LAB 03 - Roshan Poudel

PART 1

1.

```
iex(1)> File.read!("/usr/share/dict/words") |> String.split |> Enum.max_by(&String.length/1)
"antidisestablishmentarianism"
```

2.

```
iex(2)> File.stream!("/usr/share/dict/words") |> Enum.max_by(&String.length/1)
"antidisestablishmentarianism\n"
```

3.

```
iex(3)> Enum.map(1..10_000_000, &(&1+1)) |> Enum.take(5)
[2, 3, 4, 5, 6]

#This took about 2 seconds to finish

Stream.map(1..10_000_000, &(&1+1)) |> Enum.take(5)
[2, 3, 4, 5, 6]

#This outputs the result in an instant
```

In this case, using stream is much less memory-intensive and efficient since stream only increments the range when the Enum.take() demands it hence lazily evalutaing it. On the other hand, the enum map will first generate the entire range and stores it in the memory, then the Enum.take() will operate on it after hence eager evaluation.

4.

```
iex(6)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> Enum.map(fn {class, value} ->
    <tr class='green'><td>1</td></tr>
    <tr class='white'><td>2</td></tr>
    <tr class='green'><td>3</td></tr>
    <tr class='white'><td>4</td></tr>
    <tr class='green'><td>5</td></tr>

:ok

iex(11)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 2)
[{"green", 1}, {"white", 2}]
iex(12)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 3)
[{"green", 1}, {"white", 2}, {"green", 3}]
iex(13)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 4)
[{"green", 1}, {"white", 2}, {"green", 3}, {"white", 4}]
iex(14)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 5)
[{"green", 1}, {"white", 2}, {"green", 3}, {"white", 4}, {"green", 5}]
```

```
iex(15)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 6)
[{"green", 1}, {"white", 2}, {"green", 3}, {"white", 4}, {"green", 5}]
iex(16)> Stream.cycle(~w{ green white }) |> Stream.zip(1..5) |> (Enum.take 7)
[{"green", 1}, {"white", 2}, {"green", 3}, {"white", 4}, {"green", 5}]
```

Taking more than the 5 will cause to just return the output upto 5. This happens because: Stream.zip/2 stops when the shortest stream ends. In this case, the 1..5 range is a finite stream with only 5 elements. Stream.cycle(~w{ green white }) is an infinite stream of alternating "green" and "white", but Stream.zip/2 will stop when the shorter stream (the range 1..5) ends.

PART 2

1.

```
Stream.unfold({0,1}, fn {f1,f2} -> {f1, {f2, f1+f2}} end) |> Enum.take(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

This works because the stream unfold takes an accumulator which is {0,1} and then the function acts on the accumulator returning the tuple of the current value and next accumulator and the stream continues to infinity in this case becuase there is no condition when the function return nil. Hence, we take the first 15 fibonacci numbers

2.

```
iex(10)> c "countdown.exs"
iex(11)> counter = Countdown.timer
#Function<54.38948127/2 in Stream.resource/3>
iex(12)> printer = counter |> Stream.each(&IO.puts/1)
#Stream<[
  enum: #Function<54.38948127/2 in Stream.resource/3>,
  funs: [#Function<39.38948127/1 in Stream.each/2>]
]>
iex(13)> printer |> Enum.take(5)
57
56
55
54
53
["57", "56", "55", "54", "53"]
iex(14)> printer |> Enum.take(5)
50
49
48
47
46
```

```
["50", "49", "48", "47", "46"]
iex(15)> printer |> Enum.take(5)
41
40
39
38
37
["41", "40", "39", "38", "37"]
iex(16)> speaker = printer |> Stream.each(&Countdown.say/1)
#Stream<[
  enum: #Function<54.38948127/2 in Stream.resource/3>,
  funs: [#Function<39.38948127/1 in Stream.each/2>,
   #Function<39.38948127/1 in Stream.each/2>]
]>
iex(17)> speaker |> Enum.take(5)
19
18
17
16
15
["19", "18", "17", "16", "15"]
iex(18)> speaker |> Enum.take(5)
7
6
5
4
3
["7", "6", "5", "4", "3"]
iex(19)>
```

This speaker thing sounds like its about to detonate