Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 19, 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])
  (:require [clojure.data.priority-map :refer [priority-map]]))
```

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)

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

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])
```

It's sometimes useful to convert a grid to a graph: a mapping of nodes in a 2d grid to their cardinal neighbours. AoC seems to have the standard notation that a `#` is a wall in the grid, so we don't connect those nodes.

```
(defn grid-to-graph [grid]
  (letfn [(neighbours [grid node]
            (->> (map aoc.util/move aoc.util/cardinal-directions (repeat node))
                 (filter #(= \. (get-in grid %)))
                 (map #(vector % 1))
                 (into {})))]
    (let [coords (for [x (range (count grid))
                       y (range (count (first grid)))]
                   [x y])]
      (->> coords
           (filter #(not= \# (get-in grid %)))
           (map #(vector % (neighbours grid %)))
           (into {})))))
```

Dijkstra

I got a lot of inspiration from dandormans implementation, especially the way they keep track of the routes, which contains `parent` and `weight` in one.

```
(defn relax [routes node node-weight neighbour neighbour-weight]
  (let [weight (+ node-weight neighbour-weight)]
    (cond-> routes
      (< weight (get-in routes [neighbour :weight] ##Inf))
      (assoc neighbour {:parent node :weight weight}))))

(defn dijkstra
  [graph src]
  (loop [q (priority-map src 0)
         routes {src {:parent nil :weight 0}}
         visited #{src}]
    (if-let [node (first q)]
      (let [node (key node)
            q (pop q)
            weight (get-in routes [node :weight] 0)
            neighbours (get graph node)
            routes (reduce-kv (fn [routes neighbour neighbour-weight]
                                (relax routes
                                       node weight
                                       neighbour neighbour-weight))
                              routes
                              neighbours)
```

```
87              neighbours (-> (filter #(not (contains? visited (key %))) neighbours)
88                             (update-vals #(+ % weight)))]
89          (recur (into q neighbours) routes (conj visited node)))
90        routes)))
```

Use it like this:

```
(let [result (dijkstra {:start {:a 1 :b 1}
                        :a {:start 1 :c 1}
                        :b {:start 1 :c 1}
                        :c {:a 1 :b 1}}
                       :start)]
  (:c result))

{:parent :b, :weight 2}
```

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`:

```
(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`.

```
(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.

```
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 (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:

```
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.

```clojure
(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.

```clojure
(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.

```clojure
(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.

```
(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`.

```
1 (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:

```clojure
(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.

```clojure
(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.

```
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.

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.

```
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          (pmap sort)
64          (pmap middle-num)
65          (reduce +))))

66 (assert (= 123 (p2 example)))
67 (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. "
```

```
6     [grid start])
7
8   (defn p1 [input]
9     (let [grid (aoc.util/string-as-lines input)
10          start (first (aoc.util/char-locations grid \^))
11          route (guard-route grid start)]
12      (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.

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

```
56      (loop [location start
57             directions (cycle aoc.util/cardinal-directions)
58             visited #{}]
59        (let [[x y] location
60              [x' y'] (first directions)
61              next-location [(+ x x') (+ y y')]
62              next-object (get-in grid next-location)
63              pair [next-location [x' y']]]
64          (if (contains? visited pair)
65            true ;; we have a loop!
66            (condp = next-object
67              nil false ;; we exited the puzzle
68              \. (recur next-location
69                        directions
70                        visited)
71              \# (recur location
72                        (next directions)
73                        (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!)

```
74  (assert (= 6 (p2 example)))
75  (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.

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

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

```
(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 +)))

(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 (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.

```
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, I don't really care. Personal growth!

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

```
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.

```
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`, 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**.

```
(ns aoc.10)

(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.

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

```
17          higher-neighbours
18          (->> aoc.util/cardinal-directions
19               (map (partial aoc.util/move position)) ;; all neighbours
20               (filter (fn [pos] (= (inc height)      ;; higher neighbours
21                                    (get-in grid pos)))))]
22      (if (= 8 height)
23        higher-neighbours
24        (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.

```
25 (def input (slurp "inputs/10"))
26 (assert (= 36 (p1 example)))
27 (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.

```
28 (defn p2 [input]
29   (let [routes (trail-routes (aoc.util/adjacent-num-list-per-line input))]
30     (->> routes
31          (map count)
32          (reduce +))))
33
34 (assert (= 81 (p2 example)))
35 (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 12

Haven't done this in literate yet: see day12.py.

13 Day 13

13.1 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 $L2$.

```
\begin{align*}
  L1 = 94a + 22b & = 8400 \\
  L2 = 34a + 67b & = 5400
\end{align*}
```

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

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:

```
$L2 = L2 - \frac{34}{94} \cdot L1$
```

$L2 = L2 - \frac{34}{94} \cdot L1$
Which gives us this:

```
\begin{align*}
  L1 = 94a + 22b             & = 8400 \\
  L2 = 0 + \frac{5550}{94}b & = \frac{222000}{94}
\end{align*}
```

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

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

```
\begin{align*}
  L1 = 94a + 22b & = 8400 \\
  L2 = L1 = 0 + 5550b & = 222000
L\end{align*}
```

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

Now, let's remove b from equation 1, in this case with this update:

```
$L1 = L1 - \frac{22}{5550} \cdot L1$
\begin{align*}
  L1 = 94a + 0    & = 7520 \\
  L2 = 0 + 5550b & = 222000
\end{align*}
```

$L1 = L1 - \frac{22}{5550} \cdot L1$

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

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

```
\begin{align*}
  a & = 80 \\
  b & = 40
\end{align*}
```

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

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 arbtirary 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))

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.

```
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.

```
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)

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

```
37901
```

13.2 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:

```
57 def parse_machine(machine: str) -> tuple[Equation]:
58     add = 10000000000000
59     (ax, ay), (bx, by), (gx, gy) = map(parse_line, machine.split("\n"))
60     return (Equation(ax, bx, gx + add),
61             Equation(ay, by, gy + add))
62
63 print(p1(puzzle))
```

```
77407675412647
```

14 Day 14

14.1 Part 1

We have a bunch of robots that are moving in a grid. Each one is given an initial coordinate and a velocity, like so:

```
p=0,4 v=3,-3
p=6,3 v=-1,-3
p=10,3 v=-1,2
p=2,0 v=2,-1
p=0,0 v=1,3
p=3,0 v=-2,-2
p=7,6 v=-1,-3
p=3,0 v=-1,-2
p=9,3 v=2,3
p=7,3 v=-1,2
p=2,4 v=2,-3
p=9,5 v=-3,-3
```

We need to find out where the robots are after `100` *seconds*, assuming a grid of `101x103` - the robots wrap around if they exit the grid. We can just multiply the velocities by `100` and mod them with the width and length.

The solution is found by grouping the elements in four quadrants, and taking the product of the count of robots per quadrant. Taking the input above in mind, let's define `parse-line` which quickly grabs the relevant numbers, and `move-n` as well while we're at it.

```
(ns aoc.14
  (:require [clojure.string :as str]))

(defn parse-line [line]
  (->> (re-matches #"p=(\d+),(\d+) v=(-?\d+),(-?\d+)" line)
       rest
       (map read-string)))

(defn move-n [width length n [x y dx dy]]
  [(mod (+ x (* n dx)) width)
   (mod (+ y (* n dy)) length)])
```

Now for doing Part One:

```
(defn quadrant [width length [x y]])

(def width 101)
(def length 103)
```

```clojure

(defn p1
  ([input] (p1 input 101 103))
  ([input width length]
   (let [quadrant (partial quadrant width length)
         move-n (partial move-n width length)]
     (->> (str/split input #"\n")
          (map parse-line)
          (map #(move-n 100 %))
          (group-by quadrant)
          (filter #(some? (first %)))
          (map last)
          (map count)
          (reduce *)))))
```

To rehash the operations above in the text: we need to find all robots, move them 100 spaces along the grid, group them into quadrants, count the quadrants and take the product of the counts.

We left `quadrant` empty, the input for `group-by`. We want it to return a vector of size 2, where `[0 0]` means the northwest quadrant, and `[1 1]` the souteast quadrant. The items in the middle shouldn't belong to any quadrant, and we'll not return anything for that - hence the `(filter #(some? (first %)))` line above.

```clojure
(defn quadrant
  [width length [x y]]
  (let [midx (dec (/ (inc width) 2))
        midy (dec (/ (inc length) 2))]
    (if-not (or (= x midx) (= y midy))
      [(int (/ x (inc midx))) (int (/ y (inc midy)))])))
```

Let's run it!

```clojure
(let [width 11
      length 7]
  (assert (= 12 (p1 example width length))))

(def input (slurp "inputs/14"))
(time (p1 input))
```

```
"Elapsed time: 22.91994 msecs"
228410028
```

14.2 Part 2

We need to find out what the first time a *Christmas tree* appears in the grid. We don't get a definition, so I figured let's print out a boatload of grids. I did so, and found two recurring patterns:

- A horizontal pattern appeared on step `18` and reappears every `103` steps.
- A vertical pattern appeared on step `77` and reappears every `101` steps.

If we find out when these appear at the same time, we surely find the christmas tree. So we need to find whole numbers x for which $18 + 103x = 77 + 101y$. In other words, we need to solve the formula $101y + 59 = 0 \bmod 103$. You can throw it in WolframAlpha and get a result (take the first natural number), or I can quickly brute-force it.

```
42 (let [x 101 y 103
43       startx 77
44       starty 18]
45   (->> (range y)
46        (filter #(= 0 (mod (+ (* x %) (- startx starty)) y)))
47        first))
```

```
81
```

```
48 (+ 77 (* 101 81))
```

```
8258
```

And let's verify that this is indeed correct:

```
(let [ysolution (/ (- 8258 18) 103)]
  (println ysolution)
  (println (+ 18 (* 103 ysolution))))
```

```
80
8258
```

15 Day 15

15.1 Part 1

We get a grid and a list of instructions. They're separated by two newlines, the instructions themselves have newlines as well, but they need to be removed (??). The instructionset is just a list of directions for the robot (`@`) to move. If the robot encounters a box (`O`), it pushes that box and any following blocks, unless there is a wall (`#`) in the way, in which case it ignores the instruction.

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

<vv>^<v^>v>^vv^v>v<>v^v<v<^vv<<<^><<><>>v<vvv<>^v^>^<<<><<v<<<v^vv^v>^
vvv<<^>^v^^><<>>><>^<<><^vv^^<>vvv<>><^^v>^>vv<>v<<<<v<^v>^<^^>>>^<v<v
><>vv>v^v^<>><>>>><^^>vv>v<^^^>>v^v^<^^>v^^>v^<^v>v<>>v^v^<v>v^^<^^vv<
<<v<^>>^^^^>>>v^<>vvv^><v<<<>^^^vv^<vvv>^>v<^^^^v<>^>vvvv><>>v^<<^^^^^
^><^><>>><^^<<^^v>>><^<v>^<vv>>v>>>^v><>^v><<<<v>>v<v<v>vvv>^<><<>^><
^>><>^v<><^vvv<^^<><v<<<<<><^v<<<><<<^^<v<^^^><^>>^<v^><<<^>>^v<v^v<v^
>^>>^v>vv>^<<^v<>><<><<v<<v><>v<^vv<<<>^^v^>^^>>><<^v>>v^v><^^>>^<>vv^
<><^^>^^^<><vvvvv^v<v<<>^v<v>v<<^><<><<><<<^^<<<^<<>><<><^^^>^^<>^>v<>
^^>vv<^v^v<vv>^<><v<^v>^^^>>>^^vvv^>vvv<>>>^<^>>>>>^<<^v>^vvv<>^<><<v>
v^^>>><<^^<>>^v^<v^vv<>v^<<>^<^v^v><^<<<><<<^<v><v<>vv>>v><v^<vv<>v^<<^
```

At the end of the instructions, the example above looks like this:

```
##########
#.O.O.OOO#
#........#
#OO......#
#OO@.....#
#O#.....O#
#O.....OO#
#O.....OO#
#OO....OO#
##########
```

After which we have to perform the following calculation: $100 \cdot x_b + y_b, \forall b \in \mathcal{B}$ where $\mathcal{B}$ is the set of Boxes, in the input denoted as `O`. This gives us **10092** for the example input.

We're going to follow the instructions via `follow-instructions`, which just iterates over the instructions until they're empty and calls `move-in-grid` for each one, which returns the updated grid and a new location.

Since we're going to update the grid quite often, I'm not going to store it as a vec of strings, but as a vec of vecs (of characters).

```
(ns aoc.15
  (:require [clojure.string :as str]))

(defn parse-puzzle [input]
  (let [[grid instructions] (str/split input #"\n\n")
        grid (aoc.util/string-as-lines grid)
        instructions (-> (str/replace instructions #"\n" "")
                         (str/trim))]
    [(mapv vec grid) instructions]))

(defn move-in-grid [grid location direction])

(defn follow-instructions
  ([grid instructions]
   (let [location (first (aoc.util/char-locations grid \@))]
     (follow-instructions grid instructions location)))

  ([grid instructions location]
   (loop [grid grid
          instructions instructions
          location location]
     (if (empty? instructions)
       grid
       (let [[grid location] (move-in-grid grid location (first instructions))]
         (recur grid (drop 1 instructions) location))))))

(defn compute-gps-coords
  ([grid] (compute-gps-coords grid \O))
  ([grid char]
   (->> (aoc.util/char-locations grid char)
        (map (fn [[x y]] (+ (* 100 x) y)))
        (reduce +))))

(defn p1 [input]
  (let [[grid instructions] (parse-puzzle input)
```

```
36          grid (follow-instructions grid instructions)]
37      (compute-gps-coords grid)))
```

`move-in-grid` takes the `grid`, a `location` and a `direction`, and does the following things in order:

1. Check what items we have to move (this depends on the amount of `O`-s in front of us);
2. If there's a `#` in the way, don't update the grid or location;
3. Otherwise, update the grid. The way we do this seems somewhat convoluted but ensures that we don't have to do a lot of grid updates if we have to push many blocks. What we do is:
   (a) Move a `O` to the "block destination". If we're just moving 1 step and there is no `O` we have to move, the destination is also "our destination" and this will be overriden. If there are `n` blocks to push, this moves the next block to the end of that blocklist.
   (b) Replace our position by an empty one (`.`)
   (c) Move our position by one step in the direction.

We use two helper functions:

- `move`, which returns the target coordinates given a start, direction and `n`;
- `items-to-move` this computes the amount of blocks we have to move. If there's no block in front of us, that's just the next tile. Otherwise, it's the tiles in the given direction until there's no more blocks.

```
38  (defn move
39    ([[x y] direction]
40     (move [x y] direction 1))
41    ([[x y] direction n]
42     (condp = direction
43       \^ [(- x n) y]
44       \> [x (+ y n)]
45       \v [(+ x n) y]
46       \< [x (- y n)])))
47
48  (defn items-to-move [grid location direction]
49    (loop [location (move location direction)
50           items []]
51      (let [it (get-in grid location)
52            items (conj items it)]
```

```
53        (condp = it
54          \# items
55          \. items
56          \O (recur (move location direction) items)))))
57
58  (defn move-in-grid
59    "Takes the grid, a start location and a direction to move in, and
60    returns an updated grid and the new location. Does not move if
61    stuck, and updates boxes correctly in the grid when moving."
62    [grid location direction]
63    (let [items (items-to-move grid location direction)]
64      (if (some #(= \# %) items)
65        ;; Stuck
66        [grid location]
67        (let [moveto (move location direction (count items))]
68          [(-> grid
69               (assoc-in moveto \O)
70               (assoc-in location \.)
71               (assoc-in (move location direction) \@))
72           (move location direction)])))))

73  (assert = (10092 (p1 example)))
74  (def input (slurp "inputs/15"))
75  (time (p1 input))
```

```
"Elapsed time: 77.573369 msecs"
1453699
```

15.2 Part 2

Part Two is concerned with a warehouse that's twice the size: `#` is converted to `##`, `.` to `..`, `@` to `@.`, and `O` to `[]`. Look at this example to see how this impacts us:

```
###############
##......##..##
##...[][]...##
##....[]....##
##....@.....##
##..........##
###############
```

In this case, we cannot push `^`, since one of the boxes touches a wall. The rest of the puzzle stays the same, but this means that we cannot use the same logic to update the grid (pushing a `O` at the end of a stack).

```clojure
(defn p2 [input]
  (let [input (-> input
                  (str/replace #"\#" "##")
                  (str/replace #"O" "[]")
                  (str/replace #"\." "..")
                  (str/replace #"@" "@."))
        [grid instructions] (parse-puzzle input)
        grid (follow-instructions grid instructions)]
    (compute-gps-coords grid \[)))
```

This is the correct shell of Part Two, except that our functions `push` and `items-to-move` had Part One things hardcoded. Let's recall what they need to do and redefine them for Part Two.

`items-to-move` Given a location and a direction, it finds the elements in the grid that should be moved. In P1, this was just the next item, except if we have multiple blocks after each other in our direction, in which case we'll have to move all of them.

The goal is of course the same, but the way of finding these items becomes more complicated in this part. If we are facing east or west, little changes, except that the blocks now take up two spaces in this direction. If we are facing north or south, we will always face one half of the blocks, and we will have to look for other blocks to move in our direction for the other half of the block as well.

`push` Again, if we are facing east or west, nothing changes. Otherwise, we will have to push each block in `items-to-move` in the same way we checked for which ones we'll have to move. I'm afraid that this will get ugly.

Let's focus on east and west first, since that is easiest.

```clojure
(defn follow-instructions
  ([grid instructions]
   (follow-instructions grid instructions (first (aoc.util/char-locations grid \@))))
  ([grid instructions location]
   (loop [grid grid
          instructions instructions
          location location]
     (if (empty? instructions)
       grid
       (let [direction (first instructions)
             [grid location]
             (try [(push grid location direction) (move location direction)]
                  (catch Exception e [grid location]))]
```

```
98            (recur grid (drop 1 instructions) location))))))
99
100 (defn push [grid location direction]
101   (case direction
102     (\^ \v) (push-vertical grid location direction)
103     (\< \>) (push-horizontal grid location direction)))
104
105 (defn push-horizontal [grid location direction]
106   (let [spot (move location direction)
107         nextbox (move location direction 2)]
108     (condp = (get-in grid spot)
109       \] (-> grid
110              (push-horizontal nextbox direction)
111              (move-box (move spot direction) (move spot direction 2)))
112       \[ (-> grid
113              (push-horizontal nextbox direction)
114              (move-box spot (move spot direction)))
115       \# (throw (Exception. "Cannot push through a wall."))
116       grid)))
117
118 (defn move-box [grid from to]
119   (let [[fx fy] from
120         [tx ty] to]
121     (-> grid
122         (assoc-in from \.)
123         (assoc-in [fx (inc fy)] \.)
124         (assoc-in to \[)
125         (assoc-in [tx (inc ty)] \]))))
126
127 (defn push-vertical [grid location direction]
128   (let [spot (move location direction)]
129     (condp = (get-in grid spot)
130       \[ (-> grid
131              (push-vertical spot direction)
132              (push-vertical (move spot \>) direction)
133              (move-box spot (move spot direction)))
134       \] (-> grid
135              (push-vertical spot direction)
136              (push-vertical (move spot \<) direction)
137              (move-box (move spot \<) (move (move spot direction) \<)))
138       \# (throw (Exception. "Cannot push through a wall."))
139       grid)))

140 (assert (= 9021 (p2 example)))
```

```
141 (time (p2 input))

    "Elapsed time: 148.803022 msecs"
    1457703
```

16 Day 16

16.1 Part 1

Let's traverse some mazes! We get an input that looks like this:

```
###############
#.......#....E#
#.#.###.#.###.#
#.....#.#...#.#
#.###.#####.#.#
#.#.#.......#.#
#.#.#####.###.#
#...........#.#
###.#.#####.#.#
#...#.....#.#.#
#.#.#.###.#.#.#
#.....#...#.#.#
#.###.#.#.#.#.#
#S..#.....#...#
###############
```

And we need to go from `S` to `E`. We start facing east, and each rotation (clock- or counterclockwise) increases our cost by `1000`; each step by `1`. For the example above, the lowest possible score is **7036**.

I thought of using $A*$, but since a trivial heuristic (Manhattan / Manhattan + 1000 per 90° angle) is likely to be way off, it's probably not that much faster than Dijkstra, so let's use that.

```
(ns aoc.16
  (:require [clojure.data.priority-map :refer [priority-map]]))

(defn dijkstra [grid start])

(defn paths
  "Takes a puzzle input, parses it as a grid, and runs dijkstra on the
  result. Returns what dijkstra returns: a map of {:dist dist :prev
  prev}."
  [grid]
  (let [start (first (aoc.util/char-locations grid \S))]
    (dijkstra grid start)))

(defn p1 [grid paths]
  (let [end (first (aoc.util/char-locations grid \E))
        path (:dist paths)
```

```
17          scores (for [x [0 1 2 3]]
18                   (path {:state end :dir x}))]
19      (apply min scores)))
```

This is more or less part one, but for Dijkstra's algorithm. We use the algorithm from the wiki page, modified so that the cost is appropriately computed: 1 for moving forwards, 1000 for turning clock- or counterclockwise. We use a couple of helper functions, let's go through them one by one.

First, Dijkstra itself. Note that we use as state `{:state start :dir 1}` here, `start` is a `[x y]` vector, `dir` is a direction-the index of `aoc.utils/cardinal-directions`. The goal of Dijkstra is to end up with two maps: `dist` and `prev`, by using the priority queue `q`. Hence, all helper functions are going to take these as input, and return an updated version of them.

```
20  (defn process-vertex [grid q dist prev])
21
22  (defn dijkstra [grid start]
23    (let [start-state {:state start :dir 1}] ; East
24      (loop [q (priority-map start-state 0)
25             dist {start-state 0}
26             prev {start-state #{}}]
27        (if (empty? q)
28          {:dist dist :prev prev}
29          (let [[q dist prev] (process-vertex grid q dist prev)]
30            (recur q dist prev))))))
```

`process-vertex` takes the next vertex from the queue and concatenates the results from `forward-neighbour` and `side-neighbours` to the result. If we find a vertex via a path that is already more expensive than a previously found one, we don't update the maps.

```
31  (defn forward-neighbour [grid q dist prev parent cost])
32  (defn side-neighbours [grid q dist prev parent cost])
33
34  (defn process-vertex [grid q dist prev]
35    (let [[u cost] (peek q)
36          [x y] (:state u)
37          d (:dir u)
38          q (pop q)]
39      (if (< (dist u ##Inf) cost)
40        [q dist prev] ; We already have a cheaper path to node u
41        (let [[q dist prev] (forward-neighbour grid q dist prev u cost)
42              [q dist prev] (side-neighbours grid q dist prev u cost)]
43          [q dist prev]))))
```

`forward-neighbour` updates the `q,dist,prev` maps if we find a new cost that is 1 lower than the existing cost. We use `update-states` to update the maps appropriately, since `side-neighbours` is going to do the exact same things.

Note that `update-states` is slightly different than the standard Dijkstra: instead of `prev` being a map mapping a state to a previous state, we map a state to a set of previous states that have the same cost. Doing this makes Part Two automatic.

```
(defn update-states [new-state new-cost q dist prev parent]
  (let [existing-cost (dist new-state ##Inf)]
    (cond
      (< new-cost existing-cost)
      [(assoc q new-state new-cost)
       (assoc dist new-state new-cost)
       (assoc prev new-state #{parent})]
      (= new-cost existing-cost)
      [q dist
       (update prev new-state (fnil conj #{}) parent)]
      :default
      [q dist prev])))

(defn is-movable? [grid location]
  (let [element (get-in grid location)]
    (cond
      (nil? element) false
      (= \# element) false
      :default true)))

(defn forward-neighbour [grid q dist prev u cost]
  (let [[x y] (:state u)
        d (:dir u)
        new-loc (aoc.util/move [x y] (aoc.util/cardinal-directions d))
        new-state {:state new-loc :dir d}
        new-cost (inc cost)]
    (if (not (is-movable? grid new-loc))
      [q dist prev]
      (update-states new-state new-cost q dist prev u))))
```

And `side-neighbours` is very similar, except that the direction changes and `[x y]` don't and that the cost goes `+ 1000`. We use `reduce` to easily make the same change for left (`(mod (dec d) 4)`) and right (`(mod (inc d) 4)`).

```
(defn side-neighbours [grid q dist prev u cost]
  (let [[x y] (:state u)
```

```
75          d (:dir u)
76          pc (count prev)]
77      (reduce
78       (fn [[q dist prev] nd]
79         (let [new-state {:state [x y] :dir nd}
80               new-cost (+ cost 1000)]
81           (update-states new-state new-cost q dist prev u)))
82       [q dist prev]
83       [(mod (dec d) 4)
84        (mod (inc d) 4)])))
```

Note that we depart slightly from established tradition of calling `(p1 input)` and `(p2 input)`. That is because calling `(dijkstra)` is the biggest time burden, and I don't want to do that for both parts.

```
85  (def exgrid (aoc.util/string-as-lines example))
86  (def expaths (paths exgrid))
87
88  (assert (= 7036 (p1 exgrid expaths)))
89
90  (def input (slurp "inputs/16"))
91  (def grid (aoc.util/string-as-lines input))
92
93  (print "Computing Dijkstra, ")
94  (time (def path (paths grid)))
95
96  (time (p1 grid path))
```

```
Computing Dijkstra, "Elapsed time: 1153.071665 msecs"
"Elapsed time: 0.240615 msecs"
85420
```

16.2 Part 2

Now we need to find all items that lie on an optimal path. Since we kept track of all of these in `prev`, that is very simple: we just need to reconstruct the paths that can end up at the end state (in any direction), and count the distinct elements on that path.

```
97  (defn reconstruct-path [prev state]
98    (let [prev-states (prev state)]
99      (if (empty? prev-states)
100       [[state]]
101       (for [prev-state prev-states
```

```
            path (reconstruct-path prev prev-state)]
        (conj path state)))))

(defn p2 [grid paths]
  (let [end (first (aoc.util/char-locations grid \E))
        prev (:prev paths)
        all-paths (for [x [0 1 2 3]]
                (reconstruct-path prev {:state end :dir x}))]
    (->> (flatten all-paths)
         (map :state)
         distinct
         count)))

(assert (= 45 (p2 exgrid expaths)))
(time (p2 grid path))
```

```
"Elapsed time: 1260.567037 msecs"
492
```

17 Day 17

17.1 Part 1

We get to implement a basic computer, with 8 different *operations* and 3 *registers*. I'll cite the first two here:

> The adv instruction (opcode 0) performs division. The numerator is the value in the A register. The denominator is found by raising 2 to the power of the instruction's combo operand. (So, an operand of 2 would divide A by 4 (2^2); an operand of 5 would divide A by 2^B.) The result of the division operation is truncated to an integer and then written to the A register.
>
> The bxl instruction (opcode 1) calculates the bitwise XOR of register B and the instruction's literal operand, then stores the result in register B.

You might notice (I didn't at first) that the division is a simple bitshift, that'll come in handy. The testinput looks like this:

```
Register A: 729
Register B: 0
Register C: 0

Program: 0,1,5,4,3,0
```

With the program being a list of `operation,operand` codes. There's some magic with the *operand* going on: some operations take a *literal operand* (see `bxl` above), and some a *combo operand* (see `adv` above). The *combo operand* looks like this:

- Combo operands 0 through 3 represent literal values 0 through 3.
- Combo operand 4 represents the value of register A.
- Combo operand 5 represents the value of register B.
- Combo operand 6 represents the value of register C.
- Combo operand 7 is reserved and will not appear in valid programs.

A quick look at the testinput and the general input shows us that registers `B` and `C` are always zero, so we can create a function named `computer` that takes as input the value of register `A` and the programlist. It'll be a loop through the operations. It can't be a clean `reduce`, since we have a `jnz` operation that jumps around by modifying the instruction pointer.

As for our puzzle result: we have an `out` operation that outputs a number. The output format of our puzzle should be those numbers joined by a comma. Let's get hacking:

```
(ns aoc.17
  (:require [clojure.string :as str]))

(defn parse-input [puzzle])

(defn computer [a program])

(defn p1 [input]
  (let [[a, program] (parse-input input)
        output (computer a program)]
    (str/join "," output)))
```

And `computer` is the banger. Check the puzzle instructions for what all operations do. Note that they can be seen as bitshifts rather than divisions. The `combo` map takes care of the combo operator logic.

```
(defn computer [a program]
  (loop [i 0
         a a
         b 0
         c 0
         output []]
    (if (>= i (count program))
      output
      (let [combo {0 0, 1 1, 2 2, 3 3, 4 a, 5 b, 6 c}
            [opcode op] (nth program i)]
        (condp = opcode
          0 (recur (inc i) (bit-shift-right a (combo op)) b c output)
          1 (recur (inc i) a (bit-xor b op) c output)
          2 (recur (inc i) a (mod (combo op) 8) c output)
          3 (recur (if (zero? a) (inc i) op) a b c output)
          4 (recur (inc i) a (bit-xor b c) c output)
          5 (recur (inc i) a b c (conj output (mod (combo op) 8)))
          6 (recur (inc i) a (bit-shift-right a (combo op)) c output)
          7 (recur (inc i) a b (bit-shift-right a (combo op)) output))))))
```

Finally, `parse-input` is perhaps a bit ugly, but since we only care about register `A`, we just look for that and ignore the rest. We output a vector with element 1 being the `A` register, and element 2 is the program vector, which looks like this: `[(0 1) (5 4) (3 0)]`: `(operation operand)`-pairs.

```clojure
(defn parse-input [puzzle]
  (let [[registers program] (str/split puzzle #"\n\n")
        rega (last (re-find #"Register A: (\d+)" registers))
        program (last (re-find #"Program: ([\d,]+)" program))]
    [(Integer/parseInt rega)
     (->> (str/split program #",")
                      (map read-string)
                      (partition 2)
                      vec)]))

(assert (= "4,6,3,5,6,3,5,2,1,0" (p1 example)))
(def input (slurp "inputs/17"))
(time (p1 input))
```

```
"Elapsed time: 0.367492 msecs"
"1,6,7,4,3,0,5,0,6"
```

17.2 Part 2

Now, we need to find out what input register makes the computer return an output that is equal to the program itself. I like it, strange loops vibes. For example, the program below:

```
Register A: 2024
Register B: 0
Register C: 0

Program: 0,3,5,4,3,0
```

Will output `0,3,5,4,3,0` if the input register is **117440**. We need to find the *lowest register value* for which the computer outputs its own program. We could brute-force this, but I tried that for a couple of minutes and have a feeling that we're going to have do something more intelligent.

If you take a look at the example input, you'll see that the program does this:

1. bit-shift `A` left by 3 bits (divide by 8).
2. output `A % 8`.
3. if `A > 0`, go back to step 1.

This means that we can try a number, and see what the program outputs. If we initialize `A` to something below `8`, the output is of course `0` (since it's bit-shifted to the left by 3 bits). If `A` is `8`, the output is `[1 0]`. Look for a minute at the code below:

```
(defn example-program [a]
  (str "Register A: " a "

Program: 0,3,5,4,3,0"))

(for [x (range 0 208 8)]
  (println [x (p1 (example-program x))]))
```

```
[0 0]
[8 1,0]
[16 2,0]
[24 3,0]
[32 4,0]
[40 5,0]
[48 6,0]
[56 7,0]
[64 0,1,0]
[72 1,1,0]
[80 2,1,0]
[88 3,1,0]
[96 4,1,0]
[104 5,1,0]
[112 6,1,0]
[120 7,1,0]
[128 0,2,0]
[136 1,2,0]
[144 2,2,0]
[152 3,2,0]
[160 4,2,0]
[168 5,2,0]
[176 6,2,0]
[184 7,2,0]
[192 0,3,0]
[200 1,3,0]
```

Now that's interesting! Looking at the example above, we see that we don't have to try all possible register values, but we can do something more intellingent:

Start with the number `res` (initially zero) and run it in the program. If the program result is not equal to the last element of our target program, increment it by `1` and try again. If it is equal, save `res` and continue with the next element. To do so, bit-shift `res` to the left by 3 bits (or multiply by 8): save `res * 8`. Now do the same for the second-to-last element: try `res * 8` in our program: if not equal: try `res * 8 + 1`, if equal: bit-shift again: save `res * 8 * 8`. Continue until we have our desired `res`.

In our implementation below, we also bitshift `res` if we find our last element. That's actually one shift too many and therefore means that we have to bitshift back once again when we return `res`.

```
43 (defn p2 [input]
44   (let [[_ program] (parse-input input)
45         goal (flatten program)]
46     (loop [res 0
47            n 0
48            i 0]
49       (if (>= i (count goal))
50         (bit-shift-right res 3)
51         (let [target (take-last (inc i) goal)
52               current (take (inc i) (computer (+ res n) program))]
53           (if (= target current)
54             (recur (* (+ res n) 8) 0 (inc i))
55             (recur res (inc n) i)))))))

56 (assert (= 117440 (p2 example)))
57 (time (p2 input))
```

```
"Elapsed time: 106.573594 msecs"
216148338630253
```

18 Day 18

18.1 Part 1

Puzzle input:

```
5,4
4,2
4,5
3,0
2,1
6,3
2,4
1,5
0,6
3,3
2,6
5,1
1,2
5,5
2,5
6,5
1,4
0,4
6,4
1,1
6,1
1,0
0,5
1,6
2,0
```

The idea is that these are `X,Y` coordinates of walls that fall down in a grid of size `71x71`. We take the first `1024` walls, put them in the grid, and find the shortest path between `[0 0]` and `[70 70]`.

Pretty simple, especially now that we've added a Dijkstra algorithm in Utils. First, let's parse the input and create a grid with walls in the correct spot. When parsing the input, we take care to reverse each element, since the coordinates above are `X,Y`, but we naturally access the grid via `Y,X`.

```
1  (ns aoc.18
2    (:require [clojure.string :as str]))
3
4  (defn parse-input [input]
5    (->> (str/split input #"\n")
```

```
6         (map #(str/split % #","))
7         (map #(map Integer/parseInt %))
8         (map reverse)))
```

Creating the grid is rather straightforward: add a `.` everywhere first, add `#` later.

```
9  (defn create-grid [input size steps]
10   (let [grid (vec (repeat size (vec (repeat size \.))))
11         bytes (take steps input)]
12     (reduce (fn [g pos]
13               (assoc-in g pos \#))
14             grid
15             bytes)))
```

And `p1` is easy with our new Dijkstra:

```
16 (defn helper-dijkstra [input size steps]
17   (-> input
18       (create-grid size steps)
19       aoc.util/grid-to-graph
20       (aoc.util/dijkstra [0 0])))
21
22 (defn p1 [input size steps]
23   (-> input
24       parse-input
25       (helper-dijkstra size steps)
26       (get [(dec size) (dec size)])
27       :weight))

28 (assert (= 22 (p1 example 7 12)))
29 (def input (slurp "inputs/18"))
30 (time (p1 input 71 1024))
```

```
"Elapsed time: 201.257991 msecs"
316
```

18.2 Part 2

We need to find the first element after which we cannot find a route anymore. Binary search is good for this, and I decided to create a generic function that takes as input a sorted list and a predicate; `(pred i)` returns a boolean, and our `binarysearchp` returns the first `i` for which it returns `true`.

There's an off-by-one error related to even/odd input lengths which made the result work on the testinput (len 25) and not on the real input (len 3450).

I fixed it, and then it worked on the input but not on the testinput. My ugly workaround is that when the list has an even length, it instead returns the index of the found element - 1.

```
(defn binsearchp
  "Returns the first index `i` in `list` for which `(pred i)` returns
  true."
  [list pred]
  (loop [start 0
         end (count list)]
    (if (>= start end)
      start
      (let [i (quot (+ start end) 2)]
        (if (pred i)
          (recur (inc start) i)
          (recur i (dec end)))))))
```

This is very handy, but we need to think for a second how to use it. `(p1 input size 1024)` returned the longest route on the grid after `1024` steps. If we input it for some step `i`, it returns `nil` if there is no such route, and finding that first `i` is the objective of Part Two. So, our predicate should return `true` if `(p1 input size i)` returns `nil`, and thus should be `(fn [i] (nil? (p1 input size i)))`.

```
(defn p2 [input size]
  (let [list (parse-input input)
        pred (fn [i] (nil? (p1 input size i)))
        idx (if (even? (count list))
              (binsearchp list pred)
              (dec (binsearchp list pred)))
        [y x] (nth list idx)]
    (str x "," y)))

(assert (= "6,1" (p2 example 7)))
(time (p2 input 71))
```

```
"Elapsed time: 330.052123 msecs"
"45,18"
```

We might optimize this: right now we parse the constant input each time we try out a new element. However, I enjoy running `p1` and `p2` with the raw puzzle input, and that means that `p1` will have to do its own parsing. Since it runs comfortably in less than a second I'm fine by it.

19 Day 19

19.1 Part 1

Let's recap for a minute. Our chief objective is to find the Chief Historian because he's necessary for the big Christmas sleigh launch. He has been missing for months, so it might be something serious. His family is purportedly extremely anxious about his life.

We find ourselves on a hot spring island, and we want to visit an onsen! Unfortunately, they won't accept our money, but they do tell us that if we're able to appropriately arrange their towels, we can enter for free! Let's embark on this important journey.

```
r, wr, b, g, bwu, rb, gb, br

brwrr
bggr
gbbr
rrbgbr
ubwu
bwurrg
brgr
bbrgwb
```

Above is our example puzzle input, starting with a list of towels, and followed by a list of designs. Our objective is to find out how many designs we can create with the given towels.

This lends itself nicely to a recursive solution (given some memoization), so let's do that. First, the bulk of today: `can-be-made?`:

```
(ns aoc.19
  (:require [clojure.string :as str]))

(defn can-be-made? [towels design]
  (if (empty? design)
    true
    (some #(and (str/starts-with? design %)
                (can-be-made? towels (subs design (count %))))
          towels)))

(def can-be-made? (memoize can-be-made?))
```

Quite simple, and this makes the rest of today a piece of cake.

```
(defn parse-input [input]
  (let [[towels designs] (str/split input #"\n\n")]
```

```
    [(str/split towels #", ")
     (str/split designs #"\n")]))

(defn p1 [input]
  (let [[towels designs] (parse-input input)]
    (->> designs
         (filter #(can-be-made? towels %))
         count)))

(assert (= 6 (p1 example)))
(def input (slurp "inputs/19"))
(time (p1 input))
```

```
"Elapsed time: 164.75102 msecs"
330
```

19.2 Part 2

For Part Two, we need to find with *how many possible arrangements* each design can be created. We use a very similar function, except that we don't have the luxury of returning early when `some` of our towel arrangement already filled up the design.

```
(defn count-possibilities [towels design]
  (if (empty? design)
    1
    (->> towels
         (filter #(str/starts-with? design %))
         (map #(count-possibilities towels (subs design (count %))))
         (reduce +))))

(def count-possibilities (memoize count-possibilities))

(defn p2 [input]
  (let [[towels designs] (parse-input input)]
    (->> designs
         (map #(count-possibilities towels %))
         (reduce +))))

(assert (= 16 (p2 example)))
(time (p2 input))
```

```
"Elapsed time: 430.049032 msecs"
950763269786650
```