MATH 327 Build Your Own Assesment Write-up

Created by Aiden Taylor(30092686) and Noah Pinel(30159409)

Abstract

This file contains the written portion of our assignment, we briefly introduce each piece of code we wrote, the subject, and how it relates to the course content. Alongside this write-up there will be the source code for each of our respected programs and a short video demonstrating their functionality.

Cheers, Aiden.T and Noah.P

Contents

3 Implementation of The Sieve of Eratosthenes in Python

3.1 Overview

The Sieve of Eratosthenes is a type of prime sieve. Prime sieves are algorithms that are very good at generating prime numbers up to some upper bound n, where $n \in \mathbb{Z}^+$. The following sections explain the Implementation process of simulating the Sieve of Eratosthenes in the programming language Python.

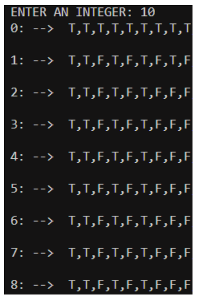
3.2 The Implementation

My first run in with the sieve of Eratosthenes was during my number theory course, the mechanical nature of the sieve seemed to be a perfect way to utilize a computer to do some interesting math. The entirety of the program relies on the algorithm below, so here are the meat and potatoes of the whole thing,

Algorithm 1 : An algorithm for The Sieve of Eratosthenes

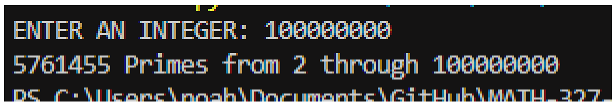
```
Require: : $n > 0$
Ensure: : List of prime numbers 2, ..., n
 1: Sieve($n$) :
 2: Populate a boolean list P size 2, ..., n.
 3: Initialize all index values to True.
 4: for $i = 2, ..., n + 1$: do
 5:     if P[i] TRUE
 6:     for $j = i^2, i^2 + 1, i^2 + 3, ..., n$: do
 7:         set P[$j$] False
 8:     end for
 9: end for
10: Return P
```

Simply put, we have a list of size $2, ..., n$. We set all values of our list to True, then starting at 2 we loop through each index setting multiples of the respected current position to false. After this has run, we are left with a list where all prime values are unchecked, that is, they remain True while composite numbers are false. Below is a visual of the algorithm working for a list of size 10.



```
ENTER AN INTEGER: 10
0: -->  T,T,T,T,T,T,T,T,T

1: -->  T,T,F,T,F,T,F,T,F

2: -->  T,T,F,T,F,T,F,F,F

3: -->  T,T,F,T,F,T,F,F,F

4: -->  T,T,F,T,F,T,F,F,F

5: -->  T,T,F,T,F,T,F,F,F

6: -->  T,T,F,T,F,T,F,F,F

7: -->  T,T,F,T,F,T,F,F,F

8: -->  T,T,F,T,F,T,F,F,F
```

We can see that after 8 iterations of following the algorithm the only index values set True are $2, 3, 5, 7$, i.e., the first 4 primes. Now as cool as it is to visualize the algorithm I feel like finding the number of primes $\leq$ some large n is a lot more interesting. To help with run time I disabled the visual aspect of the algorithm and was able to find a value for the primes up to $100,000,000$.



```
ENTER AN INTEGER: 100000000
5761455 Primes from 2 through 100000000
```

Using the forbidden site Wolfram Alpha to verify, there are indeed 5,761,455 primes in between 2 and 100,000,000.

3.3 Conclusion

The last calculation presented took just about an epsilon under a minute to calculate, which for an algorithm that has been around for thousands of years I feel is very impressive, thus, Eratosthenes sieve is a powerful tool to find primes up to some appropriate positive integer. This problem also displays the interplay between Mathematics and Computer Science and how they complement each other very well. The source code for Sieve.py will be included in our submission with documentation if you're wanting to poke around.

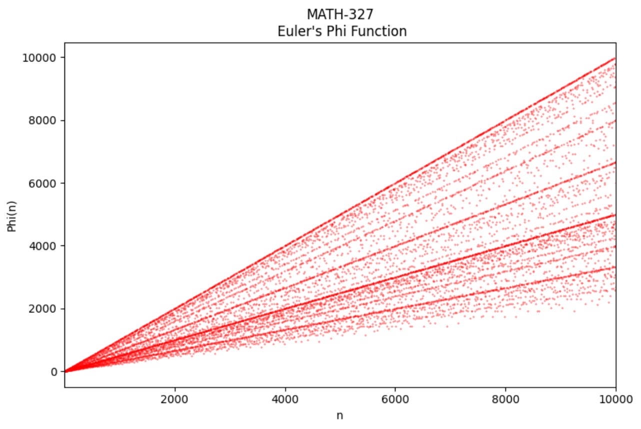


Figure 1: Euler's Phi function plotted up to 10,000

4 Visual Representation of Euler's Phi Function in Python

4.1 Overview

Euler is one of the most well known Mathematicians ever, the following program I implemented is a visual representation of a function he created, namely the Euler Phi function.

4.2 What is it?

Euler's Phi function takes in a positive value n, and counts the number of relatively prime numbers up to the original n. For example $\Phi(10) = |\{1, 3, 7, 9\}| = 4$, so Euler's function is telling us that from 1-10, there are 4 numbers coprime to 10.

4.3 The Implementation

While reading the MATH-327's assigned text book I really enjoyed seeing all the visual representations of the math we were learning. I decided to use python to create a visual of this special function. The process of coding this wasn't too bad, it consisted of reading in an n value, then for each i up to n, I would call the Phi function that I wrote, it would simply just check for a $\gcd(i_j, n) == 1$ and if it returned true it was added to a list. After I had this list of all 1,..., n's respected Phi function values I was able to use the python library Matplotlib.pyplot to graph this list against the corresponding n value, figure(1) is the awesome visual that was generated.

4.4 Conclusion

The visual representations that arise from ideas in number theory are amazing to look at, and Euler's Phi function is no excuse. The power computers give us to generate these models is just another example of how technology can give us deep insight into mathematical concepts.

5 Implementation of a 2-Variable LDE Solver in C

5.1 Overview

The definition from Weissman's Illustrated Theory of Numbers says, "A Diophantine Equation is an equation involving integers, variables, and the elementary operations of addition and substraction. Diophantine equations are linear if variables are never multiplied with variables." This program simply solves two variable Linear Diophantine Equations (LDE's) provided by the user.

5.2 The Implementation

We learned this semester that to solve LDE's we need to use the Extended Euclidean Algorithm (EEA) to find a base solution that itself is used to find the general solution to the LDE in question. With that in mind, my implementation of the EEA by Tabular Method is as follows,

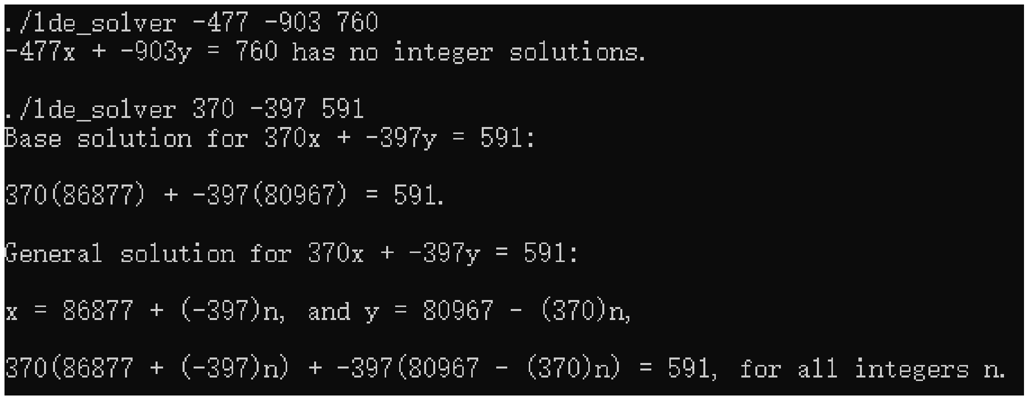
Algorithm 2 : An algorithm for the EEA by Tabular Method

```
Input: $a, b$                                   ▷ Coefficients of the two variables of the LDE
 1: $a1 \leftarrow a$                            ▷ Placeholder variables for calculating remainder
 2: $b1 \leftarrow b$
 3: $i = j \leftarrow 1$                         ▷ Placeholder variables for EEA table calculations
 4: $k = \ell = temp\_\ell = temp\_j \leftarrow 0$
 5: $r = prev\_r = q \leftarrow 0$   ▷ Remainder, Previous Remainder, and Quotient for GCD calculation
 6: while true do
 7:     $prev\_r \leftarrow r$
 8:     $r \leftarrow a1 \mod b1$                ▷ Calculating remainder
 9:     if $r = 0$ then
10:         break out of loop
11:     end if
12:     $q \leftarrow (a1 - r)/b1$               ▷ Calculating quotient for division with remainder
13:     $a1 \leftarrow b1$
14:     $b1 \leftarrow r$
15:     $temp\_\ell \leftarrow \ell$
16:     $temp\_j \leftarrow j$
17:     $\ell \leftarrow i - (q * \ell)$
18:     $j \leftarrow k - (q * j)$
19:     $i \leftarrow temp\_\ell$
20:     $k \leftarrow temp\_j$
21: end while
22: Return $\ell, j$                             ▷ Values to use in base solution for LDE
```

In the above algorithm i, j, k, and ℓ are the integers representing

a	b	1	Row
$i = 1$	$k = 0$	$a1 = a$	R_1
$\ell = 0$	$j = 1$	$b1 = b$	R_2
...	...	...	...

which are used to represent the table values for the EEA by Tabular Method. After the base solution has been computed by the EEA we need to do some simple sign management as the solution from the EEA may not always be as desired. Then, with the correct base solution, we can calculate the general solution using **Corollary 1.25** from the text, or state that there are no solutions if c is not a multiple of $GCD(a, b)$, from $ax + by = c$.


```
./lde_solver -477 -903 760
-477x + -903y = 760 has no integer solutions.

./lde_solver 370 -397 591
Base solution for 370x + -397y = 591:

370(86877) + -397(80967) = 591.

General solution for 370x + -397y = 591:

x = 86877 + (-397)n, and y = 80967 - (370)n,

370(86877 + (-397)n) + -397(80967 - (370)n) = 591, for all integers n.
```

5.3 Conclusion

This was a very interesting program to implement, where most of my time actually went into dealing with the weird sign behaviour generated by the EEA, which was a little annoying. Also, I should probably improve my variable naming as I understand using i, j, k, and ℓ most likely makes absolutely no sense to people reading the code for the first time... I'll work on it. In the end, I did not test the runtime of my implementation, but hopefully my use of C can mask the fact that I don't how to optimize code.

6 Implementation of the Miller-Rabin Test in C

6.1 Overview

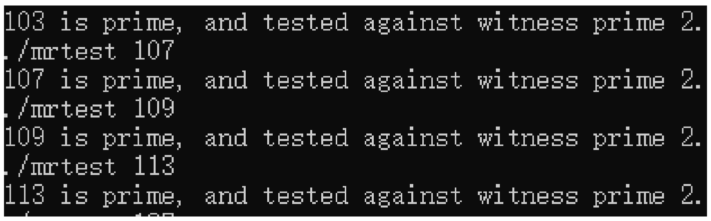
The Miller-Rabin Test is a probabilistic primality test for all positive odd integers greater than 2. However, in this implementation, we will only be looking at integers between 2 and 2,047 so that we can use the witness prime 2 to give us precise results (and so that the exponential calculations are not super huge).

6.2 The Implementation

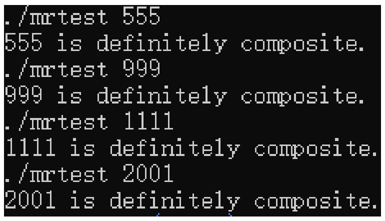
For this implementation I took inspiration from the pseudo-code at https://en.wikipedia.org/wiki/Miller-Rabin_primality_test, which would give a much more in depth explanation of the Miller-Rabin Test. Firstly, the user passes an odd integer n that is within the specified bound. Then, it is broken down into the equation $n - 1 = 2^s d$, where $s > 0$, $d > 0$, and d is odd. The reason we need to factor out powers of 2 from $n - 1$ is to check if one of the following two congruence relations hold

- $2^d \equiv 1 \mod n,$
- $2^{2^r d} \equiv -1 \mod n, \text{ for some } 0 \leq r < s,$

where these congruence relations follow from Fermat's Little Theorem. If one of these relations holds with $2 < n < 2,047$ (n odd) then we can say n is prime, where 2 is a witness for the primality of n. Otherwise, if neither of these hold, then n is composite, where 2 is a witness for the compositeness of n.



```
103 is prime, and tested against witness prime 2.
./mrtest 107
107 is prime, and tested against witness prime 2.
./mrtest 109
109 is prime, and tested against witness prime 2.
./mrtest 113
113 is prime, and tested against witness prime 2.
```



```
./mrtest 555
555 is definitely composite.
./mrtest 999
999 is definitely composite.
./mrtest 1111
1111 is definitely composite.
./mrtest 2001
2001 is definitely composite.
```

6.3 Conclusion

Although my implementation only covers a small bound, it is easy to see that by adding more witness primes one could increase the bound considerably, at the cost of having to deal with huge numbers. I should also mention my use of geeksforgeeks implementation of a Modular Exponentiation function as dealing with the huge exponential calculations was definitely the biggest hurdle in this implementation (see the source code for the reference). In the end, I am pretty happy with this implementation, and I had a lot of fun researching this test and learning how important probability is in primality tests (even if my implementation didn't use any probability).