Springs II

Dhruba Das

Spring Term 2024

Introduction

This is the second part of the previous assignment, which makes our life easier by throwing dynamic programming into the mix. In this assignment, we are told that each row we get, is in fact much larger once it's unfolded, i.e we are to multiply the each row of springs given to us FIVE times, with a "?" between each sequence of springs.

Without dynamic programming

Life is hard without dynamic programming to rescue us in this part of the assignment; not only from the coding side, but the operation itself also becomes exponentially more expensive as we progress. Nonetheless, shown below is the *getsprings* method (which is essentially the same as the previous assignment) that handles the duplicating of the spring-sequence every time a "?" is encountered after the end of the sequence, using the $List.duplicate/1$ method, and $List.flatten$ method which concatenates all the smaller duplicated lists into one list. This is done for both the list of numbers, and the list of springs:

```
def get_springs(row, mul) do
    [sprs,nums] = String.split(row)
    sprs =  String.graphemes(sprs)
    [_|mulSprings] = ["?"|springs] |> List.duplicate(mul) |> List.flatten

    nums = String.split(nums,",")
    nums = char_to_int(nums)
    mulNums = nums |> List.duplicate(mul) |> List.flatten

    detect(mulSprings, mulNums)
end
```

The original list took $4216ms$ (milliseconds), a multiple of two took $76990ms$, a multiple of three took $2293621ms$ and I did not have the patience to wait

for a multiple of four. We can see that the time taken increases exponentially as the puzzle gets larger.

Dynamic progamming

For the dynamic programming, I went with a completely new approach when it came to the representation of the springs, with [:opr], [:dam] AND [:unk] representing operational, damaged and unknown springs respectively. A tuple of {: $spec, springs, nums$} is used to represent a row, with *springs* representing the list of springs, and *nums* representing the list of numbers:

```
def dynamic(springs, seq, mem) do
        case Map.get(mem, {:spec, springs, seq}) do
          :nil ->
            {n, mem} = solve(springs, seq, mem)
            {n, Map.put(mem, {:spec, springs, seq}, n)}
          n ->
            {n, mem}
        end
end
```

Same as the non-dynamic version, we have methods for handling both the solution of the problem until we encounter a damaged spring, and the methods for handling damaged and unknown springs, called *solve* and *damaged* respectively.

The difference here is that we use **memory** to speed things up. We extend the functions so that they first check if we have already an answer for the query, or if we have to calculate it. If we calculate it, then we store the query and the answer in the memory and return both the answer and the updated memory.

solve methods

The first base case handles the scenario where both the springs and number lists are empty. In this case, there are no more springs and sequences left to process, so it returns {1, mem}. This implies that the solution has been found, and there's one possible configuration of springs that satisfies the given sequence.

The second base case returns {0, mem}, indicating that no valid configuration of springs could satisfy the given sequence. This could happen if there are still springs left, but the sequence is exhausted, meaning there are more springs than required by the sequence.

If the first spring is operational, we continue checking for the consistency of the spring sequence against the list of numbers; if it's unknown, the function recursively explores different configurations of springs and sequences,

considering the possibility that the current unknown spring could be operational or damaged. It uses the *damaged* function to determine the status of the current spring and adjusts the recursive calls accordingly.

```
def solve([:dam|springs], [ok|seq], mem) do
        case damaged(springs, ok-1) do
          {:ok, springs} ->
              dynamic(springs, seq, mem)
          :no ->
              {0, mem}
        end
end
...
def solve([:unk|springs], [ok|rest]=seq, mem) do
        {n, mem} = dynamic(springs, seq, mem)
        case damaged(springs, ok-1) do
            {:ok, springs} ->
              {alt, mem} = dynamic(springs, rest, mem)
              {alt+n, mem}
            :no ->
              {n, mem}
        end
end
def solve([:unk|springs], []) do
        solve(springs, [])
end
```

damaged methods

The *damaged* functions recursively navigate through the list of springs to determine their status (operational, unknown, or damaged) at a given position. They return {:ok, ...} if the spring is operational or unknown and :no if the spring is damaged or if the requested position is invalid.

There are THREE base cases: the first one handles the case where the springs list is empty and the position 0 is requested. Since there are no springs left, it returns {:ok, []}, indicating that there are no more springs to process. The second one handles the case where the first spring in the springs list is operational and the position 0 is requested. It returns {:ok, springs}, indicating that the first spring is operational and the remaining springs are unchanged. The third one handles the case where the first spring in the springs list is unknown and the position 0 is requested. It returns {:ok, [:opr—rest]}, indicating that the first spring is assumed to be operational while the rest of the springs remain unchanged.

We then recursively handle the case where the first spring in the springs list is unknown or damaged and the position n (where n ¿ 0) is requested:

```
def damaged([:unk|rest], n) when n > 0 do
        damaged(rest, n-1)
end
def damaged([:dam|rest], n) when n > 0 do
        damaged(rest, n - 1)
end

def damaged(_,_) do :no end
```

The last clause acts as a catch-all for any other cases not handled by the previous clauses. It returns :no, indicating that the requested position is not found or is invalid for the given input.

Benchmarking

Testing the dynamic was not too difficult, as I didn't have to wait nearly as much as I did for the non-dynamic implementation. I tested both using the actual rows of springs, i.e a 1000 rows. Shown below are the times I obtained, as well as the multiples of the puzzle used:

Multiple (n)	**Time taken (ms)**
1	25700
2	65200
3	106000
4	172000
5	287000

Table 1: Time taken for 1000 rows; time in milliseconds

Conclusion

While the non-dynamic implementation is barely able to get past the multiple of three, the dynamic implementation has no difficulty to get past even the fifth multiple; this tells us how much more efficient dynamic programming is, as we are able to not only improve on, but drastically change the algebraic time complexity of the operations.