# Estudo de caso de algoritmos para a fatoração de inteiros

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, o algoritmo RSA. Será apresentado o estado-da-arte de algoritmos exatos, métodos utilizados para resolver o problema das formas mais eficientes já demonstradas, além de heurísticas e algoritmos aproximativos para o problema. Por fim, terá a implementação de um algoritmo exato e um aproximativo para a resolução do problema da fatoração, com aplicações e comparação dos seus resultados.

*Palavras-chave:* criptografia; fatoração de inteiros; problema da fatoração; números primos; 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 pro-

tocolo 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 soluções para o problema da fatoração de números primos, o uso de um algoritmo exato e de um algoritmo aproximativo utilizando a aproximação a partir da operação de raiz quadrada e fazer a comparação de seus resultados.

Inicialmente iremos descrever melhor o problema da fatoração de inteiros, analisar sua complexidade, mostrar o estado-da-arte para algoritmos exatos e heurísticos e por fim terá uma seção para tratar sobre a implementação de alguns algoritmos que resolvem o problema da fatoração.

## 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 $n$.

### *2.1. Complexidade do Problema*

A fatoração de inteiros poder ser resolvida em tempo polinomial em um computador clássico é um problema não resolvido na computação. 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 $k$. Ainda 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 NP e co-NP, de forma que as respostas "sim"e "não"podem ser verificadas em tempo polinomial [2]. Além disso, a classe está na classe UP e co-UP de acordo com o teorema fundamental da aritmética e [3].

Seja $b$ o comprimento da entrada de valor $n$. Considerando $b = \log_2 n$, para calcular a complexidade $O(n)$ teremos $O(2^b)$, 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 $O\left(\sqrt{2^b}\right) = O\left(2^{\frac{2}{b}}\right)$, ainda sendo exponencial.

## 3. Estado-da-Arte de Algoritmos Exatos

A seguir descreveremos um pouco do estado-da-arte de algoritmos exatos, apresentando algumas das soluções mais famosas para a resolução do problema.

### *3.1. 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 teste de primalidade apresentado por Fibonacci em seu livro *"liber abaci"* [4] e reforçado pelo teorema que demonstra que se $n$ for um inteiro composto, então $n$ tem um divisor primo menor ou igual que $\sqrt{n}$ [5]. 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.

### *3.2. Método da curva elíptica*

O método da curva elíptica foi proposto por Lenstra [6] 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 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 [7]

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

### *3.3. 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,

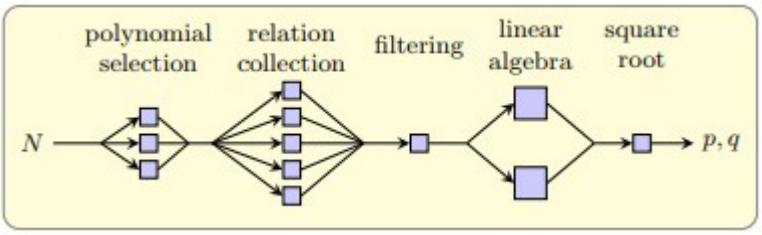


Figura 1: Etapas do algoritmo GNFS.[8]

temos $MDC(119+1287,N) = 37$ e $MDC(119-1287,N) = 73$ [8], 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. [8]

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

*3.4. Algoritmo de Fermat*

O Algoritmo de Fermat [9] baseia-se na representação de um inteiro ímpar ser representado pela diferença de dois quadrados de forma única. Então por exemplo, dado um inteiro ímpar $N$ é possível representá-lo da forma

$$N = X^2 - Y^2 = (X+Y)(X-Y)$$

Manipulando essa fórmula teremos:

$$\begin{aligned} N = X^2 - Y^2 &\implies X^2 = N + Y^2 \\ &\implies X = \sqrt{N+Y^2} \end{aligned}$$

De forma que o algoritmo irá testar diferentes valores para $Y$ até que seja encontrado um valor inteiro para $X$. No momento que $\sqrt{N+Y^2}$ seja um inteiro, os valores $A$ e $B$ de resultado do algoritmo serão obtidos substituindo $X$ e $Y$ em $N = (X+Y)(X-Y)$, onde $A = (X+Y)$ e $B = (X-Y)$.

O algoritmo também pode ser usado para números pares, porém, para estes não se tem a garantia que os números gerados serão os únicos possíveis. É possível destacar

também que os primos gerados são números muito próximos, então para chaves RSA geradas com primos muito próximos podem ser quebradas facilmente com o método de fatoração de Fermat.

## 4. Heurísticas e Algoritmos Aproximativos para o Problema

Abaixo citaremos algumas heurísticas encontradas para a resolução do problema da fatoração de inteiros.

### *4.1. Heurística de Shoup*

SHOUP [7] 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.

### *4.5. Algoritmo Rho de Pollard*

O algoritmo de Rho [15] 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 $q$. [16]

## 5. Links Importantes

- Implementações dos algoritmos disponível no Github: `https://github.com/joaoguilac/prime_factorization`
- Algoritmo Aproximativo de base do artigo: `https://www.sciencedirect.com/science/article/pii/S0898122111001131`

## 6. Implementação

Inicialmente, foi implementado o algoritmo apresentado no artigo *Prime factorization using square root approximation* [16] como algoritmo aproximativo. Para o algoritmo exato foi implementado um algoritmo baseado no *Trial Division* 3.1.

### *6.1. Algoritmo Aproximativo*

O pseudocódigo do algoritmo aproximativo implementado pode ser visto abaixo:

**Algoritmo 1:** Pseudocódigo do algoritmo aproximativo baseado na aproximação de raiz quadrada

**Entrada:** $n$: inteiro a ser fatorado, $P_r$: uma lista de subconjunto de números primos, $U_{pr}$: um limite indicando o maior $r = pq$ permitido, $Highstep$: um limite que indica as etapas permitidas para encontrar uma solução

**Saída:** Vetor com 2 fatores de $n$.

```
1  ε := 0.02, r := 2, steps := 0;
2  fx := √n − [√n];
3  repita
4      enquanto r < U_pr E fx × r ≠ [fx × r] ± ε faça
5          Encontre um possível múltiplo r de fx que gera seu inteiro mais
             próximo;
6          steps := steps + 1;
7          fatorar r em p e q;
8          A := [√n] × q/p;
9          se A não é um divisor de n então
10             Encontre em P_r o valor mais próximo P_ri a A;
11             A := MDC(P_ri−2 × P_ri−1 × P_ri × P_ri+1 × P_ri+2, n);
12         fim
13     fim
14 até A é um divisor inteiro de n OU steps > Highstep;
15 se steps < Highstep então
16     B := n/A;
17 fim
18 senão
19     print("n é primo ou nenhum solução foi encontrado para os limites
         escolhidos");
20 fim
```

Foi notado alguns problemas na implementação deste algoritmo 1 devido a falta

de explicação de passos no artigo *Prime factorization using square root approximation* [16]. 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.

Dessa forma, o maior problema na implementação do algoritmo foi do passo da linha 5, em que não é explicado de forma exata em como realizar essa busca. A solução que adotamos foi de ir incrementando o possível valor de $r$ e caso fosse obtido um valor de $\varepsilon'$, onde $\varepsilon' = [fx * r] - fx * r$, menor que o anterior, teríamos encontrado um possível valor de $r$.

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 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.

Como o foco deste trabalho é na implementação do algoritmo exato, para entender melhor a aproximação e a correção do algoritmo é possível obter em [16].

### *6.2. Algoritmo Exato: Trial Division*

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

**Algoritmo 2:** 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 seus fatores;

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

É importante destacar que este algoritmo é baseado no método da divisão sucessiva

3.1, mas não é exatamente o algoritmo por conta da última linha, pois o objetivo do *Trial Division* é obter todos os fatores de um número $n$, aqui objetivamos obter 2 fatores $A$ e $B$ que multiplicados resultam $n$.

Também é possível notar que a técnica utilizada é basicamente força bruta, não é utilizado nenhuma técnica avançada a fim de melhorar a eficiência.

#### *6.2.1. Correção*

O algoritmo é correto pois é obtido todos os números primos até a $\sqrt{n}$, que é o máximo que um número $n$ pode ser divisível. Para garantir que todos os números obtidos são primos, para um possível primo $i$ é feito a divisão desse número por todos os números de 2 até $\sqrt{i}$, caso $i$ seja divisível por algum desses números, então $i$ não é primo. Depois de obter todos os primos, faz divisão sucessivas de $n$ por cada primo possível até que seja armazenado todos os primos que dividiram o número. Por fim, pega o primeiro fator obtido e divide pelo número $n$, o resto da divisão será o outro fator gerado, gerando assim 2 números que multiplicados resulta em $n$.

#### *6.2.2. Análise*

Sobre a complexidade do algoritmo, na primeira linha 2 iremos pegar todos os números primos até $\sqrt{n}$ e para cada número desse iremos testar no máximo até $\sqrt{i}$ para saber se ele é divisível ou não por um número maior que 2, ou seja, é primo. Dessa forma, no pior caso teríamos $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$. Na segunda linha, ele fará divisões sucessivas de $n$, em pior caso teria $O(\sqrt{n})$. E na última linha irá tentar agrupar os fatores em 2 e para isso irá no máximo até $\sqrt{n}$. Portanto, o algoritmo tem complexidade $O(n)$. Isso se dá porque o algoritmo é pseudopolinomial, já que seu desempenho é exponencial de acordo com o comprimento da entrada 2.1.

### *6.3. Algoritmo Monte Carlo baseado na Heurística Rho de Pollard*

O algoritmo abaixo é um algoritmo probabilístico Monte Carlo, baseado na heurística Rho de Pollard que irá tentar encontrar um dos fatores de um inteiro $n$, entretanto seu tempo de execução e sucesso não são garantidos. O algoritmo é proposto por Richard P. Brent no artigo "*A monte carlo method for factorization*"[15], mas também é possível obter detalhes dele no livro *Introduction to Algorithms* de Thomas H. Cormen [17].

No nosso algoritmo iremos seguir os mesmos passos só que ao final iremos dividir o número fatorado pelo fator obtido para ter 2 fatores assim como requerido pelo algoritmo RSA:

**Algoritmo 3:** Algoritmo baseado na *heurística Rho de Pollard*

**Entrada:** Inteiro $n$ a ser fatorado.

**Saída:** Vetor com 2 fatores de $n$

```
1  i = 1;
2  x = Random(0, n − 1);
3  y = x;
4  k = 2;
5  fatores[ ];
6  enquanto Verdadeiro faça
7      i = i + 1;
8      x = (x^2 − 1) mod n;
9      d = MDC(|y − x|, n);
10     se d ≠ 1 e d ≠ n então
11         fatores.add(d);
12         pare;
13     fim
14     se i == k então
15         y = x;
16         k = 2k;
17     fim
18 fim
19 se fatores.tamanho() > 0 então
20     B = n/factores[0];
21     fatores.add(B);
22 fim
```

### *6.3.1. Correção*

O algoritmo se da a partir da criação de uma sequência de valores, o $x$, onde, dado os inteiros $c \geq 1$ e $t \geq 0$, então $x_0, x_1, ..., x_{c+t-1}$ são valores distintos que formam a "cauda"da letra grega Rho ($\rho$) , e para um $i \geq t$, $x_{c+i}$ serão os $i$ valores no ciclo, que formam a "cabeça"da letra Rho, e $x_{c+i} \equiv x_i (mod\ n)$. [15]

A capacidade do algoritmo encontrar o fator de $n$ vem da observação de que, se $x_t^{'} + 1 = x_t^{'} + u + i$, ou seja um elemento da cauda seja igual do ciclo, então $n|(x_t + u + i - x_t + i)$, logo $mdc(x_t + u + i - x_i + i, n) > 1$, sendo justamente o mdc entre $(|y - x|, n)$ dado que $y$ guardará um valor que estará na cauda. [17]

### *6.3.2. Análise*

Sendo $p$ o menor fator primo de $n$. O algoritmo de Pollard terá tempo $O(n^{\frac{1}{4}})$, pois o algoritmo é dependente da função $mdc(|y - x|, n)$, supondo um $j > 0$, quando $x'_{2j} = x'_j$, então $mdc(|x'_{2j} - x'_j|, n \geq p)$, logo o Rho Pollard executará até $j \leq p$ iterações. Como $p \leq n^{\frac{1}{2}}$, então o número de vezes que a função $mdc(|y - x|, n)$ ocorrerá é $n^{\frac{1}{4}}$, e tome que o algoritmo para fazer o mdc é o algoritmo Euclidiano que tem complexidade $O(log\ n)$, e $log\ n \leq n^{\frac{1}{4}}$, então o algoritmo Rho tomará o pior caso como $O(n^{\frac{1}{4}})$. [18]

### *6.3.3. Melhorias*

Richard P. Brent [18] propõe uma melhoria de até 24% na eficiência do algoritmo a partir da omissão de determinadas iterações no *mdc* da linha 7, pois é sabido que eles podem ser dispensados, para isso, é calculado um produtório $q_i$ em um determinado $z_j$ e computado o $mdc(q_i, n)$ quando $i$ é um múltiplo de $m|log\ n \ll m \ll n^{\frac{1}{4}}$. Entretanto a probabilidade do algoritmo falhar a partir do resultado $q_i = 0$ é maior, para isso Brent propõe a implementação de um backtracking para retornar ao estado do último *mdc* computado.

## *6.4. Algoritmo Exato: Algoritmo de Fermat*

O Algoritmo de Fermat tem o seguinte pseudocódigo:

**Algoritmo 4:** Algoritmo de Fermat

**Entrada:** Inteiro $n$ a ser fatorado.

**Saída:** Fatores $A$ e $B$ de $n$.

```
1  Y = 1;
2  X = sqrt(n + Y^2);
3  enquanto Verdadeiro faça
4      se X é um quadrado perfeito então
5          A = X + Y;
6          B = X - Y;
7          pare;
8      fim
9      Y = Y + 1;
10     X = sqrt(n + Y^2);
11 fim
```

Mais detalhes do algoritmo é possível ter em 3.4.

Também é possível notar que a técnica utilizada é basicamente força bruta, não é utilizado nenhuma técnica avançada a fim de melhorar a eficiência.

### *6.4.1. Correção*

Como é possível representar $n$ da forma $n = X^2 - Y^2 = (X+Y)(X-Y)$, ao se descobrir um valor inteiro de $Y$ que gere um quadrado perfeito temos um inteiro válido para $X$ e para $Y$, e como esses valores fazem parte de $(X+Y)(X-Y)$, podemos obter 2 fatores que multiplicados geram $n$.

### *6.4.2. Análise*

O algoritmo tem complexidade $O(\sqrt{n})$. Isso pois em $\sqrt{n+Y^2}$ no máximo ele irá ter que executar o *loop* $\sqrt{n}$ vezes até que obtenha um quadrado perfeito.

## *6.5. 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);
}
```

### *6.6. 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;
```

## 7. Resultados

O método de aproximação por raiz quadrada é uma proposta melhor para fatoração de números grandes, no entanto, em alguns casos o algoritmo não conseguiu encontrar os fatores.

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

| 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 do algoritmo aproximativo com o algoritmo baseado no *Trial Division* (Algoritmo 2) 2

É possível notar que o algoritmo aproximativo 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.

Agora iremos comparar o algoritmo aproximativo 1 com o algoritmo probabilístico implementado 3:

| Número | Fatores Algoritmo 1 | Tempo Algoritmo 1 | Fatores Algoritmo 3 | Tempo Algoritmo 3 |
|---|---|---|---|---|
| 15 | 5 e 3 | 0.041337ms | 5 e 3 | 0.013635ms |
| 40.000 | 100 e 400 | 0.004368ms | 5 e 8000 | 0.01561ms |
| 51.923 | 379 e 137 | 0.041708ms | 137 e 379 | 0.013235ms |
| 55.872 | 9 e 6208 | 0.210324ms | 24 e 2328 | 0.011141ms |
| 111.000 | 500 e 222 | 0.006192ms | 600 e 185 | 0.011371ms |
| 450.000 | 3 e 150.0000 | 0.154289ms | 240 e 1875 | 0.013485ms |
| 37.527.851 | N/A | 1.65729ms | 6131 e 6121 | 0.024977ms |
| 260.100.000 | 2 e 130.050.000 | 1.16428ms | 3600 e 72.250 | 0.012053ms |
| 542.448.900 | 81 e 6.696.900 | 0.438151ms | 420 e 1.291.545 | 0.012383ms |
| 2.147.483.648 | N/A | 0.553065ms | 16 e 134.217.728 | 0.011983ms |
| 10.000.000.000 | 50000 e 200000 | 0.057468ms | 4 e 2.500.000.000 | 0.013435ms |
| 14.789.166.241 | N/A | 1.10207ms | N/A | 0.192641ms |
| $10^{12}$ | $5 \cdot 10^5$ e $2 \cdot 10^6$ | 0.37923ms | 16 e $625 \cdot 10^8$ | 0.012253ms |
| $10^{14}$ | $5 \cdot 10^6$ e $2 \cdot 10^7$ | 2.89217ms | 5 e $2 \cdot 10^{13}$ | 0.012734ms |

Tabela 2: Resultados da execução do algoritmo aproximativo com o algoritmo probabilístico Monte Carlo (Algoritmo 3) 3

Podemos notar que o algoritmo probabilístico foi o que obteve o melhor resultado dos 3, não tendo grande variação no tempo de execução diante do aumento do tamanho da instância. O grande problema se deve pelo algoritmo não ter garantia de sucesso 100% garantido como comentado anteriormente.

Por fim, comparando o aproximativo com o Algoritmo de Fermat implementado (exato) 4:

| Número | Fatores Algoritmo 1 | Tempo Algoritmo 1 | Fatores Algoritmo 4 | Tempo Algoritmo 4 |
|---|---|---|---|---|
| 15 | 5 e 3 | 0.041337ms | 5 e 3 | 0.000722ms |
| 187 | N/A | 0.041337ms | 17 e 11 | 0.000612ms |
| 3233 | 53 e 61 | 0.017704ms | 61 e 53 | 0.000832ms |
| 40.000 | 100 e 400 | 0.004368ms | 250 e 160 | 0.001022ms |
| 51.923 | 379 e 137 | 0.041708ms | 379 e 137 | 0.001613ms |
| 55.872 | 9 e 6208 | 0.210324ms | 288 e 194 | 0.000972ms |
| 111.000 | 500 e 222 | 0.006192ms | 370 e 300 | 0.001453ms |
| 450.000 | 3 e 150.0000 | 0.154289ms | 750 e 600 | 0.001243ms |
| 37.527.851 | N/A | 1.65729ms | 6131 e 6121 | 0.000671ms |
| 14.789.166.241 | N/A | 1.10207ms | 983911 e 15031 | 4.27874ms |

Tabela 3: Resultados da execução do algoritmo aproximativo com o algoritmo de Fermat (Algoritmo 4) 4

Podemos observar um tempo melhor que o algoritmo 2 2 em relação ao tamanho da instância, e uma tendência de piora em relação ao aproximativo.

### *7.1. 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++

## 8. Conclusão

Neste artigo foi tentado explorar implementações de algoritmos para resolver o problema da fatoração de inteiros e comparar os resultados de seus comportamentos. Mesmo com a dificuldade na implementação do algoritmo aproximativo, ainda assim foi possível observar que o ele possui uma melhor eficiência quando há um aumento no tamanho da instância em comparação ao algoritmo exato. 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] S. G. Krantz, A prova está no pudim: a natureza mutável da prova matemática, Nova Iorque: Springer, p. 203, 2011.

[3] 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

[4] R. A. Mollin, A brief history of factoring and primality testing bc (before computers), Mathematics magazine 75 (1) (2002) 18–29.

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

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

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

[8] 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.

[9] N. A. Carella, Note on integer factoring algorithms ii (2007). arXiv:math/0702227.

[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.

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

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

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2022.

[18] R. P. Brent, An improved monte carlo factorization algorithm, BIT Numerical Mathematics 20 (2) (1980) 176–184.