Primes

Lucas Larsson

Spring Term 2022

Description

As per usual no introduction is needed, only a description of the solution and a discussion of the results.

The primes were calculated according to the assignment instructions i.e using `Sieve of Eratosthenes` Algorithm.

First

The first algorithm was pretty straightforward to implement, due to clear instruction.

A list is generated using the `Enum.to_list()/2` , this is for all three functions not only the first.

A method `rem/2` is used to determine if a number is divisible by a prime, i.e if the number is a prime. And a `filter/2` method is used to remove non primes from the list. More in depth explanation is provided below with example.

Second

The second method is similar to the first, with the difference of:

- Using an additional list to store the primes.
- Not removing numbers from the original list.

Third

The third method is nearly identical to the second method with the distinction of different inserting method to the additional list.

Results

The first function is the same in all three methods, it is a built in function that generates integers between lower and upper bound.

A function then called recursively `Filter/2` , Filter goes through the list and remove numbers divisible by the head, and then moves to the second element. that is also a prime because it was not removed in the first round and so on until it has removed all non prime numbers from the list.

```
def prime(n) do
  [h|t] = Enum.to_list(2..n)
  [h | filter(h,t)]
end

def filter(_, []) do [] end
def filter(x,[h|t]) do
  [h|filter(h, Enum.filter(t, fn p -> rem(p, x) != 0 end))]
end
```

The second function takes in two lists as parameters, one containing the list of numbers `2..n` and an empty list to store the primes in.

It first checks if the number is a prime using the same built in function used in the first method, but without filtering/removing elements from the list. If a number is a Prime it is inserted to the list. As shown below.

```
def checkPrime(_, []) do true end
def checkPrime([h|t], [h2|t2]) do
  cond do  (rem(h,h2) == 0) ->
      false
      true -> checkPrime([h|t], t2)
  end
end

def insert(bool, x, primes) do
  case bool do
    true -> primes ++ [x]
    false -> primes
  end
end
```

The third implementation is nearly identical to the second so no code is shown to explain it, the main difference between them is that the prime element is inserted at the beginning of the list instead of the end of it as demonstrated above in the second method.

```
def insert(bool, x, primes) do
   case bool do
     true ->  [x|primes]
     false -> primes
   end
 end
```

Down below a table showing run time of the different algorithms.

N	Function [ms]		
	`First`	`Second`	`Third`
2^7	0.12	0.11	99
2^8	0.33	0.17	0.27
2^9	0.61	0.60	0.90
2^{10}	1.8	1.7	3.3
2^{11}	9.5	5.1	12
2^{12}	20	12	45
2^{13}	74	34	169
2^{14}	0.2 sec	0.09 sec	0.6 sec
2^{15}	0.7 sec	0.3 sec	2 sec
2^{16}	2.9 sec	1.2 sec	10 sec

Table 1: A Table showing run time

discussion

As indicated above the algorithm ranked from fastest to slowest are Second, First, Third.

Reason behind the first being slow compered to the second is that the first need to create a new list in each iteration when it removes an element from the list.

Comparing the second to the third was a bit tricky at first glance the second is much slower because it need to iterate over the whole list each time it adds/inserts a new element in the prime list while the third adds the element directly to the beginning without needing to iterate over the list.

```
# Second
primes ++ [x]
#Third
[x| primes]
```

But after examining the run time data for the functions i realised that third is the slowest, the reason behind it being slower than 2 is because it adds the newly discovered primes to the beginning of the list, which is effective time and space wise for the insert function, but it is highly costly for the `checkPrime` function, since the function starts by checking divisibility by the first element in the list, as the list grows an element that is grater than 1000 will have around 10 elements that can be divided by it in a list of 10k, while if you take any list it is guaranteed have 50% of its content divided by 2. In other words the problem is that since the elements are added in reverse order to the list it takes much more time to check for primes, since 2 will be the last element in the list.