Advent Of Code

Azer Hojlas

February 2022

Introduction

This assignment is set in a virtual scenario where the protagonist, Santa Claus in first person, is deprived of his sleigh and is forced to conduct festive operations by utilizing a submarine. Christmas can be saved with the help of the latest technology in sonar systems, one must however first determine how quickly the depth increases. To do this, one must measure the depth increases from measurement to measurement. How this is achieved will be discussed below.

Benchmark

The author saw no use for a benchmark as it seemed that no measure of runtime was necessary. The student has however tested both the teachers input, test input and the input generated by the website for each specific user, i.e the writer in this case. As a verification of successful code implementation, the results are posted below:

Figure 1: Proof of part two working

Figure 2: Proof of part one working

Here is a screenshot of running the code in a terminal, where both the teachers and the authors input are used:

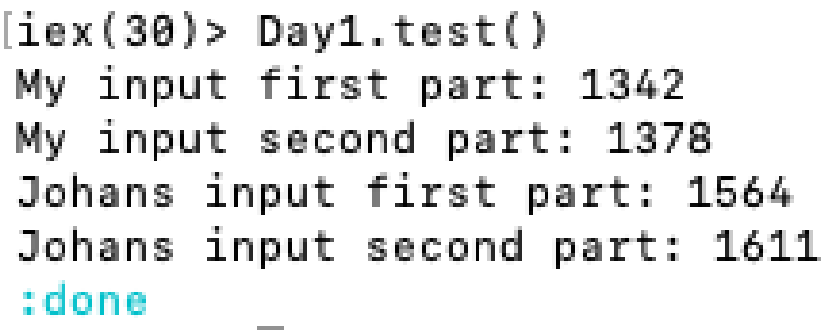

Figure 3: Proof of johans input working on both parts

Code in-depth

The first implementation was, with regards to assignment requirements, rather long (about 50 lines of code). As a fun exercise, the author managed to get the relevant code down to seventeen lines. All lines are discussed below.

First Part

The first part is straight-forward. In order to measure depth increases, all elements in the list will be compared to their neighbours next to them (index incremented by one). If a neighbour is larger, it will count towards a depth-increase. The first function clause, compare() with arity one, receives the desired list to perform operations on and forwards it to the next tail recursive function clause (compare()/2). The base case returns the acc variable, i.e the amount of depth increases once all elements in the list have been processed. The recursive clause compares the current head with the second head of the list. If the current is smaller than the second, accumulator increments and a list consisting of next and tail are passed on recursively to the same function. If the current is not smaller, the same function call will occur, only differing in that the accumulator will not increment. This is repeated until the tail is empty.

```
def compare(lst) do compare(lst, 0) end

def compare([_ | []], acc) do acc end
def compare([current, next | tail], acc) do
    case current < next do
      true -> compare([next | tail], acc + 1)
      false -> compare([next | tail], acc)
    end
end
```

Second Part

The second part is almost identical to the first, this part however requires that successive sums of three elements are compared to each other. This part utilizes the same function as above, however, one more function is used which creates a list of the aforementioned three-element sums. The initial function sumsOfThree() takes input and passes it along to the tail recursive clause, where the base case returns an accumulator (the list of sums) if sumsOfThree() has iterated to the point where there are only two elements left, thus being unable to create another sum of threes. The recursive clause simply takes the first three elements of the original list and appends their sum to the accumulator. The same list is the passed along to sumsOfThree()/2, however this time without the first head, thus creating a new sum of threes. If not made clear earlier, sumsOfThree() is used as input in the original compare().

```
  def second_compare(lst) do compare(sumsOfThree(lst)) end

  def sumsOfThree(lst) do sumsOfThree(lst, []) end
  def sumsOfThree([_, _| []], acc) do acc end
  def sumsOfThree([first, second, third | tail], acc) do
  sumsOfThree([second, third | tail], acc ++ [first + second + third])
  end
```

Conclusion

This assignment was a fun exercise in shortening code. The following test code was used to showcase the results. No explanation will be given as it is straightforward:

```
def test() do
    myInput = Lst.myinput()
    IO.write("My input first part: #{compare(myInput)}\n")
    IO.write("My input second part: #{second_compare(myInput)}\n")

    johansInput = Lst.input()
    IO.write("Johans input first part: #{compare(johansInput)}\n")
```

```
    IO.write("Johans input second part: #{second_compare(johansInput)}\n")
    :done
end
```

Lastly, here is the result from executing the test()(example()) scenario that was included in the teachers code.

```
iex(37)> Day1.compare(Lst.test())
7
iex(38)> Day1.second_compare(Lst.test())
5
```

All in all it was a fun exercise. What might be worth adding is discussing why the second part is a more useful implementation than the first, to quote the assignment instructions: "Considering every single measurement isn't as useful as you expected: there's just too much noise in the data. Instead, consider sums of a three-measurement sliding window". By using sums of threes, one implements a rolling average where highly diverging values have a lesser impact on said average, thus being able to discern depth-increases more correctly and definitely, resulting in a smoother curve or table if it was to be plotted. The second part is also slower, as it requires additional steps to process the original list.