**1. Hệ thống xử lý số lớn - BigInteger**

**1.1 Giới thiệu**

Lớp BigInteger được thiết kế để xử lý các phép toán trên số nguyên có kích thước lớn (lớn hơn kích thước của kiểu dữ liệu nguyên thủy trong C++). Đây là giải pháp hữu ích khi làm việc với các số có độ lớn vượt quá giới hạn của int hay long long int, đặc biệt là khi xử lý các số nguyên tố lớn cho các bài toán mật mã. Dưới đây là mô tả chi tiết về lớp BigInteger và các hàm trong lớp.

**1.2 Thiết kế và Cấu trúc của Lớp**

**a. Các thuộc tính**

* **string value:** Chuỗi lưu trữ giá trị của số lớn, được biểu diễn dưới dạng chuỗi để có thể chứa số có kích thước tùy ý.

**b. Các hàm private**

Các hàm này phục vụ cho các phép toán nội bộ trên các số lớn dưới dạng chuỗi:

* **add(string num1, string num2) const:** Thực hiện phép cộng hai chuỗi num1 và num2, trả về chuỗi kết quả.
* **subtract(string num1, string num2) const:** Thực hiện phép trừ num1 - num2, trả về kết quả dưới dạng chuỗi.
* **isGreaterOrEqual(string num1, string num2) const:** Kiểm tra xem num1 có lớn hơn hoặc bằng num2 không.
* **multiply(string num1, string num2) const:** Thực hiện phép nhân num1 \* num2, trả về chuỗi kết quả.
* **divideBy2(string num) const:** Thực hiện phép chia num cho 2.
* **mod(string num, string a) const:** Thực hiện phép chia lấy dư của num cho a.
* **modInternal(string num, string a) const:** Hàm nội bộ hỗ trợ tính phần dư, sử dụng phép trừ lặp lại.

**c. Các hàm public**

Các hàm public bao gồm constructor và các toán tử cơ bản:

* **BigInteger(string val):** Constructor nhận một chuỗi val và gán nó vào thuộc tính value.
* **BigInteger operator+(const BigInteger& other) const:** Toán tử cộng hai đối tượng BigInteger, trả về kết quả cộng dưới dạng một đối tượng BigInteger.
* **BigInteger& operator=(const BigInteger& other):** Toán tử gán, gán giá trị của other cho đối tượng hiện tại.
* **bool operator==(const BigInteger& other) const:** So sánh hai số, trả về true nếu bằng nhau.
* **bool operator!=(const BigInteger& other) const:** So sánh hai số, trả về true nếu khác nhau.
* **bool operator>=(const BigInteger& other) const:** Kiểm tra xem số hiện tại có lớn hơn hoặc bằng other không.
  + **bool operator<=(const BigInteger& other) const:** Kiểm tra xem số hiện tại có nhỏ hơn hoặc bằng other không.
  + **BigInteger operator-(const BigInteger& other) const:** Thực hiện phép trừ, trả về đối tượng BigInteger mới.
  + **BigInteger operator\*(const BigInteger& other) const:** Thực hiện phép nhân hai số lớn, trả về kết quả dưới dạng đối tượng BigInteger.
  + **BigInteger operator/(int divisor) const:** Thực hiện phép chia cho 2, các phép chia khác không được triển khai.
  + **BigInteger operator%(const BigInteger& other) const:** Thực hiện phép chia lấy dư.

**1.3 Mô tả chi tiết một số hàm**

**a. Hàm add**

Hàm cộng các số lớn, sử dụng cách cộng từng chữ số từ phải sang trái với biến nhớ (carry).

**b. Hàm subtract**

Khi thực hiện phép trừ, carry sẽ được sử dụng nếu chữ số bên trái không đủ để trừ.

**c. Hàm multiply**

Thực hiện phép nhân dựa trên phương pháp nhân thủ công, nhân từng chữ số và cộng vào đúng vị trí trong mảng kết quả, sau đó trả về chuỗi kết quả.

**d. Hàm divideBy2**

Phép chia cho 2, chia từng chữ số của chuỗi từ trái sang phải.

**e. Hàm mod**

Hàm lấy phần dư sử dụng cách chia và trừ lặp lại.

**1.4 Kết luận**

Lớp BigInteger cung cấp các phép toán cơ bản trên số lớn dưới dạng chuỗi, đảm bảo tính chính xác cho các phép toán số học lớn. Việc sử dụng chuỗi giúp lưu trữ các số có độ dài tùy ý, hỗ trợ xử lý số nguyên tố có độ lớn ít nhất 512 bit như yêu cầu.

1. **Hàm kiểm tra số nguyên tố:**
2. **Giới thiệu về Kiểm Tra Số Nguyên Tố**

Kiểm tra số nguyên tố là quá trình xác định xem một số tự nhiên có phải là số nguyên tố hay không. Số nguyên tố là một số tự nhiên lớn hơn 1 và chỉ chia hết cho 1 và chính nó. Kiểm tra số nguyên tố là một bước quan trọng trong mật mã học, vì nhiều thuật toán mã hóa, như RSA hoặc Diffie-Hellman, yêu cầu sử dụng các số nguyên tố lớn để đảm bảo tính bảo mật.

1. **Các Thuật Toán Kiểm Tra Số Nguyên Tố** 
   1. **Phương pháp Chia Thử (Trial Division)**

Phương pháp chia thử là cách kiểm tra số nguyên tố đơn giản và trực tiếp nhất:

* Ý tưởng cơ bản là kiểm tra xem số n có chia hết cho bất kỳ số nào từ 2 đến không.
* Nếu có một ước số nào trong khoảng này, thì n không phải là số nguyên tố; nếu không, n là số nguyên tố.

Ví dụ: Để kiểm tra số 29, ta thử chia nó cho các số nguyên từ 2 đến (tức là 5). Vì 29 không chia hết cho bất kỳ số nào trong khoảng này, nên 29 là số nguyên tố.

**Ưu điểm**:

* Phương pháp dễ hiểu và dễ triển khai, đặc biệt phù hợp với các số nhỏ.

**Nhược điểm**:

* Với các số lớn, phương pháp này rất kém hiệu quả vì số lượng phép chia tăng theo căn bậc hai của n. Điều này khiến nó không thực tiễn cho các ứng dụng cần kiểm tra các số nguyên tố lớn, như trong mật mã học.
  1. **Thuật toán Fermat**

Thuật toán Fermat là một phương pháp kiểm tra số nguyên tố xác suất dựa trên định lý Fermat nhỏ:

* Định lý Fermat nhỏ phát biểu rằng: Với một số nguyên tố p và một số nguyên a sao cho , thì
* Để kiểm tra tính nguyên tố của n, ta chọn ngẫu nhiên một số a trong khoảng và tính
  + Nếu kết quả là 1, thì n có khả năng là số nguyên tố.
  + Nếu kết quả khác 1, thì n chắc chắn không phải là số nguyên tố.

Thuật toán Fermat thực hiện nhiều lần với các giá trị ngẫu nhiên khác nhau của a. Nếu tất cả các lần kiểm tra đều cho kết quả là 1, thì n có thể là số nguyên tố. Tuy nhiên, có những số không nguyên tố vẫn thỏa mãn điều kiện của định lý Fermat cho nhiều giá trị của a (được gọi là số **Carmichael**), nên phương pháp này không hoàn toàn đáng tin cậy.

**Ưu điểm**:

* Hiệu quả hơn phương pháp chia thử cho các số lớn, đặc biệt khi số lần lặp tăng lên.

**Nhược điểm**:

* Đây là phương pháp xác suất nên không hoàn toàn chính xác. Đối với một số số đặc biệt (số Carmichael), thuật toán có thể trả về kết quả sai, khiến nó không an toàn cho các ứng dụng yêu cầu độ bảo mật cao.
  1. **Thuật toán Miller-Rabin**

Thuật toán Miller-Rabin là một thuật toán kiểm tra số nguyên tố xác suất nâng cao, khắc phục được hạn chế của thuật toán Fermat:

* Thuật toán này kiểm tra tính nguyên tố của n bằng cách kiểm tra dạng biểu diễn của là , với d là số lẻ.
* Sau đó, chọn một số ngẫu nhiên a trong khoảng và tính
  + Nếu hoặc , thì n có khả năng là số nguyên tố.
  + Nếu không, thuật toán tiếp tục bình phương x (với ) để kiểm tra các giá trị. Nếu trong quá trình này x trở thành n - 1, thì n vẫn có khả năng là số nguyên tố.
  + Nếu không, n không phải là số nguyên tố.

Giống như thuật toán Fermat, Miller-Rabin cũng lặp lại quy trình nhiều lần với các giá trị ngẫu nhiên khác nhau của a. Sau mỗi lần kiểm tra, xác suất để n là số nguyên tố tăng lên. Khi lặp đủ số lần kiểm tra, xác suất sai sẽ rất nhỏ, khiến thuật toán Miller-Rabin trở thành lựa chọn tin cậy cho các ứng dụng mật mã.

**Ưu điểm**:

* Hiệu quả cao, đặc biệt với các số lớn, phù hợp cho các ứng dụng mật mã.
* Độ chính xác cao, giảm thiểu khả năng sai số nhờ vào cách xử lý với các số ngẫu nhiên a.

**Nhược điểm**:

* Là phương pháp xác suất, nên vẫn có xác suất sai số rất nhỏ. Tuy nhiên, khả năng sai số này có thể giảm xuống thấp không đáng kể bằng cách tăng số lần kiểm tra.

1. **So sánh các thuật toán**

| **Thuật toán** | **Độ chính xác** | **Hiệu quả với số lớn** | **Ứng dụng** |
| --- | --- | --- | --- |
| Chia thử | Chính xác tuyệt đối | Kém hiệu quả | Số nhỏ |
| Fermat | Xác suất, có thể sai với số Carmichael | Tốt hơn | Số lớn nhưng cần lặp nhiều lần |
| Miller-Rabin | Xác suất, độ chính xác cao hơn | Rất tốt | Số lớn và dùng trong mật mã |

1. **Safe Prime**
   1. **Khái Niệm Safe Prime**

Safe prime (số nguyên tố an toàn) là một số nguyên tố có dạng:

Trong đó:

* p là một số nguyên tố,
* q cũng là một số nguyên tố.

Ví dụ, với q = 11, ta có thể tính , và 23 là một safe prime vì cả 23 và 11 đều là số nguyên tố.

* 1. **Lý do chọn Safe Prime trong mật mã học**

Safe prime được ưa chuộng trong mật mã học vì chúng tăng cường độ bảo mật và làm cho việc phân tích số trở nên khó khăn hơn. Các ứng dụng quan trọng của safe prime bao gồm:

* **Trao đổi khóa Diffie-Hellman**: Safe prime thường được sử dụng để tạo các tham số trong giao thức Diffie-Hellman. Giao thức này cần một số nguyên tố lớn và một phần tử sinh (generator) để tính toán khóa chung giữa các bên giao tiếp. Khi sử dụng safe prime, nó sẽ làm cho việc tấn công giao thức trở nên khó khăn hơn.
* **Chống lại các cuộc tấn công**: Safe prime giúp ngăn ngừa các cuộc tấn công dựa trên tính toán nhóm con (subgroup attacks). Khi p là một safe prime, nó đảm bảo rằng nhóm các phần tử sinh sinh ra là nhóm có độ lớn lớn nhất, làm giảm cơ hội để kẻ tấn công khai thác các nhóm con nhỏ hơn trong các tính toán khóa mật mã.
  1. **Các đặc điểm của Safe Prime**

Một safe prime có các đặc điểm nổi bật sau:

* **An toàn trong việc chọn generator**: Khi sử dụng safe prime, việc chọn phần tử sinh g (generator) để sinh ra toàn bộ các phần tử modulo p (ngoại trừ 0) sẽ dễ dàng hơn.
* **Tăng độ phức tạp tính toán cho kẻ tấn công**: Với safe prime, các phép toán trong mật mã sẽ phức tạp hơn, yêu cầu nhiều phép tính hơn và tốn kém hơn về mặt tính toán, nhờ đó tăng độ an toàn.
  1. **Cách tạo Safe Prime**

Để tạo một safe prime, quá trình thường bao gồm các bước sau:

* **Sinh một số nguyên tố nhỏ hơn (q)**: Chọn một số nguyên tố lớn ngẫu nhiên q.
* **Tính toán p = 2q + 1**: Với q đã chọn, tính p theo công thức trên.
* **Kiểm tra tính nguyên tố của p**: Sử dụng thuật toán kiểm tra số nguyên tố (như Miller-Rabin) để đảm bảo rằng p cũng là số nguyên tố.
* **Lặp lại nếu cần thiết**: Nếu p không phải là số nguyên tố, chọn lại q và tính lại p cho đến khi có một safe prime.

1. **Kết luận**

Trong các ứng dụng yêu cầu kiểm tra số nguyên tố lớn, đặc biệt là các ứng dụng mật mã, thuật toán Miller-Rabin thường được ưu tiên nhờ vào hiệu quả và độ chính xác cao. Mặc dù là phương pháp xác suất, Miller-Rabin cung cấp mức độ tin cậy cao khi lặp lại đủ số lần kiểm tra, giúp nó trở thành một công cụ quan trọng trong các hệ thống mã hóa.

1. **Số ngẫu nhiên**
2. **Giới thiệu về Tính Ngẫu Nhiên**

Tính ngẫu nhiên là khả năng xuất hiện các giá trị không tuân theo một quy luật hay dự đoán trước. Trong máy tính, số ngẫu nhiên thường được sử dụng để tạo khóa, sinh các giá trị ban đầu và nhiều mục đích khác trong tính toán, đặc biệt là trong mật mã học.

1. **Vai trò của Tính Ngẫu Nhiên trong Mật Mã Học**

Tính ngẫu nhiên có vai trò vô cùng quan trọng trong mật mã học, đặc biệt để đảm bảo an toàn cho các hệ thống bảo mật:

* **Sinh khóa bí mật**: Mã hóa khóa công khai (như RSA) và các hệ thống trao đổi khóa (như Diffie-Hellman) cần các khóa bí mật được tạo ngẫu nhiên để tránh bị dự đoán hoặc tấn công.
* **Giá trị khởi tạo (IV)**: Trong các chế độ mã hóa, các giá trị khởi tạo ngẫu nhiên giúp đảm bảo rằng các bản mã không lặp lại ngay cả khi nội dung giống nhau, giúp bảo vệ nội dung khỏi các cuộc tấn công dựa trên mẫu.
* **Chữ ký số và bảo vệ giao dịch**: Số ngẫu nhiên giúp tạo ra các chữ ký số độc nhất, khó bị giả mạo và bảo vệ các giao dịch khỏi bị sao chép.

Nếu các số ngẫu nhiên không đủ an toàn hoặc dễ dự đoán, kẻ tấn công có thể lợi dụng để phá vỡ hệ thống bảo mật. Do đó, việc sinh ra các số ngẫu nhiên an toàn là rất cần thiết để đảm bảo tính bảo mật của toàn bộ hệ thống.

1. **Các Loại Số Ngẫu Nhiên**

Có hai loại số ngẫu nhiên phổ biến:

* **Số ngẫu nhiên giả (Pseudo-Random Numbers - PRNs)**: Các số này được sinh ra từ các thuật toán và phụ thuộc vào một "seed" (hạt giống). Mặc dù chuỗi số được tạo ra có vẻ ngẫu nhiên, chúng có thể được tái tạo lại nếu biết "seed". PRNs thường được sử dụng khi cần ngẫu nhiên trong tính toán, nhưng không phù hợp cho các ứng dụng bảo mật.
* **Số ngẫu nhiên thật (True Random Numbers - TRNs)**: Các số này được sinh ra từ các nguồn ngẫu nhiên thực sự, như âm thanh, ánh sáng, hoặc các dao động nhiệt, nhằm đảm bảo rằng không có mẫu dự đoán. Các số ngẫu nhiên thật đảm bảo độ ngẫu nhiên cao hơn và thường được sử dụng trong mật mã học.

1. **Yêu cầu của Số Ngẫu Nhiên An Toàn trong Mật Mã Học**

Một số ngẫu nhiên an toàn trong mật mã học cần phải đảm bảo các yếu tố sau:

* **Không thể dự đoán trước**: Các giá trị ngẫu nhiên không được phép bị dự đoán bởi bất kỳ giá trị nào trước đó.
* **Đủ độ dài và độ rộng bit**: Đảm bảo độ lớn và phạm vi số ngẫu nhiên đủ lớn để không dễ dàng dò tìm qua brute-force.
* **Khả năng chống trùng lặp**: Số ngẫu nhiên an toàn cần đảm bảo rằng các giá trị sinh ra không bị trùng lặp, đặc biệt trong các ứng dụng quan trọng như sinh khóa hoặc chữ ký số.

1. **Cách Tạo Số Ngẫu Nhiên An Toàn Trên Máy Tính**

Trên máy tính, số ngẫu nhiên an toàn có thể được tạo bằng cách kết hợp các nguồn ngẫu nhiên thật với các thuật toán mạnh để sinh số ngẫu nhiên giả.

Dưới đây là một số phương pháp tạo số ngẫu nhiên an toàn:

* 1. **Sử dụng *std::random\_device* và *std::mt19937* trong C++**

Trong C++, thư viện random cung cấp các công cụ để sinh số ngẫu nhiên an toàn. Cụ thể, ta có thể dùng *std::random\_device* để lấy *seed* an toàn từ hệ thống và dùng nó để khởi tạo bộ sinh số ngẫu nhiên *std::mt19937*. Tuy nhiên, chỉ *std::random\_device* mới là nguồn ngẫu nhiên an toàn, còn *mt19937* là bộ sinh số ngẫu nhiên giả.

Ví dụ:

*#include <random>*

*#include <iostream>*

*int main() {*

*// Sử dụng std::random\_device để lấy seed an toàn*

*std::random\_device rd;*

*std::mt19937 gen(rd()); // Khởi tạo bộ sinh số ngẫu nhiên giả với seed từ random\_device*

*std::uniform\_int\_distribution<int> dist(0, 100);*

*// Sinh số ngẫu nhiên trong khoảng từ 0 đến 100*

*int random\_number = dist(gen);*

*std::cout << "Số ngẫu nhiên: " << random\_number << std::endl;*

*return 0;*

*}*

* 1. **Sử dụng */dev/random* và */dev/urandom* trên Linux**

Trên hệ thống Linux, */dev/random* và */dev/urandom* là các thiết bị cung cấp số ngẫu nhiên an toàn:

* ***/dev/random***: Cung cấp các số ngẫu nhiên thật dựa trên các dao động hệ thống. Khi hết entropy, nó sẽ tạm ngừng để đợi thêm entropy, nên có thể chậm.
* ***/dev/urandom***: Lấy số ngẫu nhiên từ entropy khi có, nhưng khi hết sẽ sử dụng thuật toán để tiếp tục sinh số, tốc độ nhanh hơn nhưng kém an toàn hơn.

Ví dụ:

*#include <fstream>*

*#include <iostream>*

*int main() {*

*std::ifstream urandom("/dev/urandom", std::ios::binary);*

*if (urandom) {*

*unsigned int random\_value;*

*urandom.read(reinterpret\_cast<char\*>(&random\_value), sizeof(random\_value));*

*std::cout << "Số ngẫu nhiên từ /dev/urandom: " << random\_value << std::endl;*

*urandom.close();*

*} else {*

*std::cerr << "Không thể mở /dev/urandom" << std::endl;*

*}*

*return 0;*

*}*

* 1. **Sử dụng Thư Viện Mã Hóa Chuyên Dụng**

Các thư viện mật mã như OpenSSL, Crypto++, hoặc thư viện trong các ngôn ngữ như Python, cung cấp các hàm sinh số ngẫu nhiên an toàn và đã được kiểm định. Ví dụ, trong OpenSSL, ta có thể sử dụng RAND\_bytes() để sinh số ngẫu nhiên cho các ứng dụng mật mã.

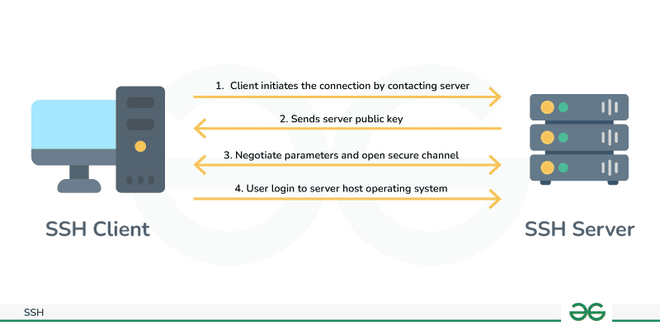
1. **Kết Luận**

Tính ngẫu nhiên là yếu tố không thể thiếu trong mật mã học. Để bảo vệ các hệ thống mật mã khỏi bị xâm nhập, các giá trị ngẫu nhiên phải đảm bảo tính không thể đoán trước, đủ độ rộng và chống trùng lặp. Việc sử dụng các phương pháp sinh số ngẫu nhiên an toàn như *std::random\_device, /dev/urandom*, hoặc thư viện mã hóa sẽ giúp tăng cường độ bảo mật cho các ứng dụng, đặc biệt là trong các hệ thống cần tạo khóa bí mật hoặc các tham số bảo mật.

1. **Trao đổi khóa Diffie-Hellman**

Thuật toán Diffie-hellman có rất nhiều ứng dụng quan trọng trong thực tế đặc biệt là sử dụng như một giao thực để trao đổi khóa. Một trong những ứng dụng tiêu biểu nhất của Diffie-hellman trong thực tế là Secure Shell.

* 1. **Secure Shell (SSH) là gì?**
* Secure Shell là một giao thức hỗ trợ các nhà quản trị mạng truy cập vào máy chủ từ xa thông qua mạng internet không bảo mật. Nói cách khác thì SSH là một giao thức mạng được sử dụng để truyền dữ liệu mã hóa qua mạng. SSH cho phép user kết nối với server mà không cần phải nhớ mật khẩu hay nhập mật khẩu cho từng hệ thống. Nó luôn đi kèm với hai khóa:
  + Public key: sử dụng cho mã hóa.
  + Private key: sử dụng cho giải mã.
  1. **Cơ chế của SSH.**



* Khởi tạo kết nối: Client kết nối đến server qua SSH, đầu tiên hai bên sẽ lựa chọn các thuật toán mã hóa, băm, và trao đổi khóa mà cả hai đều hỗ trợ.
* Xác thực server: Để đảm bảo rằng client đang kết nối với đúng server, server gửi public key của nó đến client. Client sẽ kiểm tra khóa này với danh sách các khóa mà nó đã lưu trước đó.
* Trao Đổi Khóa và Thiết Lập Kênh Mã Hóa: Để thiết lập kênh mã hóa bảo mật, SSH sử dụng thuật toán trao đổi khóa (thuật toán **Diffie-Hellman**)
* Xác Thực Người Dùng: Người dùng tiến hành gửi mật khẩu đã được mã khóa để server xác nhận.
  1. **Chức năng của SSH.**
* Cung cấp giao thức bảo mật cao cho các thông tin được trao đổi giữa client và server.
* Cung cấp giao thức bảo mật cho trao đổi file.
* Cho phép đăng nhập từ xa.
  1. **Ứng dụng của SSH.**
* Sử dụng trong datacenter: Giao thức SSH được sử dụng trong hầu hết datacenter.
* Ứng dụng cho hệ thống đăng nhập một lần: sử dụng trong hệ thống đăng nhập một lần SSO. Trong đó, người dùng có thể dễ dàng đăng nhập hoặc chuyển qua lại giữa các tài khoản một cách nhanh chóng.
* Mã hóa dữ liệu: SSH hỗ trợ mã hóa dữ liệu file, web, nhập lệnh …
* Xác thực thông tin: SSH có thể sử dụng để kết hợp với ID người dùng và mật khẩu khi xác thực thông tin.

1. **Mô tả chương trình đã cài đặt**

**Mô tả chương trình**

Chương trình đã cài đặt thực hiện giao thức chia sẻ khóa Diffie-Hellman. Đây là một phương pháp để hai bên (Alice và Bob) có thể chia sẻ một khóa bí mật qua một kênh công khai. Chương trình sử dụng lớp BigInteger để hỗ trợ các phép toán trên các số nguyên lớn và các hàm để tạo số nguyên tố an toàn, tính khóa công khai và khóa chung.

**Các bước chính của chương trình:**

1. **Tạo số nguyên tố an toàn (p) và căn nguyên thủy (g):**
   * Chương trình tạo một số nguyên tố an toàn p có kích thước 512-bit. Số nguyên tố an toàn là số nguyên tố sao cho (p-1)/2 cũng là số nguyên tố.
   * g là căn nguyên thủy của p, được chọn là 2 cho đơn giản.
2. **Sinh khóa riêng và công khai:**
   * Cả Alice và Bob đều tạo khóa riêng ngẫu nhiên trong khoảng [2, p-2].
   * Dựa trên khóa riêng và căn nguyên thủy, Alice và Bob tính toán khóa công khai bằng hàm lũy thừa theo mô đun (modular\_exponentiation).
3. **Tính khóa chung:**
   * Sử dụng khóa công khai của đối phương, mỗi bên tính khóa chung qua hàm lũy thừa mô đun. Khóa chung này được kỳ vọng giống nhau ở cả hai bên, nhờ vào tính chất của giao thức Diffie-Hellman.
4. **Xác nhận khóa chung:**
   * Chương trình kiểm tra và in ra kết quả so sánh hai khóa chung của Alice và Bob để đảm bảo chúng trùng nhau.

**Cách chạy chương trình**

**Để biên dịch và chạy chương trình:**

1. **Chuẩn bị các tệp tin mã nguồn:**
   * Đảm bảo các tệp mã nguồn (main.cpp, modular\_exponentiation.cpp, BigInteger.cpp, generate\_safe\_prime.cpp, generate\_private\_key.cpp) nằm trong cùng một thư mục.
2. **Biên dịch chương trình:**
   * Sử dụng trình biên dịch g++ để biên dịch chương trình:

**g++ main.cpp modular\_exponentiation.cpp BigInteger.cpp generate\_safe\_prime.cpp generate\_private\_key.cpp -o diffie\_hellman**

1. **Chạy chương trình:**
   * Sau khi biên dịch, chạy chương trình với lệnh:

**./diffie\_hellman**

* + Kết quả sẽ in ra khóa chung (shared secret) của Alice và Bob cùng với thông báo xác nhận hai khóa có trùng khớp không.

**Điều chỉnh tham số**

Ta có thể thay đổi kích thước số nguyên tố hoặc các tham số khác để điều chỉnh độ bảo mật của chương trình:

1. **Thay đổi kích thước số nguyên tố an toàn (p):**
   * Trong hàm generateSafePrime, ta có thể thay đổi kích thước bit của số nguyên tố. Ví dụ, để tăng độ bảo mật lên 1024-bit, thay dòng:

**BigInteger p = generateSafePrime(512);**

thành:

**BigInteger p = generateSafePrime(1024);**

* + Kích thước lớn hơn sẽ tăng độ bảo mật nhưng cũng khiến chương trình chạy chậm hơn do các phép toán trên số lớn tốn nhiều tài nguyên hơn.

1. **Thay đổi căn nguyên thủy (g):**
   * Thay đổi giá trị g nếu cần. Ví dụ, ta có thể thử các giá trị khác như 5 hoặc 7:

**BigInteger g("5");**

* + Đảm bảo rằng g là căn nguyên thủy của p để đảm bảo tính bảo mật của giao thức.

1. **Điều chỉnh phạm vi khóa riêng:**
   * Có thể thay đổi phạm vi của khóa riêng (private\_key\_A và private\_key\_B). Thay vì [2, p-2], ta có thể thử các phạm vi hẹp hơn, nhưng điều này sẽ làm giảm bảo mật.
2. **Kích thước của khóa riêng:**
   * Nếu generate\_private\_key cho phép, ta có thể điều chỉnh độ dài bit của khóa riêng để có thêm lựa chọn độ phức tạp.