

**0xSnowyField** ([Github](https://github.com/snowyfield1906) · [Blog](https://snowyfield.vercel.app/) )

# General information

|  |  |  |  |
| --- | --- | --- | --- |
| MSSV | Họ và Tên | Môn học | Giảng viên |
| 21120566 | Nguyễn Hữu Thuận | Nhập môn Mã hoá - Mật mã | Đặng Trần Minh Hậu |

1. **Greatest fulfillments**

Điều tâm đắc nhất phải kể đến là được học về môn học, được tạo điều kiện và được trau dồi thêm đam mê của bản thân về lĩnh vực này.

Trong quá trình thực hiện đồ án này, em đã triển khai một thư viện Big Unsigned Integer (số Nguyên Dương Lớn) trong C++ để phục vụ quá trình nhập xuất các Hexadecimal (số Thập lục phân) và các phép tính trên đó.

Thư viện này một phần được xây dựng và tối ưu hoá lại từ một số thư viện khác trên các ngôn ngữ khác nhau, phần còn lại được em triển khai trong thời gian làm đồ án.

# Diﬀiculties

## Unit-level bugs in Big Unsigned Integer

Trong quá trình thực hiện đồ án này, em đã gặp phải rất nhiều khó khăn, đặc biệt là về vấn đề xử lí các hexadecimal và các phép tính cơ bản trên đó. Vì không được test kĩ càng trước khi triển khai các phép tính nâng cao hơn nên đã mất rất nhiều thời gian để debug các lỗi mà không biết vấn đề nằm ở đâu, những khó khăn này kéo dài xuyên suốt quá trình thực hiện đồ án.

## 60-second time limit handling

Một khó khăn khác là yêu cầu tự ngắt chương trình khi thời gian thực hiện vượt quá 60s. Em đã thử nhiều cách nhưng không thể triển khai nó được, dẫn đến việc phải bỏ qua yêu cầu này.

## Vague requirements

Khó khăn còn lại là về vấn đề không đến lớp, em học chủ yếu từ Google và kiến thức cũ, dẫn đến việc cách triển khai của em có thể khác với cách triển khai và yêu cầu của giảng viên. Trong suốt các đồ án em chỉ bám vào yêu cầu của đề bài và làm đúng theo đó, dẫn đến nhiều hệ quả mà chính em cũng đã nhận thức được từ trước nhưng không thể tránh khỏi.

# Improvement ideas

## Time optimization for Big Unsigned Integer

Thư viện Big Unsigned Integer tuy hoạt động tốt nhưng lại không đạt hiểu quả cao về tốc độ và hiệu suất. Nó đã vượt quá thời gian 60s đối với các test mà số nguyên có độ dài 1024 bit.

Vì vậy, em đề xuất một số ý tưởng cải tiến như sau (xếp theo mức độ ảnh hưởng đến hiệu suất):

2

* Tạo riêng các method tối ưu cho việc tính toán với số (như phép nhân, chia hoặc lũy thừa) thay vì dùng chung với các method khác. Điều này có thể làm giảm một phần thời gian thực hiện các thuật toán.
* Giảm thiểu các phép tính hoặc phép gán thừa trong các thuật toán, hạn chế sự lặp lại trong code để tối ưu thời gian thực hiện.
* Thử và thay đổi một số thuật toán khác để xem xét hiệu quả về thời gian của chúng.
* Sử dụng hexadecimal cho việc lưu trữ và tính toán thay vì dùng thuật toán thông thường để chuyển đổi nó sang Decimal (hệ Thập phân) rồi chuyển ngược lại.

# Method/Algorithm implementations

Vì 95% thuật toán và thời gian thực hiện của đồ án này nằm ở thư viện Big Unsigned Integer, trong khi đó các phần khác khá đơn giản và ngắn. Do đó báo cáo này sẽ tập trung hầu hết vào phần thư viện Big Unsigned Integer mà em đã triển khai.

## Big Unsigned Integer library

Chia một số nguyên dương thành các blocks. Mỗi block lưu trữ 6 chữ số thập phân. Do đó, giá trị của một block nằm trong khoảng [106; 107 − 1] (trừ block cuối cùng).

vector<int> data

Trên máy 64-bit, kích thước của một int là 32(bits) = 4(bytes). Do đó giới hạn kích thước của vector trên là 264/4 − 1 = 262 − 1 (phần tử).

Từ đây, ta có thể suy ra được (trong trường hợp lý tưởng):

* Kích thước lưu trữ tối đa ∈ [𝑙𝑛(106) × (262 − 1); 𝑙𝑛(107 − 1) × (262 − 1)] ≈ 259 (bytes)
* Giá trị lớn nhất ∈ [262 × 106; 262 × (107 − 1)] ≈ 282

Lí do tại sao lại chia như vậy là vì thuật toán **Fast Fourier Transform** bên dưới.

* + 1. **Fast Fourier Transform:** 𝐴 × 𝐵

Phép nhân 2 số nguyên là một trong những bài toán lớn trong lĩnh vực Mật Mã Học. Nó ảnh hưởng trực tiếp đến hiệu suất của các thuật toán mã hoá, giải mã và các thuật toán khác.

𝑂(𝑛 (𝑛) ( (𝑛))

Nhân 2 số nguyên với n chữ số có thể được thực hiện với độ phức tạp log log log bằng cách sử dụng Fast Fourier Transform thay vì 2 bằng cách thông thường. Kĩ thuật này sử dụng phương pháp **Polynomial Multiplication** (Nhân Đa thức) và **Recursive Divide and Conquer** (Chia Để Trị).

𝑂(𝑛 )

Một trong những triển khai hiệu quả của Fast Fourier Transform trong việc tìm tích của 2 số nguyên lớn là thuật toán Schönhage–Strassen.

* + - 1. **Polynomial Multiplication** Ta sẽ coi một số nguyên dương 𝑁 là một đa thức 𝑃 (𝑥) thoả mãn 𝑃 (𝑥) = ∑𝑛−1 𝑎𝑖𝑥𝑖 với 𝑎𝑖 là các chữ số trong 𝑁. Ta gọi 𝑥 là Base (Cơ số) của 𝑁, 𝑎 là **Coeﬀicient**

**Vector** (Vector𝑖=H0ệ số) của .

𝑁

Ví dụ, số nguyên 123456789 là một đa thức 𝑃 (𝑥) = 9𝑥0 + 8𝑥1 + 7𝑥2 + 6𝑥3 + 5𝑥4 + 4𝑥5 + 3𝑥6 + 2𝑥7 + 1𝑥8

với base 𝐵 = 10 và coeﬀicient vector 𝑎 = [9, 8, 7, 6, 5, 4, 3, 2, 1]. Ngoài ra:

* + - * + Với base 𝐵 = 100, ta sẽ có 𝑎 = [89, 67, 45, 23, 1]
        + Với base 𝐵 = 1000, ta sẽ có 𝑎 = [789, 456, 123]
        + Với base 𝐵 = 16, ta sẽ có 𝑎 = [5, 1, 13, 12, 11, 5, 7] (tương đương 0x51d0b57, là biểu diễn hexadec- imal của 123456789 dưới dạng little-endian)

Đó là lí do tại sao ta lại chia một số nguyên thành các blocks như đã đề cập.

* + - 1. **Recursive Divide and Conquer** Chúng ta sẽ chia đa thức 𝑃 (𝑥) thành 2 đa thức con

𝑃𝑒𝑣𝑒𝑛(𝑥) và 𝑃𝑜𝑑𝑑(𝑥). Trong đó:

* + - * + 𝑃𝑒𝑣𝑒𝑛(𝑥) là đa thức con của 𝑃 (𝑥) với các hệ số ở vị trí chẵn.
        + 𝑃𝑜𝑑𝑑(𝑥) là đa thức con của 𝑃 (𝑥) với các hệ số ở vị trí lẻ.

Ví dụ đa thức 𝑃 (𝑥) = 𝑎0 + 𝑎1𝑥 + 𝑎2𝑥2 + ⋯ + 𝑎𝑛−1𝑥𝑛−1 sẽ được chia thành:

𝑃𝑒𝑣𝑒𝑛(𝑥) = 𝑎0 + 𝑎2𝑥 + 𝑎4𝑥2 + ⋯ + 𝑎𝑛−2𝑥𝑛/2−1

𝑃𝑜𝑑𝑑(𝑥) = 𝑎1 + 𝑎3𝑥 + 𝑎5𝑥2 + ⋯ + 𝑎𝑛−1𝑥𝑛/2−1

Khi đó, ta có thể viết lại đa thức 𝑃 (𝑥) thành:

𝑃 (𝑥) = 𝑃𝑒𝑣𝑒𝑛(𝑥2) + 𝑥𝑃𝑜𝑑𝑑(𝑥2)

Cho số phức 𝑤 là **Primitive n-th Root of Unity** (Căn Đơn vị Nguyên thủy bậc n). Ta định nghĩa **n-th root of unity** (căn đơn vị bậc n) là các số phức thoả phương trình 𝑤𝑛 = 1. Trong đó tính **primitive** (nguyên thủy) cho biết các số phức 𝑤 được coi là nghiệm khi và chỉ khi nó không phải là k-th root of unity với bất kì 𝑘 < 𝑛.

Ta có 2 tính chất quan trọng sau [1](#_bookmark0):

* + - * + **Reflection** (Phản xạ): Nếu 𝑤 là primitive n-th root of unity, trong đó 𝑛 ≥ 2 và 𝑛 chẵn, thì

𝑤𝑘+𝑛/2 = −𝑤𝑘 với mọi 𝑘 = 0, 1, … , 𝑛/2 − 1.

* + - * + **Reduction** (Phân rã): Nếu là primitive (2n)-th root of unity, thì 2 là primitive n-th root of unity.

𝑤 𝑤

Từ 2 tính chất trên, đặt 𝑥 = 𝑤𝑘, ta có thể viết lại đa thức 𝑃 (𝑥) thành:

𝑃 (𝑤𝑘) = 𝑃𝑒𝑣𝑒𝑛(𝑤2𝑘) + 𝑤𝑘𝑃𝑜𝑑𝑑(𝑤2𝑘)

⟹ 𝑃 (𝑤𝑘+𝑛/2) = 𝑃𝑒𝑣𝑒𝑛(𝑤2𝑘) − 𝑤𝑘+𝑛/2𝑃𝑜𝑑𝑑(𝑤2𝑘)

Từ đây, ta cũng viết lại đa thức 𝑦 = 𝑃 (𝑥) thành:

𝑦𝑖 = ∑𝑛−1 𝑎𝑗 × 𝑤𝑖𝑗 với 𝑖 = 0, 1, … , 𝑛 − 1

𝑗=0

* + - 1. **Schönhage–Strassen** Cho 2 số nguyên viết dưới dạng đa thức là 𝐴(𝑥) và 𝐵(𝑥), khi đó [2](#_bookmark1):

(𝐴 × 𝐵)(𝑥) = 𝐴(𝑥) × 𝐵(𝑥)

⟹ FFT(𝐴 × 𝐵) = FFT(𝐴) × FFT(𝐵)

⟹ 𝐴 × 𝐵 = FFT−1(FFT(𝐴) × FFT(𝐵))

Ta sẽ gọi FFT−1 là **Inverse Fast Fourier Transform**, có thuật toán tương tự như FFT, tuy nhiên FFT−1 sẽ có tác dụng chuyển đổi từ một vector các giá trị 𝑦 về một coeﬀicient vector 𝑎.

1Các chứng minh sẽ không được đề cập để giản lược nội dung của báo cáo

2Các chứng minh sẽ không được đề cập để giản lược nội dung của báo cáo

Hay nói cách khác FFT−1 giống như một phép **interpolation** (nội suy) tìm một đa thức thoả mãn các giá trị cho trước, bằng một vài phép biến đổi ma trận, ta sẽ có được công thức bên dưới [3](#_bookmark2):

𝑎𝑖 = 1 ∑𝑛−1 𝑦𝑗 × 𝑤−𝑖𝑗 với 𝑖 = 0, 1, … , 𝑛 − 1

𝑛

𝑗=0

Tất nhiên độ dài vector giá trị 𝑦 và độ dài coeﬀicient vector 𝑎 là bằng nhau và bằng 𝑛. Vì với 𝑛 phần tử là đủ để xác định một đa thức bậc 𝑛 − 1. Đây là tính chất cơ bản trong phép interpolation.

𝐴 × 𝐵

Sau khi có được coeﬀicient vector , ta sẽ thực hiện một số phép toán như **normalize** (chuẩn hoá) hay **carry and reduce** (nhớ và rút gọn) để có được kết quả cuối cùng.

3Các chứng minh sẽ không được đề cập để giản lược nội dung của báo cáo

### Fast Fourier Transform pseudocode

**Algorithm:** Fast Fourier Transform (FFT)

### Input:

An n-length coeﬀicient vector 𝑎 → [𝑎0, 𝑎1, … , 𝑎𝑛−1]

If FFT is used for inverse transform, set 𝑖𝑛𝑣𝑒𝜏𝑡 → 𝑡𝜏𝝊𝑒

**Procedure:** FFT(𝑎, invert)

**if** 𝑛 → 1 **then**

**return** 𝑦 → 𝑎

### end if

*divide step*

𝑎even → [𝑎0, 𝑎2, 𝑎4, … , 𝑎𝑛−2]

𝑎odd → [𝑎1, 𝑎3, 𝑎5, … , 𝑎𝑛−1]

*recursive calls with* 𝑤2 *as* (𝑛/2)*-th root of unity*

FFT(𝑎even, invert) FFT(𝑎odd, invert)

*combine step, using* 𝑥 → 𝑤𝑖

**if** invert **then**

angle → −2 × 𝜋/𝑛

### else

angle → 2 × 𝜋/𝑛

𝑤 → 1, 𝑤𝑛 → cos(angle) + 𝑖 × sin(angle)

**for** 𝑖 → 0, 1, … , 𝑛/2 − 1 **do**

𝑦[𝑖] → 𝑎even[𝑖] + 𝑤 × 𝑎odd[𝑖]

𝑦[𝑖 + 𝑛/2] → 𝑎even[𝑖] − 𝑤 × 𝑎odd[𝑖]

**if** invert **then**

𝑦[𝑖] → 𝑦[𝑖]/2

𝑦[𝑖 + 𝑛/2] → 𝑦[𝑖 + 𝑛/2]/2

### end if

𝑤 → 𝑤 × 𝑤𝑛

### end for End Procedure

Sử dụng **Master Theorem** (Định lý Master), ta nhận thấy thuật toán chia ra làm 2 phần, mỗi phần có kích thước là 𝑛/2 còn lại có độ phức tạp là 𝑂(𝑛), do đó độ phức tạp thời gian của thuật toán là:

𝑇 (𝑛) = 2 × 𝑇 ( 𝑛) + 𝑂(𝑛) = 𝑂(𝑛 log(𝑛))

2

### Schönhage–Strassen pseudocode

**Algorithm:** Schönhage–Strassen algorithm for big integers multiplication

### Input:

A left-hand side coeﬀicient vector 𝑎 → [𝑎0, 𝑎1, … , 𝑎𝑛−1]

A right-hand side coeﬀicient vector 𝑏 → [𝑏0, 𝑏1, … , 𝑏𝑛−1]

### Output:

A coeﬀicient vector 𝑐 → [𝑐0, 𝑐1, … , 𝑐2𝑛−1] where 𝑐 = 𝑎 × 𝑏

**Function:** Schönhage–Strassen(𝑎, 𝑏)

𝑛 → 1

*split and zero-pad step*

**while** 𝑛 < max(𝑎.size(), 𝑏.size()) **do**

𝑛 → 𝑛 × 2

### end while

𝑎.resize(𝑛), 𝑏.resize(𝑛)

*apply FFTs*

FFT(𝑎), FFT(𝑏)

𝑐 → 𝑎 × 𝑏

FFT−1(𝑐)

*perform carry and reduce*

recarry(𝑐)

### End Function

Với mỗi FFT ta có độ phức tạp là 𝑂(𝑛 log(𝑛)), do đó độ phức tạp thời gian của thuật toán là:

𝑇 (𝑛) = 𝑛 × 𝑇 (2𝑛 + 1 log(𝑛)) + 𝑂(𝑛 log(𝑛)) = 𝑂(𝑛 log(𝑛) log(log(𝑛)))

2

* + 1. **Right-to-left Binary algorithm:** 𝐴**exp** mod 𝑛

**Modular Exponentiation** (Lũy thừa theo Modulo) cũng là một trong những phép toán quan trọng trong các bài toán **Public Key Cryptography** (Mã hoá Khoá Công khai) như RSA, ElGamal, Diﬀie-Hellman,…

Phép modular exponentiation được biểu diễn: 𝐴exp mod 𝑛.

Trong trường hợp 𝐴 là một số nguyên dương lớn, việc vừa tính 𝐴exp vừa lấy modulo 𝑛 sẽ rất tốn thời gian với độ phức tạp là 𝑂(exp). Thay vào đó, chúng ta có một số thuật toán để thực hiện phép tính này với độ phức tạp nhỏ hơn nhiều là 𝑂(log(exp)).

* + - 1. **Right-to-left Binary** Ý tưởng của thuật toán này là chuyển đổi số mũ exp sang hệ nhị phân:

exp = ∑𝑘−1 𝑎𝑖 × 2𝑖 với 𝑎𝑖 ∈ {0, 1}

2

𝑖=0

Trong đó:

* + - * + 𝑎𝑖 ∈ {0, 1} là bit thứ 𝑖 của exp2.
        + 𝑘 là số bit của exp2.
        + exp2 = 𝑎𝑘−1𝑎𝑘−2 … 𝑎1𝑎0 là biểu diễn nhị phân của exp. Khi đó, giá trị của 𝐴exp có thể được viết lại thành:

𝐴exp = 𝐴(∑ℎ−1 𝑎𝑖×2𝑖)

𝑖=0

= 𝑘∏−1 𝐴𝑎𝑖×2𝑖

𝑖=0

Bằng cách này, thay vì lặp exp lần, ta chỉ cần lặp 𝑘 lần, nhanh hơn rất nhiều.

### Right-to-left Binary pseudocode

**Algorithm:** Right-to-left Binary

### Input:

A base 𝐴

An exponent exp A modulo 𝑛

**Function:** Right-to-left Binary(𝐴, exp, 𝑛)

**if** 𝑛 = 1 **then**

### return 0

**end if**

result → 1

𝐴 → 𝐴 (mod 𝑛)

**while** exp > 0 **do**

**if** exp = 1 (mod 2) **then**

result → result × 𝐴 (mod 𝑛)

𝐴 → 𝐴 × 𝐴 (mod 𝑛)

exp → exp >> 1

**end while return** result

### End Function

Ta nhận thấy trong vòng lặp while, số lần lặp tuân theo điều kiện exp > 0 và mỗi lần lặp, exp dịch bit sang phải 1 (tương đương với việc chia cho 2). Hay nói cách khác, exp liên tục phân rã với hệ số 2, do đó số lần lặp của vòng lặp while tuân theo hàm log là 𝑂(log(exp)).

### Others [4](#_bookmark3)

Ngoài ra, còn có một số thuật toán khác đơn giản hơn như:

* **Bisection method** cho √𝐴
* **Extended Euclidean algorithm** cho 𝐴−1 (mod 𝑛)
* **Euclidean algorithm** cho gcd(𝐴, 𝐵)
* …

4Chi tiết các thuật toán sẽ không được giới thiệu để giản lược nội dung của báo cáo

## Primality test

Kiểm tra tính số nguyên tố là một lĩnh vực lớn trong toán học, tính đến nay đã có vô số các phương pháp để kiểm tra tính số nguyên tố của một số nguyên dương . Trong đó có một phương pháp kiểm (thử) tra số nguyên tố bằng xác suất, đánh đổi độ chính xác với độ phức tạp. So với các số nguyên nhỏ, các số nguyên tương đối lớn không có sự lựa chọn giữa các phương pháp nào khác ngoài phương pháp này.

𝐴

### Fermat’s little theorem

Theo **Fermat’s little theorem** (định lý Fermat nhỏ), với mọi số nguyên tố 𝑝 và mọi số nguyên 𝑎

không chia hết cho 𝑛, ta có:

𝑎𝑛−1 ≡ 1 (mod 𝑛)

⟹ 𝑎𝑛−1 − 1 ≡ 0 (mod 𝑛)

Giả sử 𝑛 là một số nguyên tố, với 𝑛 ≥ 3. Khi đó, ta có thể biết rằng 𝑛 − 1 là một số chẵn, do đó ta có thể viết lại 𝑛 − 1 thành:

𝑛 − 1 = 2𝑠 × 𝑑

Trong đó:

* 𝑠 là một số nguyên dương bất kì.
* 𝑑 là một số nguyên dương lẻ.

Đặt 𝑎 là một số nguyên dương bất kì, 𝑎 ∈ [1, 𝑛 − 1]. Ta có thể viết lại thành:

𝑎𝑛−1 − 1 ≡ 0 (mod 𝑛)

⟹ 𝑎2𝑠×𝑑 − 1 ≡ 0 (mod 𝑛)

⟹ (𝑎2𝑠−1×𝑑)2 − 1 ≡ 0 (mod 𝑛)

⟹ (𝑎2𝑠−1×𝑑 − 1) (𝑎2𝑠−1×𝑑 + 1) ≡ 0 (mod 𝑛)

⟹ ((𝑎2𝑠−2×𝑑)2 − 1) (𝑎2𝑠−1×𝑑 + 1) ≡ 0 (mod 𝑛)

⟹ (𝑎2𝑠−2×𝑑 − 1) (𝑎2𝑠−2×𝑑 + 1) (𝑎2𝑠−1×𝑑 + 1) ≡ 0 (mod 𝑛)

…

⟹ (𝑎𝑑 − 1) (𝑎𝑑 + 1) (𝑎2𝑑 + 1) … (𝑎2𝑠−1×𝑑 + 1) ≡ 0 (mod 𝑛)

Điều này có nghĩa là khi 𝑛 là một số nguyên tố, một trong các nhân tử phía trên sẽ phải chia hết cho

𝑛. Do đó:

𝑎𝑑 ≡ 1 (mod 𝑛) ∨ 𝑎2𝑖×𝑑 ≡ 1 (mod 𝑛) với 𝑖 = 0, 1, … , 𝑠 − 1

Trong trường hợp này, ta nói rằng 𝑎 là một **Miller-Rabin non-witness** (không phải nhân chứng Miller-Rabin) của 𝑛 (hay 𝑛 là một **strong pseudoprime** (giả nguyên tố mạnh) với cơ số 𝑎).

### Miller-Rabin algorithm

Tương tự với ví dụ trên, giả sử 𝑛 là một số nguyên tố, với 𝑛 ≥ 3. Chọn một số nguyên dương 𝑎 bất kì, 𝑎 ∈ [1, 𝑛 − 1]. Ta có thể viết lại thành:

Ta nói rằng 𝑎 là một **Miller-Rabin witness** (nhân chứng Miller-Rabin) của 𝑛 nếu:

𝑎𝑑 𝐺 1 (mod 𝑛) ∧ 𝑎2𝑖×𝑑 𝐺 −1 (mod 𝑛) với 𝑖 = 0, 1, … , 𝑠 − 1

Cụm từ “witness” (nhân chứng) có nghĩa là 𝑎 là một nhân chứng để chứng minh rằng 𝑛 là một hợp số. Do đó nếu 𝑛 không có Miller-Rabin witness nào, ta có thể kết luận rằng 𝑛 là một số nguyên tố (trừ trường hợp 𝑛 = 2).

Thuật toán Miller-Rabin sẽ chọn ngẫu nhiên một số 𝑎 trong khoảng [1, 𝑛 − 1] và kiểm tra biểu thức trên với mọi 𝑖 = 0, 1, … , 𝑠 − 1. Nếu tất cả các biểu thức đều thoả mãn, ta có thể kết luận rằng 𝑛 là một số nguyên tố với một xác suất sai là 41 .

Do đó, chỉ cần lặp 𝑘 lần thuật toán Miller-Rabin, xác suất sai sẽ giảm xuống chỉ còn 4−𝑘.

### Miller-Rabin pseudocode

**Algorithm:** Miller-Rabin

### Input:

A tested number 𝑛

A number of iterations 𝑘

**Function:** Miller-Rabin(𝑛, 𝑘)

**if** 𝑛 = 2 **then**

**return** true

### end if

**if** 𝑛 = 1 **or** 𝑛 is even **then return** false

### end if

𝑛 − 1 → 2𝑠 × 𝑑

**for** 𝑖 → 0, 1, … , 𝑘 − 1 **do**

𝑎 → random(1, 𝑛 − 1)

𝑥 → 𝑎𝑑 (mod 𝑛)

**if** 𝑥 = 1 **then**

### continue end if

**for** 𝑗 → 0, 1, … , 𝑠 − 1 **do**

**if** 𝑥 = 𝑛 − 1 **then continue**

### end if

𝑥 → 𝑥2 (mod 𝑛)

**end for return** false

**end for return** true

### End Function

Bằng một số phương pháp xác suất và thống kê, ta chứng minh được độ phức tạp thời gian của thuật toán trên là 𝑂(𝑘 log3(𝑛)).

## RSA - Key Generation

Gọi 𝑒 là **public exponent** (số mũ công khai), 𝑑 là **secret exponent** (số mũ bí mật).

Cho trước 2 số nguyên tố 𝑝, 𝑞. Khi đó với một số 𝑒 là **public exponent** (số mũ công khai), ta có thể tìm được một số 𝑑 là **secret exponent** (số mũ bí mật) bằng cách:

𝑑 ≡ 𝑒−1 (mod 𝜑)

⟹ 𝑑 ≡ 𝑒−1 (mod (𝑝 − 1)(𝑞 − 1))

## RSA - Decryption and Encryption

Cho trước một public key (𝑁, 𝑒), ta có thể mã hoá một số nguyên 𝑚 là tin nhắn cần gửi đến địa chỉ

(𝑁, 𝑒) bằng cách:

𝑐 ≡ 𝑚𝑒 (mod 𝑁)

Tương tự, ta có thể dùng secret key (𝑁, 𝑑) để giải mã bản mã 𝑐 bằng cách:

𝑚 ≡ 𝑐𝑑 (mod 𝑁)

Chứng minh rất đơn giản, theo **Fermat’s little theorem** (định lý Fermat nhỏ), ta có:

𝑒𝑑 ≡ 1 (mod 𝑝 − 1)

𝑒𝑑 ≡ 1 (mod 𝑞 − 1)

Vì và là 2 số nguyên tố cùng nhau, nên theo **Chinese remainder theorem** (định lý số dư Trung Hoa), ta có:

𝑝 𝑞

𝑒𝑑 ≡ 1 (mod (𝑝 − 1)(𝑞 − 1))

⟹ 𝑚 ≡ 𝑚𝑒𝑑 (mod 𝑁)

⟹ 𝑚 ≡ (𝑚𝑒)𝑑 (mod 𝑁)

⟹ 𝑚 ≡ 𝑐𝑑 (mod 𝑁)

# Disclaimer

Toàn bộ nội dung của báo cáo này đều được research trong thời gian ngắn, do đó không thể tránh khỏi sai sót.

Tuy nhiên tất cả nội dung được ghi trên báo cáo đã được bản thân em chứng minh, hiểu rõ và nắm chắc. Do đó bất kì sai sót nào trong báo cáo này đều là do sai sót của chính bản thân em, không phải do copy hay cắt ghép từ các nguồn lại với nhau.

Với những sai sót nghiêm trọng, xin vui lòng liên hệ với em qua email: [nguyenhuuthuan25112003@](mailto:nguyenhuuthuan25112003@gmail.com) [gmail.com](mailto:nguyenhuuthuan25112003@gmail.com). Em xin cảm ơn.

# Final words

Em là Nguyễn Hữu Thuận, 3 năm học tại HCMUS, 2 năm tiếp xúc với Blockchain và 1 năm kinh nghiệm làm việc Blockchain Developer. Tuy vậy, đến gần đây em mới hứng thú với lĩnh vực Mã hoá - Mật mã, đặc biệt là các ứng dụng của nó và Zero Knowledge.

Vì sự chủ quan về kiến thức của bản thân, không để ý đến % điểm thành phần, bận rộn trong cuộc sống và một số tai nạn ngoài ý muốn. Em đã bỏ lỡ các bài tập trên lớp và về nhà, thứ chiếm đến điểm tổng kết môn học. Điều này khiến em cảm thấy rất hối hận và tiếc nuối vì

60% × 70% = 42%

nguy cơ phải học lại. Hơn nữa, đây lại là môn học và em yêu thích nhất.

Hi vọng bài báo cáo này có thể giúp em có thể đạt được điểm số mong muốn để qua môn, em xin cảm ơn các thầy.