Primality Testing With Regular Steps

February 7, 2018

1 Introduction

Consider the naive method for primality testing, if we would like to know if a natrual number n is prime, we can just check if natural numbers between 2 and $\lfloor\sqrt{n}\rfloor$ divide n, if any single number divides n then we know that n is not prime, otherwise it is. In the following implementation of naive primality test,

```
Version 1.0

1   def prime(n):
2     let hi = fl(sqrt(n))
3     for(i=0; i<=hi; i++):
4       n%i === 0?  return false :  continue
5     return true
```

we notice that there is some redundancy. For example, if we run $prime(5)$ the at some point in the function execution, we will compute both 5 mod 2 and 5 mod 4, but $(5 \bmod 2 \neq 0) \implies (5 \bmod 4 \neq 0)$ so it is not necessary to compute that again. This article will explore how we can make this primality test algorithmn by applying some preprocessing steps to try and determine which divisibility checks we can skip before we begin we enter the main loop of the program(line 3).

Lemma 1: $(k \equiv p) \implies (k \equiv pq), \forall p, q \in \mathbb{Z}^{\geq 0}, pq \leq k$

2 Base Case

From Lemma 1, the most obvious thing to try is to change our primality test to first check for the parity of n then modify the main loop of the function.

```
Version 1.1

def prime(n):
  let hi = fl(sqrt(n))
  n%2 === 0?  return false:  continue
  for(i=3; i<=hi; i+=2):
    n%i === 0?  returh false
  return true
```

Now in the worst case scenario where n is prime, our version 1.1 algorithm only needs to check if n is divisible by half as many add 1 numbers as our version 1.0 algorithm.

3 Case $n = 1$

To take the idea in our base case further, before we enter the main loop of the program we can first check if n is divisible by 2 and 3 and then we can modify our loop to take advantage of it.

```
Version 1.2

def prime(n):
  let hi = fl(sqrt(n))
  let step = [2, 4]
  let j = -1
  n%2 === 0?  return false:  continue
  n%3 === 0?  return false:  continue
  for(i=5; i<=n; i+=step[j]):
    n%i === 0?  return false:  continue
    j = (j+1)%2
  return true
```

If we run $prime(n)$ on our version 1.2 algorithm and n is prime, we only need to do $\frac{2}{3}$ add 2 as many divisibility checks as version 1.0.

4 Case $n = 2$

Lets take a look at one more iteration of this idea before we do some further analysis of our program.

```
Version 1.3
```

```
def prime(n):
  let hi = fl(sqrt(n))
  let ls = [2, 3, 5]
  let step = [2, 6, 4, 2, 4, 2, 6, 4]
  let j = -1
  for(i=0; i<=ls.size; i++):
    n%i === 0?  return false:  continue
  for(i=0; i<=n; i+=step[j]):
    n%i === 0?  return false:  continue
    j = (j+1)%step.size
    return true
```