input]
  (->> (aoc.util/num-list-per-line input)
       (filter dampened-is-safe?)
       (count)))

(assert (= 4 (p2 testinput)))
(time (p2 input))
```

```
"Elapsed time: 53.09667 msecs"
311
```

2.2.1 Benchmark results

The old solution took on average `38` milliseconds to execute `(p2 input)`, and the new solution about `60`. 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")]
  (time (p1 input)))
```

```
"Elapsed time: 12.06208 msecs"
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`.

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

```
(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.

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

```
"Elapsed time: 0.70303 msecs"
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.

This gives rise to the obvious helper function `is-xmas?`, which takes the `grid`, a `start` coordinate and a `direction`. It returns `true` if `"XMAS"` occurs in the `grid` from `start` in the given `direction`.

```
(ns aoc.4)
```

Using the `util` functions in Util/Grids, we can traverse the grid like so:

```
(let [start [4 5]
      direction (first aoc.util/cardinal-directions)] ; Up
  ; move up like so
  (println (aoc.util/move start direction))
  (let [[x y] start
        [x' y'] direction]
    ; or like so
    (print [(+ x x') (+ y y')])))
```

```
[3 5]
[3 5]
```

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

```
(defn is-xmas? "Does the grid `grid` contain the string \"XMAS\",
 starting at `start` and going in `direction`?"
  [grid start direction]
  (loop [location start
         chars (seq "XMAS")]
    (if (empty? chars)
      true
      (if (not= (get-in grid location) (first chars))
        false
        (recur (aoc.util/move location direction)
               (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.

```
(defn count-xmases-at [grid start directions]
  (count (filter #(is-xmas? grid start %) directions)))

(defn p1 [input]
  (let [grid (aoc.util/string-as-lines input)
        xs (aoc.util/char-locations grid \X)]
    (->> xs
         (map #(count-xmases-at grid % aoc.util/all-directions))
         (reduce +))))

(assert (= 18 (p1 example)))
(def input (slurp "inputs/4"))
(time (p1 input))
```

```
"Elapsed time: 65.706921 msecs"
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`. However, 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. `is-mas?` is now pretty trivial:

```
(defn is-mas? [grid middle direction]
  (let [opposite-direction (mapv #(* -1 %) direction)]
    (and (= \M (get-in grid (aoc.util/move middle direction)))
         (= \S (get-in grid (aoc.util/move middle opposite-direction))))))
```

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

```
(defn count-mases-at [grid middle directions]
  (count (filter #(is-mas? grid middle %) directions)))

(defn p2 [input]
  (let [grid (aoc.util/string-as-lines input)
        as (aoc.util/char-locations grid \A)]
    (->> as
         (map #(count-mases-at grid % aoc.util/diagonal-directions))
         (filter #(= % 2))
         (count))))

(assert (= 9 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 39.701527 msecs"
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:

```
(ns aoc.5
  (:require [clojure.string :as str]))

(defn sort [dependency-graph update])
(defn build-dependency-graph [orderings])

(defn sorted? [dependency-graph update]
  (= update (sort dependency-graph update)))

(defn middle-num
  "Finds the middle string in a list of string, and parses it to a
  number. Assumes the length of the list list is odd."
  [update]
  (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.

```
(defn parse-input
  "Parses an input string and returns three useful objects.
  The first obj is a list of orderings, strings of type \"A|B\".
  The second obj is a list of updates, each one a list of strings.
  The third obj is a dependency graph, a map."
  [input]
  (let [[orderings updates] (str/split input #"\n\n")
        orderings (str/split orderings #"\n")
        updates (str/split updates #"\n")
        updates (map #(str/split % #",") updates)
        dependency-graph (build-dependency-graph orderings)]
    [orderings updates dependency-graph]))

(defn p1 [input]
  (let [[orderings updates dep-graph] (parse-input input)
        sorted? (partial sorted? dep-graph)]
    (->> updates
         (filter sorted?)
         (map middle-num)
         (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.

```
(defn build-dependency-graph
  [orderings]
  (let [order-pairs (->> orderings
                         (map #(str/split % #"\|"))
                         (map #(hash-map (second %), [(first %)])))]
    (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.

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

```
(defn sort
  "Sort a list of strings based on a dependency map.
   The map defines which elements should come after others."
  [dep-graph update]
  (let [graph (reduce (fn [acc item]
                        (assoc acc item
                               (set (get dep-graph item []))))
                      {} update)
        local-deps (fn [deps] (filter #(contains? (set update) %) deps))]
    (vec (sort-by (fn [item]
                    (let [deps (get dep-graph item [])]
                      (count (local-deps deps))))
                  update))))
```

Now we got everything, ain't we?

```
(assert (= 143 (p1 example)))
(def input (slurp "inputs/5"))
(time (p1 input))
```

```
"Elapsed time: 957.383223 msecs"
4637
```

yes

5.1.1 Optimization

Instead of doing the filtering in `p1` like above, we can do it like so:

```
(defn p1 [input]
  (let [[orderings updates dep-graph] (parse-input input)
        sorted? (partial sorted? dep-graph)]
    (->> updates
         (pmap #(list % (sorted? %)))
         (filter last)
         (pmap first)
         (pmap middle-num)
         (reduce +))))
```

This is a bit uglier, but it does make it about 3 times as fast:

```
(time (p1 input))
```

```
"Elapsed time: 321.296271 msecs"
4637
```

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.

```
(defn p2 [input]
  (let [[orderings updates deps] (parse-input input)
        is-sorted? (partial sorted? deps)
        sort (partial sort deps)]
    (->> updates
         (filter #(not (is-sorted? %)))
         (pmap sort)
         (pmap middle-num)
         (reduce +))))
```

```
(assert (= 123 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 1151.020532 msecs"
6370
```

5.2.1 Same optimization

Again, first do the sorting in parallel, save that alongside the unsorted list, filter the ones that differ, and then do the rest.

```
(defn p2 [input]
(let [[orderings updates deps] (parse-input input)
      is-sorted? (partial sorted? deps)
      sort (partial sort deps)]
  (->> updates
       (pmap #(list % (sort %)))
       (filter #(not= (first %) (last %)))
       (pmap last)
       (pmap middle-num)
       (reduce +))))
```

```
(assert (= 123 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 312.860191 msecs"
6370
```

Nice!

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 only use the four directions in `aoc.util/cardinal-directions` (see Util/Grids).

```
(ns aoc.6)
```

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

```
(defn guard-route
  "Takes a `grid` as input returns a vector of 2d coordinates: the route
  of the guard, starting at `start` and turning clockwise at \"#\"
  characters. "
```

```
  [grid start])

(defn p1 [input]
  (let [grid (aoc.util/string-as-lines input)
        start (first (aoc.util/char-locations grid \^))
        route (guard-route grid start)]
    (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.

```
(defn replace-char
  [grid [x y] new-char]
  (update grid x
          #(str (subs % 0 y)
                new-char
                (subs % (inc y)))))

(defn guard-route [grid start]
  (let [size (count grid)
        grid (replace-char grid start \.)]
    (loop [location start
           directions (cycle aoc.util/cardinal-directions)
           route []]
      (let [[x y] location
            [x' y'] (first directions)
            next-location [(+ x x') (+ y y')]
            next-object (get-in grid next-location)
            route (conj route location)]
        (condp = next-object
          nil route
          \. (recur next-location
                    directions
                    route)
          \# (recur location
                    (next directions)
                    route))))))
```

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

```clojure
(assert (= 41 (p1 example)))
(def input (slurp "inputs/6"))
(time (p1 input))
```

```
"Elapsed time: 15.44612 msecs"
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.

```clojure
(defn route-has-loop? [grid start])

(defn p2 [input]
  (let [grid (aoc.util/string-as-lines input)
        start (first (aoc.util/char-locations grid \^))
        route (disj (set (guard-route grid start)) start)]
    (->> route
         (pmap (fn [new-obstacle]
                 (route-has-loop? (replace-char grid new-obstacle \#) start)))
         (filter true?)
         (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
(defn route-has-loop? [grid start]
  (let [size (count grid)
        grid (replace-char grid start \.)]
```

```
    (loop [location start
           directions (cycle aoc.util/cardinal-directions)
           visited #{}]
      (let [[x y] location
            [x' y'] (first directions)
            next-location [(+ x x') (+ y y')]
            next-object (get-in grid next-location)
            pair [next-location [x' y']]]
        (if (contains? visited pair)
          true ;; we have a loop!
          (condp = next-object
            nil false ;; we exited the puzzle
            \. (recur next-location
                      directions
                      visited)
            \# (recur location
                      (next directions)
                      (conj visited pair))))))))
```

On my laptop, this takes about 20 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!)

```
(assert (= 6 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 6011.385025 msecs"
1972
```

6.2.1 Timing of different methods

The following table shows a short overview of the results of `(time (p2 input))` (it's too slow for `(crit/quick-bench)`) with some different variants:

Method	time
Set - always add & check	9s
Set - only add on obstacle	6s
Set - only check on north	5.5
Counter (7000)	4s

1. `set` methods. These methods refer to keeping track of a `set` of visited `(node, direction)` pairs. If we've seen one before, we must be in a loop. My original implementation was *Set - always add & check*: add

every location we visit to the `visited` set and check if we've seen it before. That turned out to be the slowest one—my code spent about 10% of its time hashing entries. One step faster is *Set - only add on obstacle*, which only adds an element to the set when we visit an obstacle.

The fastest `set`-related method (though only slightly) was *Set - only check on north*, and this only checks if the current `(node, direction)`-pair is in `visited` if `direction =` North=. This was counterintuitive for me since that meant it was actually doing some extra work: it might be traversing the current path for up to 3 extra obstacles compared to the previous one. However, the hashing was apparently so expensive compared to traversing the grid that this was still a hair faster.

Since this was only slightly faster but made the code a bit convoluted and difficult to understand ("why are we only checking if we've been here if we're heading North right now?"), I opted for the second method.

2. `Counter`. This is kind of a hack, but it's faster than the `set`-methods. Instead of a `visited` set, we keep track of a `counter` of nodes we've visited. Once we reach 7000, we assume we're in a loop and exit. 6500 also worked for me, but that might be too input-dependent.

   Still, any arbitrary number fails for some input, so I've opted to not do this.

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

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

```clojure
(defn is-correct? [equation])

(defn parse-equations [input]
  (let [lines (str/split input #"\n")
        equations (map #(str/split % #": ") lines)]
    (map (fn [[lhs rhs]]
           [(read-string lhs) (vec (map read-string (str/split rhs #" ")))])
         equations)))

(defn p1 [input]
  (->> input
       (parse-equations)
       (filter is-correct?)
       (map first)
       (reduce +)))
```

```clojure
(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.

```
(defn possible-ops
  [x y]
  [(* x y)
   (+ x y)])

(defn equation-possibilities
  [target nums]
  (->> (range 1 (count nums))
       (reduce (fn [possible-results idx]
                 (->> possible-results
                      (mapcat (fn [result]
                                (possible-ops result (nth nums idx))))))
               [(first nums)])))

(defn is-correct? [equation]
  (let [[result numbers] equation
        targets (equation-possibilities result numbers)]
    (some #(= % result) targets)))

(assert (= 3749 (p1 example)))
(def input (slurp "inputs/7"))
(time (p1 input))
```

```
"Elapsed time: 158.039096 msecs"
12839601725877
```

7.2 Part 2

Now this is an elegant Part Two.

```clojure
(defn possible-ops
  [x y]
  [(* x y)
   (+ x y)
   (Long/parseLong (str x y))])

(assert (= 11387 (p1 example)))
(time (p1 input))
```

```
"Elapsed time: 5462.497567 msecs"
149956401519484
```

7.3 Optimization

There is a nice way to optimize this. Since this one actually takes quite long (Part One takes about 150ms, Part Two around 5-6s), 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.

```
(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 antinodes for all pairs, and uses the function `antinodes-for-pair` to find the antinodes for a given pair.

```
(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]]
```

```
(defn all-antinodes-for-freq [grid freq]
  (let [indices (aoc.util/char-locations grid freq)
        pairs (combo/combinations indices 2)]
    (mapcat antinodes-for-pair pairs)))

(defn all-antinodes-for-freq-in-grid [grid freq]
  (let [antinodes (all-antinodes-for-freq grid freq)]
    (filter (partial in-grid? grid) antinodes)))
```

```
(assert (= 14 (p1 example)))
(def input (slurp "inputs/8"))
(time (p1 input))
```

```
"Elapsed time: 41.71457 msecs"
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.

```
(defn harmonic-antinodes-for-pair [grid [a b]])

(defn all-antinodes-for-freq-in-grid [grid freq]
  (let [indices (aoc.util/char-locations grid freq)
        pairs (combo/combinations indices 2)]
    (->> pairs
         (mapcat #(harmonic-antinodes-for-pair grid %))
         (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.

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

(defn harmonic-antinodes-for-pair [grid [a b]]
  (let [[ax ay] a
        [bx by] b
        [dx dy] (dx-dy-pair a b)]
    (for [n (range (- (count grid)) (count grid))]
      [(+ 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, I don't really care. Personal growth!

```
(assert (= 34 (p1 example)))
(time (p1 input))
```

```
"Elapsed time: 53.923789 msecs"
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)
```

```python

    result = list()
    j = len(diskmap) - 1
    i = 0
    while i < j:
        group, space = diskmap[i]
        fileblock, _ = diskmap[j]

        result.extend(group)

        if space == len(fileblock):
            result.extend(fileblock)
            i += 1
            j -= 1
        elif space < len(fileblock):
            result.extend([fileblock[0]] * space)
            diskmap[j][0] = fileblock[:len(fileblock) - space]
            i += 1
        elif space > len(fileblock):
            result.extend(fileblock)
            diskmap[i][0] = []
            diskmap[i][1] -= len(fileblock)
            j -= 1

    result.extend(diskmap[j][0])

    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.

```python
def checksum(diskmap):
    result = 0
    for i, elem in enumerate(diskmap):
        elem = 0 if elem == '.' else elem
        result += i * elem
    return result

def p1(puzzle):
    diskmap = defragment(puzzle)
    return checksum(diskmap)

with open("inputs/9") as f:
    contents = f.read()

```

```
assert p1(example) == 1928
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.

```
def defrag_2(puzzle):
    diskmap = parse_puzzle(puzzle)

    j = len(diskmap) - 1
    while j > 1:
        tomove, innerspace = diskmap[j]
        for i, (group, space) in enumerate(diskmap[:j]):
            if len(tomove) <= space:
                diskmap[i][1] = 0  # immediately after i
                diskmap[j-1][1] += len(tomove) + innerspace  # add room where j was
                del diskmap[j]
                diskmap.insert(i+1, [tomove, space - len(tomove)])
                break
        j -= 1

    return flatten_diskmap(diskmap)

def flatten_diskmap(diskmap):
    result = list()
    for (group, space) in diskmap:
        result.extend(group)
        result.extend(['.'] * space)
    return result

def p2(puzzle):
    diskmap = defrag_2(puzzle)
    return checksum(diskmap)
assert p2(example) == 2858
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`, and always increases by a height of exactly 1 at each step. A *trailhead* is any position that starts one or more hiking trails. 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**.

```clojure
(ns aoc.10)
```

```clojure
(defn trailhead-routes [grid trailhead])

(defn trail-routes [grid]
  (let [trailheads (aoc.util/char-locations grid 0)]
    (map #(trailhead-routes grid %) trailheads)))

(defn p1 [input]
  (let [routes (trail-routes (aoc.util/adjacent-num-list-per-line input))]
    (->> routes
         (map distinct)
         (map count)
         (reduce +))))
```

`trailhead-routes` gives us the 9s we can reach from a `trailhead`, duplicated if they are reachable via multiple paths. 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.

```clojure
(defn trailhead-routes
  [grid position]
  (let [height (get-in grid position)
```

```
        higher-neighbours
        (->> aoc.util/cardinal-directions
             (map (partial aoc.util/move position)) ;; all neighbours
             (filter (fn [pos] (= (inc height)      ;; higher neighbours
                                  (get-in grid pos)))))]
    (if (= 8 height)
      higher-neighbours
      (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.

```
(def input (slurp "inputs/10"))
(assert (= 36 (p1 example)))
(time (p1 input))
```

```
"Elapsed time: 28.457049 msecs"
482
```

10.2 Part 2

Foreshadowing is complete! We need to find out many 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.

```
(defn p2 [input]
  (let [routes (trail-routes (aoc.util/adjacent-num-list-per-line input))]
    (->> routes
         (map count)
         (reduce +))))

(assert (= 81 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 28.731688 msecs"
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 engraved number with an even amount of digits will be split into two stones;
- Otherwise, an engraved `n` will be converted to `n`·`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, caching/memoization can take care of that nicely.

```
125 17
```

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.

```
(ns aoc.11 (:require [clojure.string :as str]))

(defn num-stones [stone n])

(defn p1
  ([input] (p1 input 25))
  ([input blinks]
   (let [numbers (str/split input #"\s")]
     (->> (map read-string numbers)
          (pmap #(num-stones % blinks))
          (reduce +)))))
```

`num-stones` is a recursive function that's pretty straightforward given the rules above. It computes the amount of stones we get after `n` blinks and a given starting `stone`. We introduce a small function `split-stone` to help us with the second rule:

```
(defn split-stone [stone]
  (let [sstone (str stone)
        mid (/ (count sstone) 2)]
    (map Long/parseLong [(subs sstone 0 mid) (subs sstone mid)])))

```

```clojure
(defn num-stones [stone n]
  (if (= 0 n)
    1
    (cond (zero? stone) (num-stones 1 (dec n))
          (even? (count (str stone)))
          (let [[l r] (split-stone stone)]
            (+ (num-stones l (dec n))
               (num-stones r (dec n))))
          :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.

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

```
"Elapsed time: 124.957979 msecs"
200446
```

11.2 Part 2

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

```clojure
(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.

```clojure
(def num-stones (memoize num-stones))
(time (p1 input))
(time (p1 input 75))
```

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

12 Day 13

12.1 Part 1

This is our example input:

```
Button A: X+94, Y+34
Button B: X+22, Y+67
Prize: X=8400, Y=5400

Button A: X+26, Y+66
Button B: X+67, Y+21
Prize: X=12748, Y=12176

Button A: X+17, Y+86
Button B: X+84, Y+37
Prize: X=7870, Y=6450

Button A: X+69, Y+23
Button B: X+27, Y+71
Prize: X=18641, Y=10279
```

Which is a set of *machines* separated by empty lines, with the goal to find a and b such that $ax + bx = px$ and $ay + by = py$. So, each machine is a system of linear equations which we're have to solve. Let's use the first machine as an example and call the equations $L1$ and $L2$.

$$L1 = 94a + 22b = 8400$$
$$L2 = 34a + 67b = 5400$$

We're going to use some form of Gaussian elimination to solve this. This entails removing a from equation 2, and removing b from equation 1. In this case, removing a from equation 2 is done by performing the update: $L2 = L2 - \frac{34}{94} \cdot L1$

Which gives us this:

$$L1 = 94a + 22b = 8400$$
$$L2 = 0 + \frac{5550}{94}b = \frac{222000}{94}$$

Which we can simplify (we don't *need* to, but it makes this easier to follow) by multipling $L2$ by 94:

$$L1 = 94a + 22b = 8400$$
$$L2 = L1 = 0 + 5550b = 222000L$$

Now, let's remove b from equation 1, in this case with this update: $L1 = L1 - \frac{22}{5550} \cdot L1$

$$L1 = 94a + 0 = 7520$$
$$L2 = 0 + 5550b = 222000$$

Now that we've isolated the variables we know our solution:

$$a = 80$$
$$b = 40$$

And our final puzzle result is obtained by doing this for all machines where you can find a solution, multiplying a by 3 and b by 1, and summing this for all equations. There is only one caveat, and that is that we're going to run into floating point issues. You could use other number formats, but for efficiency I'm just going to keep it to floats and fix the issue with a somewhat arbitrary epsilon (though this is hard to do generally, it'll probably work fine for this input). Let's get coding.

All machines are separated by a newline, so split them on that:

```
def p1(puzzle):
    equations = puzzle.strip().split("\n\n")
    return sum(price(*parse_machine(eq))
               for eq in equations)
```

I'm using two functions that I haven't defined yet, `parse_machine` and `price`. I'm splitting the puzzles up in newlines, so I'll get a block of 3 lines that define a *machine*. I want `parse_machines` to return an `Equation`, which is a tranposition of the how we get them line-by-line (see equations above).

```
from collections import namedtuple

Equation = namedtuple('Equation', ['x', 'y', 'g'])

def parse_machine(machine: str) -> tuple[Equation]:
    """ Parses a 'machine' into an Equation.
    Example machine:

        Button A: X+94, Y+34
        Button B: X+22, Y+67
        Prize: X=8400, Y=5400
    """
    (ax, ay), (bx, by), (gx, gy) = map(parse_line, machine.split("\n"))
    return (Equation(ax, bx, gx),
            Equation(ay, by, gy))
```

```python

def parse_line(line: str) -> list[int]:
    pattern = r"(?:Button|Prize) ?[AB]?: X[+=](\d+), Y[+=](\d+)"
    return map(int, re.match(pattern, line).groups())
```

The `price` of an equation is defined as $3a+b$ if a and b solve the equation, and 0 otherwise. The puzzle actually says that you must find the *lowest* price possible, but by looking at the input I found out that all systems of equations actually have a unique solution, so I can freely ignore that red herring.

```python
def price(one: Equation, two: Equation) -> int:
    a, b = 0, 0
    solution = solve(one, two)
    if solution:
        a, b = solution
    return a*3 + b*1
```

Now for some high school math. If it doesn't make sense anymore, look at what we did above or read the wiki page on Gaussian elimination and it should.

```python
from typing import Optional

EPSILON = 0.001

def solve(one: Equation, two: Equation) -> Optional[int]:
    # Remove x from equation 2
    factor = two.x / one.x
    two = Equation(two.x - factor * one.x,  # Always 0
                   two.y - factor * one.y,
                   two.g - factor * one.g)

    # Remove y from equation 1
    factor = one.y / two.y
    one = Equation(one.x - factor * two.x,  # Just one.x
                   one.y - factor * two.y,  # Always 0
                   one.g - factor * two.g)

    # Simplify to forms 'a + 0 = gx' and '0 + b = gy'
    one = Equation(1, 0, one.g / one.x)
    two = Equation(0, 1, two.g / two.y)

    if all(abs(round(s) - s) < EPSILON for s in [one.g, two.g]):
        return round(one.g), round(two.g)
```

```python
assert 480 == p1(example)
with open("inputs/13") as f:
    puzzle = f.read()
print(p1(puzzle))
```

```
37901
```

12.2 Part 2

The prize locations weren't correct: we need to add `100000000` to the `X` and `Y` position of each prize. We can easily alter `parse_machine` and rerun `p1` on the input:

```python
def parse_machine(machine: str) -> tuple[Equation]:
    add = 10000000000000
    (ax, ay), (bx, by), (gx, gy) = map(parse_line, machine.split("\n"))
    return (Equation(ax, bx, gx + add),
            Equation(ay, by, gy + add))

print(p1(puzzle))
```

```
77407675412647
```