BENEDICTINE UNIVERSITY

INTRO TO MODERN CRYPTOLOGY

SECTION A - CLASS NBR 5221

On the importance of prime factorization and primality testing

Authors:

Luis PEREZ

Nicholas JOHNSON

May 5, 2022

On the importance of prime factorization and primality testing

Luis Perez Nicholas Johnson

May 5, 2022

Abstract

Prime numbers are among the most important numbers in mathematics and their applications in cryptology have elevated their status as modern methods of encryption rely on properties of prime numbers. Obtaining the product of two large prime numbers is considered to be easy to compute, however obtaining the prime factors from a given arbitrary product of large primes numbers is difficult to compute. By easy and difficult we refer to the time complexity for computing an algorithm given some arbitrary value. We will discuss the importance of the efficiency of algorithms for factoring integers and testing if a given integer is prime and its applications. We will provide Java and Python implementations for each discussed algorithm along with its respective time complexity.
Keywords: *Properties of primes, Python and Java implementation, Algorithmic time complexity*

1 Introduction

Since ancient times prime numbers have been the most extensively studied numbers in number theory, but why? Well primes have, amongst numbers, the most complex and intriguing properties, for one there is a theorem that states that all natural numbers greater than 1 can expressed as a product of unique primes to some power. This theorem is called the fundamental theorem of arithmetic and it is so important that its is used to proof there are infinitely many prime numbers. The fundamental theorem of arithmetic was first proven in ancient Greece by the great mathematician Euclid in 200 BC [5], Euclid was also the one who proved there are infinitely many primes by the method of contradiction [5]. The fundamental theorem of arithmetic can be expressed as the following if $n > 1 \in \mathbb{N}$, then $n = p_1 \cdot p_2 \dots p_m, m \in \mathbb{N}$, where p_i is prime. Suppose $n = 30$, then by the fundamental theorem of arithmetic $n = 2^1 \cdot 3^1 \cdot 5^1$. As mentioned this important theorem is used to proof there are infinitely many primes, the following proves the fundamental theorem of arithmetic.

Proof. We show there infinitely many primes by contradiction. Assume to the contrary that there are finitely many prime numbers, the quantity of primes is expressed as a finite integer n, then there exists a largest prime number call it p_n. Suppose there is a positive integer $m > 1$ that is a product of all primes up to and including p_n, then by the fundamental theorem of arithmetic

$m = p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_n$. Let $N = m + 1$, then $p_i \nmid N$, hence N is prime or divisible by an integer larger than p_n, but since p_n is the largest prime, this means N is prime and $N > p_n$, but this is a contradiction since the supposition is p_n is the largest prime, therefore there are infinitely many primes [7] . ■

2 Factorization and primality testing

In 1764 the great mathematician Leonard Euler published what we now know as Euler's theorem ([10]). In his work Euler defines a function $\varphi(n)$ as the totient function [10]. This function takes a positive integer greater than 1, and outputs the quantity of integers less than a given input n, that are relatively prime to n, or in other words the quantity of integers such that given a positive integer k less than n the $\gcd(n,k) = 1$ [10]. Euler's theorem states that if a, p are relatively prime or $\gcd(a,p) = 1$, then $a^{\varphi(p)} = 1(\mod p)$ [10]. The importance of this theorem presented by the fact that if p,q are primes and $r = p \cdot q$, then $\varphi(r) = (p-1)(q-1)$[10]. Since RSA public key cryptography holds on the assumption that integer factorization is difficult, then it is just as difficult to obtain $\varphi(n)$ for some large n, where n is a product of primes. Consider that on March 11, 2020, an international team of computer scientists from France and the United States set a record for integer factorization of an 829-bit semi prime issued by RSA laboratories as RSA-250 [**?**]. Each prime factor was 125 decimal digits long and in total took over 2700 years of combined computation which was distributed over thousands of machines that would compute the factors in a span of a few months [**?**]. Which brings to question how are product of primes factorized?

There are many ways to factorize a product of primes; one such way is the naive approach. Consider an integer N which is a product of two primes, the naive way to find the factors of N is to divide N by each integer less than the square root of N until there is an integer that divides with no remainder. Suppose N is not a prime then, there is an integer $k > 1$ such that when N is divided by this integer there is no remainder, this means that k is a factor of N, and N is not prime. Suppose N, is some arbitrary positive integer then, for all positive integers less than the square root of N, but greater than 1, if no integers will divide N, this means N is a prime number. This method of trivial division behaves as both a way to factor and test for primality, however this method is very inefficient and can take a long to compute when numbers are large. Nevertheless there is a distinction between factoring and primality testing as we will see the most efficient method for factoring is computed in exponential time whilst the best known method for deterministic primality testing is computed in polynomial time. Another method for factoring is Fermat's factorization method. In this method, to find the prime factors for a product of primes N, we first take the square root of N and round it up to the nearest integer. Let $a = \lceil \sqrt{N} \rceil$, let $b^2 = a^2 - N$. If b is not an integer then N is not prime. If b is not an integer then the value of a is increased by

one and the process is repeated. If no number is found then N is prime. If b is an integer then the factors are $(a+b)$ and $(a-b)$ [10]. This method can be less efficient than trial division if the prime factors of N are very far apart, however a combination of trial division and Fermat's factorization method is more efficient than either on their own [10].

Proved independently by Hadamard and de la Vallée Poussin in 1896 [9] this mathematical theorem is one of the most important results regarding primes of the last two centuries, this is the Prime Number theorem which states that the number of primes less than some positive integer n is defined as the function $\pi(n) \sim \frac{n}{\log(n)}$, where $\sim$ describes that as $lim_{n\to\infty}$, the amount of primes less than n get closer and closer to $\frac{n}{\log(n)}$ or in other words $\pi(n)$ is similar to $\frac{n}{\log(n)}$ [9]. The Prime Number theorem is important because it tells us approximately how many primes we can expect to find less than some integer, but not how to test if an integer is prime, this is were the concept of primality testing comes to play. Suppose we have a positive integer M, $M > 1$ and we want to know if M is a prime number, what are some ways we can test this. One such guaranteed deterministic way to test for primality, is using Wilson's theorem, Wilson's theorem states that M is prime if and only if $(M-1)!+1$ is divisible by M [6]. By deterministic we mean an algorithm that computes the same result while performing the same computational steps on a given input [2]. Wilson's theorem for primality testing is extremely inefficient for larger integers since we have to compute $(M-1)!$. But Wilson's theorem makes a point regarding the time complexity of computing an algorithm, and shows the importance as to why we should care about the efficiency of algorithms. Let the following function represent an algorithm that computes primality using Wilson's theorem where 1 means true and 0 means false in terms of is the input prime. Then $W : \mathbb{Z}^+ - \{1\} \to \{0,1\}$

$$W(m) = \begin{cases} 1 & \text{if } m \mid (m-1)!+1 \\ 0 & \text{if } m \nmid (m-1)!+1 \end{cases} \tag{1}$$

Suppose $m = 5$, then $W(5) = 1$, since $(5-1)! = 24$, and $24+1 = 25$, $5 \mid 25$, since $5 \cdot 5 = 25$. If $m = 6$, then $W(6) = 0$, since $(6-1)! = 120$, and $120+1 = 121$, $6 \mid 121$, since there is no positive integer value that satisfies the following equation $6 \cdot x = 121$. Now how about $W(171)$, so we need to compute 170!, but notice that for most calculators 170! is computed as infinity since this number is to large to be stored in memory. Thus this is one reason as to why Wilson's theorem for primality testing is inefficient, we will discuss more on this algorithm in the Algorithm implementations in Python and Java section. During the mid-1600s a mathematician named Pierre de Fermat asserted a theorem regarding primality testing, however, Pierre de Fermat did not provide a mathematical proof for his assertion [1]. His theorem on primality testing was later proven by Leonard Euler and published in 1736 [1]. This theorem became known as Fermat's little the-

orem, Fermat's little theorem states that if n is some prime and $a \in \mathbb{Z}$, then $a^n \equiv a(\mod n)$ in other words $n \mid a^{n-1} - 1$ if and only if p is prime and $p \nmid a$. Fermat's little theorem is actually a corollary to Euler's theorem regarding the φ function. However there is a set of positive integers known as the Carmichael numbers that yield prime using Fermat's method when in reality these numbers are composite. By definition a Carmichael number is an odd composite positive integer m, that satisfies Fermat's little theorem, where $m \mid a^{m-1} - 1$ for all positive integers a that are relatively prime to m [8]. The first three Carmichael numbers are 561, 1105, 1729, with largest ever found being 60351 digits long [8]. Suppose $C(n)$ is a function that approximates the quantity of Carmichael numbers less than n, for some sufficiently large n, then $C(n) > n^{2/7}$, this implies there exists infinitely many Carmichael numbers [8]. So given these facts Fermat's little theorem can effectively demonstrate if a number is composite, however it cannot definitely show that a number is prime due to Carmichael numbers. But the likely hood of of an odd integer being a Carmichael number is exceptionally low for sufficiently large integers. Thus why Fermat's little theorem is important as a basis for more advanced primality testing algorithms that take in to consideration the Carmichael numbers.

3 Algorithm implementations in Python and Java

Trial division algorithm in Python

```
import math
def factor(N):
    if(N==0):
        return(0,0)
    sqrt_N = math.ceil(math.sqrt(abs(N)))
    if(N%sqrt_N==0):
        return (sqrt_N,sqrt_N)
    if(N%2==0):
        return (2,N//2)
    i = 3
    while(i<sqrt_N):
        if(N%i==0):
            return(i,N//i)
        i+=1
    return(1,N)

Function Call               Function output

print(factor(12345))        (3, 4115)
```

```
print(factor(91))            (7, 13)
print(factor(17))            (1, 17)
print(factor(121))           (11, 11)
print(factor(158))           (2, 79)
print(factor(-33))           (3, -11)
```

Step by step procedure trial division algorithm

Firstly we need to import the math library for the use of ceiling, absolute and square root functions. We define our functions in Python using the **def** keyword followed by function name and the parameters of the function. Our function takes in 1 parameter N and returns a tuple as the factors of N. As a function - Let $F : \mathbb{Z} \to \{(a_1, a_2)\}, a_1, a_2 \in \mathbb{Z}$
Algorithm - First we obtain $\lceil \sqrt{|N|} \rceil$, thus we obtain a an integer greater than the square root of N, we then check to see if $N \mod \lceil \sqrt{|N|} \rceil = 0$. This means $\lceil \sqrt{|N|} \rceil \mid N$. Now the next condition we check is if N is even or $N \mod 2 = 0$, If N is even then 2 divides N thus factoring N. Next condition N is not even or a perfect square, we use a **while** loop to do trial division starting from 3 up to the square $\lceil \sqrt{|N|} \rceil$, if N is composite non-perfect square number, then then one of the factors of N is less than $\lceil \sqrt{|N|} \rceil$, if none of the conditions are satisfied then it follows that N is a prime number. The time complexity to compute the factors of N as N approaches infinity using trial division is $\mathcal{O}(e^{\frac{1}{2}\ln(N)})$, since we require calculating all integers less than $\sqrt{N}$, thus it is an exponential time algorithm ([10]).

Fermat's factorization method in Java

```
import java.util.Scanner;

public class Fermats_Factorization_Theorem {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        long number = input.nextLong();
        int a = (int)Math.ceil(Math.sqrt(number));
        double b = Math.sqrt(a*a - number);
        while (b % 1 != 0) {
            a++;
            b = Math.sqrt(a*a - number);
        }
        int factor = a - (int)b;
        System.out.println("Factor of " + number + " is " + factor);
        input.close();
```

```
    }
}

Examples:

Enter an integer: 2249
Factors of 2249 are 13 and 173
Enter an integer: 11478657843
Factors of 11478657843 are 18567 and 618229
Enter an integer: 7907
Factors of 7907 are 1 and 7907
Enter an integer: 2878
Factors of 2878 are 2 and 1439
```

Step by step procedure Fermat's factorization method

The script begins by prompting the user to enter an integer. The input is stored as a long integer N so that large numbers can be entered and used. Fermat's factorization method does not work with even numbers so first we check to see if the number is even by dividing the number by 2 and checking for a remainder. If the number is even then the factors are 2, $N/2$. For odd numbers the script uses Fermat's factorization algorithm. This starts by defining a as $\sqrt{N}$ rounded up. b is then defined as $\sqrt{a^2-N}$. If b has no remainder then $a-b$ and $a+b$ are factors of N. Otherwise a is increased by 1 and b is recalculated. this is repeated until b has no remainder.

Wilson's theorem for primality testing in Python

```
def isPrime(N):
    if( N<=1 or N%2==0):
        return False
    i = 1
    factorial = 1
    while (i<N):
        factorial = factorial*i
        i+=1
    return ((factorial +1) % N ==0)
primes = []
composites = []
for i in range(3,999,2):
    if(isPrime(i)):
        primes.append(i)
    else:
        composites.append(i)
```

```
print(primes) # prints list all of odd primes less than 999
print(composites)  # prints list of all odd composites less than 1000

Function Call                Function output
print(isPrime(1))            False
print(isPrime(5))            True
print(isPrime(-9))           False
print(isPrime(1223))         True
print(isPrime(71))           True
print(isPrime(98773))        True
```

Step by step procedure on primality testing using Wilson's theorem algorithm
First we define a function using **def** keyword that takes in a positive integer as input. We check if N is less than or equal to 1 since prime numbers are defined as positive integers greater than 1 which have only 1 and itself as factors. We also need to check if N is even, because this implies N is not prime. We use a **while** loop to calculate the factorial of $N-1$ and we return a boolean. Next we want to find all prime numbers between 3 and 999, so we create two lists that will will hold primes and composites. The time complexity to compute the primality of N as N approaches infinity using Wilson's theorem is approximately $\mathcal{O}(n(\log(n)\log(\log(n)))^2)$, since this is the time complexity of computing the factorial of an integer using the fastest method of multiplication ([3]), we require computing $(N-1)!$, thus it is a pseudo-polynomial time algorithm. However it does not take in to account how much memory is required to store the factorial of $N-1$ as N approaches infinity. The limitations of computer memory required for storing and calculating $(N-1)!$ is why Wilson's theorem is a very inefficient primality test but nevertheless interesting.

4 Discussion and Conclusion

Each of these methods for factoring and primality testing demonstrate the importance of time complexity. By time complexity we mean the concept of describing the amount of computer time used to execute an algorithm from some given input. It is assumed that a certain amount of time is taken to perform some elementary operation, with the amount of operations performed as the input to a function increases or approaches infinity relating to a function. As we have seen the naive approach is the least efficient method, the trial division algorithm is one such example of both factoring and testing for primality that is the naive approach, as we saw it is an exponential time algorithm. Fermat's factorization method is faster than the naive approach of trial division in some cases where p,q the factors of a semi prime are closer apart. Wilson's theorem for primality testing needs to calculate $(N-1)!$ which is very memory intensive for large numbers N. As we

have seen primes have astonishing properties, that make them truly unique. But as we investigate further into the primes we begin to ask more question about the primes such as are there infinitely many primes that are evenly spaced out by two, can every even integer greater than two be expressed as the sum of primes, does there always exist a prime in between n^2 and $(n+1)^2$ [4]. In terms of computational complexity, does there exist a method of prime factorization that can be computed in polynomial time. Another question, is there more efficient algorithms for computing primality and factoring then the ones presented. The answer is yes, the fastest known method for factoring is called the number field sieve which is a subexponential algorithm [10], with the fastest known primality test being the AKS primality test that is computed in polynomial time.

References

[1] R. e. Bishop. On fermat's little theorem, 2008.

[2] A. Bockmayr and K. Reinert. Concept: Types of algorithms, Oct 2010.

[3] P. B. Borwein. On the complexity of calculating factorials, Sep 2004.

[4] C. K. Caldwell. Prime conjectures and open questions.

[5] J. J. O'Connor and E. F. Robertson. Prime numbers, Jan 2018.

[6] A. Ohana. A generalization of wilson's theorem, Jun 2009.

[7] Susan. Euclid's proof that there are an infinite number of primes.

[8] E. W. Weisstein. Carmichael number.

[9] E. W. Weisstein. Prime number theorem.

[10] N. Zemel. Integer factorization and rsa encryption, Feb 2016.