Advent of Code 2024

Rens Oliemans

December 3, 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 code to explain, clarify, justify or otherify something, and these lines aren't necessary to the final solution and thus aren't numbered.

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, tranpose them (so we have two lists), sort them, tranpose them again (pairs), and take the difference and sum it. Since we might require the input as lists of numbers later separately, we can create a function that parses the input and returns pairs of numbers:

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

(defn numbers "Converts the puzzle input into pairs of numbers" [input]
  (let [lines (str/split input #"\n")]
    (->> lines
         (map #(str/split % #" +"))
         (map #(map read-string %)))))
```

Verify that it works:

```clojure
(assert (= '((3 4) (4 3) (2 5) (1 3) (3 9) (3 3))
           (numbers testinput)))
```

Now, I'm going to tranpose these lists, sort them, tranpose them again, take the difference, and sum it. Makes sense? We need the two tiny helper functions `sum` and `tranpose`:

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

(defn- transpose "Tranposes 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.

```clojure
(defn p1 [input]
  (let [input (numbers 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`.

```clojure
(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 naïve 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`!

```clojure
(frequencies (last (transpose (numbers testinput))))
```

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

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

```
(defn p2 [input]
  (let [input (transpose (numbers 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

Analyzing 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. First, we'll need to convert the input to a matrix of numbers, like we did in Day 1. I'm not sure how to best make common functions in literate programming, I have to think about that.

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

(defn numbers "Takes a string and returns a matrix of numbers" [input]
  (let [lines (str/split input #"\n")]
    (->> lines
         (map #(str/split % #" +"))
         (map #(map read-string %)))))
```

Verification:

```clojure
(assert (= (last (numbers testinput))
           '(1 3 6 7 9)))
```

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:

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

(diffs (last (numbers 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]
  (->> (numbers 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]
  (->> (numbers input)
       (filter dampened-is-safe?)
       (count)))

(p2 testinput)
```

```
4
```

```clojure
(p2 input)
```

```
311
```

2.2.1 Benchmark results

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

3 Day 3

3.1 Part 1

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

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

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

```clojure
(ns aoc.3)

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

```clojure
(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 thats 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`.

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

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

```
100189366
```