O uso de um algoritmo aproximativo para a fatoração de primos

João Guilherme Lopes Alves da Costa, Isaque Barbosa Martins

Universidade Federal do Rio Grande do Norte - UFRN / Departamento de Informática e Matemática Aplicada - DIMAp

Resumo

Os sistemas criptográficos atuais são todos baseados em problemas matemáticos difíceis de se resolver, pode-se citar o uso de logaritmos discretos, curvas elípticas e problemas com reticulados. No decorrer deste artigo apresentaremos outro problema também usado na base de algoritmos criptográficos, o problema da fatoração, dessa forma analisaremos o funcionamento deste problema aplicado em um algoritmo de criptografia, algoritmo RSA, trataremos sobre métodos utilizados para resolver o problema das formas mais eficientes já demonstradas, e falaremos sobre heurísticas e algoritmos aproximativos para o problema, utilizando de uma implementação de um algoritmo aproximativo para a resolução do problema, com aplicações e resultados do mesmo.

Palavras-chave: criptografia; fatoração de inteiros; problema da fatoração; RSA

1. Introdução

Na área da criptografia, sistemas criptográficos são quaisquer sistemas que, dada uma mensagem e uma chave, consiga gerar uma nova mensagem ilegível que, consequentemente, trará mais segurança ao conteúdo da mensagem. Dentre estes sistemas estão os sistemas de criptografia assimétrica, ou criptografia de chave pública, onde o usuário possui duas chaves, uma pública e uma privada, e às utiliza separadamente para encriptar e decriptar a mensagem. Para a criação das chaves é utilizado problemas matemáticos que não admitem solução eficiente, tais quais logaritmos discretos, aplicado no criptossistema de ElGamal e no algoritmo DSA, curvas elípticas, utilizado no protocolo Diffie-Hellman de curva elíptica (ECDH) e o problema da fatoração, utilizado

no algoritmo RSA.

O problema da fatoração tem grande importância na criptografia, pois a decomposição de um número n tal que $n = pq$ onde p e q são números primos tão grandes, é computacionalmente inviável para os algoritmos e computadores atuais, entretanto a busca por algoritmos mais eficazes para o problema vêm sendo realizada desde a criação do algoritmo RSA.

Neste artigo temos como objetivo mostrar o uso de um algoritmo aproximativo para o problema da fatoração de números primos utilizando a aproximação a partir da operação de raiz quadrada.

2. Descrição do problema

O teorema fundamental da aritmética descreve que para todo inteiro $a > 1$, a pode ser fatorado unicamente na forma:

$$a = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_n^{a_n}$$

Em específico, existe inteiros $a > 1$ tal que $a = pq$, onde p e q são números primos. Este fato é importante, pois a multiplicação de dois números primos é utilizada em algoritmos para criação de chaves privadas em sistemas criptográficos de chaves públicas. Por exemplo, para o algoritmo RSA podemos aplicá-la na função totiente de Euler, onde $\Phi(a) = (p-1)(q-1)$, este valor será usado para gerar a chave-privada d a partir do cálculo $de \equiv 1 (\mathrm{mod}\ \Phi(a))$, onde e é um primo relativo de $\Phi(a)$ e menor que ele. A partir da chave d obtida do cálculo, podemos encriptar uma mensagem M a partir do cálculo $C = M^e \bmod a$, onde C será a mensagem encriptada, e decriptar a partir do calculo $M = C^d \bmod a$. [1]

A fatoração dos dois valores primos p e q também tem grande importância, e possibilita a quebra do algoritmo RSA. Se os valores de p e q são conhecidos, então é possível calcular $\Phi(a)$ que por sua vez pode ser utilizado para calcular d e encontrar M. [1]

Dessa forma, vem sendo feito diversos estudos para criar algoritmos otimizados para encontrar a fatoração de um inteiro n. A seguir descreveremos alguns dos métodos exatos já criados para a otimização do algoritmo. [2]

2.1. Método de divisão por tentativa

O método de divisão por tentativa (*trial division*) é o método mais trivial de se descobrir os fatores de um inteiro arbitrário n. O método se baseia no teorema que demonstra que se n for um inteiro composto, então n tem um divisor primo menor ou igual que $\sqrt{n}$ [3]. Dessa forma, verifica-se para cada número primo p menor que $\sqrt{n}$ se n é divisível por p, caso seja, substitui n pelo resultado de $\frac{n}{p}$ e volta-se a verificar a divisibilidade do novo valor de n pelos primos p. Este método é efetivo para números inteiros não muito grandes, mas dado que para sistemas criptográficos são usados números de centenas de bits, o algoritmo perde sua efetividade.

2.2. Algoritmo Rho de Pollard

O algoritmo de Rho [4] tem como objetivo evitar ciclos, então temos que, para fatorar um inteiro n,assumindo que $n = pq$ onde p e q são dois fatores primos de n, o método começa com a iteração de uma fórmula polinomial, como $x_{n+1} = x_n^2 + a(mod\ n)$. Como p e q são primos relativos, o teorema do resto chinês garante que cada valor de $x(mod\ n)$ corresponde exclusivamente ao par de valores $(x(mod\ p), x(mod\ q))$. Portanto, a sequência de x_n segue exatamente a mesma fórmula módulo p e q:

$$x_{n+1} = [x_{n+1}(mod\ p)]^2 + a(mod\ n)$$

$$x_{n+1} = [x_{n+1}(mod\ q)]^2 + a(mod\ n)$$

A sequência $(x(mod p))$ cairá em um ciclo muito mais curto de comprimento $\sqrt{p}$. É possível verificar que dois valores x_1 e x_2 têm o mesmo resto com $p(x_1(mod\ p) = x_2(mod\ p))$ se MDC$(|x_2 - x_1|, n) = p$. O mesmo acontece para q. [2]

2.3. Método da curva elíptica

O método da curva elíptica foi proposta por Lenstra [5] e é um método com um propósito especial, dado que ele é mais eficiente para encontrar pequenos fatores de um inteiro, atualmente sendo o melhor algoritmo para divisores que não excedem 60 digítos. Para fatorar um inteiro n escolhe-se uma curva elíptica randômica E em $(\mathbb{Z}/n\mathbb{Z})$ com uma equação da forma $y^2 = x^3 + ax + b(mod\ n)$ juntamente com um ponto $P(x + 0, y_0)$ e computa-se $Q = k \times P$, onde k é o produto de vários primos pequenos

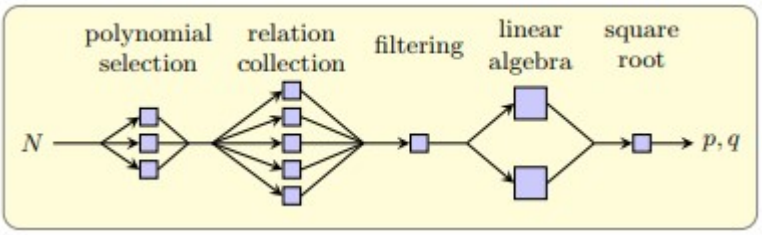

Figura 1: Etapas do algoritmo GNFS.[7]

elevados a um expoente pequeno, onde o produto é menor que um teto B_1. Seja p um divisor primo de n, se a ordem de E em $E(\mathbb{Z}/n\mathbb{Z})$ dividir k, então Q é um elemento neutro na curva eliptíca, logo sua coordenada z será 0 módulo p, a partir do MDC(z,n) teremos o fator p, analogamente, pode-se fazer o mesmo com o segundo fator de n.

Diferentemente de todos os outros algoritmos conhecidos de tempo subexponencial, o tempo de execução desse algoritmo é sensível aos tamanhos dos fatores de n. Em particular, se p for o menor primo que divide n, o algoritmo encontrará p no tempo esperado [6]

$$\exp\left((c+o(1))(logn)^{1/3}(log/logn)^{2/3}\right)$$

2.4. Algoritmo GNFS

O algoritmo GNFS (General Number Field Sieve) é atualmente o algoritmo mais eficiente para o problema da fatoração, ele parte da seguinte ideia geral, dado $N = pq$ um número composto, o módulo de N possui até 4 raizes quadras, por exemplo 2701 possui $(\pm 119)^2 = (\pm 1287)^2 = 656 \bmod N$. Um par adequado dessas raízes quadradas pode ser usado para fatorar N: temos $1192 - 12872 = (119 + 1287)(119 - 1287) = 0 \bmod N$. Possivelmente os fatores dessa diferença de quadrados dividem os fatores de N, e podemos usar o algoritmo MDC para encontrar esses fatores. Em nosso exemplo, temos $MDC(119 + 1287, N) = 37$ e $MDC(119 - 1287, N) = 73$ [7], como $37 * 73 = 2701$, foi possível encontrar os fatores de 2701. Esta ideia parte da fatoração de Fermat, entretanto para otimizar essa resolução, o GNFS é dividido em 5 etapas mostradas na figura 1.

Este algoritmo tem complexidade assintótica. [7]

$$\exp\left((64/9)^{1/3}(log\ N)^{1/3}(log\ log\ N)^{2/3}(1+o(1))\right)$$

3. Complexidade do Problema

Não há algoritmos que resolvam o problema da fatoração de primos em tempo polinomial, de forma a fatorar um número n de bits b no tempo $O(b^k)$ para alguma constante k. Não foi provado que estes algoritmos existem ou não existem, mas se suspeita que este algoritmo não está na classe P, e está na classe P e co-NP , de forma que as respostas ”sim”e ”não”podem ser verificadas em tempo polinomial [8]. Além disso, a classe está na classe UP e co-UP de acordo com o teorema fundamental da aritmética e [9].

Seja b o comprimento da entrada de valor n. Considerando $b = \log_2 n$, para calcular a complexidade $O(n)$ teremos $O(2^k)$, de forma que a complexidade é exponencial de acordo com o tamanho da entrada. Mesmo que fosse realizado o cálculo até os fatores da raiz quadrada de n, teríamos $O\left(\sqrt{2^k}\right) = O\left(2^{\frac{2}{k}}\right)$, ainda sendo exponencial.

4. Heurísticas e Algoritmos Aproximativos para o Problema

Não há muitas heurísticas construídas para a resolução do problema da fatoração, entretanto abaixo citaremos algumas das encontradas.

4.1. Heurística de Shoup

SHOUP [6] demonstra, a partir do teste de Miller-Rabin [10][11], que o problema da fatoração e o problema para computar $\Phi(n)$, a função totiente de Euler, são equivalentes, ou seja, dado um algoritmo eficiente para um problema, ele também será eficiente para o outro, e vice-versa. No decorrer de sua demonstração, Shoup propõe um algoritmo para a fatoração de um inteiro n da seguinte maneira: Se n for 1, o algoritmo para, senão verifica se n é primo, se for primo também para. Entretanto, se n não for primo, divide-se n como $n = d_1 d_2$, e recursivamente fatora d_1 e d_2. Shoup então demonstra que, se $n > 1$, n não for primo e n for par, ou se n for uma potência perfeita,

certamente é possível encontrar sua fatoração, todavia, se $n > 1$ primo for ímpar e não for uma potência perfeita, é apresentado um algoritmo probabilístico, no qual possui probabilidade pelo menos de $\frac{1}{2}$ de encontrar a fatoração de n.

4.2. Heurística de Shor

SHOR [12] apresenta uma heurística para a computação quântica, na qual é possível realizar o problema da fatoração em tempo polinomial $O((log\ n)^2(log\ log\ n)(log\ log\ log\ n))$ Shor, utiliza do fato de que utilizando randomização, a fatoração pode se reduzir à encontrar a ordem de um elemento [10], dessa forma, para encontrar um fator de um número ímpar n, dado um método para computar a ordem r de $x(mod\ n)$,é necessário computar o MDC$(x^{r/2}-1,n)$, isto se da pois $(x^{r/2}+1)(x^{r/2}-1) = x^r - 1 \equiv 0(mod\ n)$. Então o MDC$(x^{r/2}-1,n)$ só falha de encontrar um divisor de n somente se r for ímpar, ou se $x^{r/2}-1 \equiv -1(mod\ n)$, é demonstrado então que ao utilizar esse procedimento, quando aplicado a um $x(mod\ n)$ randômico, encontra um fator de n com uma probabilidade de, pelo menos, $1-1/2^{k-1}$, onde k é o número de fatores primos distintos de n. O método que Shor utiliza para computar a ordem r de $x(mod\ n)$ é baseado em computação quântica, e desta forma ficará fora deste escopo.

4.3. Heurística baseada em redes neurais

JANSEN e NAKAYAMA [13] propõem uma resolução do problema da fatoração a partir do uso de redes neurais nas quais as entradas e saídas de dados se dão em base binária, sendo utilizado o algoritmo Resilient Back-Propagation para o treinamento da rede neural, os autores fizeram testes para inteiros N separadamente quando $N < 1000$, $N < 10000$, $N < 10000$ e $N < 1000000$, e fazem a verificação de acerto do algoritmo a partir da quantidade de bits errados no valor de saída.

4.4. Heurística baseada em otimização da geometria molecular

Por fim, MISHRA, CHATURVEDI e SHUKLA [14] expõem duas formas de resolver o problema da fatoração a partir do algoritmo Molecular Geometry Optimization (MGO), no qual se inspira na química computacional sobre o arrengamento de grupo de átomos no espaço de forma que as forças da rede inter-atômica entre os átomos sejam

minimizadas. Dessa forma, utilizando esta minimização, estabelecem as formas um, na qual minimiza a função $f(x,y) = x^2 - y^2 (mod\ N)$ e $x,y \in [2,(N-1)/2]$, e a forma 2, onde minimiza a função $f(x) = N(mod\ x)$ e $x \in [2,\sqrt{N}]$. Na primeira forma teremos vários pares x,y que satisfazem a função $f(x)$ e pode-se determinar um desses pares que satisfazem o problema. Já na segunda forma, como a função possui o parâmetro x, é necessário verificar o valor que será o fator do N. A partir de testes realizados pela equipe, foi possível verificar que a utilização do algoritmo MGO pode satisfazer o problema para valores baixos, mas a medida que os valores dos casos de teste aumentam em quantidade de bits, a taxa de sucesso em encontrar seus fatores diminui.

5. Implementação

Inicialmente, foi implementado o algoritmo apresentado no artigo *Prime factorization using square root approximation* disponível em [2]. No entanto, foi notado que em diversos momentos, é suprimido do artigo detalhes de implementação de certos passos. Com isso, um dos exemplos citados no artigo não foi possível calcular os fatores primos, e foi feito a tentativa de outra implementação utilizando *Trial Division*.

O algoritmo baseado no *Trial Division* realiza basicamente três passos:

Algoritmo 1: Algoritmo baseado no *Trial Division*

Entrada: Inteiro n a ser fatorado.

Saída: Vetor com 2 fatores de n.

1 Pega todos os números primos até $\sqrt{n}$;

2 Faz divisão sucessivas de n utilizando os números primos, gerando os fatores de n;

3 A partir dos fatores, tenta agrupar 2 números que a multiplicação é igual a n;

5.1. Gerador de Instâncias

Não foi possível encontrar um gerador de testes automáticos. Diante disso, foi tentado utilizar uma forma de gerar números aleatórios dentro um intervalo de valores com métodos do próprio C++. Para isso foi utilizado o mnt19937 para gerar uma semente pseudoaleatória de acordo com o tempo do relógio e distribuir os números do

intervalo em um distribuição uniforme. Esse intervalo de valores será passado pelo usuário. O código pode ser visualizado abaixo:

```
int randint(int a, int b) {
    std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
    return std::uniform_int_distribution<int>(a, b)(rng);
}
```

5.2. Benchmarking

Para fazer análise empírica do algoritmo, foi utilizado o método std::chrono para obter o tempo no início da execução do algoritmo e o tempo no final. Após isso, foi feito a diferença de tempo, como pode ser observado no código:

```
auto start = std::chrono::steady_clock::now();
vector<size_t> factors = factorization(n, subset_primes, 15100, Highstep);
auto end = std::chrono::steady_clock::now();
auto time = end - start;
```

5.3. Resultados

Foi notado alguns problemas no artigo *Prime factorization using square root approximation* na explicação de passos do algoritmo. Por exemplo, no exemplo 4.1 do artigo é utilizado uma multiplicação por 10 sem explicar o porquê de fazer aquilo, além de considerar $p = 10$ e $q = 81$ e ao realizar o cálculo de A inverter os valores. Ou seja, o maior problema na implementação do algoritmo foi no passo "*find a possible multiplier r of fx which generates its nearest integer*", em que não é explicado de forma exata em como realizar essa busca.

É importante notar que o artigo diz que o r pode ser fatorado utilizando qualquer método de fatoração de primos, e nesse caso, foi utilizado também o *Trial Division*. No primeiro exemplo fornecido pelo artigo na seção 3 (51.923), é possível encontrar os fatores, no entanto, o algoritmo se saiu pior do que o outro algoritmo implementado utilizando somente divisões sucessivas de números primos. Além disso, para alguns valores pequenos, não foi possível gerar os fatores utilizando esse algoritmo. Para o

exemplo 4.1 do artigo (14.789.166.241), somente o algoritmo da *Trial Division* foi possível gerar os fatores.

O método de aproximação por raiz quadrada é uma proposta melhor para fatoração de números grandes, no entanto, dada problemas de implementação em alguns ele não consegue encontrar os fatores, mas é possível perceber que possui um desempenho melhor nesses casos.

É possível notar um resultado aproximado do benchmarking, note que "Algoritmo 1"é o algoritmo baseado no artigo "*Prime factorization using square root approximation*"e "Algoritmo 2"é o algoritmo implementado baseado em *Trial Division*:

Número	Fatores Algoritmo 1	Tempo Algoritmo 1	Fatores Algoritmo 2	Tempo Algoritmo 2
15	5 e 3	0.041337ms	3 e 5	0.003647ms
40.000	100 e 400	0.004368ms	2 e 20.000	0.011932ms
51.923	379 e 137	0.041708ms	137 e 379	0.008786ms
55.872	9 e 6208	0.210324ms	2 e 27936	0.01048ms
111.000	500 e 222	0.006192ms	2 e 55.500	0.012533ms
450.000	3 e 150.0000	0.154289ms	2 e 225.000	0.022142ms
37.527.851	N/A	1.65729ms	6121 e 6131	0.361537ms
260.100.000	2 e 130.050.000	1.16428ms	2 e 130.050.000	1.18754ms
542.448.900	81 e 6.696.900	0.438151ms	2 e 271224450	1.86303ms
2.147.483.648	N/A	0.553065ms	2 e 1073741824	4.91744ms
10.000.000.000	50000 e 200000	0.057468ms	2 e 5000000000	14.6602ms
14.789.166.241	N/A	1.10207ms	15031 e 983911	18.205ms
10^{12}	$5 \cdot 10^5$ e $2 \cdot 10^6$	0.37923ms	2 e $5 \cdot 10^{11}$	351.941ms
10^{14}	$5 \cdot 10^6$ e $2 \cdot 10^7$	2.89217ms	2 e $5 \cdot 10^{13}$	9369.91ms

Tabela 1: Resultados da execução dos algoritmos de fatoração de primos

É possível notar que o algoritmo 1 realmente atende a expectativa de que ter melhor desempenho para números grandes, com o aumento no tamanho da instância o algoritmo tem um desempenho melhor. Aqueles que estão em 'N/A' não conseguiram

gerar os fatores.

5.4. Análise

No primeiro algoritmo é dado um explicação melhor na matemática por trás do algoritmo em [2]. Sobre o segundo algoritmo, temos que a função de obter os números primos é $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$. O *trial division* também acaba sendo $O(n)$, além do último *loop*, o problema se dá porque o algoritmo é pseudopolinomial, já que seu desempenho é exponencial de acordo com o comprimento da entrada 3.

5.5. Especificações do Sistema

Sistema operacional: Windows 11 no WSL (Windows Subsistem for Linux) com Ubuntu 22.04 LTS
Processador: AMD Ryzen 3 3100 4-Core Processor 3.60 GHz
Memória RAM: 16GB
Linguagem utilizada: C++

6. Conclusão

Neste artigo foi tentado explorar um exemplo de implementação de algoritmo aproximativo para fatoração de números primos utilizando aproximação da raiz quadrada. Mesmo com a falta de clareza dada pelos autores em alguns passos do algoritmo, ainda assim foi possível observar que o algoritmo 1 possui uma melhor eficiência quando há um aumento no tamanho da instância. Portanto, neste artigo foi tentado construir uma solução construtiva (gulosa) utilizando o método do *Trial Division*, e foi feito uma comparação entre as 2 implementações.

Referências

[1] W. Stallings, Cryptography and Network Security: Principles and Practice, International Edition: Principles and Practice, Pearson Education Limited, 2014.
URL `https://books.google.com.br/books?id=q_6pBwAAQBAJ`

[2] J. Zalaket, J. Hajj-Boutros, Prime factorization using square root approximation, Computers & Mathematics with Applications 61 (9) (2011) 2463–2467.

[3] K. H. Rosen, Discrete mathematics and its applications, The McGraw Hill Companies,, 2007.

[4] J. M. Pollard, A monte carlo method for factorization, bit 15 (1975), 331–334.

[5] H. W. Lenstra Jr, Factoring integers with elliptic curves, Annals of mathematics (1987) 649–673.

[6] V. Shoup, A computational introduction to number theory and algebra, Cambridge university press, 2009.

[7] F. Boudot, P. Gaudry, A. Guillevic, N. Heninger, E. Thomé, P. Zimmermann, The state of the art in integer factoring and breaking public-key cryptography, IEEE Security & Privacy 20 (2) (2022) 80–86.

[8] S. G. Krantz, A prova está no pudim: a natureza mutável da prova matemática, Nova Iorque: Springer, p. 203, 2011.

[9] L. Fortnow, Blogue de complexidade computacional: aula de complexidade da semana: fatoração.
URL `https://blog.computationalcomplexity.org/2002/09/complexity-class-of-week-factoring.html`

[10] G. L. Miller, Riemann's hypothesis and tests for primality, in: Proceedings of the seventh annual ACM symposium on Theory of computing, 1975, pp. 234–239.

[11] M. O. Rabin, Probabilistic algorithms. (1976).

[12] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM review 41 (2) (1999) 303–332.

[13] B. Jansen, K. Nakayama, Neural networks following a binary approach applied to the integer prime-factorization problem, in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., Vol. 4, IEEE, 2005, pp. 2577–2582.

[14] M. Mishraa, U. Chaturvedib, K. Shukla, A new heuristic algorithm based on molecular geometry optimization and its application to the integer factorization problem, in: 2014 International Conference on Soft Computing and Machine Intelligence, IEEE, 2014, pp. 115–120.