Task 8 - Advent of Code

Casper Kristiansson

February 11, 2022

Introduction

Task eight of the course Programming II consisted of solving two different tasks of the "Advent of code". Advent of code is a website which provides different programming tasks everyday of December until Christmas. The website users can compete to solve different type of tasks as fast as possible. The author will be solving first two days of "2021 advent of code".

Result

The first day exercises that the author completed involved in creating an algorithm that detects increase of depth in the ocean floor. The program has access to an extensive list of entries and the goal is to detect and count how often the depth of the ocean floor is increased. The second part of the report involves the second day of advent of code which consisted of calculating the final position after navigating either forward, up, or down with a submarine.

Input

The instructions of the problem consisted of two different inputs, one example input which consisted of around ten different values and one which consisted of thousands of various values. Because the website Advent of code generates different type of data to each participate the author decided to build a function which reads from a text file and converts the input to a list in elixir. Figure one consists of reading the text file and converting the string to a list of integers.

```
def readData() do
  file = File.read!("lib/data.txt")
  String.split(file, "\r\n")
  |> Enum.map(&String.to_integer/1)
end
```

Figure 1: Example input

Task One

Task one involves counting the number of times the ocean floor depth is increased. This could easily be done by checking if the previous value is larger than the current value. For the first value the program will just ignore as no previous value exists. Figure two consists of an example input and the result of that input is three because in three different situations the depth has increased.

```
199 (N/A - no previous measurement)
200 (increased)
199 (decreased)
210 (increased)
200 (decreased)
263 (increased)
```

Figure 2: Example input

Figure three consists of the solution to task one. The solution starts of by calling the function **measurementsIncreasing** with three different arguments: data, the first element and 0 (number of increases). The function will recursively navigate through the list and check if the current value is bigger than the previous value. If the value is bigger, the amount will be increased by one. When the list has reached the end, the function will return the amount.

```
def measurementsIncreasing([], _, result) do result end
def measurementsIncreasing([head | tail], previous, result) do
  if head > previous do
    measurementsIncreasing(tail, head, result + 1)
  else
    measurementsIncreasing(tail, head, result)
  end
end
```

Figure 3: Recursively navigating a list to count increases

Task two

Because the task does not really register an accurate result in the depth increasing if one value just spikes. A solution to this is to count the sum of three inputs in a row and compare it the three next inputs. By doing this the function will create a sliding window of three inputs. The function builds on the same structure as task one, but it constantly compares the sum of three values instead one.

```
def getThreeValue([], _, _) do :nil end
def getThreeValue([head | tail], value, sum) do
  if value < 3 do
    getThreeValue(tail, value + 1, sum + head)
  else
    sum
  end
end

def measureSlidingWindow([], _, result) do result end
def measureSlidingWindow([head | tail], value, result) do
  newValue = getThreeValue([head | tail], 0, 0)
  cond do
    newValue == :nil ->
      result
    newValue > value ->
      measureSlidingWindow(tail, newValue, result + 1)
    true ->
      measureSlidingWindow(tail, newValue, result)
  end
end
```

Figure 4: Sliding window of three values to detect increase of depth

The function **getThreeValue** will recursively fetch three values in a row of the list by keeping track using a counter. The function **measureSliding-Window** will than compare the sum of those three values to the previous value.

Day Two

The second day of Advent of code consisted of calculating the final position of a coordinate system. The author thought that the hardest part of the task was to implement the function for reading the input. The data had the following format "forward 10" where the first argument is the direction, and the second argument is the step length. The author solved this by converting the input to two different lists, one for the direction and one for the step distance.

```
def readCoordinates() do
  file = File.read!("lib/DataCoordinates.txt")
  list = String.split(file, "\r\n")
  extractData(list, [], [])
end

def extractData([], directions, steps) do {directions, steps} end
def extractData([head | tail], directions, steps) do
  [direction, step] = String.split(head, " ")
  extractData(tail, directions ++ [direction],
                    steps ++ [String.to_integer(step)])
end
```

Figure 5: Input for day two

The author solved the task by simply checking the direction type and either increasing or decreasing the different coordinates.

```
def coordinates({[], []}, x, y) do x * y end
def coordinates({[direction | directions], [step | steps]}, x, y) do
  case direction do
    "forward" ->
      coordinates({directions, steps}, x + step, y)
    "down" ->
      coordinates({directions, steps}, x, y + step)
    "up" ->
      coordinates({directions, steps}, x, y - step)
  end
end
```

Figure 6: Input for day two

Part two of the advent of code consisted of keeping track of a third variable which was the aim. Instead of keeping track of the y coordinate the aim of the submarines would either increase or decrease when it either moves up or down. When the submarine would move forward the depth would be increased by the product of the step and current aim amount. This code will not be shown due to limit of space.