IMPLEMENTAÇÃO DE ALGORITMO QUÂNTICO EM LINGUAGEM FUNCIONAL E MULTIPARADIGMA APLICADO À SEGURANÇA DA INFORMAÇÃO

Mariana Godoy Vazquez Miano [1]
Aleccheevina Silva de Oliveira[2]

doi: 10.47283/244670492022100257

RESUMO

O presente trabalho tem como objetivo apresentar o desenvolvimento do algoritmo quântico de Shor, em linguagem funcional (Haskell) e multiparadigma (Python), avaliando seu desempenho nessas linguagens quanto ao tempo de execução. Para o desenvolvimento da simulação e testes em ambiente quântico na linguagem Python, utiliza-se a plataforma IBM-Q, juntamente com o Qiskit e o Jupyter Notebook (Open-source). Para os testes com a linguagem Haskell é utilizada conjuntamente a biblioteca monada QIO. Com a realização desses testes, faz-se a comparação em termos de desempenho computacional. Ao longo do processo é possível notar que Haskell obtém os melhores resultados com relação à eficiência (em relação à Python), utilizando menor quantidade de memória RAM. Porém, com Haskell, a simulação se limita a 5 qubits enquanto em Python, é possível utilizar até 7 qubits. Em relação à segurança da informação, para demonstrar a capacidade do algoritmo de fatoração Shor quebrar o método RSA, é usada a versão implementada em Python e Qiskit, uma vez que essa biblioteca suporta maiores volumes de dados.

PALAVRAS-CHAVES: Haskell. Python. Desempenho. Algoritmo de Shor. Criptografia RSA.

ABSTRACT

This work aims to present the development of Shor's quantum algorithm, in functional language (Haskell) and multiparadigm (Python), evaluating its performance in these languages in terms of execution time. For the development of simulation and tests in a quantum environment in the Python language, the IBM-Q platform is used, together with Qiskit and Jupyter Notebook (Open-source). For tests with the Haskell language, the monada QIO library is used together. With the performance of these tests, a comparison is made in terms of computational performance. Throughout the process, it is possible to notice that Haskell obtains the best results in terms of efficiency (compared to Python), using a smaller amount of RAM. However, with Haskell, the simulation is limited to 5 qubits while in Python, it is possible to use up to 7 qubits. Regarding information security, to demonstrate the ability of the Shor factorization algorithm to break the RSA method, the version implemented in Python and Qiskit is used, since this library supports larger volumes of data.

KEYWORDS: *Haskell. Python. Performance. Shor's algorithm. RSA cryptography.*

[1] Docente e pesquisadora da Fatec Americana nos Cursos de Tecnologia da Informação (ADS, SI e JD). E-mail: mariana.miano@fatec.sp.gov.br ;
[2] Aluna do Curso de Tecnologia em Segurança da Informação da Fatec Americana. E-mail: aleccheevina.oliveira@fatec.sp.gov.br

INTRODUÇÃO

A Computação Quântica é um novo paradigma da Computação, uma nova área de pesquisa inter e multidisciplinar que necessita, para seu pleno desenvolvimento, o aporte da Matemática, da Física e da Computação. Com esse novo paradigma, os futuros programadores deverão conhecer bem a forma como a informação deve ser tratada na perspectiva quântica, de forma que deter conhecimento sobre mecânica quântica deixará de ser um privilégio restrito aos físicos (GRILO, 2014).

Nos últimos anos a evolução do poder de processamento dos computadores tem crescido de forma constante e veloz. Atualmente, o número de transistores que os processadores mais modernos possuem, estão perto de chegar ao nível atômico. Entretanto, a forma como essas máquinas funcionam hoje não seria adequada para operar no nível microscópico (ou nanométrico). Segundo Miano e Oliveira, (2021), em geral, a computação quântica utiliza de conceitos da mecânica quântica e da física em uma visão computacional, como emaranhamento, sobreposição e interferência, possibilitando um grande poder computacional.

Desta forma, o modelo clássico de computação começa a ser questionado por seus limites físicos e o modelo de computação quântica ganha mais destaque devido as mudanças que causará no setor de Tecnologia da Informação (TI). O desenvolvimento da Computação Quântica tem sido uma preocupação para empresas de Tecnologia de Informação e Universidades. Empresas como IBM e Microsoft, entre muitas outras, estão fazendo altos investimentos em pesquisas para a construção de computadores quânticos e softwares que funcionarão conjuntamente.

A utilização de computadores quânticos é capaz de reduzir drasticamente o tempo necessário para executar as mesmas tarefas que já podemos executar em computadores clássicos. Além disso, problemas que são considerados insolúveis para a computação clássica passam a ser solúveis em computação quântica.

O algoritmo de Shor foi a primeira grande evidência que o modelo quântico de computação poderia superar o modelo clássico das máquinas de Turing (BENNETT *et al*, 1997). Além dos novos algoritmos quânticos, os trabalhos de Shor também impulsionaram as pesquisas por novos algoritmos criptográficos e levantaram questões sobre a viabilidade e as limitações do modelo quântico.

A partir da publicação do algoritmo de Shor e de suas consequências para a área de TI, surgiram outros algoritmos quânticos. Embora alguns algoritmos quânticos existentes mostrem uma aceleração teórica em relação a algoritmos clássico, a implementação e execução desses algoritmos apresentam vários desafios, segundo Ristè *et al*. (2017). Os dados de entrada determinam, por exemplo, o número necessário de qubits e portas de um algoritmo quântico. A implementação de um algoritmo também depende do software de desenvolvimento usado, o que restringe o conjunto de computadores quânticos utilizáveis.

O algoritmo de Shor é considerado o breakthrough da computação moderna, uma vez que demonstrou efetivamente a importância da computação quântica, pois resolveu um problema matemático que era estudado há anos e em tempo polinomial. O problema da fatoração de grandes números é considerado tão difícil de resolver em um computador clássico que foi implementado no sistema de criptografia RSA (STALLINGS, 2014).

Atualmente, é considerado um dos melhores algoritmos para fatoração. Na versão quântica, é capaz de fatorar números de altas ordens em segundos (LOMONACO, 2002). Essa capacidade pode ser usada para "quebrar" algoritmos de encriptação atuais, como o RSA. O objetivo do algoritmo é achar o período de uma função, e na sequência, encontrar os fatores do valor solicitado.

O algoritmo de Shor é um algoritmo quântico que, dado um inteiro n composto ímpar que não é potência de primo, devolve um fator de n com probabilidade limitada de erro. Como n é produto de no máximo log n inteiros, o algoritmo de Shor pode ser utilizado para resolver o problema da fatoração em tempo polinomial no tamanho da entrada (MARTINS, 2018). O algoritmo de Shor baseia-se numa redução do problema da busca de um fator de n ao problema da busca do período de uma sequência. Como a redução utiliza aleatorização é possível que ela falhe, ou seja, que nenhum fator de n seja encontrado. Porém, a probabilidade de ocorrência deste evento é limitada (MIANO, 2020).

Na computação quântica, é comum que as linguagens de programação sejam funcionais. A programação funcional é um paradigma que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis.

1 LINGUAGENS

Neste tópico serão apresentadas as linguagens escolhidas para a comparação de desempenho do algoritmo de Shor, bem como suas respectivas ferramentas para programação quântica, a saber Python com a biblioteca Qiskit e Haskell com o módulo QIO.

2.1 Haskell, Python e Qiskit

Haskell é uma linguagem funcional baseada no conceito de funções matemáticas, isto é, modela um problema computacional como uma coleção de funções matemáticas, cada uma com um domínio de entrada e um resultado. A linguagem haskell foi desenvolvida para ser simples, direta e objetiva, tanto é que uma de suas maiores características diferenciais em comparação a outras linguagens é a sua simplificação de códigos, simplesmente sendo composto por uma ou mais funções nas linhas do código. A linguagem haskell é baseada no modelo computacional cálculo lambda (MALAQUIAS, 2017).

P*ython* é uma linguagem de programação de alto nível, interpretada e imperativa. Também é orientada a objetos, funcional, de tipagem dinâmica e com suporte nativo a números complexos (BEAZLEY, 2013). Surgiu em 1991 e possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation.

O Qiskit é um framework de código aberto para programação quântica com Python em níveis de circuito, pulsos e algoritmos. É usado para experimentos e simulações em computadores quânticos reais. É uma ferramenta fullstack, dividida em quatro elementos básicos: Qiskit Terra, Qiskit Aer, Qiskit Igns e Qiskit Aqua (QISKIT DOCUMENTATION).

Terra é a base da ferramenta e se trata dos componentes responsáveis pela programação quântica backend em nível de circuito e pulsos. É dividida em seis módulos: qiskit.circuit, qiskit.pulse, qiskit.transpiler, qiskit.providers, qiskit.quantum_info e qiskit.visualization.

Aer é responsável pela simulação tanto dos circuitos programados no Qiskit Terra quanto de ruídos que podem interferir no projeto. Usado para simular um computador quântico em cima de um computador clássico com o máximo de desempenho possível. Possui três simuladores backend de alto nível: QasmSimulator, StatevectorSimulator e UnitarySimulator.n.

Ignis é parte da ferramenta responsável pela correção de erros e tratamentos de ruídos, através da melhoria das portas e circuitos bem como otimização para ambientes específicos. É dividido em três módulos: Circuits gera uma lista de circuitos para cada tipo de experimento executado no Terra ou no Aer. Fitters é responsável por analisar os resultados de acordo com o modelo físico definido para o experimento. O último módulo, Filter, é um objeto gerados em alguns experimentos com o objetivo de mitigação de erros. Além disso, o Ignes pode ser usado em três tipos de experimento: o qiskit.ignis.characterization é usado para medir parâmetros do sistema, qiskit.ignis.verification é usado para verificação das portas lógicos e de pequenos circuitos e o qiskit.ignis.mitigation usam circuitos de calibração para medição de resultados que podem ser ajustados pelo módulo Fitters ou usado para mitigação pelo módulo Filter.

Aqua é a parte da ferramenta usada efetivamente para a criação dos algoritmos e aplicativos para computação quântica sendo acessível para uma diversidade usuários.

3 PROCEDIMENTOS METODOLÓGICOS

Inicialmente explorou-se o Qiskit para o desenvolvimento do circuito quântico, com visualização de suas portas lógicas. Na sequência, ocorreu o desenvolvimento do algoritmo de Shor nas linguagens Python e Haskell (teste utilizando a mônada QIO). Na sequência, houve a medição de desempenho do algoritmo de Shor quanto aos tempos de execução nas linguagens Haskell e Python. Para finalizer, houve o teste do algoritmo de Shor com o RSA.

3.1 Conceitos e portas lógicas quânticas

Para criar um circuito quântico, é preciso primeiro importar algumas bibliotecas do qiskit, tanto para criação do circuito como no caso do *QuantumCircuit,* como para visualizá-lo como no caso do *plot_histogram*. Conforme a complexidade do algoritmo cresce se faz necessária a chamada de bibliotecas mais específicas. A figura 1 mostra importações de bibliotecas básicas para a formação e visualização dos circuitos quânticos.

Figura 01: Importações das bibliotecas do Qiskit

```
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import plot_histogram
n = 8
n_q = n
n_b = n
qc_output = QuantumCircuit(n_q,n_b)
for j in range(n):
    qc_output.measure(j,j)
qc_output.draw()
counts = execute(qc_output,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

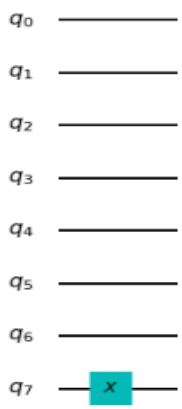
Fonte: Qiskit Textbook

Nesse exemplo, *n_q* é o número de qubits de entrada e *n_b* é o número de qubits de saída. A biblioteca *QuantumCircuit* cria o circuito *qc_output*. A função mesure mede cada *qubit* de entrada e guarda no qubit de saída. Todos os qubits são iniciados em zero no qiskit, como não foi feita nenhuma operação nesse exemplo, somente a medição, então os qubits de saída também terão valores zero. O Qiskit apresenta portas básicas importantes e que podem ser usados na construção de outras portas, essas são a porta NOT, CNOT, Hadamard e Toffoli.

No próximo exemplo, será usado a operação NOT representado por X no circuito quântico representado pela figura 2. Cria-se um circuito *qc_encode* com 8 qubits e no último é feito uma operação NOT representado pela função x.

Figura 2: Função x - porta lógica quântica NOT

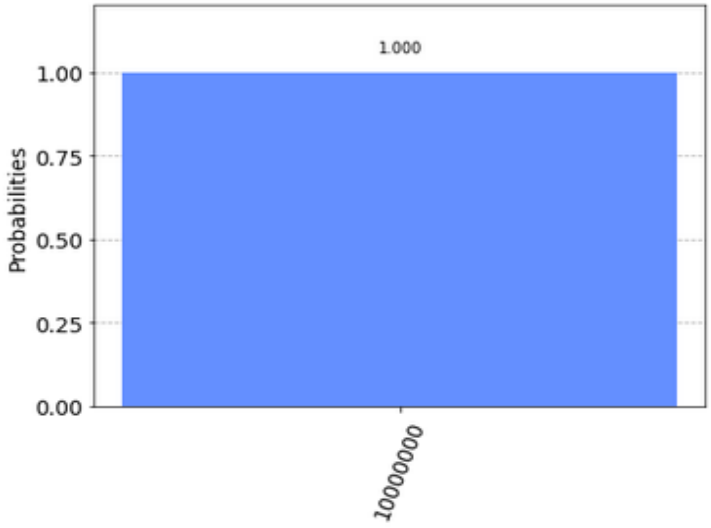
```
In [4]: qc_encode = QuantumCircuit(n)
        qc_encode.x(7)

        qc_encode.draw()
```



Fonte: Qiskit Textbook

Na figura 3, o Qiskit mostra a sequência de qubits da direita para esquerda.

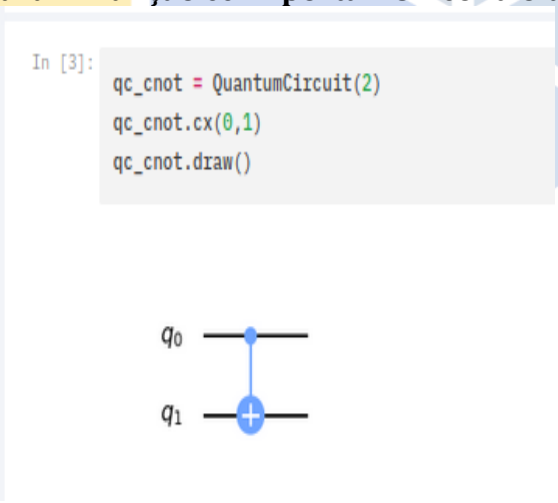
Figura 3: Gráfico do resultado da medição do circuito quântico



Fonte: Qiskit Textbook

Outra porta lógica importante para a computação quântica é a Controled NOT (CNOT), seu uso pode ser exemplificado em um programa para soma de qubits. Na computação clássica seria usado uma porta ou exclusiva para fazer esse mesmo cálculo, pois o resultado só será igual a 1 se somente uma das parcelas tiverem o valor 1. Na computação quântica é possível obter esse efeito através da porta CNOT com a função *cx*, conforme a figura 4:

Figura 4: Função ccx – porta NOT controlada

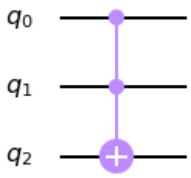


Fonte: Green, 2010

A porta de Toffoli, também conhecida CCNOT por utilizar dois qubits de controle, é dada pela função *ccx*, conforme mostra a figura 5:

Figura 5: Função ccx - porta de Toffoli

```
In [9]:
#porta Toffoli ou ccnot
qc_toffoli = QuantumCircuit(3)
a = 0
b = 1
t = 2
qc_toffoli.ccx(a,b,t)
qc_toffoli.draw()
```

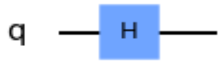


Fonte: Green, 2010

Os dois exemplos na sequência são, respectivamente, a porta de Hadamard que mapeia os qubits da base Z (|0> e |1>) para base X (|+> e |->) e a porta swap que move um estado entre dois qubits. A porta de Hadamard é representada pela função *h* na figura 6:

Figura 6: Função h - porta de Hadamard

```
In [4]:
#Hadamard
qc_hadamard.h(0)
qc_hadamard.draw()
```

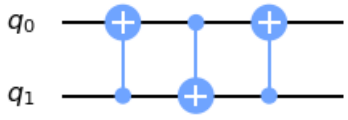


Fonte: Green, 2010

No caso do porta swap é possível implementá-la com a função swap ou com três operações CNOT consecutivas, como mostra a figura 7:

Figura 7: Funções cx usada para simulação do porta swap

```
In [7]:
#porta swap usando cnot
a = 0
b = 1
qc_s = QuantumCircuit(2)
qc_s.cx(b,a)
qc_s.cx(a,b)
qc_s.cx(b,a)
qc_s.draw()
```



Fonte: Green, 2010

3.2 Linguagem Haskell

Segundo Hudak *et al* (2000), a linguagem Haskell tem como característica ser uma linguagem puramente funcional, que é feita através das chamadas das funções e definição, além de ser fortemente "tipada", ou seja, sempre que o programador colocar um "*input*", ele deverá informar se é uma *string*, *interger* ou *boolean,*entre outros*.* Para obter o "*output".* Haskell é uma linguagem bem voltada para área de matemática em si, trazendo soluções simplificadas, se comparada com outras linguagens, o programador teria que importar bibliotecas ou procurar por soluções mais complexas. Pelo fato de ser funcional e usar mônadas, Haskell é utilizada como linguagem na simulação da computação quântica, pois permite modelar os efeitos computacionais quânticos separadamente.

3.3 Haskell Quantum Monad I/O

Haskell possui uma biblioteca chamada Quantum Monad IO (QIO), permitindo a realização de simulações de qubits através do algoritmo quântico de Shor e assim, realizar a fatoração dos números para encontrar números primos, para obter a chave privada da criptografia RSA. Segundo Feitosa (2016), a mônada permite que o programador não precise programar manualmente as transformações e novas combinações todas as vezes que ele for programar.

A ideia desta mônada é construir os estados quânticos, representados matematicamente por um vetor de números complexos contendo as amplitudes de probabilidades dos Qbits, com a possibilidade de transformação destes estados através de portas quânticas, representadas por matrizes unitárias, que são aplicadas através da operação bind.

Uma das grandes vantagens do uso de QIO é o fato de que o próprio *framework* faz a separação entre os dados reversíveis e os dados irreversíveis. O primeiro são os dados

unitários e o segundo são dados baseados em probabilidade. Por exemplo, seria possível usar a fórmula da superposição (Hadamard Gate) H = $\frac{1}{\sqrt[2]{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$ colapsando em H|0⟩=|+⟩ ou H|1⟩=|−⟩. Utilizando, o Quantum Monad IO e um um pseudocódigo, tem-se:

```
|+⟩::QIO Qbit
|+⟩ = do q ← |0⟩
applyU (uhad q)
return q

|−⟩::QIO Qbit
|−⟩= do q ← |1⟩
applyU (uhad q)
return q
```

É explicito que após o "::" foi usado pra QIO Qbit para declarar que está sendo usado para declarar que é um bit quântico, e que caso houvesse dois bits, apesar do comportamento serem bem iguais, eles poderiam colapsar em 0⟩ ou 1⟩. Este exemplo ilustra o uso de um pseudocódigo para simular como seria usado no framework.

3.4 Implementação do algoritmo de Shor

Antes de iniciar o algoritmo de Shor para fatorar um número N, é preciso fazer algumas checagens, tais como verificar se N não é um número primo e se é par. No primeiro caso, se N for primo não tem fatores não trivial; no segundo caso, se N for par, é preciso dividi-lo por dois até encontrar um número ímpar (MARTINS, 2018, p.54-55).

Após esses passos iniciais deve-se escolher um número m, tal que, $1 \leq m \leq N-1$e se $mdc(m,N) \neq 1$ então m é um fator não trivial de N. Caso contrário, se $mdc(m,N) = 1$, ou seja, não tem divisores comum, então é preciso encontrar a ordem multiplicativa de $m\ mod\ N$. Para essa etapa é usado um computador quântico para determinar o período P da função dada por $x \rightarrow m^x mod\ N$. O período pode ser escrito na forma $P = ord_N m$. Se P for ímpar, é preciso escolher um novo m e recomeçar, mas se não for há duas probabilidades que determina o curso do algoritmo:

$$(m^{\frac{P}{2}} - 1)(m^{\frac{P}{2}} + 1) = m^P - 1 \equiv 0\ mo\ N$$

ou

$$(m^{\frac{P}{2}} - 1) \neq 0\ mod\ N$$

Se $(m^{\frac{P}{2}} + 1) \equiv 0\ mod\ N$ é preciso escolher um m e recomeçar o algoritmo. Se $(m^{\frac{P}{2}} + 1) \neq 0\ mod\ N$, então para encontrar um fator não trivial de N basta calcular $d = mdc(m^{\frac{P}{2}} - 1, N)$. Para calcular o período da função, é preciso de dois registradores composto por $log_2(N+1)$qubits iniciados em 0.

3.4.1 Implementação com Qiskit

Para a implementação deste algoritmo foram escolhidos números ímpares e não primos entre 0 e 100. (9, 15, 21, 25, 27, 33, 35, 39, 45, 49, 51, 55, 57, 63, 75, 77, 81, 85, 87, 91,

93, 95, 99) para ser fatorado. O programa inicia gerando um número aleatório tal que $2 \leq a \leq N$ e $mdc(N, a) = 1$, como mostra a figura 08:

Figura 08: Função Gerar a

```
def gerar_a (n):
    a = 0
    while gcd(n,a) != 1 or a == 1:
        a = randint(2,n)
    return a
```

Fonte: Green, 2010

O circuito de Shor terá dois registradores. O primeiro contará $2 * \log_2(N+1)$ e os segundo com $log_2(N+1)$. Isso ocorre porque o primeiro registrador é responsável por realizar o cálculo da função $a^x \, mod \, N$, executado por um circuito unitário controlado de tamanho $log_2(N+1)$, assim a cada chamada deste circuito é preciso de um qubit de controle e de um para o resultado. A figura 09 mostra a implementação do circuito de Shor:

Figura 09: Circuito de Shor

```
def q_circuito_shor (r,s):
    cq_shor = QuantumCircuit(r + s, r)
    #coloca em superposição
    for i in range(r):
        cq_shor.h(i)
    #O ultimo qubit será |1> (ancilla)
    cq_shor.x(s-1+r)
    #chama a função U a_mod_n
    for j in range(r):
        cq_shor.append(q_a_mod_n(a, 2**j, s), [j] + [k+r for k in range(s)])

    cq_shor.append(qft_dagger(r), range(r))
    return cq_shor
```

Fonte: Green, 2010

É preciso aplicar QFT inversa no final do circuito, pois ao colocar os dados de entrada em sobreposição consequentemente eles passam para a base de Fourier a qual não é possível medir. A transformada quântica de Fourier transforma da base computacional (Z) para a base de Fourier (~x) e para um qubit de estado x $H|x> \equiv QFT|x> = |\sim x>$ (no Qiskit). Então a função qtf_dagger faz o oposto, colocando os dados da base x para a base z para que a medição seja possível.

A figura 10 mostra o operador Unitário U responsável pelo cálculo da função $a^x \, mod \, N$.

Figura 10: Operador U

```
def q_a_mod_n(a, r, s):
    U = QuantumCircuit(s)
    for i in range(r):
        if s == 2:
            U.swap(0,1)
        if s == 3:
            U.swap(0,1)
            U.swap(1,2)
        if s == 4:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if s == 5:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
            U.swap(3,4)
    U = U.to_gate()
    U.name = "%i^%i mod n" % (a, r)
    c_U = U.control()
```

Fonte: Green, 2010

Após a medição com a função $get\_counts()$ do Qiskit e obtenção da fase, calcula-se o período da função e se este for par, obtém-se um fator não trivial de N, como pode ser visualizado na figura 11.

Figura 11: Código que calcula um fator de N

```
def ordem_multiplicativa_n (periodo):
    ord_mult_n = periodo//2
    return ord_mult_n

#calcula um fator não trivial de N
def fator_n (ord_mult_n, a, n):
    fator = gcd(((a**ord_mult_n)-1), n)
    return fator
```

Fonte: Green, 2010

A tabela 1 contém os resultados obtidos para os valores de N escolhido usando o Simulador IBM Q com a biblioteca Qiskit.

Tabela 1. Fatoração de números não primos entre 0 e 100 – Qiskit

Fatoração de Números Ímpares Não Primos Entre 0 e 100

Número Fatorado (N)	Número aleatório a	Fator não trivial de N	Tempo de Execução em segundos
9	2	3	2
15	8	3	2
21	19	3	11
25	22	5	42
27	10	9	11
33	25	3	132
35	2	7	531
39	35	13	66
45	22	9	132
49	39	7	66
51	16	3	67
55	46	5	131
57	17	19	66
63	53	7	132
75	37	3	595
77	9	7	600
81	4	9	2426
85	71	5	6108
87	67	3	2452
91	44	7	615
93	4	3	613
95	61	5	613
99	82	9	3681

Fonte: Green, 2010

3.4.2 Implementação com QIO (Haskell)

A mônada QIO conta com um exemplo do algoritmo de Shor implementado pelos seus desenvolvedores. Originalmente, suporta 4 qubits e assim é capaz de fatorar números entre 1 e 15, como mostra a figura 12 (GREEN, 2010, p.21-27).

Figura 12: Limite da mônada QIO

```
-- | Quantum integers are of a fix
-- Currently, this is set to 4.
qIntSize :: Int
qIntSize = 4
```

Fonte: Green, 2010

Assim como no Qiskit, em Haskell foi implementado o operador U com a função $a^x \ mod \ N$ em sobreposição. Após essa etapa, a função *S*hor, apresentada na figura 13, mede o registrador em busca do período desta função.

Figura 13: Shor em Haskell

```
shorU :: QInt -> QInt -> Int -> Int -> U
shorU k i1 x n = hadamardsI k `mappend` modExp n x k i1 `mappend` qftI k

shor :: Int -> Int -> QIO Int
shor x n = do
  i0 <- mkQ 0
  i1 <- mkQ 1
  applyU (shorU i0 i1 x n)
  measQ i0
```

Fonte: Green, 2010

Como a mônada QIO comporta até quatro qubits, somente os números entre 1 e 15 poderiam ser candidatos a fatoração, uma vez que o maior número binário a ser representado por esses quibits é 15 (1111). Para ter mais opções de números na avaliação do desempenho de Haskell, alterou-se algumas partes da biblioteca de modo que comportasse cinco qubits, tornando possível fatorar números entre 1 e 31 como mostra a figura 14:

Figura 14: Controle de Tamanho QIO

```
qIntSize :: Int
qIntSize = 5
```

Foi preciso alterar outras funções da própria mônada QIO, das quais o operador U$a^x \ mod \ N$. Essas funções tinha uma limitação x e assim consequentemente limitavam o cálculo do período. A figura 15 apresenta a função modExpStep que controla as iterações de funções modulares.

Figura 15: Função modExpStep

```
modExpStep :: Qbit -> Int -> Int -> QInt -> Int -> U
modExpStep qc n a o p = letU 0 (\z -> (condMultMod qc n p' o z)
                        `mappend` (ifQ qc (swapQInt o z))
                        `mappend` (urev (condMultMod qc n (inverseMod n p') o z)))
  where
    p' = (a^(2^p)) `mod` n
```

A tabela 2 apresenta os valores obtidos na execução do programa de fatoração em Haskell.

Tabela 2. Fatoração de números não primos entre 0 e 30 – QIO

Fatoração de Números Ímpares Não Primos Entre 0 e 30			
Número Fatorado (N)	Número aleatório a	Fator não trivial de N	Tempo de Execução em segundos
9	5	3	4
15	2	3	1
21	2	3	4
25	6	5	38
27	16	9	9

Tabela 2. Fatoração de números não primos entre 0 e 30 – QIO

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

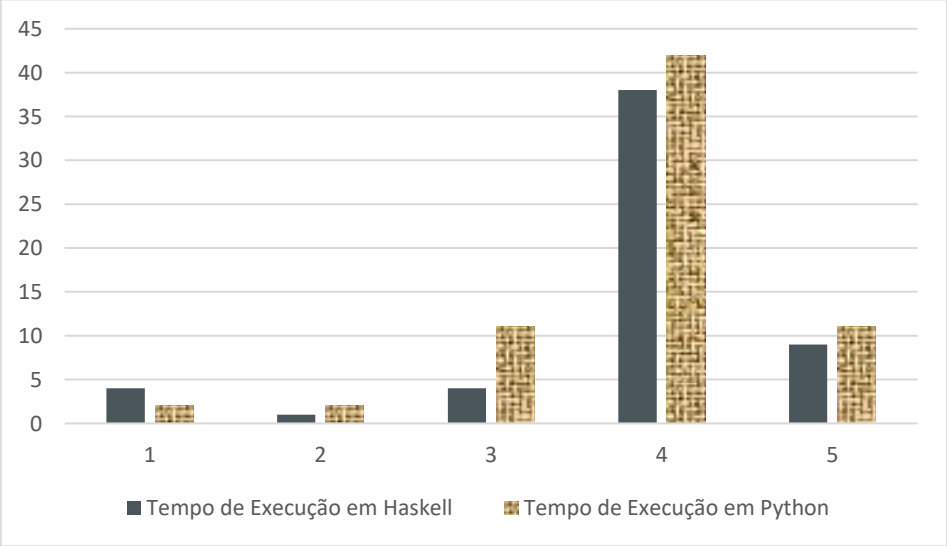
Para uma comparação justa entre o algoritmo de Shor em Python e em Haskell, usou-se a simulação local (sem requisição ao servidor IBM). Assim, tanto o uso do Qiskit (Python) quanto do QIO (Haskell) será feito usando o hardware local. Além disso, o número de qubits do algoritmo em Python foi limitado para que avaliação de desempenho seja justa em relação aos recursos de cada ferramenta. A tabela 3 contém os valores do tempo de execução do algoritmo de Shor nas duas linguagens.

Tabela 3. Comparação de Desempenho entre Haskell e Python

Número Fatorado	Tempo(segundos) da Execução em Haskell	Tempo (segundo) de Execução em Python
9	4	2
15	1	2
21	4	11
25	38	42
27	9	11

A figura 16 apresenta o gráfico de desempenho comparativamente nas linguagens Haskell e Python.

Figura 16: Comparação de desempenho em tempo de execução.



O tempo (segundos) de execução de Haskell é consideravelmente mais rápido que o tempo de execução em Python, o que mostra que, apesar do limite de 5 Qubits da mônada QIO, a linguagem tem potencial para programação de algoritmos quânticos principalmente pelo seu tempo de execução.

4.1 Decriptação

O Algoritmo de Shor resolve o problema da fatoração de grandes números. Para demonstrar o impacto na criptografia, o algoritmo escrito em Python será usado para decifrar

um texto criptografado, uma vez que, como visto antes, essa linguagem comportou números maiores. Primeiramente, foi usado um programa RSA para gerar um par de chaves públicas e a chave privada, assim é possível usar a chave pública, com intuito de descobrir qual era a chave privada. Nos testes realizados para decriptar um número simulando uma chave criptográfica, usando o algoritmo de Shor e o IBM Q, utilizou-se a chave pública (77, 47), a fim de encontrar a chave privada. Então foi adicionado o número 77 ao programa e iniciou-se a fatoração do número 77, resultando no valor 17.

Durante a execução do algoritmo quântico levou-se 0.001480 segundos entre o começo e fim do teste do número 77. Ele fatorou o valor e imprimiu. Foram 7 períodos quânticos, que são os momentos em que o programa faz a medição após a onda da partícula chegar ao topo.

O algoritmo de Shor, gerou os valores 77, 57,7,7 e por fim, ele resultou em final de 17. Descobrindo assim qual era a chave privada (17). Vale ressaltar que a figura 20 contêm os períodos 0 e 10, respectivamente. Estes dois períodos não são mostrados como resultados, pois o algoritmo não conseguiu calcular o valor. O algoritmo gera um valor aleatório e se este não satisfizer o cálculo da fatoração, o programa automaticamente o descartará e gerará um novo período. Foi o que aconteceu com os períodos 0 e 10, eles foram descartados até que o período 7 foi gerado, pois conseguiu contemplar os requisitos da fatoração. O algoritmo quântico em si é rápido de ser usado, entretanto, a disponibilidade de uso do IBM Q é bem limitada. Então, usando o Jupyter Notebook, importou-se uma biblioteca Qiskit e desenvolveu-se todo o algoritmo de Shor, no qual foi adicionado um número para resultar na fatoração.

Através da biblioteca da IBM, é possível utilizar seus computadores quânticos e obter resultados em computadores clássicos. Contudo, é necessário entrar em uma fila, que pode demorar para devolver o resultado do algoritmo. Na escolha de ferramentas para execução dos algoritmos quânticos, levou-se em consideração não só a familiaridade com a linguagem, mas também a disponibilidade de documentação.

Desse modo, o IBM Q com o Qiskit, têm uma ampla documentação, possuindo vários tutoriais que auxiliaram na compreensão e por fim, no desenvolvimento do algoritmo quântico. O fato de as ferramentas serem Open Source permitiram algumas modificações necessárias.

5 DISCUSSÃO E CONSIDERAÇÕES FINAIS

A linguagem Haskell foi escolhida por seu caráter funcional que por ter uma proximidade com a própria linguagem matemática imaginou-se em princípio que comportaria bem o paradigma de computação quântica. Além disso, são ferramentas que contam com ampla documentação. Ao longo do processo foi possível notar que Haskell obteve os melhores resultados com relação ao desempenho. Em relação às linguagens utilizadas, esperava-se esse resultado tanto pelas ferramentas utilizadas, uma vez que a instalação da monada QIO foi mais leve em questão de consumo de RAM, quanto pelo caráter funcional de Haskell, que comporta mais intuitivamente as funções quânticas. Ainda assim, o resultado foi surpreendente devido à grande diferença do tempo de execução entre as duas ferramentas. Por exemplo, quando fatorado o número 21, o tempo em Haskell foi de apenas 4 segundos,

enquanto o mesmo número em Python, levou 11 segundos. Sendo assim, Haskell conseguiu realizar a fatoração, aproximadamente 63,64% mais rápido do que em Python.

Outro exemplo foi o número 25, que em Haskell obteve o número fatorado em 38 segundos e em Python levou 42 segundos. Com o número 25, Haskell conseguiu ser 9,5% mais rápido que Python. Outros resultados podem ser revistos no gráfico 19, onde se apresentou a comparação de desempenho entre Haskell e Python. Além disso, há a tabela 1, que foi dedicada somente para o IBM Qiskit e seus resultados. Na tabela 2 há somente resultados do framework em Haskell. Entretanto, vale ressaltar que ele ficou limitado a 5 qbits, permitindo a fatoração de números entre 1 e 32. Diferentemente de Python, que foi possível utilizar até 7 qbits, permitindo a fatoração de números entre 1 e 100.

Além da limitação de ferramenta como no caso da mônada QIO, outro obstáculo foi a limitação de hardware. Com os recursos disponíveis foi possível avaliar um conjunto de dados suficientes para analisar e realizar o comparativo entre essas duas linguagens.

Na avaliação do impacto do algoritmo quântico de Shor para a Segurança da Informação no que diz respeito à criptografia, foi escolhido o método criptográfico RSA, por seu uso em larga escala em sistemas de informação e por ser baseado no problema da fatoração de grandes números. O RSA resolveu o problema em tempo exponencial, tornando o algoritmo de Shor uma ameaça que se concretizará com a popularização e comercialização dos computadores quânticos. Para demonstrar a capacidade do algoritmo de fatoração quebrar o método RSA foi usado a versão implementada em Python e Qiskit, uma vez que essa biblioteca suporta maiores volumes de dados.

REFERÊNCIAS

BEAZLEY, D.; JONES, B. K. ***Python Cookbook*** .3 ed. Sebastopol. Califórnia, 2013. 706 p. *ISBN 978-1-4493-4037-7.*

BENNETT, C.H.; BERNSTEIN, E.; BRASSARD, G.; VAZIRANI, U. Strengths and weaknesses of quantum computing. **SIAM J. Comput**., 26(5), p. 1510–1523, 1997.

FEITOSA, S. S. **Uma linguagem de programação quântica orientada a objetos baseada no Featherweight Java**. Dissertação (mestrado) - Universidade Federal de Santa Maria - RS. Programa de Pós-Graduação em Informática (PPGI).2016.

GREEN, A. S. **Shor In Haskell - The quantum IO Monad**. The Univ. Nottingham, UK, 2008. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.447.6097&rep=rep1&type=pdf. Acesso em: 05 mai. 2022.

GRILO, A. B**. Computação Quântica e Teoria da Computação.** Dissertação (mestrado) - UNICAMP - Instituto de Computação – Campinas, 2014.

HUDAK, P.; PETERSON, J.; FASEL, J.H. **A Gentle Introduction to Haskell – Version 98**. Los Alamos National Laboratory, 2000. https://www.haskell.org/tutorial/. Acesso em: 10 abr. 2022.

LOMONACO, S. J. *Shor's quantum factoring algorithm*. **Proceedings of Symposia in Applied Mathematics**, vol. 58, p. 161–180, 2002.

MALAQUIAS, J. R. **Programação Funcional em Haskell**. UFOP, Departamento de Computação, 2017. Disponível em: http://www.decom.ufop.br/romildo/2018-1/bcc222/slides/progfunc.pdf. Acesso em: 03 mar. 2022.

MARTINS, R. C. **O Algoritmo de Fatoração de Shor**. 2018. Dissertação de mestrado. Pontifica Universidade Catolica do Rio de Janeiro. Disponível em: https://www.maxwell.vrac.puc-rio.br/35511/35511.PDF. Acesso em: 12 mai. 2022.

MIANO M. G. V. Aplicação De Protocolos Quânticos E Algoritmo De Shor Para a Segurança Da Informação. **Revista Tecnológica da Fatec Americana**, vol. 08, n. 01, p.54-65, 2020.

MIANO, M. G. V.; OLIVEIRA, A. S. Desempenho de algoritmos quânticos e clássicos em treinamento de machine learning supervisionado. **Revista Tecnológica da Fatec Americana**, vol. 09, n. 02, p.81-99, 2021.

QISKIT DOCUMENTATION. **Qiskit 0.20.1 documentation**. 2020. Disponível em: https://qiskit.org/documentation/. Acesso em: 09 abr. 2022.

QISKIT TEXTBOOK. **Learn Quantum Computation using Qiskit**. Disponível em: https://qiskit.org/textbook/preface.html. Acesso em: 09 abr. 2022.

RISTÈ, D.; SILVA, M.; RYAN, C. Demonstration of quantum advantage in machine learning. ***Quantum Inf*** vol 3, n. 16, 2017. https://doi.org/10.1038/s41534-017-0017-3

STALLINGS, W. **Criptografia e Segurança de Redes: Princípios e Práticas**; 6 ed., Pearson. São Paulo, 2014. 560p.