F#ing Up the Advent of Code 2023

A Deeper Dive into the Challenges

Jindřich Ivánek - jiv@ciklum.com - jindraivanek.hashnode.dev

Solutions: https://github.com/jindraivanek/adventofcode/tree/main/2023/fsharp

Day 2 - parsing with Regex active pattern

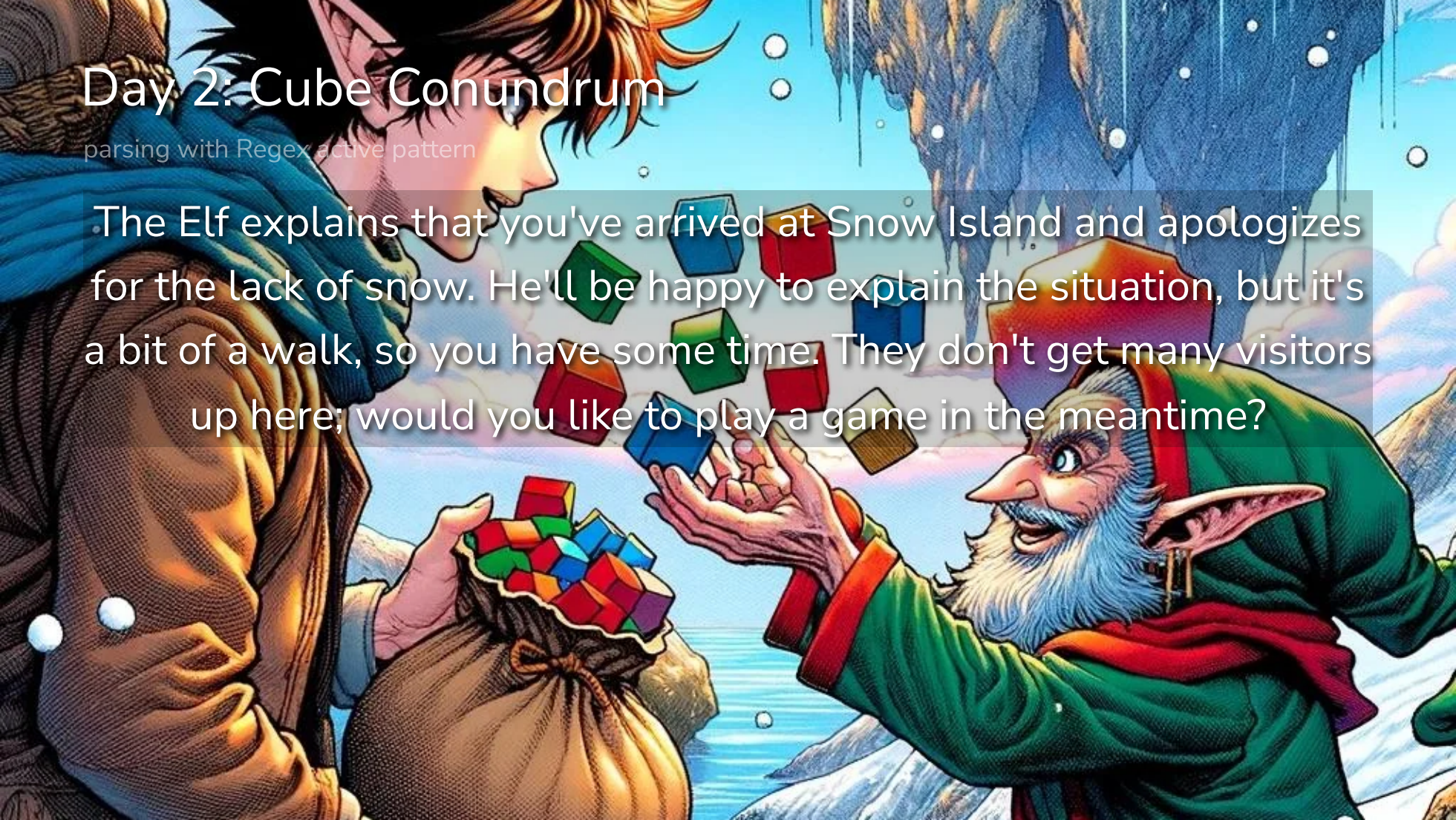
Day 8 - infinite sequences and `Seq.scan`

Day 14 - Moving in grid with `Set`

Day 13 - Grid based problem without indexes

Day 12 - Easy dynamic programming with `memoizeRec`

adventofcode.com



Day 2: Cube Conundrum

parsing with Regex active pattern

The Elf explains that you've arrived at Snow Island and apologizes for the lack of snow. He'll be happy to explain the situation, but it's a bit of a walk, so you have some time. They don't get many visitors up here; would you like to play a game in the meantime?

Part One

Determine which games would have been possible if the bag had been loaded with only 12 red cubes, 13 green cubes, and 14 blue cubes. What is the sum of the IDs of those games?

```
SAMPLE:
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue; 2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1 red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4 red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6 red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red, 2 green
```

```fsharp
type Cube =
    | Red
    | Green
    | Blue

let parse line =
    match line with
    | Match "Game ([0-9]+): (.*)" [ gameNumber; game ] →
        let cubes =
            game.Split([| ','; ';' |])
            ▷ Array.map (function
                | Match "([0-9]+) red" [ x ] → Red, (int x)
                | Match "([0-9]+) green" [ x ] → Green, (int x)
                | Match "([0-9]+) blue" [ x ] → Blue, (int x))
            ▷ Seq.toList

        int gameNumber, cubes
```

```
SAMPLE:
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue; 2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1 red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4 red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6 red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red, 2 green
```

```
let (|Match|_|) (pat: string) (inp: string) =
    let m = Regex.Match(inp, pat) in

    if m.Success then
        Some(List.tail [ for g in m.Groups → g.Value ])
    else
        None
```

```fsharp
let filterGames games =
    games
    ▷ Seq.filter (fun (_, cubes) →
        cubes
        ▷ Seq.exists (function
            | Red, x → x > 12
            | Green, x → x > 13
            | Blue, x → x > 14)
        ▷ not)
    ▷ Seq.map fst

let part1 = lines ▷ Seq.map parse ▷ filterGames ▷ Seq.sum
```

Part Two

The power of a set of cubes is equal to the numbers of red, green, and blue cubes multiplied together. The power of the minimum set of cubes in game 1 is 48. In games 2-5 it was 12, 1560, 630, and 36, respectively. Adding up these five powers produces the sum 2286.

For each game, find the minimum set of cubes that must have been present. What is the sum of the power of these sets?

```
SAMPLE:
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue; 2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1 red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4 red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6 red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red, 2 green
```

```
let gamePower (_, cubes) =
    cubes
    ▷ List.groupBy fst
    ▷ List.map (fun (_, cubes) → cubes ▷ List.map snd ▷ List.max)
    ▷ List.reduce (*)

let part2 = lines ▷ Seq.map parse ▷ Seq.map gamePower ▷ Seq.sum
```

Day 8: Haunted Wasteland

infinite sequences and `Seq.scan`

You're still riding a camel across Desert Island when you spot a sandstorm quickly approaching. When you turn to warn the Elf, she disappears before your eyes! To be fair, she had just finished warning you about ghosts a few minutes ago.

Part One

One of the camel's pouches is labeled "maps" - sure enough, it's full of documents (your puzzle input) about how to navigate the desert. At least, you're pretty sure that's what they are; one of the documents contains a list of left/right instructions, and the rest of the documents seem to describe some kind of network of labeled nodes.

This format defines each node of the network individually. For example:

```
RL

AAA = (BBB, CCC)
BBB = (DDD, EEE)
CCC = (ZZZ, GGG)
DDD = (DDD, DDD)
EEE = (EEE, EEE)
GGG = (GGG, GGG)
ZZZ = (ZZZ, ZZZ)
```

Starting at AAA, follow the left/right instructions. How many steps are required to reach ZZZ?

```fsharp
let instructions = lines[0]

let paths =
    lines
    |> Seq.skip 2
    |> Seq.collect (function
        | Match "(.*) = \((.*), (.*)\)" [ n; l; r ] -> [ (n, 'L'), l; (n, 'R'), r ])
    |> Map.ofSeq

let instrCycle = Seq.initInfinite (fun i -> instructions.[i % instructions.Length])

let makeSteps xs startNode =
    (startNode, xs) |||> Seq.scan (fun n dir -> paths.[n, dir])

let part1 =
    makeSteps instrCycle "AAA" |> Seq.takeWhile (((=) "ZZZ") >> not) |> Seq.length
```

Part Two

Simultaneously start on every node that ends with A. How many steps does it take before you're only on nodes that end with Z?

```
LR

11A = (11B, XXX)
11B = (XXX, 11Z)
11Z = (11B, XXX)
22A = (22B, XXX)
22B = (22C, 22C)
22C = (22Z, 22Z)
22Z = (22B, 22B)
XXX = (XXX, XXX)
```

> Part1 solution doesn't scale, we need to detect cycles of beign on end node, and then use them to calculate the answer.

```fsharp
let findEndCycles xs startNodes endCondition =
    startNodes
    |> List.map (fun n ->
        let ends =
            makeSteps xs n
            |> Seq.indexed
            |> Seq.filter (snd >> endCondition)
            |> Seq.map fst
            |> Seq.take 2
            |> Seq.toList

        let offset = ends.[0]
        let cycleLength = ends.[1] - offset
        assert (offset = cycleLength)
        cycleLength)

let startNodes =
    paths
    |> Map.keys
    |> Seq.map fst
    |> Seq.filter (fun s -> s.EndsWith("A"))
    |> Seq.distinct
    |> Seq.toList

let endIndexes =
    findEndCycles instrCycle startNodes (fun n -> n.EndsWith "Z") |> List.map int64
```

```fsharp
let firstStep = List.min endIndexes

let gcd a b =
    let rec gcd' a b = if b = 0L then a else gcd' b (a % b)
    gcd' (abs a) (abs b)

let part2 =
    // minimum common multiple
    endIndexes ▷ List.fold (fun acc i → (acc * i) / gcd acc i) firstStep
```

Day 14: Parabolic Reflector Dish

Moving in grid with `Set`

The dish is made up of many small mirrors, but while the mirrors themselves are roughly in the shape of a parabolic reflector dish, each individual mirror seems to be pointing in slightly the wrong direction. If the dish is meant to focus light, all it's doing right now is sending it in a vague direction.

Part One

Start by tilting the lever so all of the rocks will slide north as far as they will go:

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

```
OOOO.#.O..
OO..#....#
OO..O##..O
O..#.OO...
........#.
..#....#.#
..O..#.O.O
..O.......
#....###..
#....#....
```

Tilt the platform so that the rounded rocks all roll north. Afterward, what is the total load on the north support beams?

Part Two

The parabolic reflector dish deforms, but not in a way that focuses the beam. To do that, you'll need to move the rocks to the edges of the platform. Fortunately, a button on the side of the control panel labeled "spin cycle" attempts to do just that!

Each cycle tilts the platform four times so that the rounded rocks roll north, then west, then south, then east.

```
After 1 cycle:
.....#....
....# ... O#
 ... OO## ...
.OO#......
.....OOO#.
.O# ... O#.#
....O#....
......OOOO
# ... O### ..
# .. OO#....
```

```
After 2 cycles:
.....#....
....# ... O#
.....## ...
 .. O#......
.....OOO#.
.O# ... O#.#
....O# ... O
.......OOO
# .. OO### ..
#.OOO# ... O
```

```
After 3 cycles:
.....#....
....# ... O#
.....## ...
 .. O#......
.....OOO#.
.O# ... O#.#
....O# ... O
.......OOO
# ... O###.O
#.OOO# ... O
```

Run the spin cycle for 1000000000 cycles. Afterward, what is the total load on the north support beams?

```
let lines = System.IO.File.ReadAllLines("input")

let dirs = [ (0, -1); (-1, 0); (0, 1); (1, 0) ]
let posPlus (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

let indexed =
    lines
    ▷ Seq.mapi (fun i s → s ▷ Seq.mapi (fun j c → (j, i), c))
    ▷ Seq.collect id

let space = indexed ▷ Seq.filter (snd >> (<>) '#') ▷ Seq.map fst ▷ set
let rocks = indexed ▷ Seq.filter (snd >> (=) 'O') ▷ Seq.map fst ▷ set
let cubes = indexed ▷ Seq.filter (snd >> (=) '#') ▷ Seq.map fst ▷ set
```

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

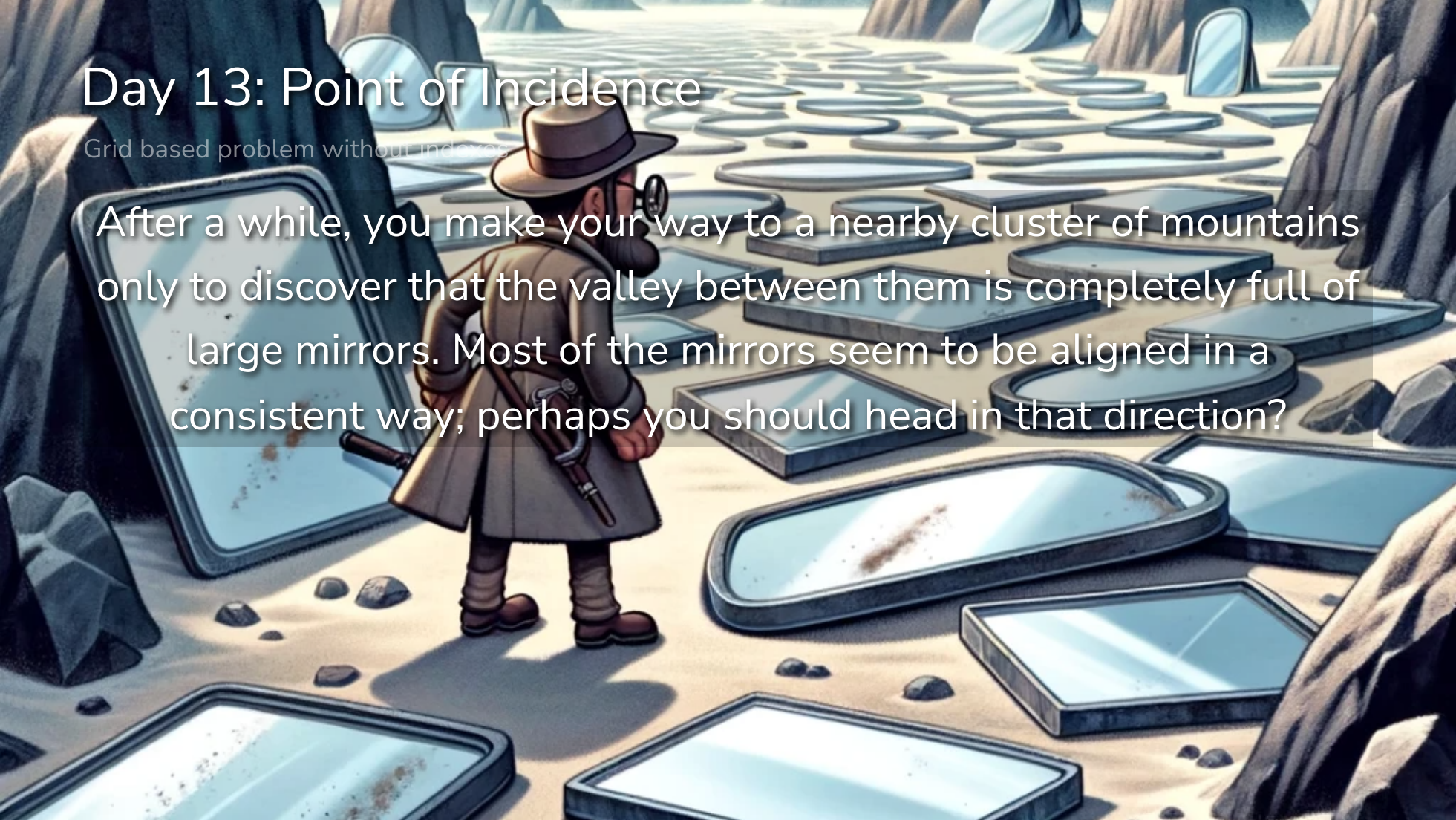
```
let rec slideRock moveDir p rocks =
    let p2 = posPlus p moveDir

    if
        Set.contains p2 space
        && not (Set.contains p2 cubes)
        && not (Set.contains p2 rocks)
    then
        let rocks2 = rocks ▷ Set.remove p ▷ Set.add p2
        slideRock moveDir p2 rocks2
    else
        rocks

let slideAllRocks moveDir rocks =
    let sortFun (x, y) =
        match moveDir with
        | (0, a) → x, y * a * -1
        | (a, 0) → y, x * a * -1
        | _ → failwith "bad move"

    let ps = rocks ▷ Set.toList ▷ List.sortBy sortFun
    (rocks, ps) |▷ List.fold (fun rocks p → slideRock moveDir p rocks)
```

```fsharp
let n = lines.Length

let rocksLoad rocks =
    rocks ▷ Seq.sumBy (fun (_, y) → n - y)

let part1 () =
    let r = slideAllRocks dirs[0] rocks
    //printfn $"%A{r}"
    rocksLoad r
```

```fsharp
let rec slideAllRocksCycle visited dirs rocks =
    let i = Map.count visited
    let visited2 = Map.add rocks i visited
    //printfn $"%A{i} {rocksLoad rocks}"
    //rocks ▷ Set.toList ▷ List.iter (printfn "%A")
    let rocks2 = dirs ▷ List.fold (fun rocks dir → slideAllRocks dir rocks) rocks

    Map.tryFind rocks2 visited
    ▷ Option.map (fun j → j, i - j + 1, visited2)
    ▷ Option.defaultWith (fun () → slideAllRocksCycle visited2 dirs rocks2)

let part2 () =
    let cycleCount = 1000000000
    let offset, cycleLen, history = slideAllRocksCycle Map.empty dirs rocks

    let values =
        history ▷ Map.toList ▷ List.map (fun (r, i) → i, rocksLoad r) ▷ Map.ofList

    let target = offset + (cycleCount - offset) % cycleLen
    values.[target]
```

Day 13: Point of Incidence

Grid based problem without indexes



After a while, you make your way to a nearby cluster of mountains only to discover that the valley between them is completely full of large mirrors. Most of the mirrors seem to be aligned in a consistent way; perhaps you should head in that direction?

Part One

To find the reflection in each pattern, you need to find a perfect reflection across either a horizontal line between two rows or across a vertical line between two columns.

In the first pattern, the reflection is across a vertical line between two columns; arrows on each of the two columns point at the line between the columns:

```
    ><
#.##..##.
..#.##.#.
##......#
##......#
..#.##.#.
..##..##.
#.#.##.#.
    ><
```

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

To summarize your pattern notes, add up the number of columns to the left of each vertical line of reflection; to that, also add 100 multiplied by the number of rows above each horizontal line of reflection.

Find the line of reflection in each of the patterns in your notes. What number do you get after summarizing all of your notes?

Part Two

Upon closer inspection, you discover that every mirror has exactly one smudge: exactly one . or # should be the opposite type.

> (smudge = exactly one error in reflection)

```
1  ..##..##. 1
2  ..#.##.#. 2
3v##......#v3
4^##......#^4
5  ..#.##.#. 5
6  ..##..##. 6
7  #.#.##.#. 7
```

```
1v#...##..#v1
2^#...##..#^2
3  ..##..### 3
4 #####.##. 4
5 #####.##. 5
6  ..##..### 6
7 #....#..# 7
```

In each pattern, fix the smudge and find the different line of reflection. What number do you get after summarizing the new reflection line in each pattern in your notes?

```fsharp
let patterns =
    lines |> Seq.toList |> chunkBy ((<>) "") |> List.map (List.map Seq.toList)

let diffCount xs ys =
    List.zip xs ys |> List.filter (fun (x, y) -> x <> y) |> List.length

let findReflectionWithDiffCount k pattern =
    let sumDiffs xs ys =
        List.zip xs ys |> List.sumBy (fun (x, y) -> diffCount x y)

    let rec go before after =
        let n = min (List.length before) (List.length after)

        if n > 0 && sumDiffs (List.take n before) (List.take n after) = k then
            Some before
        else
            match after with
            | [] -> None
            | x :: after -> go (x :: before) after

    go [] pattern |> Option.map (List.length >> int64)
```

```
let patternScore k pattern =
    let rowReflect = findReflectionWithDiffCount k pattern ▷ Option.defaultValue 0L

    let columnReflect =
        findReflectionWithDiffCount k (List.transpose pattern) ▷ Option.defaultValue 0L

    rowReflect * 100L + columnReflect

let part1 = patterns ▷ List.map (patternScore 0) ▷ List.sum
let part2 = patterns ▷ List.map (patternScore 1) ▷ List.sum
```

Day 12 Hot Springs

Easy dynamic programming with `memoizeRec`

In the giant field just outside, the springs are arranged into rows. For each row, the condition records show every spring and whether it is operational (.) or damaged (#). This is the part of the condition records that is itself damaged; for some springs, it is simply unknown (?) whether the spring is operational or damaged.

Part One

This list always accounts for every damaged spring, and each number is the entire size of its contiguous group (that is, groups are always separated by at least one operational spring: #### would always be 4, never 2,2).

However, the condition records are partially damaged; some of the springs' conditions are actually unknown (?). For example:

```
???.### 1,1,3
.??..??...?##. 1,1,3
?#?#?#?#?#?#?#? 1,3,1,6
????.#...#... 4,1,1
????.######..#####. 1,6,5
?###???????? 3,2,1
```

For each row, count all of the different arrangements of operational and broken springs that meet the given criteria. What is the sum of those counts?

```fsharp
// ???.### 1,1,3
// .?? .. ?? ... ?##. 1,1,3
// ?#?#?#?#?#?#?#? 1,3,1,6
// ????.# ... # ... 4,1,1
// ????.###### .. #####. 1,6,5
// ?###???????? 3,2,1
let combNum xs gs =
    let rec recF xs curG gs =
        match xs, curG, gs with
        // correct combination
        | [], None, []
        | [], Some 0, [] -> 1L
        // '#' part
        | '#' :: xs, Some g, gs
        | '#' :: xs, None, (g :: gs)
        | '?' :: xs, Some g, gs when g > 0 -> recF xs (Some(g - 1)) gs
        // '.' part
        | '.' :: xs, Some 0, gs
        | '.' :: xs, None, gs
        | '?' :: xs, Some 0, gs -> recF xs None gs
        // '?' as '.' at the end
        | '?' :: xs, None, [] -> recF xs None []
        // main recursion - '?' as '.' or '#'
        | '?' :: xs, None, g :: gs -> recF xs (Some(g - 1)) gs + recF xs None (g :: gs)
        // invalid combination
        | _ -> 0L

    recF xs None gs
```

```fsharp
let part1 () =
    parsed ▷ Seq.map (fun (xs, cs) → combNum (Seq.toList xs) cs) ▷ Seq.sum

printfn $"{part1 ()}" // 7191
```

Part Two

To unfold the records, on each row, replace the list of spring conditions with five copies of itself (separated by ?) and replace the list of contiguous groups of damaged springs with five copies of itself (separated by ,).

The first line of the above example

```
???.### 1,1,3
```

would become:

```
???.###????.###????.###????.###????.### 1,1,3,1,1,3,1,1,3,1,1,3,1,1,3
```

Unfold your condition records; what is the new sum of possible arrangement counts?

```
let memoize f =
    let cache = System.Collections.Generic.Dictionary<_, _>()

    fun x ->
        match cache.TryGetValue x with
        | true, v -> v
        | _ ->
            let v = f x
            cache.Add(x, v)
            v
```

```fsharp
let combNumMem xs gs =
    let rec recF =
        memoize // warning FS0040: This and other recursive references to the object(s) being defined will be checked fo
        <| fun (xs, curG, gs) ->
            let recF x curG g = recF (x, curG, g)

            match xs, curG, gs with
            // correct combination
            | [], None, []
            | [], Some 0, [] -> 1L
            // '#' part
            | '#' :: xs, Some g, gs
            | '#' :: xs, None, (g :: gs)
            | '?' :: xs, Some g, gs when g > 0 -> recF xs (Some(g - 1)) gs
            // '.' part
            | '.' :: xs, Some 0, gs
            | '.' :: xs, None, gs
            | '?' :: xs, Some 0, gs -> recF xs None gs
            // '?' as '.' at the end
            | '?' :: xs, None, [] -> recF xs None []
            // main recursion - '?' as '.' or '#'
            | '?' :: xs, None, g :: gs -> recF xs (Some(g - 1)) gs + recF xs None (g :: gs)
            // invalid combination
            | _ -> 0L

    recF (xs, None, gs)
```

```fsharp
let part2 () =
    parsed
    ▷ Seq.map unfold
    ▷ Seq.map (fun (xs, cs) → combNumMem (Seq.toList xs) cs ▷ int64)
    ▷ Seq.sum

printfn $"{part2 ()}" // 6512849198636
```

```
let memoize f =
    let cache = System.Collections.Generic.Dictionary<_, _>()

    fun x ->
        match cache.TryGetValue x with
        | true, v -> v
        | _ ->
            let v = f x
            cache.Add(x, v)
            v
~~~
let memoizeRec f =
    let cache = System.Collections.Generic.Dictionary<_, _>()

    let rec recF x =
        match cache.TryGetValue x with
        | true, v -> v
        | _ ->
            let v = f recF x
            cache.Add(x, v)
            v

    recF
```

```fsharp
let combNumMemRec xs gs =
    let recF =
        memoizeRec
        <| fun recF (xs, curG, gs) ->
            let recF x curG g = recF (x, curG, g)

            match xs, curG, gs with
            // correct combination
            | [], None, []
            | [], Some 0, [] -> 1L
            // '#' part
            | '#' :: xs, Some g, gs
            | '#' :: xs, None, (g :: gs)
            | '?' :: xs, Some g, gs when g > 0 -> recF xs (Some(g - 1)) gs
            // '.' part
            | '.' :: xs, Some 0, gs
            | '.' :: xs, None, gs
            | '?' :: xs, Some 0, gs -> recF xs None gs
            // '?' as '.' at the end
            | '?' :: xs, None, [] -> recF xs None []
            // main recursion - '?' as '.' or '#'
            | '?' :: xs, None, g :: gs -> recF xs (Some(g - 1)) gs + recF xs None (g :: gs)
            // invalid combination
            | _ -> 0L

    recF (xs, None, gs)
```

```fsharp
let part2 () =
    parsed
    ▷ Seq.map unfold
    ▷ Seq.map (fun (xs, cs) → combNumMemRec (Seq.toList xs) cs ▷ int64)
    ▷ Seq.sum

printfn $"{part2 ()}" // 6512849198636
```

Thank you!

Jindřich Ivánek (jiv@ciklum.com)

Blog:

https://jindraivanek.hashnode.dev

Slides

Questions?