Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 4, 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]))

(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]
  (let [lines (string-as-lines input)]
    (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.

```
(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]
  (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`:

```
(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))
```

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.

```
(defn p1 [input]
  (let [input (aoc.util/num-list-per-line input)]
    (->> input
         (transpose)
         (map sort)
         (transpose)
         (map #(abs (- (first %) (second %))))
         (sum))))

(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`.

```
(def input (slurp "inputs/1"))
(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.

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

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

```
23177084
```

2 Day 2

2.1 Part 1

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

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

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

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

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

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

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

```
1 (ns aoc.2)

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

test it out:

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

```
[2 3 1 2]
```

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

```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)))

(p1 testinput)
```

```
2
```

```clojure
(def input (slurp "inputs/2"))
(p1 input)
```

```
242
```

2.2 Part 2

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

First I had a smart idea. Check out e2dcab2f0de76c21477c5e871e029f0282c8fabc. It is much more efficient than the current solution, but much more 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)))

(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.

```
(ns aoc.3)

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

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

```
155955228
```

3.2 Part 2

We get a new example string for Part Two:

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

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

- The input file has some newlines, and in some cases a `do()` instruction is on a later line than the previous `don't()` instruction.
- You need to match non-greedy in between a `don't()` and a `do()`.
- If you call `(replace-regexp)` with just the regex and replacement string, it will move point to the last match. This is easily fixed by adding the fourth and fifth arguments to `replace-regexp`: `START` and `END`.

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")]
  (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. In the text below, `XMAS` occurs **18** times.

```
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` occurences 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.

```
(ns aoc.4)

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

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

Well that's the outer logic of the program, but we've used two helper functions that we haven't yet defined:

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

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

Since we're working in the 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:

```
(def directions
  [[-1  0]   ; Up
   [ 1  0]   ; Down
   [ 0 -1]   ; Left
   [ 0  1]   ; Right
   [-1 -1]   ; NW
   [-1  1]   ; NR
   [ 1 -1]   ; SW
   [ 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:

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

(defn is-xmas? "Does the grid `grid` contain the string `text`,
 starting at `start` and going in `direction`?"
  [grid start direction text]
  (let [[x y] start
        [x' y'] direction]
    (loop [row x
           col y
```

```
            chars (seq text)]
      (if (empty? chars)
        true
        (if (or (neg? row) (neg? col)
                (>= row (count grid)) (>= col (count (nth grid row)))
                (not= (get-in grid [row col]) (first chars)))
          false
          (recur (+ row x')
                 (+ col y')
                 (rest chars)))))))
```

Aaaand, run:

```
(assert (= (p1 example)
           18))

(def input (slurp "inputs/4"))
(p1 input)
```

```
2447
```

4.2 Part 2

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

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

We could have reused the functionality above to search for `MAS`-es, and then only count a `MAS` that has a nice diagonal partner sharing the `A`. 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 now that we got an `X-MAS`. Again, let's first create the outer shell of the program:

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

```clojure

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

Note that we're only counting **X**-MAS-es, so only use diagonals:

```clojure
(def diagonal-directions
  [[-1 -1]   ; NW
   [-1  1]   ; NE
   [ 1 -1]   ; SW
   [ 1  1]]) ; SE
```

This only leaves us `is-mas?`, which is pretty trivial. Since every direction is passed, we can just look for the ordered `MAS` string.

```clojure
(defn is-mas? [lines start direction]
  (let [[x y] start
        [x' y'] direction]
    (and (= \M (get-in lines [(+ x x') (+ y y')]))
         (= \S (get-in lines [(- x x') (- y y')])))))

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

```
1868
```