TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG HCM

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO NHÓM
NHẬP MÔN MÃ HÓA - MẬT MÃ
Lab 01
SỐ NGUYÊN TỐ & TRAO ĐỔI KHÓA DIFFIE-HELLMAN

Lớp: Nhập môn mã hóa - mật mã 22_22

GV LT: Đặng Trần Minh Hậu

Giảng viên TH: Ngô Đình Hy

MSSV	Họ và tên
21120596	Trần Đoàn Thanh Vinh
22120270	Bùi Hồng Phúc
22120276	Nguyễn Lê Anh Phúc
22120293	Võ Hoàng Anh Quân

CHƯƠNG TRÌNH CHÍNH QUY - KHÓA 22

MỤC LỤC

1. Bảng phân công công việc và tự đánh giá

Bảng phân công công việc:

Thành viên	Công việc	Mức độ hoàn thành
Trần Đoàn Thanh Vinh	- Cài đặt hàm lũy thừa mô-đun, trao đổi khóa Diffie-Hellman - Viết báo cáo phần công việc thực hiện	100%
Bùi Hồng Phúc	- Cài đặt hàm sinh số nguyên tố ngẫu nhiên, hàm sinh khóa riêng ngẫu nhiên - Viết báo cáo phần công việc thực hiện	100%
Nguyễn Lê Anh Phúc	- Cài đặt hàm sinh số nguyên tố ngẫu nhiên, hàm sinh khóa riêng ngẫu nhiên - Viết báo cáo phần công việc thực hiện	100%
Võ Hoàng Anh Quân	- Cài đặt kiểu dữ liệu chứa số nguyên tố lớn, hàm lũy thừa mô-đun, hàm sinh số ngẫu nhiên - Viết báo cáo phần công việc thực hiện, tổng hợp báo cáo	100%

Bảng chức năng, trọng số và tự đánh giá:

Chức năng	Trọng số	Tự đánh giá
Hàm lũy thừa mô-đun	25%	25%
Sinh số nguyên tố	25%	25%
Sinh khóa riêng	25%	25%
Trao đổi khóa Diffie-Hellman	25%	25%
	Tổng điểm tự đánh giá	100%

2. Hệ thống xử lý số lớn

Trong bài tập này, nhóm đã cài đặt một kiểu dữ liệu lớn riêng để sử dụng là **BigUInt512**

a. Cấu trúc dữ liệu

```
class BigUInt512 {
private:
 static const int SIZE = 512 / 64;
public:
 std::array<uint64_t, SIZE> data;

 ...
};
```

- **SIZE** = 512 / 64 = 8: Mảng data phía dưới sẽ có kích thước 8 phần tử uint64_t, mỗi phần tử có 64 bits. Mặc định của kiểu dữ liệu này thì phần tử sẽ có 512 bits, tuy nhiên vẫn có thể điều chỉnh số lượng bits thông qua biến bit_size phía ngoài (ở hàm main)

- **std::array<uint64_t, 8> data**: dữ liệu sẽ được lưu trong mảng, với bit ít quan trọng nhất (LSB) nằm ở data[0] và bit quan trọng nhất (MSB) nằm ở data[7]

b. Hàm khởi tạo

```
BigUInt512() {
 data.fill(0);
}
BigUInt512(const std::string& number) {
 data.fill(0);
 fromString(number);
}
```

- **Constructor mặc định (Default Constructor)**: Khởi tạo số **BigUInt512** bằng cách đặt tất cả các bit về 0
- **Constructor khởi tạo với chuỗi**: Khởi tạo từ chuỗi ký tự số thập phân. Chuỗi được chuyển đổi thành số thập phân lớn và lưu vào mảng data thông qua hàm **fromString**

c. Hàm chuyển đổi từ chuỗi sang số thập phân lớn và hàm chuyển đổi từ số thập phân lớn sang chuỗi

```
void fromString(const std::string& number);
std::string toString() const;
```

- Hàm **fromString()**: chuyển đổi chuỗi ký tự sang số thập phân lớn (ở trong bài tập này nhóm đã dùng uint64_t để lưu, rồi sau đó đưa vào **BigUInt512**)
- Hàm **toString()**: chuyển đổi giá trị **BigUInt512** thành chuỗi ký tự thập phân. Mục đích cho việc này là để in ra kết quả dễ dàng hơn thông qua kiểu dữ liệu string

d. Toán tử cộng

```
BigUInt512 operator+(const BigUInt512& other) const;
```

- Cộng hai số **BigUInt512** với nhau. Ở đây nhóm sử dụng kỹ thuật cộng dồn với biến *carry* để xử lý phần dư nếu xảy ra việc tràn số

e. Toán tử nhân

```
BigUInt512 operator*(const BigUInt512& other) const;
```

- Nhóm đã thiết kế hàm phụ **multiply_uint64** để nhân từng phần tử 64 bits của mảng data với nhau. Sau đó xử lý dữ liệu với tất cả các phần tử 64 bits của mảng data để cho ra được kết quả

f. Toán tử dịch bit trái và dịch bit phải

```
BigUInt512 operator<<(int shift) const;
BigUInt512 operator>>(int shift) const;
```

- Mục đích dùng để dịch bit trái hoặc phải **shift** vị trí, các bit dư ra sẽ bị loại bỏ, và các bit mới thêm vào là bit "0"

g. Toán tử so sánh

```
bool operator==(const BigUInt512& other) const;
bool operator!=(const BigUInt512& other) const;
bool operator>=(const BigUInt512& other) const;
bool operator>(const BigUInt512& other) const;
```

- So sánh hai số **BigUInt512** bằng cách duyệt qua từng phần tử của mảng **data**, duyệt từ phần tử có trọng số cao nhất data[7] xuống thấp nhất data[0] để so sánh

3. Hàm kiểm tra số nguyên tố

a. Giới thiệu

Trong mật mã học, kiểm tra tính nguyên tố là một bước quan trọng khi cần tạo ra các số nguyên tố lớn phục vụ cho việc sinh khóa và bảo vệ thông tin. Một số nguyên tố là một số tự nhiên lớn hơn 1 và chỉ có hai ước là 1 và chính nó,do đó có tính bảo mật cao khi sử dụng trong các thuật toán mã hóa. Có nhiều thuật toán kiểm tra số nguyên tố khác nhau, mỗi thuật toán có các ưu và nhược điểm riêng về độ chính xác và hiệu suất, phù hợp với từng mục đích cụ thể trong mật mã.

b. Thuật toán

- **Thuật toán phân chia thử (Trial Division):** Thuật toán này kiểm tra xem số cần kiểm tra có bị chia hết bởi bất kỳ số nào từ 2 đến căn bậc hai của nó không. Nếu không có số nào trong khoảng đó chia hết cho số cần kiểm tra, thì đó là một số nguyên tố.
    - Ưu điểm: Dễ hiểu đơn giản trong cài đặt.
    - Nhược điểm: Hiệu suất kém đặc biệt là khi xử lý trong số nguyên tố lớn.
    - Các bước thực hiện của thuật toán này:
        1. Kiểm tra điều kiện đầu vào: Nếu n≤1, n không phải là số nguyên tố. Nếu n=2 hoặc n=3, thì n là số nguyên tố.
        2. Loại bỏ số chẵn lớn hơn 2: Nếu n là số chẵn và lớn hơn 2 (tức là nnn chia hết cho 2), thì nnn không phải là số nguyên tố.
        3. Duyệt qua các số nguyên từ 3 đến **:** Chỉ cần kiểm tra các ước số lẻ từ 3 đến . Nếu nnn chia hết cho bất kỳ số nào trong dãy này, thì nnn không phải là số nguyên tố.
        4. Kết luận: Nếu không có số nào trong khoảng từ 3 đến chia hết n, thì n là số nguyên tố.

- **Thuật toán Sàng Eratosthenes:** Phương pháp này tạo ra tất cả các số nguyên tố nhỏ hơn hoặc bằng một số cho trước bằng cách đánh dấu các bội số của mỗi số nguyên tố bắt đầu từ 2.
    - Ưu điểm: Dễ hiểu đơn giản trong cài đặt.
    - Nhược điểm: Hiệu suất kém đặc biệt là khi xử lý trong số nguyên tố lớn.
    - Các bước thực hiện:
        1. Khởi tạo danh sách: Tạo một danh sách hoặc mảng đánh dấu(isPrime[]) từ 2 đến n, và ban đầu giả sử tất cả các số trong khoảng đó là số nguyên tố. Do đó, đặt isPrime[i] = True cho mọi I từ 2 đến n.
        2. Bắt đầu từ 2: Bắt đầu với số đầu tiên p = 2 (là số nguyên tố đầu tiên).
        3. Sàng các bội số:
            - Nếu isPrime[p] là True, nghĩa là p là một số nguyên tố.
            - Đánh dấu tất cả các bội số của p(bắt đầu từ p^2 vì các bội số nhỏ hơn p^2 đã được đánh dấu bởi các số nguyên tố trước đó) là False, nghĩa là chúng không phải số nguyên tố.
        4. Tăng p : Tăng giá trị của p lên 1 và lặp lại bước 3 cho đến khi p lớn hơn √n.
        5. Kết quả : Sau khi hoàn tất quá trình, các số i trong danh sách mà isPrime[i] vẫn là True sẽ là ác số nguyên tố từ 2 đến n.
- **Thuật toán Fermat:** Dựa trên định lý nhỏ Fermat, thuật toán này kiểm tra một số nguyên tố bằng cách tính toán biểu thức an-1 mod n. Nếu kết quả không bằng 1 cho bất kỳ số a nào trong khoảng từ 2 đến n-1, thì n không phải là số nguyên tố
    - Ưu điểm: Hoạt động nhanh hơn thuật toán phân chia thử.
    - Nhược điểm: Có thể đưa ra kết quả sai cho các số Carmichael(số giả nguyên tố).
- **Thuật toán Miller - Rabin:** Đây là thuật toán kiểm tra xác suất mở rộng của thuật toán Fermat. Nó kiểm tra với một loạt các giá trị ngẫu nhiên của a. Nếu số vượt qua một số lần kiểm tra nhất định, nó được coi là nguyên tố với xác suất cao.
    - Ưu điểm: Dễ cài đặt có thể tối ưu cho các số lớn, có độ phức tạp $O(k * \log^3 n)$, với k là số lần kiểm tra, thường được dùng rộng rãi trong mật mã vì độ chính xác cao và tốc độ.
    - Nhược điểm: Miller-Rabin là một thuật toán xác suất, nghĩa là nó không đảm bảo 100% tính chính xác, mặc dù xác suất sai rất thấp, nhưng không thể loại trừ hoàn toàn khả năng một số hợp số có thể được xác định là nguyên tố (gọi là số giả nguyên tố).
    - Các bước thực hiện:
        1. Kiểm tra điều kiện cơ bản:

        - Nếu n = 2, trả về “nguyên tố ” vì 2 là số nguyên tố duy nhất là số chẵn.
        - Nếu n≤1 hoặc là một số chẵn lớn hơn 2, trả về “không nguyên tố”.
    2. Chọn ngẫu nhiên a: chọn một số nguyên a ngẫu nhiên sau cho $1 < \alpha < n-1$.
    3. Tính a^(n-1)(mod n): sử dụng phương pháp “lũy thừa có lặp” (exponentiation by squaring) để tính giá trị a^(n-1)(mod n) một các hiệu quả.
    4. Kiểm tra điều kiện đồng dư:
        - Nếu $a^{n-1} \not\equiv 1 \pmod n$, kết luận rằng n không phải số nguyên tố và dừng thuật toán.
        - Nếu $a^{n-1} \equiv 1 \pmod n$, n có khả năng là số nguyên tố. Tuy nhiên, điều này không đảm bảo chắc chắn n là số nguyên tố.
    5. Lặp lại (tùy chọn):
        - Để tăng độ tin cậy, lặp lại các bước 2-4 với các giá trị a khác nhau(chẳng hạn 5-10 lần). nếu tất cả các lần kiểm tra đều thỏa mãn $a^{n-1} \equiv 1 \pmod n$, thì n có khả năng cao là số nguyên tố.
- **Thuật toán AKS (Agrawal-Kayal-Saxena):** thuật toán AKS là một trong những phương pháp đầu tiên có khả năng xác định tính nguyên tố của một số nguyên một cách hiệu quả mà không dựa vào các giả định chưa được chứng minh. Đây là một thuật toán thuộc lớp thời gian đa thức và là thuật toán đầu tiên chứng minh rằng bài toán kiểm tra nguyên tố có thể giải được trong thời gian đa thức mà không cần phải dựa vào bất kỳ giả thuyết toán học nào.
    - Ưu điểm: AKS đưa ra kết quả chính xác, không dựa vào xác suất, do đó đảm bảo tính đúng đắn 100% khi xác định một số có phải là số nguyên tố hay không.
    - Nhược điểm: AKS có thời gian chạy đa thức, hệ số trong hàm thời gian phức tạp khá lớn, khiến thuật toán không hiệu quả với các số có hàng triệu chữ số.
    - Các bước thực hiện:
        1. Kiểm tra điều kiện cơ bản:
            - Nếu n là số chính phương (tức là $n=a^b$ với $a,b \in N, b \geq 2$), thì n không phải là số nguyên tố.
        2. Tìm số r thích hợp:
            - Tìm số r sao cho bậc của n modulo r (tức là số k nhỏ nhất thỏa mãn $n^k \equiv 1 \bmod r$) lớn hơn $(\log n)^2$
        3. Kiểm tra số nguyên tố nhỏ:
            - Kiểm tra xem $1 < \gcd(a,n) < n$ cho các a từ 2 đến r. Nếu tồn tại một ước d của n trong khoảng này, kết luận rằng n là hợp số.
        4. Kiểm tra đồng dư đa thức:

- Kiểm tra xem đa thức $(x+a)^n$ có đồng dư với x^n+a mod (x^r-1,n) hay không, với mọi a nguyên từ 1 đến $\sqrt{(\varphi(r))}\log n$. Nếu bất kỳ đồng dư nào không thỏa mãn, n không phải là số nguyên tố.

5. Kết luận: Nếu *n* vượt qua tất cả các bước kiểm tra trên, kết luận rằng n là số nguyên tố.

- **Thuật toán Baillie-PSW:** Thuật toán kiểm tra tính nguyên tố kết hợp giữa hai phương pháp: kiểm tra Miller-Rabin và kiểm tra Lucas. Đây là một thuật toán xác suất nhanh và được coi là đủ chính xác cho việc xác định tính nguyên tố của các số lớn.
  - Ưu điểm: Baillie-PSW có độ chính xác gần như tuyệt đối với các số đến 2^{64}. Thực tế, không có số nào dưới giới hạn này có thể vượt qua Baillie-PSW nếu không phải là số nguyên tố.
  - Nhược điểm: Baillie-PSW không phải là kiểm tra chắc chắn; nó phụ thuộc vào việc kiểm tra xác suất, mặc dù khả năng sai sót rất nhỏ.
  - Các bước thực hiện:
    1. Kiểm tra các trường hợp đặc biệt:
       - Nếu $n \leq 1$, trả về "không nguyên tố".
       - Nếu n là một số chẵn lớn hơn 2, trả về "không số nguyên tố".
       - Nếu n là một số nhỏ nằm trong danh sách các số nguyên tố nhỏ, kết luận "nguyên tố".
    2. Kiểm tra Miller-Rabin với cơ sở 2:
       - Sử dụng cơ sở cố định là 2, thực hiện kiểm tra Miller-Rabin. Nếu không vượt qua được kiểm tra này, kết luận "không nguyên tố".
    3. Kiểm tra Lucas:
       - Thực hiện kiểm tra Lucas để kiểm tra tính số nguyên tố của n bằng cách sử dụng chuỗi Lucas đặc biệt cho *n*.
       - Nếu n không thỏa mãn điều kiện kiểm tra Lucas, kết luận "không nguyên tố".
    4. Kết luận: Nếu *n* vượt qua cả kiểm tra Miller-Rabin và Lucas, kết luận rằng *n* là số nguyên tố.

4. Số ngẫu nhiên

a. Giới thiệu: Tính ngẫu nhiên là yếu tố nền tảng của bảo mật trong mật mã học, ảnh hưởng đến các quá trình sinh khóa, mã hóa, và xác thực. Số ngẫu nhiên được sử dụng trong nhiều ứng dụng, từ sinh khóa bảo mật cho đến xác định các tham số ngẫu nhiên trong các thuật toán. Độ ngẫu nhiên của số càng cao thì hệ thống càng an toàn, vì vậy cần có các nguồn số ngẫu nhiên chất lượng cao, khó dự đoán. Tuy nhiên, việc tạo ra số ngẫu nhiên hoàn toàn không dễ dàng trong môi trường máy

tính, bởi vì máy tính vốn là một hệ thống xác định.Các nguồn số ngẫu nhiên có thể được chia thành hai loại: số ngẫu nhiên giả và số ngẫu nhiên thực

- Số ngẫu nhiên giả (Pseudo-Random Number): Số ngẫu nhiên giả được tạo từ các hàm giả ngẫu nhiên (PRNG) có hạt giống (seed) ban đầu. Từ một hạt giống cụ thể, PRNG sẽ tạo ra chuỗi số có vẻ ngẫu nhiên, nhưng thực chất là có thể được dự đoán nếu biết hạt giống. Số ngẫu nhiên giả thường được sử dụng trong các mô phỏng và ứng dụng yêu cầu ngẫu nhiên nhanh, nhưng không đủ an toàn cho mật mã học do dễ bị dự đoán.
- Số ngẫu nhiên thật (True Random Number): Số ngẫu nhiên thật được sinh ra từ các nguồn vật lý không thể dự đoán trước, như tiếng ồn nhiệt, bức xạ, hoặc thời gian thực. Do phụ thuộc vào các hiện tượng ngẫu nhiên vật lý, số ngẫu nhiên thật khó bị dự đoán và thường được sử dụng trong các hệ thống bảo mật yêu cầu tính ngẫu nhiên cao. Tuy nhiên, việc tạo ra số ngẫu nhiên thật có thể chậm hơn và đòi hỏi các thiết bị đặc biệt

b. Tính ngẫu nhiên và độ quan trọng trong mật mã học:

- Đảm bảo tính an toàn: Tính ngẫu nhiên tốt giúp tạo ra các khóa mật mã không thể dự đoán, đảm bảo thông tin mã hóa là duy nhất mỗi lần thực hiện.
- Giảm thiểu rủi ro tấn công: Các hệ thống mật mã dựa vào những yếu tố ngẫu nhiên để chống lại các tấn công dựa trên dự đoán hoặc thử hết các khả năng (brute force).

c. Hàm sinh số ngẫu nhiên an toàn trong máy tính:

Hệ thống sinh số ngẫu nhiên an toàn là một thành phần quan trọng để đảm bảo tính bảo mật trong mật mã học. Các hệ thống này sử dụng các thuật toán và nguồn ngẫu nhiên chất lượng cao để tạo ra số ngẫu nhiên khó bị dự đoán. Các hàm sinh số ngẫu nhiên an toàn thường dựa trên các phương pháp sau:

- Thuật toán DRBG (Deterministic Random Bit Generator): DRBG là một loại hàm giả ngẫu nhiên được thiết kế đặc biệt cho các ứng dụng bảo mật, sinh ra các bit ngẫu nhiên bằng các phương pháp mã hóa. DRBG được sử dụng rộng rãi trong các thư viện bảo mật để đảm bảo tính ngẫu nhiên và tính an toàn cao của các bit sinh ra.
- Bộ sinh số ngẫu nhiên an toàn của hệ thống (CSPRNG); CSPRNG thường sử dụng các hàm mật mã mạnh hoặc các giao diện hệ thống như `/dev/random` và `/dev/urandom` trên Linux để sinh số ngẫu nhiên an toàn. Các bộ sinh ngẫu nhiên này đảm bảo chất lượng số

ngẫu nhiên cao và khó bị dự đoán, là lựa chọn phổ biến cho các ứng dụng bảo mật đòi hỏi độ an toàn cao.

- Nguồn Entropy (Entropy Source):
  - Để đảm bảo độ ngẫu nhiên cao, CSPRNG thường dựa vào các nguồn entropy từ hệ thống, như hoạt động của người dùng (chuyển động chuột, nhấn phím), trạng thái phần cứng hoặc các đặc tính không đoán trước của hệ điều hành. Những nguồn này là cơ sở để khởi tạo và tiếp thêm entropy, giúp tăng cường độ khó dự đoán của số ngẫu nhiên sinh ra.
- Hạt giống (Seeding):
  - Các hệ thống sinh số ngẫu nhiên an toàn thường cần một "hạt giống" ban đầu từ nguồn entropy để khởi tạo bộ sinh số ngẫu nhiên. Việc seeding chính xác và đủ entropy là yếu tố quyết định trong việc tạo ra chuỗi số ngẫu nhiên mạnh mẽ, an toàn. Thay đổi hoặc bổ sung hạt giống giúp duy trì tính ngẫu nhiên khi hệ thống sinh nhiều số.
- Các thuật toán DRBG phổ biến:
  - Một số DRBG sử dụng rộng rãi bao gồm:
    - **HMAC_DRBG**: Sử dụng hàm HMAC để tạo chuỗi bit ngẫu nhiên.
    - **CTR_DRBG**: Sử dụng mã khối theo chế độ đếm để tạo số ngẫu nhiên.
    - **Hash_DRBG**: Sử dụng hàm băm mật mã như SHA-256.
  - Các thuật toán này đều dựa trên mật mã học và tuân theo tiêu chuẩn của Viện Tiêu chuẩn và Công nghệ Quốc gia Hoa Kỳ (NIST) để đảm bảo tính an toàn.
- CSPRNG trong các thư viện mã nguồn:
  - Nhiều ngôn ngữ lập trình hiện đại cung cấp các hàm sinh số ngẫu nhiên an toàn sẵn có trong thư viện chuẩn:
    - **Python**: secrets và os.urandom là các tùy chọn an toàn cho các ứng dụng yêu cầu bảo mật.
    - **Java**: SecureRandom được thiết kế để tạo ra các số ngẫu nhiên mật mã an toàn.
    - **C#/.NET**: Lớp RNGCryptoServiceProvider cung cấp một cách sinh số ngẫu nhiên an toàn.
  - Các thư viện này thường truy cập các API của hệ điều hành hoặc sử dụng thuật toán mật mã mạnh để sinh số ngẫu nhiên.

5. Trao đổi Diffie-Hellman

a. Lý thuyết

- Giao thức Diffie-Hellman là một phương pháp trao đổi khóa nổi tiếng, được Whitfield Diffie và Martin Hellman giới thiệu vào năm 1976. Giao thức này cho phép hai bên tham gia (thường được gọi là Alice và Bob) chia sẻ một khóa bí mật chung qua một kênh không an toàn mà không cần chia sẻ trực tiếp khóa đó.
- Diffie-Hellman hoạt động dựa trên nguyên lý toán học về phép lũy thừa modulo và độ khó của bài toán logarit rời rạc.
- Các bước cụ thể:
    + Giao thức yêu cầu một số nguyên tố lớn **p** và một cơ số **g** (phần tử sinh của **p - 1**) mà hai bên đều biết (được công khai).
    + Mỗi bên tạo ra một số ngẫu nhiên riêng tư của mình (private key), gọi là **a** cho Alice và **b** cho Bob.
    + Sau đó, mỗi bên tính một giá trị công khai (public key) dựa trên cơ số **g** và số ngẫu nhiên riêng tư của mình:
        + Alice gửi **A** = $\mathbf{g^a \bmod p}$ cho Bob.
        + Bob gửi **B** = $\mathbf{g^b \bmod p}$ cho Alice.
    + Cuối cùng, cả hai bên tính khóa bí mật chung dựa trên khóa công khai của đối phương:
        + Alice tính $\mathbf{K = B^a \bmod p}$.
        + Bob tính $\mathbf{K = A^b \bmod p}$.

Nhờ vào các tính toán trên, cả hai bên có thể tính ra cùng một khóa bí mật K, vì $\mathbf{B^a \bmod p = (g^b)^a \bmod p = (g^a)^b \bmod p = A^b \bmod p}$. Đối với kẻ tấn công, việc tính ra **K** từ các giá trị công khai là rất khó vì yêu cầu giải bài toán logarit rời rạc, vốn rất khó khi số nguyên tố **p** lớn.

b. Ứng dụng thực tế

- **Mã hóa:** Các lược đồ mã hóa khóa công khai dựa trên trao đổi khóa Diffie–Hellman đã được đề xuất. Lược đồ đầu tiên như vậy là mã hóa ElGamal. Một biến thể hiện đại hơn là Lược đồ mã hóa tích hợp.
- **Chuyển tiếp bí mật:** Các giao thức đạt được tính bảo mật chuyển tiếp tạo ra các cặp khóa mới cho mỗi phiên và loại bỏ chúng vào cuối phiên. Trao đổi khóa Diffie–Hellman là lựa chọn thường xuyên cho các giao thức như vậy, vì khả năng tạo khóa nhanh.
- **Thỏa thuận khóa xác thực bằng mật khẩu:** Khi Alice và Bob chia sẻ mật khẩu, họ có khẩu thỏa thuận khóa xác thực bằng mật khẩu (PK) của Diffie–Hellman để ngăn chặn các cuộc tấn công trung gian.

-**Khóa công khai:** Trao đổi khóa bí mật giữa hệ mã không đối xứng

6. Mô tả chương trình đã cài đặt :

a. Mô tả các phần chính

- **Lớp BigUInt512**
  - Lớp này đại diện cho các số nguyên lớn 512-bit và cung cấp các phép toán số học trên các số nguyên này như cộng, trừ, nhân, chia, và các phép toán khác.
  - Cấu trúc dữ liệu data sử dụng một mảng 64-bit để lưu trữ các giá trị số nguyên, hỗ trợ các phép tính lớn.
  - Các phép toán như +, -, *, và % được cài đặt theo cách thủ công để có thể xử lý các số nguyên lớn hơn giới hạn của kiểu dữ liệu nguyên bản trong C++.
  - Hàm modular_exponentiation tính toán lũy thừa theo mô-đun, đây là cơ sở cho các tính toán mã hóa.
- **Thuật toán Miller-Rabin (kiểm tra số nguyên tố)**
  - Hàm isProbablePrime sử dụng thuật toán Miller-Rabin để kiểm tra xem một số có khả năng là số nguyên tố hay không.
  - Trong thuật toán này, một số ngẫu nhiên được chọn làm cơ sở để kiểm tra tính nguyên tố, lặp lại nhiều lần để tăng độ chính xác.
- **Thuật toán Pollard's Rho (phân tích thừa số nguyên tố)**
  - Thuật toán này được triển khai trong hàm pollardsRho để phân tích số thành các thừa số nguyên tố, giúp tìm các ước số nguyên tố của số nguyên lớn.
  - Hàm factorize sử dụng thuật toán này để tạo danh sách các thừa số nguyên tố.
- **Tìm số nguyên tố an toàn và generator**
  - Hàm generate_safe_prime tạo một số nguyên tố an toàn p, với điều kiện (p−1)/2 cũng là một số nguyên tố.
  - Hàm findGenerator tìm một generator (số gốc) cho nhóm modulo p.

b. Giao thức Diffie-Hellman

Chương trình thực hiện các bước sau để mô phỏng trao đổi khóa Diffie-Hellman:

- **Sinh số nguyên tố an toàn và generator**:
  - Sử dụng generate_safe_prime để sinh số nguyên tố an toàn prime.
  - Sử dụng findGenerator để tìm một generator cho prime.

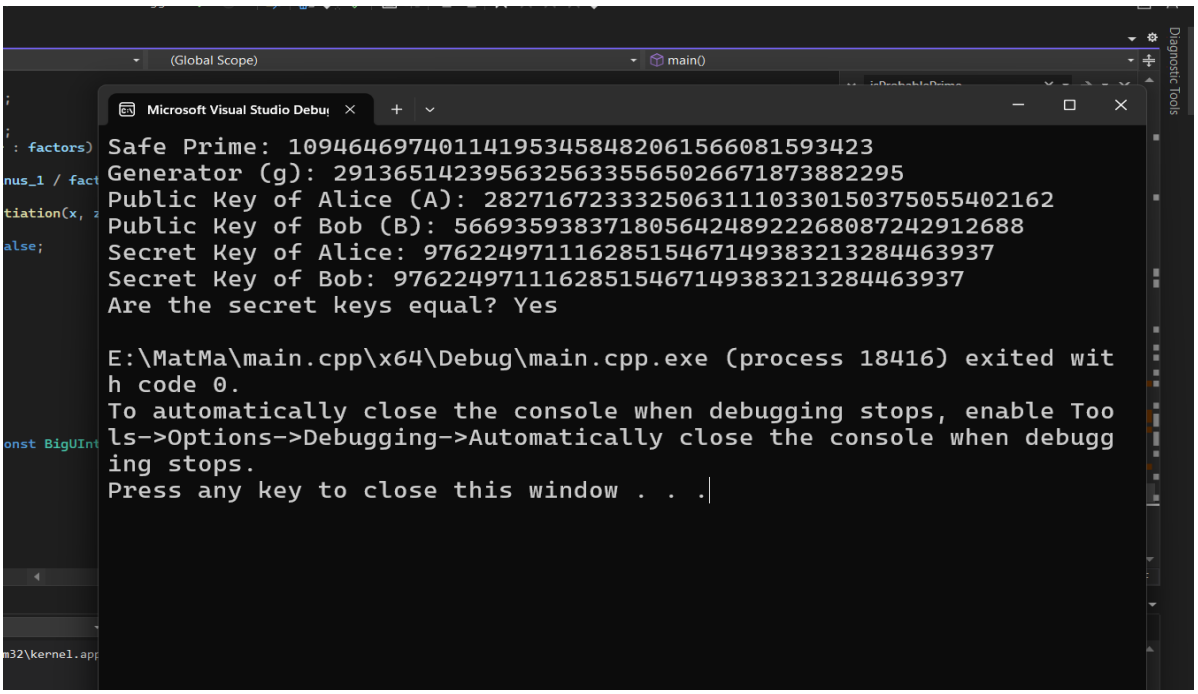
- **Sinh khóa riêng**:
  - Alice và Bob mỗi người sẽ tạo một khóa riêng ngẫu nhiên a và b bằng hàm generatePrivateKey.

- **Tính khóa công khai**:
  - Alice và Bob tính toán khóa công khai A và B bằng cách nâng generator lên lũy thừa khóa riêng theo mô-đun prime.
- **Tính khóa bí mật chung**:
  - Alice tính toán khóa bí mật chung bằng cách nâng khóa công khai của Bob B lên lũy thừa a (khóa riêng của mình) theo mô-đun prime.
  - Tương tự, Bob cũng tính toán khóa bí mật chung bằng cách nâng khóa công khai của Alice A lên lũy thừa b (khóa riêng của mình) theo mô-đun prime.

c. Output của chương trình:

- Giá trị prime (số nguyên tố an toàn).
- Giá trị generator g.
- Khóa công khai của Alice và Bob (A và B).
- Khóa bí mật của Alice và Bob.
- Kết quả so sánh khóa bí mật của Alice và Bob.

Trong hàm main(), nhóm có đặt một bit_length để thay đổi độ lớn của số nguyên tố. Mặc định hiện tại nhóm đang để 128, nhưng nếu muốn thay đổi kích thước khác chỉ cần khai báo độ lớn lớn hơn 64 (lớn hơn 512 vẫn được, nhưng chương trình sẽ khởi chạy lâu do độ lớn của số nguyên tố)



Hình ảnh chương trình khi chạy với số 128 bit, khi số bit lớn hơn thì việc tạo random 1 số và kiểm tra nó có phải là số nguyên tố không sẽ lặp đi lặp lại rất

nhiều lần do không tạo random ra được số nguyên tố, nên phải tạo random lại số khác và có thể tạo lại số cũ do không thể lưu trữ các số đã duyệt.

TÀI LIỆU THAM KHẢO

1. Wikipedia, Diffie–Hellman key exchange, en.wikipedia.org, https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

2. Wikipedia, Trao đổi khóa Diffie-Hellman, vi.wikipedia.org, https://vi.wikipedia.org/wiki/Trao_%C4%91%E1%BB%95i_kh%C3%B3a_Diffie-Hellman

3. Wikipedia, Pollard's rho algorithm, en.wikipedia.org, https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

4. Wikipedia, Modular exponentiation, en.wikipedia.org, https://en.wikipedia.org/wiki/Modular_exponentiation