1 2 9 0

UNIVERSIDADE DE COIMBRA

Pedro Dinis de Paulo Machado

CRIPTOANÁLISE QUÂNTICA DA CIFRA RSA

Dissertação no âmbito do Mestrado em Matemática, Ramo Matemática Aplicada e Computação orientada pelo Professor Doutor Pedro Henrique e Figueiredo Quaresma de Almeida e apresentada ao Departamento de Matemática da Faculdade de Ciências e Tecnologia.

Julho de 2025

Criptoanálise Quântica da Cifra RSA

Pedro Dinis de Paulo Machado

1 2 9 0

UNIVERSIDADE DE COIMBRA

Mestrado em Matemática
Master in Mathematics

Julho 2025 / July 2025

Agradecimentos

Aos meus pais, ao meu irmão, à Carolina e ao novo membro da nossa família Teodoro, por sempre confiarem nas minhas capacidades e pelo esforço associado à continuação do meu percurso escolar nas melhores condições possíveis.

Ao professor Pedro Quaresma, pela oportunidade de trabalhar consigo, pela sua orientação científica e pela disponibilidade que sempre demonstrou para me ajudar.

A todos os outros professores que, de certa forma, marcaram o meu percurso académico até ao momento.

Aos Resistentes, que é como quem diz Tinoco, Rafa, Rodrigo, Xico Zé, Carolina e Catarina, pelas noites de convívio, pelas aventuras inesquecíveis e por todos os momentos que passámos juntos. Certamente que, sem vocês, esta caminhada teria sido bastante diferente.

À FAN-Farra, pelos dias de alegrias e vivências.

Resumo

A computação quântica representa uma mudança de paradigma com implicações profundas na área da criptografia. Esta dissertação tem como objetivo analisar o impacto da computação quântica nos sistemas criptográficos clássicos, com ênfase na vulnerabilidade de algoritmos baseados na fatorização de números inteiros, como o *RSA*. O foco central é o estudo do algoritmo de Shor, que permite a fatorização em tempo polinomial, contrastando com os método não-quânticos que o método mais otimizado faz em tempo subexponecial.

Ao longo do trabalho, são apresentados os fundamentos teóricos da computação quântica e na criptografia, seguidos de uma explicação detalhada do algoritmo de Shor. Foi também realizada uma implementação prática com recurso às bibliotecas disponibilizadas pelo qiskit, ilustrando o funcionamento do algoritmo em ambiente simulado. Por fim, discute-se a viabilidade prática da sua execução em computadores quânticos atuais e são apontadas direções futuras no contexto da criptografia pós-quântica.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Qual a importância de transmitir uma mensagem? Desde os primórdios da humanidade, o conceito de mensagem, uma informação que é transmitida de A para B, é utilizado na forma mais básica de comunicação, sendo que desde cedo existiu a necessidade de acrescentar à mensagem o fator da encriptação, para assim evitar cair nas mãos erradas. Podemos mencionar um exemplo de uma mensagem encriptada ao tempo do Imperador Romano Júlio César. O mesmo usou o sistema hoje conhecido por "*Cifra de César*", que consiste em usar um deslocamento de uma letra para qualquer posição no alfabeto, da seguinte forma: A -> D, B -> E, e por aí em adiante [17].

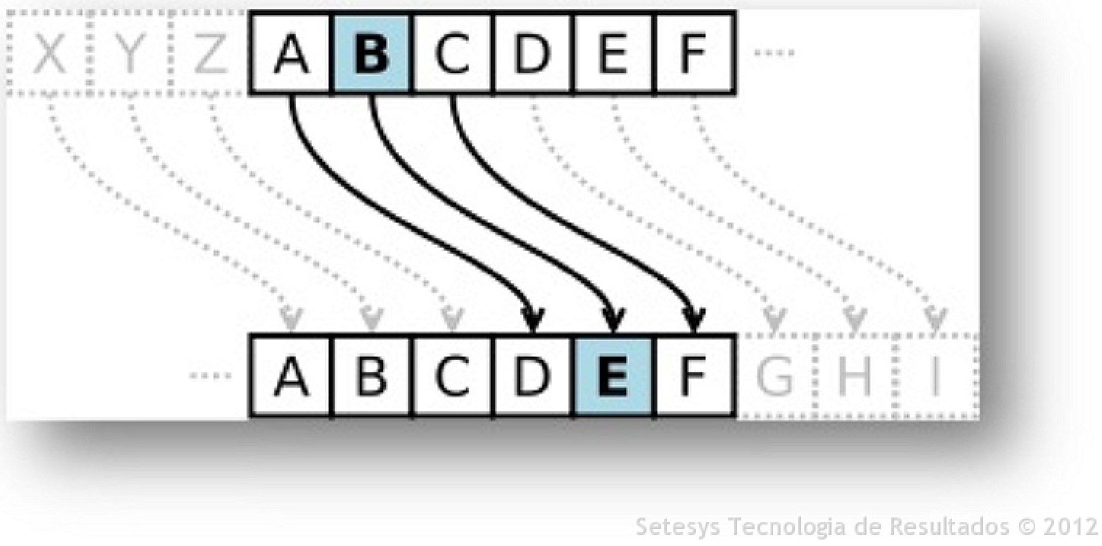

Fig. 1.1 Cifra César[1]

Atualmente, dispomos de mensagens protegidas por vários sistemas de encriptação. Contudo, existe também a criptoanálise, que consiste em múltiplas aproximações para quebrar os sistemas criptográficos, desencriptando, mesmo que parcialmente, as mensagens encriptadas. Um exemplo

[1]Fonte: adaptado de Vacatronics (2021). Disponível em: https://medium.com/vacatronics/cifra-de-césar-em-python-8d02d3bc7d42

curioso é a desencriptação por análise de frequência, que consiste em observar a frequência dos caracteres numa mensagem intercetada. Contamos a ocorrência de cada carácter e tentamos substituí-los, tendo em conta que as vogais estão sempre mais presentes no alfabeto e são, geralmente, as letras mais usadas. Por exemplo, é mais provável encontrar um 'p' antes de um 'a'. À medida que vamos fazendo as substituições, torna-se mais fácil descobrir outras letras, facilitando progressivamente a desencriptação da mensagem.

A criptografia teve uma enorme importância durante a Segunda Guerra Mundial, como sabemos. Caso uma guerra volte a instalar-se no nosso planeta, esta terá ainda maior relevância, dado que vivemos na era da informação. A maior ameaça nesta época é o acesso não autorizado ao conteúdo de mensagens com informação crucial, que pode determinar o resultado de batalhas ou até da própria guerra.

A computação quântica representa uma das maiores ameaças aos sistemas de criptografia atualmente utilizados, devido à sua capacidade de processar informações de forma exponencialmente mais rápida do que os computadores clássicos. Embora ainda estejamos nos estágios iniciais dessa tecnologia, os avanços recentes têm sido significativos, sinalizando que, em um futuro próximo, sistemas como o *RSA*, baseados na dificuldade de fatorização de números primos, podem se tornar vulneráveis. Essa possibilidade reforça a necessidade de explorar alternativas robustas para garantir a segurança dos dados na era da computação quântica.

Capítulo 2

Chaves Públicas e Chaves Simétricas

Atualmente, existem dois tipos de métodos comuns utilizados para encriptar informação:

- criptografia assimétrica (chaves públicas)
- criptografia simétrica

2.1 Criptografia Assimétrica ou Sistema de Chaves Públicas

A criptografia assimétrica, também conhecida como sistema de chaves públicas, consiste num sistema criptográfico que cria um par de chaves:

- chaves públicas - são de conhecimento público.
- chaves privadas - são conhecidas apenas pelos proprietários.

Este sistema utiliza ambas as chaves em dois processos principais:

1. Autenticação: chave privada emparelhada enviou a mensagem.
2. Encriptação: apenas o portador da chave privada emparelhada pode desencriptar a mensagem encriptada com a chave pública.

Os algoritmos de chave pública baseiam-se em problemas matemáticos que, atualmente, não têm uma solução eficiente. Estes sistemas são fáceis de implementar por um utilizador, uma vez que basta gerar computacionalmente um par de chaves (uma pública e uma privada, ambas baseadas em números primos grandes), e utilizá-las para encriptar e desencriptar mensagens. A força deste sistema reside na dificuldade de decifrar a chave privada a partir da chave pública, um processo que é computacionalmente impraticável. Por esta razão, a chave pública pode ser partilhada sem comprometer a segurança, desde que a chave privada seja mantida secreta. Em suma, o sistema é seguro enquanto a chave privada permanecer confidencial.

Vantagens:

- Simplifica a verificação de identidades e a distribuição de chaves públicas.
- Oferece alta segurança para autenticação e sigilo.

Desvantagens:

- Requer maior poder computacional, resultando num desempenho mais lento.
- As chaves públicas precisam de ser verificadas para evitar ataques.

Fig. 2.1 Criptografia assimétrica[2]

2.2 Criptografia Simétrica ou Sistema de Chaves Simétricas

O sistema de chaves simétricas, ou criptografia simétrica, consiste em encriptar uma mensagem utilizando uma única chave, que é usada tanto para encriptar como para desencriptar uma mensagem. Dois dos esquemas de encriptação simétrica mais comuns atualmente, são baseados em cifras de bloco (*block ciphers*).

Cifras de Bloco (*block ciphers*)

As cifras de bloco agrupam os dados em blocos de tamanho predeterminado. Cada bloco é encriptado usando a chave correspondente e o algoritmo de encriptação.

Vantagens :

- Grande velocidade de processamento.
- Menor uso de força computacional para gerar a chave,

Desvantagens :

- Caso a chave seja descoberta o sistema é comprometido.
- Em sistemas com muitos utilizadores é necessário uma boa gestão das chaves.

[2]Fonte: Adaptado de ResearchGate (2020). Disponível em: https://www.researchgate.net/figure/Figura-7-Esquema-da-criptografia-assimetrica_fig5_332948161

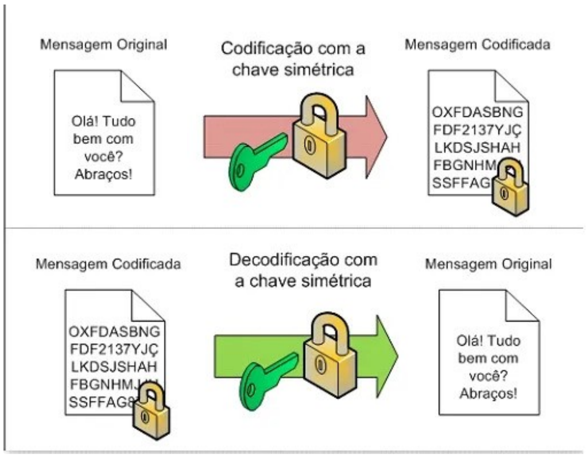

Fig. 2.2 Encriptar/Desencriptar Mensagens[4]

2.3 Sistemas de Chaves Públicas e de Chaves Simétricas

Como vimos anteriormente, estes dois sistemas diferem na maneira como operam. Então, consoante o nosso objetivo, pode ser mais vantajosos usar um ou outro.

Tabela 2.1 Comparação entre Criptografia Simétrica e Assimétrica

Aspeto	Criptografia Simétrica	Criptografia Assimétrica
Definição	Usa a mesma chave para encriptar e desencriptar.	Usa um par de chaves: pública (para encriptação) e privada (para desencriptação).
Vantagens	• Grande velocidade de processamento. • Menor uso de força computacional.	• Simplifica a distribuição de chaves públicas. • Oferece alta segurança para autenticação e sigilo.
Desvantagens	• A segurança depende do segredo da chave. • Exige boa gestão em sistemas com muitos utilizadores.	• Requer maior poder computacional. • Chaves públicas precisam de ser verificadas para evitar ataques.

[4]Fonte: Adaptado de UFRJ – Grupo de Teleinformática e Automação (2008). Disponível em: https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2008_2/hugo/Criptografia.html

Capítulo 3

Cifra *RSA*

3.1 Introdução

Em 1978, Rivest, Shamir e Adleman (*RSA*) criaram uma nova maneira de cifrar e decifrar mensagens. Este sistema consiste em multiplicar dois números primos de grande dimensão,[1] e juntar um valor auxiliar e atribuímos o resultado obtido a uma mensagem. A cifra *RSA* usa o sistema de chaves públicas anteriormente mencionado, sendo que a nossa chave pública criada é o produto dos dois números primos anteriormente selecionados. Este sistema tem vários níveis de segurança sendo que o mais comum é *RSA*-2048 que tem um com o comprimento cerca de 248 dígitos o sistema mais seguro usado para situações de máxima segurança é o *RSA*-4096 tem um comprimento cerca de 512, sendo que para um computador clássico para conseguir fatorizar a mensagem, demoraria milhões de anos. Para percebermos mais detalhadamente como funciona a cifra *RSA*, primeiro precisamos de ter alguns fundamentos [16].

3.2 Fundamentos

Definição 1 (Divisibilidade) *Sejam a e b dois números inteiros. Se $a \neq 0$ e existe um inteiro c tal que $b = ac$, dizemos que a divide b ou que a é um divisor de b, ou ainda, que b é um múltiplo de a e escrevemos $a \mid b$. Se a não divide b, escrevemos $a \nmid b$.*

Definição 2 (Número Primo) *A qualquer número inteiro maior do que 1 com apenas dois divisores inteiros positivos, o 1 e o próprio número, chamamos número primo. A um número inteiro maior do que 1 que não seja número primo diz-se número composto.*

Definição 3 (Máximo Divisor Comum (mdc)) *Sejam a e b dois inteiros tais que pelo menos um deles é não nulo. Chamamos máximo divisor comum ao maior inteiro do conjunto dos divisores comuns de a e de b. Denotamos este elemento por $mdc(a,b)$ [18].*

[1] Podemos consultar as necessidades desta cifra atualmente em: https://www.ibm.com/docs/en/zos/2.4.0?topic=2-algorithms-key-sizes

Exemplos de mdc(a,b)

- $\text{mdc}(25,5) = 5$
- $\text{mdc}(20,10) = 10$
- $\text{mdc}(10,15) = 5$

Através da fatorização podemos descobrir o $\text{mdc}(168,180)$, fazendo a fatorização de ambos tal que, 168 = $2^3 \times 3 \times 7$ e que 180 = $2^2 \times 3^2 \times 5$. O próximo passo consiste em obter o produto dos fatores comuns de menor expoente, ou seja, concluímos assim que o máximo divisor comum entre 168 e 180 é 12, pois $2^2 \times 3$, que está presente nas duas fatorizações.

Teorema 1 (Algoritmo de Euclides) *[14]*
O algoritmo de Euclides calcula o $mdc(a,b)$ recursivamente utilizando o resto da divisão, dado dois números a e b, tal que a > b:

$$a = b \times q + r$$

Sendo r o resto da divisão e q o quociente da divisão.
Substituindo agora o a por b e b por r, repetimos o processo sucessivamente, até o nosso resto final for zero, quando isto se verificar temos o nosso mdc.

Exemplo do Algoritmo de Euclides
Vamos calcular o $\text{mdc}(252,105)$

1. Dividindo 252 por 105 temos um resultado de 2.4, sendo assim o nosso q = 2, e 2 x 105 = 210, sendo que o nosso r será 252 - 210 = 42

   252 = 105 x 2 + 42

2. Repetindo o processo com 105 e 42 temos:

   105 = 42 x 2 + 21

   em que q igual a 2 e r igual a 21

3. Repetindo o processo com 42 e 21 temos:

   42 = 21 x 2 + 0

   Então temos o processo terminado, porque r igual a zero, sendo que o nosso mdc(252,105)=21

Definição 4 (Algoritmo de frações contínuas) *[9]*

Uma fração contínua é uma forma de representar números racionais (ou irracionais) como uma sequência aninhada de inteiros. A forma geral é:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

Onde $a_0, a_1, a_2, \ldots$ são inteiros e $a_1, a_2, \ldots > 0$.
Por exemplo, a fração contínua de $\frac{415}{93}$ é:

$$\frac{415}{93} = 4 + \cfrac{1}{2 + \cfrac{1}{6 + \cfrac{1}{7}}} = [4; 2, 6, 7]$$

Definição 5 (Função Totiente ou função φ) *[4]*

Seja n um número inteiro positivo. Então, a função totiente de Euler $\varphi(n)$ é dada por:

$$\varphi(n) = \#\left\{k \in \mathbb{Z}^+ \mid 1 \leq k \leq n,\ mdc(k,n) = 1\right\}$$

Ou seja, $\varphi(n)$ conta quantos números k existem entre 1 *e n que possuem* $mdc(k,n) = 1$.

Exemplo da Função Totiente

Vamos calcular $\varphi(9)$, ou seja, todos os números coprimos entre 1 e 9.

- mdc(1, 9) = 1
- mdc(2, 9) = 1
- mdc(3, 9) = 3 (não é coprimo)
- mdc(4, 9) = 1
- mdc(5, 9) = 1
- mdc(6, 9) = 3 (não é coprimo)
- mdc(7, 9) = 1
- mdc(8, 9) = 1
- mdc(9, 9) = 9 (não é coprimo)

Os números que são coprimos com 9 são: $1, 2, 4, 5, 7, 8$.
Portanto, o número de inteiros coprimos com 9 é 6. Logo, temos:

$$\varphi(9) = 6$$

Teorema 2 (Teorema de Euler) *[4]*

Se $mdc(a,n) = 1$, então

$$a^{\varphi(n)} \equiv 1 \mod n.$$

Exemplo do Teorema de Euler

Seja $a = 3$ e $n = 10$. Como $\text{mdc}(3,10) = 1$, podemos aplicar o Teorema de Euler.

Primeiro, vamos calcular $\varphi(10)$ usando a definição: contar quantos inteiros entre 1 e 10 são coprimos com 10:

- $\text{mdc}(1,10) = 1$
- $\text{mdc}(2,10) = 2 \quad$ (não é coprimo)
- $\text{mdc}(3,10) = 1$
- $\text{mdc}(4,10) = 2 \quad$ (não é coprimo)
- $\text{mdc}(5,10) = 5 \quad$ (não é coprimo)
- $\text{mdc}(6,10) = 2 \quad$ (não é coprimo)
- $\text{mdc}(7,10) = 1$
- $\text{mdc}(8,10) = 2 \quad$ (não é coprimo)
- $\text{mdc}(9,10) = 1$
- $\text{mdc}(10,10) = 10 \quad$ (não é coprimo)

Os números que são coprimos com 10 são: $1, 3, 7, 9$. Logo:

$$\varphi(10) = 4$$

Aplicando o Teorema de Euler:

$$3^{\varphi(10)} = 3^4 = 81 \Rightarrow 81 \mod 10 = 1$$

Portanto:

$$3^4 \equiv 1 \mod 10,$$

o que confirma o Teorema de Euler neste caso.

Teorema 3 (Pequeno Teorema de Fermat[3]) *Se p é primo, então*

$$a^p \equiv a \mod p,$$

para qualquer inteiro a.

Se n tem a seguinte decomposição em fatores primos:

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k},$$

onde $p_1, p_2, \ldots, p_k$ são primos distintos e $e_1, e_2, \ldots, e_k$ são inteiros positivos, então a função totiente $\varphi(n)$ é dada por:

$$\varphi(n) = n \times \left(1 - \frac{1}{p_1}\right) \times \left(1 - \frac{1}{p_2}\right) \times \cdots \times \left(1 - \frac{1}{p_k}\right).$$

Em particular:

- Se n é um número primo, então $\varphi(n) = n - 1$.
- Se $n = 7$, então $\varphi(7) = 6$.

Definição 6 (Teorema de Fatorização Baseado no Período) *[10]*
Seja N um número composto, r um expoente par, e m uma base inteira positiva. Então:
Os divisores de N podem ser encontrados avaliando:

$$mdc\left(m^{\frac{r}{2}} - 1, N\right) \quad e \quad mdc\left(m^{\frac{r}{2}} + 1, N\right),$$

pois estas expressões podem conter fatores primos de N, isto não acontece quando r não é bem escolhido, ou m não é uma base aceitável.

Exemplo do Teorema de Fatorização Baseado no Período
Seja $N = 15$, $r = 4$ e $m = 7$, então:

$$mdc(7^{4/2} - 1, N) = mdc(49 - 1, N) = 3$$

$$mdc(7^{4/2} + 1, N) = mdc(49 + 1, N) = 5$$

Então podemos ver que os fatores de N são 3 e 5, tal que N = 3 × 5.

3.2.1 Cifra *RSA*

Agora, com os conhecimentos anteriormente mencionados, vamos iniciar o processo da criação de uma cifra pelo sistema *RSA*, dividindo em 3 etapas: Geração das chaves; encriptação e desencriptação de uma mensagem.

1. **Geração das Chaves**

   Este passo consiste em criar uma chave pública e uma privada.

   - Começamos por escolher dois números primos de grande dimensão: p e q.
     Sejam $p = 61$ e $q = 53$.
   - Calculamos $n = p \times q$. O n vai ser utilizado na criação da chave pública e na chave privada.
     $n = 61 \times 53 = 3233$

[3] https://mathworld.wolfram.com/FermatsLittleTheorem.html

- Utilizamos a agora a função de ϕ.

$$\phi(n) = (p-1) \times (q-1)$$

  Seja ϕ(3233) = (61 - 1) x (53 - 1) = 60 x 52 = 3120.

- Escolhe-mos um número a, em que 1 < a < $\phi(n)$, seja coprimo com $\phi(n)$(que o máximo divisor comum entre os dois é 1).

  Escolhemos a = 17.

- Vamos encontrar agora d, que é o inverso multiplicativo de a e o módulo de $\phi(n)$. Ou seja $d \times a \equiv 1 \bmod \phi(n)$.

  Seja d = 2753, como $17 \times 2753 \equiv 1 \bmod 3120$.

- A nossa chave pública é composta por (a,n).
- A chave privada é composta por (d,n).

2. **Encriptação**

   Seja a nossa mensagem M usando as chaves anteriores

   - Começamos por converter a nossa mensagem M em um número m, tal que este tem que ser menor que n, sendo que caso a mensagem M não seja menor que n temos que a dividir em blocos, tendo estes todos o mesmo tamanho.
   - O valor cifrado vai ser calculado a partir da seguinte fórmula.

$$c \equiv m^a \bmod n$$

   Seja m = 65 da nossa mensagem M então,

$$c \equiv 65^{17} \bmod 3233 = 2790$$

   Então o valor cifrado é 2790.

3. **Desencriptação**

   Agora no nosso passo final vamos descodificar a nossa cifra a usando a nossa chave privada (d,n)

   - Vamos calcular a nossa mensagem original M usando m pelo processo anteriormente mencionado, sendo que agora aplicamos o inverso.

$$m \equiv c^d \text{ mod n}$$

   Seja o nosso valor de a = 2790:

$$m \equiv 2790^{2753} \bmod 3233 = 65$$

   Como podemos ver, voltamos a recuperar o nosso valor de m = 65.

Então para resumir, neste processo criamos uma chave pública, para enviar uma mensagem encriptada, e que apenas quem possuir a chave privada a irá conseguir desencriptar. A cifra *RSA* veio por este meio dificultar a desencriptação da nossa mensagem, pois encontrar os fatores primos de n, descobrir o p e q, identificando a chave privada, é um problema de grande complexidade computacional.

Capítulo 4

Algoritmo de Shor

4.1 História

Os sistemas de chave pública são uma componente muito importante dos sistemas de comunicação atuais, mais especificamente a cifra *RSA*. Em 1980, Richard Feynman e um grupo de investigadores, começaram a especular sobre a possibilidade da criação de um computador quântico, e assim tirar proveito das suas novas propriedades. Eles especularam que quando essa máquina fosse criada, iria conseguir resolver problemas de grande complexidade, que não são possíveis de resolver facilmente com computadores clássicos, tal como o quebrar da cifra *RSA*. Em 1990, David Deutsch formalizou a ideia de existir um computar quântico capaz de realizar qualquer cálculo computacional, fornecendo assim uma base teórica para a computação quântica. Peter Shor, o autor do tema abordado neste documento, em 1994 propôs um algoritmo que iria resolver o problema de fatorização de números inteiros de uma forma mais rápida. Este algoritmo combina a teoria dos números, com os princípios da mecânica quântica, assim como a transformada de Fourier quântica, para encontrar os fatores primos de um número em tempo polinomial. Este novo algoritmo seria capaz de decifrar a cifra *RSA* e pôr em causa a sua segurança e assim os sistemas de comunicação atualmente conhecidos. Contudo, os computadores quânticos existentes ainda não têm capacidade de *qubits* (as unidades de informação quântica) para conseguir decifrar uma mensagem. Isto é claramente um problema para o futuro, visto que os avanços feitos nesta área têm sido de grande escala.

4.2 Fundamentos de Computação Quântica

Para compreendermos como um computador quântico funciona, primeiro precisamos de perceber como funciona um computador clássico .

4.2.1 Computação Clássica

Os computadores clássicos armazenam informações usando elementos de memória que guardam $bits$, as menores unidades de informação. Cada bit pode ter um dos dois estados, 0 ou 1. Um conjunto de 8 $bits$ forma um $byte$, que pode representar $2^8 = 256$ combinações distintas.

Por exemplo, o $byte$ 00110110 corresponde ao número 54.

As operações básicas nos computadores clássicos são realizadas através de **portas lógicas** (como *AND*, *OR*, *NOT*, *XOR* e *CNOT*), que manipulam os *bits* e permitem a realização de cálculos e tomada de decisões.

Porta lógicas ou circuitos lógicos

As portas lógicas desempenham funções básicas usando a álgebra booleana, em que cada porta realiza uma certa operação, representa-se como 0 (falso) e 1 (verdadeiro) [1].

Porta AND :

Saída é 1 se todas as entradas forem 1. Tabela de verdade:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Porta OR :

Saída é 1 se pelo menos uma entrada for 1. Tabela de verdade:

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Porta NOT :

Inverte a entrada. Tabela de verdade:

A	$\neg A$
0	1
1	0

Operação XOR (OU exclusivo) :

Retorna 1 apenas se os dois *bit*s forem diferentes.

Tabela verdade:

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Porta CNOT (Controlled-NOT) :

Inverte o segundo *bit* apenas se o primeiro *bit* for 1.

[1] https://www.alura.com.br/artigos/portas-logicas-tipos-caracteristicas

Tabela verdade:

A	B	$A, A \oplus B$
0	0	0, 0
0	1	0, 1
1	0	1, 1
1	1	1, 0

Logo as portas lógicas estão na base da computação, em que a partir destas operações os computadores conseguem criar e processar tudo que realizamos de forma digital.

4.2.2 Computação Quântica

Como vimos anteriormente um computador clássico armazena informação em *bit*s, dos quais podem ter os valores 0 ou 1 a eles associados. Na computação quântica isto já não se mantêm, pois, a unidade de armazenamento de um computador quântico é em *qubits*.

qubit

O *qubit* é a unidade de informação de um computador quântico correspondente ao *bit* num computador clássico, em que a grande diferença entre os dois é que o *qubit* pode estar em ambos os estados (0 ou 1).

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Em que | x ⟩ é chamado de *ket* e representa o estado dos *qubits* no sistema quântico, ou seja cada *ket* pode representar um número binário, número decimal ou uma sequência de *bit*s.

Exemplos de *ket*:

- | 0 ⟩ - o *qubit* está num estado de 0
- | 1 ⟩ - o *qubit* está num estado de 1
- | 00 ⟩ - ambos *qubits* estão num estado 0
- | 01 ⟩ - o primeiro está num estado de 0 e o segundo 1
- | x ⟩ - representa qualquer estado em que x pode representar uma combinação de *qubits*
- | ϕ ⟩ = α| vivo ⟩ + β| morto ⟩, no exemplo do gato de Schrödinger

Sobreposição Quântica

Cada *qubit* pode estar numa sobreposição quântica, isto significa que pode representar o valor 0 ou 1 simultaneamente. Erwin Schrödinger para explicar como isto é possível usou uma analogia chamada "O gato de Schrödinger", que consiste em colocar um gato dentro de uma caixa e quando a fechamos, pode ou não soltar um veneno mortal. Assim sendo, quando fechamos o gato, pode estar vivo ou não. Então, a nossa sobreposição quântica indica-nos que o gato toma dois estados de vida, estado "morto vivo" e só sabemos o verdadeiro estado quando abrimos a caixa. Matematicamente este estado é escrito da seguinte forma [13]:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

- $|\psi\rangle$ - representa o estado de um *qubit*.
- α e β - são coeficientes complexos que determinam a sobreposição quântica.
- $|0\rangle$ e $|1\rangle$ - são os estados básicos do *qubit*.

Devido a esta sobreposição quântica, um *qubit* toma vários estados ao mesmo tempo, o que implica que o computador explora vários resultados ao mesmo tempo, este processo é chamado como **paralelismo quântico** [13]

Entrelaçamento Quântico

Outro facto importante da computação quântica, é o entrelaçamento entre *qubits*. Isto é, a informação de um *qubit* está relacionada com outro *qubit*, ou seja, o valor de um *qubit* determina o estado de outro.

Uma consequência deste entrelaçamento quântico é a **correlação instantânea** entre *qubits*, isto significa que conseguimos obter alguns resultados instantaneamente.

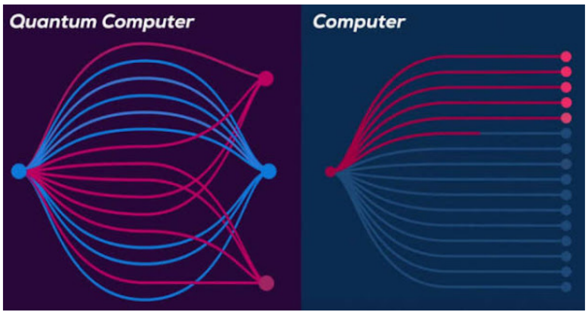

Fig. 4.1 Diferença entre computação quântica e clássica[2]

4.3 Fundamentos

Teorema 4 (Transformada de Fourier) *Para percebermos como funciona o algoritmo de Shor, primeiro precisamos de perceber o que é a transformada de Fourier*[3] *que é dada pela seguinte fórmula:*

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$$

[2] Fonte: Adaptado de Brasil Acadêmico (2015). Disponível em: https://blog.brasilacademico.com/2015/12/computadores-quanticos-explicados.html

[3] https://mathworld.wolfram.com/FourierTransform.html

Em que:

- $F(\omega)$ é a função transformada no domínio da frequência,
- $f(t)$ função no domínio do tempo
- ω é a frequência angular (em radianos por segundo),
- $e^{-i\omega t}$ é o termo exponencial complexo que decompõe $f(t)$ em componentes de diferentes frequências.

Seja um sinal $f(t)$, a transformada de Fourier decompõe esse sinal em uma soma de ondas sinusoidais de diferentes frequências. Isso permite analisar as frequências presentes no sinal, bem como a amplitude de cada uma dessas frequências. Um exemplo é no caso de uma música, a Transformada de Fourier pode ser usada para decompor a música na sua frequência, permitindo assim a observação das suas amplitudes e das diferentes frequências que compõem o som. Uma característica importante da Transformada de Fourier é que pode ser aplicada a qualquer função, desde que a função seja suficientemente bem comportada, ou seja, tem que ser integrável.

Teorema 5 (Inversa da Transformada de Fourier) *A transformada de Fourier possui uma inversa que podemos usar para reconstruir a frequência original, chama-se Inversa da Transformada de Fourier, que a sua fórmula é dada por [8]:*

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{-i\omega t}d\omega$$

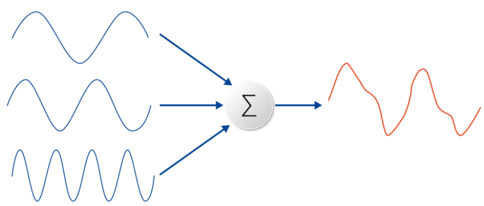

Fig. 4.2 Decomposição de uma frequência[4]

Teorema 6 (Transformada de Fourier Discreta) *Para um conjunto de dados usamos a transformada de Fourier discreta, em que a sua fórmula matemática é dada por [8]:*

$$F(k) = \sum_{n=0}^{N-1} f(n)e^{\frac{-i2\pi kn}{N}}$$

Em que:

- N número total de pontos no nosso conjunto

[4]Fonte: Adaptado de Svantek (s.d.). Disponível em: https://svantek.com/pt/academia/transformada-rapida-de-fourier-fft/

- k representa a posição na frequência
- n representa a posição da série, sendo que vai percorrer todas as posições

Resumidamente, a Transformada de Fourier é usada quando estamos a trabalhar com funções contínuas. A Transformada de Fourier discreta é computacionalmente implementável daí ser usada quando temos um conjunto de dados que queremos analisar, tal como acontece no algoritmo de Shor.

Teorema 7 (Transformada de Fourier Quântica (TFQ)) *A transformada de Fourier Quântica é uma versão da transformada de Fourier Discreta, anteriormente mencionada, e esta é fundamental no algoritmo de Shor, pois permite identificar padrões na periodicidade. A sua fórmula é a seguinte [1]:*

$$TFQ(|x\rangle) = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{\frac{2\pi ixk}{2^n}} |k\rangle$$

Onde:

- $|x\rangle$ representa o estado de entrada,
- $|k\rangle$ são os estados base da sobreposição quântica na saída,
- $e^{\frac{2\pi ixk}{2^n}}$ são os coeficientes de fase complexos.

Com a TFQ conseguimos construir um circuito quântico precisando apenas $\frac{n(n-1)+2}{2}$ portas quânticas. Este processo será dividido em duas etapas.

A primeira etapa será reescrever a operação em uma forma equivalente, de forma a ser mais facilmente implementada por um circuito.

A segunda etapa será construir o circuito capaz de realizar a transformação.

Teorema 8 (Inversa da Transformada de Fourier Quântica (TFQ^T)) *A inversa da Transforma de Fourier transforma um estado quântico de n qubits $|X\rangle_n$ em outro estado quântico de n qubits $|x\rangle_n$, sendo que é definida por [7]:*

$$|x\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{-2\pi ixk/2^n} |k\rangle.$$

Definição 7 (Vetor Unitário (U)) *O operador unitário U é uma matriz que satisfaz a prioridade de [13]:*

$$U^T U = UU^T = I$$

E que U^T é a matriz transposta de U, e I corresponde à matriz identidade, os operadores unitários são muito importantes pois preservam a norma dos vetores, porque os estados quânticos devem ter sempre norma 1 para representar probabilidades válidas, isto vai ser importante para nós pois com este vetor unitário vamos conseguir transformar os estados quânticos da nossa base para uma frequência, e assim aplicar a TFQ.

Definição 8 (Porta de Hadamard) *A porta de Hadamard é uma operação fundamental na computação quântica, que transforma o estado de um qubit em sobreposição quântica, permitindo assim que a mesma probabilidade de ser medido nos estado $|0\rangle$ e $|1\rangle$. A porta de Hadamard é representada pela seguinte matriz 2×2 [15]:*

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Se aplicarmos a porta de Hadmard aos seguintes *qubits*, $|0\rangle$ e $|1\rangle$, terá o seguinte comportamento:
Para o estado $|0\rangle$:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

Para o estado $|1\rangle$:

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Conceito de Fase

A fase é um conceito fundamental no contexto da mecânica quântica e está relacionada com a forma como os estados quânticos evoluem sob a ação de operadores unitários U. Dado um operador unitário U e um estado quântico $|\psi\rangle$, se multiplicarmos U sobre $|\psi\rangle$ [11]:

$$U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$$

Onde θ é uma constante real chamada de fase.

No caso do algoritmo de Shor, a fase está relacionada com o período r da função modular $f(x) = a^x \mod N$, que queremos determinar. A fase θ está diretamente associada ao inverso do período r, o que nos permite, determinar o período r de uma função a partir da fase, e assim, fatorizar números de grande dimensão.

Estado Inicial e Operador Unitário

Suponha que o número a seja coprimo com N (ou seja, $\text{mdc}(a,N) = 1$), e queremos encontrar o período r da função $f(x) = a^x \mod N$. A ideia é construir um operador unitário U que, quando aplicado a um estado $|x\rangle$, produza [10]:

$$U|x\rangle = |a^x \mod N\rangle$$

O operador U transforma o estado $|x\rangle$ em $|a^x \mod N\rangle$, o que é a forma geral da operação que queremos realizar. Outra variação desta fórmula, da qual também é importante para determinar o período é:

$$U^r_{a,N}|1\rangle = |a^r mod(N)\rangle = |1\ mod(N)\rangle$$

Seja o seguinte exemplo com N = 15 e y = 7

k	$7^k \mod 15$
0	1
1	7
2	4
3	13
4	1
5	7
6	4
7	13
8	1
9	7
10	4
11	13
12	1
13	7
14	4

Os resultados seguem um padrão cíclico: $1, 7, 4, 13$, com um período de 4, como esperado.

Teorema 9 (Teorema da Estimação de Fase Quântica (EFQ)) *Se U é um operador unitário tal que $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$, onde θ é a fase que desejamos estimar e $|\psi\rangle$ é um vetor próprio de U, então, após aplicar a Estimação de Fase Quântica (EFQ) em um estado de entrada preparado adequadamente, a fase θ pode ser estimada com precisão arbitrária. Em particular, a precisão do estimador depende do número de qubits no registo de controle do EFQ [2].*

Período e Fase

O valor r, o período da função, está relacionado com a fase θ. A fase estimada pelo EFQ está entre $0 \leq \theta < 1$, e a fase θ está relacionada a uma fração simples [13]:

$$\theta = \frac{s}{r}$$

Onde s é um valor inteiro (o numerador) e r é o período que queremos encontrar (o denominador). O EFQ permite-nos estimar θ com precisão suficiente para determinar r de maneira eficiente.

Este teorema garante-nos que, dado um operador unitário U e um estado $|\psi\rangle$ tal que $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$, podemos usar o EFQ para estimar a fase θ com eficiência e precisão. Essa estimativa da fase é crucial para o algoritmo de Shor, pois permite descobrir o período r da função $f(x) = a^x \mod N$ a partir da relação $\theta = \frac{s}{r}$.

4.4 Algoritmo de Shor

Agora já adquirimos os conhecimentos necessários para percebermos o algoritmo de Shor, vamos aplica-lo com os respetivos valores de $N = 15$ (3×5).

4.4.1 Sem o Uso da Computação Quântica

1. Começamos por escolher um m aleatório entre $1 < m < N$, escolhi $m = 7$.
2. Agora vamos verificar se m já é um fator de N, calculando o mdc$(m, N) \neq 1$, então mdc(m, N) não é um fator trivial de N. Seja $m = 7$ e $N = 15$.

$$\text{mdc}(7,15) = 1$$

   Logo, 7 não é um fator de 15.

3. Vamos calcular o período(r) da função $f(x) = m^x \bmod N$, usando a propriedade dos números em aritmética modular.

$$m^r \equiv 1 \pmod{N}$$

   Conseguimos encontrar o nosso r com o uso da transformada de Fourier quântica. Mas neste caso, vou mostrar o cálculo como se fosse na computação clássica.

   Seja $7^x \bmod 15$:

   - $7^1 \bmod 15 = 7$
   - $7^2 \bmod 15 = 4$
   - $7^3 \bmod 15 = 13$
   - $7^4 \bmod 15 = 1$
   - $7^5 \bmod 15 = 7$
   - $7^6 \bmod 15 = 4$
   - $7^7 \bmod 15 = 13$

   Podemos ver que o nosso r é 4, pois é o valor com que o ciclo se repete. Como estamos a falar de um exemplo bastante simples, conseguimos descobrir rapidamente o período da nossa função. Mas quando estamos a tratar de uma cifra *RSA* em que temos a multiplicação de dois números primos de grande dimensão, este cálculo num computador clássico torna-se muito difícil, daí a necessidade de uma forma de aceder a um computador quântico para podermos obter o mesmo resultado de forma mais eficiente.

4. No próximo passo vamos verificar se o nosso r é par e que $m^{\frac{r}{2}} \not\equiv -1 \pmod{N}$

   No nosso exemplo $r = 4$ é par. Então calculamos $m^{\frac{r}{2}}$.

$$7^{\frac{4}{2}} = 7^2 \equiv 4 \pmod{15}$$

   Como podemos ver as condições verificam-se pois, $4 \not\equiv -1 \mod 15$.

5. Calculamos os fatores de N, usando o teorema que nos diz $m^{\frac{r}{2}}+1$ e que $m^{\frac{r}{2}}-1$, normalmente partilham fatores com N. Calculamos então o nosso mdc($m^{\frac{r}{2}}-1, 15$) com mdc($m^{\frac{r}{2}}+1, N$) .

$$mdc(4-1,15) = mdc(3,15) = 3$$

$$mdc(4+1,15) = mdc(5,15) = 5$$

   Então encontramos os nossos dois fatores, 3 e 5.

6. A partir da factorização de n nos seus dois factores primos, 15 = 3 × 5, é fácil obter a chave privada e deste modo quebrar a cifra *RSA*.

4.4.2 Com o uso da Computação Quântica

O exemplo anteriormente apresentado do algoritmo de Shor, descobrimos o período da função de uma forma pouco otimizada, pois caso o nosso N seja grande, um computador clássico demoraria muito tempo a descobrir os fatores de N. No exemplo seguinte vamos descobrir o período usando as propriedades de computador quântico. Reformulando assim o passo 3.

Vamos então encontrar o período r da função

$$f(x) = 7^x \mod 15.$$

Inicialização do Sistema Quântico

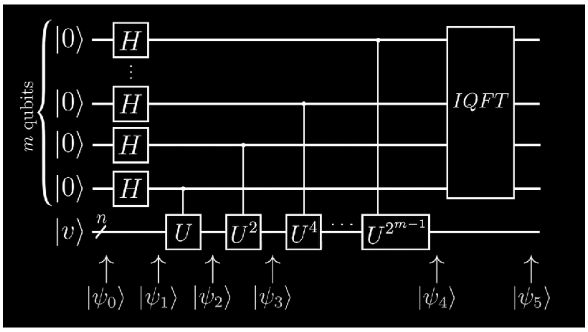

Fig. 4.3 Circuito Quântico[5]

No seguinte circuito quântico podemos ver como será descoberto o valor da fase [13].

Começamos por aplicar as portas de Hadamard a todos os *qubits* $|0\rangle$ para criar uma superposição quântica e assim poderem tomar qualquer valor, o *qubits* $|v\rangle$ será usado para guardar o valor da fase.

No nosso primeiro valor de ψ temos [19]:

$$|\psi_0\rangle = |0\rangle^m |v\rangle$$

Multiplicando agora pelas portas de Hadamard temos:

[5]Fonte: Adaptado de aula "Quantum Phase Estimation". Disponível em: https://www.youtube.com/watch?v=Ex96GyRIFes

$$|\psi_1\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\right)\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\right)\otimes\cdots\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\right)|v\rangle$$

Aplicando agora o nosso vetor unitário U para transformarmos o nosso problema de fatorização em um problema de periodicidade.

$$|\psi_2\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\right)\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\right)\otimes\cdots\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+e^{i\theta}|1\rangle)\right)|v\rangle$$

E assim sucessivamente até obtermos:

$$\begin{aligned}|\psi_4\rangle = \quad & \left(\frac{1}{\sqrt{2}}(|0\rangle+e^{2^{m-1}i\theta}|1\rangle)\right)\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+e^{2^{m-2}i\theta}|1\rangle)\right)\otimes\cdots\otimes\\ & \left(\frac{1}{\sqrt{2}}(|0\rangle+e^{2i\theta}|1\rangle)\right)\otimes\left(\frac{1}{\sqrt{2}}(|0\rangle+e^{i\theta}|1\rangle)\right)|v\rangle\end{aligned}$$

Sabendo então que $\theta = 2\pi j$, tal que $j = 0.j_0 j_1 j_2 \ldots j_{m-1}$ onde $j_i \in \{0,1\}$, como j é um número decimal, podemos transformar para um número fracionário de base 2^n, vamos então substituir por $j = \frac{j_0}{2} + \frac{j_1}{4} + \ldots + \frac{j_{m-1}}{2^m}$ [13]:

$$\begin{aligned}|\psi_4\rangle \quad = \quad & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\cdot 2^{m-1}\left(\frac{j_0}{2}+\frac{j_1}{2^2}+\cdots+\frac{j_{m-1}}{2^m}\right)}|1\rangle\right)\right)\otimes\\ & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\cdot 2^{m-2}\left(\frac{j_0}{2}+\frac{j_1}{2^2}+\cdots+\frac{j_{m-2}}{2^{m-1}}\right)}|1\rangle\right)\right)\otimes\cdots\otimes\\ & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\cdot\left(\frac{j_0}{2}\right)}|1\rangle\right)\right)|v\rangle\end{aligned}$$

Distribuindo agora o valor de 2^{m-1} obtemos:

$$\begin{aligned}|\psi_4\rangle \quad = \quad & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\left(2^{m-2}j_0+j_1+\cdots+j_{m-2}+\frac{j_{m-1}}{2^m}\right)}|1\rangle\right)\right)\otimes\\ & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\left(\frac{j_0}{2}+\frac{j_1}{2^2}+\cdots+\frac{j_{m-2}}{2^{m-1}}\right)}|1\rangle\right)\right)\otimes\cdots\otimes\\ & \left(\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2\pi i\left(\frac{j_0}{2}\right)}|1\rangle\right)\right)|v\rangle\end{aligned}$$

Como este valores $2^{m-2}j_0 + \ldots + j_{m-2}$ serão inteiros, então assim, podemos exclui-los pois vão ser múltiplos de 2π, ficando assim com:

$$
\begin{aligned}
|\psi_4\rangle \quad = \quad & \left(\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i\left(\frac{j_{m-1}}{2}\right)}|1\rangle\right)\right) \otimes \\
& \left(\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i\left(\frac{j_{m-2}}{2}+\frac{j_{m-1}}{2^2}\right)}|1\rangle\right)\right) \otimes \cdots \otimes \\
& \left(\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i\left(\frac{j_0}{2}+\frac{j_1}{2^2}+\cdots+\frac{j_{m-1}}{2^m}\right)}|1\rangle\right)\right) |v\rangle
\end{aligned}
$$

Podemos observar que o estado obtido, seria o mesmo se aplicasse-mos a transformada Quântica de Fourier ao estado j. Então aplicando a inversa da transformada de Fourier obtendo o nosso estado j.

$$|\psi_5\rangle = QFT^T|\psi_4\rangle = |j\rangle$$

Então agora que já temos o nosso valor de j, já conseguimos descobrir o valor da nossa fase, tal que $\theta = 2\pi j$, e assim que soubermos o período conseguimos descobrir facilmente os fatores de qualquer número, sendo por exemplo $\theta = 0,25$, então usando o teorema de funções contínuas obtemos a seguinte fração, $s = \frac{1}{4}$, assim podemos dizer que temos um período $r = 4$.

Como 4 é par e $7^{4/2} \not\equiv \pm 1 \pmod{15}$, o algoritmo funcionou, vamos agora descobrir os nossos fatores:

$$\text{mdc}(7^2 - 1, 15) = 3 \quad \text{e} \quad \text{mdc}(7^2 + 1, 15) = 5$$

é possível concluir que:

$$15 = 3 \cdot 5$$

Capítulo 5

Biblioteca qiskit

Este capítulo descreve os passos necessários para aceder às bibliotecas do qiskit, que nos permitirão programar em *python* utilizando *qubits* e, assim, executar o algoritmo de Shor. Sendo que o qiskit é uma biblioteca criada pela *IBM* (International Business Machines Corporation) que permite criar, simular e executar algoritmos quânticos em computadores quânticos ou em simuladores.

Começamos por abrir o *Google colab*,[1] e criamos um novo *notebook*. De seguida escrevemos as seguintes linhas de código.

```
!pip install qiskit --quiet
!pip install pylatexenc --quiet
!pip install matplotlib --quiet
!pip install qiskit-aer --quiet
from qiskit.utils import Quantum Instance
from qiskit import QuantumCircuit, Aer, execute
from qiskit.tools.visualization import plot_histogram
import numpy
```

Em que cada uma terá a seguinte finalidade:

- `pip`: Package de instalação do *python*, para gerir *packages* e instalar as versões mais atualizadas
- `quiet`: Para não aparecer na linha de comandos as instalações
- `!pip install qiskit -quiet`: Instala a biblioteca do qiskit (de computação quântica)
- `!pip install pylatexenc -quiet`: Instala a biblioteca para ler fórmulas em Latex
- `!pip install matplotlib -quiet`: Instala a biblioteca para fazer gráficos
- `!pip install qiskit-aer -quiet`: Instala módulo Aer do qiskit para simular circuitos quânticos
- `from qiskit.utils import QuantumInstance`: Permite configurar o ambiente de execução quântico

[1] https://colab.google/, Google Colaboratory, Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.

- `from qiskit import QuantumCircuit, Aer, execute`: Importa módulos do qiskit para executar circuitos quânticos usando Aer
- `import numpy`: Importa a biblioteca de operações matemáticas
- `from qiskit.tools.visualization import plot_histogram`: Importa a função para visualizar os resultados

Apresento um simples exemplo de um circuito quântico, conforme mostrado na figura 5.1.

```
from qiskit import QuantumCircuit

qc=QuantumCircuit(2)

qc.h(0)
qc.cx(0,1)

qc.draw(output='mpl')
```

- `qc=QuantumCircuit(2)`- criamos um circuito quântico com 2 *qubits*
- `qc.h(0)`- aplica a porta de Hadamard ao primeiro *qubit*(0)
- `qc.cx(0,1)`- Aplica uma porta *CNOT* entre os dois *qubits*, onde o *qubit* 0 é o *qubit* de controle, e o *qubit* 1 é o *qubit* alvo. Como o *qubit* 0 está em superposição, essa operação cria um estado emaranhado
- `qc.draw(output='mpl')`- desenha o circuito

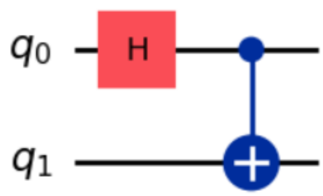

Fig. 5.1 Circuito criado.

De seguida, criamos uma conta no site da *IBM Quantum* (https://quantum.IBM.com/), onde teremos o nosso *API Token* presente, mostrado na figura 5.2, que podemos usar para aceder aos computadores da *IBM*.

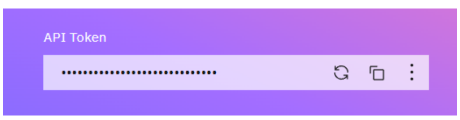

Fig. 5.2 *API Token* da *IBM Quantum*.

Após obtermos o nosso *API Token* da *IBM*, podemos aceder aos computadores disponíveis. No site, em *Compute Resources*, temos a lista de todos os computadores existentes. Para selecionar os computadores disponíveis, podemos verificar a barra *All Instances*, como mostrado na Figura 5.4:

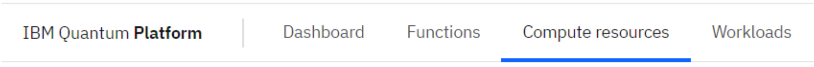

Fig. 5.3 Computadores da *IBM*.

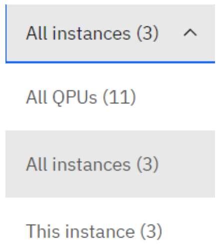

Fig. 5.4 Computadores disponíveis da *IBM*.

Nas próximas linhas de código (Listagem 5.1), apresento como usar o *API Token* da *IBM* para aceder a um computador quântico e verificar o número de *qubits* disponíveis.

```
from qiskit_IBM_runtime import QiskitRuntimeService

service = QiskitRuntimeService(
    channel="IBM_quantum",  # ou "IBM_cloud"
    token="******"
)

backend = service.backend(name="IBM_brisbane")
print(backend.num_qubits)  # Exemplo: 127
```

Listing 5.1 Uso do *API Token* da *IBM* para aceder a computadores quânticos

Capítulo 6

Código do Algoritmo de Shor

Neste capitulo vamos implementar o algoritmo de Shor em *python* na computação clássica, explicar as suas limitações e demonstrar como a computação quântica consegue resolver problemas de fatorização num tempo de processamento menor. Uma das grandes dificuldades que encontrei para aplicar o algoritmo de Shor, foi a questão de grande parte das bibliotecas apresentadas, usarem bibliotecas que já foram descontinuadas.

6.1 Código do Algoritmo de Shor na computação clássica

```python
    import math
    import random
```

Começamos por importar a biblioteca `math` para podermos usar funções matemáticas, e a biblioteca `random` para escolher valores aleatórios.

Função para verificar se 2 números são coprimos

```python
def mdc(a,b):
    while b !=0:
      a, b = b,a % b
    return a
```

A função calcula o máximo divisor comum entre dois números a partir do algoritmo de Euclides, devolvendo assim o maior divisor com um e confirmando se são coprimos.

Cálculo do período

```python
def find_period(x, N):
    r = 1
    while pow(x, r, N ) != 1:
        r += 1
    return r
```

Esta função é responsável por encontrar o período r de $x \mod N$, que é o menor inteiro r tal que $x^r \equiv 1 \pmod{N}$, devolvendo assim o valor do período.

Algoritmo de Shor

```
def shor_classical(N):
    # Verificar se N par
    if N % 2 == 0;
        return 2, N // 2

    # Escolher x aleatoriamente
    x = random.randint(2, N - 1)

    #Garantir que x e N sao coprimos
    while mdc(x, N) != 1:
        x = random.randint(2, N - 1)

    # Encontrar o periodo r
    r = find_period(x, N)
    # Verificar se r par
    if r % 2 != 0:
        return None

    # Calcular os fatores de N
    p = mdc(pow(x, r // 2) - 1, N)
    q = mdc(pow(x, r // 2) + 1, N)

    if p * q == N:
    #Verificar se os fatores estao corretos
        return p,q
    return None
```

Esta função é uma aplicação clássica do algoritmo de Shor para podermos encontrar os fatores de N.

Começa por verificar que o valor que queremos fatorizar N não é par, de seguida escolhe-se aleatoriamente o valor x, que é coprimo com N, após isto se verificar usamos a função *find_period* para descobrirmos o período da nossa função, caso o r não seja par a função repete. Sabendo o período podemos começar o cálculo para descobrir os fatores do N, $(p = \gcd\left(x^{r/2} - 1, N\right))$ e $(q = \gcd\left(x^{r/2} + 1, N\right))$, depois verificamos estes multiplicando-os e obtendo o nosso N, caso não se verifique, a fatorização não foi bem sucedida.

Imprimir

```
# Exemplo de uso
N = 15 # Numero a ser fatorizado
result = shor_classical(N)
```

```

# Verificar se o resultado nao e vazio antes de aceder aos indices
if result is now None;
    print(f"Os fatores de {N} sao: {result[0]} e {result[1]}")
    print(f"O periodo e: {result[2]}") #Agora imprimimos corretamente
        o valor do periodo
else:
    print(f"Nao foi possivel fatorizar {N} com este metodo")
```

Aqui escolhemos o valor N que pretendemos fatorizar, aplicamos o algoritmo de Shor anteriormente mencionado, imprimimos os fatores de N e o seu período r.

Observações:

Este código apresentado tem as suas limitações, se o valor de N for grande, então o algoritmo demora muito tempo a processar o valor do período. Outro problema é o caso do nosso x ser escolhido aleatoriamente, isto significa que para o mesmo N podemos ter tempos diferentes de processamento, ou seja o nosso sucesso é escolhido aleatoriamente pelo valor escolhido de x.

6.2 Código do Algoritmo de Shor em Computação Quântica

Vamos agora explicar como é feita a aplicação prática do algoritmo de Shor na computação quântica. A principal diferença em relação à abordagem clássica está na descoberta da fase, tal que esta é fundamental para a determinar do período da função modular. Esse período é utilizado para obter os fatores do número a ser decomposto.

De seguir, apresenta-se o código utilizado, com destaque para as funções principais e suas respetivas finalidades:

```
    def check_if_power(N)
```

- N - número que pretendemos fatorizar
- Função - Verifica se N pode ser escrito como uma potência a^b, onde a e $b > 1$ são inteiros. Retorna True se for o caso, o que permite fatorizar N diretamente

```
    def get_value_a(N)
```

- Função - encontra e devolve um valor a tal que $1 < a < N$, em que a seja coprimo com N

```
    def get_factors(x_value,t_upper,N,a)
```

- *x_value* - valor medido no registo
- *t_upper* - nº de *qubits* no registo
- Função - Estima o período da função $f(x) = a^x \bmod N$ com frações contínuas e tenta usar esse período para calcular fatores de N. Retorna os fatores encontrados ou False se não for possível

```
def egcd(a, b)
```

- *b* - Neste caso será o módulo da função modular
- Função - Algoritmo de Euclides, será essencial no calculo do inverso modular de a módulo b. Este inverso é fundamental para reverter operações do circuito quântico durante a multiplicação modular, como as operações quânticas devem ser reversíveis, ao multiplicar por a, é necessário depois multiplicar por $a^{-1} \mod b$ para restaurar o estado original dos registos. Esta função retorna valores que permitem determinar esse inverso

```
def modinv(a, m)
```

- *m* - módulo da função modular
- Função - Calcula o inverso modular de a módulo m, utilizando o Algoritmo de Euclides. Esta função permite inverter operações como a multiplicação modular. A função retorna o valor de x tal que $ax \equiv 1 \mod m$, desde que $\gcd(a,m) = 1$. Caso contrário, o inverso não existe e a função gera um erro

```
def create_QFT(circuit,up_reg,n,with_swaps)
```

- *circuit*: circuito quântico criado pelo qiskit
- *up_reg*: registro de *qubits*
- *n*: número de *qubits* usados
- *with_swaps*: indica se deve-se inverter a ordem dos *qubits* usados
- Função: Cria a transformada de Fourier Quântica, sem devolver nada apenas alterando o circuito

```
def create_inverse_QFT(circuit,up_reg,n,with_swaps)
```

- Função - Cria a transformada de Fourier Quântica inversa, usada para retornar do domínio da frequência ao domínio computacional, modificando o circuito diretamente

```
def getAngles(a,N)
```

- Função - Converte o número a em ângulos de fase para serem aplicados como rotações quânticas no circuito

```
def ccphase(circuit,angle,ctl1,ctl2,tgt)
```

- *angle* - ângulo de rotação
- *ctl1, ctl2* - *qubits* de controlo
- *tgt* - *qubit* alvo

- Função - Aplica uma rotação controlada por dois *qubits* sobre o *qubit* alvo

```
    def phiADD(circuit,q,a,N,inv)
```

- *q* - registo de *qubits*
- *a* - Valor a ser somado
- *inv* - Se verdadeiro, aplica a operação inversa.
- Função - Realiza a adição de *a* ao registo *q*, em módulo *N*, usando fases

```
    def cphiADD(circuit,q,ctl,a,n,inv)
```

- Função - Versão controlada da função *phiADD*. Só soma se o *qubit* de controlo estiver ativo

```
    def ccphiADD(circuit,q,ctl1,ctl2,a,n,inv)
```

- Função - Versão duplamente controlada do *phiADD*

```
    def ccphiADDmodN(circuit, q, ctl1, ctl2, aux, a, N, n)
```

- aux - *qubit* auxiliar
- Função - Realiza uma adição modular de *a* ao registo, controlada por dois *qubits*, mantendo o resultado dentro do módulo *N*

```
    def ccphiADDmodN_inv(circuit, q, ctl1, ctl2, aux, a, N, n)
```

- *a* - valor a ser retirado
- Função - Função inversa da *ccphiADDmodN*, desfazendo a adição modular controlada

```
    def cMULTmodN(circuit, up_reg, aux_reg, a, N, n, ctl)
```

- *aux_reg* - Registo auxiliar (resultado).
- *a* - Valor multiplicador
- *ctl* - *qubit* de controlo
- Função - Executa a multiplicação modular $a * x \mod N$controlada por um *qubit*, gravando o resultado num registo auxiliar e aplicando depois a operação inversa para garantir reversibilidade quântica

```
    def main
```

- Recebe os valores de entrada: número *N* a fatorizar, base *a*, e número de repetições. Cria os circuitos quânticos e executa o algoritmo de Shor, utilizando as funções anteriores

6.3 Tipos de Algoritmos de Fatorização

Vamos agora comparar a fatorização do algoritmo de Shor com outros métodos anteriormente usados, entre eles o método de o método de Fermat e o método do Crivo Quadrático, sendo que vamos prestar maior atenção ao método Crivo Quadrático pois este é o que tem maior capacidade de fatorização. Na seguinte tabela apresentada temos as diferentes complexidades dos métodos apresentados [3, 5]:

Tabela 6.1 Comparação de complexidade dos métodos: Fermat, Crivo Quadrático e Shor

Método	**Tipo**	**Complexidade**
Fermat	Fatorização	$O(\sqrt{n})$
Crivo Quadrático	Fatorização	$O\left(\exp\left(\sqrt{\log n \cdot \log\log n}\right)\right)$
Shor	Fatorização (quântico)	$O(log(n)^3)$

6.3.1 Crivo Quadrático vs Algoritmo de Shor

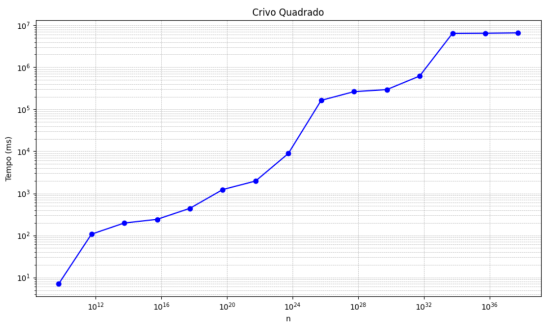

Fig. 6.1 gráfico Crivo Quadrático

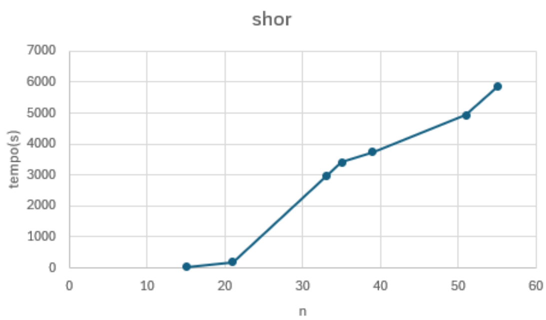

Fig. 6.2 Gráfico algoritmo de Shor

O Crivo Quadrático é um algoritmo de fatorização de inteiros que, embora mais eficiente do que métodos anteriores, como o de Fermat, ainda apresenta limitações significativas ao lidar com números de grande magnitude. Tem uma complexidade subexponencial, sendo estimada em $O\left(\exp\left(\sqrt{\log n \cdot \log\log n}\right)\right)$ [20].

Isso implica que o tempo computacional necessário para fatorizar números cresce rapidamente à medida que aumenta o número de *bits*, o que torna o método inviável em larga escala.

Por outro lado, o algoritmo de Shor é dominado em termos de complexidade pelo cálculo do período r, que é de $O(\log^3 n)$ [13], evidenciando assim a sua superioridade teórica em relação aos métodos clássicos.

Apesar de seu potencial revolucionário, a implementação prática do algoritmo de Shor ainda encontra obstáculos substanciais devido às limitações tecnológicas da computação quântica atual. Com o exemplo apresentado, o maior número fatorizado foi o 55, utilizando um processador quântico com apenas 26 *qubits* . Esse resultado evidencia a diferença entre a complexidade teórica e a capacidade computacional dos dispositivos quânticos. Estima-se que seriam necessários milhões de *qubits*, com níveis adequados de correção de erros, para fatorizar com sucesso um número de 2048 *bits*, um padrão comum em sistemas criptográficos modernos [6].

Dessa forma, embora o algoritmo de Shor representa um marco fundamental na teoria da criptografia quântica, o método Crivo Quadrático ainda se destaca na prática como uma das abordagens mais eficazes e viáveis com os recursos computacionais atualmente disponíveis.

Capítulo 7

Criptografia Pós-Quântica

Agora compreendemos que é necessário tomar medidas para proteger os nossos dados contra a computação quântica, nomeadamente o algoritmo de *Shor*, daí o foco na criptografia Pós-Quântica. A criptografia pós-quântica (*PQC*) que tem como objetivo substituir os algoritmos dos sistemas criptográficos atualmente usados, para proteger dados ou informações contra ataques quânticos. Resumidamente, os algoritmos de *PQC* dependem de equações matemáticas, como criptografia baseada em rede ou multivariada, que são muito difíceis de decifrar com o uso de um computador quântico e clássico.

7.1 Tipos de *Post-Quantum Cryptography*

Seguidamente, são apresentadas algumas abordagens à criptografia pós-quântica [12].

7.1.1 Criptografia Baseada em Redes (*Lattice-based cryptography*)

A criptografia baseada em redes fundamenta-se na dificuldade de resolver problemas geométricos (por exemplo: distância entre dois pontos) em espaços de alta dimensão. Dois problemas notáveis nesta categoria são:

- **Learning With Errors** (LWE): Baseia-se na dificuldade de distinguir entre amostras geradas aleatoriamente e amostras perturbadas por erro num espaço vetorial, porque os erros fazem com que o problema não se torne linearmente reversível, o que dificulta ataques mesmo dos computadores quânticos.

- **Shortest Vector Problem** (SVP): Consiste em encontrar o vetor mais curto numa rede (conjunto de vetores obtidos por combinações lineares inteiras de vetores base), em que quando esta passa uma *dim(n) > 100* o problema é computacionalmente muito difícil de resolver visto que ainda não existem algoritmos para o resolver.

Exemplos de esquemas criptográficos baseados nessa abordagem incluem **Kyber** (LWE) para cifração e **Dilithium** para assinaturas digitais.

7.1.2 Criptografia Baseada em Códigos (*Code-based cryptography*)

Esta abordagem baseia-se na dificuldade de descodificar códigos lineares aleatórios, um problema considerado resistente mesmo perante computadores quânticos. Um dos exemplos mais antigos e ainda hoje considerado seguro é o esquema **McEliece**, que utiliza códigos de *Goppa* (tipo de código corretor de erros) para cifrar mensagens.

Por exemplo, suponha que a mensagem original seja "0 1". Ao aplicar o esquema de **McEliece**, adiciona-se redundância segundo um código de *Goppa*, resultando, por exemplo, na mensagem codificada "1 0 1". Se durante a transmissão ocorrer um erro e a mensagem recebida for "1 1 1", o código de *Goppa* permite detetar e corrigir esse erro, recuperando a mensagem original.

A segurança deste método reside na dificuldade de descodificar mensagens sem conhecimento da chave privada, devido à estrutura oculta do código utilizado.

7.1.3 Criptografia Baseada em Funções hash (*Hash-based cryptography*)

A segurança desta técnica depende da resistência de funções hash criptográficas (função matemática que transforma uma mensagem numa sequência fixa de *bits*). Como não se baseia em problemas como a fatorização ou logaritmos discretos, é considerada resistente a ataques quânticos. Um exemplo de função hash utilizada é o **SHA-3** (Secure Hash Algorithm version 3).

Os esquemas baseados nesta abordagem, como o **SPHINCS+**, são especialmente adequados para assinaturas digitais. No entanto, a principal desvantagem é o tamanho das assinaturas digitais, bem como a complexidade computacional envolvida no processo.

Exemplo de uma função Hash:
Entrada: "mensagem"
Hash: c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31f6e5cda

7.1.4 Criptografia Baseada em Multivariáveis (*Multivariate-based cryptography*)

Baseia-se na complexidade de resolver sistemas de equações polinomiais multivariáveis. Um exemplo desse tipo de criptografia foi o **Rainbow**, que, no entanto, não foi selecionado pelo **NIST** devido a vulnerabilidades descobertas.

7.2 Conclusão

A computação quântica representa um grande desafio para a segurança digital, exigindo o desenvolvimento de novas abordagens criptográficas. O **NIST** (Instituto Nacional de Padrões e Tecnologia) selecionou **Kyber** (LWE) como padrão para encriptação da chave pública e **Dilithium** e **SPHINCS+** para assinaturas digitais. Embora alguns métodos se revelem promissores, a investigação prossegue no sentido de garantir soluções mais seguras e eficientes.

Capítulo 8

Conclusão

Neste trabalho, foi explorada a vulnerabilidade do sistema de criptografia *RSA* face à evolução da computação quântica, com especial destaque para o algoritmo de Shor. Foram introduzidos fundamentos teóricos da computação quântica, para permitir a compreensão do algoritmo de Shor e a sua capacidade disruptiva face aos sistemas criptográficos baseados na fatorização de números primos.

O trabalho incluiu a implementação prática do algoritmo de Shor na computação clássica e na computação quântica. Verificou-se que, embora a computação clássica permita apenas demonstrar o funcionamento do algoritmo para números pequenos, pelo problema de escalabilidade. Por outro lado, os computadores quânticos atuais, embora ainda limitados em número de *qubits* e estabilidade, demonstram um enorme potencial.

Adicionalmente, foi apresentada uma comparação entre diferentes métodos de fatorização, destacando-se a eficiência superior do algoritmo de Shor em termos de complexidade. Também se abordaram brevemente a criptografia pós-quântica, cuja finalidade é resistir a ataques de computadores quânticos.

Com este estudo, pretendeu-se alertar para os riscos reais que a computação quântica representa para a criptografia moderna, e incentivar a comunidade científica a adotar medidas proativas no desenvolvimento e implementações de soluções resistentes à computação quântica.

Como trabalho futuro, propõe-se o aprofundamento da análise de algoritmos de criptografia pós-quântica, com ênfase em sua viabilidade prática e desempenho em diferentes cenários de aplicação. Seria também relevante acompanhar a evolução da computação quântica, nomeadamente o aumento do número de *qubits* lógicos estáveis e a melhoria dos sistemas de correção de erros. Outra linha promissora consiste em explorar a implementação híbrida de algoritmos clássicos e quânticos, procurando transições seguras e eficientes para criptográficos.

Bibliografia

[1] Barenco, A., Ekert, A., Suominen, K.-A., and Törmä, P. (1996). Approximate quantum fourier transform and decoherence. *Physical Review A*, 54(1):139.

[2] Chapeau-Blondeau, F. and Belin, E. (2020). Fourier-transform quantum phase estimation with quantum phase noise. *Signal Processing*, 170:107441.

[3] Cohen, H. (2013). *A course in computational algebraic number theory*, volume 138. Springer Science & Business Media.

[4] Dias, D. M. G. and Rodriguez, J. E. A. (2017). O teorema de Euler e aplicações. *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, 5(1).

[5] Erra, R. and Grenier, C. (2009). The Fermat factorization method revisited. *IACR Cryptol. ePrint Arch.*, 2009:318.

[6] Gidney, C. and Ekerå, M. (2021). How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433.

[7] Hales, L. and Hallgren, S. (2000). An improved quantum fourier transform algorithm and applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 515–525.

[8] Heckbert, P. (1995). Fourier transforms and the fast fourier transform (fft) algorithm. *Computer Graphics*, 2(1995):15–463.

[9] Jones, W. B. and Thron, W. J. (1980). Continued fractions. *Encyclopedia of Mathematics and its Applications*, 11.

[10] Kumar, M. and Mondal, B. (2024). Study on implementation of Shor's factorization algorithm on quantum computer. *SN Computer Science*, 5(4):413.

[11] Manogue, C. A. and Dray, T. (2017). Properties of unitary matrices. Oregon State University.

[12] Micciancio, D. and Regev, O. (2009). Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer.

[13] Nielsen, M. A. and Chuang, I. L. (2010). *Quantum computation and quantum information*. Cambridge university press.

[14] Pottier, L. (1996). The euclidean algorithm in dimension n. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation*, pages 40–42.

[15] Santos, A. C. (2016). O computador quântico da IBM e o IBM quantum experience. *Revista Brasileira de Ensino de Física*, 39.

[16] Sousa, A. N. L. (2013). Criptografia de chave pública, criptografia RSA. Master's thesis, Universidade Estadual Paulista, Brasil.

[17] Spillman, R. J. (2004). *Classical and contemporary cryptology*. Prentice-Hall, Inc.

[18] Tavares, J. N. and Geraldo, Â. (2021). Máximo divisor comum. *Revista de Ciência Elementar*, 9(1).

[19] Veliche, A. (2018). Shor's algorithm and its impact on present-day cryptography. *no. Math*, 4020:1–19.

[20] Yimsiriwattana, A. and Lomonaco Jr, S. J. (2004). Distributed quantum computing: A distributed Shor algorithm. In *Quantum Information and Computation II*, volume 5436, pages 360–372. SPIE.

Anexo A

Code Listings

A.1 Python Example

```

""" Imports from qiskit"""
from collections import Counter
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, transpile
from qiskit_ibm_runtime import QiskitRuntimeService, Session, Sampler
from qiskit_aer import AerSimulator
import sys

from qiskit_ibm_runtime import SamplerV2 as Sampler

""" Imports to Python functions """
import math
import array
import fractions
import numpy as np

import time

start = time.time()

""" Function to check if N is of type q^p"""
def check_if_power(N):

    b=2
    while (2**b) <= N:
        a = 1
        c = N
        while (c-a) >= 2:
            m = int( (a+c)/2 )

```

```
            if (m**b) < (N+1):
                p = int( (m**b) )
            else:
                p = int(N+1)

            if int(p) == int(N):
                print('N is {0}^{1}'.format(int(m),int(b)) )
                return True

            if p<N:
                a = int(m)
            else:
                c = int(m)
        b=b+1

    return False

""" Function to get the value a ( 1<a<N ), such that a and N are coprime """
def get_value_a(N):

    """ ok defines if user wants to used the suggested a (if ok!='0') or not (if ok=='0') """
    ok='0'

    """ Starting with a=2 """
    a=2

    """ Get the smallest a such that a and N are coprime"""
    while math.gcd(a,N)!=1:
        a=a+1

    """ Store it as the smallest a possible """
    smallest_a = a

    """ Ask user if the a found is ok, if not, then increment and find the next possibility """
    ok = input('Is the number {0} ok for a? Press 0 if not, other number if yes: '.format(a))
    if ok=='0':
        if(N==3):
            print('Number {0} is the only one you can use. Using {1} as value for a\n'.format(a,a))
            return a
        a=a+1

    """ Cycle to find all possibilities """
    while ok=='0':
```

```python

        """ Get a coprime with N """
        while math.gcd(a,N)!=1:
            a=a+1

        """ Ask user if ok """
        ok = input('Is the number {0} ok for a? Press 0 if not, other number if yes: '.format(a))

        """ If user says it is ok, then exit cycle, a has been found """
        if ok!='0':
            break

        """ If user says it is not ok, increment a and check if are all possibilites  """
        a=a+1

        """ If all possibilities for a are rejected, put a as the smallest possible value and exit cycle """
        if a>(N-1):
            print('You rejected all options for value a, selecting the smallest one\n')
            a=smallest_a
            break

    """ Print the value that is used as a """
    print('Using {0} as value for a\n'.format(a))

    return a

""" Function to apply the continued fractions to find r and the gcd to find the desired factors"""
def get_factors(x_value,t_upper,N,a):

    if x_value<=0:
        print('x_value is <= 0, there are no continued fractions\n')
        return False

    print('Running continued fractions for this case\n')

    """ Calculate T and x/T """
    T = pow(2,t_upper)

    x_over_T = x_value/T

    """ Cycle in which each iteration corresponds to putting one more term in the
```

```
    calculation of the Continued Fraction (CF) of x/T """

    """ Initialize the first values according to CF rule """
    i=0
    b = array.array('i')
    t = array.array('f')

    b.append(math.floor(x_over_T))
    t.append(x_over_T - b[i])

    while i>=0:

        """From the 2nd iteration onwards, calculate the new terms of
            the CF based
        on the previous terms as the rule suggests"""

        if i>0:
            b.append( math.floor( 1 / (t[i-1]) ) )
            t.append( ( 1 / (t[i-1]) ) - b[i] )

        """ Calculate the CF using the known terms """

        aux = 0
        j=i
        while j>0:
            aux = 1 / ( b[j] + aux )
            j = j-1

        aux = aux + b[0]

        """Get the denominator from the value obtained"""
        frac = fractions.Fraction(aux).limit_denominator()
        den=frac.denominator

        print('Approximation number {0} of continued fractions:'.
            format(i+1))
        print("Numerator:{0} \t\t Denominator: {1}\n".format(frac.
            numerator,frac.denominator))

        """ Increment i for next iteration """
        i=i+1

        if (den%2) == 1:
            if i>=15:
                print('Returning because have already done too much 
                    tries')
                return False
```

```
            print('Odd denominator, will try next iteration of continued fractions\n')
            continue

        """ If denominator even, try to get factors of N """

        """ Get the exponential a^(r/2) """

        exponential = 0

        if den<1000:
            exponential=pow(a , (den/2))

        """ Check if the value is too big or not """
        if math.isinf(exponential)==1 or exponential>1000000000:
            print('Denominator of continued fraction is too big!\n')
            aux_out = input('Input number 1 if you want to continue searching, other if you do not: ')
            if aux_out != '1':
                return False
            else:
                continue

        """If the value is not to big (infinity), then get the right values and
        do the proper gcd()"""

        putting_plus = int(exponential + 1)

        putting_minus = int(exponential - 1)

        one_factor = math.gcd(putting_plus,N)
        other_factor = math.gcd(putting_minus,N)

        """ Check if the factors found are trivial factors or are the desired
        factors """

        if one_factor==1 or one_factor==N or other_factor==1 or other_factor==N:
            print('Found just trivial factors, not good enough\n')
            """ Check if the number has already been found, use i-1 because i was already incremented """
            if t[i-1]==0:
                print('The continued fractions found exactly x_final/(2^(2n)) , leaving funtion\n')
                return False
            if i<15:
```

```
                aux_out = input('Input number 1 if you want to continue searching, other if you do not: ')
                if aux_out != '1':
                    return False
            else:
                """ Return if already too much tries and numbers are huge """
                print('Returning because have already done too many tries\n')
                return False
        else:
            print('The factors of {0} are {1} and {2}\n'.format(N, one_factor,other_factor))
            print('Found the desired factors!\n')
            return True

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m

""" Function to create QFT """
def create_QFT(circuit,up_reg,n,with_swaps):
    i=n-1
    """ Apply the H gates and Cphases"""

    while i>=0:
        circuit.h(up_reg[i])
        j=i-1
        while j>=0:
            if (np.pi)/(pow(2,(i-j))) > 0:
                circuit.cp( (np.pi)/(pow(2,(i-j))) , up_reg[i] , up_reg[j] )
                j=j-1
        i=i-1

    """ If specified, apply the Swaps at the end """
    if with_swaps==1:
        i=0
        while i < ((n-1)/2):
```

```python
            circuit.swap(up_reg[i], up_reg[n-1-i])
            i=i+1

""" Function to create inverse QFT """
def create_inverse_QFT(circuit,up_reg,n,with_swaps):
    """ If specified, apply the Swaps at the beggining"""
    if with_swaps==1:
        i=0
        while i < ((n-1)/2):
            circuit.swap(up_reg[i], up_reg[n-1-i])
            i=i+1

    """ Apply the H gates and Cphases"""

    i=0
    while i<n:
        circuit.h(up_reg[i])
        if i != n-1:
            j=i+1
            y=i
            while y>=0:
                 if (np.pi)/(pow(2,(j-y))) > 0:
                    circuit.cp( - (np.pi)/(pow(2,(j-y))) , up_reg[j]
                        , up_reg[y] )
                    y=y-1
        i=i+1

"""Function that calculates the array of angles to be used in the
    addition in Fourier Space"""
def getAngles(a,N):
    s=bin(int(a))[2:].zfill(N)
    angles=np.zeros([N])
    for i in range(0, N):
        for j in range(i,N):
            if s[j]=='1':
                angles[N-i-1]+=math.pow(2, -(j-i))
        angles[N-i-1]*=np.pi
    return angles

"""Creation of a doubly controlled phase gate"""
def ccphase(circuit,angle,ctl1,ctl2,tgt):
    circuit.cp(angle/2,ctl1,tgt)
    circuit.cx(ctl2,ctl1)
    circuit.cp(-angle/2,ctl1,tgt)
    circuit.cx(ctl2,ctl1)
    circuit.cp(angle/2,ctl2,tgt)
```

```
"""Creation of the circuit that performs addition by a in Fourier
    Space"""

def phiADD(circuit,q,a,N,inv):
    angle=getAngles(a,N)
    for i in range(0,N):
        if inv==0:
            circuit.p(angle[i],q[i])
        else:
            circuit.p(-angle[i],q[i])

"""Single controlled version of the phiADD circuit"""
def cphiADD(circuit,q,ctl,a,n,inv):
    angle=getAngles(a,n)
    for i in range(0,n):
        if inv==0:
            circuit.cp(angle[i],ctl,q[i])
        else:
            circuit.cp(-angle[i],ctl,q[i])

"""Doubly controlled version of the phiADD circuit"""
def ccphiADD(circuit,q,ctl1,ctl2,a,n,inv):
    angle=getAngles(a,n)
    for i in range(0,n):
        if inv==0:
            ccphase(circuit,angle[i],ctl1,ctl2,q[i])
        else:
            ccphase(circuit,-angle[i],ctl1,ctl2,q[i])

"""Circuit that implements doubly controlled modular addition by a"""
def ccphiADDmodN(circuit, q, ctl1, ctl2, aux, a, N, n):
    ccphiADD(circuit, q, ctl1, ctl2, a, n, 0)
    phiADD(circuit, q, N, n, 1)
    create_inverse_QFT(circuit, q, n, 0)
    circuit.cx(q[n-1],aux)
    create_QFT(circuit,q,n,0)
    cphiADD(circuit, q, aux, N, n, 0)

    ccphiADD(circuit, q, ctl1, ctl2, a, n, 1)
    create_inverse_QFT(circuit, q, n, 0)
    circuit.x(q[n-1])
    circuit.cx(q[n-1], aux)
    circuit.x(q[n-1])
    create_QFT(circuit,q,n,0)
    ccphiADD(circuit, q, ctl1, ctl2, a, n, 0)

"""Circuit that implements the inverse of doubly controlled modular
    addition by a"""
```

```
def ccphiADDmodN_inv(circuit, q, ctl1, ctl2, aux, a, N, n):
    ccphiADD(circuit, q, ctl1, ctl2, a, n, 1)
    create_inverse_QFT(circuit, q, n, 0)
    circuit.x(q[n-1])
    circuit.cx(q[n-1],aux)
    circuit.x(q[n-1])
    create_QFT(circuit, q, n, 0)
    ccphiADD(circuit, q, ctl1, ctl2, a, n, 0)
    cphiADD(circuit, q, aux, N, n, 1)
    create_inverse_QFT(circuit, q, n, 0)
    circuit.cx(q[n-1], aux)
    create_QFT(circuit, q, n, 0)
    phiADD(circuit, q, N, n, 0)
    ccphiADD(circuit, q, ctl1, ctl2, a, n, 1)

"""Circuit that implements single controlled modular multiplication
    by a"""
def cMULTmodN(circuit, ctl, q, aux, a, N, n):
    create_QFT(circuit,aux,n+1,0)
    for i in range(0, n):
        ccphiADDmodN(circuit, aux, q[i], ctl, aux[n+1], (2**i)*a % N,
            N, n+1)
    create_inverse_QFT(circuit, aux, n+1, 0)

    for i in range(0, n):
        circuit.cswap(ctl,q[i],aux[i])

    a_inv = modinv(a, N)
    create_QFT(circuit, aux, n+1, 0)
    i = n-1
    while i >= 0:
        ccphiADDmodN_inv(circuit, aux, q[i], ctl, aux[n+1], math.pow
            (2,i)*a_inv % N, N, n+1)
        i -= 1
    create_inverse_QFT(circuit, aux, n+1, 0)

""" Main program """
if __name__ == '__main__':

    """ Ask for analysis number N """

    N = int(input('Please insert integer number N: '))

    print('input number was: {0}\n'.format(N))

    """ Check if N==1 or N==0"""

    if N==1 or N==0:
```

```
        print('Please put an N different from 0 and from 1')
        exit()

    """ Check if N is even """

    if (N%2)==0:
        print('N is even, so does not make sense!')
        exit()

    """ Check if N can be put in N=p^q, p>1, q>=2 """

    """ Try all numbers for p: from 2 to sqrt(N) """
    if check_if_power(N)==True:
        exit()

    print('Not an easy case, using the quantum circuit is necessary\n')


    """ Get an integer a that is coprime with N """
    a = get_value_a(N)


    """ Get n value used in Shor's algorithm, to know how many qubits are used """
    n = math.ceil(math.log(N,2))

    print('Total number of qubits used: {0}\n'.format(4*n+2))

    """ Create quantum and classical registers """

    """auxilliary quantum register used in addition and multiplication"""
    aux = QuantumRegister(n+2)
    """quantum register where the sequential QFT is performed"""
    up_reg = QuantumRegister(2*n)
    """quantum register where the multiplications are made"""
    down_reg = QuantumRegister(n)
    """classical register where the measured values of the QFT are stored"""
    up_classic = ClassicalRegister(2*n)

    """ Create Quantum Circuit """
    circuit = QuantumCircuit(down_reg , up_reg , aux, up_classic)

    """ Initialize down register to 1 and create maximal superposition in top register """
    circuit.h(up_reg)
```

```
    circuit.x(down_reg[0])

    """ Apply the multiplication gates as showed in the report in order to create the exponentiation """
    for i in range(0, 2*n):
        cMULTmodN(circuit, up_reg[i], down_reg, aux, int(pow(a, pow(2, i))), N, n)

    """ Apply inverse QFT """
    create_inverse_QFT(circuit, up_reg, 2*n ,1)

    """ Measure the top qubits, to get x value"""
    circuit.measure(up_reg,up_classic)

    """ Select how many times the circuit runs"""
    number_shots=int(input('Number of times to run the circuit: '))
    if number_shots < 1:
        print('Please run the circuit at least one time...')
        exit()

    if number_shots > 1:
        print('\nIf the circuit takes too long to run, consider running it less times\n')

    """ Print info to user """
    print('Executing the circuit {0} times for N={1} and a={2}\n'.format(number_shots,N,a))



    backend = AerSimulator()  # Definindo o backend do simulador


       tqc = transpile(circuit, backend)  # Transpilando o circuito para o backend

    simulation = backend.run(tqc, shots=number_shots)  # Executando o circuito no backend e atribuindo a variavel 'simulation'


    """ Get the results of the simulation in proper structure """

    sim_result=simulation.result()
    counts_result = sim_result.get_counts(circuit)

    print("=== sim_result.data type ===")
    print(type(sim_result.data))

```

```
    # Try printing the available keys or attributes
    print("\n=== dir(sim_result.data) ===")
    print(dir(sim_result.data))

    # If possible, try converting to dict or listing items
    try:
        print("\n=== sim_result.data keys ===")
        print(sim_result.data.keys())
    except Exception as e:
        print(f"Could not access .keys(): {e}")

    # Print the full data object (if small)
    print("\n=== sim_result.data content ===")
    print(sim_result.data)

    """ Print info to user from the simulation results """
    print('Printing the various results followed by how many times 
        they happened (out of the {} cases):\n'.format(number_shots))
    i=0
    while i < len(counts_result):
        print('Result \"{0}\" happened {1} times out of {2}'.format(
            list(sim_result.get_counts().keys())[i],list(sim_result.
            get_counts().values())[i],number_shots))
        i=i+1

    """ An empty print just to have a good display in terminal """
    print(' ')

    """ Initialize this variable """
    prob_success=0

    """ For each simulation result, print proper info to user and try
         to calculate the factors of N"""
    i=0
    while i < len(counts_result):

        """ Get the x_value from the final state qubits """
        output_desired = list(sim_result.get_counts().keys())[i]
        x_value = int(output_desired, 2)
        prob_this_result = 100 * ( int( list(sim_result.get_counts().
            values())[i] ) ) / (number_shots)

        print("------> Analysing result {0}. This result happened in 
            {1:.4f} % of all cases\n".format(output_desired,
            prob_this_result))

        """ Print the final x_value to user """
```

```
        print('In decimal, x_final value for this result is: {0}\n'.format(x_value))

        """ Get the factors using the x value obtained """
        success=get_factors(int(x_value),int(2*n),int(N),int(a))

        if success==True:
            prob_success = prob_success + prob_this_result

        i=i+1

    print("\nUsing a={0}, found the factors of N={1} in {2:.4f} % of the cases\n".format(a,N,prob_success))

    end = time.time()
    print(f"Execution time: {1000 * (end - start):.2f} ms")  # in milliseconds    counts_result = sim_result.get_counts(circuit)


    """ Print info to user from the simulation results """
    print('Printing the various results followed by how many times they happened (out of the {} cases):\n'.format(number_shots))
    i=0
    while i < len(counts_result):
        print('Result \"{0}\" happened {1} times out of {2}'.format(list(sim_result.get_counts().keys())[i],list(sim_result.get_counts().values())[i],number_shots))
        i=i+1

    """ An empty print just to have a good display in terminal """
    print(' ')

    """ Initialize this variable """
    prob_success=0

    """ For each simulation result, print proper info to user and try to calculate the factors of N"""
    i=0
    while i < len(counts_result):

        """ Get the x_value from the final state qubits """
        output_desired = list(sim_result.get_counts().keys())[i]
        x_value = int(output_desired, 2)
        prob_this_result = 100 * ( int( list(sim_result.get_counts().values())[i] ) ) / (number_shots)

```

```python
            print("------> Analysing result {0}. This result happened in {1:.4f} % of all cases\n".format(output_desired, prob_this_result))

            """ Print the final x_value to user """
            print('In decimal, x_final value for this result is: {0}\n'.format(x_value))

            """ Get the factors using the x value obtained """
            success=get_factors(int(x_value),int(2*n),int(N),int(a))

            if success==True:
                prob_success = prob_success + prob_this_result

            i=i+1

        print("\nUsing a={0}, found the factors of N={1} in {2:.4f} % of the cases\n".format(a,N,prob_success))

        end = time.time()
        print(f"Execution time: {1000 * (end - start):.2f} ms")  # in milliseconds
```

[1]

[1] https://github.com/tiagomsleao/ShorAlgQiskit/blob/master/Shor_Normal_QFT.py