ĐẠI HỌC BÁCH KHOA HÀ NỘI
KHOA TOÁN - TIN

KỸ THUẬT LẬP TRÌNH

CHỦ ĐỀ: XÂY DỰNG CHƯƠNG TRÌNH SINH SỐ NGUYÊN TỐ LỚN VÀ MÃ HÓA RSA

Giảng viên hướng dẫn:	TS. Vũ Thành Nam
Sinh viên thực hiện:	Lê Thành Đạt
Mã sinh viên:	20216811
Mã lớp:	150328

HÀ NỘI - 2024

Mục lục

1 Cơ sở lý thuyết **3**

1.1 Modular số học . . . 3

1.2 Số nguyên tố . . . 3

1.3 Ước chung lớn nhất . . . 3

2 Phép toán với modular **4**

2.1 Phép lũy thừa modular . . . 4

2.1.1 Thuật toán Exponent by Squaring (lũy thừa bằng cách bình phương) 4

2.1.2 Cài đặt . . . 5

2.1.3 Kiểm thử . . . 5

2.2 Phép nhân modular . . . 7

2.2.1 Cài đặt . . . 7

2.2.2 Kiểm thử . . . 8

3 Thuật toán kiểm tra số nguyên tố Miller-Rabin **10**

3.1 Thuật toán Miller-Rabin . . . 10

3.2 Cài đặt . . . 11

3.3 Kiểm thử . . . 12

4 Thuật toán sinh số nguyên tố **14**

4.1 Thuật toán sinh số nguyên tố ngẫu nhiên . . . 14

4.2 Cài đặt . . . 14

4.3 Kiểm thử . . . 15

5 Mã hóa RSA **16**

5.1 Thuật toán . . . 16

5.2 Cài đặt 16
5.3 Kiểm thử 18

Kết luận **19**

Chương 1

Cơ sở lý thuyết

1.1 Modular số học

Về mặt cơ bản $a \equiv b(mod\ n)$ nếu $a = b + kn$. Modulo số học cũng giống như số học bình thường, bao gồm các phép giao hoán, kết hợp và phần phối. Mặt khác giảm mỗi giá trị trung gian trong suốt quá trình tính toán.

$$(a+b)\ mod\ n = ((a\ mod\ n) + (b\ mod\ n))\ mod\ n$$

$$(a-b)\ mod\ n = ((a\ mod\ n) - (b\ mod\ n))\ mod\ n$$

$$(a \times b)\ mod\ n = ((a\ mod\ n) \times (b\ mod\ n))\ mod\ n$$

$$(a \times (b+c))\ mod\ n = (((a \times b)\ mod\ n) + ((a \times c)\ mod\ n))\ mod\ n$$

1.2 Số nguyên tố

Số nguyên tố là các số tự nhiên lớn hơn 1, chỉ có hai ước là 1 và chính nó. Nói cách khác, đó là các số tự nhiên không thể chia hết cho bất kỳ số nào khác ngoài 1 và chính nó.

1.3 Ước chung lớn nhất

Trong toán học, ước số chung lớn nhất (ƯCLN) hay ước chung lớn nhất (ƯCLN) của hai hay nhiều số nguyên là số nguyên dương lớn nhất là ước số chung của các số đó.

Ví dụ, ước chung lớn nhất của 6 và 15 là 3.

Chương 2

Phép toán với modular

2.1 Phép lũy thừa modular

2.1.1 Thuật toán Exponent by Squaring (lũy thừa bằng cách bình phương)

Kĩ thuật lũy thừa bằng cách bình phương có thể tính lũy thừa một cách hiệu quả chỉ với độ phức tạp $O(log_2 b)$. Kĩ thuật này chỉ cần $O(log_2 b)$ lần bình phương và $O(log_2 b)$ phép nhân để ra kết quả

Ví dụ:

$a^b = (a^{b/2})^2$ nếu b chia hết cho 2

$a^b = a.(a^{[\frac{b}{2}]})^2$ nếu b không chia hết cho 2

$a^b = 1$ nếu $b = 0$

Thuật toán được mô tả như sau

Bước 1: Khởi tạo kết quả ban đầu: Đặt *result = 1*.

Bước 2: Chuẩn bị cơ số và điều chỉnh mô-đun:

Tính *base = base % modulus* để đảm bảo cơ số nằm trong phạm vi từ *0* đến *modulus - 1*.

Bước 3: Lặp qua các bit của số mũ:

Duyệt qua từng bit của *exponent* từ trái sang phải.

Nếu bit hiện tại của *exponent* là *1*, thực hiện nhân *result* với *base* và lấy modulo *modulus* để cập nhật *result*.

Sau đó, dịch bit của *exponent* sang phải *(exponent = exponent » 1)*.

Cập nhật cơ số bằng cách lấy bình phương và lấy modulo modulus *(base = (base ˆ base) % modulus)*.

Bước 4: Kết thúc lặp: Sau khi duyệt hết các bit của *exponent*, *result* sẽ chứa kết quả lũy thừa *base êxponent % modulus*.

Bước 5: Trả về kết quả: *result* là kết quả cuối cùng của phép lũy thừa *base ˆ exponent % modulus*.

2.1.2 Cài đặt

```python
def modular_exponentiation(base, exponent, modulus):
    result = 1
    base = base % modulus
    while exponent > 0:
        if (exponent % 2) == 1:
            result = (result * base) % modulus
        exponent = exponent >> 1
        base = (base * base) % modulus
    return result
```

Listing 2.1. Thuật toán lũy thừa bằng cách bình phương

2.1.3 Kiểm thử

Kiểm thử với các trường hợp số nhỏ, số rất lớn, cơ số hoặc số mũ bằng 0, với modular bằng 1

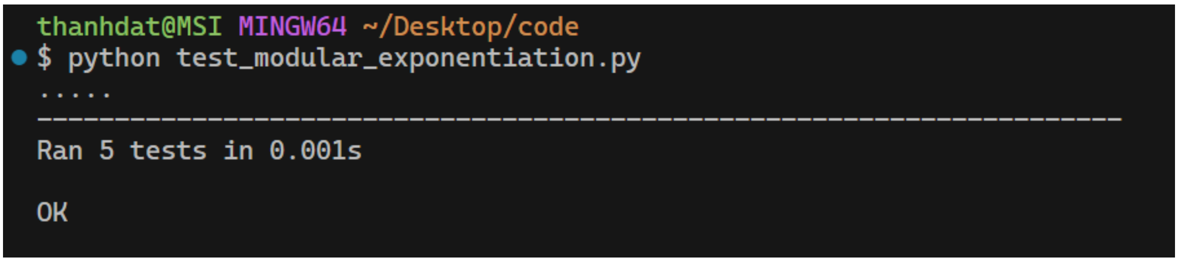
```python
import unittest
from modular_calculation import modular_exponentiation
class TestModularExponentiation(unittest.TestCase):

    def test_small_numbers(self):
        self.assertEqual(modular_exponentiation(2, 3, 5),
            3)
```

```python
        self.assertEqual(modular_exponentiation(3, 3, 7), 6)
        self.assertEqual(modular_exponentiation(4, 2, 9), 7)

    def test_large_numbers(self):
        self.assertEqual(modular_exponentiation(10123465234878998, 123456789, 10005412336548794), 8080341694206197)
        self.assertEqual(modular_exponentiation(987654321, 123456789, 1000000007), 652541198)
        self.assertEqual(modular_exponentiation(987654321987654321, 987654321, 999999937), 142857143)

    def test_zero_exponent(self):
        self.assertEqual(modular_exponentiation(5, 0, 3), 1)
        self.assertEqual(modular_exponentiation(10, 0, 7), 1)

    def test_zero_base(self):
        self.assertEqual(modular_exponentiation(0, 5, 3), 0)
        self.assertEqual(modular_exponentiation(0, 10, 7), 0)

    def test_modulus_one(self):
```

```
        self.assertEqual(modular_exponentiation(12345, 67890, 1), 0)
        self.assertEqual(modular_exponentiation(987654321, 123456789, 1), 0)

if __name__ == '__main__':
    unittest.main()
```

Listing 2.2. Kiểm thử phép lũy thừa modular

Kết quả thu được:



Hình 2.1. Kết quả kiểm thử

2.2 Phép nhân modular

Phép nhân modular có thể được thực hiện bằng công thức:

$$(a \times b)\ mod\ n = ((a\ mod\ n) \times (b\ mod\ n))\ mod\ n$$

2.2.1 Cài đặt

```
def modular_multiplication(a, b, m):
    a = a % m
    b = b % m
    return (a * b) % m
```

Listing 2.3. Phép nhân modular

2.2.2 Kiểm thử

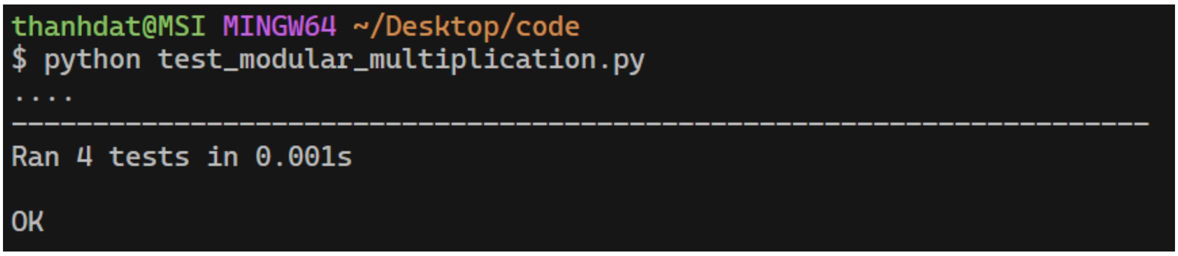
Kiểm thử với các trường hợp số nhỏ, số lớn, các trường hợp đặc biệt với 0 và 1

```python
import unittest
from modular_calculation import modular_multiplication
class TestmodularMultiplication(unittest.TestCase):

    def test_small_numbers(self):
        self.assertEqual(modular_multiplication(2, 3, 5), 1)
        self.assertEqual(modular_multiplication(10, 10, 7), 2)

    def test_large_numbers(self):
        self.assertEqual(modular_multiplication(123456789123456789, 987654321987654321, 999999937), 827637387)
        self.assertEqual(modular_multiplication(999999999999999999, 888888888888888888, 777777777), 0)

    def test_zero(self):
        self.assertEqual(modular_multiplication(0, 12345, 6789), 0)
        self.assertEqual(modular_multiplication(12345, 0, 6789), 0)
        self.assertEqual(modular_multiplication(0, 0, 6789), 0)

```

```python
    def test_mod_one(self):
        self.assertEqual(modular_multiplication(12345, 67890, 1), 0)
        self.assertEqual(modular_multiplication(0, 67890, 1), 0)
        self.assertEqual(modular_multiplication(12345, 0, 1), 0)

if __name__ == '__main__':
    unittest.main()
```

Listing 2.4. Kiểm thử phép nhân modular

Kết quả thu được:



Hình 2.2. Kết quả kiểm thử

Chương 3

Thuật toán kiểm tra số nguyên tố Miller-Rabin

3.1 Thuật toán Miller-Rabin

Thuật toán Miller-Rabin dựa trên một kỹ thuật gọi là "kiểm tra nguyên tố không chính xác". Ý tưởng cơ bản của thuật toán là kiểm tra xem một số nguyên dương n có phải là nguyên tố hay không bằng cách chọn ngẫu nhiên một số nguyên a và kiểm tra một số điều kiện liên quan đến tính chất của số nguyên n.

Dưới đây là thuật toán Miller-Rabin để kiểm tra xem một số n có phải là nguyên tố hay không:

Bước 1: Đặt số nguyên dương $n-1 = 2^r \times m$ với một số lẻ m.

Bước 2: Chọn một số nguyên a ngẫu nhiên từ khoảng $[2, n-2]$.

Bước 3: Tính giá trị $x_0 = a^m \ mod\ n$.

Bước 4: Với i từ 0 đến $r-1$, tính giá trị $x_i + 1 = (x_i^2)\ mod\ n$

Bước 5: Nếu tồn tại một số i trong khoảng từ 0 đến $r-1$ mà x_i là bằng 1 và $x_i - 1$ khác 1 và $n-1$, thì số n không phải là số nguyên tố.

Bước 6: Nếu x_r không bằng 1, số n cũng không phải là số nguyên tố.

Bước 7: Nếu không thỏa mãn các điều kiện trên, số n có khả năng lớn là số nguyên tố.

Thuật toán Miller-Rabin có độ phức tạp thời gian $O(klogn)$, trong đó k là số lần kiểm tra. Với mỗi lần kiểm tra, thuật toán sử dụng phép tính modulo và lũy thừa modulo để tính toán giá trị x_i. Khi số lượng lần kiểm tra tăng lên, độ chính xác của thuật toán cũng tăng.

Tuy thuật toán Miller-Rabin không đảm bảo xác định tính nguyên tố của một số, nhưng với số lần kiểm tra đủ lớn, nó cho kết quả chính xác đối với hầu hết các số nguyên dương.

Với 50 lần thử nếu cả 50 lần, phép thử đều "dương tính" thì xác suất sai giảm xuống chỉ còn là một số rất nhỏ không vượt quá 9×10^{-29}

3.2 Cài đặt

```
import random
def miller_rabin(n, k):
    if n == 1:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

    r, m = 0, n - 1
    while m % 2 == 0:
        r += 1
        m //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, m, 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
```

Listing 3.1. Thuật toán Miller-Rabin

3.3 Kiểm thử

Các trường hợp kiểm thử bao gồm các số nguyên tố và không phải số nguyên tố từ nhỏ đến lớn

```
import unittest
from miller_rabin import miller_rabin
class TestMillerRabin(unittest.TestCase):
    def test_prime_numbers(self):
        self.assertTrue(miller_rabin(2, 40))
        self.assertTrue(miller_rabin(3, 40))
        self.assertTrue(miller_rabin(5, 40))
        self.assertTrue(miller_rabin(11, 40))
        self.assertTrue(miller_rabin(1000000007, 40))

    def test_non_prime_numbers(self):
        self.assertFalse(miller_rabin(1, 40))
        self.assertFalse(miller_rabin(4, 40))
        self.assertFalse(miller_rabin(100, 40))
        self.assertFalse(miller_rabin(1000000008, 40))

if __name__ == '__main__':
    unittest.main()
```

Listing 3.2. Kiểm thử thuật toán Miller-Rabin

Kết quả thu được:

```
thanhdat@MSI MINGW64 ~/Desktop/code
$ python test_miller_rabin.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Hình 3.1. Kết quả kiểm thử

Chương 4

Thuật toán sinh số nguyên tố

4.1 Thuật toán sinh số nguyên tố ngẫu nhiên

Thuật toán tổng quát của việc sinh số nguyên tố ngẫu nhiên này gồm hai bước chính:

Bước 1: Sinh số lẻ ngẫu nhiên lớn hơn ngưỡng N.

Bước 2: Kiểm tra tính nguyên tố của số đó bằng thuật toán Miller-Rabin. Nếu số đó không phải là nguyên tố, quay lại bước 1.

Quá trình này được lặp lại cho đến khi tìm được một số nguyên tố. Việc sử dụng hàm Miller-Rabin với nhiều lần kiểm tra giúp đảm bảo số được sinh ra có xác suất rất cao là số nguyên tố.

4.2 Cài đặt

```
from random import randint
from miller_rabin import miller_rabin

def generate_candidate_number_greater_than(n):
    while True:
        candidate = randint(n + 1, n + 101)
        if  candidate % 2 == 1 or candidate == 2:
            return candidate
def generate_prime_number_greater_than(n):
    p = 4
    while not miller_rabin(p, 40):
```

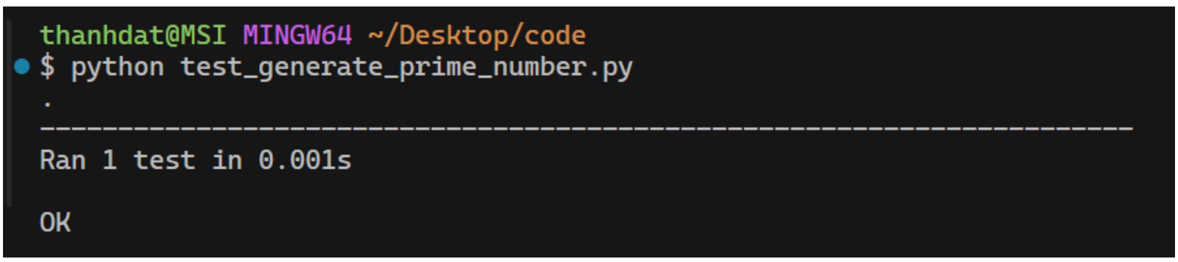
```
        p = generate_candidate_number_greater_than(n)
    return p
```

Listing 4.1. Thuật toán sinh số nguyên tố ngẫu nhiên

4.3 Kiểm thử

```
import unittest
from miller_rabin import miller_rabin
from generate_prime_number import
   generate_prime_number_greater_than

class TestRandomNumberGeneration(unittest.TestCase):
    def test_generate_prime_number_greater_than(self):
        n = 10
        for _ in range(10):
            result = generate_prime_number_greater_than(n)
            self.assertTrue(result > n)
            self.assertTrue(miller_rabin(result, 40))

if __name__ == '__main__':
    unittest.main()
```

Listing 4.2. Kiểm thử thuật toán sinh số nguyên tố



Hình 4.1. Kết quả kiểm thử thuật toán sinh số nguyên tố

Chương 5

Mã hóa RSA

5.1 Thuật toán

Thuật toán RSA là một phương pháp được sử dụng để mã hóa và giải mã tin nhắn. Nó được đặt theo tên của những người tạo ra nó, Ron Rivest, Adi Shamir, và Leonard Adleman đã phát triển nó vào năm 1977.

Tính bảo mật của thuật toán RSA dựa trên thực tế là không khả thi về mặt tính toán để phân tích một hợp số lớn thành các số nguyên tố. Nói cách khác, với một số là tích của hai số nguyên tố lớn, rất khó để tìm ra những số nguyên tố đó là gì.

Khi các khóa đã được tạo, thuật toán RSA có thể được sử dụng để mã hóa và giải mã tin nhắn. Để mã hóa thư, người gửi sử dụng khóa công khai của người nhận để mã hóa thư. Để giải mã tin nhắn, người nhận sử dụng khóa riêng của họ để giải mã nó.

Thuật toán được mô tả như sau:

Bước 1: Chọn p và q là hai số nguyên tố khác nhau.

Bước 2: Tính $n = p * q$.

Bước 3: Tính hàm phi Euler (totient) $t = (p-1)*(q-1)$.

Bước 4: Chọn e bằng *UCLN(t, e) = 1* với *1 < e < t*.

Bước 5: Tính d theo công thức *(d * e % t) = 1*.

Bước 6: Cặp khóa *(e, n)* và *(d, n)* là khóa công khai và bí mật.

Bước 7: Để mã hóa dùng: *CipherText = (Message ^ e) % n* trong đó *Message < n*.

Bước 8: Để giải mã dùng: *Message = (CipherText ^ d) % n*.

5.2 Cài đặt

```python
from math import gcd
from modular_calculation import modular_exponentiation,
    modular_multiplication
from generate_prime_number import
    generate_prime_number_greater_than
def RSA(p: int, q: int, message: int):
    n = p * q
    t = (p - 1) * (q - 1)
    # public key
    for i in range(2, t):
        if gcd(i, t) == 1:
            e = i
            break

    # private key
    j = 0
    while True:
        if modular_multiplication(j, e, t) == 1:
            d = j
            break
        j += 1

    print(f"Public key is ({e}, {n})")
    print(f"Private key is ({d}, {n})")
    # encryption
    e_mes = modular_exponentiation(message, e, n)
    print(f"Encrypted message is {e_mes}")

```

```python
    # decryption
    d_mes = modular_exponentiation(e_mes, d, n)
    print(f"Decrypted message is {d_mes}")
```

Listing 5.1. Mã hóa RSA

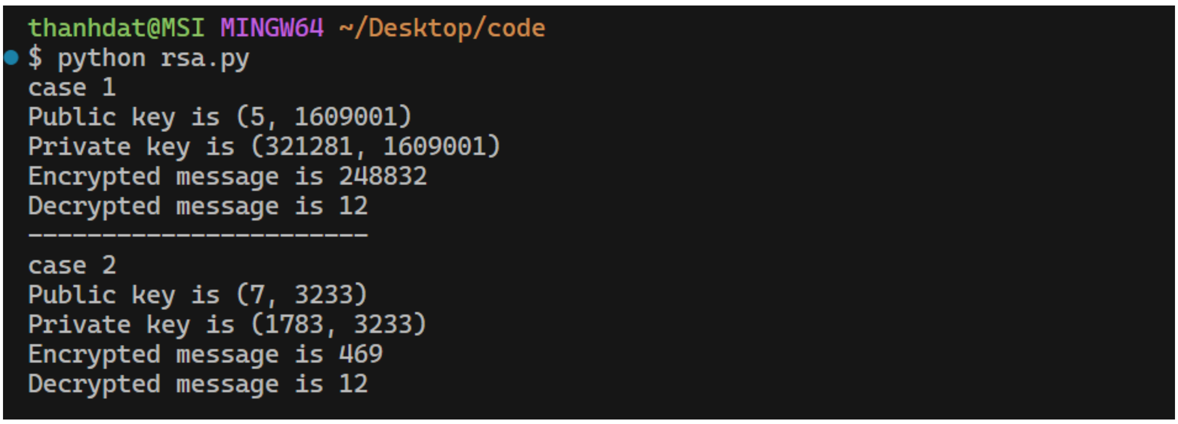
5.3 Kiểm thử

Kiểm tra với số nguyên tố lớn và nhỏ

```python
# Testcase - 1
print('case 1')
RSA(generate_prime_number_greater_than(1500),
    generate_prime_number_greater_than(1000), message=12)
print('-----------------------')
# Testcase - 2
print('case 2')
RSA(generate_prime_number_greater_than(10),
    generate_prime_number_greater_than(14), message=12)
```

Listing 5.2. Kiểm thử RSA

Kết quả thu được



Hình 5.1. Kết quả kiểm thử

Kết luận

Trong học phần Kỹ thuật lập trình do thầy Vũ Thành Nam giảng dạy trong kỳ này, em đã học được rất nhiều kiến thức liên quan đến các nguyên tắc cơ bản trong lập trình và các nền tảng cơ bản về các phương pháp lập trình, v.v... Đây là thành quả cuối khóa của em thông qua bài tập lớn cuối kỳ. Em hy vọng rằng những gì đã học được có thể được thể hiện đầy đủ trong báo cáo này.

Em xin gửi lời cảm ơn chân thành đến thầy Vũ Thành Nam, người đã giúp đỡ em và các nhóm khác trong quá trình thực hiện bài tập lớn này.