Integer Factorization Algorithms

Yiheng Su

May 2022

1 Introduction

Integer factorization is the decomposition of a composite number into a product of smaller integers. "Can integer factorization be solved in polynomial times?" It is a famous unsolved problem in computer science and mathematics. Till today, no classical algorithm[1] is known that can factor integers in polynomial time. However, neither the non-existence of such algorithms has been proved[1].

In this writing assignment, I will introduce five integer factorization algorithms and compare the runtimes between them. Firstly, I will describe the most straightforward algorithm, the trial division, and the wheel factorization improvement. Secondly, I will describe Fermat's factorization method and its improvement, Kraitchik-Fermat's factorization method. Thirdly, I will focus on a more intriguing and complicated algorithm called Pollard's $p-1$ factorization. Lastly, I will compare the runtimes between different algorithms on factoring various composite integers.

2 Definitions

1. **Semiprime**: a semiprime is a natural number that is the product of exactly two prime numbers.
2. **d-digit number**: An d digit number is a positive number with exactly d digits. For example, 1234 is a 4-digit number.
3. **Trivial and Nontrivial Factor**: For any number n, trivial factors are: ± 1 and $\pm n$. Any other factor, if it exists, is nontrivial factor.

[1]We do not consider any quantum integer factorization algorithm in this writing assignment.

3 Algorithms

3.1 Trial Division

Trial division is the most straightforward algorithm for factoring an integer. In the trial division, we need to divide a number n successively by the element in the sequence $S = \{2, 3, 4, ..., \lfloor\sqrt{n}\rfloor\}$ until we find a divisor. The following theorem guarantees that the upper bound for the testing numbers is $\lfloor\sqrt{n}\rfloor$.

Theorem 3.1. *If $n = pq$, p and q are nontrivial factors of n and $p \leq q$, then $p \leq \sqrt{n}$ [2].*

Proof. Suppose, by contradiction that, $p > \sqrt{n}$. Then, $q \geq p > \sqrt{n}$. Hence, $pq > \sqrt{n} \cdot \sqrt{n} = n$. This contradicts with $n = pq$. □

According to Theorem 3.1, we can design a trial division algorithm by testing all numbers from 2 to $\lfloor\sqrt{n}\rfloor$.

3.2 Trial Division: Algorithm

Algorithm 1 Trial Division

```
Require: Give a composite number $n \in \mathbb{N}$.
Ensure: Find a nontrivial factor of $n$.
  $f \leftarrow 2$
  while $f * f < n$ do
      if $n \equiv 0 \pmod{f}$ then
          return $f$
      end if
      $f \leftarrow f + 1$
  end while
  return $n$
```

3.3 Wheel Factorization

Wheel factorization is an improvement of the trial division. In the trial division, we need to divide a number n successively by the element in the sequence $S = \{2, 3, 4, ..., \lfloor n\rfloor\}$ until we find a divisor. For the wheel factorization, we create a sequence of the first few primes, called the **basis**, and then we generate a sequence, called the **wheel**, of integers that are relatively prime to all primes of the basis[3]. The basis, B, and the wheel, W, are all subsequences of S. Then, we divide n successively by the element in the wheel until we find a divisor. The following theorem and Chinese Reminder Theorem helps us generate W.

Theorem 3.2. *Suppose p is a prime, if $pm|n$ for some $m \in \mathbb{N}$ and $m \geq 1$, then $p|n$.*

This theorem directly follows the transitivity property of divisibility. The contrapositive statement of Theorem 3.2 gives us the essential idea of the wheel factorization. If n is not divisible by a prime p, then n is not divisible by any scalar multiple of p.

For example, when $B = \{2, 3\}$, notice that if n is not divisible by 2, n is not divisible by $2m$ for all $m \in \mathbb{N}$. If n is also not divisible by 3, n is not divisible by $3m$ for all $m \in \mathbb{N}$. By Chinese Reminder Theorem, suppose $x \in S$,

$$\begin{cases} x \equiv 1 \pmod 2 \\ x \equiv 1 \pmod 3 \end{cases} \text{and} \begin{cases} x \equiv 1 \pmod 2 \\ x \equiv 2 \pmod 3 \end{cases},$$

all numbers in W are either $x \equiv 1 \pmod 6$ or $x \equiv 5 \pmod 5$. It indicates that for every 6 consecutive numbers in S, only two of them are in W. Hence, the method leaves about 34% numbers in S to be checked. Moreover, if $B = \{2, 3, 5\}$ and n is not divisible by all numbers in B, then by CRT, we only need to check $(2 - 1) \times (3 - 1) \times (5 - 1) = 8$ numbers for every $2 \times 3 \times 5 = 30$ consecutive numbers in S. The method leaves roughly 27% numbers in S to be tested!

We can extend this to even larger primes $p > 5$. As long as $p < \lfloor n \rfloor$, the method will reduce the length of W. Although extending to larger primes may reduce the length of W, it increases the length of B at the same time. The wheel factorization becomes more and more similar to the trial division as we increase the length of B.

3.4 Wheel Factorization: Algorithm

Here is an algorithm for the wheel factorization with the basis $B = \{2, 3\}$.

Algorithm 2 Wheel Division

```
Require: Give a composite number $n \in \mathbb{N}$.
Ensure: Find a nontrivial factor of $n$.
  if $n \equiv 0 \pmod 2$ then
      return 2
  end if
  if $n \equiv 0 \pmod 3$ then
      return 3
  end if
  increments $\leftarrow \{2, 4\}$
  index $\leftarrow 0$
  while $f * f < n$ do
      if $n \equiv 0 \pmod f$ then
          return $f$
      end if
      $f \leftarrow f +$ increments[index $\pmod 2$]
      index $=$ index $+ 1$
  end while
  return $n$
```

The numbers, x, in the wheel are either $x \equiv 1 \pmod 6$ or $x \equiv 5 \pmod 6$. Hence, each time, we need to increase our factor variable f by either 2 or 4, because $5+2 \equiv 1 \pmod 6$ and $1+4 \equiv 5 \pmod 6$. This is the major difference and improvement from the trial division.

3.5 Fermat's Factorization Method

Fermat's factorization method is based on the following theorem we proved in homework:

Theorem 3.3. *Every odd integer is the difference of two squares.*

Given an odd number n, Fermat's factorization method is looking for integer x and y such that

$$n = x^2 - y^2 = (x + y)(x - y).$$

Conversely, if $n = ab$, $a > b \geq 1$, then we have

$$n = (\frac{a + b}{2})^2 - (\frac{a - b}{2})^2.$$

Notice that n is odd, so a, b must be odd. $\frac{a+b}{2}$ and $\frac{a-b}{2}$ are non negative integers.

For an algorithm, we first need to find the smallest k such that $k^2 \geq n$. We create a sequence

$$F = \{k^2 - n, (k + 1)^2 - n, (k + 2)^2 - n, ..., (\frac{n + 1}{2})^2 - n\}.$$

Then, we successively test whether numbers in F is a square. Notice that the sequence F ends at $(\frac{n+1}{2})^2 - n$ because

$$(\frac{n+1}{2})^2 - n = (\frac{n-1}{2})^2,$$

which is a square.

3.6 Fermat's Factorization Method: Algorithm

Since Fermat's factorization may not work for even numbers, we need to exclude the even number at the beginning of the algorithm.

Algorithm 3 Fermat's Factorization Method

```
Require: Give a composite number $n \in \mathbb{N}$.
Ensure: Find a nontrivial factor of $n$.
  if $n \equiv 0 \pmod 2$ then
      return 2
  end if
  if $n$ is a square then
      return square root of $n$
  end if
  $k \leftarrow$ ceiling of square root of $n$
  while $k^2 - n$ is not a square do
      $f = f + 1$
  end while
  $y \leftarrow$ square root of $k^2 - n$
  $x \leftarrow k$
  return $x + y$
```

3.7 Kraitchik-Fermat's Factorization Method

Kraitchik-Fermat's factorization is an improvement for Fermat's factorization invented by Manuruce Kraitchik[4]. Instead of looking for possible x, y satisfying $n = x^2 - y^2$, we want to find x, y satisfying $n | x^2 - y^2$, i.e.,

$$x^2 \equiv y^2 \pmod n.$$

Then, the following theorem can help us find factors:

Theorem 3.4. *If $x^2 \equiv y^2 \pmod n$ and $x \not\equiv \pm y \pmod n$, then $gcd(n, x+y)$ and $gcd(n, x-y)$ is not 1 or n.*

Proof. By contradiction, if $gcd(n, x+y) = 1$ and $n|(x+y)(x-y)$, then $n|x-y$. However, $x = y \pmod n$ contradicts with $x \not\equiv \pm y \pmod n$. If $gcd(n, x+y) = n$, then $n|x+y$, which contradicts with $x \not\equiv \pm y \pmod n$. The proof for $gcd(n, x-y) \neq 1$ or n will be the same as above. □

For Kraitchik-Fermat's factorization, we are looking for x, y satisfying

$$y^2 = x^2 - t \cdot n, \tag{1}$$

where $t \in \mathbb{N}$ and $t \cdot n < x^2$. If integers x, y satisfy equation 1 and $x + y \not\equiv 0 \pmod{n}$ and $x - y \not\equiv 0 \pmod{n}$, we find a pair of nontrivial factors $gcd(x - y, n)$ and $gcd(x + y, n)$.

3.8 Kraitchik-Fermat's Factorization Method: Algorithm

Algorithm 4 Kraitchik-Fermat's Factorization Method

```
Require: Give a composite number $n \in \mathbb{N}$.
Ensure: Find a nontrivial factor of $n$.
  if $n \equiv 0 \pmod{2}$ then
      return $2$
  end if
  $k \leftarrow$ ceiling of square root of $n$
  while True do
      $t \leftarrow 1$
      $y_2 \leftarrow k^2 - t \cdot n$
      while $y_2 \geq 0$ do
          $y_2 \leftarrow k^2 - t \cdot n$
          if $y_2$ is a square then
              $y \leftarrow \sqrt{y_2}$
              if $x + y \not\equiv 0 \pmod{n}$ and $x - y \not\equiv 0 \pmod{n}$ then
                  $x \leftarrow k$
                  return $gcd(x - y, n)$
              end if
          end if
          $t \leftarrow t + 1$
      end while
      $k \leftarrow k + 1$
  end while
```

If the input n is a prime, the algorithm will not terminate. Hence, we need to use the Primality test to ensure n is not a prime.

3.9 Pollard's p-1 Factorization

The Pollard's $p - 1$ factorization is based on Fermat Little Theorem:

Theorem 3.5. *Let p be a prime number, and let a be an integer. If $gcd(a, p) = 1$, then*

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

for all $a \in \mathbb{Z}$ and any prime p.

Suppose that n has a prime factor p. Suppose that $p-1$ is a factor of some integer M. It implies that $M = (p-1)k$ for some integer k. By Fermat Little Theorem, we know that for all integers a relatively prime to p and for all positive integer k:

$$a^{k(p-1)} \equiv (a^{p-1})^k \equiv 1^k \equiv 1 \pmod p.$$

Thus, we have $a^M \equiv 1 \pmod p$. Since $p | a^M - 1$ and $p|n$, we have found that p is a factor of n[5]. However, we do not know p is a factor of n at first when factoring n. We need to compute a^M with $M = 1, 2, 3, ...$ until we got

$$1 < gcd(a^M, n) < n.$$

Notice it would be inefficient to compute a^M with $M \in \mathbb{N}$. To improve it, we can choose M to be a product of many small primes. And then, it is more likely for us to meet the condition $M = k(p-1)$.

For an algorithm for computer, it would be easier to compute $a^M!$ for $M \in \mathbb{N}$, instead of looking for many small primes. The factorial of a large M is likely to contain a product of small primes. Then, we compute

$$gcd(a^M - 1 \pmod n), n).$$

The advantage of computing modular is that modular exponentiation is efficient to compute, even for huge integers. However, one weakness of using factorial is that M may not include enough large powers of small primes, so the method may fail if the factorization of $p-1$ contains a prime to large power.

Given an integer n, the method above still does not always guarantee to produce a nontrivial factor for n. For example, if n is prime, $gcd(n, a^M - 1) = 1$ for all $M \in \mathbb{N}$. Hence, the algorithm never finds a nontrivial factor, and never terminates. For another example, let $n = 65, a = 2$. Notice that when $M = 4!$,

$$\begin{aligned} &2^{4!} - 1 \equiv 16777215 \equiv 0 \pmod{65} \\ &gcd(0, 65) = 65. \end{aligned}$$

For any $M > 4!$, $gcd(a^M - 1 \pmod{65}, 65) = 65$[2]. Thus, the algorithm never terminates.

To avoid the cases above to happen, we need to improve our algorithm. Firstly, we can use MillerRabin primality test to ensure the input is not prime. Secondly, we need to set a maximal iteration B for our algorithm so that it would not run forever. Here is the improved algorithm[5]:

1. Use a primality test to ensure the input is not prime.

2. Let the max iteration $B = 10^5$, $a = 2$, and $M = 2$.

3. Compute $c = gcd(a, n)$. If $c \neq 1$, return c. If $c = 1$, go to step 4.

4. Compute $a^{M!} \pmod n$.

5. Compute $d = gcd(a^{M!} - 1 \pmod{n}, n)$. If $1 < d < n$, return d. Else, go to step 6.
6. If not reach max iteration, $M = M + 1$ and go to step 4. If reach the max iteration and $d = 1$, increase B and go to step 4. If reach the max iteration and $d = n$, change a's value and go to step 2.

3.10 Pollard's p-1 Factorization: Algorithm

Algorithm 5 Pollard's p-1 Factorization

```
Require: Give a composite number $n \in \mathbb{N}$.
Ensure: Find a nontrivial factor of $n$.
  if $n \equiv 0 \pmod{2}$ then
      return 2
  end if
  $a \leftarrow 2$
  $M \leftarrow 2$
  $B \leftarrow 10^5$                          ▷ Set the max iteration for the algorithm.
  while True do
      $a \leftarrow a^M \pmod{n}$     ▷ Use modular exponentiation to compute $a^{M!} \pmod{n}$.
      $d \leftarrow gcd(a - 1, n)$
      if $d > 1$ then
          return d
      end if
      if Reach the max iteration $B$ and $d = n$ then
          Choose another $a$ where $gcd(a, n) = 1$
          Reset $M$ to 2 and continue
      end if
      $M \leftarrow M + 1$
  end while
```

I exclude the case of even number at the beginning of the algorithm. Hence, I can safely set $a = 2$ because of $gcd(a, n) = 1$. I set the max iteration to be 10^5. Although bigger B can increase the probability of finding a factor, it increases the algorithm's runtime at the same time[5].

4 Runtime Comparisons

I have implemented all integer factorization methods above in Python. My programs may not be the optimal implementations, but I have tried my best to optimize them. The codes for algorithms are in Appendix A. The codes for simulations and graphs are in Appendix B.

I want to analyze the runtimes of trial division versus wheel factorization and Fermat's versus Kraitchik-Fermat's factorization separately, because they are different special-purpose factoring algorithms. Trial and wheel factorization are efficient to factor numbers with small factors, while Fermat's and Kraitchik-Fermat's factorization are good at factoring the number n with factors near $\sqrt{n}$. Hence, those two types of factorization methods are not comparable.

Lastly, I want to analyze the runtimes of all my algorithms on factoring semiprimes because one of the significant uses of factorization methods is to conquer RSA problems.

4.1 Trial Division and Wheel Factorization

This section compares trial division runtimes with wheel factorization runtimes. Figure 1 shows the number of decimal digits d in the inputs versus the average runtimes in seconds. I randomly generated 50 d-digit numbers for each d from 2 to 15 and computed the average runtimes on factoring each group of 50 d-digit numbers. From figure 1, we can see that both algorithms work fast when inputs are small. Overall, the wheel factorization is faster than the trial division. The results meet my expectation, because I have analyzed that my wheel factorization will be 66.67% more efficient than my trial division.

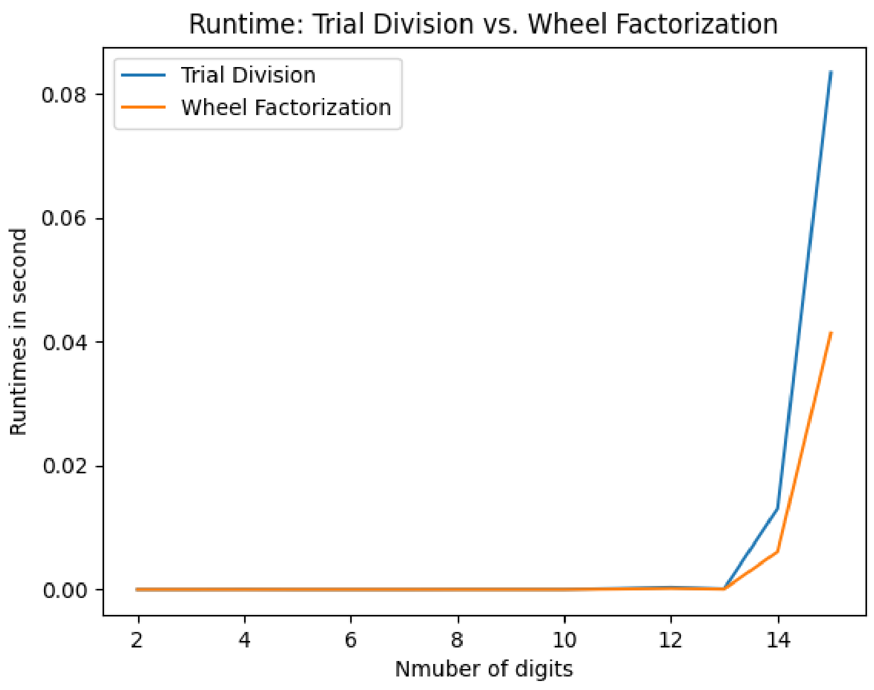

Figure 1: TRIAL DIVISION VS. WHEEL FACTORIZATION

4.2 Fermat's Factorization and Kraitchik-Fermat's Factorization

This section compares Fermat's factorization runtimes with Kraitchik-Fermat's factorization runtimes. Figure 2 shows the number of decimal digits d in the inputs versus the average runtimes in seconds. I randomly generated 50 d-digit for each d from 2 to 5 and computed the average runtimes for each group of 50 d-digit numbers. I set the max d to be five because my Kraitchik-Fermat's factorization becomes really inefficient to factor some large numbers containing only small prime factors. For example, my Kraitchik-Fermat's factorization takes 74.79 seconds to find a factor of $396687 = 3 \times 132229$! Figure 2 shows that the runtime for my Kraitchik-Fermat's factorization rises steeply from 4-digit to 5-digit numbers, while my Fermat's algorithm is efficient at factoring small numbers.

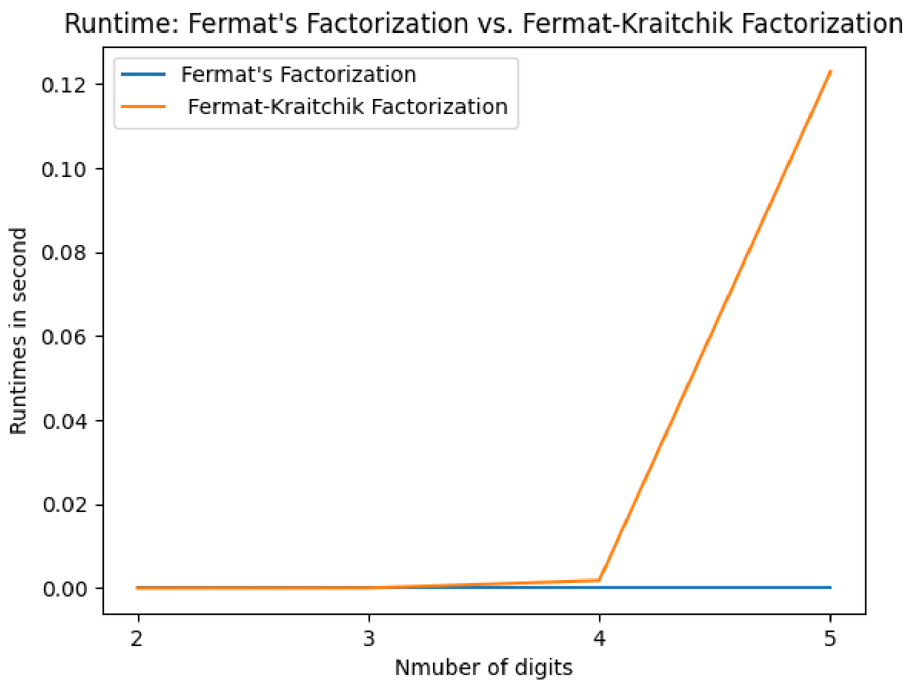

Figure 2: FERMAT'S FACTORIZATION VS. KRAITCHIK-FERMAT'S FACTORIZATION

4.3 Runtimes of Factoring Semiprimes

One real-world application of the integer factorization method is to solve RSA problems. Given a large semiprime s, we need to find two prime factors of s. I ran two simulations with two different kinds of semiprimes versus each algorithm's runtimes on factoring them. I did not test my Kraitchik-Fermat's factorization in this simulation because of its inefficiency at factoring large numbers.

In the first simulation, I used semiprimes in the forms of $s_1 = p_1 q_1$, where p_1 and q_1 are both randomly generated d-digit primes. Hence, p_1 and q_1 are close to each other. Figure 3 shows the scatter plots of inputs versus runtimes for all algorithms. As I expected, Fermat's method works much better than trial and wheel factorization methods because it starts by looking for factors near $\sqrt{n}$. I was a little surprised to see Pollard's method did best in this simulation. I think it is because of the fast exponential growth of $a^{M!}$ in Pollard's method. Moreover, the power of p_1 and q_1 in $s_1 = p_1 q_1$ is 1, so it is more likely for $a^{M!} - 1$ has a factor p_1 or q_1. There are strong positive linear relationships between the size of input and runtimes of trial and wheel factorization. The runtimes of them increase steadily as the increase of input integer.

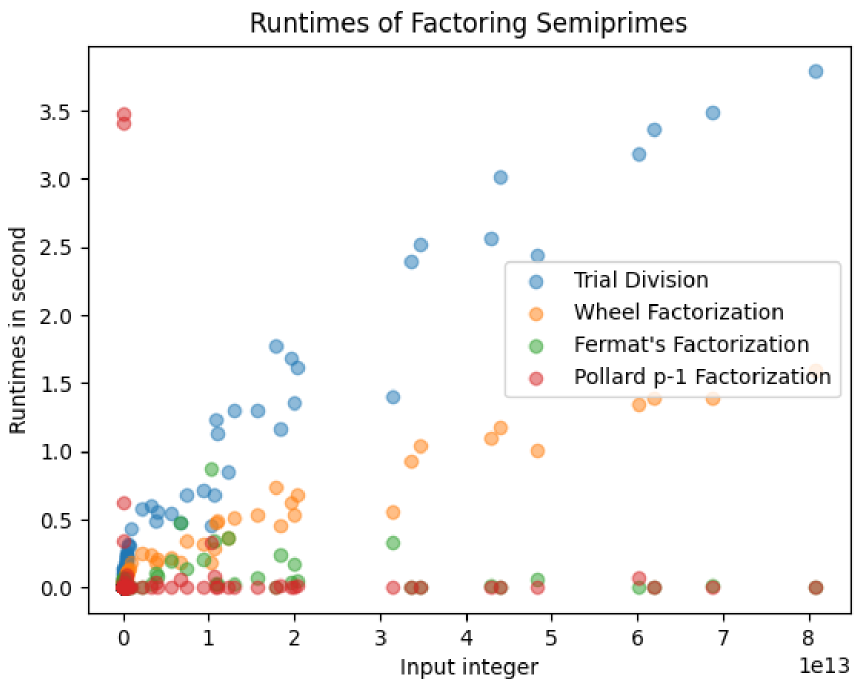

Figure 3: RUNTIMES OF FACTORING SEMIPRIMES

In my second simulation, I used semiprimes $s_2 = p_2 q_2$ with random primes p_2 and q_2 between 10 and 10^8. Figure 4 shows the scatter plots of inputs s_2 versus runtimes for all algorithms. The figure shows that Pollard's method is still the fastest for most inputs, expect some outliers for small inputs. However, Fermat's method spreads out across the x axis and is generally slower than other methods.

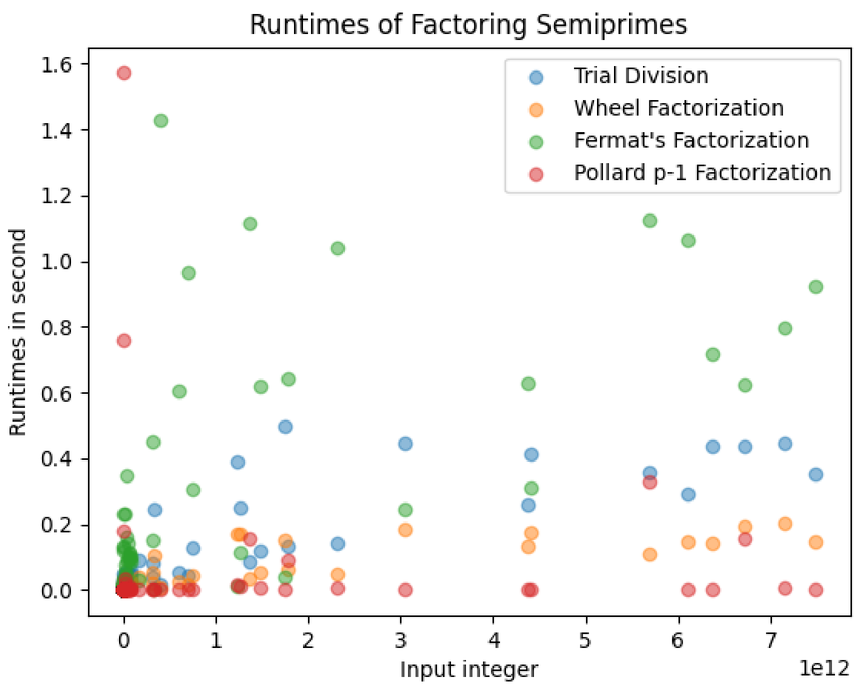

Figure 4: RUNTIMES OF FACTORING SEMIPRIMES

4.4 Conclusion:

In this writing assignment, I have discussed five integer factorization methods, trial division, wheel factorization, Fermat's factorization, Kraitchik-Fermat's factorization, and Pollard $p-1$ factorization. I have implemented them in Python and compared their runtimes on factoring various integers and semiprimes.

After several simulations, I discovered that trial division and wheel factorization work most efficiently when the input number n contains small factors. Fermat's and Kraitchik-Fermat's methods work most efficiently when the input number n has factors near $\sqrt{n}$. Pollard $p-1$ method performs consistently well for various input integers.

I only investigated a small part of the area of integer factorization algorithms. I did not have the opportunity to explore other mathematically challenging factorization algorithms, such as Quadratic Sieve and the General Number Field Sieve. I am also not able to analyze the time complexity for each algorithm because I have not learned much computational theories.

Overall, I enjoy writing this project. I enjoy learning number theory knowledge behind factorization methods, developing algorithms for them, and writing computer programs and simulations.

5 Acknowledgements

At last, I would like to thank Prof. Fernando Gouvea for explaining the mathematical theories behind algorithms. Thank you very much for your number theory class. I appreciate you and your time helping me during office hours and in the class.

6 Appendix A

```
import gmpy2
import math
```

6.1 Trial Division: Implementation

```
def trial_division(n: int) -> int:
    f = 2
    while f * f < n:
        if n % f == 0:
            return f
        f += 1
    return n
```

6.2 Wheel Factorization: Implementation

```
def wheel_factorization(n: int) -> int:
    # test whether n is divisible by 2
    if n % 2 == 0:
        return 2
    # test whether n is divisible by 3
    if n % 3 == 0:
        return 3

    # the first possible prime factor is 5
    f = 5
    # instead of increasing f by 1 each time
    # we increase f by 2 or 4 each time
    increments = [2,4]
    i = 0
    while f * f < n:
        if n % f == 0:
            return f
        f += increments[i%2]
        i += 1
    return n
```

6.3 Fermat's Factorization method: Implementation

```
def fermat(n :int) -> int:
    # if n is a perfect root
    # then both its square roots are its factors
    if(gmpy2.is_square(n)):
        k = gmpy2.isqrt(n)
        return k

    # if n is not a perfect root, find smallest k s.t. k**2 >= n
    k = gmpy2.isqrt(n) + 1

    while(not gmpy2.is_square(k**2-n)):
        k += 1

    y = gmpy2.isqrt(k**2-n)

    return k-y
```

6.4 Kraitchik-Fermat's factorization method: Implementation

```
def fermat_kraitchik(n :int) -> int:
    k = gmpy2.isqrt(n) + 1
    while True:
        t = 1
        y_2 = k*k - t*n
        while y_2 >= 0:
            y_2 = k*k - t*n
            if gmpy2.is_square(y_2):
                y = gmpy2.isqrt(y_2)
                if (k+y)%n != 0 and (k-y)%n != 0:
                    return math.gcd(k-y, n)
            t += 1
        k += 1
```

6.5 Pollard p-1 Factorization

```
def pollard(n :int) -> int:
    # test whether n is divisible by 2
    if n % 2 == 0:
        return 2

    # pick a coprime to n (for odd number, we pick n=2)
```

```
a = 2
a_record = a
k = 2
B = 100000
i = 0
# steps = 0

# iterate till a prime factor is obtained
while(True):
    # steps += 1
    a = pow(a, k, n)
    d = math.gcd((a-1), n)

    if (1 < d < n):
        # print(steps)
        return d

    if i >= B:
        if d == n:
            i = 0
            k = 2
            a_record += 1
            a = a_record

            d = math.gcd(a, n)
            if d != 1:
                return d
            continue;
    k += 1
    i += 1
```

7 Appendix B

```
from trial_division import trial_division
from fermat_factorization import fermat
from pollard import pollard
from wheel_factorization import wheel_factorization
from fermat_kraitchik_factorization import fermat_kraitchik
import random
from numpy import median, average
import time
import matplotlib.pyplot as plt
import gmpy2
```

7.1 Simulation for Trial Division versus Wheel Factorization

```
def test_runtime_t_w():
    """Use trial division and wheel factorization to factor integers
    with different number
    of digits

    Return: list of number of digits
    list of runtimes for trial division
    list of runtims for wheel factorization"""
    random.seed(6)
    runtime_trial = []
    runtime_wheel = []
    x = []
    for test_digit in range(2, 16):
        x.append(test_digit)
        test_numbers = []
        for i in range(50):
            random_num = random.randint(10**(test_digit-1),
            10**test_digit-1)
            while(random_num % 2 == 0 or gmpy2.is_prime(random_num)):
            # if the number is not odd,
            #randomly generate another number until it is odd
                random_num = random.randint(10**(test_digit-1),
                10**test_digit-1)
            test_numbers.append(random_num)

        start = time.time()
        for number in test_numbers:
            trial_division(number)
        end = time.time()
        runtime_trial.append((end-start) / 50)

        start = time.time()
        for number in test_numbers:
            wheel_factorization(number)
        end = time.time()
        runtime_wheel.append((end-start) / 50)

    return x, runtime_trial, runtime_wheel

def draw_trial_wheel():
    x, trial, wheel = test_runtime_t_w()
    plt.plot(x,trial, label="Trial Division")
    plt.plot(x,wheel, label="Wheel Factorization")
    plt.title('Runtime: Trial Division vs. Wheel Factorization')
    plt.xlabel('Nmuber of digits')
    plt.ylabel('Runtimes in second')
    plt.legend()
    plt.show()
```

7.2 Simulation for Fermat's versus Kraitchik-Fermat's Factorization

```
def test_runtime_f_k():
    """Use Fermat's and Kraitchik-Fermat's factorization to factor
    integers with different number
    of digits

    Return: list of number of digits
    list of runtimes for Fermat's factorization
    list of runtims for  factorization"""
    random.seed(5)
    runtime_f = []
    runtime_k = []
    x = []
    for test_digit in range(2, 6):
        x.append(test_digit)
        test_numbers = []
        for i in range(50):
            random_num = random.randint(10**(test_digit-1),
            10**test_digit-1)
            while(random_num % 2 == 0 or gmpy2.is_prime(random_num)):
            # if the number is not odd,
            # randomly generate another number until it is odd
                random_num = random.randint(10**(test_digit-1),
                10**test_digit-1)
            test_numbers.append(random_num)

        print(test_numbers)
        for number in test_numbers:
            start = time.time()
            fermat(number)
            end = time.time()
        runtime_f.append((end-start)/50)

        start = time.time()
        for number in test_numbers:
            fermat_kraitchik(number)
        end = time.time()
        runtime_k.append((end-start)/50)

    return x, runtime_f, runtime_k

def draw_fermat_kraitchik():
    x, trial, wheel = test_runtime_f_k()
    plt.plot(x,trial, label="Fermat's Factorization")
    plt.plot(x,wheel, label=" Kraitchik-Fermat's Factorization")
    plt.title("Runtime: Fermat's Factorization vs.
        Kraitchik-Fermat's Factorization")
    plt.xlabel('Nmuber of digits')
    plt.ylabel('Runtimes in second')
    plt.xticks(x)
    plt.legend()
    plt.show()
```

References

[1] Integer factorization, Apr 2022. URL: https://en.wikipedia.org/wiki/Integer_factorization.

[2] Connelly Barnes. Integer factorization algorithms. Dec 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.1230&amp;rep=rep1&amp;type=pdf.

[3] Wheel factorization. URL: https://en.wikipedia.org/wiki/Wheel_factorization.

[4] Kraitchik-fermat's factorization method. URL: https://mathworld.wolfram.com/FermatsFactorizationMethod.html.

[5] Anne-Sophie Charest. Pollards p-1 and lenstras factoring algorithms. Oct 2005. URL: https://www.math.mcgill.ca/darmon/courses/05-06/usra/charest.pdf.