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/CompQuanticaUni2.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.

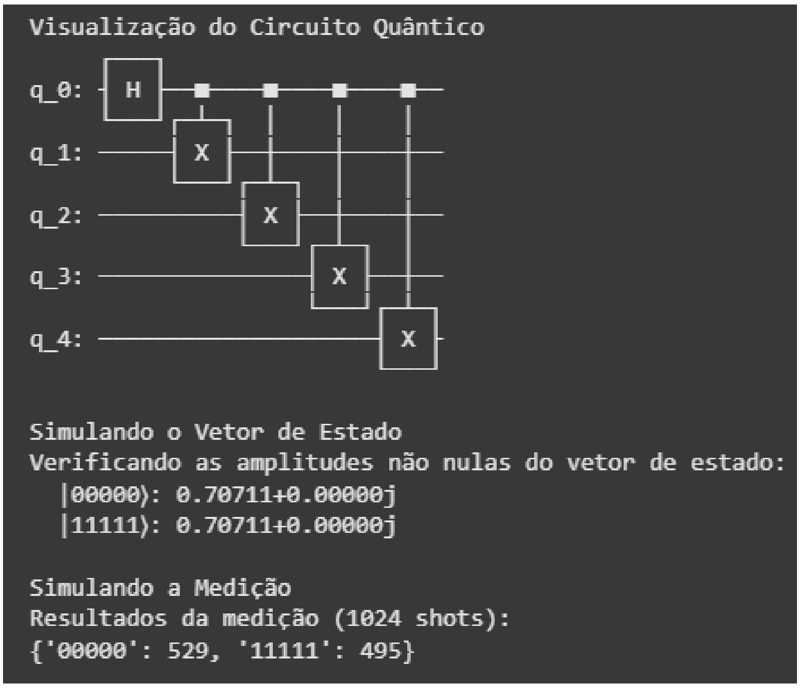
Depois, basta usar o qubit que está em superposição como controle para aplicar portas CNOT em todos os outros qubits, gerando o emaranhamento e a saída esperada.

```
# 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:



Cálculo teórico:

1-a)

$|\psi_0\rangle = |0\rangle^{\otimes 5} = |00000\rangle$; $H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$

$|\psi_1\rangle = (H|0\rangle) \otimes |0000\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0000\rangle$

0 é o qubit em super posição. Rest são os outros 4 qubits.

$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle_0 \otimes |0000\rangle_{Rest} + |1\rangle_0 \otimes |1111\rangle_{Rest}) = \frac{1}{\sqrt{2}}(|00000\rangle + |00001\rangle)$

O qiskit usa little-endian, então 1 está na posição menos significativa.

$q_0 = 0$; $|00000\rangle \to |00000\rangle$ $\quad |\psi_{final}\rangle = \frac{1}{\sqrt{2}}(|00000\rangle + |00001\rangle)$

$q_0 = 1$; $|00001\rangle \to |11111\rangle$

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.

```
# 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}
```

Cálculo teórico:

b) Sendo $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ com

$\alpha = \cos(\frac{\theta}{2})$; $\beta = \sin(\frac{\theta}{2})$; $|\Phi^+\rangle_{12} = \frac{1}{\sqrt{2}}(|00\rangle_{12} + |11\rangle_{12})$

O estado inicial:

$$|\Psi_0\rangle = |\psi\rangle_0 \otimes |\Phi^+\rangle_{12} = (\alpha|0\rangle_0 + \beta|1\rangle_0) \otimes \frac{1}{\sqrt{2}}(|00\rangle_{12} + |11\rangle_{12})$$

Temos CNot de q_0 para q_1 e H em q_0:

$$|\Psi\rangle_{após} = \frac{1}{2}\Big(|00\rangle_{01}(\alpha|0\rangle_2 + \beta|1\rangle_2) + |01\rangle_{01}(\alpha|1\rangle_2 + \beta|0\rangle_2) + |10\rangle_{01}(\alpha|0\rangle_2 - \beta|1\rangle_2) + |11\rangle_{01}(\alpha|1\rangle_2 - \beta|0\rangle_2)\Big)$$

A ordem dos qubits é $q_0 q_1 q_2$

Usando $\begin{cases} X|0\rangle = |1\rangle ; X|1\rangle = |0\rangle \\ Z|0\rangle = |0\rangle ; Z|1\rangle = -|1\rangle \end{cases}$, temos

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

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

$\alpha|0\rangle - \beta|1\rangle = Z|\psi\rangle$

$\alpha|1\rangle - \beta|0\rangle = XZ|\psi\rangle$

A expressão composta do resultado seria:

$$|\Psi\rangle_{após} = \frac{1}{2}\sum_{m,n\in\{0,1\}} |mn\rangle_{01}(X^n Z^m|\psi\rangle_2)$$

m é a medição de q_0

n é a medição de q_1

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)
```

Cálculo teórico:

c)

Soma $S = A \oplus B \oplus C_{in}$. $\quad$ $\oplus$ é o XOR

Carry out $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$ $\quad$ $+$ é Or

$\cdot$ é AND

$CCX(q_0, q_1, q_3) \rightarrow q_3 = A \wedge B$

$CNOT(q_0, q_1) \rightarrow q_1 = A \oplus B$

$CCX(q_1, q_2, q_3) \rightarrow q_3 = (A \wedge B) \oplus ((A \oplus B) \wedge C_{in})$

Como $x \oplus y = x + y \pmod 2$:

$q_3 = C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$

$CNOT(q_1, q_2) \rightarrow q_2 = q_1 \oplus q_2 = (A \oplus B) \oplus C_{in}$

$q_2 = S = \underline{\underline{A \oplus B \oplus C_{in}}}$.

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}
```

Cálculo teórico:

d)

$f(x) = x_0 \oplus (x_1 x_2)$

Começando com n qubits de entrada em $|0\rangle^{\otimes n}$ e um ancilla em $|1\rangle$.

ancilla $|1\rangle \to |-\rangle = \frac{1}{\sqrt{2}} |0\rangle - |1\rangle$ (aplicando X e depois H)

O oráculo U_f age da seguinte forma:

$U_f(|x\rangle \otimes |-\rangle) = (-1)^{f(x)} |x\rangle \otimes |-\rangle$.

No caso $n=3$; $f(x) = x_0 \oplus (x_1 x_2)$ e $x = (x_0, x_1, x_2)$

$S := \sum_{x \in \{0,1\}^3} (-1)^{f(x)} = \sum_{x_0 x_1 x_2} (-1)^{x_0 \oplus (x_1 x_2)}$, $(-1)^{a \oplus b} = (-1)^a (-1)^b$

logo: $(-1)^{x_0 \oplus x_1 x_2} = (-1)^{x_0} (-1)^{x_1 x_2}$

$S = \sum_{x_1 x_2} (-1)^{x_1 x_2} \sum_{x_0} (-1)^{x_0}$, Mas $\sum_{x_0} (-1)^0 = (-1)^0 + (-1)^1 = 1 - 1 = 0$

$S = 0$

Após a segunda camada $H \to \langle 000 | \psi_{final} \rangle = \frac{1}{2^3} S = \frac{1}{8} \cdot 0 = 0$

A probabilidade de medir $|000\rangle$ é 0, então o algoritmo identifica que f é balanceada.

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 |-⟩. 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.

```python
# 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.
```

Cálculo teórico:

e) S = string secreta

$$f_s(x) = S \cdot x \pmod 2 = \sum_{i=0}^{n-1} s_i x_i \pmod 2$$

Começa com n qubits em $|0\rangle^{\otimes n}$ e ancilla $|1\rangle$

Aplicando H nos qubits de entrada e X e H na ancilla temos:

$$|\Psi_1\rangle = \left(\frac{1}{\sqrt{2^n}} \sum_x |x\rangle\right) \otimes |-\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

O oráculo U_f faz: $U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f_s(x)\rangle$, quando ancilla é $|-\rangle$ ele implementa uma fase:

$$U_f(|x\rangle \otimes |-\rangle) = (-1)^{f_s(x)} |x\rangle \otimes |-\rangle$$

Daí:

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_x (-1)^{f_s(x)} |x\rangle \otimes |-\rangle$$ O ancilla será ignorado porque fatoriza e não altera as medições

Aplicando $H^{\otimes n}$ novamente:

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_y (-1)^{x \cdot y} |y\rangle \; ; \quad |\Phi\rangle = \frac{1}{2^n} \sum_y \left(\sum_x (-1)^{x \cdot (y \oplus s)} \right) |y\rangle$$

~~pois (...)~~ Usando a identidade (para $z \in \{0,1\}^n$)

$$\sum_x (-1)^{x \cdot z} = \begin{cases} 2^n, & \text{se } z = 0; \\ 0, & \text{se } z \neq 0; \end{cases}$$ Aplicando como $z = y \oplus s$, o coeficiente de entrada $|y\rangle$ é:

$$\frac{1}{2^n} \sum_x (-1)^{x(y \oplus s)} = \begin{cases} 1, & \text{se } y \oplus s = 0 \\ 0, & \text{caso contrário} \end{cases} \rightarrow |\Phi\rangle = |S\rangle$$

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 $|\beta\rangle$=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⟩).

```
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.

```
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.

```python
# 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 $|\beta\rangle = 01$:

```
Simulação do Vetor de Estado
Amplitudes finais para |00>, |01>, |10>, |11>:
[-0.+0.j -1.+0.j -0.+0.j -0.+0.j]

Simulação de Medição

Circuito final do algoritmo de Grover (n=2, marca='01')
      ┌───┐ ░           ░ ┌───┐┌───┐   ┌───┐┌───┐ ░ ┌─┐   
q_0: ┤ H ├─░───────■───────░─┤ H ├┤ X ├─■─┤ X ├┤ H ├─░─┤M├───
      ├───┤ ░ ┌───┐ │ ┌───┐ ░ ├───┤├───┤ │ ├───┤├───┤ ░ └╥┘┌─┐
q_1: ┤ H ├─░─┤ X ├─■─┤ X ├─░─┤ H ├┤ X ├─■─┤ X ├┤ H ├─░──╫─┤M├
      └───┘ ░ └───┘   └───┘ ░ └───┘└───┘   └───┘└───┘ ░  ║ └╥┘
c: 2/═══════════════════════════════════════════════════╩══╩═
                                                         0  1 

Resultados da Medição (1024 shots)
{'01': 1024}
```

Cálculo teórico:

7) O estado inicial é:

$$|S\rangle = \frac{1}{\sqrt{4}} \sum_{x} |x\rangle = \frac{1}{2}\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$ Todas amplitudes iniciais são $\frac{1}{2}$

O oráculo de fase aplica -1 ao estado marcado

$$O: \frac{1}{2}\begin{pmatrix}1\\1\\1\\1\end{pmatrix} \to \frac{1}{2}\begin{pmatrix}1\\-1\\1\\1\end{pmatrix}$$ As amplitudes são: $\left(\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$

Depois temos Difusor: Seja a média das amplitudes após o oráculo $= \frac{1}{4}$

$\bar{a} = \frac{1}{4}$; A ação do difusor é: $a_j \to 2\bar{a} - a_j$

Assim temos
$$\begin{cases} \text{Para } |01\rangle: a_\beta = -\frac{1}{2} \to 2\cdot\frac{1}{4} - \left(-\frac{1}{2}\right) = \frac{1}{2} + \frac{1}{2} = 1 \\ \text{Para o resto: cada } a = \frac{1}{2} \to 2\cdot\frac{1}{4} - \frac{1}{2} = \frac{1}{2} - \frac{1}{2} = 0 \end{cases}$$

O vetor final é: $$\begin{pmatrix}0\\1\\0\\0\end{pmatrix} = |01\rangle$$ Após uma iteração já obtemos $|01\rangle$.

Há derivações alternativas, usando projeções, que seria mais compacta

OBS: Para outros valores de N, o número de iterações é diferente de 1. Sendo $\left(\frac{\pi}{4}\sqrt{N}\right)$. Outros cenários podem precisar de mais iterações do difusor para ajustar as amplitudes.