The Fundamental Theorem of Arithmetic

Abstract

Throughout this project, we will cover the foundations of the Fundamental Theorem of Arithmetic, its history, and how we can program it. We will also compare the different algorithms used in factorization, and their implications for modern cryptographic practice.

Introduction

The Fundamental Theorem of Arithmetic, also known as the Unique Factorization Theorem, states that every natural number can be written using only products of prime numbers, and that this combination of prime numbers is unique to that natural number. For a long time, people have been interested in dividing numbers into smaller parts. This concept originated in Ancient Greece, where it was researched by prominent mathematicians such as Euclid and Theophrastus. Euclid authored a book called "Elements" in which he discussed prime numbers and how any number greater than 1 may be obtained by multiplying prime numbers together [1][2]. Yet, he did not demonstrate that there is just one method to achieve this. The theorem on prime factorization was not proven for a long time after it was first proposed. However, in the 18th and 19th centuries, mathematicians such as Euler, Legendre, and Gauss made important contributions to the field of number theory and in 1801, Gauss came up with a rigorous proof of the theorem that was widely accepted.

The Fundamental Theorem of Arithmetic is essential to mathematics and number theory. It helps us further understand many other mathematical concepts, such as prime numbers and coding theory. We can also use it in real-life situations like fixing errors in computer programs and making signals clearer. The Fundamental Theorem of Arithmetic is also useful when finding the greatest common divisor (GCD) of two numbers. The GCD is the largest number that can divide two numbers evenly. The theorem allows us to find the GCD by finding the common prime factors of both numbers and selecting the smallest common exponent of these factors [3].

As stated, it has aided mathematicians in better understanding prime numbers. One of the most intriguing things it enables us to do is to demonstrate that there exists an unlimited number of prime numbers. Prime numbers have been a topic of research for many centuries, and mathematicians continue to develop the field today.

Proofs

We will prove the main part of the theorem in two ways, namely, that there exists a prime factorization for every integer n bigger than 1. Given this, we will prove that this factorization is unique.

Induction

We need to show that every natural number, $N>1$ has a representation of a product of prime numbers. First the base case of 2 is already complete since 2 is prime. Then, by strong induction, suppose for all $k<N$ the theorem holds. If N is prime, then we have nothing to prove and are done. If it is not prime, it is composite and thus has factors $N = ab,$ where $1 \leq a \leq b < N$, and since a and b are less than N, by our assumption they have unique prime products, and hence so does N.

Contradiction

Suppose there exists an integer k that cannot be factored into a product of primes and that it isn't prime itself. Then we can construct a set X such that all its elements are integers that cannot be factored into primes. In other words, since the natural numbers are ordered there exists a least such k, which implies all numbers less than k have a prime factorisation. Now k cannot be prime, so it must be composite, which means $k = ab$ for some a and b, but $1 \leq a \leq b < k$. Hence k can be factored into a product of primes, and this contradicts our original statement.

Uniqueness

The uniqueness will be proved with contradiction and will be in the same vein as the last proof. Suppose there exists a natural number N such that N has two different prime factorizations $N = p_1^{R1} \cdots p_S^{RS} = q_1^{L1} \cdots q_t^{Lt}$ and construct a set X of all numbers that have more than 1 prime factorization. Again, since there is an ordering on the natural numbers, there will be a least such N. Now, since the RHS of N is divisible by p_1 there has to be some j such that q_j is divisible by p_1, and we can cancel both of these factors from N, leaving us with another integer that is a product of primes, and that has two different factorizations, namely the ones that don't include p_1 and q_j ,hence N is not the smallest such member of X, which again is a contradiction.

Different Methods for Prime Factorization

Firstly, and most simply, we should examine the iterative method for computational factorization. This method checks for perfect divisibility from 2 until $\sqrt{N}$, where N is to be factorized. A pair of factors is then taken, and the process is repeated on each factor. This continues, until the original pair has been decomposed into prime factors, at which point we have found the prime factorization of N. An example of the code is provided below.

```
import math
import numpy as np
from collections import Counter

def fact(n):
    unbroken = []
    limit = math.trunc(math.sqrt(n)) + 1
# We only need numbers up to the root, as factors mirror after that point.
    for i in range(2, limit):
        if (n / i).is_integer():  # Checking for divisibility
            unbroken.append(int(i))
            unbroken.append(int(n / i))
            break
    if len(unbroken) == 0:
        return "Prime"
    else:
        return unbroken
def primes(n):
    primefs = []
    if fact(n) == "Prime":
        return "Prime"
    else:
        trims = fact(n)
        while np.prod(primefs) != n:
            for i in trims:
                if fact(i) == "Prime":
                    primefs.append(i)
                    trims.remove(i)
                else:
                    trims.remove(i)
                    trims = trims + fact(i)
                # This takes the factors and breaks them up into their prime components.
    return Counter(primefs)  # This returns the primes and their powers in the format (Prime:Power)
```

We should now examine some key components of the more complex algorithms we will discuss, beginning with the concept of the sieve. The simplest sieve, the Sieve of Eratosthenes, goes through each integer up to $\sqrt{N}$ and eliminates its multiples to find all the primes up N. An example of a sieve algorithm for factorization is one that continuously divides N by a prime number, starting from two, until it is no longer possible, at which point it moves on to the next prime number. The prime at which the algorithm stops can be called B, and N is said to be B-Smooth, as all its prime factors are less than or equal to B.

Fermat's factorization method is a vital part of the two fastest algorithms. It begins by factoring out of N all the powers of 2, such that the remaining number is a product of odd numbers, and hence is itself an odd number, which we will call N_p. Every pair of odd numbers has a midpoint on a number line, and so each pair can be written as $(m\text{-}d)$, $(m+d)$, where m is the midpoint and d is the distance to an odd number in the pair. This is the difference of squares; the odd number N_p can be written as $N_p = m^2\text{-}d^2$. All that remains is to rearrange to $N_p+m^2=d^2$ and find a square number that adds to N_p to make another square number.

Discussing Algorithms

When discussing the best algorithm to use, the main element to consider is the size of the number we wish to factorize. For numbers under 10^5, lookup tables or simple sieve algorithms are incredibly quick [4].
For numbers <1025, Pollard's Rho Algorithm can be used [4]. This works by generating an eventually cyclic sequence and finding factors using points on that sequence. A brief program demonstrating this is provided below.

```
from math import gcd
def g(x, n):
    return ((x ** 2) + 1) % n
def rho(x0, num):
    a = x0
    b = x0
    d = 1
    facts = []
    while d == 1:
        b = g(b, num)
        a = g(g(a, num), num)
        d = gcd(abs(b-a), num)
        if d != num and d != 1:
            return d
```

Lenstra elliptic-curve factorization is best applied to numbers that do not have factors exceeding 50 digits. This is because the runtime depends heavily on the size of the smallest factor [5]. Due to this property, for numbers under 100 digits, we need the quadratic sieve[4].

The quadratic sieve works by squaring integers, starting from the $\sqrt{N}$ and increasing by one each loop, and taking mod N. The result is then factorized and, if it is B-Smooth, it is recorded. This is the data collection phase. The goal is to collect enough data such that a perfect square can be created by multiplying two B-Smooth results. By then taking the difference of two squares between the B-Smooth results and their associated squared integers, a non-trivial factor of N is found.
For numbers even greater, we need to use the general number field sieve, whose method is provided in [6].

Greatest Common Divisor

The greatest common divisor (GCD) of two or more positive integers, is the largest divisor common to every element of the set of integers. Many methods exist for calculating this, including the Euclidean algorithm, prime factorization, Stein's algorithm and Lehmer's algorithm.

The Euclidean algorithm has multiple forms but all work on the principle of creating smaller and smaller positive linear combinations of two integers. One method using the Euclidean

algorithm is to replace the larger of the two numbers by its difference with the smaller number. This process is repeated iteratively until the difference is equal to 0, when the value of both integers is the GCD. However, this method becomes inefficient when the difference is large. Thus, an alternative method involves replacing the larger of the two numbers by its remainder when divided by the smaller of the two - this method is repeated iteratively.
The proof of this method is shown below:

Let $a = bq + r$. Then find a number u which divides both a and b (so $a = su$ and $b = tu$).
Then u divides r since $r = a - bq = u(s - qt)$

Now, find a number v which divides b and r (so $b = s'v$ and $r = t'v$)
Then v divides a since $a = bq + r = v(s'q + t')$

Thus, every common divisor of a and b is a common divisor of b and r, so the procedure can be iterated.
When q *(nth +1)* term divides r *(nth-1)* term exactly, the algorithm terminates with r *(nth)* term equal to the greatest common divisor of a and b.
A program for calculating the GCD of two integers via Euclidean Algorithm is shown below.

```
# Creation of function that performs Euclidean algorithm to find GCD. Function takes arguments n1 and n2 which
# represent two integers.
def gcd_via_euclidean_algorithm(n1,n2):

    # If mod of n1 and n2 equals 0, n2 is the GCD. If the mod does not equal 0, there is a recursive loop until the
    # mod equals 0. Each iteration of the loop n1 is reassigned to n2 and n2 reassgined to the remainder of n1/n2.
    while (n1 % n2 != 0):
        (n1, n2) = (n2, n1 % n2)

    #The final value of n2 from this loop is the GCD of the two integers.
    print(f'The greatest common divisor is {n2}')
```

Another method mentioned for finding the GCD of two integers (a,b) is to write the prime factorizations of a and b, in terms of the prime factors (pi) and the number of times the prime factor appears as the exponent for that factor (a_i). From the prime factorizations, the GCD (a,b) can be found by taking the factors that appear in both expressions, with the exponent of this prime factor equal to the least number of times it appeared in either. If P_i does not appear in both factorizations, then the corresponding exponent is taken as 0. This is summarised below.

$$a \equiv \prod_i p_i^{\alpha_i}, \quad b \equiv \prod_i p_i^{\beta_i}, \longrightarrow \mathrm{GCD}(a, b) = \prod_i p_i^{\min(\alpha_i, \beta_i)}$$

Both methods can be used to successfully calculate the GCD of two numbers. However, the prime factorization method is more inefficient - largely as a result of inefficiencies in factorization - and this is accentuated when working with larger numbers. Thus, both for non-computing and computing calculations, the Euclidean algorithm is more efficient and appropriate for calculating the GCD and becomes increasingly so as the size of the numbers increase. Gabriel Lame's proof that the Euclidean algorithm never requires more steps than five times the number of digits further highlights the efficiency of this method.

Prime Factorization in Cryptography

Cryptography is simply the practice of techniques that ensure communication is secure, allowing for information to be hidden from a third party attempting to view it. These techniques are vital to be studied as they can help keep internet security high and avoid information leaks and loss of data to attackers. Many internet security measures apply computationally challenging mathematical algorithms to protect data, providing the foundation for encryption, digital signatures, and random number generation. As for its relevance to the theorem at hand, the use of prime factorization plays a key role in internet security measures.

Prime factorization is important as it forms the basis for a variety of secure communication methods across the internet. An example of one of the most commonly used techniques for this is RSA (Rivest-Shamir-Adleman) cryptography. This method of cryptography factors large integer numbers that are the product of large prime numbers. Attempting to determine the original prime numbers used would take an incredible amount of time for even some of the best modern supercomputers, hence providing a good security measure for data encryption [7] . The prime numbers are generated using the Rabin-Miller primality test algorithm which generates large prime numbers, multiplies them together and the product produced serves as the public key that encrypts the data which can be decrypted by the private key [7]. Longer key lengths provide better security, but impacts performance on websites.
Another security method used to encrypt and protect data transmitted over the internet is called ECC (Elliptic Curve Cryptography). This involves the use of elliptic curves structured algebraically across a finite field creating another link between both the public and private keys, following a very similar approach to RSA [8]. ECC however, allows shorter key lengths to be used, which helps maintain performance on websites and creates a better level of security than RSA. It is considered that ECC will become more widely used over RSA in the near future [8].

References:

[1] Jones, O. D. (2012). The history and significance of the fundamental theorem of arithmetic. The College Mathematics Journal, 43(1), 2-13. (Date accessed: 17/03/2023)
[2] - https://core.ac.uk/download/pdf/81978891.pdf (date accessed: 17/03/2023)
[3] - https://towardsdatascience.com/number-theory-history-overview-8cd0c40d0f01 (date accessed: 17/03/2023)
[4] Speiser, Jacqueline, “Implementing and Comparing Integer Factorization Algorithms”.
[5] Parker, Daniel, “Elliptic Curves and Lenstra’s Factorization Algorithm”.
[6] Pomerance, Carl; Erdös, Paul; “A Tale of Two Sieves”, 1998.
[7] - https://www.techtarget.com/searchsecurity/definition/RSA (date accessed: 12/03/2023)
[8] - https://avinetworks.com/glossary/elliptic-curve-cryptography/ (date accessed: 12/03/2023)