Task 6 - Primes

Casper Kristiansson

2022-02-04

Introduction

Task six of the course Programming II consisted of creating three different solutions for finding prime numbers in a given range. The goal of the task is to compare how the run time of these functions. The author will be discussing how the functions was implemented and compare the run time of those functions.

Result

The author will be dividing the solution into three different sections, one for each algorithm. Each section will consist of how the author implemented the algorithm and an explanation how the algorithm works.

Algorithm One

The first algorithm consisted of finding prime numbers between a given range. The function starts of by creating a list with all the elements in the range. The algorithm than will be iterating over the list using a recursion function. The function uses **Enum.filter** to remove all elements in the list that fulfills a specific argument. This means that when it starts with the prime number two it will firstly go through the complete list and remove all elements that is divisible by two. Then the function will call itself with the rest of the list and continue to the next prime and remove all non-primes from the list. When the last number has been reached the algorithm has found all the prime numbers.

```
def primes(n) do
  lst = Enum.to_list(2..n)
  recursionFilter(lst)
end
def recursionFilter([head | tail]) do
  if tail == [] do
    [head]
  else
    [head | recursionFilter(
      Enum.filter(tail, fn(x) -> rem(x, head) != 0 end)
    )]
  end
end
```

Figure 1: Algorithm One

Algorithm Two

The second algorithm consists of creating two lists, one for the range of numbers and one for the result. The algorithm starts off by iterating over the list of numbers and checking if any numbers in the result list is divisible by that number. If the element is not divisible by any number of the result list it gets added in the end of the list. This can be solved using the **Enum.any?** which returns a Boolean expression depending if any value in the list fulfills a argument.

Because the function will return if **any** element fulfills the expression it will not be required to iterate over the complete list. For example, the number one hundred. Because one hundred is already divisible by the first prime number two, it will not be needed to check against all other elements in the result list and the Enum function will return true directly.

```
def recursion([head | tail], res) do
  if tail == [] do
    res
  else
    if Enum.any?(res, fn(x) -> rem(head, x) == 0 end) do
      recursion(tail, res)
    else
      recursion(tail, res ++ [head])
    end
  end
end
```

Figure 2: Algorithm Two

Algorithm Three

The third algorithm works exactly like algorithm two except that it adds the prime numbers it found in the begging of the list. After it has finished running through all the numbers in the list the result list is reversed and returned. This is because otherwise the list would have been in reversed order where the first prime would be the last element in the list.

```
def primes(n) do
  lst = Enum.to_list(2..n)
  res = []
  Enum.reverse(recursion(lst, res))
end
def recursion([head | tail], res) do
  if tail == [] do
    res
  else
    if Enum.any?(res, fn(x) -> rem(head, x) == 0 end) do
      recursion(tail, res)
    else
      recursion(tail, [head | res])
    end
  end
end
```

Figure 3: Algorithm Three

Discussion

The three different algorithms for finding the primes in a given range works similarly to each other by constantly checking if a prime is divisible by other numbers. But even though the general structure of the algorithm is the same, the run time is not. The author rewrote the benchmark function from task four (Tree vs list) so that it could be used to bench the different run times of the algorithms.

Amount	**Algorithm one**	**Algorithm two**	**Algorithm three**
16	0	0	0
32	0	0	0
64	0	0	0
128	0.10	0	0
256	0.10	0	0.10
512	0.20	0.10	0.41
1024	0.41	0.31	1.23
2048	1.33	0.92	4.40
4096	4.30	2.89	15.46
8192	14.33	9.11	63.80
16384	48.12	30.52	217.70

Figure 4: The benchmark between the different algorithms in milliseconds

Figure four consist of the different run time of the three different algorithms. In the table it can be seen that the algorithm two outperforms the other algorithms. This is because that algorithm will consist of the least number of comparisons between elements.

In algorithm one the function will constantly remove the elements from the list that is divisible by a prime number. For example, the prime number two will result in that all elements that is divisible by two in the range will be removed. But the downside for this, if a number like ten, that is not a prime, will firstly be compared to the number two which already than will tell that it should not be added to the list. But because of the algorithm, it is still required to check against all other elements in the list. The second algorithm solves this problem using the build in function Enum.any?. The function is able to return right away if for example ten is divisible by two and therefore limit the number of comparisons performed.

But the third algorithm, which works the same why as algorithm two, has the worst run time performance. This is because the function adds the new primes to the front of the list. Because most numbers are likely to be divisible by two rather than for example the number 997 it is required to perform more comparisons to determine if a number is a prime or not.