Classical Algorithms for Factorization

Project Report for PHY631
Instructor: Professor Arvind
Submiited By Ankit (MS22189)

December 2024

Contents

0.1 Introduction

Factorization is a fundamental concept in number theory with applications in cryptography, particularly in algorithms like RSA. This report discusses classical algorithms for factorization, including their theoretical underpinnings and practical implementations.

The Python implementations of these algorithms are available in the following GitHub repository for further exploration and experimentation:

GitHub Repository: Classical Factorization Algorithms

Below are the libraries used in the Python code:

```
import json
import math
import random
```

0.2 Primality Testing

Before factorization, a crucial step is determining whether a number is prime. In this project, two approaches were used:

1. **Small Numbers**: A precomputed JSON file containing primes up to 10^6 was used for fast lookup.
2. **Large Numbers**: The Miller-Rabin Primality Test was implemented for probabilistic primality testing.

0.2.1 Managing and Extending `prime.json`

The implementation dynamically manages the `prime.json` file to ensure it contains all necessary primes for factorization or primality testing. Given a number n, the following steps are performed:

- The function first computes the floor of the square root of n, denoted $\lfloor \sqrt{n} \rfloor$.
- It then checks if the largest prime in the `prime.json` file is greater than or equal to $\lfloor \sqrt{n} \rfloor$.

- If the largest prime in the file is smaller than $\lfloor\sqrt{n}\rfloor$, the list of primes is extended by generating all primes up to this value.
- Finally, the function returns all primes from 2 up to $\lfloor\sqrt{n}\rfloor$, either from the existing list or the newly extended list.

This dynamic approach ensures that we only generate the required primes and avoid recalculating or reloading primes unnecessarily.

Code for Managing and Extending `prime.json`

```python
def generate_primes(limit):
    """Generate primes up to a given limit using the
       Sieve of Eratosthenes."""
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False  # 0 and 1 are not prime
    for i in range(2, int(limit**0.5) + 1):
        if sieve[i]:
            for multiple in range(i * i, limit + 1, i):
                sieve[multiple] = False
    return [x for x, is_prime in enumerate(sieve) if
       is_prime]

def get_primes_up_to_sqrt(n):
    """Ensure prime.json contains all primes up to floor
       (sqrt(n)) and return the list of primes up to
       floor(sqrt(n))."""
    sqrt_n = math.floor(math.sqrt(n))

    try:
        with open("prime.json", "r") as file:
            primes = json.load(file)
    except FileNotFoundError:
        primes = generate_primes(2)
        with open("prime.json", "w") as file:
            json.dump(primes, file)
```

```
    if primes[-1] < sqrt_n:
        new_primes = generate_primes(sqrt_n)
        primes = sorted(set(primes + new_primes))
        with open("prime.json", "w") as file:
            json.dump(primes, file)

    return [p for p in primes if p <= sqrt_n]
```

Example **Input**: $n = 20000$

- Compute $\lfloor\sqrt{20000}\rfloor = 141$.
- Check if the largest prime in `prime.json` is ≥ 141. If not, extend the file.
- Return primes up to 141.

Output: A list of primes up to 141 (e.g., $[2, 3, 5, \ldots, 139]$).

Why This is Important

This method ensures that factorization and primality testing can seamlessly handle numbers of any size without manual intervention. The automatic extension of `prime.json` eliminates the need for regenerating primes repeatedly, making the process efficient and scalable.

0.2.2 Miller-Rabin Primality Test

The **Miller-Rabin Primality Test** is a probabilistic algorithm used to determine whether a number is prime. It is based on properties of modular arithmetic and builds upon **Fermat's Little Theorem**, which states that for a prime number p and any integer a,

$$a^{p-1} \equiv 1 \pmod{p}$$

How It Works

The algorithm works by checking if a number n satisfies properties that are true for prime numbers. The steps involved are:

1. **Write $n-1$ as $2^s \cdot d$:** Decompose n-1 into a product of an odd integer d and a power of 2 (s).

2. **Choose random bases a:** Random integers a are selected in the range $2 \leq a \leq n-2$.

3. **Check modular properties:** Compute $x = a^d \mod n$.
   If $x \equiv 1 \pmod{n} or x \equiv -1 \pmod{n}$, the test passes for this round.

4. **Iterative squaring:** If x is neither 1 nor n-1, repeatedly square x up to s-1 times. If $x \equiv n-1$ during any iteration, the test passes for this round.

5. **Repeat the test:** The process is repeated for multiple values of a. The more rounds performed, the higher the confidence in the result.

Why It Works

The Miller-Rabin Primality Test is probabilistic because it relies on modular arithmetic properties true for prime numbers but not necessarily for composite numbers. The test selects random bases a and checks whether $a^{n-1} \equiv 1 \pmod{n}$. For prime numbers, this holds for all a, while for composites, there is at least one a where this fails.

If n is composite, a base a is a *witness* to its compositeness if it satisfies one of the following conditions:

- $a^d \equiv 1 \pmod{n}$, where a is the odd part of $n-1$,
- $a^{2^r d} \equiv -1 \pmod{n}$ for some $0 \leq r \leq s-1$, where $n-1 = 2^s \cdot d$.

If n is composite, the probability that a is not a witness to compositeness is denoted as $P_{\text{false positive}}(a)$. The chance of a composite number passing the test in a single round is at most $\frac{1}{4}$.

Since the test is repeated for k rounds with independent random bases, the probability of passing all rounds is:

$$P_{\text{false positive}} = 4^{-k}$$

Thus, after k rounds, the probability of a false positive decreases exponentially. After 5 rounds, the probability is $\leq 1/1024$.

This exponential error reduction makes the Miller-Rabin test reliable for large numbers, especially with a large k. It runs in $O(k \log^3 n)$ time, due to the efficient modular exponentiation using methods like **exponentiation by squaring**, making it suitable for cryptographic applications like RSA key generation and digital signatures.

Miller-Rabin Primality Test Code

```python
def miller_rabin(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    s, d = 0, n - 1
    while d % 2 == 0:
        s += 1
        d //= 2

    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

0.3 Classical Algorithms for Factorization

Factorization of integers is a cornerstone problem in computational number theory, with critical implications in cryptography. This section presents a comprehensive overview of the classical factorization algorithms implemented during this project. The methods range from basic techniques like Trial Division and Wheel Factorization to advanced strategies such as the General Number Field Sieve (GNFS).

0.3.1 Basic Algorithms

Basic algorithms for factorization rely on systematic checking of divisors to identify the factors of a given number n. These methods are conceptually simple and provide a foundation for more advanced techniques.
This section introduces two basic algorithms:

1. **Trial Division**, which systematically tests divisibility.
2. **Wheel Factorization**, an optimization of trial division by skipping unnecessary divisors.

Trial Division:

Idea Behind the Algorithm The Trial Division algorithm determines the factors of n by systematically checking divisibility with integers starting from 2 up to $\sqrt{n}$. To improve efficiency, only prime numbers are tested as divisors.

Algorithm Steps

1. Initialize a list of primes up to $\lfloor\sqrt{n}\rfloor$.
2. For each prime p, check if n is divisible by p.
3. If divisible, add p to the list of factors and divide n by p repeatedly until n is no longer divisible.
4. Continue until $p > \lfloor\sqrt{n}\rfloor$.
5. If $n > 1$, n itself is a prime factor.

Why It Works The algorithm is based on the fact that if n is composite, it must have at least one factor $\leq \sqrt{n}$. Testing only prime divisors eliminates unnecessary computations and ensures correctness.

Trial Division Implementation

```
def trial_division(n):
    factors = []
    for p in range(2, int(n**0.5) + 1):
        while n % p == 0:
            factors.append(p)
            n //= p
    if n > 1:
        factors.append(n)
    return factors
```

Example **Input**: $n = 100$ **Steps**:

- Compute $\lfloor\sqrt{100}\rfloor = 10$. Only divisors up to 10 need to be tested.
- Test divisors: $2, 3, 5, 7 \ldots$.
- $100 \div 2 = 50$, $50 \div 2 = 25$ (add $2, 2$ to factors).
- Next, $25 \div 5 = 5$, $5 \div 5 = 1$ (add $5, 5$ to factors).

Output: $[2, 2, 5, 5]$

Improved Trial Division Algorithm

The classical Trial Division algorithm is enhanced by incorporating two key improvements:

- **Miller-Rabin Primality Test**: A probabilistic primality test that quickly determines if a number is prime, eliminating unnecessary trial division when the number is already prime.
- **Dynamic Prime List Management**: The prime list is dynamically managed by reading from and writing to a `prime.json` file. This ensures that the list of primes always contains all primes necessary for factoring numbers up to $\lfloor\sqrt{n}\rfloor$, the largest potential factor for any number n.

These techniques improve the efficiency of the Trial Division algorithm, particularly for large numbers.

Enhanced Trial Division Algorithm with Miller-Rabin and Dynamic Prime List

```python
def trial_division_with_miller_rabin(n):
    """Enhanced Trial Division with Miller-Rabin test and dynamic prime list."""
    # Check if the number is prime using Miller-Rabin
    if miller_rabin_primality_test(n):
        return [n]

    primes = get_primes_up_to_sqrt(n)
    factors = []

    for p in primes:
        while n % p == 0:
            factors.append(p)
            n //= p

    if n > 1:
        factors.append(n)  # n is prime if greater than 1

    return factors
```

Explanation of the Code The enhanced algorithm consists of the following components:

1. **Miller-Rabin Primality Test**: If the input number n is identified as prime, the algorithm directly returns n as the sole factor without performing trial division.

2. **Dynamic Prime List Management**: The function `get_primes_up_to_sqrt(n)` dynamically retrieves primes. It ensures the `prime.json` file is updated with all primes up to $\lfloor \sqrt{n} \rfloor$ when needed.

3. **Trial Division**: The algorithm iterates over the prime list to divide n iteratively. If n is still greater than 1 after division, it is added as a prime factor.

Example **Input**: $n = 200$

- Compute $\lfloor \sqrt{200} \rfloor = 14$.
- Use the Miller-Rabin test to check primality. 200 is not prime.
- Retrieve primes up to 14 from `prime.json`: $[2, 3, 5, 7, 11, 13]$.
- Perform trial division: $200 \div 2 = 100$, $100 \div 2 = 50$, $50 \div 2 = 25$, $25 \div 5 = 5$, $5 \div 5 = 1$.

Output: $[2, 2, 2, 5, 5]$

Benefits of the Improved Algorithm The enhancements offer the following advantages:

- **Efficiency**: The Miller-Rabin test eliminates unnecessary computations by quickly identifying prime numbers.
- **Scalability**: Dynamic prime list management avoids repeated generation of primes, making the algorithm scalable to larger numbers.
- **Optimization**: Reusing stored primes minimizes computational redundancy.

Time Complexity Analysis **Classical Trial Division Algorithm**:

- Iterates through primes up to $\lfloor \sqrt{n} \rfloor$, approximately $O(\frac{\sqrt{n}}{\log n})$ primes.
- Each division operation takes $O(\log n)$ time.
- Total time complexity: $O(\frac{\sqrt{n}}{\log n} \cdot \log n) = O(\sqrt{n})$.

Improved Trial Division Algorithm:

- Miller-Rabin test: $O(k \cdot \log^3 n)$, where k is the number of rounds.
- Dynamic prime list management (using the Sieve of Eratosthenes): $O(\sqrt{n} \log \log n)$.
- Trial division: $O(\sqrt{n})$.
- Combined time complexity: $O(k \cdot \log^3 n) + O(\sqrt{n} \log \log n)$.

Comparison of Complexities

- **Classical Trial Division**: $O(\sqrt{n})$.
- **Improved Algorithm**: $O(k \cdot \log^3 n) + O(\sqrt{n} \log \log n)$. The improvements reduce unnecessary computations, making it significantly faster for large n.

By integrating the Miller-Rabin Primality Test and dynamic prime list management, the improved algorithm delivers a more efficient and scalable solution for factorization tasks, particularly for large numbers.

—

Wheel Factorization Algorithm

The Wheel Factorization algorithm is an optimization of the Trial Division algorithm. It systematically skips over numbers that are divisible by small primes, reducing the number of divisibility tests needed.

Idea Behind the Algorithm The Wheel Factorization algorithm improves the Trial Division by using a "wheel" to skip numbers that are divisible by small primes. This concept is based on the observation that the numbers that need to be tested for divisibility are only those that are not divisible by the first few small primes, typically $2, 3, 5, \ldots$. By skipping over multiples of these primes, the algorithm reduces the number of checks, speeding up the factorization process.

Algorithm Steps The steps involved in the Wheel Factorization algorithm are as follows:

1. Select two bases a_1 and a_2 (e.g., $a_1 = 2$ and $a_2 = 3$) to generate the numbers that are candidates for primality checks.
2. Generate a prime list using the formula $\gcd(a_1, a_2)$ + numbers, where numbers are those that don't belong to the group of $\text{mod}(\gcd(a_1, a_2))$.
3. Begin testing divisibility by primes from the generated list instead of testing all integers up to $\sqrt{n}$.
4. Perform divisibility tests only for numbers in the list, thereby skipping over multiples of the selected small primes.

Why It Works The Wheel Factorization works by reducing the number of unnecessary divisibility checks. By skipping over multiples of small primes (e.g., 2, 3, 5, etc.), the algorithm focuses only on numbers that are potentially prime. This reduces the number of checks needed for larger numbers and thus speeds up the factorization process.

The basic idea behind this approach is that if a number n is divisible by a small prime p, then n is certainly not prime. The wheel avoids these numbers, testing only those numbers that are not divisible by any small primes.

Code for Wheel Factorization Algorithm

```
def generate_wheel_primes(base1, base2, limit):
    wheel_period = math.lcm(base1, base2)
    wheel_iterations = (limit // wheel_period) + 1
    non_zero_remainders = []
    wheel_prime_candidates = [base1, base2]

    for i in range(1, wheel_period):
        if gcd(wheel_period, i) == 1:
            non_zero_remainders.append(i)
            if i != 1:
                wheel_prime_candidates.append(i)

    for i in range(1, wheel_iterations):
        for remainder in non_zero_remainders:
            wheel_prime_candidates.append((wheel_period
               * i) + remainder)

    return wheel_prime_candidates
```

```
1
2  def wheel_factorization(n, base1=2, base2=3):
3      """Perform factorization using the Wheel
          Factorization algorithm with base1 and base2
          bases."""
4      # Step 1: Generate wheel primes based on the gcd of
          base1 and base2
5      wheel_prime_candidates = generate_wheel_primes(base1
          , base2, int(math.sqrt(n)) + 1)
6      # Step 2: Test divisibility for numbers in the wheel
7      factors = []
8      for prime in wheel_prime_candidates:
9          while n % prime == 0:
10             factors.append(prime)
11             n //= prime
12         if n == 1:
13             break
14
15     return factors
```

Example **Input**: $n = 100$
Steps:

- Select bases $a_1 = 2$ and $a_2 = 3$.
- Then wheel remainder, we getting are 1 and 5. Therefore, wheel number are in the form of $6k + 1$ and $6k + 5$.
- Using this form, wheel primes we get are $2, 3, 5, 7 and 11$.
- Test divisibility for the generated primes.
- Factorization process: $100 \div 2 = 50$, $50 \div 2 = 25$, $25 \div 5 = 5$, $5 \div 5 = 1$.
- Then the factors of 100 are $2, 2, 5$ and 5

Time and Computational Complexity

Time Complexity The time complexity of the Wheel Factorization algorithm depends on:

- **Prime Candidate Generation**: $O(\sqrt{n}/\log n)$ for generating candidates up to $\lfloor\sqrt{n}\rfloor$.
- **Factorization**: $O(\sqrt{n}/\log n)$ due to reduced checks.

Comparison with Trial Division

- **Wheel Factorization**: $O(\sqrt{n}/\log n)$.
- **Trial Division**: $O(\sqrt{n})$.

The Wheel Factorization algorithm is more efficient as it avoids redundant divisibility checks.

0.3.2 Fermat's Algorithm s

Fermat's Factorization Method

Idea Behind the Algorithm

Fermat's Factorization Method works on the principle of representing a composite number n as the difference of two squares:

$$n = x^2 - y^2 \implies n = (x + y)(x - y)$$

If $x^2 > n$, then the factorization can be achieved by finding x and y such that $x^2 - n$ is a perfect square.

Algorithm Steps

1. Start with $x = \lceil\sqrt{n}\rceil$.
2. Compute $y^2 = x^2 - n$.
3. Check if y^2 is a perfect square. If yes, compute $y = \sqrt{y^2}$ and return the factors:
$$\text{Factors: } p = x + y,\ q = x - y$$
4. If y^2 is not a perfect square, increment x and repeat.

Why It Works This method is efficient when the two factors of n are close to each other because fewer iterations are required to find x and y.

Code for Fermat's Factorization Method

```python
def is_perfect_square(num):
    """Check if a number is a perfect square."""
    sqrt_num = int(math.sqrt(num))
    return sqrt_num * sqrt_num == num

def fermat_factorization(n):
    """Fermat's Factorization Method."""
    x = math.ceil(math.sqrt(n))
    while True:
        y2 = x * x - n
        if is_perfect_square(y2):
            y = int(math.sqrt(y2))
            return (x + y, x - y)
        x += 1
```

Example **Input**: $n = 5959$ **Steps**:

- Start with $x = \lceil \sqrt{5959} \rceil = 78$.
- Compute $y^2 = x^2 - n = 78^2 - 5959 = 6084 - 5959 = 125$ (not a perfect square).
- Increment x: $x = 79$.
- Compute $y^2 = x^2 - n = 79^2 - 5959 = 6241 - 5959 = 282$ (not a perfect square).
- Increment x: $x = 80$.
- Compute $y^2 = x^2 - n = 80^2 - 5959 = 6400 - 5959 = 441$ (perfect square).
- Compute $y = \sqrt{441} = 21$.
- Factors: $p = x + y = 80 + 21 = 101$, $q = x - y = 80 - 21 = 59$.

Output: $[59, 101]$

Time and Computational Complexity The time complexity depends on the difference between the factors of n:

- Best case: $O(1)$ when the factors are close.
- Worst case: $O(\sqrt{n})$ when factors are far apart.

—

Improved Fermat's Factorization Method

Idea Behind the Improvements The original Fermat's Factorization Method is efficient when the factors of n are close to each other. The following improvements make the algorithm more versatile and efficient:

1. **Handling Even Factors First**: If n is even, repeatedly divide by 2 and add 2 to the list of factors until n becomes odd. This simplifies further factorization and Also solve the problem if one factor is even and the other is odd, the factors can be represented as:

$$\text{factor1} = x + y, \quad \text{factor2} = x - y,$$

where:

$$x = \frac{\text{factor1} + \text{factor2}}{2}, \quad y = \frac{\text{factor1} - \text{factor2}}{2}.$$

If one factor is even, let $\text{factor1} = 2k$ and $\text{factor2} = m$, where k and m are integers. Then:

$$x = \frac{2k + m}{2} = k + \frac{m}{2}.$$

This means x can only take specific values based on the parity of factor1 and factor2.

Algorithm Steps

1. Check if n is even:
   - While n is divisible by 2, divide it by 2 and add 2 to the factors list.
2. Compute $x = \lceil \sqrt{n} \rceil$.

3. Compute $y^2 = x^2 - n$ and check if y^2 is a perfect square:
   - If true, compute $y = \sqrt{y^2}$ and find factors:
     $$\text{factor1} = x + y, \text{ factor2} = x - y.$$
   - If factor1 or factor2 is prime (using Miller-Rabin), add it to the factors list.
   - If not, recursively factorize.
4. Repeat until $n = 1$ or all factors are prime.

Code for Improved Fermat's Factorization Method

```
def improved_fermat_factorization(n):
    """Improved Fermat's Factorization Method."""
    if n <= 1:
        return []
    factors = []

    # Step 1: Handle even factors
    while n % 2 == 0:
        n //= 2
        factors.append(2)
```

```
    # Step 2: Start Fermat's Factorization for odd n
    while n > 1:
        if miller_rabin(n):
            factors.append(n)
            break

        x = math.ceil(math.sqrt(n))
        while True:
            y2 = x * x - n
            if y2 >= 0 and is_perfect_square(y2):
                y = int(math.sqrt(y2))
                factor1, factor2 = x + y, x - y
                if factor1 > 1:
                    factors +=
                        improved_fermat_factorization(
                        factor1)
                if factor2 > 1:
                    factors +=
                        improved_fermat_factorization(
                        factor2)
                n = 1
                break
            x += 1

    return factors
```

Example **Input**: $n = 126$

Steps:

- Remove even factors:

$$126 \div 2 = 63, \quad \text{factors} = [2]$$

- Compute $x = \lceil \sqrt{63} \rceil = 8$.
- Compute $y^2 = 8^2 - 63 = 64 - 63 = 1$ (perfect square).
- Compute $y = \sqrt{1} = 1$, so:

$$\text{factor1} = 8 + 1 = 9, \text{factor2} = 8 - 1 = 7.$$

- Factorize further:

$$9 = 3 \times 3, \quad \text{factors} = [2, 3, 3, 7].$$

Output: $[2, 3, 3, 7]$

Time and Computational Complexity

Time Complexity:

- **Best case:** $O(1)$, when n is even or quickly factorized.
- **Worst case:** $O(\sqrt{n})$, for numbers with no nearby factors.

Space Complexity: $O(\log n)$ for recursive calls during complete factorization.

Generalized Fermat's Factorization Method

Idea Behind the Algorithm The Generalized Fermat's Factorization Method is an extension of Fermat's Factorization Method. Instead of representing n as $x^2 - y^2$, it represents n as:

$$n = a^2 - b^2 \cdot k$$

where k is a nonzero integer that adjusts the difference of squares to improve factorization in cases where Fermat's method might fail. In this implementation, k can either be user-defined or randomly chosen in the range $[0, n-1]$ to explore different possibilities.

Algorithm Steps

1. Choose a value for k:
   - If $k = 0$, a random integer in $[0, n-1]$ is selected.
   - If $k \neq 0$, use the provided value of k.
2. Compute $a = \lceil \sqrt{n+k} \rceil$.
3. Compute $b^2 = a^2 - n$.

4. Check if b^2 is a perfect square:
   - If true, compute $b = \sqrt{b^2}$, then calculate the factors:
     $$p = \gcd(a + b, n),\ q = \gcd(a - b, n)$$
   - If false, increment a and repeat.

Code for Generalized Fermat's Factorization Method

```
def generalized_fermat_factorization(n, k=2):
    """Generalized Fermat's Factorization Method."""
    if k == 0:
        k = random.randint(0, n - 1)
    a = math.ceil(math.sqrt(n + k))
    while True:
        b2 = a * a - n
        if is_perfect_square(b2):
            b = int(math.sqrt(b2))
            p = math.gcd(a + b, n)
            q = math.gcd(a - b, n)
            return (p, q)
        a += 1
```

Example **Input**: $n = 5913$, $k = 4$
Steps:

- Compute $a = \lceil \sqrt{5913 + 4} \rceil = \lceil \sqrt{5917} \rceil = 77$.
- Compute $b^2 = a^2 - n = 77^2 - 5913 = 5929 - 5913 = 16$ (perfect square).
- Compute $b = \sqrt{16} = 4$.
- Calculate factors:
  $$p = \gcd(a + b, n) = \gcd(77 + 4, 5913) = \gcd(81, 5913) = 27$$
  $$q = \gcd(a - b, n) = \gcd(77 - 4, 5913) = \gcd(73, 5913) = 73$$

Output: $[27, 73]$

Key Notes

- If k is not specified or is set to 0, the algorithm randomly selects $k \in [0, n-1]$.
- Randomizing k allows the algorithm to explore different potential factorizations, increasing the likelihood of success for numbers where a specific k might fail.
- The user can also specify k for deterministic behavior, as shown in the example.

Time and Computational Complexity

Time Complexity: The time complexity depends on the value of k:

- Best case: $O(1)$, if a perfect square is found immediately.
- Average case: $O(\sqrt{n})$, as the algorithm iterates through values of a until a valid b^2 is found.

Comparison with Fermat's Method: The introduction of k allows the algorithm to work efficiently even when n does not have factors near $\sqrt{n}$, making it more versatile than the classical Fermat's Factorization Method.

Improved Generalized Fermat Factorization Method Complete Factorization

Idea Behind The Algorithm The Generalized Fermat Factorization Method extends the classic Fermat's method by introducing a shift parameter k, which enables it to handle a broader range of numbers. The key enhancements include:

1. **Randomized Selection of k:** If k is not specified, it is randomly generated in the range $1 \leq k < n$. This improves the method's adaptability to different inputs.
2. **Recursive Complete Factorization:** Once factors are identified, the method recursively factorizes them until all factors are prime.
3. **Miller-Rabin Primality Test:** Before recursive factorization, the primality of the factors is checked to avoid redundant computation.

Algorithm Steps

1. Check if n is even:
   - While n is divisible by 2, divide it by 2 and add 2 to the factors list.
2. Generate a value for k (either specified or random).
3. Compute $a = \lceil \sqrt{n + k} \rceil$.
4. Compute $b^2 = a^2 - n$ and check if b^2 is a perfect square:
   - If true, compute $b = \sqrt{b^2}$ and find factors:

     $$\text{factor1} = \gcd(a + b, n), \text{ factor2} = \gcd(a - b, n).$$

   - If the factors are composite, recursively apply the algorithm to them.
   - If the factors are prime (verified by the Miller-Rabin Primality Test), add them to the factors list.
5. Repeat until $n = 1$ or all factors are prime.

Code for Improved Generalized Fermat Factorization

```
def Improved_Generalised_Fermat(n, k=None):
    factors = []

    while n % 2 == 0:
        n //= 2
        factors.append(2)
```

```
    if n > 1 and miller_rabin(n):
        factors.append(n)
        return factors
    while n > 1:
        if k is None:
            k = random.randint(1, n - 1)
        a = math.ceil(math.sqrt(n + k))
        while True:
            b2 = a * a - n
            if is_perfect_square(b2):
                b = int(math.sqrt(b2))
                factor1 = math.gcd(a + b, n)
                factor2 = math.gcd(a - b, n)
                if miller_rabin(factor1):
                    factors.append(factor1)
                else:
                    factors +=
                        Improved_Generalised_Fermat(
                        factor1)
                if miller_rabin(factor2):
                    factors.append(factor2)
                else:
                    factors +=
                        Improved_Generalised_Fermat(
                        factor2)
                break
            a += 1
        n //= factor1 * factor2
    return factors
```

Example **Input**: $n = 5913$, $k = 4$

Steps:

- Compute $a = \lceil \sqrt{5913 + 4} \rceil = \lceil \sqrt{5917} \rceil = 77$.
- Compute $b^2 = 77^2 - 5913 = 5929 - 5913 = 16$ (a perfect square).
- Compute $b = \sqrt{16} = 4$, and factors:

$$\text{factor1} = \gcd(77 + 4, 5913) = \gcd(81, 5913) = 81,$$

$$\text{factor2} = \gcd(77 - 4, 5913) = \gcd(73, 5913) = 73.$$

- Recursively factorize 81:

$$81 = 3 \times 3 \times 3 \times 3.$$

- Factor 73 is prime.

Output: $5913 = 3^4 \times 73$.

Time and Computational Complexity

Time Complexity:

- **Best Case:** $O(\sqrt{n})$ for small k and quickly factorable numbers.
- **Worst Case:** $O(\sqrt{n} \cdot \log(n))$ with iterative calls to gcd and primality tests.

Space Complexity:

- $O(\log n)$ for recursive calls during complete factorization.

0.3.3 Continued Fraction Algorithm for Integer Factorization

Idea Behind The Algorithm The Continued Fraction Algorithm (CFA) is a method of integer factorization based on the properties of continued fractions and quadratic forms. It builds on the idea of finding a relation between the square of integers modulo n that helps factorize n. Specifically, it leverages the representation of $\sqrt{n}$ as a continued fraction to identify congruences of squares.

Algorithm Steps

1. Compute $\lfloor \sqrt{n} \rfloor$ and initialize the continued fraction representation of $\sqrt{n}$.

2. Generate successive convergents of the continued fraction:

$$\frac{p_k}{q_k} \quad \text{for } k = 1, 2, 3, \ldots$$

3. For each convergent, compute $p_k^2 \mod n$:

$$p_k^2 \equiv q_k^2 \mod n$$

4. Check if $(p_k^2 - q_k^2) \mod n = 0$. If true, compute:

$$\text{factor1} = \gcd(p_k - q_k, n), \quad \text{factor2} = \gcd(p_k + q_k, n).$$

5. If factors are composite, recursively factorize them or use a primality test to determine if they are prime.

6. Repeat until n is completely factorized.

Why It Works The Continued Fraction Algorithm relies on properties of quadratic forms and the relationship between convergents of a continued fraction and modular arithmetic. The key insight is that if a continued fraction convergent $\frac{p_k}{q_k}$ of $\sqrt{n}$ satisfies:

$$p_k^2 \equiv q_k^2 \mod n,$$

then $(p_k^2 - q_k^2) \mod n = 0$, which implies:

$$(p_k - q_k)(p_k + q_k) \equiv 0 \mod n.$$

This equation reveals a potential factorization of n if $\gcd(p_k - q_k, n)$ or $\gcd(p_k + q_k, n)$ produces nontrivial factors.

The algorithm works because continued fraction convergents are excellent approximations of $\sqrt{n}$. These approximations systematically yield values of p_k and q_k that are highly likely to satisfy the above congruence for composite n. This makes it a deterministic method for factorization once a valid convergent is identified.

Additionally, the representation of $\sqrt{n}$ as a periodic continued fraction ensures that the algorithm explores all possible congruences efficiently, increasing the likelihood of identifying factors.

Code for Continued Fraction Algorithm

```
def continued_fraction_factorization(n):
    """Continued Fraction Algorithm for Integer
       Factorization."""
    if n <= 1:
        return []

    factors = []

    # Step 1: Handle even factors
    while n % 2 == 0:
        factors.append(2)
        n //= 2

    if n > 1 and miller_rabin(n):
        factors.append(n)
        return factors

    # Step 2: Initialize continued fraction
       representation
    m = 0
    d = 1
    a0 = math.floor(math.sqrt(n))
    a = a0
    if a * a == n:
        factors.append(a)
        return factors +
           continued_fraction_factorization(n // a)

    # Initialize convergents
    p_prev, p_curr = 1, a
    q_prev, q_curr = 0, 1
```

```
    # Step 3: Begin continued fraction factorization
    for _ in range(1000):
        m = d * a - m
        d = (n - m * m) // d
        a = (a0 + m) // d

        # Compute the next convergent
        p_next = a * p_curr + p_prev
        q_next = a * q_curr + q_prev

        # Update previous values
        p_prev, p_curr = p_curr, p_next
        q_prev, q_curr = q_curr, q_next

        # Check for factorization
        factor1 = gcd(p_curr - q_curr, n)
        factor2 = gcd(p_curr + q_curr, n)
```

```
1
2           if 1 < factor1 < n:
3               if miller_rabin(factor1):
4                   factors.append(factor1)
5               else:
6                   factors +=
                        continued_fraction_factorization(
                        factor1)
7               n //= factor1
8               if n > 1:
9                   factors +=
                        continued_fraction_factorization(n)
10              break
11
12          if 1 < factor2 < n:
13              if miller_rabin(factor2):
14                  factors.append(factor2)
15              else:
16                  factors +=
                        continued_fraction_factorization(
                        factor2)
17              n //= factor2
18              if n > 1:
19                  factors +=
                        continued_fraction_factorization(n)
20              break
21
22      if n > 1:
23          factors.append(n)
24
25      return factors
```

Example **Input**: $n = 8051$

Steps:

- Compute $\lfloor \sqrt{8051} \rfloor = 89$.

- Generate the continued fraction for $\sqrt{8051}$ and compute convergents:
$$\frac{p_1}{q_1} = \frac{89}{1}, \quad \frac{p_2}{q_2} = \frac{178}{2}, \quad \frac{p_3}{q_3} = \dots$$
- For $p_3 = 267, q_3 = 3$:
$$p_3^2 - q_3^2 \mod n = 0 \quad \Longrightarrow \quad (267-3)(267+3) = 3 \times 268.$$
- Compute $\gcd(267-3, 8051) = 13$, and the remaining factor is $8051/13 = 619$.
- Recursively factorize 619 using the same method:
$$619 = 13 \times 47 \,(\text{both primes}).$$

Output: $8051 = 13 \times 13 \times 47$.

Time and Computational Complexity

Time Complexity:

- **Best Case:** $O(\sqrt{n})$ for quickly reducible numbers.
- **Worst Case:** $O(n^{1/4})$ for numbers requiring long continued fraction chains.

Space Complexity:

- $O(\log n)$ for storing convergents.

0.3.4 Pollard's Rho Algorithm

Pollard's Rho Algorithm

Idea Behind the Algorithm Pollard's Rho Algorithm is a probabilistic factorization method that exploits the properties of modular arithmetic and pseudo-random sequences. The algorithm is based on the principle of detecting a non-trivial divisor of a composite number n using the Floyd cycle detection method. By iterating a polynomial function $f(x)$ modulo n, the algorithm generates a sequence of numbers. If two numbers in the sequence share a common factor with n, the gcd function can reveal a factor.

The algorithm is efficient for finding small factors of n and is particularly effective for numbers with multiple small prime factors.

Algorithm Steps

1. **Initialization**: Choose a polynomial $f(x) = x^2 + 1 \mod n$ with a random starting value x_0 and a constant $c = 1$.

2. **Generate Sequence**: Compute the pseudo-random sequence $x_{i+1} = f(x_i) \mod n$.

3. **Cycle Detection**: Use Floyd's cycle detection method to find two sequence values x_i and x_j such that $\gcd(|x_i - x_j|, n)$ produces a non-trivial factor.

4. **Output**: If a factor is found, return it along with its complementary factor. Otherwise, change the initial values and repeat.

Why It Works Pollard's Rho Algorithm works because of the **Pigeonhole Principle** and the periodicity of modular arithmetic. The sequence x_i generated by $f(x)$ modulo n must eventually repeat due to the finite residue set $\{0, 1, \ldots, n-1\}$. The periodicity creates a cycle, and differences between sequence values within the cycle may reveal factors of n when their greatest common divisor with n is computed.

The effectiveness of the algorithm depends on the choice of $f(x)$ and the starting values, which affect how quickly a cycle is detected. Adding randomness improves the chances of finding a factor.

Code for Pollard's Rho Algorithm

```
def pollards_rho(n, k=1):
    """Pollard's Rho Algorithm for integer factorization
       ."""
    if k == -1:  # Randomize if k is -1
        k = random.randint(1, n-1)

    if n % 2 == 0:
        return 2, n // 2
```

```
    # Define the polynomial function
    def f(x):
        return (x**2 + k) % n

    x, y, d = random.randint(1, n-1), random.randint(1,
       n-1), 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = math.gcd(abs(x - y), n)
        if d == n:
            return pollards_rho(n, k=random.randint(1, n
               -1))  # Retry with new parameters
    factor2 = n // d
    return d, factor2
```

Example **Input**: $n = 8051$ **Steps**:

- Start with $f(x) = x^2 + 1 \mod 8051$, $x_0 = 2$, $k = 1$.
- Generate sequence: $x_1 = 5, x_2 = 26, x_3 = 677, \ldots$.
- Use Floyd's cycle detection: Compute $\gcd(|x_i - x_j|, 8051)$.
- Detect a factor: $\gcd(|677 - 5|, 8051) = 97$.

Output: The factors of 8051 are 97 and 83 ($8051/97 = 83$).

Time and Computational Complexity

- **Time Complexity**: $O(\sqrt{p})$, where p is the smallest prime factor of n. The performance depends on finding a collision in the sequence modulo p.
- **Space Complexity**: $O(1)$, as the algorithm uses only a few variables to maintain the sequence and compute the gcd.

Improved Pollard's Rho Algorithm

The improvements to the Pollard's Rho Algorithm include:

- **Primality Test Integration**: The Miller-Rabin Primality Test is used to check if the factors are prime.
- **Recursive Factorization**: Once a non-trivial factor is found, it is recursively factorized until the complete prime factorization of the number is obtained.
- **Handling Edge Cases**: If the algorithm fails to find a factor (when gcd equals n), it retries with new parameters.

Code for Improved Pollard's Rho Algorithm

```
def pollards_rho_improved(n, k=1):
    """Improved Pollard's Rho Algorithm with recursive
       factorization."""

    if k == -1:  # Randomize if k is -1
        k = random.randint(1, n-1)

    if n <= 1:
        return []
    if miller_rabin(n):  # Use predefined Miller-Rabin
       Test
        return [n]

    # Define the polynomial function
    def f(x):
        return (x**2 + k) % n
```

```
    x, y, d = random.randint(1, n - 1), random.randint
        (1, n - 1), 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = math.gcd(abs(x - y), n)
        if d == n:
            # Retry with different parameters
            return pollards_rho_improved(n)

    # Recursive factorization for complete prime
        factorization
    factors = pollards_rho_improved(d) +
        pollards_rho_improved(n // d)
    return sorted(factors)
```

Code Explanation

- The `miller_rabin` function is used to verify the primality of factors found during the factorization process.
- The polynomial function $f(x) = x^2 + 1 \mod n$ generates pseudo-random sequences, and the gcd of sequence values detects non-trivial factors of n.
- Factors are recursively factorized until the complete prime factorization of n is achieved.
- If no factor is found, the algorithm retries with new random initial values.

Example **Input**: $n = 8051$ **Steps**:

- Use the polynomial function $f(x) = x^2 + 1 \mod 8051$, starting with random x_0 and y_0 values.
- Compute $\gcd(|x_i - x_j|, n)$ to detect a factor. In this case, $\gcd(677 - 5, 8051) = 97$.
- Verify 97 as prime using the Miller-Rabin Primality Test.

- Compute $n/97 = 83$ and verify 83 as prime.
- The factors of 8051 are $[83, 97]$.

Output: $[83, 97]$

Time and Computational Complexity

- **Time Complexity**: $O(k \cdot \sqrt{p})$, where p is the smallest prime factor of n, and k is the number of Miller-Rabin test iterations.
- **Space Complexity**: $O(1)$, as the algorithm uses only a few variables to manage the sequence and gcd computations.

0.3.5 Quadratic Sieve (QS)

The Quadratic Sieve (QS) is a number factorization algorithm that is particularly efficient for factoring large composite numbers, especially those with 100–200 digits. The algorithm works by finding smooth numbers (numbers whose prime factors are all smaller than a certain bound) and using these smooth numbers to construct a system of linear equations. Solving these equations then reveals a non-trivial factor of the number being factored.

The implementation of this algorithm incorporates various number-theoretic techniques such as Rabin-Miller primality test, Pollard's Rho, Brent's variant of Pollard's Rho, trial division, and others to identify small prime factors. These small prime factors are subsequently used in the Quadratic Sieve for finding larger factors.

Algorithm Description The Quadratic Sieve follows a series of steps:

- **Primality Testing:** The Rabin-Miller primality test is employed to check whether a number is prime.
- **Trial Division:** Small prime factors are found using trial division.
- **Perfect Power Check:** If the number is a perfect power, the factorization process is halted early.
- **Pollard's Rho Algorithm:** If small factors are not found, Pollard's Rho algorithm is used to attempt to find factors.

- **Main Factorization:** The core factorization process involves combining the methods above to eventually find the prime factors.

Algorithm Steps

1. **Initialization:** The algorithm starts by initializing an array of primes up to a certain bound.
2. **Smooth Number Search:** The algorithm searches for smooth numbers by evaluating quadratic forms.
3. **Factorization using Linear Algebra:** The smooth numbers found are then used to create a system of linear equations modulo 2. Solving these equations identifies a factor of the number.
4. **Solution:** The resulting linear combination provides a factor, which is used to divide the original number.

GitHub Repository

https://github.com/NachiketUN/Quadratic-Sieve-Algorithm

Code Explanation

- ***is_probable_prime(a)***:
  This function checks whether a number a is prime using the Rabin-Miller primality test. It first performs basic checks (such as checking if a is 2 or even), and if these fail, it proceeds with the Rabin-Miller test for probabilistic primality.
- ***rabin_miller_primality_test(a, iterations)***:
  This function implements the Rabin-Miller primality test. It performs modular exponentiation and checks randomly chosen bases to verify the primality of a. The test is repeated for a set number of iterations to increase accuracy.
- ***check_perfect_power(n)***:
  This function checks whether the number n is a perfect power, i.e., if $n = r^b$ for some integers r and b. If n is a perfect power, the function returns the pair (r, b); otherwise, it returns None.

- ***check_factor(n, i, factors)***:
  This function checks if i is a factor of n by trial division. It divides n by i until no further division is possible, adding i to the list of factors each time.

- ***find_small_primes(n, upper_bound)***:
  This function finds small primes up to a given upper bound and uses them to attempt to factor n. It performs trial division using these primes to find any small prime factors of n.

- ***brent_factorise(n, iterations=None)***:
  This function implements Brent's variant of Pollard's Rho algorithm. It attempts to find a non-trivial factor of n by iterating through random values and applying a specific transformation. If a factor is found, it is returned.

- ***pollard_brent_iterator(n, factors)***:
  This function serves as an iterator for Brent's Pollard Rho algorithm, repeatedly attempting to find small prime factors of n and adding them to the list of factors.

- ***factorise(n)***:
  This is the main function for factorizing n. It first attempts trial division, then checks for perfect powers, and if no small factors are found, it applies Pollard's Rho algorithm. The function recursively finds factors until all prime factors of n are discovered.

- ***main()***:
  This function serves as the entry point for the program. It accepts an integer n as input, calls the factorization function, and prints the resulting prime factors.

Improvement

- **Integration with Quadratic Sieve:** The algorithm can be extended to use the Self-Initializing Quadratic Sieve (SIQS), which collects smooth numbers and solves a system of linear equations to identify non-trivial factors.

- **Optimization:**

  - **Multithreading:** Parallel processing can be used to speed up tasks such as trial division and Pollard's Rho factorization.
  - **Better Primality Testing:** The Miller-Rabin test can be optimized by increasing the number of iterations for more accurate results when testing large numbers.

- **Testing with Large Inputs:** For very large numbers (e.g., hundreds of digits), the General Number Field Sieve (GNFS) could be integrated for better performance in large-scale factorizations.

Conclusion This Python implementation provides an efficient approach to integer factorization using several advanced techniques such as the Rabin-Miller primality test, Pollard's Rho, and trial division. It can be further enhanced with the integration of the Self-Initializing Quadratic Sieve for large numbers and parallel processing for faster execution. The code demonstrates the power of combining multiple number-theoretic methods to solve the complex problem of integer factorization efficiently.

0.3.6 General Number Field Sieve (GNFS)

The General Number Field Sieve (GNFS) is one of the most advanced algorithms used for factoring large composite numbers. It is particularly efficient for numbers with hundreds of digits and is considered the most effective classical algorithm for integer factorization for such large numbers. The GNFS works by finding relations between certain number fields and solving systems of linear equations over finite fields.

The GNFS is based on algebraic number theory and involves several key phases: polynomial selection, sieving, matrix solving, and square root extraction. This process involves finding smooth numbers (numbers whose prime factors are all smaller than a certain bound) and using these to solve a system of linear equations, which ultimately reveals factors of the target number.

Algorithm Description The GNFS works in the following sequence of steps:

- **Polynomial Selection:** A polynomial is chosen that has small integer coefficients and whose roots can be found modulo the number to be factored. This polynomial will be used in the sieving step.

- **Sieving:** A large number of values are tested to determine if they are smooth with respect to the prime factors.
- **Linear Algebra:** The relations found during the sieving step are used to construct a matrix, which is then solved using linear algebra techniques (such as Gaussian elimination) to find dependencies between the relations.
- **Square Root Extraction:** The final step involves taking the square root of the numbers obtained through the linear algebra step to find the factors of the composite number.

Code Explanation The GNFS implementation in C is used for large-scale integer factorization. Although Python implementations exist, the C implementation is widely regarded as more efficient for computationally intensive tasks due to its lower-level optimizations and better memory management.

GitHub Repository The C-based GNFS code is available in the following GitHub repository:

https://github.com/MathSquared/general-number-field-sieve

This implementation contains the full C code required to perform GNFS on large composite numbers. It includes optimizations and methods for polynomial selection, sieving, and matrix solving.

Implementation Highlights

- **Efficient Sieving:** The code uses advanced sieving techniques to search for smooth numbers, a critical step in the GNFS.
- **Linear Algebra Optimization:** The matrix solving phase leverages highly optimized linear algebra routines to handle large matrices that arise during the factorization process.
- **Scalability:** This C implementation is designed to scale well with very large numbers, taking full advantage of the computational resources available.

Improvement

- **Parallelization:** The GNFS implementation can be further enhanced by parallelizing the sieving and linear algebra phases to take advantage of multi-core processors and distributed computing environments.
- **Integration with Python:** Although the GNFS is implemented in C, it could be interfaced with Python using tools such as Cython or the 'ctypes' library for easier integration with other Python-based algorithms or applications.

Conclusion The General Number Field Sieve is one of the most powerful integer factorization algorithms available, especially for very large numbers. While the implementation is in C for performance reasons, the algorithm provides a foundation for factoring large integers and can be integrated into larger systems for cryptography or mathematical research. The GitHub repository provides an efficient and optimized version of GNFS suitable for practical use.

0.4 Comparison of Algorithms

In this section, we compare the various factorization algorithms discussed in this report: Trial Division, Wheel Factorization, Fermat's Method, Pollard's Rho, Quadratic Sieve, and General Number Field Sieve (GNFS). Each of these algorithms has its strengths and weaknesses, making them suitable for different types of numbers and use cases. Below is a comparison based on several criteria.

0.4.1 Comparison Criteria

We compare the algorithms based on the following parameters:

- **Time Complexity:** The asymptotic time complexity that describes how the algorithm scales with the size of the input.
- **Efficiency:** Practical performance, including how quickly the algorithm performs for small, medium, and large inputs.

- **Suitability for Large Numbers:** The algorithm's effectiveness when dealing with very large composite numbers, such as those used in cryptographic applications.
- **Parallelism:** The ability to parallelize or distribute the computation across multiple processors or machines.
- **Ease of Implementation:** The complexity of implementing the algorithm from scratch.

0.4.2 Algorithm Comparison Table

Algorithm	Time Complexity	Efficiency	Suitability for Large Numbers	Parallelism	Ease of Implementation
Trial Division	$O(\sqrt{n})$	Low for large numbers	Poor for large numbers	No	Easy
Wheel Factorization	$O(\sqrt{n}/\log n)$	Moderate	Poor for large numbers	No	Moderate
Fermat's Method	$O(n^{1/4})$	Moderate	Poor for large numbers	No	Moderate
Pollard's Rho	$O(n^{1/4})$	Moderate to High	Poor to Moderate for large numbers	Yes (limited)	Moderate
Quadratic Sieve	$O(e^{\sqrt{\ln n \ln \ln n}})$	High for numbers up to 100–200 digits	Moderate for large numbers	Yes	Advanced
General Number Field Sieve (GNFS)	$O\left(e^{(64/9)^{1/3}(\ln n)^{1/3}(\ln \ln n)^{2/3}}\right)$	Very High for very large numbers	Excellent for numbers over 100 digits	Yes (high parallelism)	Very Advanced

0.4.3 Discussion

- **Trial Division:** This is the simplest algorithm, but it becomes inefficient for large numbers due to its $O(\sqrt{n})$ time complexity. It is suitable for small numbers or as a preprocessing step in other algorithms.
- **Wheel Factorization:** This improves on trial division by skipping multiples of small primes, but its performance is still limited for large numbers. It can be used to optimize trial division but is not suitable for numbers with many large factors.
- **Fermat's Method:** While it has a better time complexity of $O(n^{1/4})$, it is still inefficient for larger numbers and may not work well for numbers that are not the product of two primes close to each other.
- **Pollard's Rho:** A more efficient algorithm with a time complexity of $O(n^{1/4})$, Pollard's Rho is useful for finding smaller factors of large composite numbers. However, its performance can degrade as the number size increases.

- **Quadratic Sieve:** This algorithm is very efficient for numbers with 100–200 digits, offering significant improvements over previous methods. It is one of the best algorithms for numbers of this size but may not perform well for numbers larger than 200 digits.
- **General Number Field Sieve (GNFS):** GNFS is the most efficient algorithm for very large numbers, particularly those with over 100 digits. It has an excellent time complexity and scalability, making it the best choice for cryptographic applications like RSA encryption. However, it is very complex and requires significant computational resources, as well as advanced parallelization techniques.

0.4.4 Conclusion

The choice of factorization algorithm depends heavily on the size and nature of the number being factored. For small to medium-sized numbers, methods like trial division, Pollard's Rho, and the Quadratic Sieve are effective. For very large numbers, GNFS stands out as the most efficient method, although it requires specialized knowledge and computational power to implement. The comparison highlights the trade-offs between simplicity, efficiency, and suitability for large inputs, helping guide the selection of the best algorithm for specific use cases.

Bibliography

[1] Wikipedia. (n.d.). *Wheel Factorization*. Retrieved from https://en.m.wikipedia.org/wiki/Wheel_factorization

[2] Primality Testing & Factorization. (n.d.). YouTube Playlist. Retrieved from https://youtube.com/playlist?list=PLLtQL9wSL16gTZ-OjCJNHe8E6GWPWpvRi&si=JvKOi26F7YGxWDkG

[3] Wikipedia. (n.d.). *Pollard's Rho Algorithm*. Retrieved from https://en.m.wikipedia.org/wiki/Pollard%27s_rho_algorithm

[4] NachiketUN. (n.d.). *Quadratic Sieve Algorithm Implementation*. GitHub Repository. Retrieved from https://github.com/NachiketUN/Quadratic-Sieve-Algorithm

[5] General Number Field Sieve Implementation. (n.d.). GitHub Repository. Retrieved from https://github.com/MathSquared/general-number-field-sieve