Computação Quântica - Lista de Exercícios

Unidade 2

Aluno: Gustavo Pereira de Carvalho
Docente: Anderson Paiva Cruz

Notebook Python completo:
https://github.com/Gustavo2h/Quantica_Lista_Unidade2/blob/main/CompQuantica.ipynb

1 - A partir da plataforma de computação quântica de sua preferência, implemente os seguintes algoritmos quânticos. Ao final de cada algoritmo, faça um relatório apresentando o código e explicando o funcionamento do mesmo.

a) Faça um circuito quântico cujo estado final é $|00000\rangle + |11111\rangle$.

A questão pede a criação de um estado $|00000\rangle + |11111\rangle$, que seria um estado em que 5 qubits estão emaranhados, ou todos são 0 ou todos são 1.

Para criar esse estado primeiro é necessário começar com todos os qubits em 0, aplicando uma porta Hadamard ao primeiro qubit, o colocando no estado de superposição.

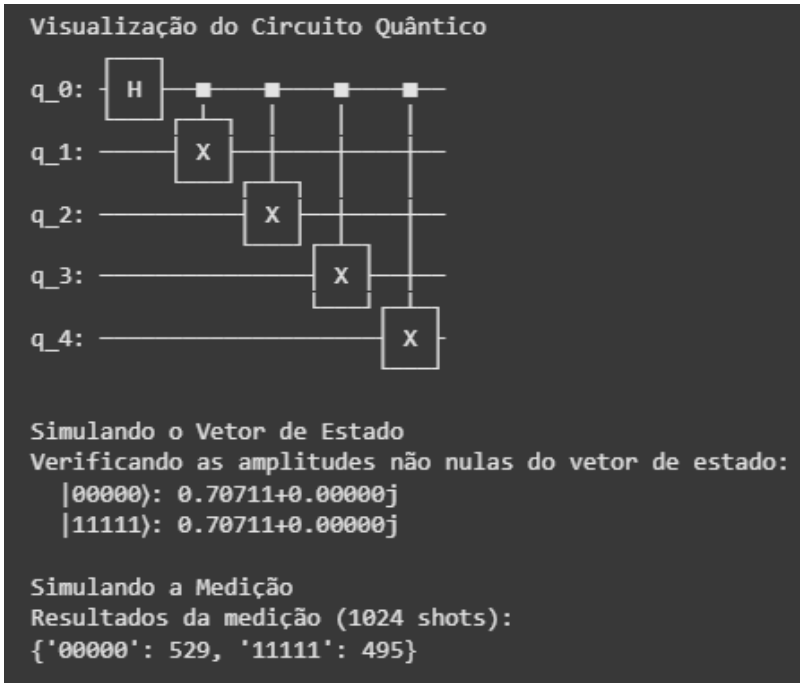
Depois, basta usar o qubit que está em superposição como controle para aplicar portas CNOT em todos os outros qubits, gerando o emaranhamento e a saída esperada.

```
# Criando 5 qubits
n_qubits = 5
qc = QuantumCircuit(n_qubits)

# Aplicando Hadamard no primeiro qubit
qc.h(0)

# Depois CNOTs entre o primeiro qubit e todos os outros
for target_qubit in range(1, n_qubits):
    qc.cx(0, target_qubit)
```

O resultado é o esperado e pedido na questão:



b) Implemente a subrotina de teleporte de informação quântica.

O teleporte quântico consiste em transportar um estado quântico, informação nesse caso, de um local para outro. Geralmente o local fonte é chamado de Alice e o destino de Bob.

Para implementação do teleporte são necessários, 1 qubit de Alice que terá o estado que vai ser teleportado, outro qubit de Alice e 1 de Bob que estão emaranhados e 2 bits clássicos que Alice envia para Bob.

Primeiro um theta qualquer que será enviado é aplicado no qubit 0 de Alice. Depois o qubit 1 de Alice e qubit de Bob são emaranhados, depois é aplicado um conjunto de CNOT e Hadamard nos qubits de Alice para realizar a medição. Por fim, os dois bits clássicos de Alice são enviados para Bob e Bob aplica as portas X e/ou Z dependendo dos bits clássicos recebidos.

```python
# Estado que vai ser teleportado
theta = np.pi / 3  # Um ângulo qualquer

# Declaração dos registradores e construção do circuito
q = QuantumRegister(3, 'q')
c_alice = ClassicalRegister(2, 'c_alice')
c_bob = ClassicalRegister(1, 'c_bob')

qc = QuantumCircuit(q, c_alice, c_bob)

# Aplicando o estado theta no primeiro qubit de Alice
qc.ry(theta, q[0])
qc.barrier()

# Criação do par entrelaçado entre Alice e Bob
qc.h(q[1])
qc.cx(q[1], q[2])
qc.barrier()

# Medição de Bell (Alice)
qc.cx(q[0], q[1])
qc.h(q[0])
qc.barrier()

# Comunicação clássica (Alice -> Bob)
qc.measure(q[0], c_alice[0]) # q0 -> c_alice[0]
qc.measure(q[1], c_alice[1]) # q1 -> c_alice[1]
qc.barrier()

# Correção (Bob)

with qc.if_test((c_alice[1], 1)): # Se c_alice[1] (de q1) == 1
    qc.x(q[2]) # Aplica X em q2

with qc.if_test((c_alice[0], 1)): # Se c_alice[0] (de q0) == 1
    qc.z(q[2]) # Aplica Z em q2

qc.barrier()
```

Para verificar, o -theta é aplicado ao qubit de Bob. Se ele estava em theta, voltará para 0, dessa forma é possível medir seu estado e verificar se teleporte deu certo.

```python
# Verificação de Bob
"""
Aplica -theta para que Bob volte pra 0. Se Bob realmente recebeu theta,
todas as saidas de Bob serão 0, pra qualquer valor de q0 e q1 de Alice.
"""
qc.ry(-theta, q[2])
qc.measure(q[2], c_bob[0]) # q2 -> c_bob[0]
```

O resultado mostra que o teleporte funcionou. O qubit de Bob é 0 para qualquer qubit de Alice e qualquer theta:

```
Resultados da simulação (1024 shots)
{'0 01': 264, '0 00': 250, '0 10': 261, '0 11': 249}
```

c) Implemente a sub rotina de soma completa de dois qubits.

O somador completo soma 3 bits, A, B e C_in, dando como saída a soma e um C_out. Nesse caso, o código usa C_in = 0 para somar apenas A e B, sendo assim, temos 3 qubits mais um qubit ancilla para o C_out. A soma (A ⊕ B ⊕ C_in) será armazenada no qubit q2 e o C_out ((A·B) + (C_in·(A⊕B))) no q3.

O circuito foi feito utilizando portas CNOT e CCX:

CCX(q0, q1, q3): Calcula A·B e armazena em q3
q3 agora é (A·B).

CNOT(q0, q1): Calcula A⊕B e armazena em q1
q1 agora é (A⊕B).

CCX(q1, q2, q3): Calcula (A⊕B)·C_in e faz um XOR com q3
q3 agora é (A·B) ⊕ ((A⊕B)·C_in), a fórmula booleana para o C_out

CNOT(q1, q2): Calcula (A⊕B) ⊕ C_in e armazena em q2
q2 agora é (A⊕B) ⊕ C_in, a fórmula para a soma

```python
def create_full_adder_circuit():
    # São 4 qubits no total
    qc = QuantumCircuit(4, name="Full Adder")

    # C_out = (A AND B)
    qc.ccx(0, 1, 3)

    # q[1] = A XOR B
    qc.cx(0, 1)

    # C_out = C_out XOR ( (A XOR B) AND C_in )
    qc.ccx(1, 2, 3)

    # Soma = (A XOR B) XOR C_in
    qc.cx(1, 2)

    return qc
```

No fim q2 contém a soma e q3 o C_out. O resultado também é o esperado, todos os resultados batem com a versão clássica.

```
Entrada (A,B,C_in): (0,0,0)
Saída Quântica (C_out, Soma): (0, 0)
Saída Clássica:               (0, 0)

Entrada (A,B,C_in): (0,0,1)
Saída Quântica (C_out, Soma): (0, 1)
Saída Clássica:               (0, 1)

Entrada (A,B,C_in): (0,1,0)
Saída Quântica (C_out, Soma): (0, 1)
Saída Clássica:               (0, 1)

Entrada (A,B,C_in): (0,1,1)
Saída Quântica (C_out, Soma): (1, 0)
Saída Clássica:               (1, 0)

Entrada (A,B,C_in): (1,0,0)
Saída Quântica (C_out, Soma): (0, 1)
Saída Clássica:               (0, 1)

Entrada (A,B,C_in): (1,0,1)
Saída Quântica (C_out, Soma): (1, 0)
Saída Clássica:               (1, 0)

Entrada (A,B,C_in): (1,1,0)
Saída Quântica (C_out, Soma): (1, 0)
Saída Clássica:               (1, 0)

Entrada (A,B,C_in): (1,1,1)
Saída Quântica (C_out, Soma): (1, 1)
Saída Clássica:               (1, 1)
```

d) Faça o algoritmo de Deutsch-Jozsa com 3 qubits de entrada e $f(x) = x_0 \oplus x_1 x_2$.

O algoritmo de Deutsch-Jozsa é usado para determinar se uma função é constante (retorna 0 ou 1 para todas as entradas) ou balanceada (retorna 0 para exatamente metade das entradas e 1 para a outra metade). Nesse caso temos 3 qubits de entrada e precisamos de 1 qubit ancilla.

Verificando manualmente a função f(x)=x0⊕x1x2, vemos que ela é balanceada. Logo, os resultados da medição devem ser diferentes de 000.

O primeiro passo no circuito é aplicar Hadamard em todos os qubits de entrada, dessa forma o oráculo pode calcular f(x) para todos os qubits de entrada de uma vez. Após isso é aplicado uma porta X seguida de Hadamard no oráculo, o deixando no estado |-⟩. Quando o qubit ancilla está no estado |-⟩, um CNOT (ou qualquer porta controlada) não "flipa" o alvo. Em vez disso, ele aplica uma fase de -1 ao qubit de controle se o controle for |1⟩. Então f(x) é aplicado.

```
# 3 qubits + 1 qubit ancilla
# O ancilla terá indice n
qc = QuantumCircuit(n + 1, n)

# Coloca a ancilla no estado |1>
qc.x(n)

# Aplica Hadamard em todo os qubits
qc.h(range(n + 1))
qc.barrier()

# f(x) = x0 ⊕ (x1 * x2)

qc.cx(0, n) # CNOT(q0, ancilla)

# Porta Toffoli: (q1 é controle, q2 é controle, n é alvo)
qc.ccx(1, 2, n)

qc.barrier()

# Aplica Hadamard nos qubits de entrada (de 0 a n-1)
qc.h(range(n))
qc.barrier()

# Mede os qubits de entrada
qc.measure(range(n), range(n))
```

Por fim é aplicada uma segunda camada de portas Hadamard em todos os qubits de entrada fazendo com que os 8 caminhos quânticos interfiram uns nos outros. Se a função fosse Constante, todas as fases seriam iguais e interferência seria 100% construtiva no estado |000⟩. Se a função for Balanceada, as fases positivas e negativas se cancelam perfeitamente (interferência destrutiva) no estado |000⟩, resultando em 0% de probabilidade de medi-lo.

No resultados podemos ver que nenhuma medição registrou |000⟩.

```
Resultados da medição
{'001': 238, '101': 244, '111': 274, '011': 268}
```

e) Um algoritmo que resolve o problema de Bernstein-Vazirani.

O algoritmo de Bernstein-Vazirani é bem parecido com o de Deutsch-Jozsa, porém ele resolve o problema de uma string secreta. No caso, o oráculo esconde uma string de n bits e uma entrada x é aceita pelo circuito, calculando o produto escalar (mod 2) para entre x e a string secreta para descobri-lá. A vantagem do algoritmo é que no caso clássico seriam necessárias n consultas, enquanto no caso quântico a string inteira é encontrada com 1 consulta.

Assim como no Deutsch-Jozsa o circuito inicia aplicando Hadamard em todos os qubits de entrada e o ancilla é colocado no estado $|-\rangle$. Após isso, um loop for aplica CNOT apenas se o qubit de entrada for 1, construindo o oráculo para a string secreta. Por fim, mais uma camada de Hadamard é aplicada no qubits de entrada.

```
# Definindo a string secreta
secret_string = '10110'
n = len(secret_string)

# n qubits de entrada + 1 qubit ancilla
qc = QuantumCircuit(n + 1, n)

# Coloca a ancilla no estado |1>
qc.x(n)

# Aplica Hadamard em todos os qubits (incluindo o ancilla)
qc.h(range(n + 1))
qc.barrier()

"""
f(x) = s . x
CNOT(q_i, ancilla) se s_i == 1
Qiskit ordena os bits em little endian
É necessário reverter a string para mapear s_0 -> q_0, s_1 -> q_1...
"""
s_reversed = secret_string[::-1] # Torna-se '01101'

for i, bit in enumerate(s_reversed):
    if bit == '1':
        print(f"Aplicando CNOT de q_{i} (s_{i}) para a ancilla.")
        qc.cx(i, n) # i é o qubit de controle, n é a ancilla

qc.barrier()

# Aplica Hadamard nos qubits de ENTRADA (0 a n-1)
qc.h(range(n))
qc.barrier()

# Mede os qubits de entrada
qc.measure(range(n), range(n))
```

Outro ponto importante é que o algoritmo de Bernstein-Vazirani é determinístico, então apenas 1 shot seria suficiente para obter o resultado.

No resultado é possível ver que o algoritmo funcionou e encontrou a string secreta:

```
Resultado da Medição (1024 shot)
{'10110': 1024}

String medida: 10110
A string secreta foi encontrada.
```

f) O algoritmo de Grover para 2 qubits e o valor em $|\beta\rangle = 01$.

O algoritmo de Grover é um algoritmo de busca quântica, o objetivo nesse caso é encontrar um item marcado dentro de um "banco de dados". Como são apenas 2 qubits, temos 4 itens possiveis e o item marcado é |β⟩=01. A vantagem de usar Grover é sua complexidade O(√N), que permite encontrar o item mais rápido que uma busca clássica.

Algumas funções auxiliares são usada no código, a primeira é uma função para criar o oráculo, marcando o estado alvo aplicando uma inversão de fase:

qc.x(q[1]): "Flipa" o estado (ex: |01⟩ para |11⟩).

qc.cz(q[0], q[1]): A porta Controlada-Z aplica a fase de -1 apenas ao estado |11⟩.

qc.x(q[1]): "Desflipa" o estado de volta (-|11⟩ para -|01⟩).

```python
def create_oracle(qc, qubits_to_mark):
    #Aplica o oráculo para marcar o estado |01>
    qc.x(qubits_to_mark[1])          # inverter q1
    qc.cz(qubits_to_mark[0], qubits_to_mark[1])
    qc.x(qubits_to_mark[1])          # desfazer inversão
```

Dessa forma o estado |01⟩ torna-se -|01⟩, e todos os outros estados (|00⟩, |10⟩, |11⟩) permanecem inalterados. Marcando o |01⟩.

A segunda função auxiliar é o difusor, que tem a função de amplificar a amplitude do item marcado, que está negativo e diminuir a amplitude dos outros. Isso é feito com duas camadas de Hadamard.

```python
def create_diffuser(qc, qubits):
    #Aplica o difusor de Grover (inversão sobre a média)
    qc.h(qubits)
    qc.x(qubits)
    qc.cz(qubits[0], qubits[1])
    qc.x(qubits)
    qc.h(qubits)
```

Por fim, a execução principal consiste na simulação do vetor de estados, colocando os qubits em superposição e aplicando as funções auxiliares, para um resultado mais teórico.

```
# Criando os 2 qubits
n = 2
qubits = list(range(n))

# Simulação (Vetor de Estado)
print("Simulação do Vetor de Estado")

qc_sv = QuantumCircuit(n)
qc_sv.h(qubits)
qc_sv.barrier()
create_oracle(qc_sv, qubits)
qc_sv.barrier()
create_diffuser(qc_sv, qubits)
qc_sv.barrier()

# Usando Statevector diretamente (AerSimulator gera um erro)
statevector = Statevector.from_instruction(qc_sv)
```

Após isso é simulada a medição para se aproximar do que um computador quântico faria, com a mesma sequência, aplicando Hadamard e depois as funções auxiliares, temos o resultado das 1024 execuções. Que corresponde ao alvo esperado, que teve sua amplitude amplificada.

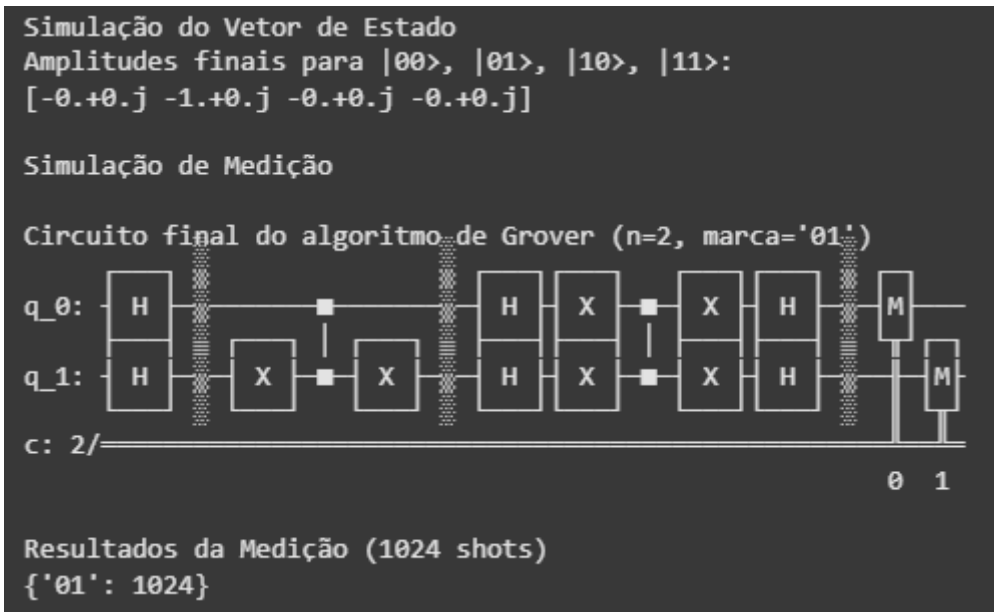
```
# Medição
print("\nSimulação de Medição")

qc_measure = QuantumCircuit(n, n)
qc_measure.h(qubits)
qc_measure.barrier()
create_oracle(qc_measure, qubits)
qc_measure.barrier()
create_diffuser(qc_measure, qubits)
qc_measure.barrier()
qc_measure.measure(qubits, qubits)

print("\nCircuito final do algoritmo de Grover (n=2, marca='01')")
print(qc_measure.draw(output='text'))

qasm_sim = AerSimulator()
transpiled_qc_m = transpile(qc_measure, qasm_sim)
job_m = qasm_sim.run(transpiled_qc_m, shots=1024)
result_m = job_m.result()
counts = result_m.get_counts()
```

O resultado da medição corresponde ao mesmo da simulação do statevector e mostra o item alvo $|\beta\rangle = 01$:



2 - Implemente o algoritmo de Shor para N=15. O desafio aqui é desenvolver um modelo híbrido (clássica-quântico) que ao final consiga retornar p=5 e q=3.

O algoritmo híbrido de Shor serve para fatorar um número, nesse caso N = 15, em seus fatores primos, p = 5 e q = 3. O código conta com duas partes, a clássica, que faz o pré e pós processamento, e a quântica, que faz a parte mais difícil, encontrar o período de uma função modular.

O código funciona com um loop while que tenta encontrar um fator até ter sucesso. A primeira etapa é o pré-processamento clássico, que verifica se há um MDC (Mínimo Divisor Comum), se 'a' não for co-primo de N (MDC = 1), significa que já encontramos os fatores.

```python
# Pré-processamento Clássico
def step_1_classical_pre_check(N):
    print("[CLÁSSICO] Pré-processamento")

    # Escolhe 'a' aleatoriamente
    a = random.randint(2, N - 1)
    print(f"N = {N}. a = {a}")

    # Verifica se 'a' compartilha um fator com 'N'
    gcd = math.gcd(a, N)
    if gcd != 1:
        print(f"gcd(a, N) = {gcd}. Fator encontrado")
        p = gcd
        q = N // gcd
        return (p, q), None

    print(f"gcd({a}, {N}) = 1. 'a' é co-primo")
    return None, a
```

Caso isso não aconteça, passamos para a parte quântica, que constrói um circuito quântico para encontrar o período 'r' da função f(x) = a^x mod(15). A função apply_C_a_x_mod15() é usada para aplicar a exponenciação modular para o caso específico de N=15, essa parte é uma implementação "hardcoded" para N=15.

```
# Sub-rotina quântica
def step_2_quantum_period_finding(N, a):
    print("\n[QUÂNTICO] Busca de período")

    n_count = 8  # Qubits de contagem
    m_work = 4   # Qubits de trabalho (para N=15)

    print(f"Construindo circuito para a = {a}")

    qr_count = QuantumRegister(n_count, name="count")
    qr_work = QuantumRegister(m_work, name="work")
    cr_count = ClassicalRegister(n_count, name="c_count")
    qc = QuantumCircuit(qr_count, qr_work, cr_count)

    # Superposição e |1>
    qc.h(qr_count)
    qc.x(qr_work[0])
    qc.barrier()

    # Exponenciação modular controlada
    print("Aplicando o oráculo de exponenciação modular")
    target_qs = [qr_work[i] for i in range(m_work)]

    for j in range(n_count):
        # Calcula a^(2^j) mod N
        a_pow = pow(a, 2**j, N)

        # Pega o qubit de controle
        control_q = qr_count[j]

        # Aplica a porta C( (a^pow) * x mod 15 )
        apply_C_a_x_mod15(qc, a_pow, control_q, target_qs)

    qc.barrier()
```

Após isso é utilizada a Transformada de Fourier Quântica Inversa (iqft_gate) para extrair esse período ‘r’ de uma superposição, sendo necessário apenas 1 shot. Esse passo executa a parte quântica completa do código, incluindo a simulação.

```
# Transformada de Fourier quântica inversa
print("Aplicando a QFT Inversa")
qft_gate = QFTGate(n_count)
iqft_gate = qft_gate.inverse()
qc.append(iqft_gate, qr_count)

qc.barrier()

# 4. Medição
qc.measure(qr_count, cr_count)

# Simulação
print("Simulando o circuito")
simulator = AerSimulator()
qc_transpiled = transpile(qc, simulator)
job = simulator.run(qc_transpiled, shots=1) # 1 shot é suficiente
result = job.result()
counts = result.get_counts()

measured_str = list(counts.keys())[0]
measured_k = int(measured_str, 2)

print(f"Resultado: k = {measured_k} (de {2**n_count})")
return measured_k, n_count
```

A última parte também é clássica e recebe a medição K feita no passo 2 (parte quântica), o k é usado para encontrar o período 'r' usando o algoritmo de frações contínuas. Se 'r' for "bom" (par e não trivial), ele o usa para calcular os fatores:

p = MDC(a^(r/2)-1, N) e q = MDC(a^(r/2)+1, N)

```python
# Pós-processamento clássico
def step_3_classical_post_process(k, n_count, N, a):
    print("\n[CLÁSSICO] Pós-processamento")

    if k == 0:
        print(f"Medição foi k=0. Falha")
        return None

    Q = 2**n_count

    # Algoritmo de frações contínuas
    print(f"Executando frações contínuas em k/Q = {k}/{Q}")
    fraction = Fraction(k, Q).limit_denominator(N)
    r = fraction.denominator

    print(f"Fração encontrada: {fraction.numerator}/{fraction.denominator}. \
     Candidato a período r = {r}")

    # Verificações do Período
    if r % 2 != 0:
        print(f"Falha: Período r={r} é ímpar. 'a'={a} falhou")
        return None

    a_r_half = pow(a, r // 2, N)

    if (a_r_half + 1) % N == 0:
        print(f"Falha: Período r={r} é trivial")
        print(f"'a'={a})")
        return None

    print(f"Período r={r} funciona")

    # Encontrar os Fatores
    p = math.gcd(a_r_half - 1, N)
    q = math.gcd(a_r_half + 1, N)

    if p == 1 or q == 1:
        print(f"Falha: Um dos fatores é 1")
        return None

    return (p, q)
```

Se 'r' for "ruim" (ímpar, ou a=14 que dá r=2 trivial), a função retorna None, e o while loop na função principal tenta tudo de novo com um novo ‘a’ aleatório.

```python
# Função principal
def run_shor_hybrid_random(N=15):
    print(f"Algoritmo de Shor para N={N}\n")

    factors_found = None

    #Loop de tentativas até um 'a' funcionar
    while factors_found is None:

        # Pré-verificação
        factors_lucky, a = step_1_classical_pre_check(N)
        if factors_lucky:
            factors_found = factors_lucky
            break

        # Busca de Período
        # Se 'a' for 14, o passo 3 irá falhar
        k, n_count = step_2_quantum_period_finding(N, a)

        # Pós-processamento
        factors_found = step_3_classical_post_process(k, n_count, N, a)

        if factors_found is None:
            print("\nTentativa falhou. Tentando um novo 'a' aleatório\n")
            print("============================================================\n")

    # Fim do loop
    p, q = factors_found
    print("\n============================================================")
    print(f"O chute final 'a' foi {a}.")
    print(f"Os fatores de {N} são p = {p} e q = {q}.")
    print("============================================================")
    return (p, q)

#Executar
if __name__ == "__main__":
    run_shor_hybrid_random(N=15)
```

Como ‘a’ é aleatório, alguns resultados são mais diretos que outros. Um resultado interessante e que usa todos os passos é esse:

```
Algoritmo de Shor para N=15

[CLÁSSICO] Pré-processamento
N = 15. a = 11
gcd(11, 15) = 1. 'a' é co-primo

[QUÂNTICO] Busca de período
Construindo circuito para a = 11
Aplicando o oráculo de exponenciação modular
Aplicando a QFT Inversa
Simulando o circuito
Resultado: k = 0 (de 256)

[CLÁSSICO] Pós-processamento
Medição foi k=0. Falha

Tentativa falhou. Tentando um novo 'a' aleatório

============================================================

[CLÁSSICO] Pré-processamento
N = 15. a = 4
gcd(4, 15) = 1. 'a' é co-primo

[QUÂNTICO] Busca de período
Construindo circuito para a = 4
Aplicando o oráculo de exponenciação modular
Aplicando a QFT Inversa
Simulando o circuito
Resultado: k = 0 (de 256)

[CLÁSSICO] Pós-processamento
Medição foi k=0. Falha

Tentativa falhou. Tentando um novo 'a' aleatório

============================================================
```

```
[CLÁSSICO] Pré-processamento
N = 15. a = 4
gcd(4, 15) = 1. 'a' é co-primo

[QUÂNTICO] Busca de período
Construindo circuito para a = 4
Aplicando o oráculo de exponenciação modular
Aplicando a QFT Inversa
Simulando o circuito
Resultado: k = 0 (de 256)

[CLÁSSICO] Pós-processamento
Medição foi k=0. Falha

Tentativa falhou. Tentando um novo 'a' aleatório

============================================================

[CLÁSSICO] Pré-processamento
N = 15. a = 4
gcd(4, 15) = 1. 'a' é co-primo

[QUÂNTICO] Busca de período
Construindo circuito para a = 4
Aplicando o oráculo de exponenciação modular
Aplicando a QFT Inversa
Simulando o circuito
Resultado: k = 0 (de 256)

[CLÁSSICO] Pós-processamento
Medição foi k=0. Falha

Tentativa falhou. Tentando um novo 'a' aleatório

============================================================

[CLÁSSICO] Pré-processamento
N = 15. a = 10
gcd(a, N) = 5. Fator encontrado

============================================================
O chute final 'a' foi None.
Os fatores de 15 são p = 5 e q = 3.
============================================================
```