TP 1 - Álgebra A

Gabriel Campos Prudente
Leonardo Romano
Vitor Costa

Universidade Federal De Minas Gerais

1. Descrição do Trabalho:

Este trabalho tem como objetivo entender e aplicar os algoritmos de:

a) Encontrar números com grandes probabilidades de ser primos - Aqui usamos o algoritmo de Miller-Rabin.
b) Encontrar um elemento gerador, também chamado de raíz primitiva, do grupo multiplicativo $\mathbb{Z}_p$ ou um elemento de ordem alta, isto é, um número cuja a ordem é um divisor grande do primo p, menos 1, isto é, $p - 1$, também conhecido como, $\phi(p)$.
c) Encontrar o logaritmo discreto do gerador *g*, encontrado na fase anterior. Para essa etapa, também será importante saber a base *a* do logaritmo discreto.

Esse trabalho prático fez uso da biblioteca GMP - GNU Multiple Precision Library - para lidar com números grandes, que estouram a precisão da máquina, gerando overflow. Além disso, os algoritmos foram divididos em diversas funções, cada uma com um propósito específico.

O trabalho essencialmente possui três divisões principais, cada uma implementando com os algoritmos supracitados acima. A descrição mais objetiva deles será dada na seção 2.

2. Descrição dos Módulos:

Antes de detalhar os algoritmos, é necessário ressaltar que todos eles fazem uso da biblioteca GMP, usando seus tipos primitivos e,além disso, implementamos funções base, como a *expMod*, que calcula o módulo da potência e o Teorema do Resto Chines.

O trabalho foi dividido em 3 partes, para lidar com as especificações propostas:

1) Módulo 1 - Testes de Miller Rabin

Para a base desse algoritmo, usamos o teste de Miller-Rabin, utilizando os 40 primeiros primos. O algoritmo recebe o valor n a ser testado a primalidade e faz uso do valor $n_{minus1} = n - 1$, e, então, testa, de modo geral, as congruências módulo N para cada um desses 40 primeiros primos. Para cada primo, a probabilidade do valor testado n ser falso positivo é igual a $1/4$. Logo, ao final dos 40 testes, a probabilidade de n ser primo será $1 - (1/2^{80})$ - vide Teorema visto em aula. Para cada primo testado, é possível o Teste de Miller-Rabin retornar Verdadeiro ou Falso. Se retornar Falso, então o valor n é composto e o algoritmo encerra. Se, ao final de todos os testes internos do Teste de Miller-Rabin, nenhum retornar falso, então o algoritmo retorna Verdadeiro, indicando que existe chance de ser possível primo.

Este módulo foi dividido em duas funções: "*isPrimeMRTest*" e "*millerRabinTest*". A primeira, inicializa os 40 primeiros primos e para cada um deles, chamada a função *millerRabinTest,* utilizando o algoritmo base de Miller-Rabin.

2) **Módulo 2 - Find Primitive Root - Encontrar o elemento gerador**

O algoritmo usado foi o *findPrimitiveRoot*. Basicamente, ele fatora o número em primos, usando o Teorema Fundamental da Aritmética - TFA - e os armazena em um vetor. Então, para verificar se um valor gerador é realmente um gerador primitivo, o algoritmo faz:

Para cada fator q de $p - 1$, ele calcula a exponenciação modular:

$$result = gerador^{p-1/q} \, mod \, p$$

Se o resultado da exponenciação for 1 para qualquer fator q, então o valor gerador não é um gerador primitivo, e o algoritmo continua tentando o próximo valor.Se o resultado nunca for 1 para todos os fatores, então o gerador testado é um gerador primitivo.

Esse módulo possui duas funções: "*factorize"* e "*findPrimitiveRoot*". A primeira fatora o primo em fatores primos menores, usando o TFA, e a segunda encontra a raíz primitiva do valor. A parte de fatorar o número pode ser problemática, portanto, para ser viável computacionalmente, limitamos o número de iteração para 10^7. Logo se a fatoração do primo p tiver mais que 10^7 fatores primos, então a fatoração encerra, e o problema torna-se encontrar um estimador de gerador, que é um número cuja a ordem é um divisor grande de $p - 1$. Com isso usamos a função *"findEstimativeGenerator"* que acha uma estimativa para um elemento de ordem alta. Para isso, a função calcula um gerador para cada primo e, então, multiplica todos esses geradores para ter um elemento de ordem alta. Além disso, ela retorna a ordem desse elemento, que é a multiplicação de cada expoente dos primos da fatoração parcial.

3) **Módulo 3 - Polling Helman, Baby-Step Giant-Step e Força Bruta - Encontrar o logaritmo discreto**

3.1 Força bruta

O algoritmo de força bruta é o mais simples e inocente dos algoritmos para se resolver o problema do logaritmo discreto. Testamos *x = 0, 1, 2, 3, …. n* até encontrarmos o valor de *x* que satisfaça $g^x = a \, mod \, p$. A complexidade, portanto, é exponencial com relação ao tamanho da entrada. Portanto, para valores de input muito grandes, acaba se provando um algoritmo muito ineficiente. Precisaríamos de outras abordagens para atacar o problema do logaritmo discreto.

3.2 Baby-step/Giant-Step

O algoritmo Baby-step/Giant-step calcula logaritmos discretos em um grupo de ordem q no tempo O($\sqrt{q}$). A ideia é simples: dados como entrada ***g*** e ***a*** ∈ <g>, podemos imaginar os elementos de <g> dispostos em um círculo como:

$$g^0 = 1, g^1, g^2, \ldots, g^{q-1}, g^q = 1$$

Sabemos que ***a*** deve estar em algum lugar deste círculo. Computar e elencar todos os pontos deste círculo levaria um tempo de pelo menos Ω(q). Em vez disso, marcamos o

círculo em intervalos de tamanho $t = floor(\sqrt{q})$. Ou seja, calculamos e registramos os $floor(q/t) + 1 = O(\sqrt{q})$ elementos.

3.3 Pohlig-Hellman:
Este é um algoritmo específico para resolver logaritmos discretos em grupos cíclicos de ordem prima. Ele fatora a ordem do grupo em fatores primos e resolve o logaritmo discreto para cada fator. Desse modo, o algoritmo calcula modúlos do primo h da fatoração e do gerador g.
$g_i := g\hat{}(n/p_i{}^{e}{}_i)$.
$h_i := h\hat{}(n/p_i{}^{e}{}_i)$

Após isso ele calcula x_i para esses valores e então, combina essas soluções usando o Teorema do Resto Chinês. Este algoritmo é eficiente para grupos de ordem prima, mas pode ser impraticável para grupos de ordem não prima, pois requer a fatoração da ordem do grupo. Isto é, para primos com fatoração pequena, o algoritmo lida bem. Porém, quanto maior a fatoração e os primos dessa fatoração, pior é o desempenho desse algoritmo.

Ao implementar progressivamente cada módulo, percebemos que, embora cada um seja independente entre si, realizando suas verificações sem depender dos outros, existe uma direta relação entre as tarefas propostas, em que a saída de um módulo é a entrada de outro. Um exemplo disso é o módulo 2 e 3, em que encontramos um gerador de um número primo (que por sua vez é gerado no módulo 1) para, assim, encontrar o logaritmo discreto. Portanto, identificamos uma interdependência de cada parte do projeto e uma relação direta entre as sucessivas tarefas.

3. Descrição do formato de entrada:

O programa utiliza a biblioteca “GNU - Multiple Precision Library” que possibilita a operação com números grandes de precisão aritmética arbitrária. A biblioteca possui o tipo “*mpz_class*” que contempla a precisão necessária para a realização das operações requisitadas.
O tipo básico *mpz_class* da biblioteca utilizada tem as seguintes características:

- ➔ Suporta uma ampla gama de operações aritméticas, incluindo adição, subtração, multiplicação, divisão, exponenciação, e operações de módulo, entre outras. Estas operações são realizadas de forma eficiente, mesmo para números extremamente grandes.
- ➔ Pode ser facilmente convertida para e a partir de outros tipos de dados numéricos, como inteiros, strings e arrays de bytes, facilitando a integração com outras partes de um programa.

A entrada padrão é:

a) um valor *mpz_class n*, que é o valor a partir do qual acharemos um valor primo maior.
b) um valor *mpz_class a*, que será a base do logaritmo discreto calculado no módulo 3.

4. Descrição do formato de saída:

Assim como a entrada, o formato de saída do programa é do tipo "*mpz_class*", uma vez que as operações do código são feitas com esse mesmo tipo e, também, do tipo "*int*" para saídas que não precisam de uma grande precisão.
A saída é, então:

1) um valor *mpz_class p*, que representa o próximo primo maior que "N".
2) um valor *int* que representa o número de vezes que o teste de Miller-Rabin foi executado para achar o menor primo maior que "N"
3) um valor *mpz_class g*, que representa o gerador "g" (ou uma estimativa de g)
4) um valor *mpz class discrete_log*, que representa o logaritmo discreto de 'a' módulo p na base g. Nem sempre esse valor é possível de ser calculado
5) caso o programa não consiga calcular o logaritmo discreto, a saída será uma string dizendo que o limite de tempo de cálculo foi excedido e, portanto, o algoritmo utilizado não foi capaz de calcular o logaritmo discreto do número fornecido.

5. Explicação sobre a utilização do programa:

Para utilizar o programa, é necessário instalar a biblioteca GMP. A biblioteca está no arquivo compactado e, para utilizá-la ,é preciso abrir um terminal Linux (ou WSL para windows) no diretório da pasta com a biblioteca.
Uma vez com o diretório aberto em um terminal Linux, deve-se executar o comando "**./configure**" para instalar as dependências da biblioteca e, depois, executar o comando "**make**" para executar os arquivos. O terminal ficará alguns minutos instalando os arquivos. Caso isso não ocorra, ou der algum tipo de erro, é necessário executar o seguinte comando: **"chmod +x configure",** e então executar o "**./configure**" e o "**make**". Após tudo instalado, deve-se executar um último comando "**make init**" para finalizar a importação.

Finalmente, para utilizar o programa, deve-se criar um executável a partir da compilação normal seguindo os seguintes comandos:

g++ -o TP1 TP1.cpp -lgmpxx -lgmp
(para compilar utilizando a biblioteca)

./TP1.exe
(para executar o arquivo)

6. Estudo da complexidade do programa:

A complexidade do programa pode ser dividida para cada algoritmo implementado.

1) Temos que a complexidade do algoritmo de Miller-Rabin é de:
$O(klog^3 n)$ onde n é o primo - 1 e k é o número de vezes que o teste é repetido.

2) Temos que a complexidade para achar a raiz primitiva de um primo é de:

$O(Ans\ log\phi(n)\ log\ n)$ onde Ans é o número de possíveis geradores candidatos e n é o primo - 1.

3) Temos que a complexidade do algoritmo “Baby Step Giant Step” é de $O(\sqrt{n})$ onde n é o primo - 1.

Assim, assumindo k e Ans constantes, temos que a complexidade total do programa é de: $O(\sqrt{n})$.

7. Listagem do programa fonte:

O programa foi escrito em c++ e o código fonte se encontra na pasta compactada com o nome TP1.cpp. Todos algoritmos citados e explicados acima estão implementados no mesmo arquivo.

8. Bibliografia:

GNU - Multiple Precision Library [https://gmplib.org/]
Primitive Root - Algoritmo [https://cp-algorithms.com/algebra/primitive-root.html]