

# Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 6, 2024

## Contents

<b>1</b>	<b>Day 1</b>	<b>5</b>
1.1	Part 1 . . . . .	5
1.2	Part 2 . . . . .	6
<b>2</b>	<b>Day 2</b>	<b>8</b>
2.1	Part 1 . . . . .	8
2.2	Part 2 . . . . .	9
2.2.1	Benchmark results . . . . .	10
<b>3</b>	<b>Day 3</b>	<b>11</b>
3.1	Part 1 . . . . .	11
3.2	Part 2 . . . . .	11
<b>4</b>	<b>Day 4</b>	<b>13</b>
4.1	Part 1 . . . . .	13
4.2	Part 2 . . . . .	15
<b>5</b>	<b>Day 5</b>	<b>17</b>
5.1	Part 1 . . . . .	17
5.2	Part 2 . . . . .	20
<b>6</b>	<b>Day 6</b>	<b>21</b>
6.1	Part 1 . . . . .	21
6.2	Part 2 . . . . .	23

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.

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

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

Often the lines contain numbers:

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

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.

```
15 (defn num-list-per-line
16   "Returns a vector of vectors, the outer vector has an element per
17   line, the inner has space-separated elements. "
18   [input]
19   (let [lines (string-as-lines input)]
```

```

20      (->> lines
21          (map #(str/split % #"\\s+"))
22          (map #(map read-string %))))))

```

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))
```

# 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`:

```
1  (ns aoc.1)

2  (defn- sum "Finds the sum of a vector of numbers" [vec]
3    (reduce + vec))
4
5  (defn- transpose "Transposes a matrix" [m]
6    (apply mapv vector m))
```

With the final function being now quite easy to follow if you keep my strategy above in mind. Recall that the correct answer for the `testinput` was 11.

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

11

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`.

```
17 (def input (slurp "inputs/1"))
18 (p1 input)

2057374
```

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

## 1.2 Part 2

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

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

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

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

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

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

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

23177084

## 2 Day 2

### 2.1 Part 1

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

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

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

[A] report only counts as safe if both of the following are true:

- The levels are either all increasing or all decreasing.
- Any two adjacent levels differ by at least one and at most three.

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

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

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

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

test it out:

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

[2 3 1 2]
```



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

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

2

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

242

## 2.2 Part 2

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

First I had a smart idea. Check out `e2dcab2f0de76c21477c5e871e029f0282c8fab`. 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.

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

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

4

32 (p2 input)

311
```

### 2.2.1 Benchmark results

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

## 3 Day 3

### 3.1 Part 1

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

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

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

```
1 (ns aoc.3)
2 (defn p1 [input]
3   (let [matches (re-seq #"mul\((\d+),(\d+)\)" input)]
4     (->> matches
5       (map #(list (Integer/parseInt (nth % 1)) (Integer/parseInt (nth % 2))))
6       (map #(apply * %))
7       (reduce +))))
8 (let [input (slurp "inputs/3")]
9   (p1 input))
155955228
```

### 3.2 Part 2

We get a new example string for Part Two:

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

This contains the substrings `don't()` and `do()`, which disable and enable `mul()` instructions. I can do fancy clojure things, but Emacs is way too good for this, so let's do it quickly in Emacs. 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.

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

100189366

## 4 Day 4

### 4.1 Part 1

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

```
MMMSXXMASM
MSAMXMSMSA
AMXSXMAAMM
MSAMASMSMX
XMASAMXAMM
XXAMMXAMA
SMSMSASXSS
SAXAMASAAA
MAMMMXMMM
MXMXAXMASX
```

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

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

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

`is-xmas?` A function that takes the `grid`, a `start` coordinate and a `direction`.

It returns `true` if "**XMAS**" occurs in the `grid` from `start` in the given `direction`.

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

```
1 (ns aoc.4)
2
3 (def directions
4   [[-1 0] ; Up
5    [ 1 0] ; Down
6    [ 0 -1] ; Left
7    [ 0 1] ; Right
8    [-1 -1] ; NW
9    [-1 1] ; NR
10   [ 1 -1] ; SW
11   [ 1 1]] ; SE
```

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

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

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

[3 5]
```

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

```
12 (defn char-locations [grid x]
13   (mapcat
14     (fn [row string]
15       (keep-indexed (fn [col char] (when (= char x) [row col]))
16                     string))
17     (range)
18     grid))
19
20 (defn is-xmas? "Does the grid `grid` contain the string \"XMAS\",
21               starting at `start` and going in `direction`?"
22   [grid start direction]
23   (let [[startx starty] start
         [x' y'] direction]
24     (loop [x startx
25            y starty
26            chars (seq "XMAS")]
27       (if (empty? chars)
28         true
29         (if (not= (get-in grid [x y]) (first chars))
30           false
31           (recur (+ x x')
32                  (+ y y')
33                  (rest chars)))))))
34
```

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.

```

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

```

class clojure.lang.Compiler\$CompilerException

## 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:

```

48 (def diagonal-directions
49   [[-1 -1] ; NW
50    [-1  1] ; NE
51    [ 1 -1] ; SW
52    [ 1  1]] ; SE

```

`is-mas?` is now pretty trivial:

```

53 (defn is-mas? [grid middle direction]
54   (let [[x y] middle
55         [x' y'] direction]
56     (and (= \M (get-in grid [(+ x x') (+ y y')]))
57           (= \S (get-in grid [(- x x') (- y y')])))))

```

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

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

```
class clojure.lang.Compiler$CompilerException
```



## 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now we got everything, ain't we?

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

4637

yes

## 5.2 Part 2

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

```
57 (defn p2 [input]
58   (let [[orderings updates deps] (parse-input input)
59         is-sorted? (partial sorted? deps)
60         sort (partial sort deps)]
61     (-> updates
62         (filter #(not (is-sorted? %)))
63         (map sort)
64         (map middle-num)
65         (reduce +))))
66 (assert (= 123 (p2 example)))
67 (p2 input)
```

6370

## 6 Day 6

### 6.1 Part 1

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

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

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

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

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

Turning clockwise means that we have only four directions:

```
1 (ns aoc.6
2   (:require [clojure.string :as str]))
3 (def directions
4   [[-1 0] ; Up
5    [ 0 1] ; Right
6    [ 1 0] ; Down
7    [ 0 -1]] ; Left
```

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

```

8 (defn guard-route
9   "Takes a `grid` as input returns a vector of 2d coordinates: the route
10  of the guard, starting at `start` and turning clockwise at `\"#\`"
11  characters. "
12  [grid start])
13
14 (defn p1 [input]
15   (let [grid (aoc.util/string-as-lines input)
16         start (first (aoc.4/char-locations grid \^))
17         route (guard-route grid start)]
18     (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.

```

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

```

```

43         (next directions)
44     route))))))

```

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

```

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

```

5208

## 6.2 Part 2

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

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

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

```

48 (defn route-has-loop? [grid start])
49
50 (defn p2 [input]
51   (let [grid (aoc.util/string-as-lines input)
52         start (first (aoc.4/char-locations grid \^))
53         route (disj (set (guard-route grid start)) start)]
54     (->> route
55         (pmap (fn [new-obstacle]
56                 (route-has-loop? (replace-char grid new-obstacle \#) start)))
57         (filter true?)
58         (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.

```

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

```

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

```

80 (assert (= 6 (p2 example)))
81 (p2 input)

```

1972