12-15-2017

New Implementations for Tabulating Pseudoprimes and Liars

Wuyang Liu
Illinois Wesleyan University

New Implementations for Tabulating Pseudoprimes and Liars

Wuyang Liu
Illinois Wesleyan University, wliu@iwu.edu

December 15, 2017

Acknowledgement

My sincere gratitude to my research advisor Professor Andrew Shallue in the Mathematics department. What he has taught me is beyond the area of number theory. I would also like to thank all the faculty members in both Mathematics department and Computer Science department. Professor Mark Liffiton in the Computer Science Department also assisted me in coding practice.

Abstract

Whether it is applied to primality test or cryptography, pseudoprimes are one of the most important topics in number theory. Regarding the study of strong pseudoprimes, there are two problems which mathematicians have been working on:

1. Given a, b, find all a-spsp up to b.
2. Given an odd composite n, find all $a \leq n$ such that n is an a-spsp.

where $n = a$-spsp means n is a strong pseudoprime to base a, and a is a strong liar of n.

The two problems are respectively referred to as the tabulation of strong pseudoprimes and the tabulation of strong liars. The main focus of my work in this research project is on the tabulation of strong liars. This can be achieved by the application of the multiplicative group modulo n, denoted by $(\mathbb{Z}/n\mathbb{Z})^{\times}$. Instead of checking each potential candidate, we can actually construct the set of Fermat liars, a "weaker" version than strong liars, from the bottom up with the help of the primitive roots of $(\mathbb{Z}/p\mathbb{Z})^{\times}$ for all prime factors p of n. We then sieve out the set of strong liars with Millerwitness() function in NTL library. By implementing the algorithms with appropriate data structures, I verified that in most cases the runtimes have been improved compared to previous algorithms or brute force. This is to be expected, since they have less computational complexities theoretically. All implementations of the algorithms in this research project are in C++.

In 2010, Professors Mark Liffiton and Andrew Shallue built a new computer system for research purposes, known as Hyperion. This is essentially a cluster of 8 computers, also referred to as nodes, which are able to finish complex distributed computations. The operating systems installed on all the nodes in the cluster is Linux. By submitting Shell scripts which include instructions from its front end, I can grab a cup of coffee while the programs are running on the cluster. This feature is particularly beneficial for this project since the programs sometimes take tens of minutes to return all the outputs.

Contents

1 Introduction

1.1 Primality Test

Definition 1.1. A natural number greater than 1 is called a **prime number** if it has exactly 2 divisors, 1 and itself.

Throughout history mathematicians have developed various method for testing the primality of any positive integer n. The brute force method is to check each integer from 2 to $n-1$ linearly. For each integer i such that $2 \leq i \leq n-1$, test whether i divides n; if no divisor is found until $n-1$, then we can confirm that n is a prime number.

Consider the factorization of n. Once we have passed $\sqrt{n}$, any factor q greater than $\sqrt{n}$ does not need to be checked since $\frac{n}{q}$ would appear before q if q were a prime factor of n. Therefore, it is only necessary to check all integers up to $\sqrt{n}$.

1.2 Algorithmic Complexity

An algorithm is a detail-oriented method for solving computational problems of a certain category. A mathematical computation can be performed by different algorithms. In computational number theory, big O notation $\mathcal{O}$ is used to denote the asymptotic bound of an algorithm.

Definition 1.2. Let f and g be two functions defined on some subset of the real numbers. Then $f(x) = \mathcal{O}(g(x))$ as $x \to \infty$ if and only if there exists a positive real number M and a real number x_0 such that $|f(x)| \leq M|g(x)|$ for all $x > x_0$.

Let's get back to primality testing. The brute force method has $\mathcal{O}(n)$ algorithmic complexity. The computers today are sufficiently fast to handle this algorithm as long as n is within a certain range. However, we can save much more resources and significantly improve the computational complexity by using a different algorithm.

Compared to the brute force method, the second approach has $\mathcal{O}(\sqrt{n})$ computational complexity, which saves a tremendous amount of time as n grows larger, as presented in Figure 1.

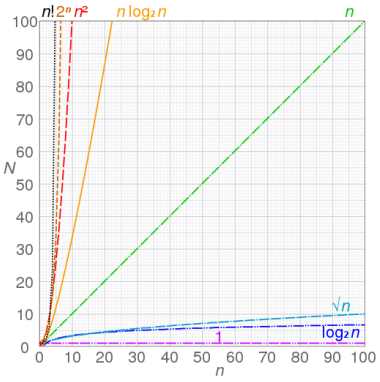

Figure 1: Graphs of different algorithmic complexities

Usually there are two main approaches to analyzing the computational complexity of an algorithm. The first approach is the theoretical analysis, often with an available pseudocode. The second approach is to implement the algorithm and run it in a computer program, and then collect the data to discover relevant differences. The second approach requires some computer programming skills in addition to knowledge of computational number theory.

2 Mathematical Background

2.1 Modular Arithmetics

Modular arithmetic is the foundation of every algorithm in this project. Normally, a division between two real numbers lead to fractions. However, if we only take into consideration the integer part of the division, we would obtain a **remainder**. For example, 7 divided by 2 makes 3, with a remainder of 1, since we know that $7 = 2 \times 3 + 1$. For any positive integer n, it can be divided by any positive integer with a remainder r such that $0 \leq r \leq n - 1$.

By manipulating the remainders combined with some arithmetic operators, we can have some interesting facts. For example, considering all remainders modulo 5, we know that $3 + 4 \equiv 2 \bmod 5$ since $7 \equiv 2 \bmod 5$. Suddenly, $3 + 4 = 2$ becomes true rather than ridiculous. In abstract algebra, we refer to such an algebraic structure as a group.

Definition 2.1. A **group** is a set G, along with one binary operation $\bullet$ with the following properties:

1. $\forall a, b \in G, a \bullet b \in G$,
2. $\forall a, b, c \in G, (a \bullet b) \bullet c = a \bullet (b \bullet c)$,
3. $\exists! I \in G$ such that $\forall a \in G, I \bullet a = a \bullet I = a$, and such I is called the **identity element**,
4. $\forall a \in G, \exists b \in G$ such that $a \bullet b = b \bullet a = I$, where I is the identity element, and b, usually denoted by a^{-1}, is called the **inverse** of a in G.

By definition, the set remainders modulo an integer with addition automatically forms a group, called the additive group modulo n, denoted by $(\mathbb{Z}/n\mathbb{Z})^+$. In this research project, a more important structure is the **multiplicative group** modulo n for $n \in \mathbb{N}$, denoted by $(\mathbb{Z}/n\mathbb{Z})^\times$. Unlike $(\mathbb{Z}/n\mathbb{Z})^+$, the additive group mod n, $(\mathbb{Z}/n\mathbb{Z})^\times$ is only a subset of the set of all integers from 0 to $n - 1$.

By group theory, the set of elements in $(\mathbb{Z}/n\mathbb{Z})^\times$ can be regarded as all positive integers $a < n$ such that $gcd(a, n) = 1$. For example, $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

Definition 2.2. The number of elements in $(\mathbb{Z}/n\mathbb{Z})^\times$ is called the **order** of $(\mathbb{Z}/n\mathbb{Z})^\times$, denoted by $\phi(n)$.

By definition we know that $\phi(15) = 8$. Note that $(\mathbb{Z}/n\mathbb{Z})^\times$ is specifically the set of x with multiplicative inverses. In the case of $(\mathbb{Z}/15\mathbb{Z})^\times$, the pairs $(2, 8)$, $(7, 13)$ are multiplicative inverses to each other; $1, 4, 11, 14$ are the multiplicative inverses of themselves respectively.

For any element $a \in (\mathbb{Z}/n\mathbb{Z})^\times$, it is natural to consider the set of powers of a mod n. Since $(\mathbb{Z}/n\mathbb{Z})^\times$ is finite, we will eventually reach to the first identity with the smallest power of a.

Definition 2.3. The smallest positive integer e such that $a^e \equiv 1 \bmod n$ is said to be the **multiplicative order** of a mod n, denoted by $l_a(n)$.

2.2 Fermat's Little Theorem

Theorem 2.1 (Fermat's Little Theorem)**.** *If p is a prime number, then $\forall a \in \mathbb{Z}$ not divisible by p, $a^{p-1} \equiv 1$ (mod p).*

If this theorem were to be applied to our primality test, there would be a significant improvement in terms of the algorithmic complexity since this would only require $\mathcal{O}(log(p))$ multiplications for any p. Unfortunately, the converse of this theorem is not necessarily true. In other words, there are some integers n that are not prime numbers but satisfy the condition in Fermat's Little Theorem, called Fermat pseudoprimes.

Definition 2.4. Let $n \in \mathbb{N}$ be composite. If $\exists a \in \mathbb{Z}$ that is not divisible by n such that $a^{n-1} \equiv 1 \pmod{n}$, then we call n a **Fermat a-pseudoprime** or **a-psp** for short, and we call a a **Fermat liar** with respect to n. The set of Fermat liars of n is denoted by $\mathcal{F}(n)$.

In essence, pseudoprimes are the composite numbers that share a common feature with prime numbers, namely that Fermat's Little Theorem is applicable to them.

A futher observation is possible. Let p be a prime number, then we know from Theorem 2.1 that $p \mid (a^{p-1} - 1)$ given that p does not divide a. We also know from number theory that $x^2 - 1 = (x+1)(x-1)$ for all $x \in \mathbb{Z}$. Let $p - 1 = 2^k \cdot d$, where k is a positive integer and d is the largest odd divisor of $p-1$, then $a^{p-1} - 1 = (a^{2^{k-1}d} + 1) \times (a^{2^{k-2}d} + 1) \times \cdots \times (a^d + 1) \times (a^d - 1)$. Since p is a prime number, $p \mid (a^{p-1} - 1)$ implies p divides one of these factors.

Definition 2.5. Let $n > 1$ be odd, composite, and $n - 1 = 2^k d$, where d is odd. Let $a \in \mathbb{N}$, $gcd(a, n) = 1$. Then n is called a **strong a-pseudoprime** or **a-spsp** for short if a satisfies either one of the following conditions:

1. $a^d \equiv 1 \bmod n$, or
2. $a^{2^i d} \equiv -1 \bmod n$ for some $0 \leq i \leq k-1$.

And a is called a **strong liar** with respect to n.

2.3 Chinese Remainder Theorem

Theorem 2.2. *Given a list of integers $\{q_1, q_2, ..., q_k\}$, if q_i are co-prime to each other and there exists a list of integer $\{r_1, r_2, ..., r_k\}$ such that $0 \leq r_i \leq q_i$ for $1 \leq i \leq k$, then there exists a smallest integer n such that $n \equiv r_i$ mod q_i for $1 \leq i \leq k$.*

A special case of Chinese Remainder Theorem is when $r_i = 0$ for all $1 \leq i \leq k$. In this case, Chinese Remainder Theorem can be applied to find the least common multiple of any set of co-prime integers.

Corollary 2.2.1. *Let n be a positive composite integer, and $n = \prod_{i=1}^{k} p_i^{r_i}$, where p_i is a distinct prime factor of n for all $1 \leq i \leq k$. Let $x > n$, then $n \mid x$ if and only if $p_i^{r_i} \mid x$ for all $1 \leq i \leq k$.*

The Chinese Remainder Theorem is especially applicable to the tabulation of strong liars. It is the most fundamental theorem in this research project besides Fermat's Little Theorem. To some extent it is even more important than Fermat's Little Theorem as it is much more frequently utilized in the various algorithms.

3 Previous Work

3.1 Brute Force

When we tabulate strong pseudoprimes to the base a, the program takes two integer inputs: a fixed upper bound b and a base a, and then tabulates all positive $n \leq x$ such that n is an a-spsp. Based upon the **Miller−Rabin primality test** successively developed by Garry Miller[1] and Michael Rabin[2], Victor Shoupe has implemented a `MillerWitness()` function in the Number Theory Library built by himself[3]. By applying to a base number a and a composite n, `MillerWitness(`a`, `n`)` will return 0 if n is an a-spsp. Therefore, a naive method of tabulating strong pseudoprimes is to simply check if `MillerWitness(`a`, `n`) == 0` is true for $1 \leq n \leq b$.

However, this function itself has a pretty significant computational complexity of $\mathcal{O}(log(n))$ multiplications for each n, thus making it necessary to reduce the number of function calls. It will be explained in later sections why the distinction can be so significant. One of the main reasons is about the size of the Fermat pseudoprimes.

3.2 Improved Algorithm

Previously, the tabulation of strong pseudoprimes has been developed and implemented by professor Shallue, with a computational complexity of $\mathcal{O}(n \cdot log(log(n)))$[5].

Theorem 3.1. *n is a a-spsp if and only if both of following conditions are satisfied:*

1. *n is a fermat pseudoprime to base a, i.e. $a^{n-1} \equiv 1$ mod n, and*
2. *For all prime factors p of n, they share the same greatest power of 2 that divides their respective multiplicative order mod p*

As mentioned in Definition 2.4, given a composite n with $n - 1 = 2^k \cdot d$ where d is odd, n is an a-spsp if and only if either one of the conditions is true. If $a^d \equiv 1 \bmod n$, then $\forall p^r | n$ as the largest prime power, we know that $a^d \equiv 1 \bmod p^r$. Because $l_a(p^r) \mid d$ and d is odd, the largest powers of 2 dividing $l_a(p^r)$ for all p^r is 0. The converse can also be proved by Chinese Remainder Theorem. If $a^{2^i d} \equiv -1 \bmod n$ for some $1 \leq i \leq k-1$, then by group theory $a^{2^{i+1} d} \equiv 1 \bmod n$, therefore the largest powers of 2 dividing $l_a(p^r)$ for all prime power factors of n are identical to $k+1$ instead of k, and vice versa.

Algorithm 1 Previous method of tabulating strong pseudoprimes by Shallue

Input: a: base, b: bound

Output: an indication array $P[]$ where $P[n] = 1$ if n is an a-spsp

1: Initialize all entries of P to 1
2: **for all** primes $p \leq b$ **do**
3: Set $P[p] = 0$
4: **if** $p|a$ **or** $p = 2$ **then**
5: Set $P[n] = 0$ for all multiples n of p
6: **end if**
7: **for all** $q = p^r \leq b$ **do**
8: Get $l_a(q)$ and the largest power of 2 dividing $l_a(q)$, denoted e
9: For any composite $n \leq b$, set $P[n] = 0$ if e is not identical for all prime factors of n
10: **if** $l_a(q) \not| (p-1)$ **then**
11: Set $P[n] = 0$ for all multiples n of p
12: **end if**
13: **end for**
14: **end for**

This improved algorithm completely eliminates the number of calls to `MillerWitness()` since the theoretical part of the algorithm is solely relevant to strong pseudoprimes, skipping the process of obtaining an array of Fermat pseudoprimes at first, which takes back the use of `MillerWitness()` in the most updated algorithm.

The algorithmic complexity is significantly reduced compared to the brute force method. However, the downside of this algorithm is the extra runtime and memory allocation to compute and store the largest power of 2 every single time. In implementing Algorithm 3.2, line 9 would incur a fair amount of overlap in computation.

4 New Progress

4.1 Tabulation of Strong Peudoprimes

There is a revision to the previous work of tabulating all a-spsp given a as base and b as the boundary of tabulation. This new algorithm had not been implemented until I started to participate in this research project. The main difference lies in the removal of power of two for the multiplicative orders of each prime factor as a reference and introduction of hash map to help with faster retrieval of multiplicative order. The tradeoff of the new algorithm is the re-introduction of `MillerWitness()` function from the NTL library. Essentially all Fermat pseudoprimes to base a are "sieved", and then strong pseudoprimes are selected among Fermat pseudoprimes. The size of the set of all Fermat pseudoprimes is significantly smaller than that of the set of all positive integer below the bound, so not many `MillerWitness()` calls are made.

Algorithm 2 Updated version of tabulating strong pseudoprimes

Input: a: base, b: bound

Output: an indication array $P[]$ where $P[n] = 1$ if n is an a-spsp

```
1: Initialize all entries of P to 1
2: for all primes p ≤ b do
3:    Set P[p] = 0
4:    if p|a or p == 2 then
5:       Set P[n] = 0 for all multiples n of p
6:       for all q = p^r ≤ b do
7:          Compute l_a(q) and store < q, l_a(q) > into a pre-defined hash table
8:       end for
9:    end if
10: end for
11: for n = 2 to b where P[n] == 1 do
12:    if l_a(q) ∤ (n − 1) for some q = p^r dividing n then
13:       Set P[n] = 0
14:    else if n fails strong pseudoprime test then
15:       Set P[n] = 0
16:    end if
17: end for
```

Theoretically, this new algorithm has a shorter runtime compared to the previous version since the step to compute and check equality for the powers of 2 has been removed. However, as I tested the runtimes for each version on Hyperion, I have found that the performance varied depending on the value of base number. For example, when I was tabulating all the strong pseudoprimes to base 2 up to 1 million, Algorithm 1 had a lower runtime than Algorithm 2; whereas the opposite was true if the base number is 5.

As demonstrated in the pseudo code, the only part of the algorithm that directly involves the base number a is the computation of multiplicative orders of a. The algorithm to compute $l_a(n)$ given a and n is presented already by my advisor Professor Shallue[5]. Therefore, it is reasonable to assume that the dependence of the computational complexity on different base numbers is derived from the computation of multiplicative orders of a.

4.2 Tabulation of Strong Liars

The most challenging part of this research project is the tabulation of strong liars since nobody had implemented this algorithm in any language previously, even though people had already developed algorithms in theory. For this part, the inputs become different from those in tabulation of strong pseudoprimes, where the base number a is fixed and all a-spsps in a certain range are tabulated. Given an odd composite n, I have to tabulate all strong liars of n that are positive and smaller than n.

Recall that `MillerWitness(a, n)` returns 0 if n is an a-spsp, i.e. a is a strong liar of n. Therefore, the brute force method of tabulating all strong liars of n is to check if `MillerWitness(x, n) == 0` returns true, exactly the same as the brute force method of tabulating strong pseudoprimes, thus causing the same trouble of computational complexity as n grows larger. However, this issue can be solved by new algorithms.

The general idea is similar to what I have implemented in tabulating strong pseudoprimes. The algorithm found all Fermat liars of n first by constructing some group algebraic structure related to the prime power factors of n and their respective multiplicative orders, and therefore strong liars of n are chosen from the set of Fermat liars with the `MillerWitness()` function.

In Section 2.1 we mentioned the notion of multiplicative group mod n, which is the key to the construction of set of Fermat liars of n. The new algorithm is highly dependent on multiplicative groups because they are the structure used to obtain the set of Fermat liars.

Any positive integer can have a multiplicative group modulo itself, and the size of the group depends on the primality of the integer. In most situations, we deal with powers of prime numbers when the candidate composite n is factorized.

Theorem 4.1 (Euler's product formula)**.** *If p^r is the rth power of prime number p, $\phi(p^r) = p^r - p^{r-1}$*

Corollary 4.1.1. *If p is a prime number, $\phi(p) = p - 1$*

Since $n = \prod_{i=1}^{k} p_i^{r_i}$, it turns out that there exists a one-to-one correspondence between $(\mathbb{Z}/n\mathbb{Z})^\times$ and $(\mathbb{Z}/p_1^{r_1}\mathbb{Z})^\times \times (\mathbb{Z}/p_2^{r_2}\mathbb{Z})^\times \times \cdots \times (\mathbb{Z}/p_k^{r_k}\mathbb{Z})^\times$, therefore $\phi(n) = \prod_{i=1}^{k} \phi(p_i^{r_i})$. Combined with Chinese Remainder Theorem we can ultimately generate a subgroup of $(\mathbb{Z}/n\mathbb{Z})^\times$.

Definition 4.1. Let $G = \{g_1, g_2, \cdots, g_k\}$ where $g_i \in \mathbb{Z}$ for all $1 \leq i \leq k$, then G is a **generator** of $(\mathbb{Z}/n\mathbb{Z})^\times$ if by combining powers of $g_i's$ modulo n we can obtain all elements in $(\mathbb{Z}/n\mathbb{Z})^\times$.

Take $(\mathbb{Z}/15\mathbb{Z})^\times$ for example again, 2 is a generator of $(\mathbb{Z}/15\mathbb{Z})^\times$ but 3 is not. Finding the generators is at the beginning of the algorithm of finding Fermat liars. The complexity of finding a generator of $(\mathbb{Z}/p\mathbb{Z})^\times$ varies depending on different implementations. Dependent on the Riemann Hypothesis[4], it takes $\mathcal{O}((log(p))^6)$ steps to obtain one assuming the Hypothesis is true. Therefore, we claim that the complexity of Algorithm 3 is $\mathcal{O}((log(p))^6)$.

Theorem 4.2. *Let p^r be a prime power, then p is a generator of $(\mathbb{Z}/p^r\mathbb{Z})^\times$ if and only if $l_g(p^r) = \phi(p)$.*

Theorem 4.3 (Euler's Theorem)**.** *Let $n \in \mathbb{N}$, then $\forall a \in (\mathbb{Z}/n\mathbb{Z})^\times$, $a^{\phi(n)} \equiv 1$ mod n.*

Recall from definition 2.4 that if a if a Fermat liar of n, then $a^{n-1} \equiv 1 \bmod n$. In addition, the multiplicative order of a mod n, $l_a(n)$ is the smallest exponent of a such that the resulting power divided by n has a remainder 1. Therefore, $l_a(n) \mid n-1$ and $l_a(n) \mid \phi(n)$.

Lemma 4.1. *Let $n > 0$ be odd composite integer, $a \in \mathcal{F}(n)$, then $l_a(n) \mid gcd(n-1, \phi(n))$.*

Algorithm 3 Find the first generator available

Input: a prime power p^r

Output: the first generator of $(\mathbb{Z}/p^r\mathbb{Z})^\times$

```
Define an integer g = 2
while g < p^r do
    if l_g(p^r) == φ(p^r) then
        Return g
    end if
    g = g + 1
end while
```

This can be applied to the prime power factors of n as well. Note that the input n is required to be odd composite integer, and that $(\mathbb{Z}/n\mathbb{Z})^\times = (\mathbb{Z}/p_1^{r_1}\mathbb{Z})^\times \times (\mathbb{Z}/p_2^{r_2}\mathbb{Z})^\times \times \cdots \times (\mathbb{Z}/p_k^{r_k}\mathbb{Z})^\times$ by Chinese Remainder Theorem. In the search of Fermat pseudoprimes, if $a^{n-1} \equiv 1 \bmod p_i^{r_i}$ for all $p_i^{r_i} \mid n$, then n is surely a Fermat pseudoprimes to base a, and a is a Fermat liar of n. Thus in the tabulation of Fermat liars, where n is fixed, factorization of n will construct the group of all Fermat liars of n.

Theorem 4.4. *Let $n > 0$ be an odd composite integer, and $n = \prod_{i=1}^{k} p_i^{r_i}$, where p_i is a distinct prime factor of n for all $1 \le i \le k$. There exists a tuple $\hat{G} = [\hat{g}_1, \hat{g}_2, \cdots, \hat{g}_k]$ such that $\hat{g}_i$ is a generator of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$.*

The structure of $\hat{G}$ is constructed from $G = [g_1, g_2, \cdots, g_k]$, where g_i is the first generator of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$ returned by algorithm 3. Then $\hat{G}$ is constructed from G such that $\forall 1 \le i \le k$, $1 \le j \le k$, $\hat{g}_i \equiv g_i \bmod p_j^{r_j}$ $\iff i \neq j$ and $\hat{g}_i \equiv 1 \bmod p_j^{r_j}$ $\iff i = j$. As a result, $\hat{g}_i$ is still a generator of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$. The method of obtaining $\hat{G}$ is through an application of Chinese Remainder Theorem, as well as the Extended Euclidean Algorithm.

Algorithm 4 requires that any pair of two elements of $P[]$ be co-prime to each other. This is obvious since each element of $P[]$ is a prime power factor of n, which is co-prime to the rest of the array. The key step in Algorithm 4 is the computation of multiplicative inverses, which is achieved by Extended Euclid Algorithm.[6]

The structure of $\hat{G}$ ensures that for each element $\hat{g}_i$ in $\hat{G}$, $\hat{g}_i$ can only generate $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$, since $\hat{g}_i \equiv 1 \bmod (\mathbb{Z}/p_j^{r_j}\mathbb{Z})^\times$ for all $j \neq i$. Therefore, by taking multiplications of $\hat{g}$'s in $\hat{G}$ with different powers, we can completely obtain all elements in $(\mathbb{Z}/n\mathbb{Z})^\times$.

Theorem 4.5. *$\forall x \in (\mathbb{Z}/n\mathbb{Z})^\times$, $\exists! E = [e_1, e_2, \cdots, e_k]$ such that $x = \prod_{i=1}^{k} \hat{g}_i^{e_i}$, where $1 \le e_i \le \phi(p_i^{r_i})$.*

To retrieve the values of all elements of $(\mathbb{Z}/n\mathbb{Z})^\times$ by construction from the group $\hat{G}$ and all possible E, we need a data structure that can smoothly keep track of different tuples E from $[1, 1, \cdots, 1]$ to $[\phi(p_1^{r_1}), \phi(p_2^{r_2}), \cdots, \phi(p_k^{r_k})]$. Professor Shallue and I borrowed the idea from car odometers. A car odometer has several digits representing the amount of mileage the car has been driven. Usually it reads from left to

Algorithm 4 Application of Chinese Remainder Theorem

Input: factorization of $n = \prod_{i=1}^{k} p_i^{r_i}$, G: an array of first generators of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$

Output: $\hat{G}$: an array of new generators of $(\mathbb{Z}/n\mathbb{Z})^\times$

1: Set an array $P[]$ of length k where $P[i-1] = p_i^{r_i}$
2: Set another array $R[]$ of the same length where $R[i] = G[i]$
3: Declare a partial product array $PP[]$ and a multiplicative inverse array $I[]$
4: Declare an array $\hat{G}$
5: **for** $i = 0$ to $k-1$ **do**
6: $\quad PP[i] = n \ / \ P[i]$
7: $\quad I[i] =$ multiplicative inverse of $PP[i]$ mod $P[i]$
8: **end for**
9: **for** $i = 0$ to $k-1$ **do**
10: $\quad$ Set $\hat{G}[i] = (\sum_{j=0}^{i-1} PP[j] \cdot I[j]) + (R[i] \cdot PP[i] \cdot I[i]) + (\sum_{j=i+1}^{k-1} PP[j] \cdot I[j])$
11: **end for**

right, and each digit ranges from 0 to 9. On any digit, if the number is 9 and increments by 1, then the digit goes back to 0 and increment the digit next to it on the left by 1, and so on.

The bases for all the digits in a car odometer are uniformly 10, thus making the combination of digits a decimal. In this project, we want an "odometer" that keeps track of the powers of the generators in the tuple. Therefore, we can define our own `Odometer.h` class by customizing the bases for each individual digit. At digit i, the base should be $\phi(p_i^{r_i})$, thus making the number on digit i range from 1 to $\phi(p_i^{r_i})$.

Consider this problem: can we have a subgroup of $(\mathbb{Z}/15\mathbb{Z})^\times$ with size 4, i.e. can we form a multiplicative group by choosing exactly 4 elements from $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$? Note that $\phi(15) = \phi(3) \times \phi(5) = (3-1) \times (5-1) = 8$. The set $\{1, 4, 11, 14\}$ can form a multiplicative group. We cannot form a subgroup of $(\mathbb{Z}/15\mathbb{Z})^\times$ of size 5 or 6, because the new size of the subgroup has to be a divisor of $\phi(15)$.

The selection of a subgroup of $(\mathbb{Z}/n\mathbb{Z})^\times$ can be generalized with generators. Let g be a generator of $(\mathbb{Z}/p\mathbb{Z})^\times$ and $\phi(p) = 6$, then $(\mathbb{Z}/p\mathbb{Z})^\times$ can be viewed as $\{g^1, g^2, g^3, g^4, g^5, g^6\}$. A subgroup of size 3 can be $\{g^2, g^4, g^6\}$. If we let $\hat{g} = g^2$, then $\hat{g}$ can be regarded as the new generator of the subgroup.

Theorem 4.6. *Let $(\mathbb{Z}/q\mathbb{Z})^\times$ be a multiplicative group mod q generated by g, where q is a prime power, then $\forall d \mid \phi(q)$, there exists a subgroup of $(\mathbb{Z}/q\mathbb{Z})^\times$ of size d, which is generated by $\hat{g} = g^{\phi(q)/d}$.*

As mentioned previously, the set of Fermat liars of n, $\mathcal{F}(n)$ is a subgroup of $(\mathbb{Z}/n\mathbb{Z})^\times$. For each $a \in \mathcal{F}(n)$, $a^{n-1} \equiv 1 \bmod n$. Since n is a positive odd composite integer, $a \in \mathcal{F}(n)$ if and only if $a^{n-1} \equiv 1 \bmod p_i^{r_i}$ for all prime power factors $p_i^{r_i} \mid n$.

Recall from Algorithm 4 that for each item $\hat{g}_i$ of the tuple $\hat{G}$, it is a generator of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$, which has a size of $\phi(p_i^{r_i}) = p_i^{r_i} - p_i^{r_i - 1}$. From Euler's Theorem, we know that $\hat{g}^{\phi(p_i^{r_i})} \equiv 1 \bmod p_i^{r_i}$. For convenience,

$\phi(p_i^{r_i})$ is denoted by s_i here.

Consider $\hat{g}' = \hat{g}^{\frac{s_i}{gcd(s_i, n-1)}}$. It generates a subgroup of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$ of size $gcd(s_i, n-1)$. In addition, $\hat{g}'^{n-1} = (\hat{g}^{\frac{s_i}{gcd(s_i,n-1)}})^{n-1} = \hat{g}^{\frac{s_i \cdot (n-1)}{gcd(s_i,n-1)}} = (\hat{g}^{s_i})^{\frac{n-1}{gcd(s_i,n-1)}} \equiv 1 \bmod p_i^{r_i}$. Thus, for any $\hat{g}'^{e_i}$ where $1 \leq e_i \leq s_i$, its $(n-1)$th power is always an identity in $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$. Therefore, the structure of $\mathcal{F}(n)$ can be constructed from $\hat{G}'$, which is derived from $\hat{G}$ by raising each item $\hat{g}_i$ of $\hat{G}$ to the power of $\frac{\phi(p_i^{r_i})}{gcd(\phi(p_i^{r_i}), n-1)}$. By Chinese Remainder Theorem, any such $a = \prod_{i=1}^{k} \hat{g}'^{e_i} \bmod n$ is a Fermat liar of n.

Algorithm 5 Tabulation of Fermat liars

Input: odd composite integer n

Output: an array $F[]$ of Fermat liars of n

1: Declare an array $F[]$ to return and an array $G[]$ to store generators
2: **for all** prime power factor $p_i^{r_i}$ of n **do**
3: Get the first generator g of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$ with Algorithm 3
4: Update the value of g with Algorithm 4
5: Raise g to the power of $\frac{\phi(p_i^{r_i})}{gcd(\phi(p_i^{r_i}), n-1)}$ and add it to $G[i]$
6: **end for**
7: Set an odometer with $\phi(p_i^{r_i})$ as the base at the ith digit for all $1 \leq i \leq k$
8: Initialize the odometer with all 1
9: **while** the odometer has not reached a cycle **do**
10: Get e_i from the ith digit for all $1 \leq i \leq k$
11: Get $a = (\prod_{i=1}^{k} g_i^{e_i}) \bmod n$
12: Add a to $F[]$
13: Spin the odometer by 1
14: **end while**

After construction of the subgroup $\mathcal{F}(n)$, the set of Fermat liars can be used as the input of the brute force method. Since $|\mathcal{F}(n)| << n$ for most $n \in \mathbb{N}$, the decrease of runtime of directly using function `MillerWitness()` is way more than the increase of runtime to compute the set of Fermat liars of n. The only exception is the Carmichael numbers, which are beyond the topic of this paper but covered in Professor Shallue's paper.[5]

4.3 Timing

One way to measure the improvement of new algorithms is to test their runtimes. Both Table 1 and Table 2 presents some comparison between the old version and the new version of tabulating strong pseudoprimes. In the tabulation of strong liars, there are no "previous" version of algorithm to compare with. As a result, brute force method was applied to test the computational complexity. Units for all timing are milliseconds.

bound	previous	improved
1000000	0.3759	0.4558
10000000	3.87	4.51
100000000	41.84	43.84
200000000	88.87	87.05
300000000	142.39	133.53

Table 1: Runtimes of two algorithms of tabulating 5-spsp

n	brute force	improved
9	0.01	0.07
51	0.25	0.07
561	2.23	0.31
5541	14.37	0.36
50001	127.74	2.87
500001	1362.85	23.84

Table 2: Runtimes of two algorithms of tabulating strong liars

As presented in Table 1, when the base number is 5, the improved algorithm started to gain advantage when the bound integer grows significantly.

Table 2 presents an example of tabulating all strong liars of a composite n. More systematic testing can be done in the future. Here my intention is to emphasize the improvement over the brute force method.

4.4 Conjecture from Liar Tabulation

The initial version of the algorithm of finding strong liars of a composite odd integer n relies on factorization of n. Given $n = \prod_{i=1}^{k} p_i^{r_i}$, since $\mathcal{F}(n)$ with the multiplication operator is a subgroup of $(\mathbb{Z}/n\mathbb{Z})^\times$ based upon Chinese Remainder Theorem, $\mathcal{F}(n)$ is supposed to be constructed with the generators g_i of $(\mathbb{Z}/p_i^{r_i}\mathbb{Z})^\times$ for all prime power factors $p_i^{r_i}$ of n.

Consider the following lemma from Cohen[9]:

Lemma 4.2. *Let p be an odd prime, and let g be a generator of $(\mathbb{Z}/p\mathbb{Z})^\times$, then either g or $g+p$ is a generator of $(\mathbb{Z}/p^r\mathbb{Z})^\times$ for every exponent $r \geq 1$.*

Theorem 4.7. *If g is a generator of $(\mathbb{Z}/p\mathbb{Z})^\times$ and $(\mathbb{Z}/p^2\mathbb{Z})^\times$, then g is a generator of $(\mathbb{Z}/p^r\mathbb{Z})^\times$ for every exponent $r \geq 1$.*

As presented in Algorithm 3, my implementation of finding the generator of $(\mathbb{Z}/p\mathbb{Z})^\times$ is to linearly search for the first integer g from 2 such that g is a generator of $(\mathbb{Z}/p\mathbb{Z})^\times$. This algorithm is based upon Euler's Theorem, after the factorization of n has been given. In the practical coding process, obtaining each prime

power factor is implemented by function `getPrimeFactors(n)`, which returns an array of all prime factors of n, including repeats. Note that it is necessary to record the count for each prime factor as its exponent.

With Theorem 4.7 in mind, I experimented both inputs p and p^2 when I was testing my function `firstGenerator()`. The return values are always the same for each prime number p.

Conjecture 1. *For any prime number p, the least generators of $(\mathbb{Z}/p^r\mathbb{Z})^\times$ are equal for all $r \in \mathbb{N}$.*

This proposition may or may not be true since it has only been partially proved by my testing code due to the limited amount of memory of any node on Hyperion. My code can only test all primes less than $2^{16} = 32768$ because p^2 will thereby goes to 2^{32}, which is the maximum amount of space the operating system can allocated on Hyperion.

Obviously, there are way more prime numbers $p \geq 32768$ than those which are less. So if you can prove my conjecture mathematically, please let me know!

The significance of this new finding is expressed by professor Shallue as below:

"This computational result by Wuyang Liu is rather surprising. Suppose that the least primitive root is random among the primitive roots modulo p, and furthermore suppose that this choice of g or $g + p$ is also random. Then the chance that Wuyang's computation is a coincidence is $2^{-\pi(32768)}$ which is much smaller than 2^{-1000}. It is likely the assumption that primitive roots are random in this sense does not hold. I certainly think this is a question ripe for further research."

4.5 Practice of Coding

In addition to the number theory I have learned in this research project, I have written thousands of lines of code to implement relevant algorithms. With permission from my research advisor, the complete source code, including my part and Professor Shallue's part, is public on GitHub[7].

Professor Shallue and his previous research students have completed a large portion of the code. My code is mainly in `formulas.cpp`, `tabpsp.cpp`, `tabliar.cpp`, *and* `Odometer.cpp`. *Functions that are completely written by myself are* `firstGenerator()`, `tabliar()`, `SieveStrongTabOnPaper()`. *Besides adding new functions to the program, I have also improved some previous implementations as well.*

Whether we are in the process of writing out the code or finding a bug during debugging, the most useful technique is to separate the snippet of code from thousands of lines of others and test the piece. When I started to work on this project, I did not know well about his technique. Now after numerous practice from this research project and other personal projects, I have become fairly proficient in writing code for any large program.

After completing the main part of the program, I revisited existing code that was working smoothly but could be further optimized. Even if the optimization may not significant improve the computational complexity, it definitely reduce the total amount of computations and make the source code more readable.

As the implementation went into the most difficult part of the algorithm. One big topic revolving around

my coding process is the tradeoff between time and space. By using C++ as the programming language, computer memory has already been saved tremendously since we can explicitly indicate pass by reference in C++. However, it is still highly important to consider the time-space balance when implementing the algorithms.

5 Future Work

Many problems still remain unsolved in this research project. Future students can delve into the problems and find new solutions. For this research project specifically, two interested problems are extensive enough to be treated as another project to do research:

1. *In both tabulations of strong pseudoprimes and strong liars, the algorithms both highly rely on a sieve of Eratosthenes.[8] It is a highly efficient method of checking the primality of a given integer. However, the downside of sieve is that it takes $\mathcal{O}(n)$ space, which is one of the biggest reason why I cannot prove further in Section 4.3. It is worth exploring the possibility of checking primality with out using sieve of Eratosthenes.*

2. *How much influence does base number have in the tabulation of strong pseudoprimes? Does the amount of tabulations have to be large enough to demonstrate the advantage of the improved algorithm compared to the brute force method?*

References

[1] *Gary L. Miller. Riemann's Hypothesis and Tests for Primality. Journal of Computer and System Sciences, 13(3):300−317, 1976.*

[2] *Michael O. Rabin. Probabilistic Algorithm for Testing Primality. Journal of Number Theory, 12(1):128−138, 1980.*

[3] *Victor Shoupe. Number Theory Library.* `http://www.shoup.net/ntl/`

[4] *Victor Shoupe. Searching for primitive roots in finite fields. Math Comp 58, 1992, number 197, pages 369-380*

[5] *Andrew Shallue. Tabulating Pseudoprimes and Tabulating Liars. ACM Transactions on Algorithms, 13(1): article 4, 2016*

[6] *Ankur. Modular Multiplicative Inverse.*
`http://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/`

[7] *Wuyang Liu. New Implementations of Tabulating Strong Pseudoprimes and Liars.*
`https://github.com/wuyangtony/tabliars`

[8] *Richard Hoche. Nicomachi Geraseni Pythagorei Introductionis arithmeticae libri II. Leipzig: B.G. Teubner, p. 31, 1866*

[9] *Henri Cohen. A Course in Computational Algebraic Number Theory. Berlin : Springer, 1996*