Shenzhen University

Report of The Experiments

Course： Information Security and Blockchain

Topic： Public-key Cryptography

Class： Wenhua honor Class

StudentID： 2022110131

Name： Zuyi Liao

Date： 2024.11.5

Score：

1. Experiment Content

A. Primality Test:

Task: find the best algorithm for Primality Test and program to implement it; use the algorithm to generate a random prime number in $[2^{1023}, 2^{1024}]$.

Primality testing algorithms are designed to determine if a given number is a prime. These algorithms have extensive applications in number theory and computer science, particularly in cryptography. They are generally divided into two main categories: deterministic algorithms and probabilistic algorithms.

1. Classical Algorithms

   These algorithms are commonly used for testing the primality of smaller integers.

1.1 Brute Force Method

This method involves iterating through integers from 2 to $\sqrt{n}$ and checking if any of them divides n evenly. Its complexity is $O(\sqrt{n})$. Suitable for small numbers but inefficient for larger numbers due to its high time complexity.

1.2 Sieve of Eratosthenes

This algorithm generates all primes less than a given number n by marking non-primes in a range. Starting from 2, it marks the multiples of each prime as composite and continues with the next unmarked number. Its complexity is $O(n\ loglog\ n)$. Efficient for generating primes within a range, but less suitable for testing the primality of individual large numbers.

2. Deterministic Algorithms

   These algorithms work well for moderate-sized numbers and provide definite results on primality.

2.1 Fermat's Little Theorem Test

Relies on Fermat's Little Theorem, which states that if n is prime, for any integer a where $1 < a < n$, $a^{n-1} \equiv 1 \pmod{n}$ should hold. Its complexity is $O(k \log \hat{} 2\, n)$, where k is the number of bases tested. Effective for large numbers, but it may misidentify certain composites as prime due to Fermat pseudoprimes.

2.2 AKS Primality Test

A deterministic, polynomial-time algorithm that confirms primality through polynomial congruences. Its complexity is $O(\log \hat{} 6\, n)$. Provides a guaranteed result on primality but is usually impractical for very large numbers due to high computational demands.

3. Probabilistic Algorithms

3.1 Miller-Rabin Primality Test

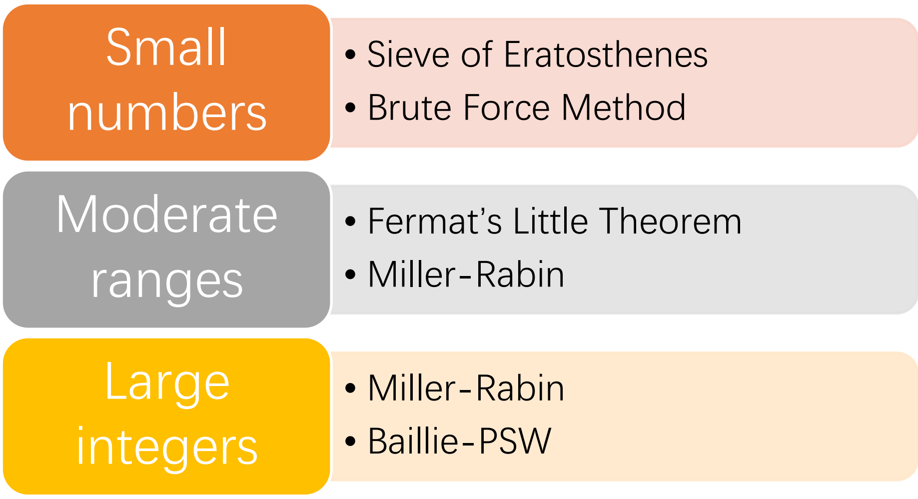
A probabilistic test that involves checking modular arithmetic conditions for random integers a. If any condition fails, n is composite; otherwise, it is likely prime. Its complexity is $O(k \log \hat{} 3\, n)$, with accuracy improving as the number of trials k increases. Reliable and efficient for large numbers with a minimal risk of error.

3.2 Solovay-Strassen Primality Test

Uses the Jacobi symbol and Euler's criterion, comparing $a^{(n-1)/2}$ with the Jacobi symbol of a. Its complexity is $O(k\log{}^\wedge 3\,n)$. Though generally less effective than Miller-Rabin, it is useful in specific mathematical cases.

3.3 Baillie-PSW Primality Test

Combines Miller-Rabin and Lucas tests for enhanced accuracy, usually performing a single Miller-Rabin test followed by a Lucas test. Its complexity is $O(\log{}^\wedge 3\,n)$. No known composites have passed this test in practice, though it lacks a theoretical guarantee of absolute accuracy.



When generating random prime numbers within the range $[2^{1023}, 2^{1024}]$, efficient and accurate **probabilistic primality testing algorithms** are typically preferred, especially the **Miller-Rabin Primality Test**. In such large ranges, deterministic algorithms like AKS become extremely inefficient, making probabilistic methods more practical.

Reason: Miller-Rabin has an efficient time complexity for large integers By selecting multiple bases (e.g., 40 trials), the probability of error can be reduced to an extremely low level, making this method highly reliable for fields like cryptography where high confidence is required

Practical Workflow

1. Randomly generate a large integer p within the range $[2^{1023}, 2^{1024}]$
2. Apply small prime filtering to remove numbers with small prime factors.
3. Perform the Miller-Rabin test on filtered candidates, repeating 40 or more times until a number passes all tests.
4. If a number passes all tests, it is considered prime and returned as output; if it fails, select a new candidate and repeat steps 1–3.

B. Computational Hard Problem:

Task: find the best algorithm for the Integer Factorization / Discrete Logarithm / Elliptic Curve Discrete Logarithm Problem (choose one of the three) and program it; test what is the maximum parameter that can be cracked with this algorithm?

I focus on the **Integer Factorization** problem, a classic problem in cryptography that aims to factor a large integer into its prime components. This problem is fundamental in cryptographic systems like RSA, where the security of the system relies on the difficulty of factoring large composite numbers.

For integer factorization, there are several algorithms with varying efficiencies:

- **Trial Division**: Simple but inefficient for large numbers.
- **Pollard's Rho Algorithm**: Effective for smaller integers, generally up to about 20–25 digits.
- **Quadratic Sieve (QS)**: A faster algorithm for moderately large numbers.
- **General Number Field Sieve (GNFS)**: Currently the fastest known algorithm for very large numbers (beyond 100 digits).

Given the options, GNFS (General Number Field Sieve) is considered the best choice for factoring large integers, especially those used in cryptographic applications.

However, GNFS is complex to implement from scratch and generally requires significant computational resources, so we will instead use **Pollard’s Rho Algorithm**, a simpler algorithm suitable for moderately sized integers (up to 20-25 digits). Pollard’s Rho is more practical for demonstration purposes and manageable to code, allowing us to test and see how it scales.

2. The experimental code and results screenshots

A. Primality Test:

Code:

```
if n <= 1:
    return False
if n <= 3:
    return True
if n % 2 == 0:
    return False

# Write n as d * 2^r + 1 with d odd
r, d = 0, n - 1
while d % 2 == 0:
    d //= 2
    r += 1

# Perform k rounds of testing
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(r - 1):
        x = pow(x, 2, n)
        if x == n - 1:
            break
    else:
        return False
return True
```

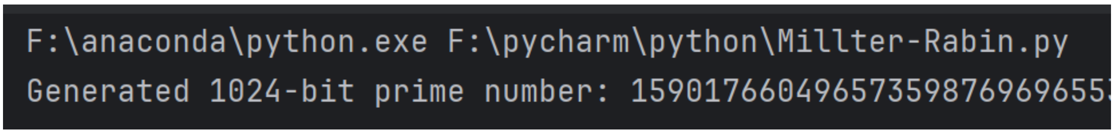
```
while True:
    # Generate a random odd number in the range [2^(bits-1), 2^bits)
    candidate = random.getrandbits(bits) | (1 << (bits - 1)) | 1

    # Perform small prime filtering
    small_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41
        , 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
    if any(candidate % p == 0 for p in small_primes):
        continue

    # Perform Miller-Rabin test
    if is_prime(candidate):
        return candidate
```

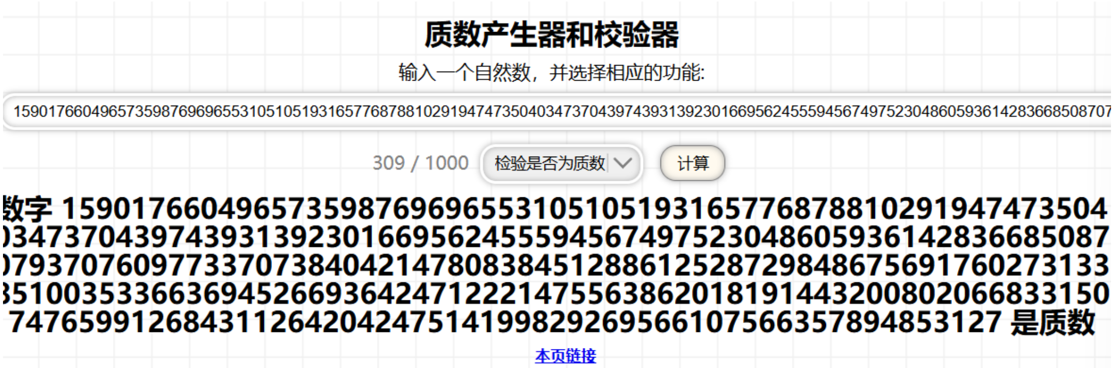
For testing large prime, I use a website (质数产生器和校验器) for verification. The follow below is the verification results.

ONE

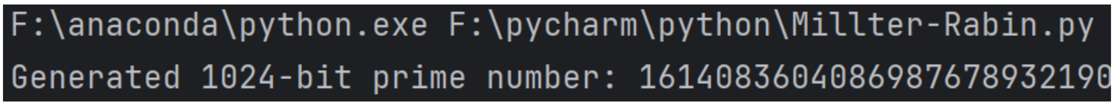


Generated 1024-bit prime number:

159017660496573598769696553105105193165776878810291947473504034737043974393139230166956245559456749752304860593614283668508707937076097733707384042147808384512886125287298486756917602731338510035336636945266936424712221475563862018191443200802066833150747659912684311264204247514199829269566107566357894853127

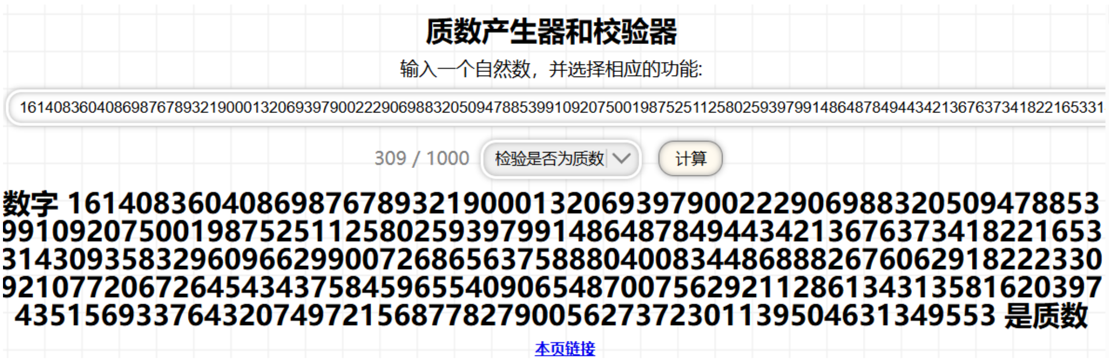


TWO

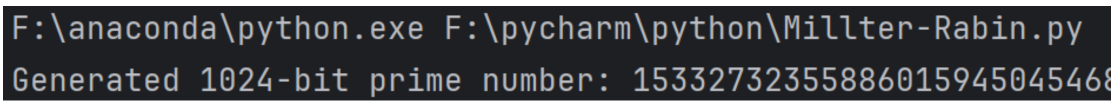


Generated 1024-bit prime number:

161408360408698767893219000132069397900222906988320509478853991092075001987525112580259397991486487849443421367637341822165331430935832960966299007268656375888040083448688826760629182223309210772067264543437584596554090654870075629211286134313581620397435156933764320749721568778279005627372301139504631349553



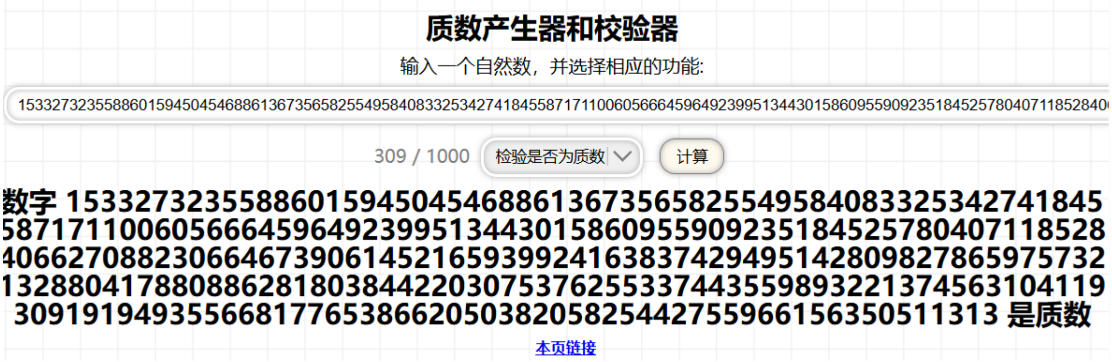
THREE



Generated 1024-bit prime number:

153327323558860159450454688613673565825549584083325342741845587171100605666459649239951344301586095590923518452578040711852840662708823066467390614521659399 24

1638374294951428098278659757321328804178808862818038442203075376255337443559893
2213745631041193091919493556681776538662050382058254427559661563505113 13



B. Computational Hard Problem:

Code:

```
def pollards_rho(n):
    """Pollard's Rho integer factorization algorithm."""
    if n % 2 == 0:
        return 2
    x = random.randint( a: 1, n - 1)
    y = x
    c = random.randint( a: 1, n - 1)
    d = 1

    while d == 1:
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        d = gcd(abs(x - y), n)
        if d == n:
            return pollards_rho(n)
    return d

def factorize(n):
    """Recursive function to factorize an integer using Pollard's Rho algorithm."""
    if n == 1:
        return []
    if is_prime(n):
        return [n]
    factor = pollards_rho(n)
    return factorize(factor) + factorize(n // factor)
```

Test for the maximum parameter:

676123456119009879801

21

```
F:\anaconda\python.exe "F:\pycharm\python\Pollard's Rho Algorithm.py"
Enter an integer to factorize: 676123456119009879801
The factorization of 676123456119009879801 is: [3, 14536147, 15504417048961]
Time taken for factorization: 0.5040 seconds
```

6761234561190098798011

22

```
F:\anaconda\python.exe "F:\pycharm\python\Pollard's Rho Algorithm.py"
Enter an integer to factorize: 6761234561190098798011
The factorization of 6761234561190098798011 is: [7, 202882723, 4760832451951]
Time taken for factorization: 5.6304 seconds
```

67612345611900987980111

23

```
Enter an integer to factorize: 67612345611900987980111
The factorization of 67612345611900987980111 is: [577, 117179108512826668943]
Time taken for factorization: 468.5266 seconds
```

876123678643200987112215

24

```
F:\anaconda\python.exe "F:\pycharm\python\Pollard's Rho Algorithm.py"
Enter an integer to factorize: 876123678643200987112215
The factorization of 876123678643200987112215 is: [3, 3, 3, 3, 5, 3, 721089447442963775401]
Time taken for factorization: 1099.3079 seconds
```

The results, as shown in the provided screenshots, indicate that Pollard's Rho successfully factorized the large integer 67612345611900987980111 into its prime components [577, 117179108512826668943]. However, this factorization took approximately **468.5266 seconds**, highlighting the limitations of Pollard's Rho for larger integers.

For the tested integer, which has around 23 digits, the factorization process took over 7 minutes, indicating that Pollard's Rho may not be practical for real-time applications when dealing with numbers of this size or larger.

Based on the time taken, Pollard's Rho is not ideal for very large integers (beyond 20-25 digits). For cryptographic purposes, factoring integers with hundreds of digits would require more advanced algorithms, such as the **General Number Field Sieve (GNFS)**, which is known to be the most efficient algorithm for large integers.