Prime numbers

Azer Hojlas

Spring Term 2022

Introduction

The assignment at hand consists of writing a function that returns a list of all prime numbers from 2 to n, where n is an arbitrarily decided positive integer. These prime values will be extracted from a list created through the Enum library. The authors code only uses two instances of different Enum library calls, to-List and filter, where the first one simply creates a list from an interval from aforementioned input, and where the latter does as the name implies. A list and a function are used as arguments, upon which the function will be used to filter out certain elements in the list (in this instance this amounts to filtering out non-prime values). Besides these two Enum function calls, the code is the sole work of author et al. with significant assistance from the internet.

This function has three separate implementations, all versions of Eratosthenes sieve, where the authors assignment is to write the programs and compare their respective runtimes. The various implementations will be elaborated on in depth further down the report.

Benchmark

The benchmark was borrowed from the "tree vs list" assignment, so it will not be explained in detail here

The benchmark did not include a dummy function call, and the author did not implement one either. It was however modified to accumulate the value of fifty loops, then divide that value with fifty in order to get the average runtime of each sieve.

Code explanation

The different code structures are to be explained below.

Instantiation

All implementations are instantiated by way of taking input n as an argument, and passing this input to Enum.to-list/2 in order to create a list of values that is to be processed. The function will then, through slightly differing function calls, pass said list to each and every version of the sieve. Below are the first and second instantiate functions, where the second is identical to the third.

```
def instantiate(n) do
    [head|tail] = Enum.to_list(2..n)
    [head|sieve(head, tail)]
end

def instantiate(n) do
    list = Enum.to_list(2..n)
    sieve(list, [])
end
```

First Implementation

The first value in the initial list is two, a prime number. A property of prime values is that they are not divisible by other values other than one or themselves, meaning that the head of the list can be used to filter out all non-primes from the tail. This is done through observing the remainder of dividing all subsequent elements with said element. If the remainder is zero, a non-prime has been detected and will be removed. The base case occurs when the entire list has been processed.

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

Second Implementation

The second version deals with two list. One with unprocessed elements and the other accumulates primes. The base case executes when there are no more elements to process in the left list, i.e all primes have been extracted and the primes list is returned. The recursive clause validates if the head is a prime by dividing it with previously found prime values (unless its the first element two in the list, in which base case it is appended). If it is confirmed as a prime value, it is inserted into the prime list at the last position, if not, the clause will resume with recursing through the tail. In order to limit code littering, only the prime validator will be submitted below, the rest of the module will be elaborated on in the third implementation, as it is nearly identical.

```
    def sieve(list, primes) do
        case list do

            [] -> primes
            [head|tail] ->

                isPrime = checkIfPrime(list, primes)

                primes = insert(isPrime, head, primes)
                sieve(tail, primes)
        end
    end

def checkIfPrime(_, []) do true end
def checkIfPrime([head|tail], [h2|t2]) do
    cond do
        rem(head, h2) == 0 -> false
        true -> checkIfPrime([head|tail], t2)
    end
end
```

Third Implementation

The third implementation is almost identical to the second. The difference lies in the insert operations. The third inserts a prime element in constant time by inserting it into the head of the prime list. The second implementation on the other hand, uses the ++ operator to concatenate the element and the list, inserting the element at the end of the list with O(n) time complexity. Consequently, the third implementations prime list is ordered from largest to smallest which is undesirable. Thus, it needs to be reversed with the function submitted below. It is tail recursive and holds an empty list as an accumulator, where added elements appear in order.

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

def reverse(list) do reverse(list, []) end
def reverse([], reversed) do reversed end
def reverse([head | tail], reversed) do reverse(tail, [head | reserved]) end
```

Differences Between Third And Second

The differing runtimes illustrate the juxtapositions of two different modus operandi, the first being O(n) insertion and constant time function return (second implementation); the latter constant time insertion and O(n) return value through reversal (third implementation). Furthermore, the third sieve checks primes by dividing elements beginning with the first element in the yet to be reversed prime list, i.e recursion through large numbers. A non-prime is far more likely discovered when divided with 2, 3, 5 or 7 etc rather than a large number e.g 1223. Thus, in the second implementation, discovering a non-prime is on average a constant-time operation, while in the third it is closer to O(n).

Table

The following are the runtimes for each sieve

List length	First	Second	Third	Fastest
16	0,18	0,13	0,19	Second
32	0,40	0,70	0,30	Third
64	0,50	0,40	0,70	Second
128	1,60	1,10	2,10	Second
512	3,70	2,60	7,30	Second
1024	10	9	24	Second
2*1024	30	26	84	Second
4*1024	100	75	352	Second
8*1024	420	270	1240	Second

Table 1: runtime in milliseconds; average runtime with varying input list length; maximum of three significant numbers excluding input

As can be observed, for small lists, the third sieve performs on par with the other two. As the input grows however, the prime checker is far more time consuming when confirming non-primes. This is better illustrated in the graph below.

Graph

The author decided to not use gnuplot in this assignment even though it is preferable. Due to the fact that the author was unable to insert four different columns of value into the program, it was easier to use google spreadsheets.

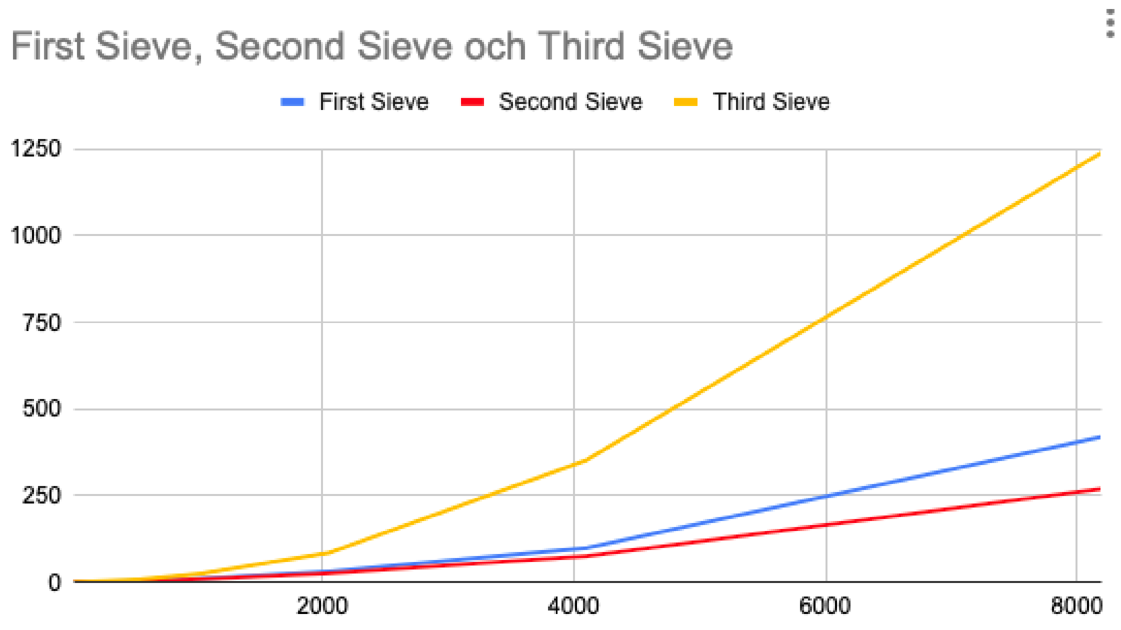

Figure 1: Runtime in milliseconds; proportional scale

Conclusion

As was confirmed with the earlier sections, the second sieve is the fastest and third the slowest. One might falsely assume that the faster insert operation of the third will outcompete the second. Meanwhile this constant time operation is negated by the reversal in the base case, i.e the net effect on the runtime should be in the vicinity of zero. The prime checker of the second, as mentioned before, is far superior to the third in terms of runtime. The first sieve is fairly similar to the second in terms of finding non-primes. i.e division with lower numbers first. The author has nothing else to add on the first implementation, as understanding it is far more difficult when using code not your own, e.g Enum functions.