Álgebra A
Documentação - Trabalho Prático 1

Caroline Carvalho, Déborah Yamamoto, Iago Zagnoli, Manuel Junio

Resumo

Essa documentação tem como objetivo reportar o funcionamento do programa criado pelo grupo que atende a três funcionalidades principais: dado um número primo `n` indicar qual é o próximo primo `p` mais próximo, retornar um elemento gerador `g` de `p` e dado `g` determinar o logarítmo discreto de um elemento no grupo.

1 Desenvolvimento do Trabalho

Para a realização do trabalho, todos os integrantes desenvolveram códigos próprios para praticar os conceitos abordados em aula. No entanto, selecionamos apenas uma das implementações para ser enviada. Além disso, os quatro alunos desempenharam outras funções essenciais, tais como a análise e a escrita do relatório do trabalho prático.

Em relação ao algoritmo, havia três preocupações fundamentais ao realizar a implementação. Primeiramente, era necessário implementar o algoritmo de *Miller-Rabin* para encontrar o primeiro número primo $P >$ N. Em seguida, uma vez obtido esse resultado, o objetivo era determinar um elemento gerador ou, alternativamente, um elemento de ordem elevada no caso de não existir um gerador do grupo multiplicativo Zp. Finalmente, dado o gerador G, a última preocupação era calcular o logaritmo discreto de um elemento dentro do grupo. Contudo, devido à magnitude dos números envolvidos, houve a necessidade de escolher uma linguagem de programação que suportasse essa manipulação.

Para resolver essa questão, utilizamos a linguagem C++, que oferece suporte a tipos de dados adequados para manipulação de grandes números inteiros. Em particular, empregamos o uso da biblioteca Boost Multiprecision que permite a manipulação de números de precisão arbitrária, ou seja, números que podem ter uma quantidade de dígitos muito maior do que os tipos de dados primitivos padrão da linguagem (como int, long, entre outros).

Portanto, a escolha da linguagem C++ e do tipo do Boost Multiprecision foi fundamental para garantir a correta implementação do algoritmo. Essa escolha permitiu lidar com números extremamente grandes de maneira eficiente, atendendo assim às exigências computacionais do trabalho. A colaboração entre os integrantes na análise e na redação do relatório, juntamente com a escolha tecnológica apropriada, foram essenciais para o funcionamento do projeto.

2 Descrição dos módulos e sua inter-depência

Este projeto organiza-se em três pastas principais: "include", "src"e "build". Na primeira pasta, encontram-se os arquivos responsáveis por definir o tipo das funções e a classe utilizada neste trabalho. Na segunda, estão os arquivos que descrevem a implementação real dessas funções, detalhando seu funcionamento. Por fim, na terceira pasta, são armazenados os arquivos objeto resultantes da compilação do código. No entanto, nossas principais pastas de interesse serão as duas primeiras.

2.1 Include

Dentro desta pasta, encontra-se somente o arquivo "functions.h". Este arquivo é responsável por definir a classe "Functions", a qual contém todas as funções e definições essenciais para a execução do trabalho.

2.2 Src

Dentro desta pasta, encontramos a implementação das funções definidas em "functions.h"e o código principal de execução.

2.2.1 functions.cpp

Este arquivo contém a implementação de todas as funções definidas em "functions.h", tornando-o dependente deste último. Abaixo, segue uma lista mais detalhada de cada uma dessas funções:

- *Functions()*: este é um construtor da classe "Functions"
- *fexp()*: esta função realiza uma exponenciação rápida
- *composite()*: esta função verifica se um número é composto para utilizar no algoritmo de *Miller Rabin*.
- *millerRabin()*: realiza a verificação se um dado número é primo.

- *NextPrime()*: encontra qual o próximo primo p de acordo com a entrada recebida n.

- *primeFact()*: este algoritmo realiza a fatoração de um número n utlizando uma maneira trivial

- *PollardRho()*: função baseada no algoritmo de *Pollar Rho*, particularmente eficaz para encontrar pequenos fatores primos de um número composto.

- *factorize()*: é uma função de fatoração de números inteiros utilizando o teste de primalidade de *Miller-Rabin* e o algoritmo de *Pollard rho* para encontrar fatores.

- *primeFactRho()*: encontra a fatoração de n utilizando o algoritmo de *Pollard Rho*.

- *Generator()*: Encontra um numero que é um possível gerador para Zp e a ordem mínima dele

- *discLogBrute()*: implementação para encontrar o logaritmo discreto por força bruta

- *discLogBabyGiantStep()*: implementação para encontrar o logaritmo discreto através do algoritmo *baby-step giant-step*.

- *mod-inv()*: calcula o inverso modular de um número x em relação a um módulo m usando o Algoritmo Estendido de Euclides.

- *congPair()*: Essa função cria equações de congruência, de modo que retorna um par (y, z) que é equivalente a $x \equiv y \pmod{z}$.

- *chinese-remainder()*: esta função resolve um sistema de congruências utilizando o Teorema Chinês dos Restos.

- *discLogPohligHellman()*: encontra o logaritmo discreto através do algoritmo de *Pohlig Hellman*.

- *DiscreteLogarithm()*: imprime o logaritmo discreto de um módulo p em base g e o tempo de processamento da resposta caso seja computado em menos de 20 segundos. Caso contrário, avisa para o usuário que o tempo excedeu o limite dos 20 segundos.

2.2.2 tp1.cpp

Neste arquivo, é realizada a manipulação com dados de entrada e também são chamadas as funções para encontrar o próximo primo, o gerador e o logaritmo discreto. Para isso, ele depende do arquivo "functions.h", onde essas funções estão definidas

3 Descrição Formato de Entrada dos Dados

O programa irá receber como entradas os valores de dois inteiros `N` e `a`.

- O valor de `N` é um número maior do que zero e possui precisão arbitrária. Ele recebe o valor do número que será utilizado como parâmetro para calcular o próximo primo 'p'.
- O valor de `a` deve pertencer ao grupo multiplicativo dos inteiros módulo p. `a` é o valor utilizado para calcular o logarítmo discreto da seguinte equação:

$$g^x \equiv a(modp) \tag{1}$$

4 Descrição Formato de Saída dos Dados

O programa irá retornar:

- O próximo número `p` primo de `N` e quantas vezes o algoritmo de Miller- Rabin, utilizado para calcular `p`, foi chamado.
- Um número `g` gerador de $\mathbb{Z}_p$. Em alguns casos, encontrar o valor `g` pode ser demorado ou computacionalmente intensivo. Dessa forma, quando for esse o caso, o programa retorna um valor de ordem alta, ou seja, deve retornar um `g` que tenha ordem `k` alta, próxima a `p-1` e além disso, o programa também deve retornar uma estimativa da ordem mínima, o que ajuda a entender o quão perto o valor fornecido está de ser um gerador de $\mathbb{Z}_p$.
- O logaritmo discreto de (1). Entretanto, dependendo do valor de `p` a resposta pode não ser calculada em tempo razoável. Para tratar esse caso, o cálculo de (1) possui um tempo limite de 20 segundos para ser executado, caso ele seja ultrapassado, apenas é retornado para o usuario de que o tempo limite foi excedido, de outro modo, o valor do logaritmo discreto é retornado e o tempo de cálculo também.

5 Como utilizar o programa

Para executar o código implementado, o procedimento é simples e direto. Primeiramente, é preciso acessar um terminal Linux e navegar até o diretório onde o projeto está localizado. Em seguida, deve-se inserir o comando "make"para compilar e construir o projeto. Após essa etapa, para executar o programa, basta digitar "./main"no terminal e fornecer os valores desejados para realizar os testes.

6 Estudo de Complexidade do Programa

Para melhor compreensão da complexidade do programa, cada módulo será analisado separadamente de acordo com dois parâmetros: espaço e tempo.

6.1 Módulo 1 - Encontrar p

O primeiro módulo é composto por quatro funções principais: *NextPrime()*, *millerRabin(ll pp)*, *composite(ll n, ll a, ll d, ll s)* e *fexp(ll a, ll b, ll m)*.

6.1.1 Tempo

A função principal, chamada no main, é a função *NextPrime()*, ela irá calcular p e o número de vezes que a função *millerRabin(ll pp)* é chamada. Nesse sentido, para facilitar a compreensão da complexidade de cada função e suas interdependências, iremos calcular a complexidade de cada uma conforme sua ocorrência e depois explicaremos a sua relação com *NextPrime()*.

- *fexp()*: Nessa função, o número de iterações é proporcional a $\log b$, já que, o looping funciona enquanto $b > 0$ e b é dividido por dois a cada operação, portanto, sua complexidade é $O(\log b)$. Além disso, a complexidade das multiplicações modulares que acontecem dentro do looping são constantes. Assim, a complexidade final é $O(\log n)$.

- *composite()*: Primeiramente, a função calcula $a^d \mod n$, utilizando a função *fexp()*, que como visto anteriormente, possui complexidade $O(\log n)$. Posterior a isso, o if possui complexidade constante $O(1)$. Depois, o looping principal possui uma quantidade de iterações proporcional a $O(\log n)$, já que ele realiza $s - 1$ operações e s é o número de fatores de 2 de n, em $2^s \times d$ dado como parâmetro da função. Assim, a complexidade final é $O(s \times \log n) = O(\log n \times \log n) = O(\log^2 n)$.

- *millerRabin()*: Nessa função, os principais responsáveis pela sua complexidade de tempo são seus dois loopings. No primeiro, `d` é dividido até que se torne ímpar, no pior caso, isso ocorre $\log pp$ vezes, portanto, sua complexidade é $O(\log n)$. No segundo looping, o número de iterações é proporcional ao tamanho do vetor de primos, $O(k)$. Dentro do looping, a complexidade é dominada pela chamada da função *composite()*, que como vimos anteriormente possui complexidade igual a $O(\log^2 n)$. Portanto, a complexidade final é: $O(\log n) + O(k \times log^2 n) = O(k \times \log^2 n)$.

Nesse sentido, a função *NextPrime()* chama a função *millerRabin(ll pp)* `cnt` vezes para determinar o próximo número primo. O número de iterações depende da distribuição de números primos consecutivos, isso pode ser estimado como sendo $\log p$. Portanto, a complexidade final da função é

$$O(\log p) \times O(k \times \log^2 n) = O(\log p \times k \times \log^2 n)$$

6.1.2 Espaço

Todas as funções desse módulo utilizam uma quantidade fixa de memória, independente do tamanho das entradas, resultando em uma complexidade de espaço constante, $O(1)$.

6.2 Módulo 2 - Encontrar g

O segundo módulo é composto por cinco funções principais: *primeFactRho(ll n)*, *factorize(ll n)*, *PollardRho(ll n)*, *primeFact(ll n, long long lim)* e *Generator()*.

6.2.1 Tempo

A função principal, chamada no main, é a função *Generator*, ela irá imprimir o valor de `g`. Nesse sentido, para facilitar a compreensão da complexidade de cada função e suas interdependências, iremos calcular a complexidade de cada uma conforme sua ocorrência e explicaremos sua relação com *Generator*.

- *PollardRho(ll n)*: Esse é um algoritmo probabilístico, portanto, nem a complexidade de execução nem seu sucesso é garantido, embora o procedimento seja muito eficiente na prática. Entretanto, é possível realizar uma análise com base no seu comportamento esperado. Nesse sentido, a sua complexidade é definida, principalmente, pelo while, nele durante cada iteração a seguinte recorrência é utilizada:

  $$x_i = (x_{i-1}^2 \mod n)$$

  para produzir o próximo valor de x_i na sequência infinita:

  $$x_1, x_2, x_3...; \tag{2}$$

E embora não exista nenhuma garantia de que o algorítimo irá retornar uma resposta para d, existem fortes chances de que, após $n^{(1/4)}$ iterações isso ocorra. Isso se dá pela relação desse algorítimo com o paradoxo do aniversário, que é um resultado da teoria das probabilidades, ele afirma que, em um conjunto de k elementos, aqui consideramos eles como dias do ano, a probabilidade de duas pessoas compartilharem o mesmo aniversário é surpreendentemente alta. De forma análoga, como n é um número finito e (2) é uma sequência infinita, então, eventualmente, a sequência entrará em ciclo.

Por conseguinte, é sabido que para um espaço de n possíveis valores, a sequência precisa de cerca de $\sqrt{n}$ elementos antes que a probabilidade de colisão seja significativa. O algorítimo de Pollard explora disso utilizando dois ponteiros que se movem com velocidades diferentes. Esse algoritmo é conhecido como "Tortoise and Hare algorithm", a Tartaruga e a Lebre, e tem o objetivo de detectar ciclos, portanto, a complexidade esperada é de $O(n^{(1/4)})$, menor que $\sqrt{n}$ devido a iterações específicas entre os valores e a estrutura da sequência gerada. Em cada iteração, a operação que possui complexidade significativa é o cálculo do mdc, $O(\log n)$. Portanto a complexidade final é:

$$O(n^{(1/4)} \times \log n)$$

- *factorize(ll n)*: A complexidade dessa função depende das demais funções que são chamadas em seu escopo e do seu número de recursões. As funções chamadas e suas complexidades são:

  - *millerRabin(ll pp)*: Como visto anteriormente, possui complexidade $O(k \times log^2 n)$.
  - *PollardRho(ll n)*: Como visto anteriormente, possui complexidade $O(n^{(1/4)} \times \log n)$.

  Supondo que o número de recursões seja j, a complexidade final é:

  $$O(j \times (log^2 n + (n^{(1/4)} \times \log n))) = O(j \times n^{(1/4)} \times \log n)$$

- *primeFactRho(ll n)*: A complexidade dessa função é determinada, principalmente, por três etapas:

  - Função *factorize(ll n)*: como visto anteriormente, possui complexidade $O(n^{(1/4)} \times \log n)$.
  - Ordenação e remoção de duplicatas: $O(k \log k)$, assumindo que k é o número de fatores encontrados.
  - Cálculo dos expoentes: $O(k \log n)$.

Assim, a complexidade final da função é

$$O(n^{(1/4)} \times \log n).$$

- *primeFact(ll n, long long lim)*: A complexidade aqui depende, principalmente de dois passos:
  - Looping principal: O laço for itera sobre i de 2 até $\sqrt{n}$ ou até lim $= 10^7$. A cada iteração o looping secundário while realiza divisões consecutivas, cujo número total de operações é proporcional ao logaritmo de n, portanto, $O(\log n)$.
  - Teste de Primalidade: É realizado uma vez no algorítimo e utiliza da função *millerRabin()* que como visto anteriormente possui complexidade $O(k \times log^2 n)$.

  Assim, a complexidade final é

  $$O(\sqrt{n}) + O(k \times log^2 n) = O(\sqrt{n})$$

- *Generator()*: Por fim, para calcular a complexidade total dessa função, vamos analisar duas partes principais:
  - Fatoração:
    - *primeFactRho(ll n)*: Como visto anteriormente, a complexidade é $O(n^{(1/4)} \times \log n)$.
  - Looping: Ele realiza k iterações, proporcionais ao tamanho de fact e em cada iteração a complexidade dominante é a da função *fexp()*, que como visto anteriormente é de $O(\log n)$. Portanto, a complexidade dessa parte é $O(k \times \log n)$.

Assim, a complexidade final dessa função é:

$$O(n^{(1/4)} \times \log n)$$

6.2.2 Espaço

A complexidade de espaço da função *Generator* é determinada pelo vetor ord que é inicializado com tamanho igual ao tamanho do vetor fact, supondo que fact tenha tamanho m, a complexidade de espaço para ord é $O(m)$ e o restante das variáveis tem espaço constante.

6.3 Módulo 3 - Encontrar o logarítmo discreto

O terceiro módulo é composto por quatro funções principais *DiscreteLogarithm()*, *discLogBrute(g, a, p)*, *discLogBabyGiantStep(g, a, p)* e *discLogPohligHellman(g, a, p)*.

6.3.1 Tempo

O terceiro módulo é composto por uma função principal chamada no main, *DiscreteLogarithm()*, ela chama três outras funções que calculam o logaritmo discreto, cada uma seguindo uma abordagem diferente, elas são: *discLogBrute(g, a, p)*, *discLogBabyGiantStep(g, a, p)* e *discLogPohligHellman(g, a, p)*. Para melhor compreensão da complexidade dos algoritmo, primeiro faremos uma análise de cada método utilizado para calcular o logaritmo discreto e depois concluiremos com a complexidade da função principal.

- discLogBrute(ll g, ll a, ll p): Esse algoritmo utiliza o método força bruta, o número de operações é proporcional a entrada `p`, $O(p)$, e em cada iteração a função *fexp()* é chamada, com complexidade, como analisada anteriormente, $O(\log n)$. Assim, a complexidade final é $O(p \times \log n)$.

- discLogBabyGiantStep(ll g, ll a, ll q, ll p): A complexidade desse algoritmo é determinada por dois loopings principais. No primeiro looping, o número de iterações é igual a $n - 1$, n, e em cada iteração a função *fexp* é chamada, $O(\log n)$ e ocorre uma inserção no map, $O(\log n)$, portanto, a complexidade final desse looping é $O(n \times (\log n + \log n))$. No segundo looping, o número de operações é igual a n, $O(n)$ e em cada looping:

  - A função *fexp* é chamada, $O(\log n)$.
  - Busca no map, $O(\log n)$.

  Nesse sentido, a complexidade desse looping é $O(n \times (\log n + \log n))$.

  Como n = $\sqrt{q}$, a complexidade da função é, portanto,

  $$O(\sqrt{q} \times \log \sqrt{q})$$

- discLogPohligHellman(ll g, ll a, ll p): Para o cálculo da complexidade dessa função, vamos analisar a complexidade de outras três funções utilizadas nela:

  - $mod_inv(llx, llm)$: A complexidade desse algoritmo é determinada pelo looping principal, que tem complexidade igual a $O(\log min(x, m))$.
  - $chineseremainder(vector < pair < ll, ll >> congruences)$: A complexidade dessa função é determinada por:

  - Primeiro looping: $O(n)$.
  - Segundo looping: O número de iterações é igual a n em cada iteração a função mod_inv é chamada, que como foi falado anteriormente, possui complexidade igual a $O(log \min(M_i, m_i))$, que no pior caso é igual a $O(\log M)$, portanto, a complexidade desse looping é $O(n \times \log M)$.

  Portanto, a complexidade final dessa função é:

  $$O(n) + O(n \times \log M) = O(n \times \log M)$$

- $congPair(llp, llq, lle, lle1, lle2)$: A complexidade dessa função é determinada por:
  - A função mod_inv possui complexidade $O(\log n)$, assim como visto anteriormente.
  - Looping Principal: Ele realiza, no máximo e iterações. E em cada uma delas, são realizadas duas chamadas da função fexp, que como visto anteriormente, possui complexidade igual a $O(\log n)$. Além disso, a função discLogBabyGiantStep é chamada e possui complexidade igual a $O(n \times \log n)$, como também visto antes. Assim, a complexidade desse looping é $O(e \times \log n) + O(e \times \log n) + O(e \times n \times \log n) = O(e \times n \times \log n)$.

  Portanto, a complexidade final dessa função é

  $$O(\log p) + O(e \times n \times \log^2 n) = O(e \times n \times \log n)$$

- $discLogPohligHellman(llg, llh, llp)$: A complexidade dessa função é determinada por:
  - Looping principal: Nele são realizadas k iterações, em que k é igual ao tamanho do vetor fact. Além disso, em cada iteração a função fexp é chamada duas vezes e a função congPair é chamada uma, ambas possuem complexidade, como visto anteriormente, iguais a respectivamente, $O(\log n)$ e $O(e \times n \times \log n)$. Assim, a complexidade desse looping é igual a $O(k \times e \times n \times \log n)$.
  - Chamada do método chineseremainder: Possui complexidade $O(k \times \log^2 k)$, como visto anteriormente.

  Assim, a complexidade final do algoritmo é:

  $$O(k \times \log^2 k) + O(k \times e \times n \times \log n) = O(k \times e \times n \times \log n)$$

Ao final, a função principal, *DiscreteLogarithm()*, vai escolher o método com base no número prime e isso determinará a complexidade do algoritmo.

6.3.2 Espaço

A complexidade de espaço desse módulo é dominada pelo espaço ocupado pela estrtura map na função *discLogBabyGiantStep(g, a, q, p)*, que é $O(\sqrt{q})$, onde q é o tamanho do intervalo de busca. As outras funções e variáveis locais contribuem com uma complexidade de espaço constante.

7 Listagem do programa fonte

```
#include "../include/functions.h"

template <
    class result_t   = chrono::milliseconds,
    class clock_t    = chrono::steady_clock,
    class duration_t = chrono::milliseconds
>
auto since(chrono::time_point<clock_t, duration_t> const&
   start){
    return chrono::duration_cast<result_t>(clock_t::now() -
       start);
}

// Constructor
Functions::Functions(ll _n, ll _a):
    prime(_n), a(_a), g(1) {}

// Fast exponentiation modulo m
ll Functions::fexp(ll x, ll b, ll m){
    ll ans = 1; x %= m;
    while(b > 0){
        if(b&1) ans = ans*x%m;
        x = x*x%m, b >>= 1;
    }
    return ans;
}

// Test if a number is composite
bool Functions::composite(ll n, ll r, ll d, ll s){
    ll x = fexp(r, d, n);
    if(x == 1 or x == n-1) return false;
    for(ll i = 1; i < s; i++){
        x = x*x%n;
        if(x == n-1) return false;
```

```
    }
    return true;
}

// Miller Rabin function that returns if a number is prime
bool Functions::millerRabin(ll pp){
    ll s = 0, d = pp-1;
    while(!(d&1)) d >>= 1, s++;

    bool found = true;
    for(auto x: primes){
        if(pp == x){ found = true; break; }
        if(composite(pp, x, d, s)) found = false;
    }
    return found;
}

// Returns the next prime after n
void Functions::NextPrime(){
    if(!(prime&1)) prime--;

    int cnt = 0;
    while(true){
        prime += 2, cnt++;
        if(millerRabin(prime)) break;
    }

    cout << "p: " << prime << ", iterations: " << cnt << endl
       ;
}

// Returns the factorization of n using trivial algorithm
void Functions::primeFact(ll n, long long lim){
    fact.clear(), exp.clear(), partial = 0;
    for (long long i = 2; i*i <= n and (lim == -1 or i < lim)
       ; i++) if(n%i == 0) {
        fact.emplace_back(i), exp.emplace_back(0);
        while(n%i == 0) n /= i, exp[fact.size()-1]++;
    }
    partial = (n != 1 and !millerRabin(n));
    if(n > 1) fact.emplace_back(n), exp.emplace_back(1);
}

// Return a factor of n
```

```cpp
ll Functions::PollardRho(ll n) {
    if(!(n&1)) return 2;
    auto s = chrono::high_resolution_clock::now();

    ll x = (rand()%n)+1, y = x, c = rand()%n+1;
    ll d = 1;
    while(d==1) {
        auto now = chrono::high_resolution_clock::now();
        if(chrono::duration_cast<chrono::seconds>(now-s).
           count()>10.0) return 0;

        x = (x*x+c+n)%n;
        y = (y*y+c+n)%n; y = (y*y+c+n)%n;
        d = x>=y? x-y : y-x;
        d = gcd(n,d);
    }
    return d;
}

// Factorize n and compute the facts array with all the
   divisors of n
void Functions::factorize(ll n){
    if(n == 1) return;

    if(millerRabin(n)){
        fact.emplace_back(n);
        return;
    }

    ll d = PollardRho(n);
    if(d==0) {
        partial = true;
        return;
    }

    factorize(d), factorize(n/d);
}

// Returns the factorization of n using pollard rho algorithm
void Functions::primeFactRho(ll n){
    fact.clear(), exp.clear(), partial = 0;
    factorize(n);
    sort(fact.begin(), fact.end());
    fact.erase(unique(fact.begin(), fact.end()), fact.end());
```

```cpp
    for(size_t i = 0; i < fact.size(); i++){
        exp.emplace_back(0);
        while(n%fact[i] == 0) exp[i]++, n /= fact[i];
    }
}

// Returns a number that is a possible generator for Zp and its order interval
void Functions::Generator(){
    ll phi = prime-1, n = phi;
    primeFactRho(n);

    vector<ll> ord(fact.size()); ll min_order = 1;
    for(size_t i = 0; i < fact.size(); i++){
        ll b = 2+rand()%(prime-2);
        while(fexp(b, phi/fact[i], prime) == 1) b = 2+rand()%(prime-2);
        g *= fexp(b, phi/(fexp(fact[i], exp[i], prime)), prime), g %= prime;
        if(!partial or i < fact.size()-1) min_order *= fexp(fact[i], exp[i], prime);
    }

    cout << "g: " << g << ", minimum order: " << min_order << endl;
}

// Brute the discrete logarithm testing if g^x % p == a
ll Functions::discLogBrute(ll g, ll h, ll p){
    h %= p; ll f = 1;
    for(int x = 0; x < p; x++){
        if(f == h) return x;
        f = f*g%p;
    }
    return -1; // Discrete log not found
}

ll Functions::discLogBabyGiantStep(ll g, ll h, ll q, ll p) {
    auto s = chrono::high_resolution_clock::now();

    h %= p;
    ll n = sqrt(q) + 1;
    map<ll, ll> vals;
    for (ll i = 1; i <= n; i++) {
```

```
        auto now = chrono::high_resolution_clock::now();
        if(chrono::duration_cast<chrono::seconds>(now-s).
           count()>20) throw length_error("");
        vals[fexp(g, i * n, p)] = i;
    }
    for (ll j = 0; j <= n; j++) {
        ll cur = (fexp(g, j, p) * h) % p;
        if (vals.count(cur))
            return vals[cur] * n - j;
        auto now = chrono::high_resolution_clock::now();
        if(chrono::duration_cast<chrono::seconds>(now-s).
           count()>20) throw length_error("");
    }
    return -1;
}

// Returns modular inverse of a modulo m, i.e., mod_inv * a =
    1 mod m
ll Functions::mod_inv(ll x, ll m) {
    ll m0 = m, t, q;
    ll x0 = 0, x1 = 1;
    if (m == 1) return 0;

    while (x > 1) {
        q = x / m;
        t = m;
        m = x % m;
        x = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0) x1 += m0;

    return x1;
}

// Returns the solution for a system of congruences x = a_i %
    m_i
ll Functions::chinese_remainder(vector<pair<ll, ll>>
   congruences){
    ll M = 1;
    for(auto const& c : congruences) M *= c.second;
```

```cpp

    ll solution = 0;
    for(auto [a_i, m_i] : congruences)
        solution += a_i * (M/m_i) * mod_inv(M/m_i, m_i),
           solution %= M;

    return solution;
}

pair<ll, ll> Functions::congPair(ll p, ll q, ll e, ll e1, ll
   e2) {
    ll inv = mod_inv(e1, p);
    ll x = 0;
    ll q_pow_i = 1;
    for (int i = 0; i < e; i++) {
        ll b = (e2 * fexp(inv, x, p)) % p;
        ll c = fexp(e1, q_pow_i, p);
        ll dlog = discLogBabyGiantStep(c, b, fexp(q, e, p), p
           );
        x = (x + dlog * q_pow_i) % (p - 1);
        q_pow_i = (q_pow_i * q) % (p - 1);
    }
    return {x, fexp(q, e, p)};
}

ll Functions::discLogPohligHellman(ll g, ll h, ll p) {
    g %= p, h %= p;

    ll phi = p - 1;
    if(partial) throw invalid_argument("");
    vector<pair<ll, ll>> cong;

    for (size_t i = 0; i < fact.size(); i++) {
        ll q = fact[i];
        ll e = exp[i];
        ll e1 = fexp(g, phi / (fexp(q, e, p)), p);
        ll e2 = fexp(h, phi / fexp(q, e, p), p);
        cong.emplace_back(congPair(p, q, e, e1, e2));
    }

    return chinese_remainder(cong);
}

// Returns the discrete logarithm of a modulo p in base g
```

```
void Functions::DiscreteLogarithm(){
    start = chrono::high_resolution_clock::now();

    ll ans = -1;
    bool b = false;
    try {
        if(prime < (ll)1e6) ans = discLogBrute(g, a, prime);
        else if(prime < (ll)1e12) ans = discLogBabyGiantStep(
           g, a, prime, prime);
        else ans = discLogPohligHellman(g, a, prime);
        b = true;
    }
    catch (length_error &e) {cout << "i: time limit\n";}
    catch (invalid_argument &e) {cout << "i: time limit\n";}

    if(b) cout << "i: " << ans << endl;
    else return;

    auto now = chrono::high_resolution_clock::now();
    cout << "t: " << chrono::duration_cast<chrono::seconds>(
       now-start).count() << "s" << std::endl;
}
}
```