Hot springs

Johan Montelius

Spring Term 2024

Introduction

This is the Advent of code for day 12 in 2023. It's a puzzle involving a field of hot springs with unknown condition. The first task is a bit tricky but you should solve it without any problems. The second task is impossible to solve if you don't think things through and use dynamic programming.

1 The record of springs

Imagine that your standing in a large field with rows of hot springs. Some springs are hot, some are cold and some are a biut unknown. The springs are heated by hot lava and should be either operational or damaged but it's not easy to tell. There is however a statis description that will tell you the condition of the springs in a row; your problem is that the description is incomplete of springs. Luckily for you there is some redundancy in the description so even from a incomplete description you should be able to work out at least the probable status of the springs in a row.

A complete description of a row of springs is a string of `#` (damaged) or `.` (operational) followed by a sequence of numbers stating how many consecutive damaged springs we have. A description of a row could look as follows:

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

So here we see that we only have two operational springs, the second and fourth. We have three segments of damaged springs of length: one, one and three. If all descriptions were as clear as this example we would not have a problem but you're now faced with descriptions where the status could be reported as unknown: `?`.

The good thing is that the sequence with numbers that describes the consecutive numbers of damaged springs is always correct. Given an incomplete description you could work out possible correct descriptions. Assume that your give a description that looks as follows:

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

Even though the first three springs are recorded as unknown, there is only one way the status of the springs would generate the sequence: one, one, three. The only possible solution is that the first and third spring are both damaged and that the second spring is operational. Note that the state "##..###" does not match "1,1,3" but "2,3". A consecutive sequence of damaged springs i recorded as one sequence.

If you're given a description that looks like follows:

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

Then the damaged springs could be either: second and sixth or second and seventh or ... All in all you have four different descriptions that match the incomplete description.

This is a small sample that you can work with to make sure that everything works:

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

If you get it right the first row has 1 solution, the second 4, then 1, 1, 4 and 10. The real problem much larger so you need to find a good representation of the problem and a strategy to count the number of possible descriptions - add them all together, row by row and present the grand total.

You first task is to parse the description of the wells and since all rows have the same structure it should be fairly easy. Begin by splitting the string into rows and then split a row into two parts (separated by " "). The description of the status is best first turned into a *charlist* using `String.to_charlist\1` and then turned into something more practical. The sequence of damaged springs is first divided in by splitting the string using "," as a separated. Each number can then be parsed using `Integer.parse\1`

The second task is of course to come up with an algorithm that solves the problem. The solution might be easier than you think, if you think recursively. First build something that checks if a *complete description* is consistent i.e. if the sequence of description of springs matches the sequence of consecutive damaged.

When you have this figured out you start to think what happens when you see a unknown state. The spring could be either operational or damaged so you need to explore both of these options. When you explore an option you should report back if it succeeded, or rather how many possible

interpretations we have. Once you get the recursive thinking in order the solution is surprisingly simple.

2 Oh, no!

The second task at first looks simple because the only thing you have to do is extend the input and then use the same algorithm. You should do this first and report the time it takes to solve the puzzle. Only when you realize that things will take too long time should you continue to solve the puzzle.

2.1 extending a row

The thing is that the descriptions that you have been given are not the full description. The sequences should be repeated five times with a `?` in between. The sample row that we looked at before is actually:

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

You should first implement a function that multiplies the original description n times. Now try to solve the puzzle when multiplying by: 1, 2, 3, etc. How large puzzles can you solve, what is happening with the execution time? Did you have the patient to solve the puzzle for a multiple of four? Give a rough estimate how long time it would take to solve it for a multiple of five.

2.2 dynamic programming to the rescue

This is where dynamic programming comes to our rescue. The dynamic programming solution is simply the same recursive function that you have but extended with a memory to avoid doing the same computations twice (or a hundred times). If you get it right you will not only reduce the execution time but completely change the algorithmic complexity.

The strategy that you should apply is to first extend your function with an extra argument which will be your *memory*. You should also introduce a new function that takes the same arguments and only calls the original function. You then change your first function to call this new function in its recursive call.

So far you have not done anything magical and you should be able to run your program as before. The only difference is that we have the new function as an extra step in each recursion. This is the function that you now should extend so that it first checks if you have already an answer for the query, or if you have to calculate it. If you calculate it then store the query and the answer in the memory and return **both the answer and the updated memory**.

Since the new function returns both the answer and the updated memory you need to change your first function. The answer should of course be used as before but the updated memory should be used in the recursion.

The memory that you use could of course be implemented specifically for this task but you can in this assignment use an Elixir map. You will still have the problem of figuring out what to use as a key etc but the importance here is not how the key value store is implemented.

If everything works you should now be able to call your new function with the problem and an empty map. Now try to solve the puzzle with a multiplier of: 1, 2, 3 etc. If it works do some benchmarking and describe the execution time in relation to the multiplier. How large puzzles can you solve?