Advent of Code: Day 1

Olivia Denbu Vilhelmsson

14 february - 2023

1 Introduction

In this assignment I have learned better how to work on enumerable and also to reflect on in what order to perform different operations on it. It was interesting to combine different concepts we have discussed from previously and understand certain details better now in how Elixir works.

I choose to work on the Day 1 problem which demanded me to create a program with takes a text file as an input and to operate on the content from the file. The file should contain groups of integer numbers (one number is one line in the text file), and each group is divided by a bland row in between.

Modules

I have understood that the module Enum in Elixir can be a trap if the input of enumerable is infinite. In this case the list of input values was not infinit, so I could use this instead of Stream.

Parse a String to Integer

Since the text file is consisting of string values then every line had to be processed. Integer.parse() returns a tuple of the integer value from the string and then the rest. This was solved by creating a tuple to catch the actual integer.The exception is when you send in a value to this function that is not an integer in the beginning. Then error is returned.

```
{value, _} = Integer.parse(currentLine)
```

Reduce and Pattern Matching

The accumulator is a list which is containing each sum of each section of integer values added together. The reduce functions second parameter is the init value of the accumulator. At first I had it set to 0, but then I made it into a list with the element of zero. The reason that I had to place zero as the intial value, and not just leave it as an empty list was because when the first number should be added there has to exist a head to refer to.

To make the sum reset to zero every time when a subset of addition had been completed then I had to set the head to zero and the tail as the accumulator. It is not possible to make pattern matching to my first number to an empty list. But if I add the zero into the empty list as the init then the pattern matching is possible since the zero now is the head.

```
defp buildAccumulator(lines) do
    Enum.reduce(lines, [0], fn currentLine, acc = [head|tail] ->
     if(currentLine == "") do
       [0|acc]
     else
       {value, _ } = Integer.parse(currentLine)
       [head + value|tail]
     end
   end)
end
```

LIFO

One thing I understood better is how this accumulator, a list that is growing dynamically has the same architechture as the LIFO data structure. I noticed this from seeing that it is the last group of lines from the input file that are being printed out first when the whole file has been read. It is good that it is at the top of the list we are inserting the finalized sums because it is very bad performance when a function is forced to go through elements of a list.

2 Part 1: Finding the maximum value stored in the list

At first I was thinking that I was going to find the max value stored into the list after the whole accumulator was build. I was thinking to create functionality that went through the list and compared each element to the currently stored maximum value. Then I realised that this would not be so efficient since I have to go all the way up through the list again, so it would not be tail recursive. When I realised this then I thought it will be better if I for every time when I am coming to ”a new line session”, that is a new sum is to be added to the accumulator then I should find a way to compare this sum to the currently stored maximum. Because then I will not have any operations to perform once the list/accumulator is finalized.

I though of a beginning to this solution in code, but then I realised there was a part two to the assignment and I had it to work with the built in function Enum.max() so then I decided to have the solution that goes through the whole list at the end anyways.

3 Part 2: Top three biggest elements

When I solved the second part of the task I realised that if I sort the list first then it is very easy to get the three biggest elements. It would have given better performance to keep the accumulator sorted in parallel when it was being created, since then I would not have needed to do this operation in the end.